Technical Report

UCAM-CL-TR-792
ISSN 1476-2986

Number 792

Machine learning and automated theorem proving

James P. Bridge

November 2010

This technical report is based on a dissertation submitted October 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Corpus Christi College.

ISSN 1476-2986

Machine learning and automated theorem proving

James P. Bridge

Summary

Computer programs to find formal proofs of theorems have a history going back nearly half a century. Originally designed as tools for mathematicians, modern applications of automated theorem provers and proof assistants are much more diverse. In particular they are used in formal methods to verify software and hardware designs to prevent costly, or life threatening, errors being introduced into systems from microchips to controllers for medical equipment or space rockets.

Despite this, the high level of human expertise required in their use means that theorem proving tools are not widely used by non specialists, in contrast to computer algebra packages which also deal with the manipulation of symbolic mathematics. The work described in this dissertation addresses one aspect of this problem, that of heuristic selection in automated theorem provers. In theory such theorem provers should be automatic and therefore easy to use; in practice the heuristics used in the proof search are not universally optimal for all problems so human expertise is required to determine heuristic choice and to set parameter values.

Modern machine learning has been applied to the automation of heuristic selection in a first order logic theorem prover. One objective was to find if there are any features of a proof problem that are both easy to measure and provide useful information for determining heuristic choice. Another was to determine and demonstrate a practical approach to making theorem provers truly automatic.

In the experimental work, heuristic selection based on features of the conjecture to be proved and the associated axioms is shown to do better than any single heuristic. Additionally a comparison has been made between static features, measured prior to the proof search process, and dynamic features that measure changes arising in the early stages of proof search. Further work was done on determining which features are important, demonstrating that good results are obtained with only a few features required.

Acknowledgments

I would like to thank my two supervisors, Professor Lawrence Paulson and Dr Sean Holden, for their guidance, feedback and encouragement during my research. Thanks must also go to the author of the theorem prover "E", Dr Stephan Schulz, who has with great patience responded promptly to many e-mails.

Contents

Chapter 1

Motivation

1.1 The thesis

This dissertation concerns computer software designed to find a mathematical proof of an expression stated in a formal language. The expression is a *conjecture* and is associated with other expressions, the *axioms*, which are already known to be, or assumed to be, true. The formal language is a *logic*, of which there are several types and the software is a *theorem prover*. These terms are covered in detail in chapter 2.

The thesis of this dissertation is that the choice of the best proof search heuristic to use in an automated first order logic theorem prover may be related to measurable features of the conjecture and associated axioms and that this relationship may be accurately approximated by a function obtained using machine learning.

The thesis is worth investigating for several reasons. First, automated theorem provers have many applications but their use is restricted by the need for human expertise and a key part of the expertise required is in the selection of the appropriate heuristic for a particular problem. Second, though machine learning has previously been applied to theorem provers, earlier work has concentrated on the learning of new heuristics with limited success. Selection between known good heuristics represents a novel application of machine learning[1]. Third, analysis of the machine learning results, in terms of which measured features are of importance, provides insight into the structure of the proof problem which is interesting in its own right.

The thesis involves logic and machine learning which are both extensive fields and detailed background is given in the next chapter. The present chapter covers motivation in terms of applications of automated theorem provers, reasons for the selection of the particular type of theorem prover selected, the importance of heuristic selection to the theorem proving process and finally some justification for applying machine learning to the problem of heuristic selection.

[1]Automatic heuristic selection is provided in the theorem prover E but this is based on prior experimentation rather than any recognised method of machine learning and is discussed in more detail later in this dissertation.

1.2 Applications of automated theorem provers

A detailed description of theorem provers and different logics is given in the background chapter that follows. For the purpose of this chapter a theorem prover is a computer program that is given a mathematical or logical statement (a conjecture) and seeks to find a proof that the statement is always true (a theorem) or is not. The logical language is assumed to be first order logic and the proof search is taken to be automatic. Theorem provers working in higher order logic requiring a high degree of user intervention will be referred to as proof assistants.

As with many software tools, automated theorem provers were originally designed for a single purpose (computer mathematics) but now have a wide range of potential applications, which provide motivation for the work of making the theorem prover more accessible.

One application of fully automated theorem provers is to work in conjunction with more flexible but less automated proof assistants. Interactive proof assistants such as Isabelle [62] or HOL [29] are very flexible in terms of the descriptive power of the logics that may be used with them but they require a lot of expert input from the user. Automated theorem provers are much easier to use but are restricted in their descriptive power. By combining the two, using the automated theorem prover for those portions of the proof that may be expressed in first order logic (for instance), the overall proof process is made both quicker and easier. See the work of Meng, Paulson and Quigley [51, 33, 52].

A second application of theorem provers is that for which they were first developed; that is, as a tool for mathematicians. This application is listed here more for completeness rather than its current importance as despite decades of development of algorithms and the power of modern computers, useful theorems in most fields of mathematics are too difficult for automated theorem provers. Even where theorem provers can find proofs, the process may be more difficult than a straightforward pen and paper approach. For example Wiedijk [94] has collected proofs from different authors using different theorem provers to prove the irrationality of $\sqrt{2}$ as well as including an informal proof in standard mathematical notation. For many of the provers it is difficult for non-specialists to follow the proofs of what is a simple theorem.

But there are exceptions. The most famous one was the solution of the Robbins problem by McCune in 1997 [50], which had eluded human mathematicians since the 1920s. Larry Wos [96, 97] has also proved many results, mainly in the field of algebra, using the automated theorem prover Otter [49].

A related application is in adding intelligence to databases of mathematical theorems (and their associated proofs). Dahn *et al.* [17] used automated theorem provers to intelligently search for equivalent theorems within a database of mathematical theorems. The automated theorem provers are used to determine if a theorem that is mathematically equivalent to one entered by the user already exists within the database. Simple string matching or other standard techniques are not good enough for such an application, as the user may phrase the theorem quite differently to how it is stored, or the searched for theorem may be a logical consequence rather than a direct replica of existing theorems.

A key modern application of theorem provers and formal methods is in the verification of hardware and software designs. Hardware verification is important. The commercial cost of an error in the design of a modern microprocessor, for example, is potentially so large that verification of designs is essential. Automated theorem provers are just one in a range of tools that are used. The applicability of first order logic provers (of the type used in the work described in this dissertation) is extended when they are used in conjunction with higher order logic interactive proof assistants. See the work of Claessen and others for example [14] [38].

Software verification is of similar importance to hardware verification. Mistakes can be very costly, examples are the destruction of the Ariane 5 rocket (caused by a simple integer overflow problem that could have been detected by a formal verification procedure) and the error in the floating point unit of the Pentium II processor. Baier and Katoen give these and several other examples [6]. For further examples of applying automatic theorem provers to software verification see the work of Schumann [80], Denney, Fischer and Schumann [18] and Bouillaguet, Kuncak, Wies, Zee and Rinard [11].

Automated theorem provers have been applied to a wide range of other problems, some far removed from the original purpose of testing logical conjectures in mathematics. One example is network security. The importance of the internet and the need for government and commercial organisations such as banks to exchange data globally in a secure manner means that computer security is an important field. Secure data exchange requires a good encryption scheme, but the security of encryption schemes is not just dependent on the particular encoding method used; the whole procedure must be designed to avoid any potential back door or other exploit that may be used by malicious agents seeking to determine secret information being transmitted. Automated theorem provers may be used to prove the safety of security protocols or conversely find flaws within them. See for example the work of Cohen on the TAPS system [15].

Another example in a completely different field: automated theorem provers have been used to find geometric proofs (see for example the work of Li and Wu [43]). This is a branch of mathematical proof but it has also been applied to machine vision to check the veracity of spatial models derived from two dimensional images in the work of Bondyfalat, Mourrain and Papadopoulo [10].

Automated theorem provers also find application in the field of artificial intelligence. Artificial intelligence originally took a world view based on logic, in particular first order logic, and made extensive use of Prolog. More recently a statistical approach has been taken which more directly reflects the uncertainty involved with the real world. Despite this, logic and automated theorem proving can play a useful role in such aspects as common sense reasoning and the event calculus (used in planning actions) first put forward by McCarthy [48] and much more recently expounded by Mueller [58].

Finally, an example of a very different application of first order logic and theorem proving is in the field of sentient computing. See work published by Katsiri and Mycroft [37].

1.3 Choice of theorem prover

The range of applications for automated theorem provers and the fact that they are not currently easy to use provides motivation for making improvements in the degree of automation and reducing the level of expertise needed. But there is a wide range of theorem provers already written and several possible approaches to making them more useable. Justification is needed for the choice of theorem prover used in the described work.

As with the previous section, the reader is referred to the next chapter on background for a detailed explanation of different logics and types of theorem prover.

1.3.1 Automation versus expressive power

Theorem provers vary as to the amount of human guidance that is required in the proof search and as to the sophistication of the logical language that may be used to express the conjecture that is to be tested. The degree of automation possible and the sophistication of the logical language that may be used must be traded off against each other. A high degree of automation is only possible if the language is constrained. Proofs for flexible high order languages generally require human guidance and the associated theorem prover is referred to as a proof assistant. This is not down to a lack of human ingenuity or programming skill but is a mathematical property of the logics involved. There is a fundamental trade-off in automated theorem proving between the expressive power of a logic and the degree to which proving conjectures within that logic may be automated. The existence of this trade-off means that there is a spectrum of tools available to do the job of theorem proving, which unfortunately adds to the complexity of the problem for non-specialists. The lines between different classes of tools are also not distinct, as efforts have been made by writers of provers to extend the power without losing the desired level of automation.

1.3.2 SAT solvers

The simplest logic generally used is propositional logic and the associated prover is a boolean satisfiability solver or SAT solver. If a problem may be expressed in boolean terms then there exist many SAT solvers (for example zChaff [57]) which can automatically search for solutions and, in theory at least, give a deterministic answer as well as a model of boolean values where one exists. Any algorithm that may be programmed on a non-deterministic Turing machine may be expressed as a SAT problem.

SAT solvers are very useful but the expressive power of propositional logic is limited and boolean expressions also may become very large. Additionally the SAT problem was the first to be shown to be NP-complete in complexity (Cook's theorem [16]) so solving large problems may be exponentially hard. (In practice modern SAT solvers are able to solve some very large problems where an underlying structure exists whilst some other comparatively small problems cannot be solved.)

Software and hardware verification through the approach of model checking also works with propositional logic. Expressions are derived from considering a finite number of possible states arising in a state machine description of the problem. The expressions are manipulated in the form of binary decision diagrams or BDDs. See Baier and Katoen [6] for an introduction.

SAT solvers and model checking tools are already well automated, the constraint on use being the need to express the problem in an appropriate form rather than in running the prover itself.

1.3.3 First order logic theorem provers

First order logic adds predicates and quantifiers to propositional logic, which greatly increases the expressive power. The method of resolution introduced by Robinson (see next chapter on background) allows for the automation of proof search in first order logic. Though the basic inference step of resolution is simple, the heuristics needed to make the search process work in practice are much more complicated than those for SAT solvers.

Many conjectures arising from practical problems involve equality. Adding equality axioms to first order logic does not work well in practice, so equational theorem provers have been developed which include equality as part of the logic itself (*i.e.* first order logic with equality). Such theorem provers may also be automated but the search methods used are complex and different heuristics work better on different problems. *It is this class of prover that the work described in this dissertation is concerned with.* The motivation is that such provers are powerful, potentially automatic but currently require a degree of human expertise to run well which may be reduced or eliminated by the successful development of automatic procedures generated through machine learning.

1.3.4 Proof assistants

First order logic with equality is powerful, but even so there are some quite simple problems that cannot be expressed in it (for example reachability, as discussed by Huth and Ryan [32]). Mathematical induction is also too high level to be expressed in first order logic but it is extremely useful for many aspects of software and hardware verification. There are therefore many circumstances where proofs in higher order logic must be found. The proof search may be aided by computer but is difficult to automate. This is the domain of proof assistants such as Isabelle [62]. Proof assistants are very powerful but require a high level of user expertise and this requirement for expertise is unlikely to be easily removed. Additionally, by working on a first order theorem prover and combining such work with existing work on combining theorem provers, some improvement can be transferred to proof assistants as in the work of Meng, Paulson and Quigley [51, 33, 52].

1.3.5 Prover used

The work described in this dissertation is concerned with automated theorem provers working in first order logic, that is theorem provers that should not require human intervention (though in practice they currently work much better with expert human input). The selected theorem prover was E [78] which supports first order logic with equality. E is relatively efficient and is potentially fully automatic while still being flexible enough to express quite powerful conjectures. The choice of theorem prover was also governed by the availability of already written and available code (as it would be a major task and not sensible to attempt to write a new prover). The use of a pre-existing prover also allowed a judgement to be made in terms of performance (in public competition with other theorem provers).

1.4 Importance of heuristic selection

As described in more detail in the next chapter on background, the core of a first order logic automated theorem prover is a proof search process based on using simple inference steps to combine existing logical clauses to create new ones until an empty clause is found. The search space of clauses keeps growing as the search proceeds and whether the search is successful or not depends on a number of choices made during the process, which choices are made is determined by the heuristic being followed. Schulz discusses the main choice points in his thesis [77]. A key point is that the best heuristic to follow depends on the conjecture whose proof is sought. There is not a single heuristic that will be best for all problems.

The motivation for looking at heuristic selection is that for the type of theorem prover selected, the main barrier to effective use by non-experts is the need to select a good heuristic. Though the existing prover has a flag that, if set, causes the program to select the heuristic, this auto mode is based on a fairly coarse classification of problems in conjunction with using the best heuristic for problems from a library that fall into the same class. To manually select a good heuristic requires a level of expertise that comes only from much use of the prover on many problems.

The thesis of the current work is that modern machine learning techniques should be able to find a more sophisticated functional relationship between the conjecture to be proved (with its related axioms) and the best method to use for the proof search.

Previous work on applying machine learning to theorem proving has concentrated on modifying an heuristic in light of previous proofs with only partial success. The work described in this dissertation concentrates on using established heuristics which are known to give good results and applying machine learning to match the best heuristic to a given proof problem.

1.5 Motivation for using machine learning

A study of the problem of automatic heuristic selection within a first order logic theorem prover leads to the conclusion that it has characteristics that are a good match for a machine learning approach.

Firstly, the best heuristic to use in an automated theorem prover depends on the problem (the conjecture to be proved and the related axioms) which indicates that there exists a relationship between problem and heuristic choice. The relationship though is not obvious even to human experts who have worked a long time in the field, so attempting to find it analytically is unlikely to be successful. Machine learning is designed to model relationships which are too complex for analysis, and has proved successful in such cases as handwriting recognition where the connection is similarly difficult to define.

Secondly, though it would be useful to know why some heuristics work better with some types of problems (and would help with heuristic development) there is still a lot to be gained from developing a black box routine that takes as input some straightforward measures (features) of the problem and produces as an output the index of the best heuristic to be used. Such a scenario is a standard classification problem, for which the field of machine learning was developed. By running all possible heuristics (out of a limited set) on a large number of sample problems, the best heuristic in each case can be determined, which provides samples for supervised learning. (Supervised learning is the process of using previous examples with known outcomes to learn the rules by which new examples should be classified.)

Even though the machine learning approach tends to lead to a black box function between features and the choice of heuristic, by analysing which features are important some clues can be provided to help in future heuristic development. Extensive feature selection work was done and described in this dissertation in chapter 6.

1.6 Dissertation summary

The present chapter has provided motivation for improving the accessability of automated theorem provers by describing some of the applications that these tools may be put to. In addition arguments have been put forward for selecting machine learning as a method and applying it to heuristic selection.

As the research straddles the disparate areas of logic and machine learning, chapter 2 gives background on both. In addition to the main experiment on heuristic selection a preliminary experiment was undertaken on classifying problems between provable ones and those that cannot be solved within a given time constraint. Chapter 3 covers methodology for both experiments. Chapter 4 gives details of the preliminary experiment. Chapter 5 covers the main heuristic selection experiment. Chapter 6 gives the results of a series of feature selection experiments and finally the work is summarised and conclusions drawn in chapter 7.

Chapter 2

Background

The work covered in this dissertation straddles two disparate fields of computer science: automated theorem proving and machine learning. To enable experts in either, or neither field, to read it with the minimum of external reference, this background chapter briefly covers the basics of both subjects. The main purpose is to place the choices made (of logic, machine learning techniques and so on) into an overall context.

2.1 Logic

Though the study of logic dates back to the ancient Greeks, the modern subject flowered towards the end of the nineteenth century and the first few decades of the twentieth century with major work being done by Peano, Frege, Russell, Whitehead, Church, Gödel and Turing amongst many others. As this section is concerned with background, references are generally to more recent books rather than original papers. Many of the original papers (in English translation) are found in van Heijenoort [89].

2.1.1 Logic levels or languages

Natural languages such as English are very expressive, able to convey subtle nuances of human thought and emotion. They have the expressive power to convey information, but they are also ambiguous and imprecise. Often the understanding of the reader or listener depends not only on what is written, but also on additional knowledge and experience that is assumed. In Shakespeare [81], Macbeth's expression of the bleakness of life following the death of his wife:

> "to-morrow, and to-morrow, and to-morrow, creeps in this petty pace from day to day"

is easily understood by most English speakers but would be very difficult for a computer programmed with an English dictionary and a set of grammar rules to comprehend.

Formal logical languages were developed by philosophers and mathematicians as a means of expressing arguments and mathematical theorems in an unambiguous way. Assumptions or premises are all explicitly stated. The steps of the argument or proof must follow defined rules to reach a conclusion. Though different proofs may be produced by the choice of different rules or a different ordering of the premises, a consistent set of premises must never produce contradictory conclusions. In general, logical languages sacrifice expressive power for ease of proof finding. If the language is restricted sufficiently, proofs may be found in a deterministic manner but the conjectures that may be written in the language are limited.

Well formed formulae and inference rules

Each logical language has rules defining what constitutes an acceptable or *well formed formula* in that language. For a conjecture to be proved as a theorem it must first be expressed as a well formed formula and this is not always possible; expressibility is discussed in a separate section. The proving of a theorem also requires inference rules. Rules of inference are relationships between sets of well formed formulae. A proof may begin with a set of premises or axioms which are converted via a series of applications of inference rules to a final set which contains the theorem. In practical proof systems it is often simpler to demonstrate that the negation of a theorem is inconsistent with the initial axioms; this is covered in the section on proof methods.

The following sections describe the more important logics in terms of the rules for well formed formulae. Inference rules are discussed in the section on proof methods.

Propositional logic

The most restricted or lowest level logical language is that of propositional logic. The basic unit of the language is the proposition, which is a statement that is either true or false. Propositions may be joined by conjunctions (logical "AND") or disjunctions (logical "OR") additionally the negation of a proposition is permissible and given an appropriate symbol. Other logical statements such as implication may be expressed in terms of disjunctions, conjunctions and negation. As logic is not concerned with the propositions themselves, only in their truth or falsehood and what this implies for the truth or falsehood of logical sentences, they are normally labelled as single letters or numbered variables. Thus a proposition may be a statement such as the classic "all men are mortal" or it may represent a bit value in a digital circuit, the logic is unaffected.

Well formed formulae in propositional logic may be defined inductively. Firstly propositions may be considered as variables over the domain represented by the set {1,0}, where 1 represents true and 0 represents false.

$$A\ proposition\ is\ a\ well\ formed\ formula.$$

If ϕ is a well formed formula then so is

$$\neg\phi\ (\ NOT\ \phi\ ).$$

If ϕ and ψ are well formed formulae then so are the conjunction

$$\phi \wedge \psi\ (\ \phi\ AND\ \psi\ ),$$

the disjunction

$$\phi \vee \psi\ (\ \phi\ OR\ \psi\ ),$$

the implication

$$\phi \to \psi$$

and the equivalence

$$\phi \leftrightarrow \psi \ .$$

The equivalence $\phi \leftrightarrow \psi$ can be expressed as a conjunction of two implications :

$$(\phi \to \psi) \wedge (\psi \to \phi).$$

The implication $\phi \to \psi$ can be expressed as the disjunction

$$\neg\phi \vee \psi.$$

A well formed formula may be converted to one of several *normal forms*. The most common such form is conjunctive normal form. *Conjunctive normal form* consists of a conjunction of clauses each of which is a disjunction of literals where a literal is a proposition or the negation of a proposition.

A conjecture expressed as a well formed formula may be valid, satisfiable or inconsistent. A valid formula is true for all values of the constituent propositions. A simple example of a valid formula is $(A \vee \neg A)$. A satisfiable formula is true for some assignment of values to the constituent propositions (or variables). A formula is inconsistent if it is false for all values, so if a formula is valid it's negation will be inconsistent. A simple example of an inconsistent formula is $(A \wedge \neg A)$.

First order logic

First order logic extends propositional logic to predicate logic. Whereas a proposition is either intrinsically true or false, a predicate is a truth valued function of terms which may be defined over any non-empty set or domain. The terms may be constant values (elements of the domain), variables or functions over the domain. Predicates may also be viewed as relations between elements of the domain. A predicate of arity n defines a relationship over the product set D^n. Additionally, variables (but not functions or predicates) may be quantified over, the two quantifiers being the universal quantifier $\forall$ (for all elements of the domain) and the existential quantifier $\exists$ (for at least one element of the domain). The requirement that the domain be non-empty avoids logical inconsistencies such as $\forall x P(x)$ being true whilst $\exists x P(x)$ is false (where x is a variable and P a predicate).

Conventionally notation within first order logic assigns capital letters to predicates (which are truth values) and lower case letters to constants, variables and functions. Constants are generally assigned letters early in the alphabet such as a, b, or c whilst variables are assigned letters towards the end of the alphabet such as x, y or z. Functions are given the letter f or following letters such as g or h. These conventions are not rigid, and function names in particular may be assigned in mathematical notation such as sin or cos or may be symbolically expressed using the standard arithmetic operators. Similarly, constants will often be expressed symbolically such as particular integer values for the domain of natural numbers. For machine-based systems it is convenient to have all functions in prefix form but for readability the use of infix functions in some systems is allowed.

In a similar fashion to propositional logic, well formed terms and formulae can be defined inductively.

The definition of a term is as follows (it is implicitly understood that constants are fixed elements of the domain, variables range over elements of the domain and functions of arity n are $D^n \Rightarrow D$ where D is the domain).

Constants or variables are terms.

If f is a function of arity n and $t_1, \ldots, t_n$ are n terms then

$$f(t_1, \ldots, t_n)$$

is also a term. Well formed formulae are defined as follows, if P is a predicate of arity n (where n may be zero) and $t_1, \ldots, t_n$ are n terms then

$$P(t_1, \ldots, t_n)$$

is a well formed formula. If ϕ is a well formed formula then so is

$$\neg\, \phi.$$

If ϕ and ψ are well formed formulae then so are the conjunction

$$\phi \wedge \psi,$$

the disjunction

$$\phi \vee \psi,$$

the implication

$$\phi \rightarrow \psi$$

and the equivalence

$$\phi \leftrightarrow \psi.$$

If ϕ is a well formed formula and x is a variable then

$$\forall x\ \phi$$

is a well formed formula and so is

$$\exists x\ \phi.$$

First order logic with equality

The nature of equality has been debated for centuries, an often referenced philosophical discussion relevant to logic is that of Max Black [8], but much of such philosophical debate lies outside computer science.

Philosophers differentiate between equality and identity. Stating that all men are equal is not the same as saying that all men are identical. This difference is also important in mathematics, for instance to count members of a set or to express the idea that exactly three elements have a particular property. Equality that corresponds with element identity is sometimes referred to as numeric equality, after Aristotle, who differentiated between proportional equality and numeric equality, and the term will be used here as a convenience.

Nieuwenhuis and Rubio in the Handbook of Automated Reasoning [61] state the following congruence axioms for dealing with equality by resolution:

$$\rightarrow\ x \simeq x \quad \text{(reflexivity)}$$

$$x \simeq y\ \rightarrow\ y \simeq x \quad \text{(symmetry)}$$

$$x \simeq y \wedge y \simeq z\ \rightarrow\ x \simeq z \quad \text{(transitivity)}$$

$$x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n\ \rightarrow\ f(x_1, \ldots, x_n) \simeq f(y_1, \ldots, y_n) \quad \text{(monotonicity-I)}$$

$$x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n \wedge P(x_1, \ldots, x_n)\ \rightarrow\ P(y_1, \ldots, y_n) \quad \text{(monotonicity-II)}$$

The first three, reflexivity, transitivity and symmetry are straightforward to express in first order logic so an equality predicate can be defined with these qualities. Such a predicate would not be sufficient for numeric equality, for instance in the domain of natural numbers the predicate defined by equality modula division by a prime would satisfy reflexivity, transitivity and symmetry but there would be an infinite number of elements in each equivalence class. For numeric equality the monotonicity axioms are also required but these are not single axioms - they are axiom schemes. "One monotonicity-I axiom is required for each non-constant n-ary function symbol f, and, similarly one monotonicity-II axiom for each predicate symbol P" Nieuwenhuis and Rubio [61].

From a practical point of view, in automated theorem provers equality needs to be treated as a special case to prevent an explosion in the number of intermediate clauses generated in the proving process. Additionally, having a specific equality predicate allows the efficient rewriting of elements within a clause as part of logical calculi or inference rules Bachmair and Ganzinger [4].

Well formed formulae in first order logic with equality are as for first order logic with the addition of a specific equality predicate of arity two. A restricted form of first order logic with equality is equational logic. In equational logic the only predicate is equality. Other predicates can be expressed as equality with the use of the special symbol $\top$, The predicate

$$P(t_1, \ldots, t_n)$$

becomes the equational literal

$$f_P(t_1, \ldots, t_n) \approx \top,$$

though the function f_P is restricted to being a head function (i.e. not a parameter within any other function) and similarly for $\top$. Such a contrivance seems messy but does allow a consistent calculus to be used in an automated theorem prover such as E written by Schulz [78].

First order logic with equality is the basis of many automated theorem provers including that used in the work described in this dissertation.

Higher order logics

Higher order logic allows quantification over functions and predicates as well as elements of the domain. A variable can itself be a function (not just the result of a function) or a predicate and can be quantified over to express properties that hold for all functions or predicates. Set theory can be expressed in higher order logic in a direct and natural way, in contrast to the convoluted approach required to express any sort of set theory in first order logic. Zermelo-Fraenkel set theory can be expressed in first order logic but it is then complex to express even simple concepts such as the ordered pair. Thus higher order logic extends the expressive power of the language but it is at a cost in terms of decidability and the degree to which the proof process may be automated. Standard second order logic is covered in Manzano [46].

Many-sorted logic and types

In first order logic, terms are of a single type or sort, that is they range over elements of a single set, the domain. Many-sorted logic (Walther[93], Manzano [46]) assigns different types or sorts to the elements of the domain. Many-sorted first order logic does not extend first order logic as it can be translated to first order logic (Enderton [23], Manzano [46]) but it provides an additional set of constraints which restrict the search space for an automated theorem prover allowing a longer proof to be found within practical computer resource constraints Walther [92].

Manzano [46] argues that many-sorted logic is a universal language suitable for expressing other logics, though this thesis is not universally accepted as Venema makes clear in his review [90].

Type theory introduces types to logic but, unlike sorts in many-sorted logic, types have a hierarchy and some types may be contained within other types. The notion of types avoids certain paradoxes such as Russell's paradox in set theory. Types are an intrinsic part of the functional programming language ML (Meta Language), which is used at the meta logic level of such proof assistants as Isabelle [62] and HOL [29]. In such systems, theorems are themselves a type and can only be returned by functions that are valid proofs from axioms or existing lemmas and theorems. Meng and Paulson [52] discuss the expression of types within first order logic.

Modal, temporal and other logics

In first order logic predicates are either true or false, the truth value may depend on terms but there is no element of time or state that may change. For modelling computer systems and software the concept of state is important, so it is useful to consider logical statements that are true in some states but not in others. Similarly, when considering intelligent agents (in the context of artificial intelligence) the truth of a predicate may depend on the knowledge of a particular agent.

Modal logics extend first order logic with the introduction of two symbols:

$$\Diamond \textit{ and } \Box.$$

The interpretation of these symbols depends on which of the many logics is being considered and the underlying scenario to which it is being applied. The most straight forward interpretation is probably temporal where:

$$\Diamond \; \textit{means} \text{ “will be true at some point in the future”}$$

and

$$\Box \; \textit{means} \text{ “is true now and for ever more ”.}$$

Temporal logics are a type of modal logic which are particularly associated with software or hardware systems modelled by state machines. They are concerned with such questions as to whether particular predicates hold for all future states. There are two main temporal logics: linear time logic (LTL) and computation tree logic (CTL), see Huth and Ryan [32] for details. These logics are associated with the verification technique of model checking (Baier and Katoen [6]).

There are many other non-classical logics, including ones where truth is not bimodal but can take on intermediary values, such as in multi-valued logic and fuzzy logic. A survey of these is given in Priest [68] from a purely logical or philosophical standpoint whilst modal and temporal logic is covered from a more practical computer science approach in Huth and Ryan [32].

2.1.2 Proof methods

Standard proof methods in logic are generally at a much lower level than those used in mathematical proofs as published in mathematical text books, in a manner analogous to the difference between machine code and high level computer programming languages. This is particularly true of automated theorem provers working in first order logic and less so of proof assistants working in higher order logic with a large library of already proved lemmas and human guidance in the proof process. An interesting comparison is given in Wiedijk [94] where the same theorem, the irrationality of the $\sqrt{2}$, is proved with seventeen different theorem provers as well as by hand (though the machine proofs are human guided and not fully automatic).

Syntax, semantics, interpretations, valuations and models

The definitions of well formed formulae given in the previous sections are simply rules relating to the syntax of symbols. Similarly, inference rules provide a grammar for syntactical manipulation of formulae. For a logic to be useful, meaning or semantics must be attached to the symbols. Restricting the discussion to first order logic, an *interpretation* is a mapping of function symbols to specific functions and predicate symbols to specific relations (over a specific domain). A *valuation* is an assignment of values (specific members of the domain set) to each variable. A combination of an interpretation and a valuation is a model.

Validity, consistency and inconsistency

A formula is *valid* if it is true under all interpretations. A set of formulae are *consistent* if there is a model which makes them true. A set of formulae is inconsistent if there is no model for which they are true.

In general, proving a formula

$$\psi$$

from a set of axioms

$$\phi_1 \wedge \ldots \wedge \phi_n$$

is equivalent to demonstrating that

$$\phi_1 \wedge \ldots \wedge \phi_n \rightarrow \psi$$

is valid, but this is equivalent to there being no interpretation in which its negation

$$\neg(\phi_1 \wedge \ldots \wedge \phi_n \rightarrow \psi)$$

holds, that is

$$\phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\psi$$

has no model or is inconsistent.

In practical terms it is often more straightforward to demonstrate the inconsistency of the negation of the formula with the axioms than it is to show the validity of the un-negated formula.

Soundness and completeness

A logical system is *sound* if for any proof that is syntactically valid the semantics of the premises and the conclusion agree in all interpretations. Informally it is a statement that following the rules of inference will lead to a correct proof in all circumstances. *Completeness* is complementary to soundness: it is the property that any correct theorem can be stated syntactically within the system. Propositional logic is complete as the truth table for any well formed formula can be constructed from the truth tables of its constituent parts, and any such truth table will be finite (though exponentially large in the number of variables). First order logic is also complete but the proof is more involved, see for instance Kaye [39]. Note that completeness is not equivalent to decidability. For a system to be decidable there must be an algorithm that will prove or disprove any conjecture within a finite number of steps. (Gödel proved that any system that can encompass arithmetic is incomplete (Smith [84]).)

Proof process

The starting point of a proof is a set of axioms which are assumed to be true and a conjecture which, if proved, will become a theorem. There are then two general approaches. One is to find a chain of logical inferences connecting some or all the axioms to the conjecture. The other is to negate the conjecture, add it to the axioms and then show there is a chain of inferences that lead to a contradiction. If the negation of the conjecture is inconsistent with the axioms then the original conjecture is valid (true for all models or values of variables within it). The former approach is used in natural deduction, sequent calculus and related methods. The latter approach is used in resolution based theorem provers and other similar automated reasoning systems.

Natural deduction and the sequent calculus

Natural deduction and the *sequent calculus* are proof systems for first order logic (and propositional logic). Elements of natural deduction are similar to the type of philosophical logical reasoning introduced by the ancient Greeks, but its origin as a complete system is much more recent. Pelletier, in his history [65], traces the origin to a lecture by Łukasiewicz in 1926 which inspired Jaśkowski to publish a system in 1934 [35], whilst at the same time, and working independently, Gentzen published a system in his two part 1934/1935 paper [27]

The inference rules in natural deduction involve either the elimination of or the introduction of a logical connective or quantifier. There also may be connected side rules involving the substitution of variables or the renaming of variables and so on. For example, if

$$A$$

is known to be true (i.e. is a premise) then

$$A \vee B$$

is also true. This is an example of $\vee$ introduction. In contrast to the very simple rule for $\vee$ introduction, the rule for $\vee$ elimination is more complex. Starting with

$$A \vee B$$

two additional subproofs are needed, each of which has the same conclusion, say C, but the starting premise of one is A and the starting premise of the other is B. Thus if both A and B lead to C then

$$A \vee B$$

may be replaced by

$$C$$

and the $\vee$ is eliminated.

In some rules, an assumption is made then a series of inferences from the assumption are followed to a conclusion. For example if A is assumed to be true and a valid inference process then leads to B being true then an implication may be introduced as

$$A \to B$$

($\to$ introduction). If A is assumed and the inference steps lead to a contradiction then

$$\neg A$$

is true ($\neg$ introduction).

Finding a proof in natural deduction is often done in reverse. Starting with the conjecture as a conclusion, a rule is chosen and the premises that would lead to the assumed conclusion are deduced. The process is then repeated, with the premises now being the conclusion of further inference rules. Any premise that is an axiom is known to be true so does not need further work. A successful proof search will lead to a set of premises which are all axioms and then the formal proof may be read off in the reverse order to which it was found. Natural deduction is covered in Huth and Ryan [32].

The sequent calculus was invented by German logician Gerhard Karl Erich Gentzen in 1934 as a means of studying natural deduction [27], hence such systems are also called Gentzen systems. Sequent calculus is very similar to natural deduction except that it is expressed as formal rules between logical statements of the form

$$\phi_1, \ldots, \phi_m \Rightarrow \psi_1, \ldots, \psi_n$$

where the left hand clauses

$$\phi_1, \ldots, \phi_m$$

are a conjunction (all have to be true) and the right hand clauses

$$\psi_1, \ldots, \psi_n$$

are a disjunction (at least one is true). A base sequent, which plays a corresponding role to that of an axiom in natural deduction, is one where one (at least) of the

$$\phi_1, \ldots, \phi_m$$

is the same as one of the

$$\psi_1, \ldots, \psi_n.$$

Such a base sequent is trivially true.

Inference rules in sequent calculus correspond with those of natural deduction but where introduction and elimination are instead expressed as right or left (of the $\Rightarrow$) rules. The advantage of the sequent calculus is that rather than assumptions being made and later discharged at different steps (as is done in natural deduction) the environment is contained within the sequents themselves. The clauses that don't take part in a particular rule are collected into sets denoted by either Γ or Δ.

There is potentially a large number of rules. This works well for doing proofs by hand but there is a lot of redundancy. For an automated approach the redundancy may be removed by restricting the rules to a minimal set. See Gallier [26].

Tableaux methods

The construction of a proof in a sequent calculus results in a tree structure: the root of the tree is the conclusion and the leaves are the axioms (in a proof of validity). An alternative approach is to start with the negation of the conjecture at the root and demonstrate that all branches lead to a contradiction to show inconsistency. This is the tableaux method. Inconsistency occurs when a branch contains both ϕ and $\neg\phi$; such a branch is closed. Tableaux methods can form the basis of an automated theorem proving system, generally using the reduced set of sequents.

Resolution, unification and factoring

Natural deduction, the sequent calculus and tableaux methods are well suited to proofs by hand where the relatively large number of possible inference steps gives flexibility. But for automatic proof systems, flexibility is undesirable as it increases the size of the search space. The resolution method of Robinson [75] has a single inference step and is well suited to computer proof systems (see Bachmair and Ganzinger [5]).

Resolution is used in proving inconsistency (i.e. when proving a conjecture is a theorem by demonstrating that the negation of the conjecture is inconsistent with the axioms). The axioms and negated conjecture are expressed in *conjunctive normal form* as a set of clauses. (A clause is an element of a larger logical expression. In conjunctive normal form the overall logical expression is expressed as a conjunction of clauses so all clauses must be true for the expression to be true. Each clause consists of a disjunction of literals so that the clause will be true if one or more literals is true. Since all clauses must be true for the overall expression to be true, they can be treated as a set of facts and combined to generate new clauses that follow in the same way as "all men are mortal" and "Socrates is a man" can be combined to state that "Socrates is mortal".) Clauses are combined by resolution to produce new clauses, which are logical implications of the original clauses. If the new clause is empty then a contradiction is proved and the original clause set must be inconsistent.

In propositional logic two clauses

$$\{\phi_1, \ldots, \phi_n, A\}$$

and

$$\{\psi_1, \ldots, \psi_m, \neg A\}$$

can be resolved to give

$$\{\phi_1, \ldots, \phi_n, \psi_1, \ldots, \psi_m\}$$

The same approach is used in first order logic but is complicated by the need to operate on truth valued functions of terms (predicates) rather than simple variables. Terms may need to be unified first, which is the process of making two terms equal by a suitable substitution of variables by other terms. (Such a procedure is justified by the fact that there is a single domain so all variables and terms range over elements of the same set. Additionally, variables are universally quantified so if a predicate containing a term containing a variable is true it will remain true if any element of the domain is substituted for the variable.) Any such substitution may reduce the generality of the original term (for example in substituting $f(y)$ for x there is no guarantee that $f(y)$ spans the whole domain even though y does and furthermore $f(y)$ cannot be unified with $g(z)$ whilst x can). At each step at which unification is carried out, the loss of generality is kept to a minimum by using a substitution that results in the most general case possible, which is called the *most general unifier* or mgu for short. Note that for completeness, *i.e.* to ensure that the empty clause may be found, resolution should also include factoring. *Factoring* is the process of making a pair of literals within a clause equal by a suitable substitution for variables (unification) and consequently reducing them to a single literal.

2.1.3 Decidability and semi-decidability

Propositional logic is decidable as it is possible to systematically construct a truth table from the constituent parts of a well constructed formula. There will be a finite number of rows in the truth table (2^n rows if there are n variables) and the truth value for each row can be calculated in a finite number of steps.

First order logic is not decidable. This can be demonstrated by relating it to the halting problem, which Turing proved undecidable (see, for example Huth and Ryan [32]). Applying restrictions to first order logic, as is done in the Prolog system, can make it decidable. Additionally, standard first order logic is semi-decidable. That is, if a conjecture is a theorem then the proof can be systematically found in a finite number of steps, but if it is not a theorem then the process may not halt.

The decision problem (decidability) is also determinable for various restricted systems. For example, Presburger [66] gives a decision procedure for integer arithmetic restricted to just plus (*i.e.* no multiplication), Tarski [86] gives a decision procedure for elementary algebra and geometry and Shostak [82] gives a procedure for arithmetic with function symbols. The procedure of Satisfiability Modula Theories (SMT) combines such decision procedures into a single satisfiability solver. An SMT solver is fundamentally a SAT solver where the Boolean expression is not confined to propositions but instead contains predicates which are then tested within a separate theory. The part of the solver that tests the predicates must be closely integrated within the SAT solver including allowing for back tracking. See for example Tinelli [87].

2.1.4 Expressive power

The choice of a logic system for computer based theorem proving necessarily involves a compromise. There is a trade-off between the extent that the proof process may be automated and the sophistication of the formulae that may be expressed in the language. SAT solvers, which operate in propositional logic, are widely used but are very restricted in what may be expressed. SAT solvers are usually used to find a satisfying model (truth assignment of the variables) rather than to prove validity (or inconsistency).

The language of first order logic with equality ($\mathcal{L}_{\approx}$) is much more powerful than propositional logic but still has limitations. Though $\mathcal{L}_{\approx}$ can express Zermelo-Fraenkel set theory (ZF or with the axiom of choice ZFC) and

> "all of the results of contemporary mathematics can be expressed and proved within ZFC, with at most a handful of esoteric exceptions"

(Wolf [95]), from a practical perspective it is difficult to work at such a low level; higher order logic provides a more natural expression of set theory. Similarly $\mathcal{L}_{\approx}$ can express the axioms of Peano arithmetic except for that of induction, which in its most straightforward form requires quantification over sets (second order logic). One way around this is to use a separate axiom for each formula, but this leads to an infinite number of axioms (an axiom scheme) Wolf [95]. Similarly, $\mathcal{L}_{\approx}$ can be used to say a structure is a group but cannot express the concept of a simple group (Kaye [39]).

Despite its faults, for automated theorem proving $\mathcal{L}_{\approx}$ is powerful enough to be useful whilst being sound, complete and semi-decidable. More powerful logics generally require human intervention to guide proofs, which requires specialist expertise.

Some of the limitations of the expressive power of $\mathcal{L}_{\approx}$ can be overcome in practical instances where objects are finite. For example $\mathcal{L}_{\approx}$ cannot express reachability in a graph (or equivalently transitive closure) but it is possible to simulate reachability within $\mathcal{L}_{\approx}$ for finite structures, as shown by Lev-ami *et al.* [42]. Another approach is to combine $\mathcal{L}_{\approx}$

theorem provers with separate proof systems in a similar manner to the SMT approach for SAT solvers; this has been done with the theorem prover SPASS where it was combined with an arithmetic system SPASS+T Prevosto and Waldmann [67].

2.2 ATPs versus proof assistants

Historically, early work on mechanical theorem proving concentrated on automated methods in first order logic. This was the case from the early 1960s through to the late 1980s. In the last couple of decades more emphasis has been placed on proof assistants operating in higher order logic, as such logics make software verification and other tasks easier to define in formal terms, though difficult and time-consuming to then carry out. There has also been recent work on combining the two allowing some sections of the proof to be found automatically by first order logic theorem provers within the context of a higher order proof assistant, *e.g.* Meng and Paulson [52].

Computer based theorem proving forms a spectrum from fully automatic SAT solvers through to proof assistants where the human user is an expert and drives the proof process. At the SAT solver end of the spectrum there are many existing tools which require no specific user expertise to run. At the other end of the spectrum, where higher order logic is used, the process cannot be fully automated. Though first order logic is undecidable, for conjectures where a proof is possible the process should be amenable to automation but in practice some human expertise is still needed to set a large number of parameters to determine heuristic choices and allow proofs to be found in many practical cases.

The knowledge needed to set appropriate parameters for such automated theorem provers is specific to those who have worked with them (or developed them) so even experts working with proof assistants who wish to combine the two techniques (see for example Meng and Paulson [52]) may not be able to set optimal parameter values.

The motivation of the work described in this dissertation is the removal of the need for this specific expertise, replacing it by machine intelligence through machine learning.

2.3 Resolution based theorem proving

This section covers automated theorem proving of the type used in the main work of the thesis. Some detail is required to provide the necessary background to some of the features (measures) used to characterise the conjectures in the machine learning process.

2.3.1 Resolution and related calculi

Robinson [75] introduced a simple calculus for mechanical theorem proving based on showing inconsistency of the negation of a conjecture and associated axioms using resolution. The problem of theorem proving is reduced to that of searching for the empty clause via a simple clause generation inference process. Other approaches to automated theorem proving were developed at a similar time, for example Loveland's [45] model elimination method, but most modern first order logic automated theorem provers are based on refinements of the resolution method.

The starting point for a resolution based proof is a conjecture expressed as a set of quantifier-free clauses in conjunctive normal form (CNF). Quantifier-free means that there are no existential quantifiers and all variables are assumed to be universally quantified so that

$$P(x, y)$$

for example is taken to be

$$\forall x \forall y P(x, y).$$

Additionally, as the universal quantifier is distributive over the clauses in CNF, each variable may be considered local to the clause that it is in, which allows the renaming of variables to avoid clashes when combining clauses. For example, for two clauses $\phi(x)$ and $\psi(x)$,

$$\forall x \ (\phi(x) \wedge \psi(x))$$

is equivalent to

$$\forall x \ \phi(x) \ \wedge \ \forall y \ \psi(y)$$

where $\psi(y)$ is $\psi(x)$ with all occurrences of x replaced by y.

The requirement for the conjecture to be in quantifier free CNF is not a restriction. Any well formed formula in first order logic may be converted to CNF. Though a naive conversion to CNF may lead to an exponential increase in the size of the formula, there are efficient algorithms that perform the conversion (by the suitable introduction of new predicate symbols). Additionally, Skolem [83] (see Heijenoort [89] for an English translation) showed that existential quantifiers may be replaced by functions (named Skolem functions) whilst maintaining consistency (or, more importantly, inconsistency). That is any set of clauses in which this Skolemisation process has been used to eliminate quantifiers will be consistent if and only if the original set of clauses is consistent. See Nonnengart and Weidenback [63] for techniques for converting general first order logic into a suitable Skolemised CNF.

The procedure for proving a conjecture is a theorem is to replace the conjecture with its negation, combine this with the axioms and place in CNF to form a set of clauses. The clauses are then combined in pairs using resolution to deduce new clauses. If the empty clause is reached then inconsistency has been proved and the original conjecture is a theorem. For ground clauses (clauses without free variables) the process of generating new clauses will saturate so that after a point no new clauses are generated. If this happens without the empty clause being reached then the clauses are consistent (there is a model which satisfies them).

With non-ground clauses (i.e. clauses containing variables) resolution is combined with unification. Unification is the process of substituting terms for variables so as to make two terms equal. This process is allowable because all terms, including variables, represent elements of a single domain and since variables are implicitly universally quantified they will range over all values.

For non-ground clauses, the process of resolution is not guaranteed to saturate as this would violate Church's theorem (see Robinson [75]) but for the inconsistent case the empty clause will eventually be found (provided factoring is also carried out), so the process is semi-decidable.

Reducing redundancy

Given that the process of resolution generates a new clause each time it is applied, it is clear that the size of the search space can get very large very quickly. Robinson had already highlighted this problem even prior to resolution ([74]) and in his resolution paper suggested two processes to reduce the number of clauses. One of them, *subsumption*, is still used in modern theorem provers. (Subsumption is the process by which clauses may be deleted if there is another, smaller, clause for which any model will also be a model of the subsumed clause.)

Since Robinson introduced resolution as a basis for automatic theorem proving, he and others have worked on methods of reducing the number of redundant clauses that are generated. The simplicity of resolution, which is an advantage in a computer based theorem prover, leads to too many options for inferences being open. The efficiency of the process can be increased by imposing restrictions that reduce the search space as long as the process remains refutationally complete, so that inference steps leading to a proof are still available. The number of generated clauses may also be reduced by introducing new inference rules which effectively combine several resolution steps without generating the intermediate clauses. Such inference steps can take advantage of additional information, in particular many are formulated to use equality, which otherwise may lead to the generation of multiple redundant clauses if expressed in standard first order logic.

Note, the various methods of reducing redundancy do not invariably lead to improvements. Some of the choices that lead to different heuristics are those as to whether or not to use particular methods. *Understanding where to use particular options involves a lot of experience on the part of human experts and it is the aim of the work described in this dissertation to make these choices automatically, based on measures of the conjecture and axioms under consideration.*

Hyper-resolution

Robinson [72] introduced hyper-resolution, which is a multi-step resolution process where intermediate clauses are discarded. The clauses to be resolved are divided into two types: clauses with only positive literals are referred to as electrons and a selected clause containing one or more negative literals which is referred to as the nucleus. The nucleus is resolved with a series of electrons until the final resultant clause itself is an electron (contains no negative literals) and this is the output of the hyper-resolution step. Hyper-resolution is complete and will reduce the number of generated clauses as only one clause is generated for several resolution steps but the proof found may require more steps overall, negating some of the advantage. The theorem prover Otter and its successor Prover9 use hyper-resolution.

Set of support

Typically theorems exist within a context. In addition to the immediate premises of a conjecture, there will be existing axioms which are needed to reach the conclusion. In the proof search all the axioms must be included with the (negated) conjecture but in an undirected resolution search many clauses may be generated from combining axioms with each other. The set of support (SOS) strategy, Wos and Robinson [98], is designed to restrict inference steps to exclude any between axioms that do not involve the premises of the conjecture or clauses derived from them by earlier inference steps. Each inference step must involve at least one clause from the SOS, clauses arising from such inference steps can then be added to the SOS.

Equational reasoning

Resolution is not efficient for automatic reasoning involving equality. Such equational reasoning is best dealt with using special inference rules (Bachmair and Ganzinger [4]) which are described in the following sections. With the E theorem prover (used in the work that forms the core of this dissertation), all logical expressions are confined to clauses containing only equational literals so that pure equational reasoning may be used. This simplifies the prover as there is no requirement to mix different types of inference rules.

Demodulation

Demodulation, Wos and Robinson [99], uses the rewriting of terms to find if new clauses are equivalent to existing clauses and then discard them if they are. Terms are rewritten using instances of equal terms from a set of equal terms provided that there are no strictly more general instances of the equal term than the one used. (This generality requirement is to guarantee finiteness of the set of possible rewrites, see Wos and Robinson [99].) The rewriting step is repeated until no further steps are possible. Essentially each clause is simplified and then only kept if it has not already been found.

It should be noted that demodulation is not a canonical reduction procedure (Wos and Robinson [99]). Care must be taken in its use, as completeness may be lost. The combination of demodulation with set of support can greatly reduce the number of kept clauses and hence the number of redundant resolutions carried out.

Paramodulation

Paramodulation (Wos and Robinson [73], Nieuwenhuis and Rubio [61]) like demodulation makes use of equalities. In demodulation the equalities are separate clauses and thus must be true if the whole clause set is to be true. In paramodulation the equality used is a literal within a larger clause.

If A is a clause containing the term t_p at position p and the equality $t_i \approx t_j$ is a literal in another clause which can be expressed

$$B \vee t_i \approx t_j$$

and lastly t_i may be unified with t_p using substitution σ (the most general unifier) then the two clauses

$$(B \vee t_i \approx t_j) \wedge A$$

may be used to infer the single new clause

$$(B \vee A[t_j]_p)\sigma$$

where $A[t_j]_p$ indicates that term t_p has been replaced by t_j at position p in A.

As with resolution, paramodulation if unconstrained leads to the generation of a large number of redundant clauses. Term ordering and literal selection (described in the following sections) are used to restrict the number of paramodulation inferences that are carried out.

Term rewriting and superposition

Demodulation and paramodulation may use equalities in either direction, for example the literal

$$a \approx b$$

may be used within another literal to either replace a by b or to replace b by a . Term rewriting uses equalities, known as rewrite rules, asymmetrically. The left hand side may be replaced by the right hand side but not vice versa. A set of rewrite rules defines a relation between terms where one term is related to another if it may be obtained by a combination of substitutions and rewrite rules. To be useful, the set of rewrite rules should be selected such that the induced relation cannot contain an infinite number of rewrite steps (*i.e.* the series of rewrite links between intermediate terms should not contain any loops). Such a relation is well-founded and the rewrite system is terminating. Terminating systems guarantee the existence of normal forms and the induced relation is known as a reduction relation. A *normal form* is one that cannot be changed further by rewrite steps. An additional property that is important is confluence. A *confluent* system will have unique normal forms, so that in which ever order rewrite steps are taken, the same normal form is reached.

A graphical way of viewing term rewriting is to consider possible literals as nodes on a graph. Replacing of terms by other terms converts literals to new literals and can be represented by edges on the graph. With rewrite rules the edges are directional. Avoiding infinite rewrite steps is equivalent to the graph being a directed acyclic graph (DAG). Normal forms are nodes with ingoing edges and no outgoing edges. Confluence is the property that starting from a single source node and following all possible edges will always lead to the same terminating node, the normal form (so the terminating node is path independent).

Superposition is paramodulation restricted to inferences that only involve left hand sides of possible rewrite steps, see Nieuwenhuis and Rubio [61] and Bachmair and Ganzinger [4].

Term ordering and literal selection

Subsumption allows redundant clauses to be deleted, but it is not sufficient to make the resolution or superposition calculus procedure efficient enough to be practical. To prevent an explosion in the number of clauses, the resolution process may be restricted by the introduction of term ordering. Term ordering may be done by applying artificial weights to variables and functions and combining them to give weights for terms (as in Knuth Bendix ordering), or an ordering is applied to functions and variables and then lexographically extended to a term ordering (as in lexographical path ordering). Term ordering is then used to impose restrictions on substitutions in paramodulation steps so that the resultant clause contains simpler terms. This may be considered as a means of only generating clauses by resolution that are simpler (by some measure) than their parent clauses.

Ordering systems were first introduced for term-rewriting systems (Dershowitz [22]). The two main orderings used in theorem provers are lexographical path ordering (LPO) (originally put forward in unpublished work by Kamin and Lévy, see Baader and Nipkow [3]) and the Knuth Bendix ordering (KBO) [41]. See Baader and Nipkow [3] for a review of ordering as applied to term rewriting systems.

Such orderings are particularly important for superposition calculus where equality is involved. The imposition of a suitable ordering constraint leads to a saturating system. Within a saturating calculus a point will be reached when further inferences between clauses within a retained set of clauses will only generate clauses that are already within the set or are otherwise redundant. If saturation is reached and the empty clause is not within the set then it will never be generated and it can be demonstrated that there is a model (*i.e.* the clauses are not inconsistent) see Bachmair and Ganzinger[4].

Important properties of a term rewriting system are confluence and termination, which is related to unfailing completion. Confluence is the property that which ever series of rewrite steps are taken the same, unique normal form is reached. Termination is the property that rewrite steps eventually reach a normal form, which cannot be further rewritten. Knuth Bendix ordering does not guarantee unfailing completion (*i.e.* that all terms can be reduced to normal forms leading to a saturated system) but developments of it do, see Nieuwenhuis and Rubio [61] and Bachmair and Ganzinger [4].

In addition to term ordering, it is also possible to further restrict clause generating inference steps by imposing a selection scheme on literals. Only selected literals can take part in the inference step, if a potential inference involves a literal which is not selected then the inference step is not performed. The E user manual (supplied with the E theorem prover) lists a number of selection strategies which may be applied.

Splitting

In seeking a refutation of a set of clauses S in union with a clause $\phi \vee \psi$, one option is to find two separate refutations, one of S in union with ϕ and the other of S in union with ψ. This process is often used in SAT solvers for propositional logic but the backtracking involved is expensive for first order logic theorem provers. It is available in the SPASS prover. The backtracking may be avoided by introducing new propositional variables, one for each split, that indicate the branch and retain a single set of clauses. This form of splitting is used in the Vampire theorem prover (Riazanov and Voronkov [70]) and also in the E theorem prover.

2.3.2 Practical implementations

Given clause and the DISCOUNT loop

Modern theorem provers for first order logic with equality work with a saturating calculus and use the given clause algorithm (Voronkov [91]). Whilst inference rules generate new clauses, at each step redundant clauses are removed. Saturation is reached when all possible inference steps will only generate existing clauses or redundant clauses. If the empty clause is found inconsistency is proved (which implies that the original conjecture, prior to negation, is a theorem). If saturation is reached without the empty clause being found then a model exists for the negated conjecture so the original conjecture is not true in all circumstances and is therefore not a theorem.

Inferences combine two or more clauses to produce a new clause. The proof search could operate on the set of all clauses using some heuristic to search for possible inferences but such a scheme would make it difficult to determine if the saturation point had been reached. It would also be difficult to determine which combination of clauses to look at in an arbitrary fashion. The given clause algorithm addresses both these issues. In the *given clause algorithm* the clauses are divided into two sets. One set, the processed set, consists of clauses which are saturated with respect of other clauses in the set. That is all inferences between clauses in the set have been done. (Note that the processed set is not saturated in the sense of such inferences generating only clauses in the same set, as clauses generated from many of the inferences are returned to the other, unprocessed set.) At each step of the given clause algorithm a single clause is selected from the set of unprocessed clauses and then all possible inferences involving that clause and clauses in the processed set are explored. Generated clauses are tested for redundancy (and also to see if they make any existing clauses redundant) before being added to the unprocessed clause set. The given clause is then added to the processed clause set.

There are variants of the given clause algorithm such as including or excluding clauses from the unprocessed set when checking for redundancy. The E theorem prover used in the work described in this dissertation uses the method originating in the DISCOUNT system (Denzinger et al. [21]).

2.4 Machine learning

2.4.1 General concepts

Machine learning as a term arises from the field of artificial intelligence, but it has close parallels with model fitting in statistics. In straightforward terms it is the process of fitting a computer model to a complex function on the basis of measured data rather than from, for example, physical arguments[1]. The structure of the computer model is normally a summation of a set of basis functions and fitting the model comes down to setting values to function parameters and to the weights applied to the individual basis functions. Though techniques such as neural networks are presented as being very general, there is always an underlying assumption of functional form, so the process is one of estimating parameter values of known functions rather than determining arbitrary new functions. (Some approaches, such as Gaussian processes, are non-parametric. The role of parameters is taken by elements of a covariance matrix.)

Machine learning is divided into two main types, supervised learning and unsupervised learning (there are other variations such as reinforcement learning but these are not relevant to the work covered in this dissertation). Supervised learning is where a set of known (previously measured) samples are used to determine estimates of function

[1]Mitchell [55] puts it that the "field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience"

parameters. The sample data consists of input values (known as *features*), which will be the arguments of the function to be learned, and outcomes which are the function values. Unsupervised learning involves seeking patterns in data. The work in this dissertation uses supervised learning exclusively.

Machine learning can be applied to two types of problems, which correspond to fitting models to discrete functions or to continuous functions. *Classification problems* place results into classes on the basis of measured features. The continuous case, where the output can take any real value, is referred to as regression. The work described in this dissertation involves choosing heuristics and so is a classification problem - each potential heuristic defines a class of problems for which the heuristic is the best choice.

In the machine learning process, an important concept is generalisation. *Generalisation* is the term used to describe how accurately a learned function predicts outcomes for data that is not part of the learning set. If the learning process uses too many parameters and is too flexible then it may be able to reproduce the learning set with a high level of accuracy but be highly inaccurate when applied to samples that are not part of the learning set. This is a case of *over-fitting*.

Within machine learning and statistics, there are two philosophical standpoints that can be taken. The frequentist view point, in simple terms, assumes that the probability of events is best estimated by taking it equal to the frequency of occurrences of the event in earlier experiments. A form of distribution (eg Gaussian) may be assumed for the data and the parameters for the distribution are estimated on the basis of measured data, usually by taking a maximum likelihood approach (the probability of the measured data is maximised as a function of the parameters of the distribution). This results in a single value for each parameter. In contrast, the Bayesian approach assumes a probability distribution for the parameter values. The assumed *prior distribution* for the parameter values is combined with the probability of the measured data viewed as a function of the parameter values (the *likelihood function*) using Bayes' theorem to give a posterior distribution for the parameter values given the measured data. In the full Bayesian approach predictions for new data are made in terms of probabilities which are obtained by integrating (or *marginalising*) over the whole posterior distribution of the parameter values.

The starting point for both approaches is generally the likelihood function. The likelihood function is not a probability distribution, but is closely associated with one. As a simple example, consider a single random variable x that arises from a normal probability distribution with some mean μ and standard deviation σ. If several sample values are measured, giving a set of x values, then for any given value of μ and of σ the probability of measuring the set of x values can be calculated. (To be strictly accurate, for a continuous distribution the measured values need to be classified into ranges rather than point values.) The calculated probability values, viewed as a function of μ and σ, yield the likelihood function. Though it is a probability function, it is not a distribution, as it is not normalised with respect to the parameters μ and σ. To continue the example, in a frequentist approach particular values of μ and σ would be determined by finding the maximum value with respect to each (hence the maximum likelihood method). In the Bayesian approach the parameters μ and σ would themselves be considered random variables with their own probability distributions, so there are further parameters associated with the probability distributions of the original parameters. Bayes' theorem allows the likelihood function to be combined with the assumed prior distribution of the parameters to give a posterior distribution — the probability distribution for the parameters given the known values of the measured data.

In mathematical terms, for the case of a classifier the Bayesian approach gives the probability of a new sample point $\boldsymbol{x}$ being placed in class 1 given a training set of samples $\{\ldots \boldsymbol{x_i} \ldots\}$ with their corresponding classes $\{\ldots C_i \ldots\}$ can be expressed as follows. In the following the training set is referred to as $\boldsymbol{s}$, the possible parameter values of the model (the hypotheses) are referred to as $\boldsymbol{h}$. First, from the integral form of the probability sum rule, the total probability of class C_1 being selected is the probability of both C_1 and the parameter values being $\boldsymbol{h}$ integrated over all possible values of $\boldsymbol{h}$

$$p(C_1 \mid \boldsymbol{x}, \boldsymbol{s}) \;=\; \int p(C_1, \boldsymbol{h} \mid \boldsymbol{s}, \boldsymbol{x}) d\boldsymbol{h}$$

further, the probability of C_1 and $\boldsymbol{h}$ is the probability of $\boldsymbol{h}$ multiplied by the probability of C_1 given $\boldsymbol{h}$

$$p(C_1 \mid \boldsymbol{x}, \boldsymbol{s}) \;=\; \int p(C_1 \mid \boldsymbol{h}, \boldsymbol{x}) p(\boldsymbol{h} \mid \boldsymbol{s}) d\boldsymbol{h}$$

Bayes theorem can then be used to re-express $p(\boldsymbol{h} \mid \boldsymbol{s})$

$$p(C_1 \mid \boldsymbol{x}, \boldsymbol{s}) \;=\; \int p(C_1 \mid \boldsymbol{h}, \boldsymbol{x}) \frac{p(\boldsymbol{s} \mid \boldsymbol{h}) p(\boldsymbol{h})}{p(\boldsymbol{s})} d\boldsymbol{h}$$

$p(\boldsymbol{s} \mid \boldsymbol{h})$ is the likelihood function, $p(\boldsymbol{h})$ is the prior distribution and $p(\boldsymbol{s})$ may be viewed as a normalising term which could be obtained from

$$p(\boldsymbol{s}) \;=\; \int p(\boldsymbol{s} \mid \boldsymbol{h}) d\boldsymbol{h}$$

It should also be noted that the above has left out, for reasons of clarity, the parameters of the prior distribution $p(\boldsymbol{h})$ which are referred to as hyperparameters.

For the Bayesian approach to make sense, the posterior distribution should be narrower than the assumed prior distribution so that the measured data narrows down the variance in the parameters. Careful selection of the form of the prior distribution in the light of the form of the likelihood function can give rise to a posterior distribution that is of the same form. These *conjugate* forms are particularly useful where the process is iteratively applied as more data is obtained.

The frequentist approach can lead to over-fitting. The Bayesian approach imposes a prior distribution so that sample data leads to shifts in the assumed distribution rather than an exact fit which reduces the likelihood of over-fitting; but the assumed distribution itself may not be a good model for reality. Note that the two approaches (maximum likelihood and Bayesian updating of a prior distribution to a posterior distribution) will converge in the limit of an infinite number of data points. The subject is well covered by Bishop [7].

2.4.2 Machine learning approaches

Though modern machine learning is a development of artificial intelligence, most of the methods are very similar to approaches developed within the more ancient field of statistical analysis. For example, Bayesian learning applied to a linear combination of basis functions based on an assumed Gaussian prior distribution leads to the same equations as arise in least squares error fitting. The Bayesian framework gives a sound basis to the process of selecting the sum of the squares of the differences as an error measure; the original developers may have chosen it on the basis of mathematical convenience, but the method is no different in its implementation. Additionally some methods which appear separate on deeper analysis are seen to be related. For example, as shown by Rasmussen and Williams [69], the relevance vector machine can be viewed as a special case of a Gaussian process and there is a close correspondence between the *maximum a posteriori* (MAP) probability solution of a Gaussian process classifier and the support vector machine (these and other terms in this paragraph are explained in the sections that follow). More complex methods in some cases can be viewed as extensions of simpler approaches, for example the nodes of a *neural network* are essentially simple *perceptrons*. Some general terms apply to a whole family of methods, in particular *kernel machines*. The parts of the terminology that arise from artificial intelligence often reflect an historic background in early attempts to mimic the human brain, for example "neural network" and "perceptron". Such terms imply a level of complexity or sophistication which is higher than the simple models on which they are based. Similarly "machine" and "agent" are often applied to computer programs that are designed for a single application rather than the multi-functional capability that a layman might associate with the terms.

The more important methods are decision trees, perceptrons, neural networks, support vector machines, relevance vector machines and Gaussian processes. Most of these also fall under the general area of kernel methods or kernel machines.

2.4.3 Decision trees

Decision trees arise from applying serial classifications, each of which refine the final outcome. That is, each decision subdivides the members of a set of samples to be classified into smaller subsets and the subsets associated with each leaf of the decision tree consist of a single class. The advantage of the decision tree approach is that each decision point depends on one or only a few features and there is potentially useful information available that is lost when the overall classification is treated as a black-box with features as input and a simple classification as output. The disadvantage is that the structure may not accurately model the behaviour of the system being modelled. Decision trees are covered in chapter 9 of Alpaydin [2].

In the work on heuristic selection described in the main body of this dissertation, a decision tree approach could be taken to combine the individual classifiers for each heuristic. To do so an ordering on the heuristics would need to be imposed, the obvious one being the numeric order. The decision tree could start with a classifier that splits the samples into two classes, one for which the conjectures are deemed too difficult to prove and the other for which a heuristic will find a proof in reasonable time (this classifier is denoted as the heuristic 0 classifier in the current work). The second branching point of the decision tree would then split the class of conjectures that can be proved into two further classes, the first for which heuristic 1 is the best heuristic to use and the second for which some other heuristic is best. The next branching point would use another classifier to split the latter class into two more classes, the first for which heuristic 2 is the best heuristic and the second for which another heuristic should be used. This process is then repeated down to a final split between heuristics 4 and 5 (there being 5 heuristics in all)[2]. Abe [1] discusses such a decision tree approach along with other options.

The decision tree approach was considered but not adopted in the work described in this dissertation. The two main arguments against it were that each classifier after the first would be trained on training sets which had been determined by an earlier, imperfect, classifier and the size of such training sets would get progressively smaller. Secondly the ordering of the heuristics would make some classifiers more important than others and each classifier would have to stand on its own, no advantage could be taken of comparative measures between classifiers (an earlier heuristic classifier might hijack a sample on the basis of a weakly positive result when a later heuristic classifier would provide a strongly positive result).

2.4.4 Linearly separable classes

A starting point for the perceptron algorithm and what are known as hard margin support vector machines is the simple case of classification into two classes which can be determined by splitting feature space into two sections using a hyperplane. For any sample the class can be determined by measuring features and determining which side of the hyperplane the sample is placed. If it is possible to position a hyperplane such that no sample ever appears on the wrong side of it (and is thus misclassified), then the data is termed *linearly separable*. It is easiest to envisage this in two dimensions where the hyperplane is simply a straight line. The two classes could have any pair of labels but it is convenient mathematically to label one class with +1 and the other with -1. Figure 2.1 shows an example set of points in a two-dimensional feature space.

The vector equation for a hyperplane may be expressed

$$\boldsymbol{n} \cdot \boldsymbol{x} \ = \ constant$$

where $\boldsymbol{n}$ is a vector normal to the hyperplane and $\boldsymbol{x}$ is a general vector from the origin to a point in the hyperplane. In the context of machine learning and classification the normal vector is considered as a vector of weights applied to each feature value within the sample vector $\boldsymbol{x}$ and labelled $\boldsymbol{w}$ rather than $\boldsymbol{n}$, the constant is referred to as a bias and labelled b with the sign selected to give the hyperplane the equation

$$\boldsymbol{w} \cdot \boldsymbol{x} \ + \ b \ = \ 0$$

It is then very simple to determine if a given sample point, $\boldsymbol{x_i}$ is below the hyperplane

$$\boldsymbol{w} \cdot \boldsymbol{x_i} \ + \ b \ < \ 0$$

or above the hyperplane

$$\boldsymbol{w} \cdot \boldsymbol{x_i} \ + \ b \ > \ 0$$

hence the equation

$$\boldsymbol{w} \cdot \boldsymbol{x} \ + \ b$$

is known as the *Discriminant*.

[2]The decision "tree" described in this example is actually a decision list as there is only a single branch leading onto further decisions at each decision point, but the general comments still apply.

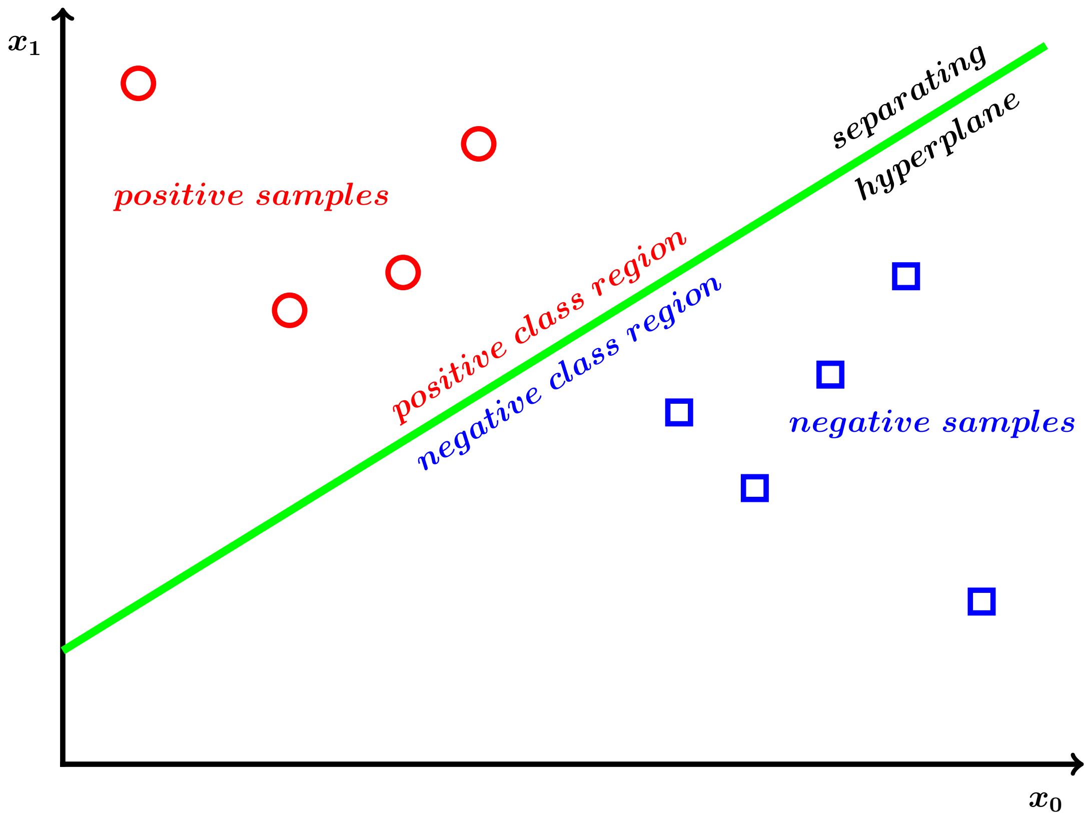

Figure 2.1: A classification problem in 2D feature space

The hyperplane defined by $(\boldsymbol{w}, b)$ is the classifier which is to be determined by machine learning. Note that it is the direction of $\boldsymbol{w}$ that is important, not the magnitude. The magnitude can be scaled by adjusting the bias b.

For a given training set of samples, if they are linearly separable, the positive samples must be in a different region of feature space to that occupied by the negative samples. The training samples are used to estimate the location of the two regions which are to be separated by the hyperplane. To aid intuition, make the assumption that the sample points are randomly but fairly uniformly distributed throughout the appropriate regions of feature space. One approach to determining a suitable $\boldsymbol{w}$ is then to estimate central points within the positive and negative class regions and set $\boldsymbol{w}$ to be the vector connecting the two points (see figure 2.2).

The two points, $\boldsymbol{x_c^+}$ and $\boldsymbol{x_c^-}$, are analogous to centres of mass of particles, with the mass associated with a sample weight. Since the symbol w_i is already associated with the i^{th} coefficient of the normal vector $\boldsymbol{w}$, the weights associated with each sample point $\boldsymbol{x_i}$ are given the symbol α_i. So the expressions for $\boldsymbol{x_c^+}$ and $\boldsymbol{x_c^-}$ are

$$\begin{aligned} \boldsymbol{x_c^+} &= \sum_{i=1}^{n^+} \alpha_i^+ \boldsymbol{x_i^+} \\ \boldsymbol{x_c^-} &= \sum_{i=1}^{n^-} \alpha_i^- \boldsymbol{x_i^-} \end{aligned}$$

where the $+$ and $-$ superscripts are class labels. Simple vector algebra gives

$$\boldsymbol{w} = \boldsymbol{x_c^+} - \boldsymbol{x_c^-}$$

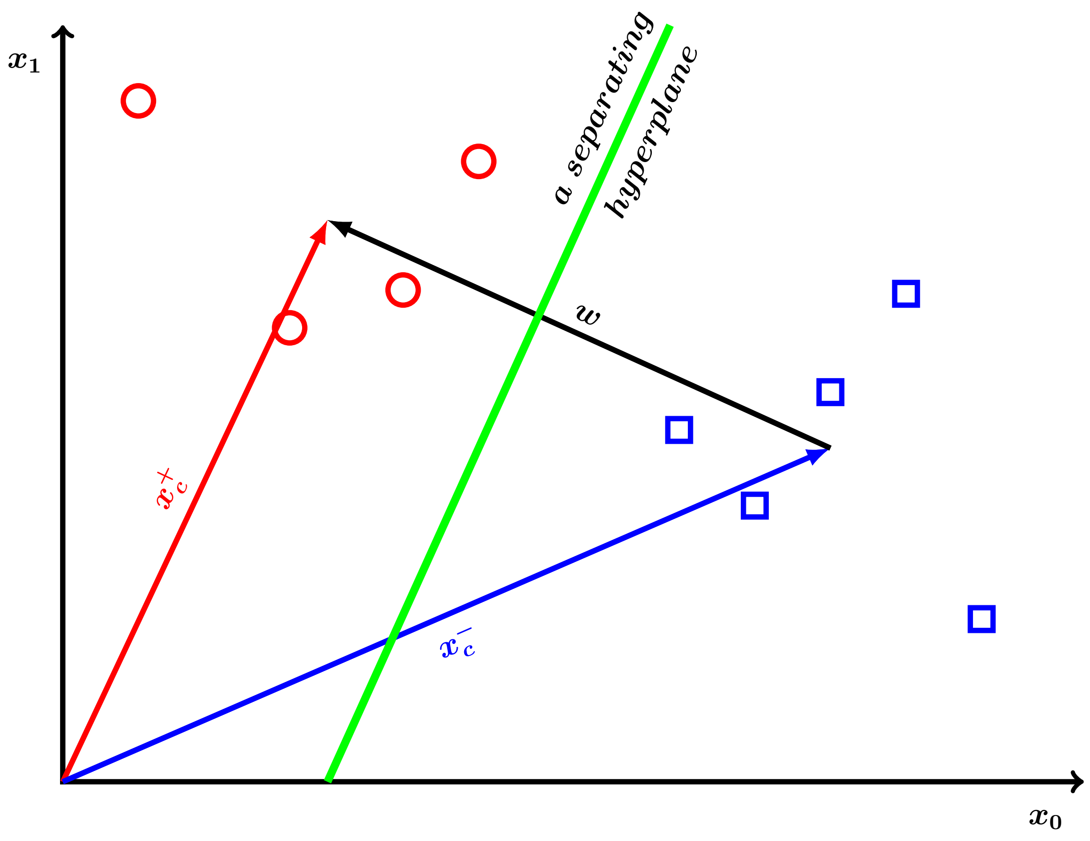

Figure 2.2: Defining the hyperplane normal in terms of weighted sample vectors

The y_i values or class labels associated with each sample point $\boldsymbol{x_i}$, as already noted, are +1 or -1 and by incorporating these into the above equations the need to label points is removed giving a simple final expression for $\boldsymbol{w}$

$$\boldsymbol{w} = \sum_{i=1}^{n} y_i \alpha_i \boldsymbol{x_i}$$

where the summation is over all samples and with a suitable renumbering/relabelling process being applied to the sample points and weights. The learning problem of finding $\boldsymbol{w}$ has been re-expressed as an associated problem of finding the values of the weights α_i. This latter form is known as the *dual form* and is very important for more sophisticated classifiers such as support vector machines. The standard approach of finding $\boldsymbol{w}$ or its coefficients w_i directly is referred to as the *primal form*.

It should also be noted that which ever approach is taken to finding $\boldsymbol{w}$, the bias b must also be determined as part of the same learning process.

The above discussion is not rigourous and it is fairly easy to think of examples where the sample points in the two classes are clustered in small subregions and so a naive joining of centres would result in an erroneous direction for $\boldsymbol{w}$. The following shows that the final equation

$$\boldsymbol{w} = \sum_{i=1}^{n} y_i \alpha_i \boldsymbol{x_i}$$

will hold for some set of α_i values, provided that there are as many samples as there are feature dimensions and these are not linearly dependant. Take a vector set of as many samples as there are feature dimensions and add the vector $\boldsymbol{w}$ to the resultant set of vectors. The new set of vectors must be linearly dependant. Therefore any one of the vectors, *i.e.* $\boldsymbol{w}$ must be expressible in terms of a sum of the other vectors, hence

$$\boldsymbol{w} = \sum_{i=1}^{n} y_i \alpha_i \boldsymbol{x_i}$$

must hold for some set of finite values α_i. If the samples are linearly dependant and do not span the space then it may be that the ideal $\boldsymbol{w}$ cannot be expressed in this way. For linearly separable samples there will always be some separating hyperplane the normal of which can be expressed

$$\boldsymbol{w'} = \sum_{i=1}^{n} y_i \alpha_i \boldsymbol{x_i}$$

but the hyperplane may not separate new samples.

2.4.5 Perceptrons

In 1958 Frank Rosenblatt [76] proposed an iterative algorithm for learning linear classifications. He was interested in modelling how the human brain may learn via visual examples, hence the iterative approach by which $\boldsymbol{w}$ and b are updated with each new training sample. The algorithm follows on simply from the above analysis, though his original paper is long with detailed arguments justifying the approach taken. The resultant classifier is named a *perceptron* as it arises from learning from perceptions.

In the perceptron algorithm most weight is given to points nearer the optimal position of the hyperplane. Such points will be the first to be misclassified if the hyperplane is shifted from its optimal position. The algorithm, as far as determining $\boldsymbol{w}$ is concerned, is to simply increase the weight associated with any sample point that is misclassified. In the full algorithm the weight is increased by a learning factor of less than one but the principle is unaffected if this is set to one in the following analysis. As a further simplification only a positive sample will be considered, the negative case follows very simply in an analogous manner.

After k misclassifications and associated adjustments to $\boldsymbol{w}$ and to b the estimates for their values are $\boldsymbol{w_k}$ and b_k. The $k+1^{th}$ misclassified point is $\boldsymbol{x_i}$ which is a positive point misclassified as negative so that

$$\boldsymbol{w_k} \cdot \boldsymbol{x_i} \ + \ b \ < \ 0$$

Increasing the weight of $\boldsymbol{x_i}$ by 1 gives the new estimate of $\boldsymbol{w}$ as

$$\boldsymbol{w_{k+1}} \ = \ \boldsymbol{w_k} + \boldsymbol{x_i}$$

even without making any change to b it can be seen that the discriminant, for $\boldsymbol{x_i}$, is more positive from

$$\boldsymbol{w_{k+1}} \cdot \boldsymbol{x_i} \ + \ b = (\boldsymbol{w_k} + \boldsymbol{x_i}) \cdot \boldsymbol{x_i} \ + \ b = \boldsymbol{w_k} \cdot \boldsymbol{x_i} \ + \ b \ + \ \boldsymbol{x_i} \cdot \boldsymbol{x_i}$$

Though the new $\boldsymbol{w_{k+1}}$ is an improvement for the point $\boldsymbol{x_i}$ it is not necessarily so for another positive point $\boldsymbol{x_j}$, say. Here the change in the discriminant is $\boldsymbol{x_i} \cdot \boldsymbol{x_j}$ which may be negative. But the overall change can be ensured to be positive if a sufficiently large positive change is made in b. This is done by setting

$$b_{k+1} \ = \ b_k \ + \ R^2$$

where

$$R = max_{1 \leq i \leq n} \| \boldsymbol{x_i} \|$$

Each misclassified point will thus shift the hyperplane in the right direction. It can also be seen that the weights will increase with each point that is repeatedly misclassified (it is assumed that sample points are repeatedly available or repeated copies of them are in the training set), so the change represented by a unit increase will decrease in relative magnitude as the algorithm proceeds. It is therefore reasonable to surmise that the process will iterate towards a stable solution where the training set is linearly separable. This is indeed the case, see Christianini and Shawe-Taylor [59] for a proof originally produced by Novikoff [64].

2.4.6 Margin

The product of the discriminant and the classification value y_i (which is plus or minus 1), gives a positive number for correctly classified points that is a measure of the distance from the point to the dividing hyperplane. This is known as the *margin.* (For the margin to be a direct measure the weight vector $\boldsymbol{w}$ should be normalised to 1, *i.e.* $\| \boldsymbol{w} \| = 1$.) The margin is generally given the symbol γ.

$$\gamma_i \ = \ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} \ + \ b \ )$$

The margin for a single sample, as defined above, is referred to as a *functional margin* (Shawe-Taylor [59]). In the work described in this dissertation, where a single sample is being referred to, the functional margin is referred to simply as the margin. In particular, this is the measure used to compare classification results for the different heuristic classifiers.

In the training phase in which the position of the hyperplane is being determined, it is the minimum value of the margin over all training samples that is of importance, (*i.e.* the amount of no-man's-land on either side of the border). This is the *functional margin of the hyperplane with respect to the training set.* The maximum value of this (minimum) margin over all possible hyperplanes is the *functional margin of the training set*.

In the remainder of this document all types of margin will be referred to simply by the term margin unless the context is not sufficiently specific to remove ambiguities.

The margin is of particular importance in classifiers using the support vector machine which is discussed in a following section.

2.4.7 Transforming the feature space

The simple perceptron algorithm relies on the data being linearly separable for the process to terminate (if no hyperplane is able to separate the two classes within the training set then there will always be points that are on the wrong side).

x_1	x_2	$\phi_1(\boldsymbol{x}) = x_1$	$\phi_2(\boldsymbol{x}) = x_2$	$\phi_3(\boldsymbol{x}) = x_1 \cdot x_2$	$\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}) + b$	$x_1\ EOR\ x_2$
0	0	0	0	0	-1	0
0	1	0	1	0	+1	1
1	0	1	0	0	+1	1
1	1	1	1	1	-1	0

Table 2.1: Discriminant for the example EOR function.

Where data is not intrinsically linearly separable it is possible that it may be made so by transforming the feature space, *i.e.* by creating new basis functions each of which is a function of one or more of the originally measured features. This process may change the dimension of the problem.

That is, a point in feature space defined by the vector $\boldsymbol{x_i}$ is transformed to a new point in transformed space defined by the vector $\boldsymbol{\Phi}(\boldsymbol{x_i})$ which has vector components $(..., \phi_j(\boldsymbol{x_i}), ...)$ each of which is a function of the original feature vector $\boldsymbol{x_i}$. Note that the dimensions of the transformed space may be different from that of the feature space, either greater or smaller.

A simple example of a data set that is not linearly separable, but can be made so by transformation, is one where the classification is given by the exclusive OR function of the two binary feature values of the samples. In this simple case, the two binary features can be transformed to a three-dimensional space with an additional feature given by their product. It is then simple to separate the classes in the transformed space using a hyperplane. Table 2.1 shows how a discriminant with $w_1 = 2$, $w_2 = 2$, $w_3 = -4$ and $b = -1$ provides a correct classification. (Note that $\boldsymbol{w}$ has not been normalised to 1 in this example to keep the numbers as integers.)

2.4.8 Kernel functions arising from transformed space

Having transformed the feature space, the same approach and expressions for hyperplanes can be used but with the vector $\boldsymbol{x}$ replaced by the transformed vector $\boldsymbol{\Phi}(\boldsymbol{x})$. So the margin for a sample point $\boldsymbol{x}$ with associated class y is

$$\gamma = y(\boldsymbol{w} \cdot \boldsymbol{\Phi}(\boldsymbol{x}) + b)$$

where

$$\boldsymbol{w} = \sum_{i=1}^{n} y_i \alpha_i \boldsymbol{\Phi}(\boldsymbol{x_i})$$

combining the two gives

$$\gamma = y(\sum_{i=1}^{n} y_i \alpha_i \boldsymbol{\Phi}(\boldsymbol{x_i}) \cdot \boldsymbol{\Phi}(\boldsymbol{x}) + b)$$

The margin γ now represents a distance in transformed space rather than the original feature space.

A key point to note in the above expressions is that the transformed vector $\boldsymbol{\Phi}(\boldsymbol{x})$, for any point in feature space $\boldsymbol{x}$, only appears as part of a scaler product with another transformed vector $\boldsymbol{\Phi}(\boldsymbol{x'})$, *i.e.*

$$\boldsymbol{\Phi}(\boldsymbol{x}) \cdot \boldsymbol{\Phi}(\boldsymbol{x'})$$

Regarding the scaler product of the transformed vectors as a function of the original feature vectors gives a *Kernel function*

$$K(\boldsymbol{x}, \boldsymbol{x'}) \ = \ \boldsymbol{\Phi}(\boldsymbol{x}) \cdot \boldsymbol{\Phi}(\boldsymbol{x'})$$

In terms of this kernel function the margin becomes

$$\gamma \ = \ y(\sum_{i=1}^{n} y_i \alpha_i K(\boldsymbol{x_i}, \boldsymbol{x}) \ + \ b \ )$$

The fact that the feature space transformation is only an intermediate step gives a great deal of flexibility. For example, the dimension of the transformed space could be infinite as long as the scaler product is well defined. It also means that it is possible to work directly in terms of a kernel function without ever defining the corresponding feature transformation. To do this requires that the kernel function conforms to necessary conditions arising from it being the scaler product of some feature space transformation[3]. The necessary and sufficient conditions[4] were first set out by Mercer [54]. Cristianini and Shawe-Taylor [59] give a detailed discussion of Mercer's theorem and include a proof for the simplified case of a finite feature space. (In a finite feature space a symmetric matrix of all possible $K(\boldsymbol{x_i}, \boldsymbol{x_j})$ values can be defined and shown to be positive definite with positive eigenvalues if, and only if, $K(\boldsymbol{x_i}, \boldsymbol{x_j})$ is a valid kernel function.)

In addition to defining necessary conditions for a function to be a kernel function, it is possible to determine functional transformations for which the Mercer conditions are invariant. That is, applying such transformations to valid kernel functions will always result in new functions that are also valid kernel functions without the need to check the Mercer conditions afresh. Thus new kernel functions can be developed from existing functions. For instance if $K_1(\boldsymbol{x}, \boldsymbol{y})$ is a kernel function then so is

$$K_2(\boldsymbol{x}, \boldsymbol{y}) = K_1(\boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{y}))$$

Cristianni and Shawe-Taylor [59] give a list in their book. By means of such transformations, new kernel functions may be derived without having to demonstrate compliance with Mercer's theorem in every case.

2.4.9 The support vector machine

To acquire the maximum accuracy in the learning process, given noisy data, as large a training set as possible should be used. One trade-off is that the time taken during

[3]To be more precise, for a function to be a kernel function there must exist some feature space transformation that may be applied to any pair of arguments of the kernel function to yield two transformed vectors whose scaler product is equal to the value of the kernel function.

[4]According to Cristianini and Shawe-Taylor [59], Mercer's theorem gives necessary and sufficient conditions for a continuous symmetric function $K(\boldsymbol{x}, \boldsymbol{z})$ to admit a representation

$$K(\boldsymbol{x}, \boldsymbol{z}) \ = \ \sum_{i=1}^{\infty} \lambda_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{z})$$

with non-negative λ_i

learning process is increased. In many circumstances this doesn't matter, as the critical time is that taken to classify new points. (In the context of the present work, being able to determine the heuristic to be used for a new problem needs to be done quickly, as this is part of the solution time, but if the learning process takes several days of computer time it doesn't matter as it is only done once.)

But for the simple perceptron approach there is a second drawback to a large training set. The time taken to do the classifying is also increased, as all elements of the training set are involved. Furthermore, many elements of the training set may be a long way from the hyperplane and it can be argued[5], provide no useful information. In Rosenblatt's algorithm, these elements may be associated with zero coefficients and thus ignored but this is not an intrinsic part of the algorithm and is also dependent on the order in which the points are examined.

The *support vector machine* (SVM) takes advantage of sparsity in a more systematic way. The set of vectors is restricted to those nearest the hyperplane and which define the location of the hyperplane. These are called the support vectors.

It is possible to fit an SVM to training data that is not linearly separable but initially, the linearly separable case will be considered. Where the data is linearly separable the functional margin of the training set is well defined and the resultant SVM is referred to as a *hard margin support vector machine*. The hard margin SVM is a *maximal margin classifier*, that is the hyperplane is selected to give the maximum possible (minimum) margin over all training samples.

One way of viewing the margin γ is as adding a thickness to the dividing hyperplane. The standard equation for the hyperplane defines its centre and its thickness is given by 2γ (a margin of γ either side of the central hyperplane). For any given direction of the hyperplane (as defined by the normal vector $\boldsymbol{w}$) the thickness can be increased until it hits sample points on both sides (*i.e.* sample points in the two classes closest to the hyperplane). Note that the constraining sample points are on both sides because if it was only constrained on one side then the thickness could be increased by moving the centre, *i.e.* the hyperplane.

Maximising the margin comes down to minimising the value of the square of the norm of $\boldsymbol{w}$ *i.e.*

$$\| \boldsymbol{w} \|^2 = \boldsymbol{w} \cdot \boldsymbol{w}$$

where $\boldsymbol{w}$ and b are scaled such that

$$\gamma_{min} \| \boldsymbol{w} \| = 1$$

and applying the constraints that no sample point has margin less than γ_{min}.

Expressed in this way the problem is a well behaved quadratic programming problem with a unique minimum (*i.e.* there is a unique minimum value of $\| \boldsymbol{w} \|$ corresponding to a

[5]The training samples define a region of feature space, or transformed feature space, and it is the boundary of this area which is important for determining the classification of new samples. If all the training points are accurate then the points nearest the boundary are the important ones. On the other hand, if the points are noisy the argument could be made that it is the centre of the spatial region that can be most accurately determined from the training samples and a classification decision should be based on which class centre a new sample is nearest. The perceptron algorithm and SVMs are predicated on determining spatial class boundaries but with some allowance, in the latter case, for noisy data in the guise of *slack variables* which are described in the main text.

unique maximum value of γ_{min}). Note that there may be many values of $\boldsymbol{w}$ corresponding to the unique value of $\parallel \boldsymbol{w} \parallel$. See Abe [1].

The constraining sample points for which the margin $\gamma = \gamma_{min}$ are known as *support vectors*. The other sample points within the training set can be discarded and $\boldsymbol{w}$ defined as a weighted sum of the support vectors alone. (In the process of fitting the hyperplane using Lagrange multipliers to impose the margin condition, only the support vectors have non-zero Lagrange multipliers or weights.)

2.4.10 Nonseparable data and soft margin classifiers

Even with transformation of the feature space some data sets may not be linearly separable. Additionally, measured data is likely to be noisy and some sample points may be erroneous. A modification of the previously described hard margin support vector machine provides a robust solution to such circumstances. The so called *soft margin* approach allows for a predetermined number of the sample points to lie within the minimum margin or even on the wrong side of the hyperplane, *i.e.* be misclassified. The points which are permitted to be less than the minimum margin away from the hyperplane are associated with *slack variables* which measure the degree to which the margin has been breached. Formally, the previous condition (where scaling has been applied to make the minimum margin 1)

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1$$

is relaxed to

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq (1 - \xi_i)$$

(see Abe[1]), the non-negative values ξ_i are the slack variables. Ideally the *number of points* with non-zero values for the slack variables should be minimised. This is a combinatorial problem and as such is not conducive to efficient numerical solution. To avoid the combinatorial problem, all points are assigned slack variables but the norm of the slack variable vector is minimised. That is the previous minimisation of

$$\parallel \boldsymbol{w} \parallel^2$$

is extended to the minimisation of

$$\parallel \boldsymbol{w} \parallel^2 + C \sum_{i=1}^{n} \xi_i^p$$

The parameter p is 1 for a 1-norm and 2 for a 2-norm, the parameter C represents a trade-off between training error and margin. In the SVM implementation used in the work described in this dissertation, SVMLight [36], the parameter C is split into two to allow different weights for positive and negative samples (to allow for unbalanced sets). The target function to be minimised is

$$\frac{1}{2} \parallel \boldsymbol{w} \parallel^2 + C_+ \sum_{i:y_i=1} \xi_i + C_- \sum_{j:y_j=-1} \xi_j$$

see Morik *et al* [56]. The SVMLight parameter j is used to set the ratio of C_+ to C_- and typically should be equal to the ratio of the number of negative samples to the number of positive samples in the training set *i.e.*

$$j = \frac{C_+}{C_-} = \frac{\text{number of negative training examples}}{\text{number of positive training examples}}.$$

The default value of j is 1. The magnitude of the parameter[6] C may be user set but defaults to the average value of

$$(\boldsymbol{x_i} \cdot \boldsymbol{x_i})^{-1}$$

As with the hard margin case, the method of Lagrange multipliers is used to integrate the constraints into the optimisation problem. The requirement for the slack variables to be positive is not necessary as negative values, corresponding to points with margins greater than the minimum, will have zero values for the corresponding Lagrange multipliers.

2.4.11 Alternatives to SVMs

SVMs were selected as the machine learning method for the work described in this dissertation for a number of reasons. SVMs are a proven technique involving a well defined optimisation problem without local minima. SVMs are well supported with existing software, the package selected for this work being SVMLight [36]. SVMs are efficient as they utilise a sparse approach, the training set is effectively reduced to the support vectors once the learning process is complete.

For completeness some other methods of machine learning are described in the following subsections.

Neural networks

The nodes of a neural network are a very similar structure to that of a perceptron. In fact neural networks are referred to as multi-layer perceptrons, but this is not a strictly accurate description as the perceptron is a step function (i.e. a classifier) whilst the nodes within a neural network use a smooth differentiable function, see Bishop [7]. The inputs of each neural network node are weighted and summed before a nonlinear threshold function is applied to give an output. The output in turn may become one of the inputs of a further node. In the early days of AI research the view was put forward that this was a simple model of the way neurons may work in the brain. The more modern view is a statistical one.

Neural networks provide a compact model once they have been optimised, but the optimisation process may not be well behaved, as the error function is not a convex function. In contrast, the support vector machine involves a convex optimisation problem to determine the model parameters (Bishop [7]). For more on neural networks see for example Ripley [71] or Bishop [7].

The relevance vector machine

The relevance vector machine is similar to the support vector machine, but has the advantage that the importance of each vector is determined as an intrinsic part of the process and doesn't need to be separately determined, as in the support vector machine. The

[6]The SVMLight user notes simply refer to C rather than C_- or C_+ but in the current work the default value was used and only j varied. See Appendix C for the effects of varying C from the default.

relevance vector machine produces a probability distribution as an output rather than a direct classification but it is straight forward to set a threshold for classification.

The drawback of the relevance vector machine is that the model parameter fitting problem is not a simple convex optimisation. See Bishop [7] for details.

Gaussian processes

Gaussian processes, Rasmussen and Williams [69], provide a more fundamental approach to the machine learning process and can be shown to be related to neural networks, support vector machines and other methods.

Gaussian processes work with a distribution of random functions directly, rather than defining basis functions with parameters and working with distributions over the parameters of the functions. The key to making the process of dealing with functions rather than point variables tractable is to consider the functions at a finite number of sample points. The values of the functions within the distribution at these sample points are related to each other through a Gaussian covariance matrix.

Gaussian processes are fully defined by second order statistics, *i.e.* the covariance and the mean. The mean is often taken to be zero, so the process is defined by the covariance matrix (this is the case when there is no prior information as to a value to set the mean to). Expressing the covariance as an expectation of the dot product of the function vectors leads naturally to a kernel function, see Bishop [7]. As with kernel machines (such as SVMs), the kernel function can either be derived from basis functions or may be selected directly. See Rasmussen and Williams [69] for details.

2.4.12 Feature selection

Kernel methods involve the scaler product of transformed feature spaces. The dimension of the transformed vectors may be different from the dimension of the feature space (the number of features). Additionally, the transformed feature space vectors are not independently calculated, the kernel function calculates the value of the scaler product of two transformed feature space vectors directly from the original untransformed feature space vectors. *It is thus possible to do machine learning using many features to gather as much information as possible without this implying the need for an infeasible number of measured samples.*

Despite this there are two reasons for reducing the set of features to only those that make a useful contribution. First, more features slow both the learning and generalisation processes, and second, it is useful and interesting to discover which features are pertinent to determining the complexity of the proof problems and the best heuristic to use (in the context of the work described in this dissertation).

The process of determining which features to retain out of a larger initial set is referred to as *feature selection.* The process is complicated by interaction between features, that is, features may be required in combination so individual features cannot be treated in isolation. To be absolutely confident that the final reduced set of features is the best possible it would be necessary to look at all subsets of the original set (all members of the powerset). For all but the smallest of feature sets this brute force approach is not feasible. (For the work described in this dissertation the number of features looked at was 53, which gives a power set of size 2^{53} or approximately 10^{16}.) It is therefore necessary to compromise and explore only a part of the potential search space.

Standard approaches

Three generic approaches to feature selection are commonly used. One method is to apply some criteria for filtering the features in a direct fashion (the filter approach). A second is to use some internal feedback from a machine learning procedure such as SVMs to determine which features to use (the embedded approach). A third approach is to run a loop in which a set of features is selected, a model fitted (using machine learning) and the model tested on a different data set to give a performance measure of some sort, this is the wrapper approach. In 2003 the Journal of Machine Learning Research ran a special issue on feature selection, including an introductory survey by Guyon and Elisseeff [31].

With these different approaches, the trade-off is between speed and accuracy of results. It should be noted that these methods have been developed in the context of possibly a very large number of features. In applications such as the text processing of internet documents, the number of variables or features involved may range from hundreds to tens of thousands. In the work that forms the basis of this dissertation, the number of features is 53, which allows the use of methods that would be impossibly slow in the case of thousands of features.

Of the three approaches, filtering is potentially the fastest. An example of a filtering method is that of feature ranking according to some criteria such as mutual information (based on probabilities) between individual features and the output variable (e.g. the class number). Guyon and Elisseeff [31] give a good overview and reference papers giving details. Feature ranking assumes an independence between features which may not be the case (pairs of features may act in concert whilst individually scoring low on the ranking criterion).

Both embedded and wrapper methods allow the consideration of subsets of features. The embedded approach involves modifying the learning approach so that feature selection is part of the model optimisation in the learning process. This may be efficient but is intrinsically more complicated than the wrapper approach, which leaves the core learning procedure unchanged. The embedded approach may have the drawback of needing to simplify the learning method to make the problem tractable. For example one method is to use a SVM to fit a linear model and then to remove features on the basis of the fitted weights. This has the advantage of being simple, but the use of a linear model may be inaccurate and lead to poor results. See Brank *et al.* [12].

Feature selection method used

The literature on feature selection as cited in the previous section is generally aimed at solving the problem of reducing a very large number of features to a manageable number. The number of features might be in the tens of thousands. In such circumstances some filtering may be required to reduce the feature set to a level that is small enough to perform machine learning. This is very different from the work described in this dissertation where the total feature set contains only 53 features. Filtering involves making a judgement on features before they are applied to the machine learning process, and given that machine learning is applied to problems that are too complex to be analysed directly, any such filtering will be imperfect and is best not done if it does not need to be done.

In the context of the work described in this dissertation, initial experimental work indicated that the linear kernel model does not work well (*i.e.* some feature space transformation should be used). This precludes the more straight forward of the embedded approaches as described in the previous section. Additionally the embedded approach partially negates the advantage of using an established software package for the fitting of the SVMs in that code modification is required. (Some code modification was already required in the theorem prover to measure the features, but this was unavoidable.)

The wrapper approach allows selection to be done on the basis of a complete machine learning cycle without having to make use of intermediate results the significance of which are difficult to determine. The disadvantage of the wrapper approach is the process is relatively slow but for a small number of features is feasible.

In the present work the number of features is small, for heuristic selection the machine learning process leads to separate classifiers which then need to be combined so it was determined that the wrapper approach was the best method to use.

Within the general wrapper approach different options on feature selection are available. The ideal would be to test every possible feature subset but this is not feasible. Instead a range of options was used from removing just a single feature from the set and replacing it, removing features successively based on an appropriate criterion and exploring all possible small subsets of features. All these are described in detail in the appropriate sections of this dissertation, in particular chapter 6.

2.5 Applying machine learning to theorem proving

The proof finding process involves a very large search space and there may be large differences in the efficacy of different heuristics used to find the proof. Unfortunately the best heuristic to use is problem dependent and the relationship between the problem and the best heuristic is not obvious even to human experts. This makes heuristic selection a good candidate for machine learning techniques. The approach taken in the work described in this dissertation is that of using machine learning to relate features to the selection of the best amongst a fixed choice of heuristics. Another approach, and one that has been tried by various researchers, is to modify the heuristic itself through machine learning.

In this section, a brief summary will be given of such previous work. A good survey paper which covers work up until 1999 is that of Denzinger, Fuchs, Goller and Schulz [20].

2.5.1 TEAMWORK and the E-theorem prover

The TEAMWORK project [19] took place in the latter half of the 1990s. The approach taken was a combination of parallel processing on a network of computers and machine learning. The computers within the network ran software programs referred to as agents (which is common parlance in the AI community). The agents were of four types referred to as “experts,” “specialists,” “referees” and a “supervisor”. The “experts” equated to copies of a theorem prover each of which needed to run on a separate computing node. The “specialists” equated to library modules performing tasks such as determining the similarity between the current problem and a library of previously solved problems. The “referees” provided a measure of the efficacy of each theorem proving approach to allow machine learning. The “supervisor” provided overall control of the process. The software was run for short fixed periods, at the end of which progress was assessed and changes made as needed.

The machine learning used *case based reasoning*, a nonparametric approach that makes use of stored previous solutions or cases. The problem (negated conjecture plus axioms) was compared with previous problems using a similarity function. The similarity function was based on signatures, which depended on the number of terms and the arity of functions within the terms. The solution of the new problem was based on the successful solution of the nearest stored example proof in an approach called “flexible re-enactment”. The aim of this approach was to learn from easier problems in a domain and use the results as a stepping stone to solving the more difficult problems. A similar idea was behind the Octopus theorem prover of Newborn and Wang [60]. In Octopus (which is a development of an earlier theorem prover named Theo) the learning process used is part of the solution process (i.e. it is done afresh for each theorem being proved rather than making use of data from previously proved theorems). The approach taken is to strengthen the conjecture clauses, (which corresponds to weakening part of the original conjecture), to produce a related but different set that is easier to prove . The proof for the modified conjecture is then used as a starting point for the more difficult original version. The process continues, until finally, the original theorem is proved. Modification of the clause takes the form of replacing a constant by a variable or a function by a variable or by deleting a literal. The modified clause subsumes the original clause Newborn and Wang [60]. An advantage of this approach is that by modifying different base clauses, several proof attempts on different modified conjectures can be performed in parallel, provided separate processors are available to do so.

Some success was reported for TEAMWORK but the method was restricted by a lack of training examples and was limited in scope. According to Schulz in his thesis [77] the method relied on an homogeneous collection of workstations and was very sensitive to small differences in performance of the machines. Newborn and Wang reported that Octopus had solved 42 previously unproved theorems [60] but the innovations of Octopus are concerned with problem modification and parallel processing rather than the application of machine learning.

Matthias Fuchs reported on similar work on instance based learning [24]. A small set of features was used and a nearest neighbour approach used as a similarity function.

Stephan Schulz built on some of the work in the TEAMWORK programme and built learning into the E theorem prover (note that this learning aspect of E is different from the use of E as a straightforward theorem prover following predetermined heuristics as was done in the work described in this dissertation). The role of the similarity function is taken by a technique called “term-space mapping” Schulz and Brandt [79]. The purpose of the machine learning is to help at decision points within the theorem proving process, in particular the selection of the next clause (the given clause). Schulz does consider a number of decision points in his thesis [77] but concludes that the selection of the given clause is the most critical.

2.5.2 Neural networks and folding architecture networks

Standard neural networks are used to learn a black-box function from a vector of real variables to a vector of real function values. The nature of the neural network constrains the input data to be in a form representable by a set of real numbers. Conjectures, axioms and other logical formulae are tree structures with terms often containing functions with arguments that are themselves terms. Folding Architecture Networks are a development of neural networks that work with tree structures.

Mareco and Paccanaro [47] applied machine learning using neural networks to improve automated theorem provers, but their work was mainly confined to term matching for rewrite systems and applied to simple problems in group theory. Though the work involved dealing with issues such as representing logical expressions in a suitable form for neural networks, the process of term matching is efficiently carried out in modern provers using indexing techniques.

Goller [28] has applied folding architecture networks to the learning of heuristic evaluation functions for the theorem prover SETHEO. In this context, a heuristic evaluation function is a measure of goodness of an inference step within a theorem proof, that is, once a proof is found the inference steps that were part of the proof are given positive values whilst those that are not part of the proof are given negative values. Goller's work was restricted to word problems within group theory, which Goller stated "are generally regarded as trivial" so though promising the results were not conclusive.

Blanchard *et al.* [9] extended Goller's work by applying it to the theorem prover Otter. Otter with the folding architecture addition was better than standard Otter but Blanchard *et al.* also found that the addition of a simple hash table to memorise previous patterns gave the best results, which implied that the folding architecture learning was giving improvement by memorising previous patterns rather than being able to generalise to new patterns.

Blanchard et al.'s results are in accord with results obtained by Meng and Paulson [53], who filtered clauses on the basis of those that had previously been used in a proof (Meng and Paulson's work is discussed more fully in the next section.)

2.5.3 Learning with symbols and large axiom libraries

The set of axioms that should be combined with the negation of a conjecture to find a proof may be part of a much larger set, most of which are not needed in the proof. In many circumstances it would be useful to automatically select useful axioms from a large database. This is a similar problem to that of clause selection during the proof search process in the given clause algorithm. Rather than the unprocessed clause set containing a large number of clauses generated from inferences applied to an initially small axiom set, the number of clauses is large from the start because it contains many axioms that are not relevant to the particular conjecture being tested.

Meng and Paulson [53] obtained useful improvements in the proof search by a simple filtering approach based on whether or not a clause (axiom) had been used in a previous proof. Meng and Paulson produced a set of relevant clauses by taking the union of the sets of clauses used in each of the set of proofs that they investigated. The simplest filtering technique is to remove all clauses that do not appear in the set, and this was found to give an improvement in the number of proofs found, but such an approach risks removing a clause that is needed for a particular proof. Meng and Paulson go on to discuss more sophisticated techniques based on symbol scores and measures to determine how close clauses are to members of the relevant clause set.

Urban's MaLARea system [88] applies machine learning to the pruning of irrelevant axioms from a large database to enable more proofs to be found. The axioms need to follow a consistent use of function names between problems. An initial run is performed on a set of conjectures, and proofs successfully found are used as the learning set for the machine learning phase. The result of the machine learning is a function that prunes axioms from the database, and then further proofs are sought with the reduced axiom set.

2.5.4 Proof planning (Omega project)

The various projects described in the previous sections, and the work covered by this dissertation are concerned with applying machine learning to automated theorem proving at a low level - heuristic control, heuristic selection, clause weakening, clause relevance and so on. There has also been work at a higher level. An example of this is the Omega project in which learning is applied to proof planning; that is, in choosing between different proof methods. See Jamnik, Kerber and Benzmuller [34].

2.6 Summary

To give proper context, the background presented in this chapter has covered a wider gamut of topics involved in theorem provers and machine learning than are directly involved in the work described in this dissertation. Given the background it is useful to summarise the choices taken in terms of the logic system, the theorem prover used and the type of machine learning selected.

The logic system (or language) used is first order logic with equality specifically treated as part of the language, rather than being added in terms of various axioms. First order logic with equality provides a system which is much more powerful than basic propositional logic whilst still being constrained enough to allow theorems to be proved in an automatic fashion without human intervention during the proof search.

The theorem prover used for the main body of work is an equational theorem prover, (E written by Schulz [78]), for which many heuristics have been tested and for which source code is openly available. The prover has done well in competitions and has proved useful as a tool by researchers outside the group within which it originated. (It was important for the work reported in this dissertation to use a prover that was of more than academic interest.)

Previous work on applying machine learning to theorem proving has concentrated on learning new heuristics, or modifying a previous heuristic based on one or a few previous examples of successful proofs (e.g. Shulz [77]). In some cases the previous examples are artificially generated from the problem itself by simplifying it (as in the Octopus prover [60]). Though such learning is built-in to the E theorem prover, it is not widely used. There are a number of drawbacks to the general approach of learning a new heuristic.

Training samples very similar to the conjecture to be proved are needed and these may not be available. It is problematic to define which stored examples are close - a distance function is needed and to some extent determining a good distance function is as difficult as determining a good heuristic, (apparently similar conjectures may require quite different proofs). The learning process needs to be applied to each new problem. The learned heuristic is likely to be less efficient than the best hand honed heuristic based on many sample problems.

For the work described in this dissertation a different approach was taken. Heuristics were predetermined and fixed and machine learning used to determine which to apply to a given conjecture proof problem. Such an approach allows machine learning to be done in advance. The resultant heuristic is a tried and tested heuristic. Such an approach is novel, though the overall function of heuristic selection from conjecture characteristics is built-in to the E theorem prover auto mode. The E auto mode does not use sophisticated machine learning though it does measure features of the conjecture and axioms. A few features are used to classify all problems into a few classes and for each class a given heuristic is used based on trials with previous examples. The E auto mode thus prejudges which features are important and is restricted to binary or ternary features so as to limit the number of classes. The work described in this dissertation uses machine learning and feature selection to learn which features are important and a functional relationship between feature values and the best heuristic to use. The features are not restricted to being binary or ternary valued. Additionally dynamic features as well as static features were used. That is some features, the dynamic ones, are measured a short way into the proof process.

The choice of using machine learning rather than a more analytic approach was determined by the fact that, though the choice of best heuristic is dependent on the conjecture to be proved, there is no obvious way of connecting the two even for human experts. Additionally, the availability of a large library of conjectures allowed the generation of many learning samples to use in training of a machine learning process.

The machine learning method selected, SVMs, was selected as it is an accepted state-of-the-art technique. Neural networks were not used as it was considered that SVMs provide a more systematic and efficient learning method. Decision trees were considered but the use of feature selection combined with other methods provides information that is as useful and less constricting in the context of the particular area of study.

As input to the machine learning, generic features were measured which made no assumptions as to the semantics of symbols used within conjectures. This complemented work done by other researchers who were concerned with symbols across many potential axioms (e.g. the work of Urban [88]). The use of generic features reduced their number which allowed more comprehensive feature selection methods to be used. Additionally it removed any arbitrary bias arising from inconsistencies in naming conventions for functions or variables.

The overall purpose of the work was to demonstrate that machine learning can successfully be applied to the selection of heuristics without the need of human expertise. This was successfully achieved. The work was restricted to a small number of heuristics to demonstrate proof of concept, but the same approach could be used with many more heuristics as a basis for an extension of the theorem prover as a practical tool.

Chapter 3

Methodology

This chapter covers the methodology of the experimental work undertaken. The results and analysis are covered in separate chapters for each experiment. Though there are differences between the experiments, there is also a large degree of commonality and it makes sense to collect reference information regarding heuristics and features in a single place.

To summarise the experimental work: an initial experiment, designed as a proof of concept, was carried out at an early stage of the project. Following promising results, a more extensive second experiment was performed. Beyond the second experiment, further analysis and experimental work was carried out to determine which measured features are significant and which are superfluous.

3.1 Generic description of experimental method

There is the potential for obfuscation in describing the experimental work, as it involves the application of software tools to other software tools, such that the output of a lower level tool is not a final outcome, but a single data point or sometimes only a contribution to a single piece of data. The experimental work involved analysing the results of applying a machine learning tool to a theorem prover, which is a tool for testing the validity of a logical conjecture. The experiments may be viewed on different levels.

At the lowest level a conjecture is read into the automated theorem prover, which is run with a particular heuristic selected. The three possible outcomes are, the conjecture is shown to be a theorem (it is proved), the conjecture is disproved or else a pre-set time limit is exceeded and the process stopped. At this level the input is a description of the conjecture (typically negated) as a text file in a defined format and the output is a yes, no or couldn't be proved either way answer. Additionally, the CPU time used in the process is stored for future use (as a measure of the efficacy of the heuristic used).

At the next level up - that of machine learning - the conjecture must be converted into a tuple or vector of numbers, *i.e.* features, which are considered as a single sample. Associated with each such sample must be a single output value, 1 or -1, which indicates in which of two classes it is placed. If the two classes are “solution found” and “solution not found” then the classification requires a single run of the theorem prover using a single heuristic. For more useful classifications such as “heuristic 3 is the best heuristic” and "heuristic 3 is not the best heuristic," the theorem prover must be run on the same conjecture once for each heuristic being looked at and the run times compared. The theorem prover runs are time-consuming but the result is a definite, correct, classification for the sample. The resultant data is used to train a classifier that is designed to calculate a classification directly from the feature vector without needing to run the theorem prover. The process is very much faster but the resultant classification will (in general) not be correct for every new sample. In fact it will be known to be incorrect for some of the pre-classified training samples, as to attempt to make it correct for all such samples leads to over-fitting and worse generalisation to new samples.

At a higher level, the classifiers that are produced as the result of the machine learning process can be combined to predict which out of the set of heuristics is the best for each conjecture. If all the classifiers agree then this is straightforward, *i.e.* the case when only one classifier says that its heuristic is the best (each heuristic being associated with a separate classifier). But, given uncertainty in the process, in many cases more than one classifier will place the sample in the positive class or possibly none of the classifiers will place the sample in the positive class. In these cases it is necessary to compare the degrees of certainty, or margin, for each classifier and select the one with the most positive (or least negative) margin. (See chapter 2 for a more precise definition of margin.)

Additionally it is useful, both for gaining insight and for streamlining the classifiers, to determine which of the measured features are pertinent and which are effectively irrelevant. As discussed in the background chapter there are different approaches to such feature selection, but the most straight forward one is the wrapper method which requires the machine learning process for the classifiers to be treated as a subroutine or function and the process to be run multiple times using different sets of features as input.

3.2 Data — conjectures to be proved

The first experiment, which was a feasibility study, used a single heuristic so in this case a wide variety of conjectures was not needed. The later experiment involved a comparison between different heuristics. Different heuristics generally work better on different types of conjectures so for a fair comparison a wide variety of conjectures needs to be included. The TPTP library [85] provides a useful central resource of conjectures from many different problem areas and these were used in this work. The TPTP library is a central repository for conjectures from many problem areas that are collected with the main purpose of aiding the development of theorem provers (TPTP stands for Thousands of Problems for Theorem Provers). Within the TPTP library the problems are collected together into different subject areas. Some of these are clearly delineated, such as conjectures in the area of group theory. Others are more arbitrary, as many conjectures arise from work that does not easily fit into the existing classifications. An example of the latter might be conjectures arising in the process of proving security protocols.

3.3 Measuring features

The aim of machine learning is to produce a function in software that reproduces the behaviour of an unknown function. The unknown function is partially known, in the sense that examples of outputs for some input states are known, but the underlying mechanism is too complicated to be understood, so that it is unknown how to predict outputs for new input states. Furthermore, even at the known data points, where output values are known, there is uncertainty as to what the input parameters are that determine the measured behaviour. That is, the input is in a known state but there is uncertainty in how to characterise that state in terms of a set of real numbers.

A key task in machine learning is thus to characterise the input state as a set of real numbers. Each real valued parameter or measurement of state is known as a "feature". In the particular case of the work described in this dissertation, the input consists of a set of clauses arising from a negated conjecture together with a collection of axioms. The clauses may contain functions as well as variables and constants. The function arguments may also be functions or variables or constants and so on. The logical meaning of the clauses is unaltered by a renaming (in a consistent manner) of any or all of these. For this reason, no meaning was attached to the names used, which is equivalent to considering each problem in isolation. To do so was a significant choice; other researchers (such as Urban [88]) have worked on considering individual proof problems within the context of a large number of potential axioms and have made use of historic proofs to assign importance to symbol names. One aim of the work described in this dissertation is the potential improvement of the theorem prover as a tool, in such a context the problems must be considered in generic terms to avoid tying the prover to a particular problem type or area.

Given a collection of clauses as a starting state, potential features can be characterised in three areas. First the size of clauses in terms of length (e.g. number of literals), depth (the degree that terms are nested within terms) and, more artificially, in terms of weight (which is a measure associated with the theorem proving method rather than intrinsic to the logical structure of the clause). The second area is that of clause type, for example the proportion of clauses that are Horn clauses (containing no more than one positive literal). Thirdly, measures can be made of connections between clauses such as a score based on shared term structures.

The collection of clauses existing during the proof search is known as the proof state, the collection being divided into separate sets as described in chapter 2. The starting proof state consists only of the initial conjecture and its axioms, but if the proof search is run for a short time there will be a much larger collection of clauses on which to measure features. Additionally, the presence of different clause sets allows the inclusion of features which compare properties between sets. This is discussed more fully in the next section.

3.4 Dynamic and static features

A novel aspect of the current work is the use of features measured on a snapshot of the proof state in addition to (or instead of) features measured on the conjecture prior to the proof search beginning. In the present work these are referred to as dynamic features and static features. (Similar use of dynamic features has been reported by Xu *et al.* in a learning approach to selecting algorithms in SAT solvers [100], though the author was not aware of this work when beginning the experiments described here, as it was in the field of SAT solvers rather than first order logic theorem provers and the paper had not yet been published. Beyond noting that Xu's work involves measuring some features after running the solver for a short time, more detailed comparison is not appropriate as SAT solvers and first order logic theorem provers are very different in nature.)

The input to the theorem prover is a negated conjecture which, together with a set of axioms, forms a set of clauses. Static features are measured on these clauses. Examples of such a feature would be the average clause length in terms of the number of literals or the proportion that are Horn clauses.

If the theorem prover is permitted to run for a short time (or for a fixed number of selected clauses, 100[1] in the case of the present work) then the proof state consists of several sets of clauses. There is a large set of unprocessed clauses which initially consisted of the original negated conjecture and axioms but has been increased by the addition of generated clauses. There is a smaller set of processed clauses which is internally saturated (all possible useful inferences involving clauses within the set have already been drawn). Additionally there is a temporary set of clauses that has been generated but may be deleted if simplifying inferences are found. Dynamic features can measure how these clause sets have changed from the initial clause set and also how measures for the processed and unprocessed sets compare. Details of the features used are given in appendix A.

An aspect of measuring dynamic features is the need to run the prover for a period of time first, albeit a short period. This raises the question as to which heuristic to follow on this initial pre-measurement phase. For the initial experiment a single heuristic was involved so this was the one applied. For the main experiment the situation was less straightforward as there were five heuristics under consideration. In this case the first heuristic was selected. Though the need to select a particular heuristic is not ideal - it may induce a bias in the feature values towards the heuristic used - out of the five heuristics the first is the best choice as it is the best heuristic in more cases than any other. Another possible alternative would have been to select a heuristic different from all five used in the experiment. This was not done as it introduces extra complication without solving the problem of a possible bias, any such new heuristic cannot be guaranteed to be equally different from all five test heuristics.

3.5 Theorem prover used

The theorem prover used was a modified version of the E theorem prover [78]. No modifications were made to the workings of the proof search engine or to the built-in heuristics. The changes made were to measure and write out features and to write classification results to an output file (together with feature values). The features included dynamic aspects of the process and so were measured after allowing the proof search to proceed for a fixed number (100) of clause selections (in the given clause process). The E theorem prover has a useful array of functions built-in that could be used for many of the features to be measured.

For the second phase of the work, further modifications were added to allow the simple selection of a fixed heuristic choice using a single command line flag and also to write out timing information as to the amount of time taken to find the proof. To make these

[1]Some experimentation was done with clause numbers up to 500 tried. 100 clauses was found to be a good compromise between allowing the prover to run and alter state and not setting the value so high as to become a significant fraction of the proof search process for most conjectures.

minor modifications involved analysing the code in some detail, as heuristics are not single entities. Additionally, the heuristic selection is done via references in tables, involving a level of indirection which required care to follow at the programming level.

3.6 Selecting the heuristics for the working set

Stephan Schulz, the author of the theorem prover E, has done much work on testing various heuristics on the problems contained in the TPTP library. The automated mode of E classifies problems according to the values of a few binary/ternary features, and the best heuristic for each such class was determined by Schulz experimentally. By taking the data from these experiments (which are available as part of the source code for E) it was possible to order the heuristics according to the number of the TPTP problems falling into the classes for which that heuristic was best. The working set of five heuristics for the work described in this dissertation was thus determined as the five most successful heuristics. For each of the heuristic descriptions, given in detail in Appendix B, the number of cases for which that heuristic is reported (by Stephan Schulz) as being best is given. It can be seen that Heuristic 1 is best in most cases with the other four heuristics being similar to each other. It would be preferable to have a set of heuristics to test that were all similar in applicability, but this was not possible.

The five heuristics in the working set used for the work described in this dissertation are simply labelled from 1 to 5. The labels used by E for the heuristics and more details of the options associated with each heuristic are given in Appendix B for reference.

3.6.1 Clause selection within heuristics

As described in the background chapter, the E theorem prover uses the given clause algorithm. A key part of the algorithm is the selection of the given clause from the set of unprocessed clauses. For the process of selecting the clause E uses a round robin of priority queues with different weighting schemes for each. It is primarily in the clause selection that the five heuristics differ. The individual weighting functions are described in the E manual provided with the software.

3.7 Fitting a support vector machine - SVMLight

As described in the background chapter, a support vector machine (SVM) is essentially a mathematical function coded in software that generates a real number (the margin), from a tuple of numbers. (The feature values may be integers or binary but are normally real valued). The core of the SVM is a kernel function, which takes a pair of tuples as input and produces a real value as output. The kernel function is applied multiple times, each time taking the same input tuple of feature values and pairing it with a different stored tuple of feature values taken from a learning sample (each application is with a tuple from a different learning sample). The results of the different applications are combined in a weighted sum. The particular set of learning samples whose tuples are used in the summation are referred to as the support vectors.

Before an SVM may be fitted, a kernel function must be selected. For the work described in this dissertation, a set of standard kernels were compared and SVMs fitted for each. The best kernel function was then determined by looking at the classification accuracy of SVMs from each. (This was done in the preliminary experiment and then the same kernel function was used throughout.) The process of selecting the best kernel function is complicated by the need to optimise the associated parameters. Each kernel function is more accurately described as a family of functions with the particular function determined by values assigned to parameters. The parameters must be entered by the user on a command line, so the optimisation process is essentially manual. (Some work was done on automating the process in the later stages of the work, but even with automation time constraints mean that the process is necessarily crude.)

The process of training a support vector machine involves finding a maximum of a quadratic objective function subject to linear constraints as outlined in the background chapter. Cristianini and Shawe-Taylor outline some implementation techniques in chapter 7 of their book [59]. The objective function involves a kernel function, and associated weights, rather than the original feature vectors. Rather than write new software to do the task, use was made of the program SVMLight [36].

The SVMLight software consists of two programs. The first program, svm_learn, fits the model parameters on the basis of a file of learning sample data and other user input such as a selected kernel function and the associated parameter values. The second program, svm_classify, uses the model to classify new samples, generating a margin value as output. SVMLight has four standard kernel functions as options: linear, polynomial, sigmoid tanh and radial. As part of the experimental work the different kernel functions were compared in the initial experiment.

3.8 Kernel functions

There are four "standard" kernel functions that are widely used, and that are provided with the program SVMLite that was used for the work described in this dissertation. Details of these kernel functions are given in the following sections. As part of the initial experiment the kernel functions were compared and one selected for use in the second, heuristic selection, experiment.

3.8.1 Linear basis function kernel

With the linear basis function kernel the support vector machine basically reduces to a linear perceptron. The feature space is not transformed (except for linear scaling). This is the simplest kernel and if it works then analysis of the results in terms of the effect of the various feature values is also straight forward but in general the model is too simple and it is unlikely that the learning sets will be linearly separable.

The general expression is just a simple scalar product of the two feature vectors without any further parameters,

$$K(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{x}.\boldsymbol{x'},$$

where $\boldsymbol{x}$ and $\boldsymbol{x'}$ are feature vectors.

3.8.2 Polynomial kernel

Provided Mercer's conditions are fulfilled by the transformations used, new kernels can be constructed from other kernel functions. The polynomial kernel is constructed from a simple linear kernel (scalar product). The general expression is as follows,

$$K(\boldsymbol{x}, \boldsymbol{x'}) = (s\boldsymbol{x}.\boldsymbol{x'} + c)^d,$$

where $\boldsymbol{x}$ and $\boldsymbol{x'}$ are feature vectors and s, c and d are user entered parameters.

3.8.3 Sigmoid tanh kernel

In the sigmoid tanh kernel function the tanh of a scaled and shifted vector product is taken. The sigmoid tanh kernel is a representation of the multi-layer perceptron with a single hidden layer, Gunn [30]. Burges notes that the hyperbolic tanh function only satisfies Mercer's conditions for some parameter values [13]. The general expression is,

$$K(\boldsymbol{x}, \boldsymbol{x'}) = tanh(s\boldsymbol{x}.\boldsymbol{x'} + c),$$

where $\boldsymbol{x}$ and $\boldsymbol{x'}$ are feature vectors, s and c are parameters.

3.8.4 Radial basis function kernel

Radial basis functions are Gaussians centered around focus points (one for each input point). Historically they were first used as a means of exact interpolation where the focus points are the input points to ensure that the interpolation passes exactly through the data points (see Bishop [7]). The general expression for the radial basis function kernel is,

$$K(\boldsymbol{x}, \boldsymbol{x'}) = e^{(-\gamma \cdot ||\boldsymbol{x} - \boldsymbol{x'}||^2)},$$

where $\boldsymbol{x}$ and $\boldsymbol{x'}$ are feature vectors and γ is a parameter.

One advantage of the radial basis function kernel is that it only has a single parameter. Additionally, this kernel gave the best results in the tests carried out (see chapter 4 on the initial experiment).

3.9 Custom software

In addition to the two major software packages used, the E theorem prover and SVMLight, software was written to perform specific experiments and to collate data and put it in the correct form needed. Software needed to be written to convert timings together with feature values to classifications, to split data into learning and test sets, to combine the output of the SVMLight software with known timings to determine how selection affects overall solution times, feature selection experiments and so on. These tasks are described more fully in the appropriate chapters on the individual experiments.

3.10 Overview of experimental work

A phased approach to the work was taken so as to determine if useful results were obtained on a small scale before committing to a large scale experiment.

The work was done in three phases. First a limited experiment was carried out as a proof of concept. The first experiment was limited to the application of the theorem prover only to problems within the SET domain of the TPTP library. Machine learning was only applied to a single classification into two classes, those conjectures which were proved and those which were not (within a time limit). The initial experiment was also used to determine the best kernel function to use when fitting a support vector machine using SVMLight.

The central part of the experimental work involved applying different heuristics to theorem proving on all problems within the TPTP library. Machine learning was applied to classification problems for each heuristic in turn and the results used to automatically select the best heuristic. The effectiveness of the automatic selection process was then assessed.

In a third phase the results of the main experiment were analysed to determine which measured features used in the machine learning made a significant contribution to the learning and classification processes. The purpose of this was twofold. Firstly, it is of interest to know what aspects of a problem determine how difficult it is to prove and the best heuristic to use. Secondly, from a practical point of view it is important to streamline models by reducing the number of features that need to be measured and used. It was also found that a reduced feature set gave better results than the full feature set; details are given in chapter 6.

3.11 Computer hardware used

For consistency all the experiments were run on the same hardware so direct comparisons could be made in terms of CPU time taken for different heuristics.

The computer used was a Linux workstation based around an Intel Core2 6600 CPU at 2.40 GHz. The processor has two cores. The total memory was just under 2 GB (1.9 GB). The hardware is not powerful by modern standards but it is relative rather than absolute performance that matters.

The later feature selection experiments were performed on a much more powerful dual Xeon workstation, but all timings relating to proof search are those obtained on the original hardware. (Having run all heuristics on all sample conjectures there was no need to collect further timing data.)

3.12 Summary

This chapter has described the features measured, the details of the heuristics used as a working set in the heuristic selection experiment, the kernel functions used with the support vector machine approach to machine learning and an overall description of the experiments carried out. More detailed descriptions of the experiments are included in separate chapters on each, including the results obtained.

Chapter 4

Initial experiment

It is incontrovertible to state that some conjectures are easier to prove, or disprove, than others. Determining which proof problems will be difficult for an automated theorem prover to prove without attempting the full proof in each case is not straightforward (or foolproof). Also, different search heuristics work better on different problems so that the best heuristic to use depends on the conjecture and axiom set. There isn't a universally best heuristic. (Such a universally optimum heuristic may be found in future, but to date no such heuristic has been published.) But, in a similar manner to the difficulty in determining which proofs will be difficult, it is not straightforward to determine what the best heuristic will be other than by looking at a history of similar problems in the manner of a human expert.

Both aspects of the proof search problem can be used to classify conjectures, either into classes of difficulty or into classes of problems best suited to a particular heuristic, where each class corresponds to a different best heuristic out of a set. The process of classifying any problem according to either of these schemes can be done in a simple, but slow, manner by the application of the automated theorem prover to the problem. The aim of the work described in this dissertation is to find a more direct route, via a function or relation, from easily measured features of the problem to the correct classification.

A premise of the thesis expounded in this dissertation is that there is a functional relationship between easily measurable features of a proof problem and the classification of the problem as defined above, and that such a relationship may be approximated by a machine-learned function. Though such a premise is reasonable, it is not necessarily correct. It may be that factors that determine how quickly a proof is found depend on very complex interactions between clauses, in a way that is too subtle to be captured by relatively crude measures such as the features described in appendix A of this dissertation.

Before investing substantial time in research, an initial experiment was undertaken as a proof of concept. The purpose of the experiment was to determine if there was a basis for assuming that machine learning could successfully relate measurable aspects of a problem to the classification of the problem. Of the two classifications, the simplest is the binary classification according to whether or not the conjecture can be proved as a theorem, by an automatic theorem prover within a reasonable time span. In addition to indicating the potential usefulness of the machine learning approach, such information is useful in its own right. There are circumstances where the user has a large number of conjectures to prove and requires only to prove a proportion of them within the available time. Being able to discard the conjectures that will not be proved, without the time consuming process of proof searching, will increase the overall number of conjectures that are proved.

4.1 Classification problem

Thus, for the initial experiment, machine learning was applied to finding a simple classification function. The classification problem that the function solves is to place conjectures into one or other of two classes. Membership of the positive class implies that the member is a theorem that is proved by the theorem prover within a preset CPU time limit. Membership of the negative class implies that the proof search did not terminate within the given time limit. In theory a conjecture may be disproved - that is proved to be invalid and not a theorem - by a saturated state being reached by the theorem prover. Given the large, and increasing size of the unprocessed clause set, in practice saturation is not reached except for pathological cases where the process never gets started due to insufficient or erroneous axioms. At the time of the experiment such cases were not specifically checked for however. Since the experiment was conducted the log files have been checked and it was found that out of the more than twelve hundred conjectures used, four led to the theorem prover saturating and in each case the result was obtained before any clauses were generated by inferences. It is not possible to determine if any of these four conjectures were included in the test set. Approximating the size of the test set as one tenth of the total number of conjectures, the probability of more than one of the pathological conjectures being in the test set is one percent or less. Even if all four were selected for the test set, a probability of approximately 10^{-4}, they would not have affected the conclusions drawn and can be considered as noise.

As a general observation, conjectures for which proofs can be found by the theorem prover are proved within a reasonable time; that is as the CPU time limit is increased from zero the number of theorems proved at first rises significantly, but after a point plateaus so that further increases in allowed time lead to very few new theorems being proved. For the initial experiment described in this chapter the aim was to set a CPU time limit that was well into this plateau without being so long as to make the experiments unduly time consuming. A value of 300 CPU seconds was set.

4.2 Data used

For the initial experiment, the purpose was to determine whether machine learning worked at all in the context of automatic theorem proving. To get a clear answer it was important to reduce the number of variables not directly related to the experiment, and to this end the sample data was restricted to conjectures from a single area of mathematics. The area of set theory was selected, as it fulfils the requirement of being homogeneous whilst being a separate classification within the TPTP library. The number of problems available was also a significant factor in its choice. (The set theory area of the TPTP library is one of the larger classifications in terms of the number of problems that have been submitted and accepted to the library.) The total number of problems was *approximately 1200.*

4.3 Heuristic used

For machine learning to be effective, learning samples are needed from both classes within a classification problem. The initial experiment concerned the classification of conjectures into those proved to be theorems within a given time and those that were not. Therefore the main constraint on the heuristic used was that it be able to solve a sizeable fraction of the test problems but not all of them. The requirement that the heuristic be not so effective as to solve all the sample problems was not, in practice, a constraint as none of the known heuristics could prove all the problems. Additionally it was important to use a heuristic that was realistic in the sense of being a good heuristic that might be selected by a user working with conjectures of the type used. As the initial experiment was confined to conjectures from a single area of the TPTP library - set theory - a single heuristic could be used. Advice as to the best heuristic to use was sought from Stephan Schulz, the author of E. Details of the heuristic are given in appendix B.

4.4 Running the theorem prover

The theorem prover, E, was modified to automatically write out values for the chosen set of features and additionally to write out the correct classification based on whether or not a proof was found within the set time limit. Values were written to a data file which could form the basis of an input file for the machine learning program SVMLight.

The modified version of E was run on all 1200 conjectures with the CPU time limit set to 300 seconds.

4.5 Training and test data sets

Software was written to randomly split the data into test and learning sets with the test set much smaller than the learning set, the approximate ratio being 90% going to the learning set and 10% going to the test set. The random number procedure was weighted to approximately maintain the same ratio of proved and unproved cases in the test and learning sets. The procedure was carried out ten times to provide ten possible splits to work with. The size of the test set and the ratio of proved theorems to unproved conjectures were not strictly enforced and there was a fairly large variation in the size of test sets. (The reason for this variation was later found to be a programming oversight, but though this affected the exact makeup of the learning and test populations it did not materially affect the validity of the results. Given also that this was a preliminary experiment, it was not re-run with corrected software.) The size of the test sets varied from 101 samples to 178 samples. In testing the different kernel functions, the bulk of the work was done with the first split with the test set containing 178 samples. Then comparisons between the best kernel function from the first split and the linear kernel function was repeated with six other splits.

4.6 Features measured

For the initial experiment a set of sixteen dynamic features was used, details are given in Appendix A. The term dynamic refers to a measurement of features during the proof search rather than using features of the conjecture and axioms prior to the start of the theorem proving process. To measure such dynamic features requires that the theorem prover is run for a pre-determined period and then interrupted. One option would be to set the period on the basis of time but such an approach is dependent on variable factors such as the specification of the computer used. It was decided, instead, to take advantage of the nature of the proof search process itself. E, in common with several other theorem provers, uses the given clause algorithm as described in chapter 2 of this dissertation. In the given clause algorithm there is a main control loop, each iteration of which begins with the selection of a clause from the set of unprocessed clauses. The period for which the theorem prover was run was set in terms of the number of clauses that were selected. A figure of 100 clause selections was used. The figure of 100 was selected on the basis of some experimentation, to provide a good compromise between allowing the proof search to proceed far enough for information on the dynamics to have emerged while not running so long that the time taken was a significant overhead, or for a significant number of conjectures to be proved before the features could be measured.

4.7 Using SVMLight and kernel selection

The machine learning part of the experiment consisted of fitting support vector machines using SVMLight software [36]. Support vector machines use a kernel function to transform measured sample feature vectors as described in chapters 2 and 3. As part of the initial experiment the four standard kernel functions, linear, polynomial, sigmoid tanh and radial were compared to determine the best.

The learning and test procedure was repeated for each of the standard kernel functions as well as for variations of parameter values within each. The bulk of the experiments were done with the first split. The test set contained 178 samples, 101 of which were in the negative class, i.e. unsolvable, and 77 of which were solvable.

4.7.1 Linear kernel

The linear kernel function is the simplest, in this the support vector machine acts in a similar manner to the simple perceptron algorithm. Such a modelling approach will work in cases where the data is close to being linearly separable without transformation, but will not perform well where it isn't.

The initial run was done with all parameters set to default values for SVMLight, which included the use of a linear kernel and an equal weighting between positive and negative samples, (the command line parameter "j" was set to 1.0, this parameter determines the relative weighting of positive to negative samples in the learning process). Note that the linear kernel function itself does not have any user supplied parameters.

The result of the initial run was a learned model or classifier which simply, but rather uselessly, classed all samples as negative. This classifier was right in 101 of the cases

γ	*correct*	*incorrect*	*false positives*	*false negatives*
2.0	122	56		
0.2	111	67	32	35
10.0	123	55	25	30
100.0	124	54	20	34

Table 4.1: Effect of varying γ whilst keeping parameter j set to 2

simply because of the bias within the samples. It is clearly not a useful result other than to demonstrate a base line to compare more useful classifiers.

Setting the parameter j to 1000.0 instead of the default of 1.0 gave a classifier that did the opposite, i.e. set all samples to be positive which was even worse.

It should be noted that trivial classifiers of this type, that place all samples into one class, produce no useful information and the number of correct classifications is dependent on the test set. A test set which contains only samples that should be correctly placed in the other class will lead to zero correct classifications.

Setting the weighting parameter j to 2.0, which accords with the approximate ratio of negative to positive samples within the population, gave a classifier which produced both false positives and false negatives as well as correctly classified results. This was a more intelligent classifier but its success rate only matched that of the always negative case, i.e. 101 correct classifications and 77 incorrect giving a successful classification rate of 56.74%.

4.7.2 Radial basis function kernel

The general expression for the radial basis function kernel is, where $\boldsymbol{x}$ and $\boldsymbol{x'}$ are feature vectors,

$$K(\boldsymbol{x}, \boldsymbol{x'}) = e^{(-\gamma \cdot ||\boldsymbol{x}-\boldsymbol{x'}||^2)}.$$

The parameter γ determines the extent of influence of the support vectors. Where γ is set to a large value, only nearby support vectors have an influence, whereas a small value of γ will bring more support vectors into play. The parameter γ acts as a scaling factor, so its effects depend on the intrinsic scale of the feature vectors. It is difficult *a priori* to determine values for γ so an empirical approach was taken.

Table 4.1 shows the effect of varying γ whilst keeping the parameter j set to 2.0 (on the basis of the best value from the linear kernel function experiments - in later experiments on feature selection the more sophisticated approach of using the exact ratio from the training set was used).

Note that the splitting of the incorrect results into sub-classes of false positives and false negatives was done after the main experiment and was applied only to the systematic variation of γ from 0.2 to 100.0, hence the missing entries in the table. It can be seen that there is a good balance between false negatives and false positives implying that the bias in the sample population has been well compensated by setting the parameter j to 2.0. This is in accord with the experience gained with the linear kernel function. Also, the

s	*c*	*correct*	*incorrect*
1.0	1.0	77	101
1.0	-1.0	77	101
10.0	0.1	77	101
0.1	10.0	101	77

Table 4.2: Varying s and c in the sigmoid tanh kernel

number of correct classifications is not very dependent on the value of gamma provided that it is of the order of unity or greater.

The most significant result is that the number of correct classifications is notably greater than the base level of 101 obtained with the trivial classifier where every case is classified as negative.

4.7.3 Sigmoid tanh kernel

The sigmoid tanh kernel function requires two user-entered parameters, s and c :

$$K(\boldsymbol{x}, \boldsymbol{x'}) = tanh(s\boldsymbol{x}.\boldsymbol{x'} + c).$$

As with the radial basis function experiments, the weighting parameter j was set to 2.0.

Various values of s and c were tried, none of which gave results better than the base case of 101 correct and 77 incorrect. In fact the resultant classifiers gave results equivalent to the trivial case of placing everything in the positive class (77 correct and 101 incorrect), or everything in the negative class (101 correct and 77 incorrect).

Though it is possible that other values of the parameters, perhaps between the last two cases in table 4.2, may have produced better results, overall the results were unpromising compared with the radial basis function case.

4.7.4 Polynomial kernel

The polynomial kernel function,

$$K(\boldsymbol{x}, \boldsymbol{x'}) = (s\boldsymbol{x}.\boldsymbol{x'} + c)^d,$$

has three parameters that must be set by the user: s, c and d.

Table 4.3 gives the results obtained with varying values of the three parameters.

The polynomial case does better than the linear case, and in general beats the base case of 101 correct classifications, but it is worse than the radial basis function kernel.

4.7.5 Further investigation of the radial basis function

Having investigated all the standard kernel functions, and having determined that the radial basis function kernel gives the best results, some further investigations were carried out.

d	*s*	*c*	*correct*	*incorrect*
2	1.0	1.0	108	70
2	10.0	0.1	109	69
2	0.1	10.0	104	74
3	1.0	1.0	111	67
5	1.0	1.0	107	71

Table 4.3: Varying d, s and c in the polynomial kernel

γ	j	*correct*	*incorrect*
10.0	3.0	121	57
10.0	2.0	123	55
10.0	1.0	123	55
1000.0	2.0	118	60

Table 4.4: Varying γ and j in the radial basis function kernel

First the effect of changing the parameter j was investigated and then the effect of setting γ to a very large value was also tried (see table 4.4). (Note that the case for $j = 2.0$ and $\gamma = 10.0$ is simply reproduced from the results given earlier.)

Changing the value assigned to j has little effect. The value of 2.0 makes the most sense, as it approximately balances the bias within the sample population where the ratio is 101 to 77. (In fact a value of 1.5 would be more exact but an integer value is simpler. The lack of change between 1.0 and 2.0 shows that an integer approximation is adequate.)

The very large value of γ, 1000.0, was tried to see what the effect would be of setting a value that would mean only very local feature vectors came into play. The results are worse than those obtained with more reasonable values of γ in the range 10.0 to 100.0.

All the experiments reported so far in this section were done on one pair of test and learning sets. The radial basis function kernel was then tested on six other splits of the samples (see table 4.5). In each case, the value of γ was set to 5.0 and j to 2.0 (based on what had given reasonable results on the first split). Additionally the linear kernel was also tested in each case, to provide a base case for comparison.

The radial basis function kernel results are good in all the splits tried.

Correct (RBF)	*Incorrect (RBF)*	*Correct (Linear)*	*Incorrect (Linear)*
89	31	67	53
105	26	83	48
81	20	70	31
96	33	73	56
89	37	75	51
117	38	36	119

Table 4.5: Results for the radial basis function kernel on other splits.

Feature Number Removed	*Correct Classifications*	*Incorrect Classifications*
1	123	55
2	*121*	*57*
3	123	55
4	*118*	*60*
5	123	55
6	123	55
7	123	55
8	123	55
9	123	55
10	123	55
11	123	55
12	123	55
13	123	55
14	123	55
15	123	55
16	123	55

Table 4.6: The effect of removing individual features.

4.8 Filtering features

Though the size of the feature set in this experiment was small, consisting of sixteen features, the number of possible subsets is still very large. The cardinality of the power set is $2^{16} - 1 = 65,535$ which is too many for exhaustive searching[1]. Instead a very simple approach was taken of removing each feature in turn to see if doing so reduced the efficacy of the learning and resultant classification. (Note that each feature is replaced before the next is removed. The alternative is to remove the features in sequence, this was done as part of the feature selection in the main experiment reported on in chapter 6.) If removing a feature does not reduce the effectiveness of the learning, it is reasonable to deduce that the feature does not make a contribution. This approach will catch features that work in concert, as removing any member of a subset of features that combine will affect the efficacy of the learning process. But if any pair of features are equivalent so there is redundancy then removing either will have no effect. This does not indicate that the feature is of no importance. With this last caveat, the approach is practical and sufficient for an initial first experiment.

The results for rerunning the radial basis function kernel with a γ of 10.0 and the value of j set to 2.0 are shown in table 4.6.

It can be seen from table 4.6 that only two of the features affect the results when they are absent. (Or more accurately, improve the results by being added back to the feature set.)

Feature 4 is a measure of the growth in the unprocessed clause set, whilst feature 2

[1]This was true for the hardware available at this stage of the project. Later on, when a dual Xeon workstation was available, checking 65,535 subsets would represent four to six days of CPU time and therefore it would be possible

γ	*Correct*	*Incorrect*	*False Positives*	*False Negatives*
1.0	116	62	28	34
10.0	119	59	27	32
100.0	122	56	24	32
500.0	121	57	20	37

Table 4.7: Results for the reduced feature set.

is the "sharing factor", a measure of how many sub-terms are held in common between clauses. See appendix A.

4.9 Results for reduced feature set

The results of removing single features indicated that only features 2 and 4 were important. As already stated, the method of removing a single feature would not detect any important features that were also redundant or duplicated by a second feature. To test how sufficient features 2 and 4 were on their own the learning process was repeated on the first split, with just those two features. The value of the parameter j was set to 2.0 as before, and various values of γ used, starting with the previous value of 10.0. The results are given in table 4.7.

It can be seen that the results, though good, are not quite equal to the number of correct classifications obtained when all features are used. This implies either that there is at least one redundant pair of features which are equivalent to each other so were missed in the process of removing and replacing one feature, or that other features make a small contribution which is insufficient to make a notable difference when one is removed but in total have some effect.

4.10 Summary

The purpose of the initial experiment was to determine the presence, or otherwise, of clear indications that machine learning works in the given context of theorem proving. This it did. There were some aspects that could have been improved upon, though there would be no point in repeating the experimental work as the outcome would not be affected. One improvement would be the removal of pathological conjectures for which the theorem prover reaches a saturated state without producing useful clauses. Another improvement would be the careful selecting of test and learning sets that, though random, contained equal numbers of positive and negative cases. A third improvement would be the performance of an additional feature removal experiment in which the removed feature is not replaced.

The initial experiment produced results which indicated that machine-learning was taking place. For example, for the first test set the trivial (and useless) classifier that places all samples in the most popular class would be right in 101 cases and wrong in 77 cases. The classifier based on machine learning was right in 123 cases and wrong in 55. The machine learning based classifier also gave much more balanced results with the erroneous cases balanced between false positives and false negatives. These results were promising enough to continue the work onto a more complex experiment involving heuristic selection.

The initial experiment also determined that, out of the four standard options, the radial basis function kernel gave the best results. Based on this outcome, the radial basis function was selected for the heuristic selection experiment. In addition to giving the best results, the radial basis function has the advantage of using only one parameter, giving a reduced search space for optimisation.

A simple feature filtering scheme determined that two of the features were dominant in determining results. Though dominant, the two features on their own gave slightly worse results than machine learning with all sixteen features.

Chapter 5

Heuristic selection experiment

The results of the initial experiment were promising enough to proceed further. In the initial experiment machine learning was applied to a simple classification problem in one area of conjectures, that of set theory. The classification itself was simply between easy and difficult proofs. Knowing whether a proof will be found quickly or not is useful in some circumstances, but it is more useful to have a better means of finding the proof.

A key decision affecting the efficacy of the proof search by an automated theorem prover is the choice of heuristic to use. The best heuristic to use depends on the proof problem, and it takes a degree of human expertise to select a good heuristic and even experts may not make the best choice. The usefulness of automated theorem proving depends upon the process being usable by scientists, engineers or mathematicians who, though expert in their own fields, are not specialists in the inner workings of theorem provers.

Previous work on machine learning, such as that built into the E theorem prover itself, has concentrated on learning new heuristics. For the work described in this dissertation the approach was to use a fixed set of heuristics and apply machine learning to selecting from the set. The selected heuristic is then applied without modification. The reasons for taking this approach were two fold. Firstly it constrains the problem to working with known heuristics which are, to some extent, tried and tested. Secondly, the published results for work done in the field of modifying heuristics indicate limited success.

5.1 Selecting a working set of heuristics

For the purposes of the experiment, given that for each heuristic to be considered a large number of proofs would be attempted, the number of heuristics in the working set was limited to five.

From an experimental point-of-view, and to aid the process of machine learning, it would be ideal to have five heuristics each of which was clearly best for a well defined subset of the set of conjectures to be used for learning and testing. Additionally it would be ideal to have each such subset of nearly equal size.

In reality heuristics are complex (see appendix B), and there is considerable overlap in the sets of conjectures for which each heuristic does a good job. There are also many proof problems for which none of the heuristics can produce a result within a limited time. Additionally, some heuristics are good over a large number of problems and others work best on only a small number of problems. The choice of heuristics was thus unlikely to be ideal, but needed to be good enough to produce useful results. The heuristics also needed to be useful in their own right rather than artificially produced just for the experiment.

To select the five heuristics, use was made of experimental work already done by Stephan Schulz, the results of which are embodied in the published source code for the E theorem prover [78]. In addition to the machine learning aspects of E as described in Schulz's thesis [77], E has an auto mode for heuristic selection. For auto mode, E uses a few binary or ternary features to divide conjectures into classes. The classification process was applied to the TPTP [85] library of problems and a large number (over one hundred), heuristics run on each class with the best heuristic noted. As much of the generation of the heuristics and the testing was done automatically, information on the results is contained within the header files of E, and from this it is possible to place the heuristics in order, based on the number of the TPTP problems for which the auto mode would select that heuristic. (Note, this is not the same as the number of problems for which that heuristic is the best heuristic. The heuristic finds the most proofs within the class, it is quite possible that there are conjectures within the class that are not proved by the heuristic but would be proved by another heuristic. Even for the conjectures within the class that are proved, another heuristic may find the proof more quickly.) The heuristics were thus ordered and the top five heuristics were selected as a working set.

The working set of five heuristics thus contains the five heuristics most likely to be selected by the auto mode of E if applied to the TPTP library as it stood when Schulz performed his assessment work. Given that E has performed well in competition and that Shulz developed the heuristics as the result of many years of experimentation and research, these heuristics are representative of the state-of-the-art and thus are practically useful heuristics to consider. The method of choice also should go some way to ensuring that different heuristics are best for different parts of the TPTP library, but it should be noted that this is only approximately true, given that the classes used in the preparation of the E auto mode are defined by features without direct reference to heuristics.

5.2 Data used

As for the initial experiment the obvious choice as a source of problems (conjectures with associated axioms) is the TPTP library, and for this experiment all the problems in the library were used. The TPTP library acts as a repository for problems from workers in diverse fields and so provides a wide spectrum of problems. One drawback of using the TPTP library is that it is used in the development of theorem provers, such as E, and also is the basis of problems used in competitions to compare theorem provers, so there is a potential issue with theorem provers being too tailored to the TPTP library and thus not as good for general problems that users may apply them to. To counter this the keepers of the TPTP library actively encourage submission from different problem areas and the library is not static. From the point of view of the work described in this dissertation the question of whether of not machine learning may be applied to the problem of heuristic selection is unaffected and if it works for the TPTP library the same methodology can be applied to future sets of problems with a likelihood of success.

5.3 Feature sets

The initial experiment had used a small set of features all of which were dynamic, that is measured a small way into the proof search process.

The heuristic selection experiment was run in two stages. Initially it was run with the same feature set as used in the preliminary experiment. In the second stage the number of features was extended and an additional set of static features was added. Static features are features of the conjecture and the associated axioms that can be measured prior to the start of the proof search process. The features are described in detail in appendix A. Note that although the experiment was rerun with a new extended feature set, there was no need to repeat the time consuming running of the theorem prover on all conjectures a second time as the time taken for each heuristic had already been recorded in the first part of the experiment. Measuring new features is a fast process as they are either measured directly on the conjecture and axioms (static features), or only a short way into the proof search process (in the case of dynamic features).

The new, extended, feature set consisted of fourteen static features and thirty nine dynamic features. The initial experiment had found that only a few of the sixteen dynamic features made a significant contribution to the learning process, so some justification for extending the feature set is needed. First, with modern machine learning techniques, feature space is transformed to a new space which may have more or fewer dimensions, this counteracts the curse of dimensionality[1]. (There is a cost in machine learning terms in having too many features in terms of optimisation time, and also in the time required to perform a classification from the resultant model, but the experimental outcomes of the present work are not significantly affected[2].) Second, the initial experiment had not examined static features at all and it was important to determine if any such features are of significance, and how such significance compares with the importance of dynamic features. Third, as part of the experiment, subsets of the features were examined so redundant features could be removed.

In summary, machine learning outcomes were compared between results obtained with all features used (the combined case), with just the static features and with just the dynamic features. In addition, experiments were done on examining subsets of features from the combined set to determine which were of significance in the machine learning process. It was discovered as a result of the feature selection work that only a very few features are needed and it was possible to consider all possible small subsets of the feature set used (see chapter 6). Thus starting with more features had no negative effect on the final outcome and had the positive effect of providing a larger pool of features to select from.

[1]The curse of dimensionality is a colourful way of describing how as dimensions increase the number of sample points needed to characterise the space grows exponentially, see Bishop [7].

[2]A difference was found between combining the feature sets and using each separately, and this is reported on in the experimental results. Additionally, small feature subsets gave better results than all 53 features combined, this is reported in the chapter on feature selection.

5.4 Initial separate classifiers

For the first stage, the overall experiment was divided into separate classification experiments for each heuristic, that is for each heuristic a separate classifier was produced. For each classification there were two possible outcomes for each proof attempt. To be placed in the positive class the conjecture must be proved and the time taken to do so must be less than the time taken to prove the same conjecture by any of the other heuristics in the set of five.

Though each heuristic classification can be regarded as a separate experiment, the classification requires that all heuristics are run on each conjecture so the experiments are interdependent.

A time limit of 100 CPU seconds was set for each proof attempt. 100 CPU seconds is sufficient for the majority of proofs that will be found to be found but is short enough to make the length of the overall experiment feasible. (The initial experiment had used 300 CPU seconds but that experiment was restricted to a just one area of the TPTP library and a single heuristic.) For some conjectures, none of the five heuristics could find a proof within the time allowed. To allow for these cases, a sixth classifier was produced for which the positive class is those conjectures for which no heuristic found a proof. For ease of notation, this case was referred to as heuristic zero.

The size of each positive class varied, but the structure of the experiment - with five competing heuristics - was such that the positive class size in each case was much smaller than the negative class. The positive class corresponds to a single heuristic (the best one) while the negative class corresponds to four heuristics, the rest of the set. Ignoring the fact that some conjectures cannot be proved by any heuristic, and assuming that all five heuristics are best in roughly equal numbers of cases leads to the positive class in each case being only a quarter the size of the corresponding negative class. This disparity in class size cannot be avoided but does not prevent useful results from being obtained.[3] The disparity in class size is addressed within SVMLight by means of the parameter j which allows separate weights to be applied to positive and negative slack variables during the optimisation, see Moric *et al* [56] and also chapter 2 of this dissertation.

5.5 Automatic heuristic selection

If each predictive classifier obtained by applying machine learning to the samples in the learning set was perfect, then for any conjecture from the test set only one classifier would place it in the positive class and all the other classifiers would place it in the negative class. Selecting the heuristic to be used in this case would be the simple matter of using the heuristic for which the conjecture was in the positive class.

Such perfect results are highly unlikely to be obtained in practice, and the more likely outcome is that more than one classifier places the conjecture in the positive class. Assuming that the requirement is to select only one heuristic as the best choice, a means

[3] A decision tree approach where heuristics are at first grouped into alternative sets which are successively reduced until a single heuristic remains, might allow for more balanced classifications. But grouping heuristics on the basis of class size is artificial and may well be inconsistent with groups determined from classifiers based on particular features as is normally done in a decision tree approach.

is needed of differentiating between two or more positive results. Fortunately the output value from a support vector machine classifier is not simply a class number (or plus or minus one), but a real number giving the margin. The margin is a measure of how far from the dividing line between the classes the particular sample has been placed (see chapter 2).

By using the margin from each classifier, the heuristics may be placed in order and the best selected. This also allows a heuristic to be selected even if each classifier places the sample in the negative class (in the latter case the least negative result is selected).

5.6 Performance measures for classifiers

For heuristic selection it is the joint performance of the set of SVM classifiers that is important, but each SVM is produced independently of the others so individual performance measures are useful in determining the best parameter values to set.

The primary measure of success of a classifier is the number of test samples that it correctly classifies. Where the test set is unbalanced between positive and negative samples the number of correct classifications alone may give a misleading picture; the classifier may be biased. A fuller picture may be obtained by considering the number of false negatives (positive samples misclassified as negative) and the number of false positives (negative samples misclassified as positive). A good classifier should have a reasonable balance between false positives and false negatives.

There are a number of more formal measures which combine the number of true positives (TP), the number of true negatives (TN), the number of false positives (FP) and the number of false negatives (FN). The fields of statistics and information retrieval use the same measures but give them different names.

The proportion of positive values that are correctly classified is called the sensitivity in statistics and the recall in information retrieval.

$$sensitivity = \frac{TP}{TP + FN}$$

A similar expression for the proportion of negative samples that are correctly classified is known as the specificity.

$$specificity = \frac{TN}{TN + FP}$$

The precision is the proportion of values that are classified as positive which are true positives.

$$precision = \frac{TP}{TP + FP}$$

Sensitivity, specificity and precision each measure a single aspect of the classifiers performance, it is useful to have a single combined measure. The F-measure combines the sensitivity and the precision into a single function, see Zhenqiu Liu *et al* [44].

$$F_\gamma = \frac{1}{\gamma(\frac{1}{sensitiviy}) + (1-\gamma)(\frac{1}{precision})}$$

or, in terms of TP, FN and FP:

$$F_\gamma = \frac{TP}{TP + \gamma FN + (1-\gamma)FP}$$

where

$$0 \le \gamma \le 1.$$

In the general case γ is neutrally weighted at $\frac{1}{2}$ so the F-measure is often expressed as

$$F = 2 \times \frac{precision \times recall}{precision \ + \ recall}$$

Note that the γ in the F_γ measure bears no relation to the parameter γ of the radial basis function kernel and it is the latter that is referred to in all results given in this dissertation.

In the results that follow judgement was based on the percentage of correct classifications and the balance between false positives and false negatives. The combined F measure was not used: at the point where such a measure would be useful in optimising the value of the radial basis function kernel parameter γ in detail, it was possible by feature reduction to use the overall efficacy of all classifiers combined instead. This is described in chapter 6 on feature selection.

5.7 Unextended feature set experiments

Though the target outcome of the experiments is a means of heuristic selection, interim results from each separate heuristic classifier may be tested to give an indication of whether or not learning is taking place. That is having produced the SVM classifier from a learning set of conjectures the classifier may then be applied to a test set of conjectures.

5.7.1 First classifications

The first run of the experiment was done with the same feature set as used for the preliminary experiment. To allow for the imbalance between positive and negative classifications the weighting parameter j in SVMLight was varied. SVMLight has separate weights for the positive and negative slack variables during the optimisation (see chapter 2) and the parameter j sets the ratio between the two weights, see Morik *et al* [56]. The radial basis function model was used in all cases as it had proved to be the best in the initial experiment. The value of the radial basis function parameter γ was set to 10.0 in most cases, though it was also increased to 100.0 in some additional trials as a check. (The value of 10.0 had proved best in the initial experiment.)

Additionally each series of trials was repeated with the SVMLight parameter i switched on. This parameter is a flag which is off by default and, if switched on, causes the SVMLight software to retrain the classifier after removing inconsistent cases (that is samples which are misclassified which is allowed by the use of slack variables, see chapter 2). This was an additional experiment which had not been tried in the preliminary experimental work.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	2755 (82%)	62	11	2693	584
10.0	10.0	2443 (73%)	314	575	2129	332
100.0	10.0	2106 (63%)	338	936	1768	308
10.0	100.0	2531 (76%)	224	397	2307	422

Table 5.1: Classification results for H1 with parameter i turned off.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	2704 (81%)	0	0	2704	646
10.0	10.0	774 (23%)	643	2573	131	3
5.0	10.0	2492 (74%)	302	514	2190	344
5.0	100.0	2549 (76%)	223	378	2326	423

Table 5.2: Classification results for H1 with parameter i turned on.

On the initial run, the test samples were split into two sets - training and test - but this was done independently for each separate heuristic classification experiment. Thus the division of the conjectures between the two sets is not identical in each case. This did not matter for the individual classifications but needed to be corrected when all the heuristic classifiers were combined to select the best heuristic by comparing margins (to ensure that there was no bias). Results are given for the runs prior to the correction (of the training and test sets), as well as after correction, as in the first instance different parameter values were experimented with, and the results used to fix the parameter values on the repeated runs with the fixed training and test sets.

Classification on H1

The number of test samples was 3,350. Out of these 646 were in the positive class and 2,704 were in the negative class. The results in table 5.1 were obtained with the default setting of off for the parameter i. The results in table 5.2 were obtained with the parameter i turned on so that the SVMLight software re-optimised after removing inconsistent cases. With the parameter i set, the results are more sensitive to the value of the weighting parameter j. In judging the results, the number of correct classifications is not the sole criterion. Given the inevitably unbalanced nature of the set, an assessment needs also to be based on the requirement to obtain a reasonable number of positive cases without an excessive number of false positives.

Classification on H2

The number of test samples was 3,350. Out of these 283 were in the positive class and 3,067 were in the negative class. The results in table 5.3 were obtained with the default setting of off for the parameter i.

The results in table 5.4 were obtained with the parameter i turned on so that the SVMLight software re-optimised after removing inconsistent cases. Similar comments apply as for the classification on H1.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	3068 (92%)	1	0	3067	282
10.0	10.0	2903 (87%)	91	255	2812	192
100.0	10.0	2481 (74%)	114	700	2367	169
10.0	100.0	2966 (89%)	71	172	2895	212

Table 5.3: Classification results for H2 with parameter i turned off.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	3067 (92%)	0	0	3067	283
10.0	10.0	2740 (82%)	97	424	2643	186
5.0	10.0	3021 (90%)	54	100	2967	229
5.0	100.0	3031 (90%)	64	100	2967	219

Table 5.4: Classification results for H2 with parameter i turned on.

Classification on H3

The number of test samples was 3,351. Out of these 453 were in the positive class and 2,898 were in the negative class. The results in table 5.5 were obtained with the default setting of off for the parameter i. The results in table 5.6 were obtained with the parameter i turned on so that the SVMLight software re-optimized after removing inconsistent cases. Similar comments apply as for the classification on H1.

Classification on H4

The number of test samples was 3,351. Out of these 337 were in the positive class and 3,014 were in the negative class. The results in table 5.7 were obtained with the default setting of off for the parameter i. The results in table 5.8 were obtained with the parameter i turned on so that the SVMLight software re-optimised after removing inconsistent cases. Similar comments apply as for the classification on H1.

Classification on H5

The number of test samples was 3,351. Out of these 349 were in the positive class and 3,002 were in the negative class.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	2900 (87%)	9	7	2891	444
10.0	10.0	2629 (78%)	138	407	2491	315
100.0	10.0	2390 (71%)	150	658	2240	303
10.0	100.0	2773 (83%)	87	212	2686	366

Table 5.5: Classification results for H3 with parameter i turned off.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	2898 (86%)	0	0	2898	453
10.0	10.0	2524 (75%)	147	521	2377	306
5.0	10.0	2798 (84%)	109	209	2689	344
5.0	100.0	2831 (84%)	68	135	2763	385

Table 5.6: Classification results for H3 with parameter i turned on.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	3025 (90%)	18	7	3007	319
10.0	10.0	2823 (84%)	168	359	2655	169
100.0	10.0	2372 (71%)	201	843	2171	136
10.0	100.0	2906 (87%)	118	226	2788	219

Table 5.7: Classification results for H4 with parameter i turned off.

The results in table 5.9 were obtained with the default setting of off for the parameter i. The results in table 5.10 were obtained with the parameter i turned on so that the SVMLight software re-optimised after removing inconsistent cases. Similar comments apply as for the classification on H1.

H0

H0 is the case where none of the five heuristics can find a proof within the time limit (of 100 CPU seconds). The number of test samples was 3,351. Out of these 1,282 were in the positive class and 2,069 were in the negative class. The results in table 5.11 were obtained with the default setting of off for the parameter i. The results in table 5.12 were obtained with the parameter i turned on so that the SVMLight software re-optimised after removing inconsistent cases. The balance for this case is different to that for classifications on H1 to H5. This is to be expected, as the classification of hard or easy problems is very different from the classification as to whether a heuristic is best out of five choices.

Parameters j and γ in more detail

In the above experiments fairly crude steps in j and γ were used. A more detailed and sophisticated approach was taken when the number of features was reduced in the feature

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	3014 (90%)	0	0	3014	337
10.0	10.0	2585 (77%)	192	621	2393	145
5.0	10.0	2959 (88%)	140	195	2819	197
5.0	100.0	2981 (89%)	99	132	2882	238

Table 5.8: Classification results for H4 with parameter i turned on.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	3006 (90%)	16	12	2990	333
10.0	10.0	2726 (81%)	184	460	2542	165
100.0	10.0	2469 (74%)	196	729	2273	153
10.0	100.0	2880 (86%)	129	251	2751	220

Table 5.9: Classification results for H5 with parameter i turned off.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	3002 (90%)	0	0	3002	349
10.0	10.0	2703 (81%)	189	488	2514	160
5.0	10.0	2760 (82%)	181	423	2579	168
5.0	100.0	2908 (87%)	122	216	2786	227

Table 5.10: Classification results for H5 with parameter i turned on.

selection experiments described in chapter 6. In the feature selection experiments the value of j was set to exactly balance the positive and negative class numbers in the training set in each case. Additionally, for the optimum feature subset the value of γ was varied in small steps over a wide range, see chapter 6 for results.

5.7.2 Identical learning and test sets

The initial classifications, reported in the previous sections, were performed as separate experiments. The splitting of samples into learning and test sets was done independently in each case, and though the sets were very similar they weren't forced to contain exactly the same samples in each case. The next stage of the experiment was to force the same conditions on each heuristic classification including using identical sets for learning in each case and similarly identical sets for test in each case.

Reviewing the results of the first classification experiments, the decision was taken to set parameter values as follows. The parameter option i was set so that a refit was done with anomalous cases removed. The value of parameter j was set to 5.0, which gave good results and also is roughly in accord with there being five heuristics competing so that, crudely approximating the unsolved problems to being equal in number to any one heuristic, each positive class should be around one fifth the size of the negative class[4].

[4]This approximation was improved upon in the feature selection experiments described in the next

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	2464 (74%)	512	117	1952	770
10.0	10.0	2331 (70%)	907	645	1424	375
100.0	10.0	2302 (69%)	910	677	1392	372
10.0	100.0	1807 (53%)	1221	1483	586	61

Table 5.11: Classification results for H0 with parameter i turned off.

j	γ	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1.0	10.0	2293 (68%)	282	58	2011	1000
10.0	10.0	1355 (40%)	1282	1996	73	0
5.0	10.0	1282 (38%)	1282	2069	0	0
5.0	100.0	1352 (40%)	1282	1999	70	0

Table 5.12: Classification results for H0 with parameter i turned on.

Heuristic	*Correct*	*True Pos*	*False Pos*	*True Neg*	*False Neg*
1	2695 (80.42%)	175	169	2520	487
2	3005 (89.67%)	48	115	2957	231
3	2888 (86.18%)	53	75	2835	388
4	2989 (89.20%)	128	139	2861	223
5	2918 (87.08%)	61	161	2857	272
0	1358 (40.53%)	1285	1993	73	0

Table 5.13: Summary of results for identical learning/test sets

The value of γ for the radial basis function was set to 10.0. The number of samples in the test set was 3,351 and the results are given in table 5.13.

Looked at as individual experiments to produce classifiers, these results don't indicate an obvious success for the machine learning. But this is not how the data should be viewed. The purpose of the experiments was to produce a set of classifiers which together provide a means of selecting the best heuristic. The key is the relative size of the margin in each case. For example, a false positive for one heuristic will not matter if the margin associated with it is less than the margin in the classifier where the same conjecture has been correctly classified as positive. In this, hypothetical, example the correct heuristic will be assigned even though one of the other heuristics is laying false claim to it.

The key test is whether or not the overall selection process does better than assigning all conjectures to any individual heuristic. This comparison is made in the next section which reports the results of the heuristic selection experiment.

5.7.3 First results of heuristic selection

To determine the efficacy of the machine learned heuristic selection process, two measures were used. One was the number of theorems successfully proved. The other was the total time taken. The two are related in that a CPU time limit of 100 seconds was set for unsuccessful attempts and so the total time taken was determined, to some extent, by the amount of time allowed for fruitless proof searches. The total time could be reduced by reducing this limit, but some proofs previously found may then be recorded as failures due to too early a cut off point. Similarly extending the time allowed may allow more proofs to be found but the overall time taken would be extended. As the experiment involved the

chapter, where the value of j was set to be equal to the exact ratio of positive to negative cases in the training set.

Method Used	*Total Time in Seconds*	*Number Proved*	*Number of Failures*
Selected Heuristic	155,861	1,751	1,600
Fixed Heuristic 1	164,344	1,725	1,626
Fixed Heuristic 2	182,951	1,525	1,826
Fixed Heuristic 3	173,376	1,584	1,767
Fixed Heuristic 4	171,674	1,623	1,728
Fixed Heuristic 5	242,484	1,489	1,862

Table 5.14: First heuristic selection results.

Heuristic	*Number of Times Selected*
1	1,128
2	212
3	1,296
4	536
5	179

Table 5.15: Number of times each heuristic was selected in the learned heuristic selection.

comparison between heuristic selection and each heuristic by itself, the ordering shouldn't be affected but the apparent time differences would vary if different parameters were set.

In the heuristic selection case, the total time taken is calculated on the basis of the measured time for the relevant selected heuristic for each sample. This is very slightly optimistic in that it does not allow for any overhead in the selection process. The selection process was not programmed into the theorem prover, so the time taken by it could not be exactly determined, but the process is very rapid and is negligible compared with the time taken in proof search. It is certainly less than differences which would arise, for instance, from taking a different CPU cut-off point as discussed in the previous paragraph.

The results for heuristic selection versus individual heuristics are summarised in table 5.14. The total number of test samples (conjectures for which proofs were sought) is 3,351.

It can be seen that the heuristic selection scheme does better than any of the heuristics individually. Thus, the machine learned algorithm for heuristic selection is making appropriate decisions. If the process of selection was random the results would likely be worse than the best individual heuristics. (See chapter 6 on feature selection for simulated results for random heuristic selection which are clearly worse than those obtained with the best fixed heuristic. The results for random feature selection give an indication of statistical significance - see figures 6.52 to 6.55, it can be seen that the probability of obtaining results as good as those obtained from the heuristic selection purely by chance is negligibly small.) To demonstrate that the selection process is significant, rather than perhaps trivially always selecting the same heuristic, table 5.15 shows how many times each heuristic was selected.

At this stage of the experiment, machine learning had been demonstrated to improve on any particular heuristic. Such single heuristics provide a useful base case, but it is also important to know if there is an upper limit. It would be pointless seeking to improve the

Heuristic	*Number of Times Selected*
1	1,053
2	220
3	317
4	213
5	222

Table 5.16: Number of times each heuristic selected with perfect heuristic choice.

heuristic choice method if the five heuristics in the working set are insufficient to provide headroom for further improvement. To test this upper limit, results were calculated for the case of a perfect heuristic choice for each conjecture, that is the best heuristic is always selected.

Such a perfect heuristic choice would prove 2,025 of the conjectures whilst still failing on 1,326 problems. The total time taken would be 137,664 seconds. Thus there is still some room for improving the heuristic selection process. The number of times each heuristic would be selected in a perfect scenario is given in table 5.16. Comparing table 5.16 with table 5.15 the largest difference is in the number of times heuristics 3 and 4 are selected. The heuristic selection is putting too much emphasis on heuristics 3 and 4 which would indicate that the associated SVM classifiers are giving too high margin values relative to the classifiers for the other heuristics.

5.8 Experiments with extended feature sets

Following positive results from machine learning with the same feature set as used in the initial experiment, the next step taken was to extend the feature set. Details of the extended features are given in Appendix A. The extended feature set can be split into two subsets. The static set of features can be determined by measuring the conjecture and axioms prior to any proof search. The dynamic set of features is measured on the proof state a short way into the proof search process. By applying the machine learning to both feature sets separately, as well as the combined set, a comparison between the two may be made.

5.8.1 Classifications with extended feature set

Parameters for SVMLight were set to values determined in the earlier experiment with the smaller feature set. The number of samples in the test set was 3,345. Table 5.17 summarises the results for each heuristic classification.

The combined case includes both the dynamic and static features. Given that the combined case is a superset of either the dynamic or static cases it should do at least as well as either, but in several instances does slightly worse. This shows that having too many features can have a detrimental effect. This will be addressed in chapter 6 which gives the results of the feature selection experiments for which optimal results were obtained with small subsets of features.

Case	*Number Correct*	*Correct Pos*	*False Pos*	*Correct Neg*	*False Neg*
H1 Static	2,743 (82%)	78	33	2,665	569
H1 Dynamic	2,712 (81.8%)	32	18	2,680	615
H1 Combined	2,702 (80.78%)	18	14	2,684	629
H2 Static	3,064 (91.60%)	9	6	3,055	275
H2 Dynamic	3,063 (91.57%)	3	1	3,060	281
H2 Combined	3,063 (91.57%)	3	1	3,060	281
H3 Static	2,878 (86.04%)	15	9	2,863	458
H3 Dynamic	2,873 (85.89%)	2	1	2,871	471
H3 Combined	2,874 (85.92%)	2	0	2,872	471
H4 Static	3,006 (89.87%)	12	20	2,994	319
H4 Dynamic	3,019 (90.25%)	37	32	2,982	294
H4 Combined	3,012 (90.04%)	14	16	2,998	317
H5 Static	2,975 (88.94%)	6	4	2,969	366
H5 Dynamic	2,976 (88.97%)	3	0	2,973	369
H5 Combined	2,976 (88.97%)	3	0	2,973	369
H0 Static	2,488 (74.38%)	531	150	1,957	707
H0 Dynamic	2,458 (73.48%)	444	93	2,014	794
H0 Combined	2,396 (71.63%)	372	83	2,024	866

Table 5.17: Summary of extended feature set results.

The dynamic and static features produce very similar results. The static feature set does better on heuristics 1, 2 and 3 and also in the case of the conjectures for which no heuristic could find a proof, heuristic 0. The dynamic feature set does better on heuristics 4 and 5.

As with the classification results of the experiments with the smaller feature set, it is difficult to draw conclusions from the individual classification experiments themselves. The important results are those of heuristic selection using the combination of all the individual heuristic SVM classifiers. These are reported in the next section.

5.8.2 Heuristic selection with extended feature set

Correcting for CPU limit bug in E

On the initial results of these experiments, the total timings for one of the cases of a fixed heuristic were excessive, given a CPU limit of 100 seconds. Further investigation, and some consultation with colleagues who also use E, showed that very occasionally E will fail to halt at the CPU limit.

Code was written in C to analyse the data files and to do two things. First, to check if in any case the extra time erroneously applied led to a proof being found. Secondly to correct the timings down to 100 seconds where this time limit was exceeded.

Fortunately, from the point of view of this experiment, it was found that in no case did the excessive time lead to a proof being found where none had been found after 100

Method Used	*Time in Seconds*	*Number Proved*	*Failures*
Static Case	157,445	1,755	1,590
Dynamic Case	158,033	1,764	1,581
Combined Case	159,602	1,751	1,594
Fixed Heuristic 1	162,852	1,739	1,606
Fixed Heuristic 2	181,452	1,541	1,804
Fixed Heuristic 3	168,737	1,626	1,719
Fixed Heuristic 4	170,238	1,616	1,729
Fixed Heuristic 5	174,317	1,542	1,803

Table 5.18: Final results for heuristic selection using extended feature sets.

CPU seconds. This is not surprising, as the 100 second CPU limit was chosen to be long enough to allow the prover to find proofs where it could; if none is found after 100 CPU seconds then it is likely that no proof will be found within any reasonable time limit. The results given in the following sections give corrected timings.

Results

The final results for heuristic selection, using extended feature sets, are given in table 5.18.

All three heuristic selection schemes do better than any of the heuristics on their own.

The results for the dynamic case are best in terms of the number of proofs found, but the total time taken is slightly more than for the static case. The combined case does a bit worse than either static or dynamic feature sets separately.

Results of including the H0 case

Heuristic H0 is used as a short hand to indicate the case where none of the five heuristics is able to find a proof within the 100 second CPU limit. Results for H0 classification have been given in the appropriate sections, but these results were not initially used in the combined process of heuristic selection. Combining the H0 classifier with the others in the heuristic selection process will not increase the number of proofs found but it may lead to a worthwhile improvement in the total time taken as time is not wasted on fruitless proof searches.

Given that the H0 classifier will not be perfect, including it in the heuristic selection process carries a cost in terms of a reduction in the number of proofs found. This cost may be worth paying if the overall reduction in time is large.

In the first attempt at including H0, it was treated in the same manner as the other heuristic classifiers. That is, if the H0 margin was the most positive, or least negative, then it was selected. This gave very bad results. In the static case the number proved dropped from 1,755 to 755. In the dynamic case the number proved dropped from 1,764 to 508. In the combined case the number proved dropped from 1,751 to just 413. Though the time taken was greatly reduced as well it was clearly too expensive a price to pay.

Method Used	*Time (sec) (no H0)*	*Time (sec) (H0)*	*Proved (no H0)*	*Proved (H0)*
Static	157,445	101,980	1,755	1,627
Dynamic	158,033	111,797	1,764	1,691
Combined	159,602	120,986	1,751	1,682

Table 5.19: Results including H0 case.

	Static		*Dynamic*		*Combined*	
CPU Time Limit	*Total Time*	*Theorems Proved*	*Total Time*	*Theorems Proved*	*Total Time*	*Theorems Proved*
100	157,455	1,755	158,033	1,764	159,602	1,751
75	119,915	1,736	120,334	1,739	121,515	1,727
50	81,536	1,694	81,734	1,699	82,605	1,690
25	42,137	1,646	42,182	1,653	42,665	1,638

Table 5.20: The effect of reducing the CPU time limit from 100 seconds to 25 seconds.

To solve this problem, H0 was treated as a special case. Only if the margin was positive for the H0 classifier would it be selected. This gave much better results which are given in table 5.19. The results in table 5.19 show that by applying the H0 filter reduces the time taken by around a third whilst reducing the number of proofs found by only 5 to 8 percent.

It should be noted that the total time taken could also be reduced by setting a lower CPU limit. This would cause all proofs that took over the new, lower limit, to be lost. Using the H0 filter, though it does reduce the number of proofs found, should not produce such a systematic error. Table 5.20 shows the effects of reducing the CPU time limit from 100 seconds to 25 seconds. It can be seen that for the TPTP data simply reducing the CPU limit actually gives better results than using H0 filtering. This result must be treated with caution. The TPTP library has grown over a number of years and initially there was an emphasis on smaller problems as early theorem provers were only capable of solving smaller problems. This means that the library is likely to be skewed towards easy problems and if such is the case, simply reducing the CPU limit works well. For a population of problems where solvable cases were all difficult and took close to the CPU limit, reducing the CPU limit would drastically reduce the number of proofs found while H0 filtering would be more robust.

5.9 Further analysis

Further analysis was carried out to obtain a more detailed picture of the results. The purpose of this analysis was to look at whether or not some problems, that is conjectures with associated axioms, are intrinsically difficult to classify while others may be intrinsically easier to classify.

Each problem used in the testing of the classifiers is taken originally from a TPTP file and the name of the file was retained as a tag on each data line written out. Software was written that for each classification by the individual heuristic SVM classifiers wrote out two files, one containing the names of samples that were correctly classified and the other containing the names of problems incorrectly classified. These files essentially listed the elements of the set of correctly classified problems and the set of incorrectly classified problems. Note that the problems from each set may be in either the positive or negative classes.

Further software was then written to find the common names between two or more files, i.e. the intersection of the sets. This filtering process was then applied to various combinations of the classifications.

First it was applied to all classifications in the order H0 to H5, starting with the static feature set results, then the dynamic feature set results and finally the combined feature set results. The number of common problems found at each stage, (that is the number of problems correctly classified by all classifiers up to that point), is given in table 5.21.

The choice of ordering is arbitrary but some general conclusions may be drawn. First, only 389 out of 3,345 problems are correctly classified by all the classifiers. Secondly, there is a significant drop when moving between feature sets as well as moving between heuristics. Thirdly, after an initial drop, the combined feature set case shows no further drop between heuristics, this is perhaps a reflection of the combined set of heuristics containing no extra features beyond those of the static and dynamic feature sets.

Performing the same experiment with problems that were misclassified leads to a rapid fall off to zero. 857 problems were misclassified by the H0 classifier, (in the static case), and of these 46 were also misclassified by the H1 classifier but of those none were misclassified by the H2 classifier.

There are many possible orderings, and several more were investigated, but are not reported here, as no significant new information was derived.

The main conclusion to be drawn from the results in table 5.21 is that the efficacy of the machine learning and the resulting SVM classifiers is not particularly dependent on which problems are being studied. The core of problems that all classifiers are able to classify correctly is small compared to the total number of problems looked at.

5.10 Conclusions

In all the results, the machine-learned heuristic selection process did better than any fixed heuristic. This demonstrates that machine learning is taking place and that there is useful information in the features being measured, and is a positive outcome to the experimental work.

The H0 classifier, which essentially classifies problems as provable or too difficult, gave good results. This is in accord with the promising results obtained in the preliminary experiment. The H0 classifier may be used as a filter to reduce the total time spent on seeking proofs for a large set of problems, though doing so leads to a small reduction in the total number of proofs found.

The dynamic and static feature sets gave very similar results. This implies that allowing the proof search to start and run for a short time does not lead to significantly more information being available. This must be a tentative conclusion. It may be that more

Classifier	*Problem Files Correctly Classified By This and All Above Classifiers*
Static Case	
H0	2,488
H1	1,932
H2	1,700
H3	1,280
H4	987
H5	651
Dynamic Case	
H0	514
H1	443
H2	435
H3	420
H4	419
H5	414
Combined Case	
H0	389
H1	389
H2	389
H3	389
H4	389
H5	389

Table 5.21: Analysis of the number of files correctly classified by more than one heuristic classifier.

information could be obtained by a number of restarts with different heuristics being tried and compared or with other features being measured. The following chapter on feature selection goes into much more detail and the optimal feature subsets found contain a mixture of dynamic and static features.

The combined feature set often gave worse results than either the dynamic or static feature sets separately. The chapter that follows gives details of extensive experiments to determine which features were significant and which were superfluous, or even detrimental. Optimal results are obtained, in fact, with only a few features being used but it is important to have a large pool of both static and dynamic features to pick such an optimal subset from.

By comparison with the results obtained by always choosing the best heuristic, it was demonstrated that there is potential for further improvement in the machine learned selection process without needing to extend the heuristic set from the five used. In other words, the use of a limited heuristic set of five did not limit the machine learning process.

Chapter 6

Feature selection

Results from the initial experiment and the follow up heuristic selection experiments demonstrated two aspects of the feature sets used. First, there is redundancy amongst the features. Some, if not most, do not contribute to the machine learning process. Second, though modern machine learning techniques such as support vector machines are very tolerant of extra features, having too many features leads to worse results. This was supported by the finding that a combination of both dynamic and static features generally led to a less effective heuristic selection scheme than either feature set used on its own.

This chapter describes a series of feature selection experiments. The first experiment was the automatic removal of features one at a time in a manner similar to the manual approach used in the preliminary experiment. The results of the first experiment indicated that, as with the preliminary experiment, only a small number of features are needed for learning to be effective. By restricting learning sets to just a few features, (three or less), it became feasible to look at all possible such subsets of the full feature set and this was done as a follow up experiment. Though it was not possible, even with a powerful workstation, to check all four feature subsets it was possible to check all four feature subsets of a reduced, thirty-five feature, subset of the original fifty-three feature set. The extension to four features did not improve the learning results for the three feature subsets, so planned further extensions in feature numbers were not carried out as it was deemed unlikely that they would lead to any further improvement.

Finally, for the top three subsets the value of γ was varied over a wide range allowing both an optimal value to be found and a clear indication of upper and lower limits outside of which results drop significantly.

Note that there are other techniques of feature selection as described in chapter 2. It was not necessary to employ additional methods in the present case because the number of features needed was shown to be small enough for *all* subsets to be tried which is guaranteed to find the best one.

6.1 Selectively removing features

As with the initial experiment, the first approach to determining the importance of features was to test the effect of removing the features individually. Whereas in the initial experiment there were few enough features to do this manually, for the extended heuristic selection experiment there were 53 features so software was written to automate the process. In addition, the experiment was both extended and made more sophisticated.

6.1.1 Ordering features for removal

In the initial experiment the feature ordering was done on the basis of removing each feature in numerical order and then replacing it before removing the next. For the heuristic selection classifiers the process was extended. As before, each feature was removed, a model fitted to a learning data set and then the classifier tested on a test data set before the feature was restored. This was done on all the features, resulting in a score for each where the score was the number of correct classifications made.

Having obtained a score for all features in the above manner, the feature with the highest score was then permanently removed before the process was repeated on all features in the reduced feature set. The highest score is used as this corresponds to the removal of the least useful feature whose absence has the least detrimental effect. If over-fitting is happening it may be that removal of a feature actually leads to a greater number of test conjectures being correctly classified.

It should be noted that, given a large number of features which are not significant, many features will give identical scores so an arbitrary decision needs to be taken as to which to remove first. The experiment was set up in two ways, in one the first of the equal scoring features was removed first. In the second, the last of the equally scoring features was removed first. Though, as far as the effect on the classifier is concerned, the order the features of equal score are removed in doesn't matter, it does matter for the purpose of assigning a rank to each feature. By combining the results of priority first and priority last experiments any bias as to the numerical value (or label) of the feature should be removed.

There were two useful outcomes to these experiments. First, by plotting score versus the number of features removed, a clear visual indication was given for each classifier as to how many features are actually needed for effective machine learning. Second, each feature was ranked for each classifier and the rankings from all classifiers then combined to give an overall score. The ranking score gives a clear indication of which features are important.

Note that in the plots shown, the axis starts at 1500, rather than zero. Also, the plots show number of features removed on the x-axis, so to assess the effect of having more features remaining they should be viewed from right to left.

Figure 6.1 shows the results of progressive feature removal for the H0 classifier (*i.e.* the classification into the class of conjectures that may be proved by at least one heuristic and the class of conjectures that cannot be proved to be theorems by any of the five heuristics within the allowed time limit). It can be seen that the first few features to be removed actually improve the the number of correct classifications obtained. After this there is little change until the last few features are removed, indicating that only these are significant. The last bar is for all features removed. This last case leads to a trivial classifier where everything is classified as negative, there are no false positives and only false negatives. It is included, not as a useful case, but to show the proportion of negative to positive samples and to give a baseline.

Figure 6.2 shows the results for the H1 classifier (*i.e.* the classification into the class of conjectures for which heuristic H1 is the best heuristic to use in the proof search and the class of conjectures for which this is not the case). Note that in this case the results for one feature remaining are the same as for zero features, *i.e.* a trivial classifier placing all samples in the negative class. The bars indicate that four features are of significance, with three of them being of more significance than the fourth.

Figure 6.3 shows the results for the H2 classifier. This classifier, with the weighting parameter j fixed, is not well balanced, there are almost no false positives. The implication is that the classifier is classifying almost every case in the negative class.

Figure 6.4 shows the results for the H3 classifier. The number of false positives is again small. The number of correct classifications changes very little with the number of features but the balance between false positives and false negatives varies much more, the number of false negatives drops while the number of false positives rises until there is only three features left. As the last few features are removed the number of false negatives rises and the number of false positives drops.

Figure 6.5 shows the results for the H4 classifier. With one feature remaining, *i.e.* at the right hand side of the plot, the classifier is essentially trivial, producing the same results as zero features. Results improve for up to four features and then are flat.

Figure 6.6 shows the results for the H5 classifier. In this case, beyond the first three features there is no change, *i.e.* there is no change until fifty features have been removed. As with the H4 case, having just one feature remaining leads to a trivial classifier giving the same results as the zero feature case.

Figure 6.7 shows a plot of a score based on rank position for every feature summed over all the heuristic classifier experiments. The scores for priority first and priority last results in each case are both included so as to remove bias. The results from this first run of the experiment do not show any particular features as being obviously outstanding. Examination of figures 6.1 to 6.6 shows that the performance in terms of correct classifications does not change on removal of many of the intermediate features implying that the feature removed is of equal importance to the the one removed previously, yet the ranking score is different. This difference is arbitrary and clouds true differences in importance. The experiment was repeated, where in addition to a number of other improvements such as optimisation, the ranking score was only increased for any particular heuristic if the feature had done better than the previous one. In other words, in the ranking of features, features of equal worth were given equal rank. This repeated experiment is described, and results given, in the next section.

6.1.2 Including optimisation and other improvements

The first runs of the feature ranking experiment used a fixed command line which set parameters j to 2.0 and γ to 10.0. (Additionally, the parameter i was not set, to simplify the process.) As noted in the previous section, with the fixed command line several of the classifiers were unbalanced. That is, they classified too many samples as negative and far too few as positive. A second run of experiments was undertaken to correct this.

Two improvements were implemented. First, j was not fixed, instead it was set on the basis of the number of negative and positive samples in the learning set. Secondly, an extra optimising loop was added to vary the value of γ. The optimisation was limited as the inner loop involved the fitting of the support vector machine model as well as testing it on a large test file, both of which are relatively slow processes involving reading and writing files from disk. The optimisation was thus rather crude, involving picking the best of a few samples over a wide range of γ values. Figures 6.8 to 6.13 show the γ values obtained. The values were limited to the range 0.5 to 256 with the value doubled between each intermediate step. In many cases γ ends up at either extreme and only shows meaningful values for H0 and H1 where the number of features is down to three or four. A more useful and detailed optimisation of γ was performed for the reduced feature set and is described later in this chapter.

Additionally, and importantly, the analysis of features on the basis of their ranking position in each heuristic gave equal scores to all features of equal worth.

The results of the more sophisticated experiments are given in figures 6.14 to 6.26. For this run the feature scores for each heuristic separately are shown in figures 6.20 to 6.25 in addition to the combined score shown in figure 6.26. It can be seen that different features are prominent for different heuristic classifiers.

Comparing, for example, figure 6.16 to figure 6.3 it can be seen that the new results are better balanced in terms of false positives and false negatives, though there is still some asymmetry.

Comparing figure 6.26 to figure 6.7 it can be seen that there is now a much clearer differential between features. Features 16 and 40 stand out, these correspond to dynamic feature numbers 2 and 26. Feature 2 is the sharing factor whilst feature 26 is the ratio of the number of generated literal clauses to the total number of processed clauses (see Appendix A).

Before drawing too firm a conclusion from these results it is necessary to consider the results of the experiments described in the rest of this chapter. The scoring scheme used, based on rank order for individual classifications summed together is perhaps too far removed from a direct measure of the ultimate outcome which is the effectiveness or otherwise of the heuristic selection process. The subset experiments which follow used a direct measure of theorems proved and considered all cases of feature combinations up to three features. The key features from the subset experiments turned out to be different. (The other difference with the subset experiments was that the pathological examples, for which conjectures were apparently proved not to be theorems due to early saturation of the clause set, were removed from the data sets. There was only a very small number of these cases, too few to directly affect the results.)

6.2 Testing small and large feature subsets

The results of successive feature removal and associated feature ordering indicated that the number of significant features is small. This opens up the possibility of exploring all feature subsets up to a size already determined as sufficient. For the total number of 53 features the number of subsets is given by $2^{53} - 1$ (the powerset) which is approximately 10^{16} and too many to exhaustively investigate. If the size of the subset is restricted to n then the number of subsets is

$$\binom{53}{n} \equiv \frac{53!}{(53-n)!n!}$$

For n = 4, this comes to 292,825 which is more manageable (though still a challenge given that for each subset a support vector machine model must be fitted, preferably optimized, and then tested with a test data set). For n = 3 the number of subsets reduces to 23,426 which was found to be a practical limit. (A recently acquired dual Xeon workstation which could run sixteen threads in parallel enabled the experiment to be performed in a couple of days.)

6.2.1 Improving the data sets

As already noted, the learning and test sets used in the earlier experiments included a few pathological conjectures which were disproved by saturation of the clause set. (That is a stage was reached where new inference steps would not produce any new nontrivial clauses so there is no possibility of finding the empty clause - see chapter 2.) These cases are erroneous results, and though there were not enough of them to invalidate the experimental results, advantage was taken of a rerun of the experiments to remove them from the data set.

Code was written to check the data set, remove pathological cases where the theorem was disproved, (generally in almost zero time), and then split the remaining data into a training set and a test set. It should be emphasised that this process did not involve any rerunning of the theorem prover or the corresponding timings. The data was unchanged, all that was altered was the removal of some invalid elements and a new division into sets of the remainder.

6.2.2 Enumerating the subsets

It is straightforward to calculate the number of, say, three feature subsets given fifty-three starting features. Some care, however, is required in coding to ensure that all unique subsets are examined and no duplicates (with differently ordered elements) are included. This was done by imposing an arbitrary ordering on the elements of each subset based on the feature number. Subsets were only included if the first element was less than the second element which in turn was less than the third. In coding terms this amounted to three nested loops with the inner loops beginning from the current value of their parent loop counter plus one.

6.2.3 Coding for parallel execution

Even restricting subsets to having no more than three features leads to a requirement to process approximately 25,000 cases including the smaller one and two feature subsets as well. Even on a 2.93GHz Xeon workstation, running one case took approximately 30 seconds. To run all cases in a serial fashion would have taken over 200 hours of CPU time. It was therefore worth coding the problem for parallel execution.

Fortunately the nature of the problem lent itself very well to parallel execution. Each subset could be assessed without reference to any of the others. The disk input and output proved to be a relatively minor part of the overall task and so did not lead to a bottleneck. Dividing the work up among sixteen threads was done by assigning a sequence number to each subset and setting a start and stop index for each thread.

Each thread was run in a separate directory, writing to separate files which could be merged at the end. Each thread ran a separate copy of the same executable code which was written to accept a thread number and total number of threads as command line parameters. The code was also written to be easy to restart should the long run be interrupted, the disk output file was opened and then closed for each output so as to leave files in a stable state and the use of subset sequence numbers allowed easy checking of completion.

6.2.4 Looking at large subsets

Just as the total number of subsets to be examined can be restricted by looking at only subsets with a few elements, the same effect can be obtained by looking only at large subsets. Large subsets are subsets where only a few elements are missing from the total set.

In parallel with looking at small subsets, large subset results were obtained but these are not reported here as they are essentially just a mirror image of the small subset results but with increased noise. The reason for this is that given only a few features are significant, removing these features will have a negative effect similar to the positive effect of selecting these features as the elements of the small subset but the effect is masked to some extent by the cumulative small contributions of the large number of less significant elements present. For example, if the small subset containing feature numbers 7 and 53 gives the best performance, then correspondingly the large subset consisting of all features but 7 and 53 should have the worst performance, each provides the same information. But the worst large subset is less well differentiated from the next worst than the best small subset is differentiated from the next best.

6.2.5 Analysis of three feature subset results

To be tractable given the large number of cases to be considered, the parameter γ was fixed at 10 rather than optimised. A later experiment varied γ over a large range for the optimal small subset of features (see figure 6.35), and this confirmed that the value of 10 is within a plateau of good, but not quiet optimal, values. As with earlier experiments, the kernel of the support vector machine was limited to the radial basis function.

For each possible one, two and three feature subset all five heuristic classifications were carried out, together with the heuristic zero classification, followed by using the resultant models for automatic heuristic selection.

The results were sorted lexicographically according first to total number of theorems proved and second according to the total time taken.

Figure 6.27 shows a plot of total number of theorems proved vs set rank (that is position of the set after ordering). The plot shows the results for all subsets, and it can be seen that there is a sharp initial drop after the first few subsets and then a much more gentle decrease before another sharp drop for the lowest ranked subsets. Figure 6.28 shows a more detailed plot of the highest ranked 500 subsets.

It is of interest to score individual features by their occurrence in higher ranked subsets. This is partly needed in order to select a reduced set of features to consider when investigating four feature subsets and partly to determine which features are of importance.

The scoring scheme is partly dependent on the choice of the number of subsets considered significant.

The first stage of obtaining feature scores was to sort the subset results lexicographically using a spreadsheet. Software was then written in C to read the sorted file and apply a cumulative score to each feature appearing in the top ranked subsets. The number of subsets considered significant was left as a parameter to be entered. If n subsets are considered significant then all members of the top subset would have n added to their individual scores, $n-1$ would be added to the scores of all elements of the second highest placed subset and so on. For example, if the first 100 subsets are considered significant and the top subset contains features 23, 45 and 48 then feature 23 would have 100 added to its score and similarly for features 45 and 48. If the next ranked subset contained features 36 and 45 then feature 36 would have 99 added to its score and feature 45 would have a further 99 added to its score making it 199. The features could then be ranked according to their resultant scores.

Figure 6.29 shows a plot of score versus feature number for the case of 100 significant subsets. Comparing this with figure 6.26 shows that prominent feature numbers are now quite different. The most significant features are 7, 17 and 52 and, though it is not directly shown in the plot, the best scoring subset consisted of just features 7 and 52 combined. Figure 6.30 shows the equivalent plot where the number of significant subsets has been extended to 3000 (note that in this case the plot is against feature *rank* rather than feature *number*). Similarly figure 6.31 shows the effect of extending the number of significant subsets to 5000.

The effect of increasing the number of significant subsets is to reduce the differential between good and bad features. The reason for this is that there are a large number of middling subsets for which many features produce similar results and by increasing the number of significant subsets more weight is given to these middling ones.

6.2.6 Partial extension to four feature subsets

To make it feasible to examine four feature subsets the total set of features needs to be reduced. A calculation was done to determine that the number of features needed to be reduced from 53 to around 33 or 34.

The problem is determining which features to use and which to exclude. The choice was done using the feature scoring method described in the previous section. The number of significant subsets was made large (five thousand), and the resultant scores plotted against feature rank as shown in figure 6.31. Note that feature rank is not the same as feature number. It can be seen that there is a step change in score after 27 features and again between 33 and 35 features. So by taking the best 34 features all useful features should be covered. Using the reduced feature set, all four feature subsets were tested and the results sorted in the same manner as the smaller subsets.

It was found that no four feature subset gave better results than the best three feature subsets.

From this result it was concluded that three features are sufficient. This conclusion is pragmatic. It may be the case that there is some combination of more than four features that lead to optimal results. But it can be assumed that this is unlikely as the three

Heuristic Number	*Number of Theorems Proved*	*Total Time Taken in Seconds*
1	1,514	162,029
2	1,352	177,530
3	1,424	168,593
4	1,421	169,598
5	1,339	176,959
Learned Heuristic Choice (Best Subset $\gamma = 48.05$*) (Includes H0 filtering)*	1,602	149,323

Table 6.1: Results for fixed heuristics.

feature subsets do better than the combination of all features and the large subsets of almost all features, so any optimal subset of features of size greater than three would also need to be negatively affected by the addition of more features. In other words, a plot of greatest number of theorems proved versus number of features in the subset would need to have more than one peak, which seems more improbable than a single maximum around three features.

Philosophical questions aside, it was deemed not worth continuing with the planned extension of subset size in combination with total feature reduction that is required to make the process tractable. (To extend the subset size to five features requires a reduction in total feature number to around 22 to 23, a very small increase in subset size requires a large decrease in the total number of features looked at.)

6.2.7 Results for fixed heuristics

As with earlier experiments, the benchmark for determining the effectiveness of the machine-learned automatic heuristic selection process was the number of theorems proved and total time taken by each of the heuristics individually. The test data set was different for this experiment so the results for the fixed heuristics were recalculated and are shown in table 6.1.

Table 6.1 also shows the results for the best subset after optimising the value of the parameter γ which is discussed in the next section.

Table 6.2 gives the comparison between the results for the full set of 53 features and the best subset of just 2 features under the same conditions of fixed γ of 10. In both cases H0 filtering is on, that is the H0 classifier is used to reject some conjectures without spending time seeking a proof (as discussed in the previous chapter). In addition the results for the full feature set without H0 filtering are shown, (note the increased number of theorems proved at the expense of a longer total time taken).

The presence of H0 filtering makes the direct comparison less clear. The small subset proves more theorems but takes more time, which indicates that the full feature set leads to too much weight being given to H0, (too many conjectures are rejected prematurely). But, even with no H0 filtering the full feature set proves fewer theorems than the small feature set with H0 filtering, *so the small, two feature, subset produces clearly better results*

Results with γ fixed at 10	*Number of Theorems Proved*	*Total Time Taken in Seconds*
Full Feature Set (H0 Filtering)	1,479	117,070
Full Feature Set (No H0 Filtering)	1,541	159,284
Best Subset (H0 Filtering)	1,589	152,867

Table 6.2: Results for 53 features and the best subset of just 2 features.

than using the full set of 53 features. Note also, that without H0 filtering the full feature set does better than any of the fixed heuristics which is consistent with the earlier results.

In summary, learning with just two features leads to an improvement over any individual heuristic in both number of theorems proved and total time taken. Especially when γ is increased to 48. Additionally, the two feature results are more balanced than those of the full feature set, avoiding too much emphasis on H0 classifications and most importantly, lead to a greater number of theorems proved.

6.2.8 Varying gamma for the best subsets

For radial basis function kernels there is a single parameter, γ, to be entered. Burges [13] discusses the effect of various values of γ which at either extreme can lead to either under-fitting or over-fitting.

In some of the earlier experiments γ had been optimised but computer resources had been limited and the steps in γ used were large and grew geometrically (*i.e.* γ was varied by doubling or halving rather than adding or subtracting a fixed step size). Additionally this γ optimisation had been done as part of the feature selection with the evaluation function for the optimisation expressed in terms of single classification results. The optimisation had to be restricted as it was carried out on every feature value sample, *i.e.* in the inner loop of the feature selection process.

By only looking at the top few subsets and with the availability of a much more powerful computer it was possible to vary γ over a wide range in very small steps and to see its effect on the whole heuristic selection process rather than just one heuristic classification result.

In the first stage of the experiment γ was varied from 0.01 to 50 for the top three feature subsets. The results are shown in figures 6.32 to 6.34. Looking at the plots two things were apparent. First, the results for the three subsets were very similar and secondly, stopping at 50 was a bit premature (the curves look like they are about to drop but have not yet done so). There is also an anomalous result at a γ value of approximately 22. This was not investigated in detail as to do so would require a detailed analysis of the code for SVMLight, how this interacts with the data and how starting points affect the built-in optimisation algorithm used.

The experiment was repeated, taking the value of γ up to 100 but only for the top feature subset. The results are shown in figure 6.35. A number of conclusions may be drawn. First, the fixed value of γ of 10, whilst not optimal, is within a plateau of good results. Second, low values of γ below about 8 or 9 are showing a drop off due to under-fitting (the radius of each centre is too large, leading to a lack of differentiation Burges [13]). Third, γ values above about 60 lead to over-fitting (models are tied too closely to individual samples within the training set leading to poor generalisation to the test set).

The optimal value of γ is just below 50 and gives results where over 1600 theorems are proved.

6.2.9 The best two features (7 and 52)

The optimal subset contains just two features, 7 and 52. The first is a static feature and the second is a dynamic feature. There was no bias in selecting these features during the experiments as the relationship between feature number and what the feature measured was not considered; that is though the description of each feature was recorded the records were not consulted during the experimental work. But, it turns out that features numbered 7 and 52 (which is dynamic feature number 38) are the same measure (the proportion of clauses containing purely negative literals), 7 being the proportion amongst the original axioms and 52 being that amongst the unprocessed clauses after running the proof search engine for a fixed number of clause selections.

Figures 6.36 to 6.47 show plots of each positive case for all heuristics, both learning and test sets, in terms of the two features 7 and 52.

It might be assumed that because these two features represent the same measure - the proportion of purely negative clauses - that they would be simply related to each other or even equal. But, as shown in figures 6.36 to 6.47, there is a fair degree of scatter when one is plotted against the other for all the samples in the data set.

6.3 Small subset results without H0

For completeness, the small subset experiment was repeated without H0 filtering being applied. That is, the heuristic selection process always selects a heuristic and attempts a proof. This leads to an increase in theorems proved at the expense of an increase in time taken.

6.3.1 Results without H0

Figure 6.48 shows the number of theorems proved for the different subsets plotted in rank order. Comparing this with figure 6.27, there is a small increase in the maximum number of theorems proved at the peak which is shown in more detail in figure 6.28. Beyond the best few subsets, the curve is much flatter than that with H0 filtering included. This is to be expected as even when the optimal heuristic is not selected, in many cases, a proof will still be found within the allowed time.

6.3.2 Feature scores without H0

A notable difference between the subset results for the case with H0 filtering and the case without is the subsets that rank first, and more generally the feature scores.

The feature scores based on the 100 top subsets are shown in figure 6.50 which should be compared with figure 6.29.

From the detail shown in figure 6.49 there is a drop in the number of theorems proved after the top four subsets. Figure 6.51 shows the feature scores based just on these top four subsets. It can be seen that feature numbers 10, 14, 15 are now most significant, these are also the members of the top subset. It should also be noted that features 7 and 52 still make an appearance.

6.3.3 The best three features without H0 (10, 14 and 15)

The top subset for the case without H0 filtering contains features 10, 14 and 15. Features 10 and 14 are static features and feature 15 is the first dynamic feature (see appendix A). Feature 10 is the average clause length and feature 14 is the average clause weight. Feature 15 is the proportion of generated clauses that have been kept at the point in the proof search at which the dynamic features are measured.

6.4 Random heuristic selection

In this dissertation so far the results from machine learning have been compared with those obtained from fixed heuristics. If heuristic selection can do better than the best results from fixed heuristics then the selection must be intelligent rather than random. The underlying assumption is that the random selection of heuristics will not do better than the best fixed heuristic. Strictly speaking, a random selection of heuristics may, by chance, pick an ideal set of heuristics and produce the best results possible but the probability of this happening is vanishingly small. To demonstrate that random heuristic selection will produce worse results, simulations were run to determine the statistical distributions. In these simulations heuristics were selected at random, with the random weighting set on the basis of the proportion of the learning set for which that heuristic was best. (In the case of H0, the weighting was based on the proportion of conjectures in the learning set that were unproven.)

Simulations were run for two cases, one with H0 filtering and one without. The standard C library function rand() was used and 1,000,000 trials performed to get smooth distributions[1]. Figures 6.52 and 6.53 show the distributions for the case with H0 filtering. Figures 6.54 and 6.55 show the case without H0 filtering. Normal distributions with the same mean and standard deviations are plotted alongside the simulated results. The distributions are close to normal but slightly skewed. Table 6.3 summarises the results in terms of means and standard deviations.

[1]Knuth[40] devotes a whole chapter of his Art of Computer Programming to random numbers. This experiment did not attempt such sophistication and it may well be that 1,000,000 samples exceeds the length of the pseudo-random sequence associated with the library rand() routine, but this should not materially affect the simulation results.

Random Heuristic Selection	*Mean Number Proved*	*standard deviation*	*Mean Time Taken*	*standard deviation*
H0 Filtering	827	20	98,145	2,116
No H0 Filtering	1,427	12	169,337	1,126

Table 6.3: Random selection results.

The simulated results for random heuristic selection are significantly worse than the results for the best fixed heuristic which in turn are worse than the results for the learned heuristic selection process. The difference is many times the standard deviation so the improvement due to the selection process is not down to random chance.

6.5 Summary and conclusions

Though a large number of features were examined it turns out that only two or three are required for effective learning in the context of heuristic selection. In fact the best subset, for the case where H0 filtering is applied, contains just two features which correspond to static and dynamic versions of the same measure, the proportion of clauses that contain only negative literals.

The fact that effective learning occurs with just two features, (which are static and dynamic aspects of a single feature), could only be found by carrying out the learning process. This is in contrast to the approach that was used in the existing auto mode of E where a range of binary features were used to split trial conjectures from the TPTP library into classes prior to heuristics being tested on each class. This is discussed more fully in the next (conclusions) chapter of this dissertation.

These results were obtained when considering the whole heuristic selection process rather than individual classification experiments for each heuristic separately and the optimal features differ from those obtained in the cruder earlier experiments which were restricted to separate classification results. The optimal features also differ where H0 filtering is not applied, though the two features from the best subset with H0 filtering also appear among the best subsets without H0 filtering. This variation in which features appear most important indicates that the selection of features to be used should be tailored to the problem.

The results were obtained with the parameter γ fixed at 10 but studying the effect of varying γ shows that 10 is a reasonable value, though the very best results are obtained with γ set to just below 50 (there is a plateau of reasonable γ results stretching between values of around 9 or 10 which are large enough to avoid under-fitting up to values of around 60 where over-fitting effects are seen to begin).

The fact that the two different sets of experiments produced a different overall ranking of features was a surprising result. Given that the overall purpose of the machine learning is to select heuristics, the second set of experiments, in which the score was based on the end result of theorems proved and time taken, provides the better data. But with the second experiment, there was a difference in feature ranking between the case with H0 filtering and that without. This raises questions which given extended resources of both research and computer time, would be interesting to answer. In particular, should the feature set used be tailored to each heuristic separately? To some extent this was looked at in the first set of experiments but then the results were combined and the measure used was not ideal (that of the proportion of correct classifications - the margin value itself is also of importance as it is this that decides the choice between heuristics). To answer the question definitively the measure of goodness would need to be the total number of theorems proved, (and time taken), whilst each machine learned classifier would be allowed to be based on a different three feature subset. Unfortunately a brute force approach to this would require looking at not just 25,000 cases but instead looking at $(25,000)^5$ cases!

It would make sense for different features to matter for different heuristics. Given that each heuristic works best on different conjectures, the purpose of the features is to define the right sort of conjectures for the heuristic and such measures may well be conjecture type dependent.

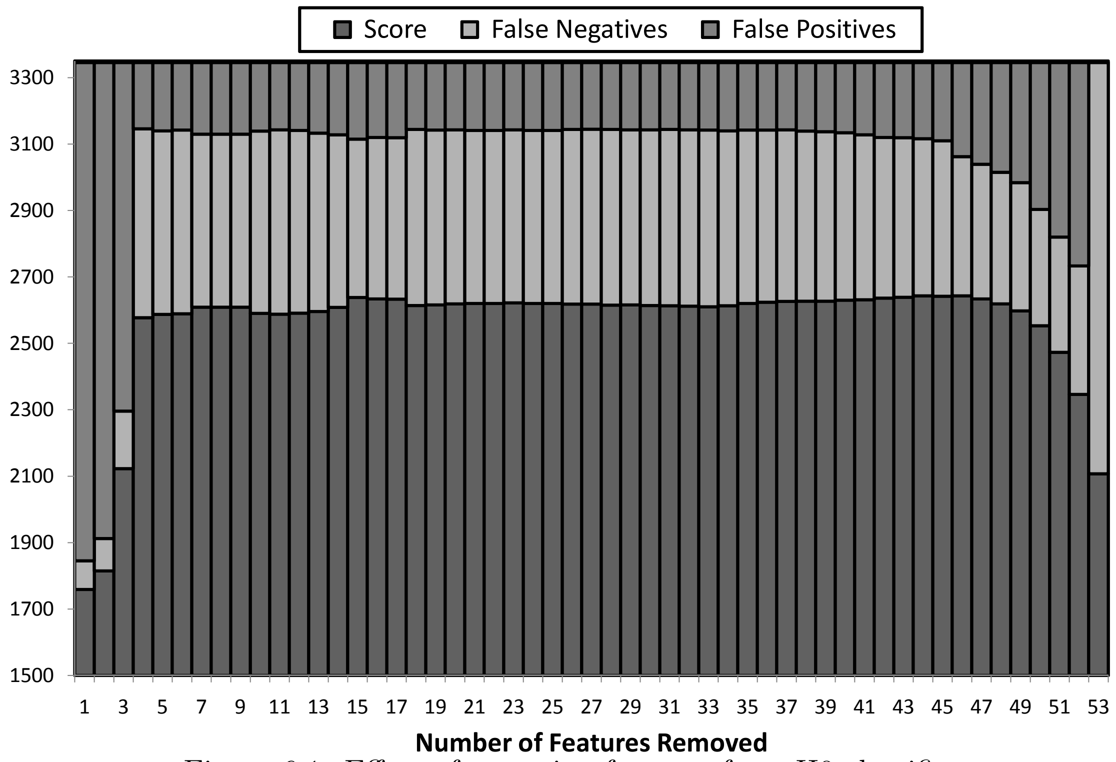

Figure 6.1: Effect of removing features from H0 classifier

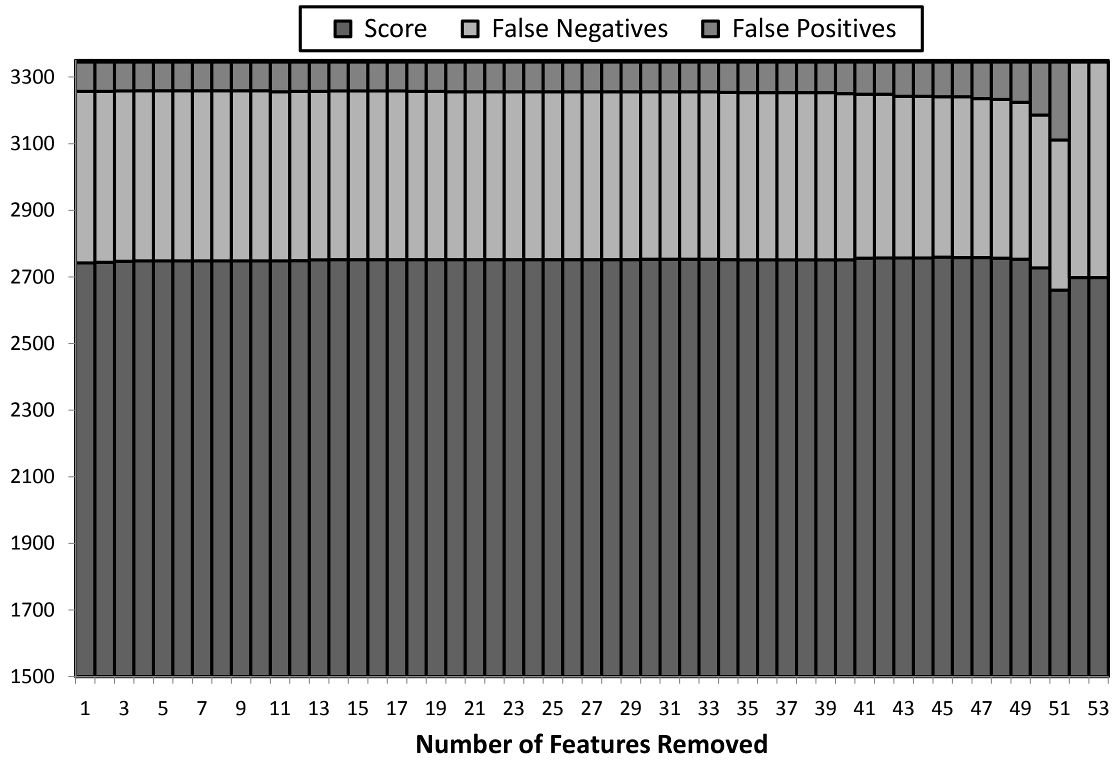

Figure 6.2: Effect of removing features from H1 classifier

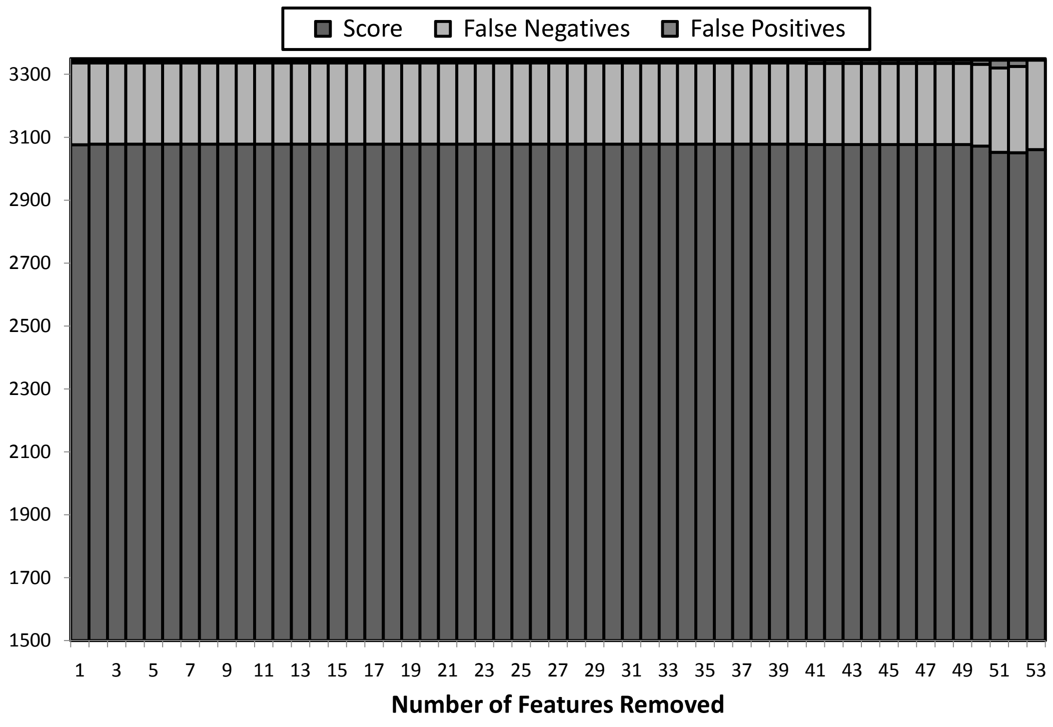

Figure 6.3: Effect of removing features from H2 classifier

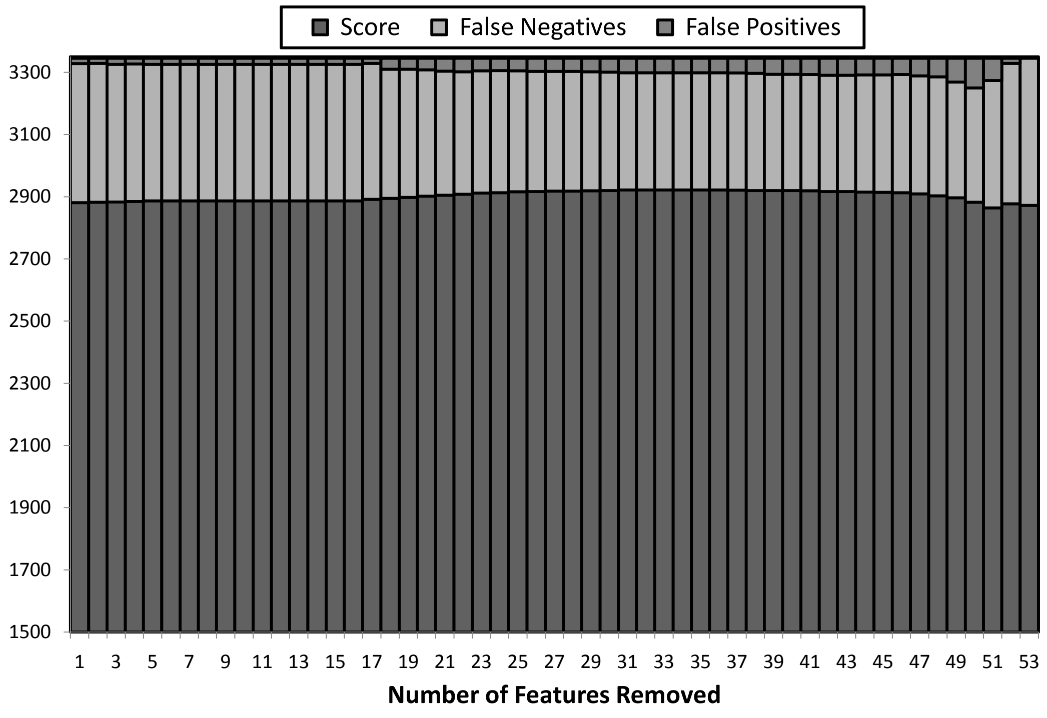

Figure 6.4: Effect of removing features from H3 classifier

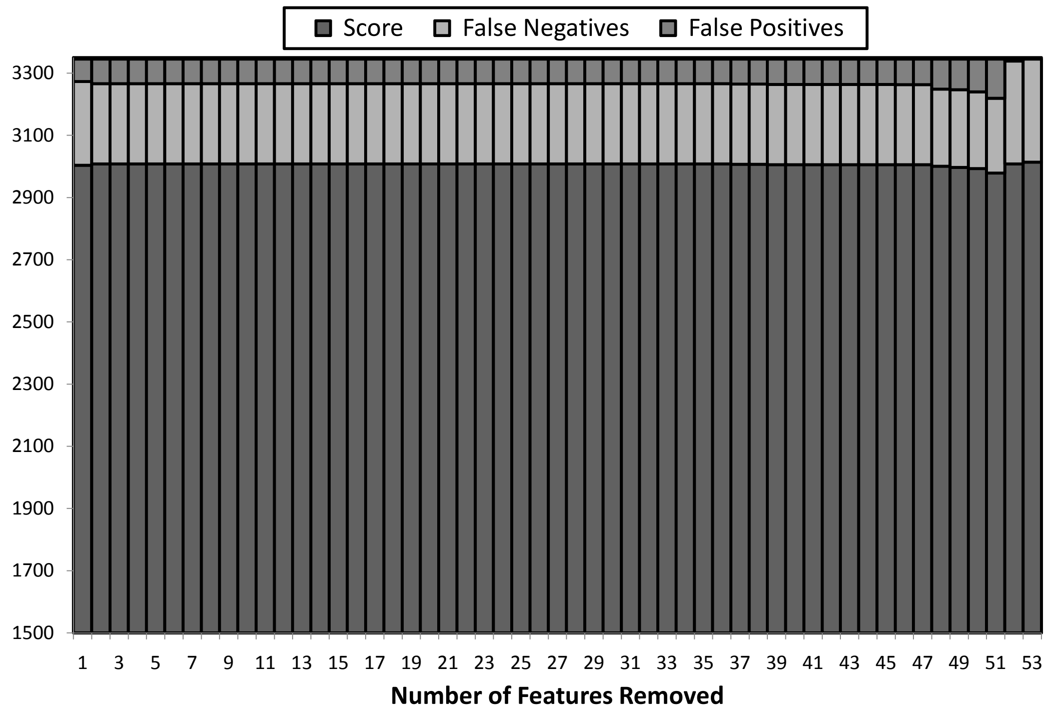

Figure 6.5: Effect of removing features from H4 classifier

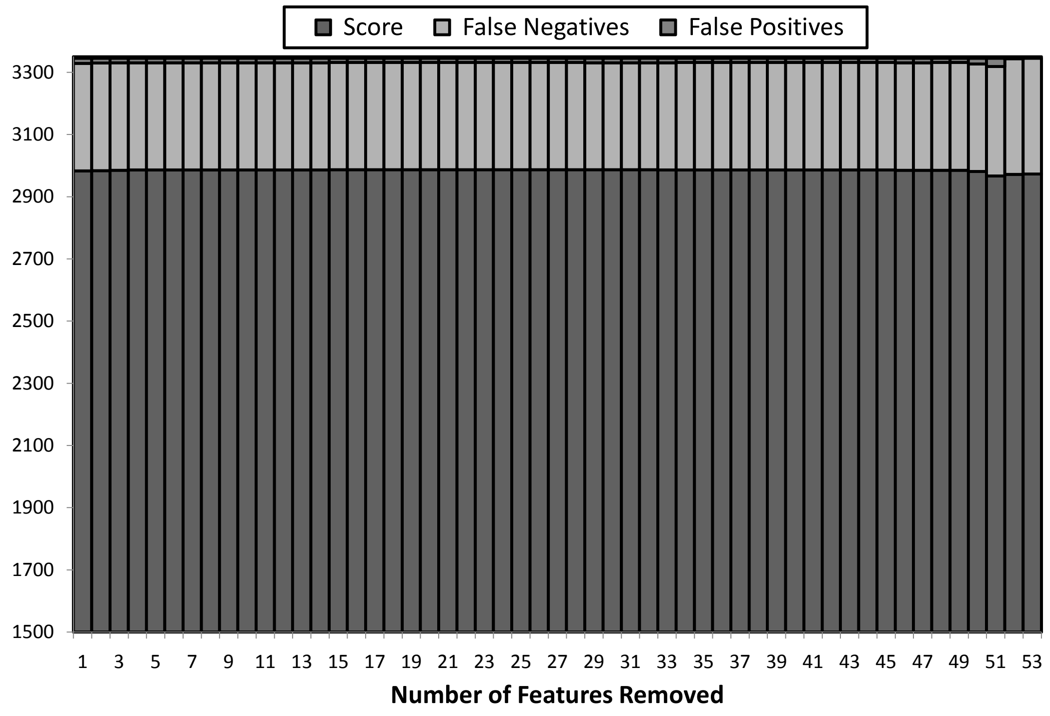

Figure 6.6: Effect of removing features from H5 classifier

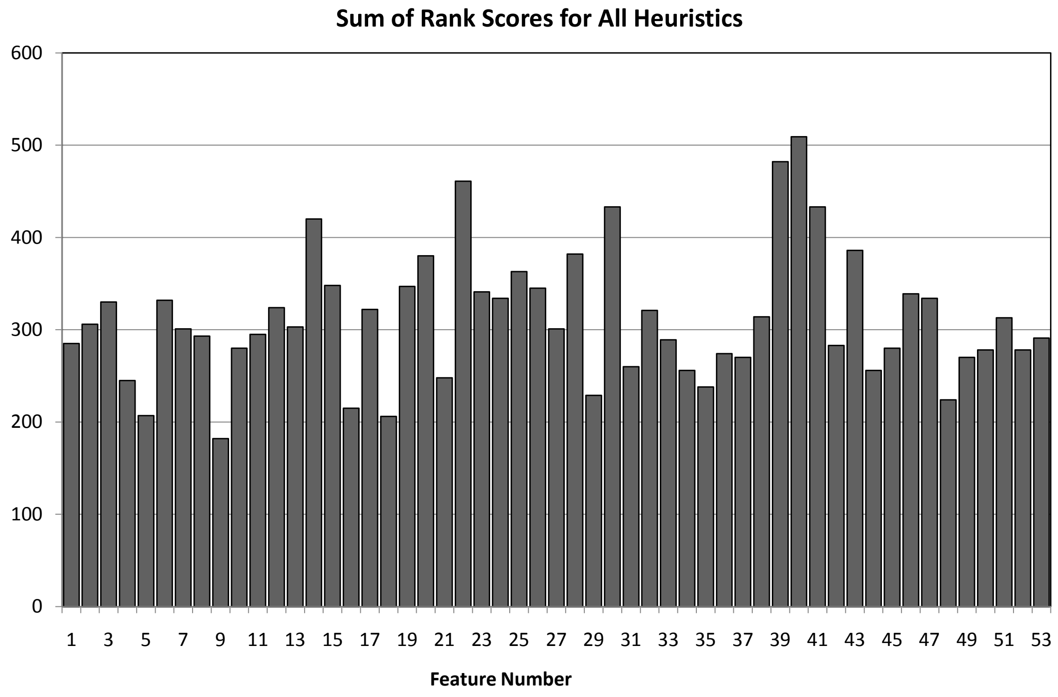

Figure 6.7: Scores for each feature summed over all heuristic classifiers

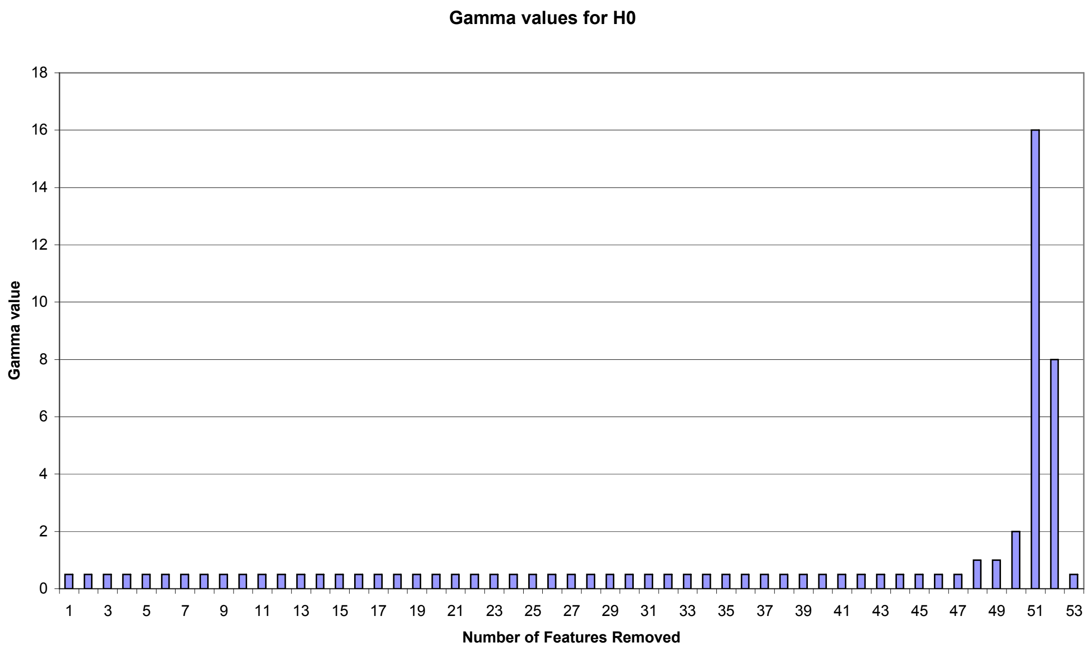

Figure 6.8: Gamma values for H0 on second run

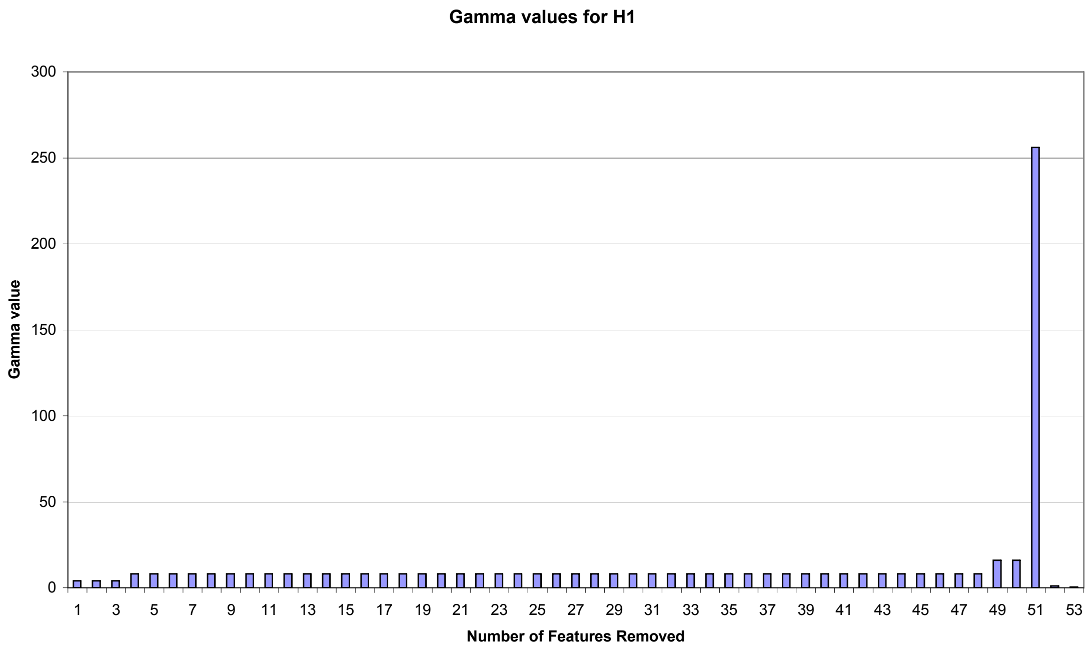

Figure 6.9: Gamma values for H1 on second run

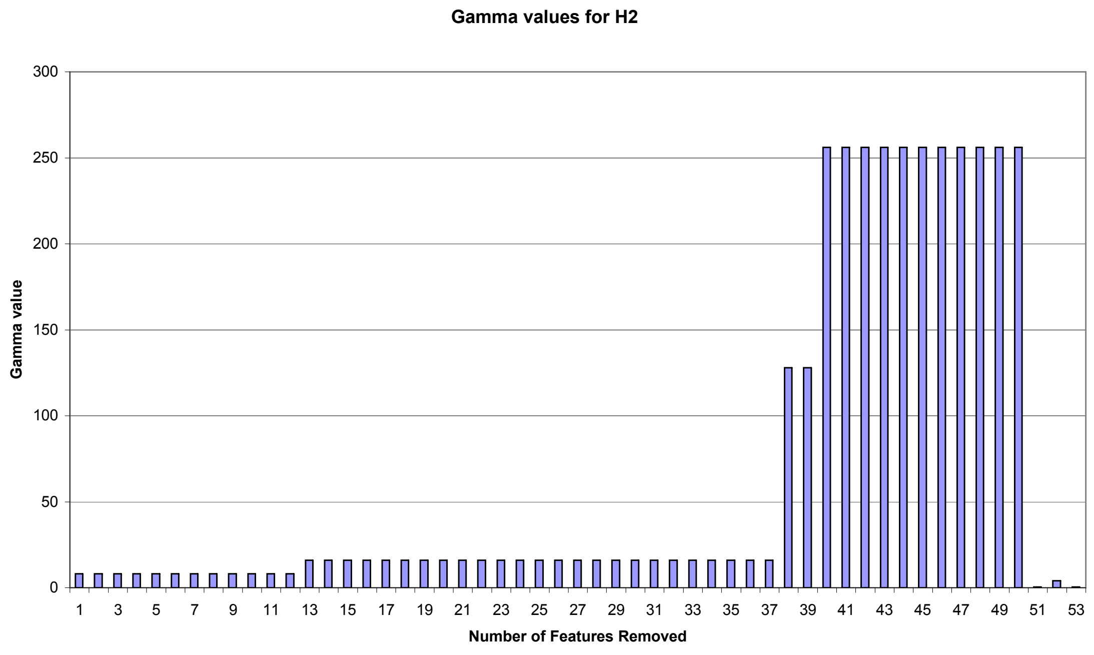

Figure 6.10: Gamma values for H2 on second run

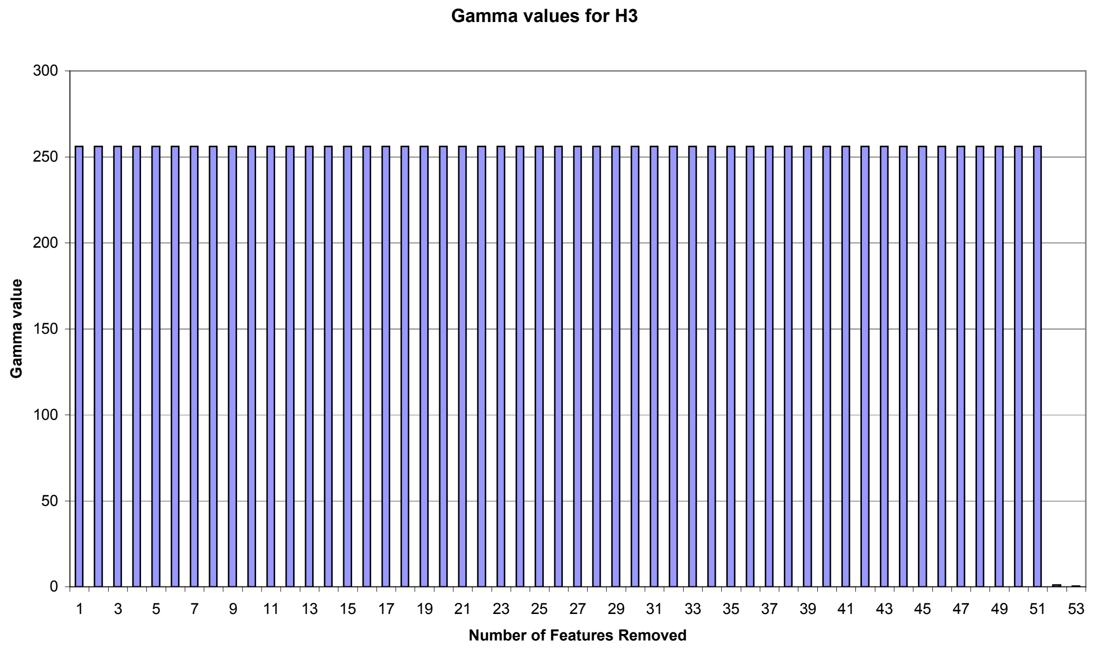

Figure 6.11: Gamma values for H3 on second run

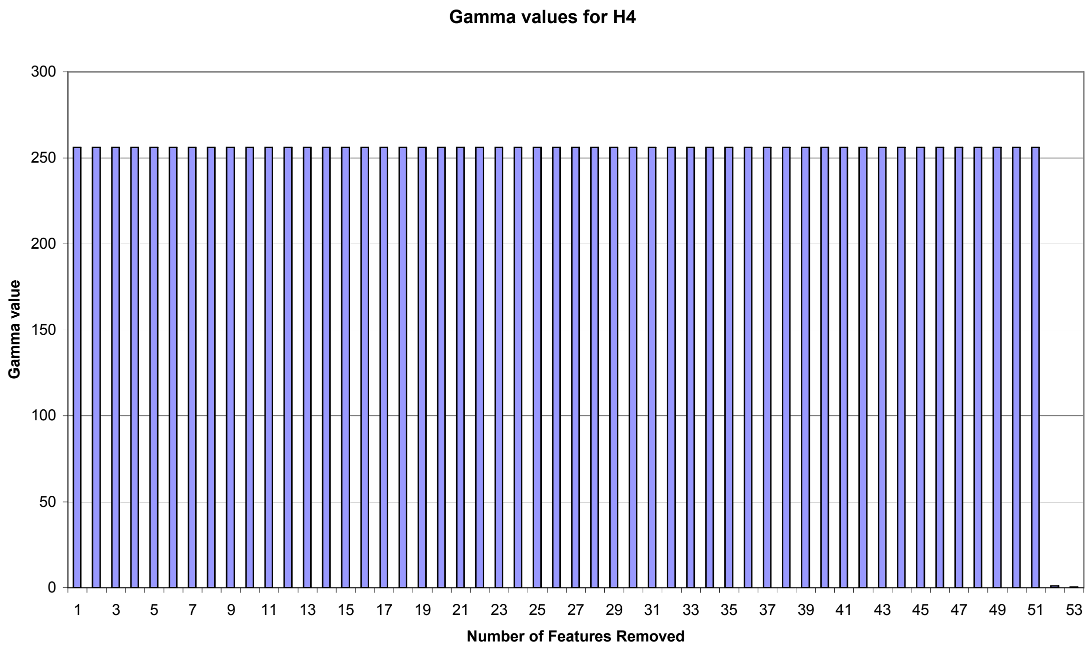

Figure 6.12: Gamma values for H4 on second run

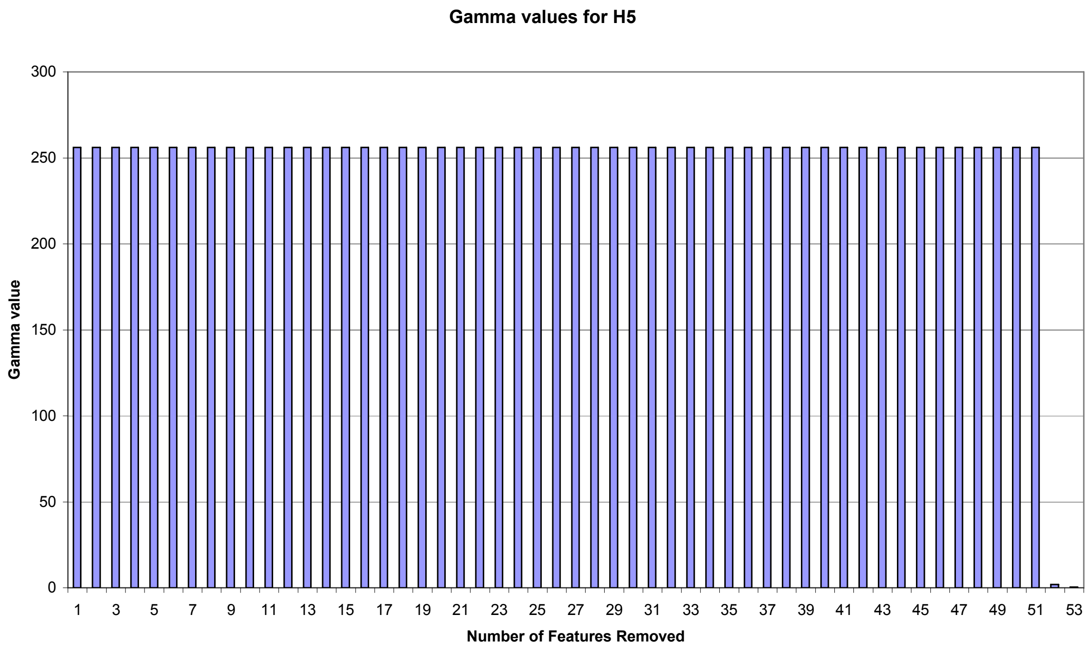

Figure 6.13: Gamma values for H5 on second run

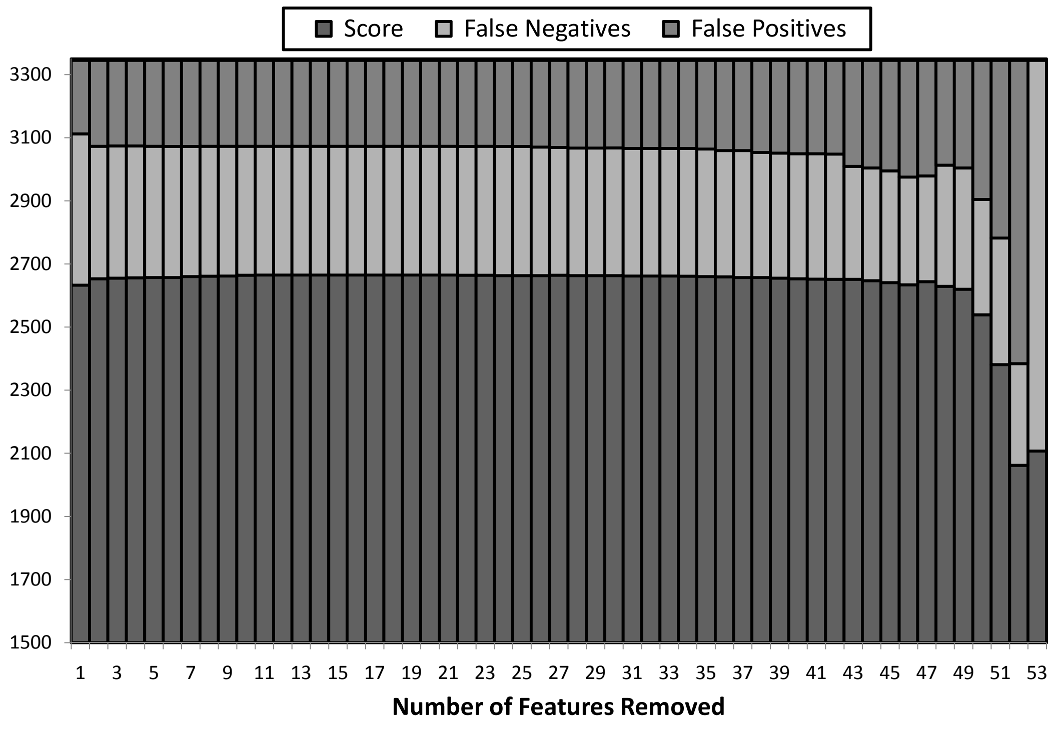

Figure 6.14: Effect of removing features from H0 classifier on second run

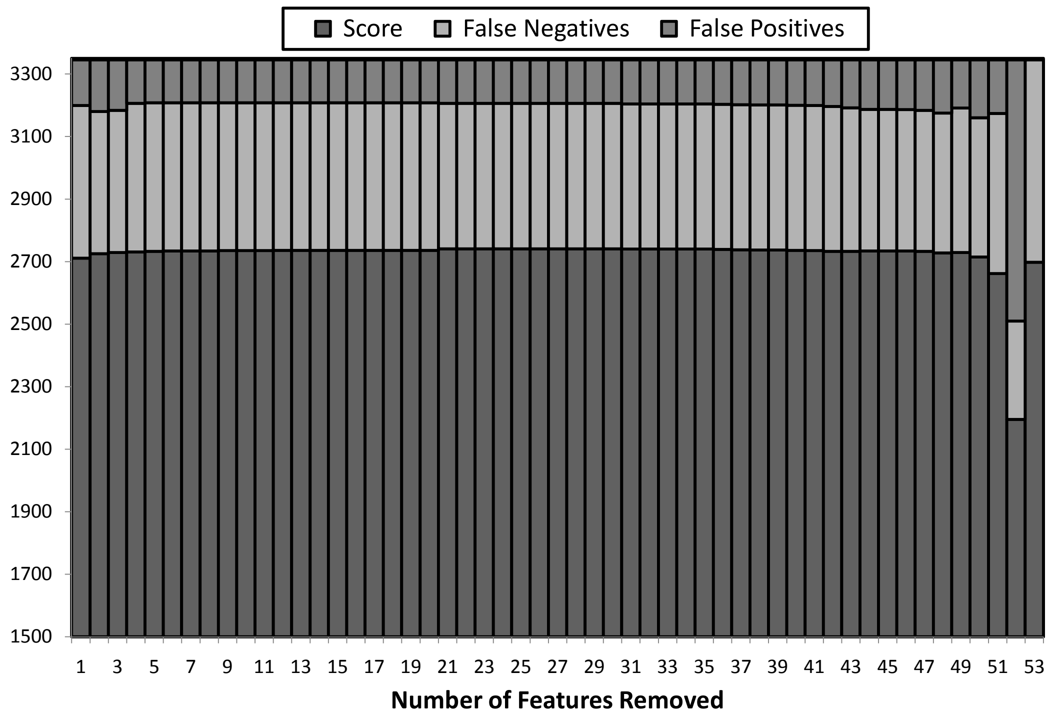

Figure 6.15: Effect of removing features from H1 classifier on second run

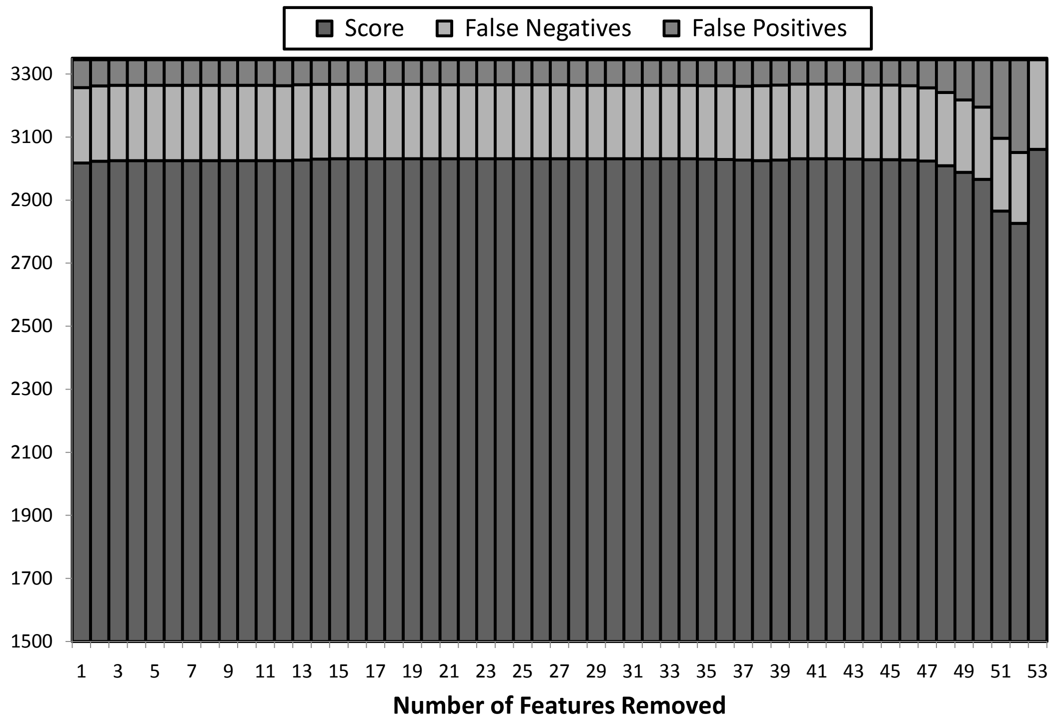

Figure 6.16: Effect of removing features from H2 classifier on second run

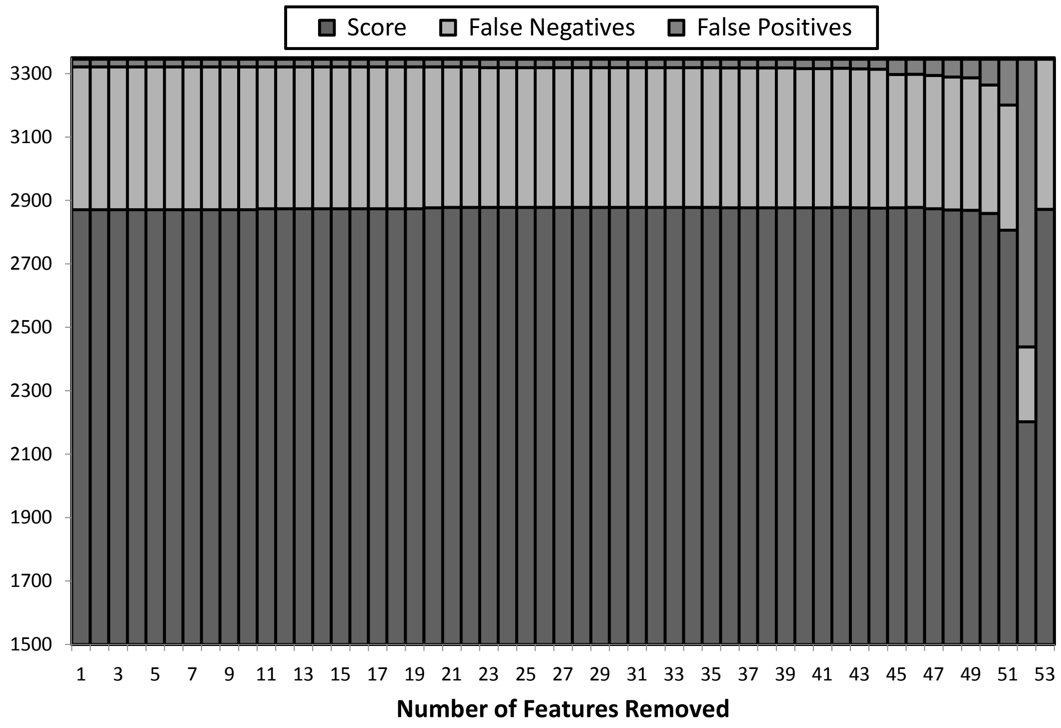

Figure 6.17: Effect of removing features from H3 classifier on second run

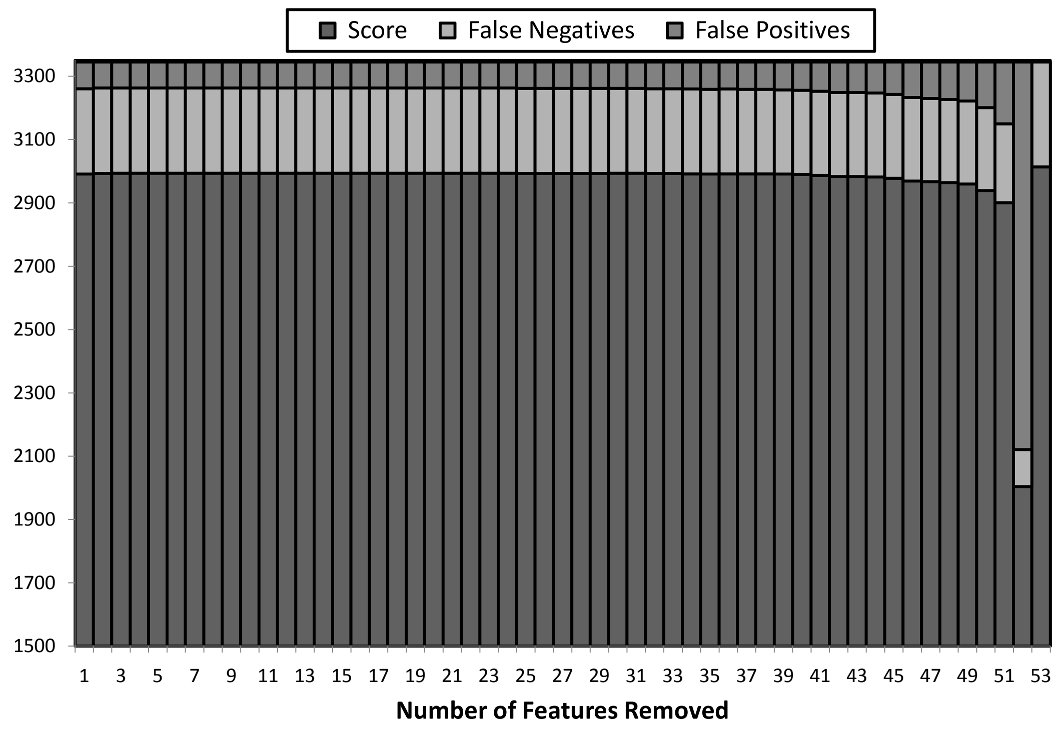

Figure 6.18: Effect of removing features from H4 classifier on second run

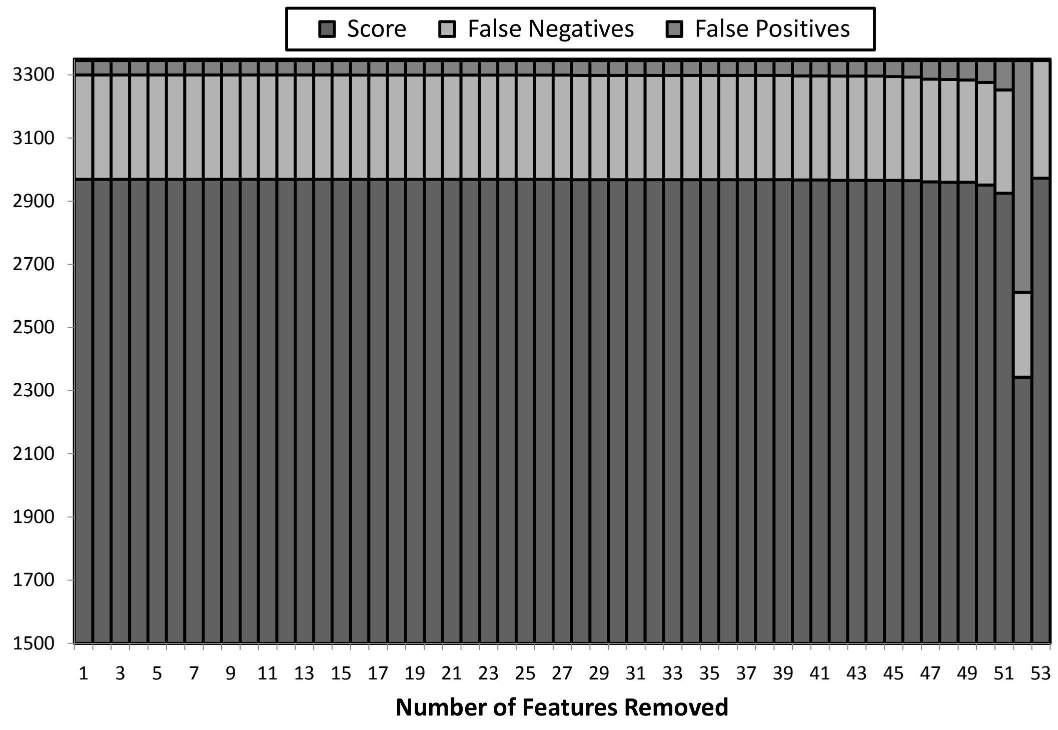

Figure 6.19: Effect of removing features from H5 classifier on second run

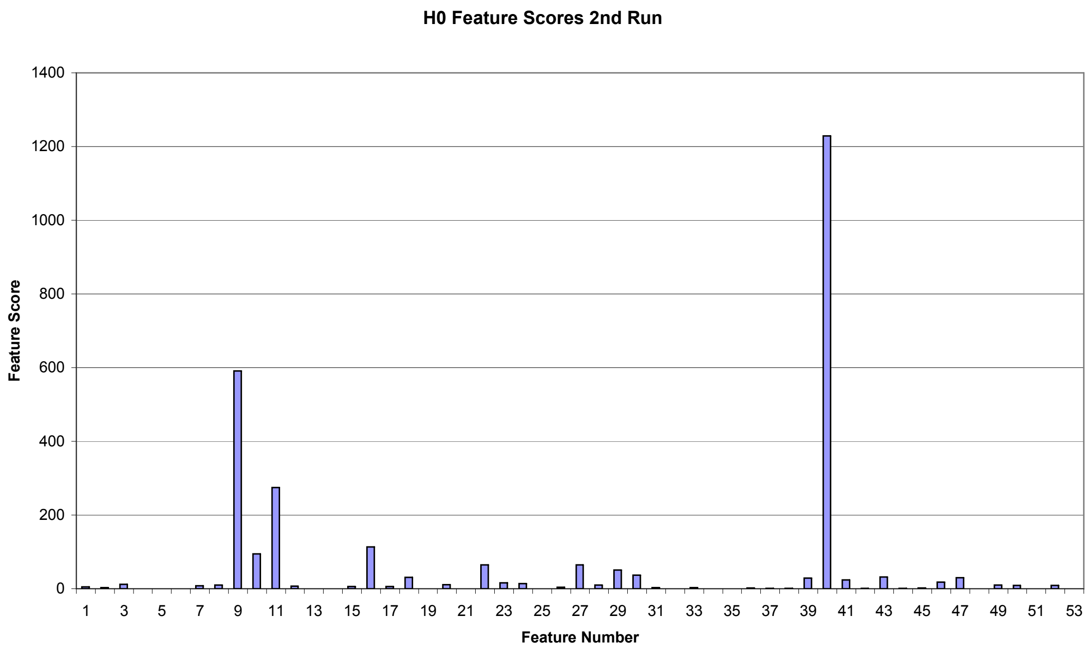

Figure 6.20: Feature scores for H0 on second run

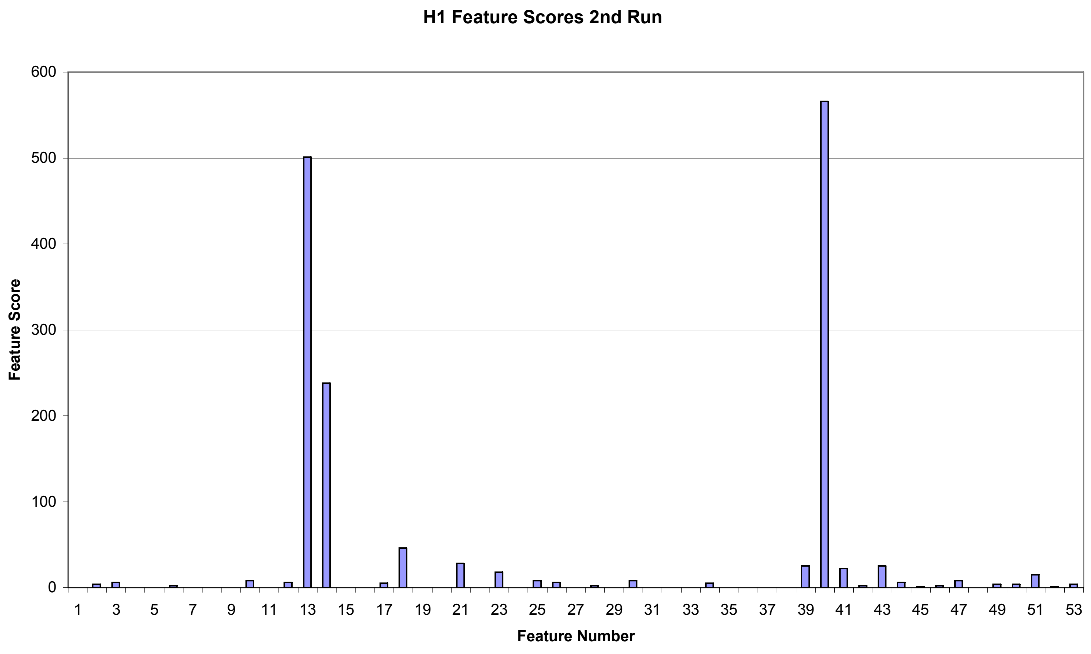

Figure 6.21: Feature scores for H1 on second run

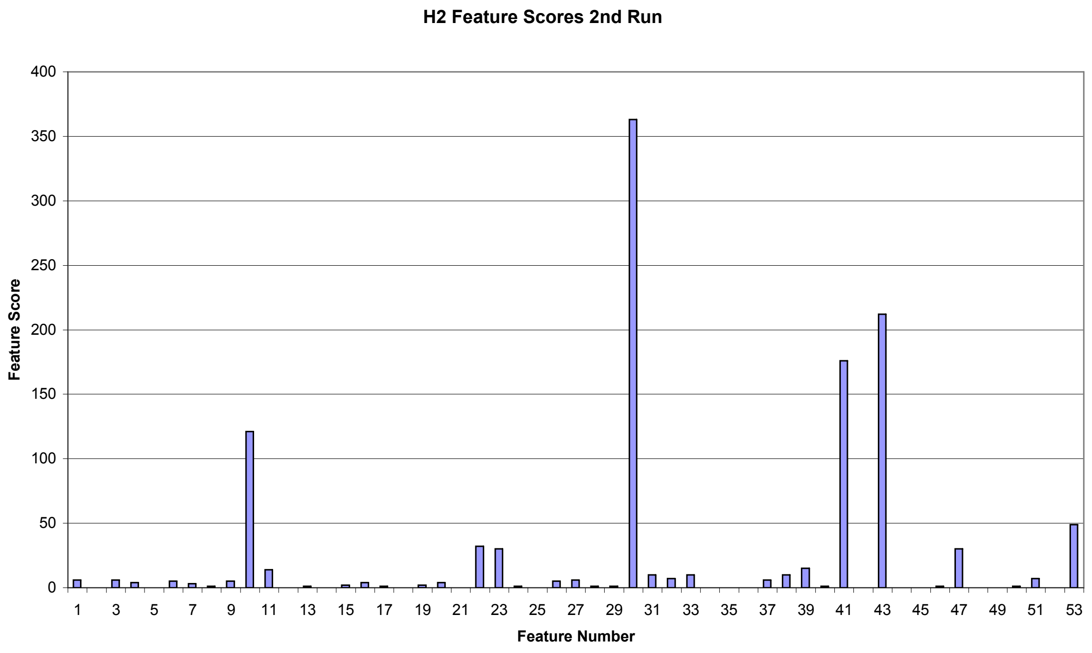

Figure 6.22: Feature scores for H2 on second run

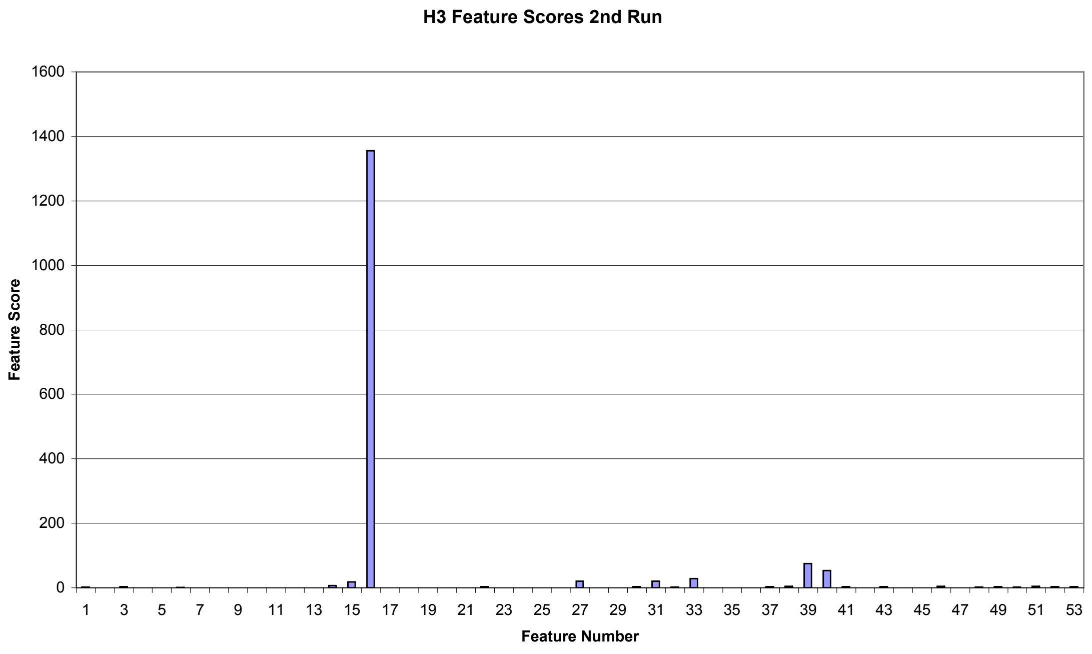

Figure 6.23: Feature scores for H3 on second run

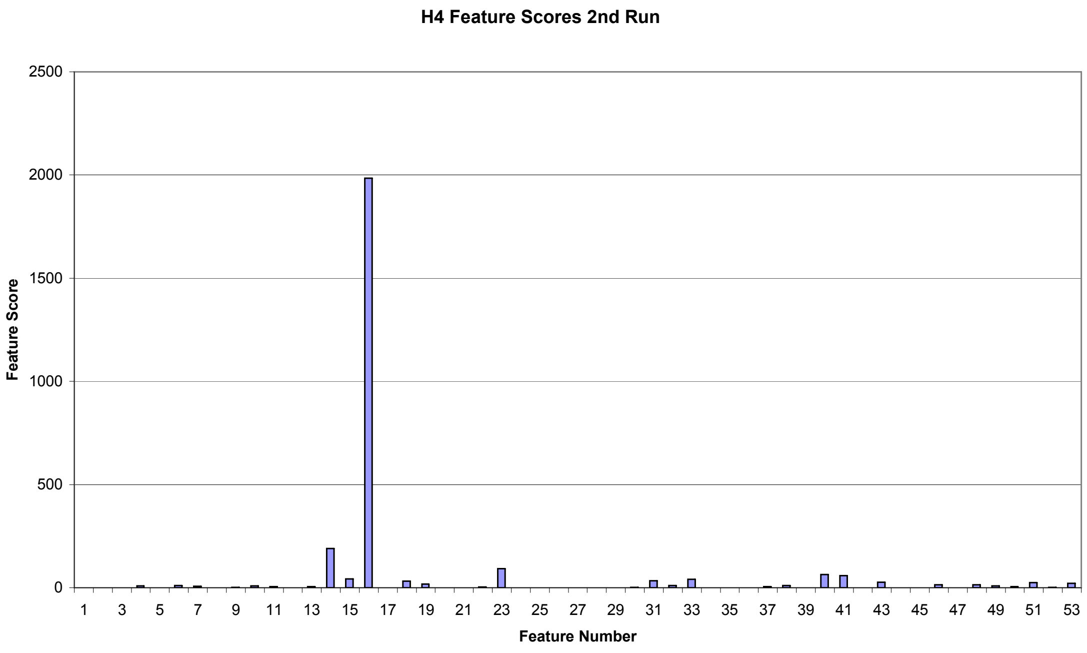

Figure 6.24: Feature scores for H4 on second run

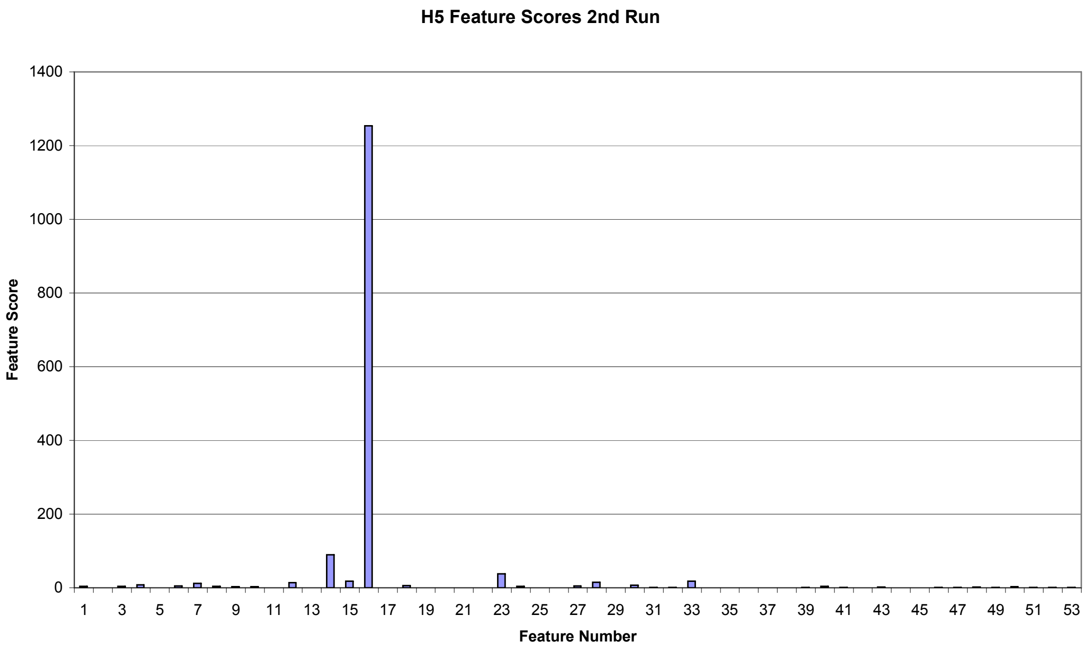

Figure 6.25: Feature scores for H5 on second run

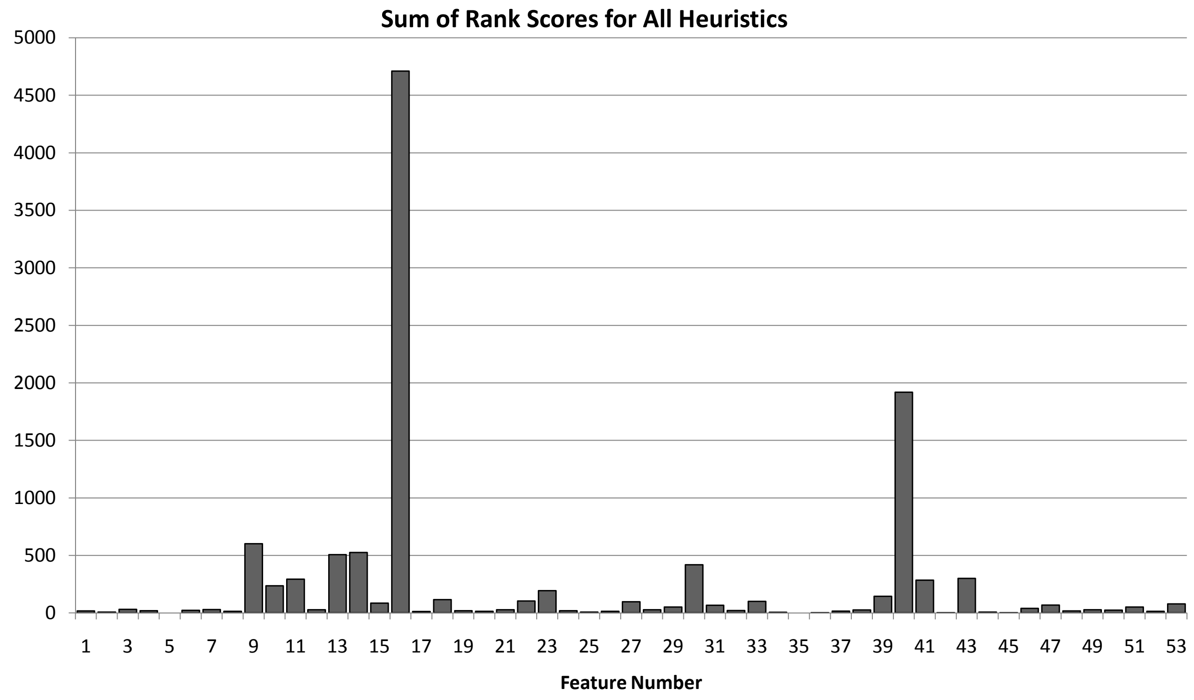

Figure 6.26: Scores for each feature summed over all heuristic classifiers for second run

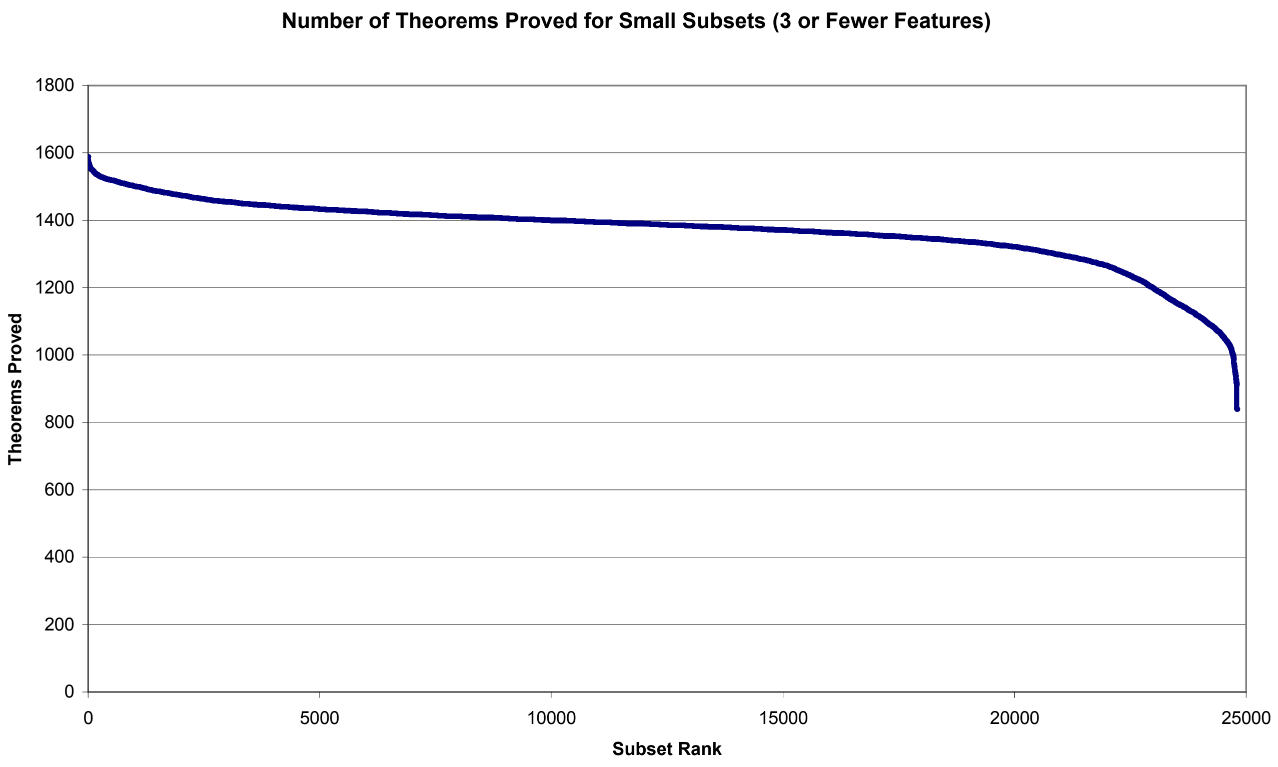

Figure 6.27: Theorems proved vs subset rank for small subsets of up to three features

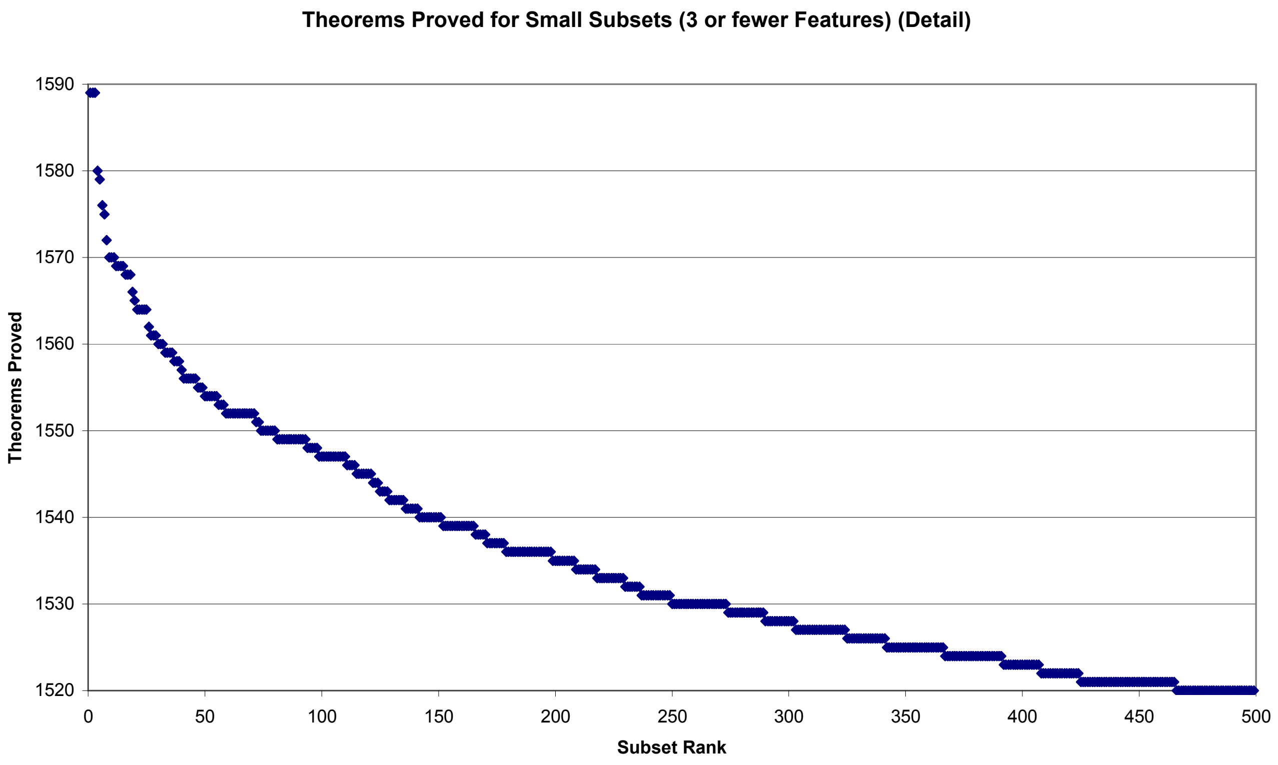

Figure 6.28: Theorems proved vs subset rank for small subsets of up to three features

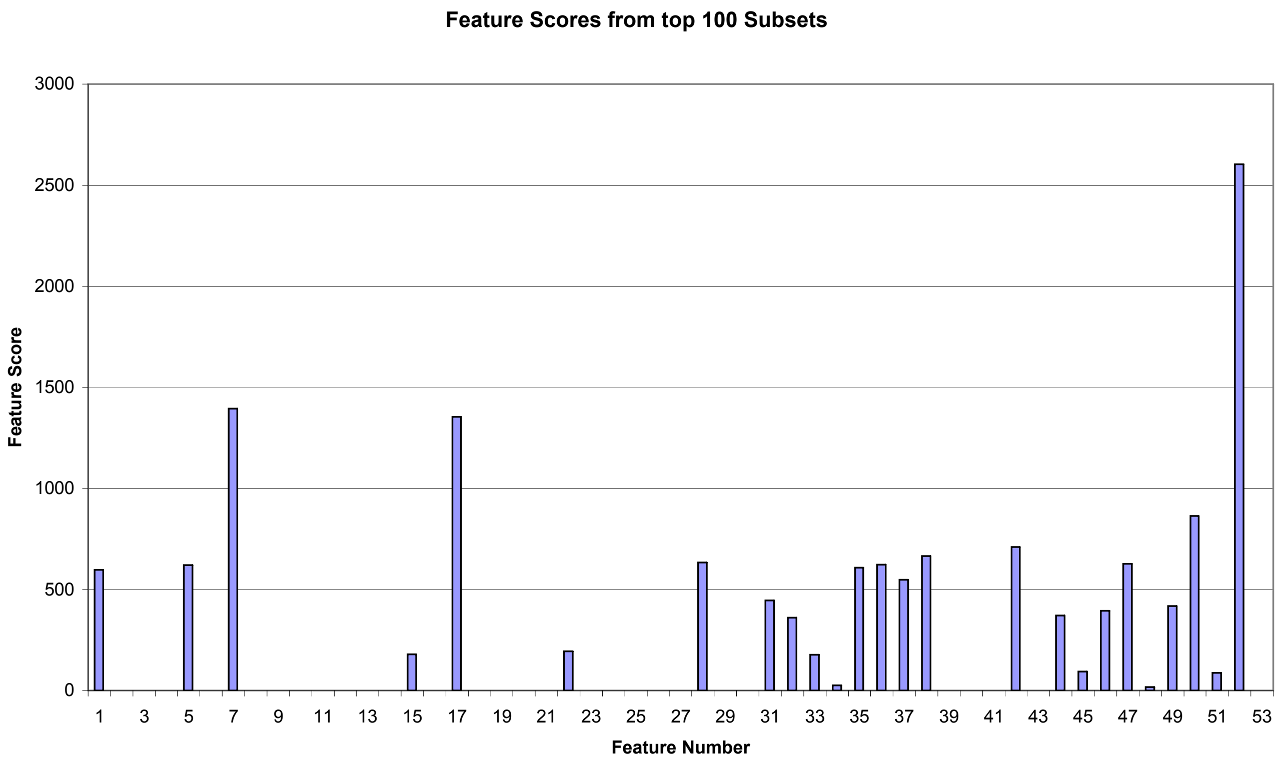

Figure 6.29: Scores for each feature based on 100 best subsets

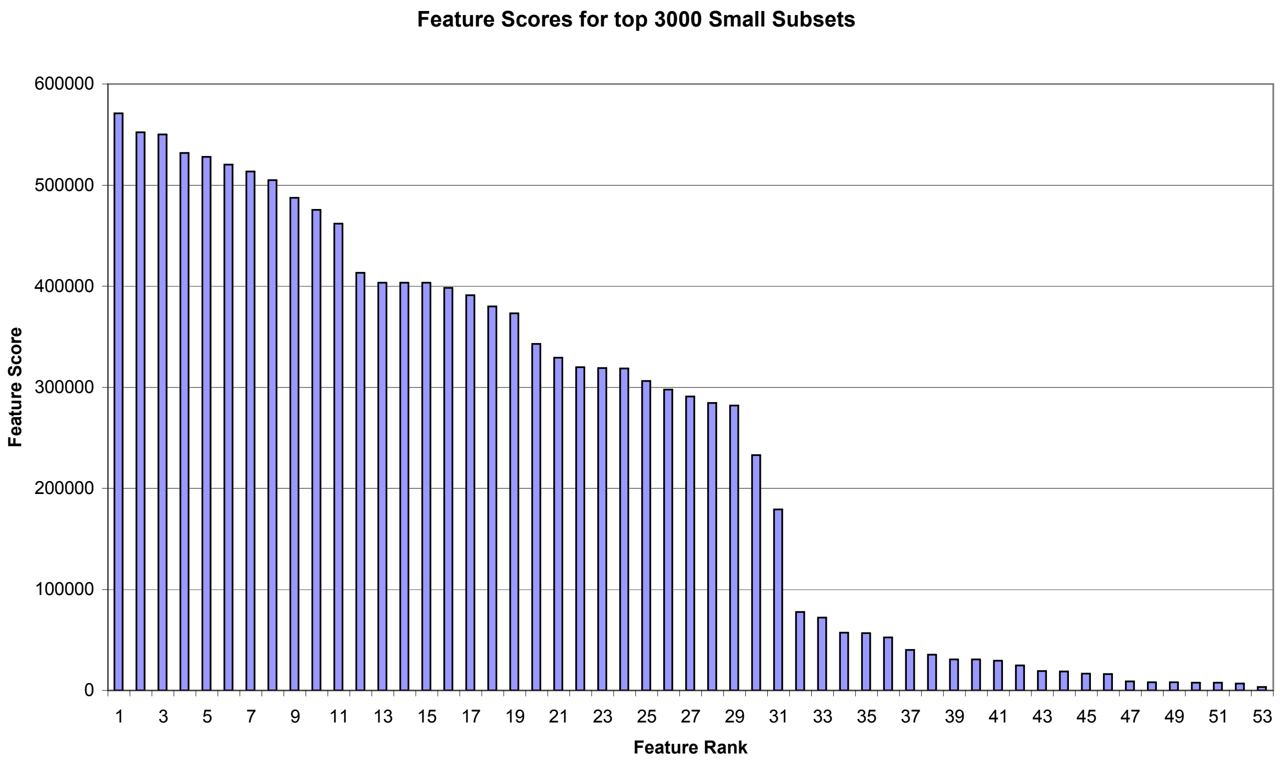

Figure 6.30: Scores vs feature rank (not number) based on 3000 best subsets

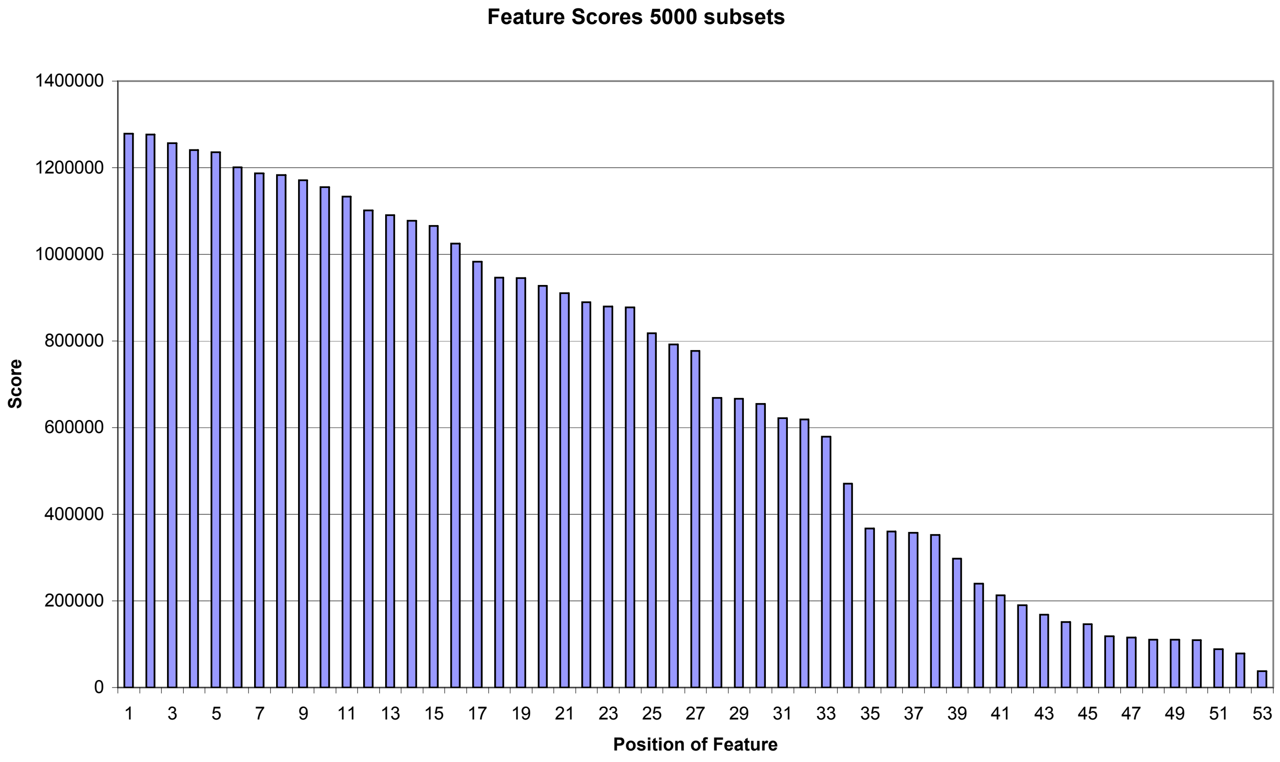

Figure 6.31: Scores vs feature rank (not number) based on 5000 best subsets

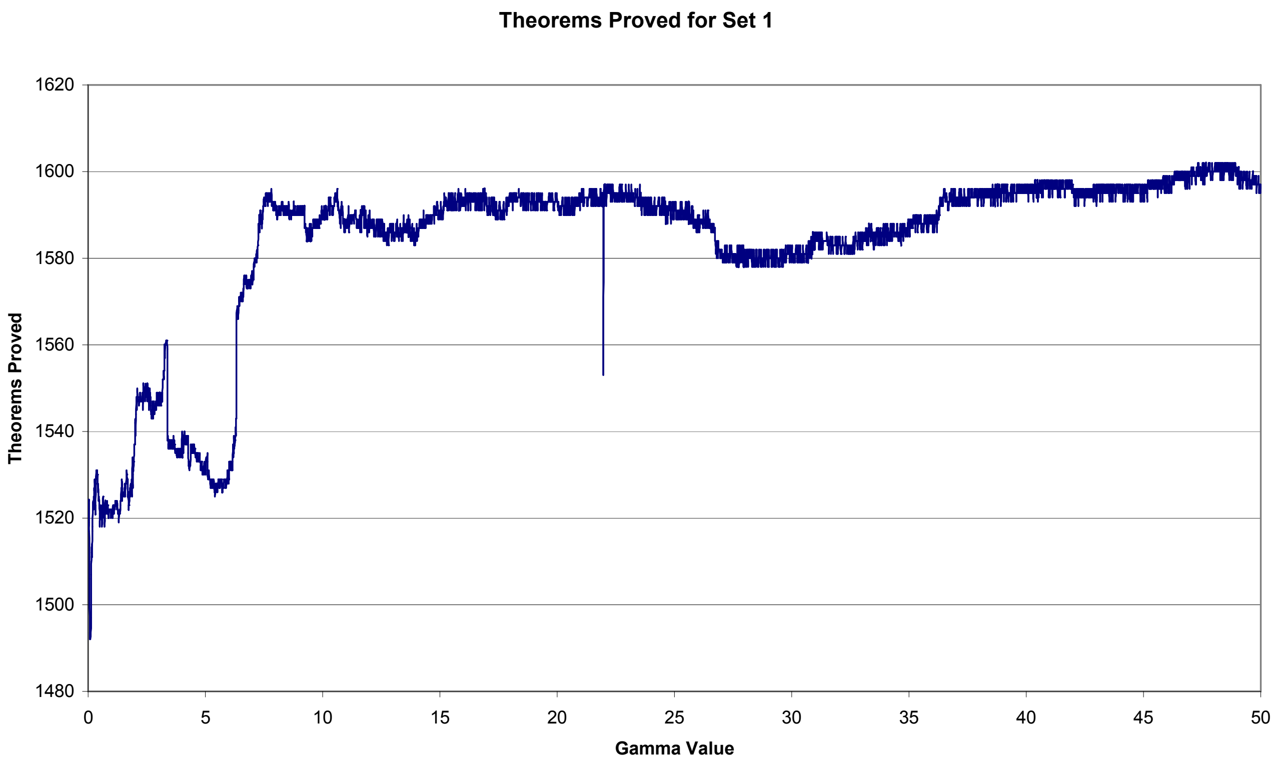

Figure 6.32: Theorems proved versus gamma value for best feature subset

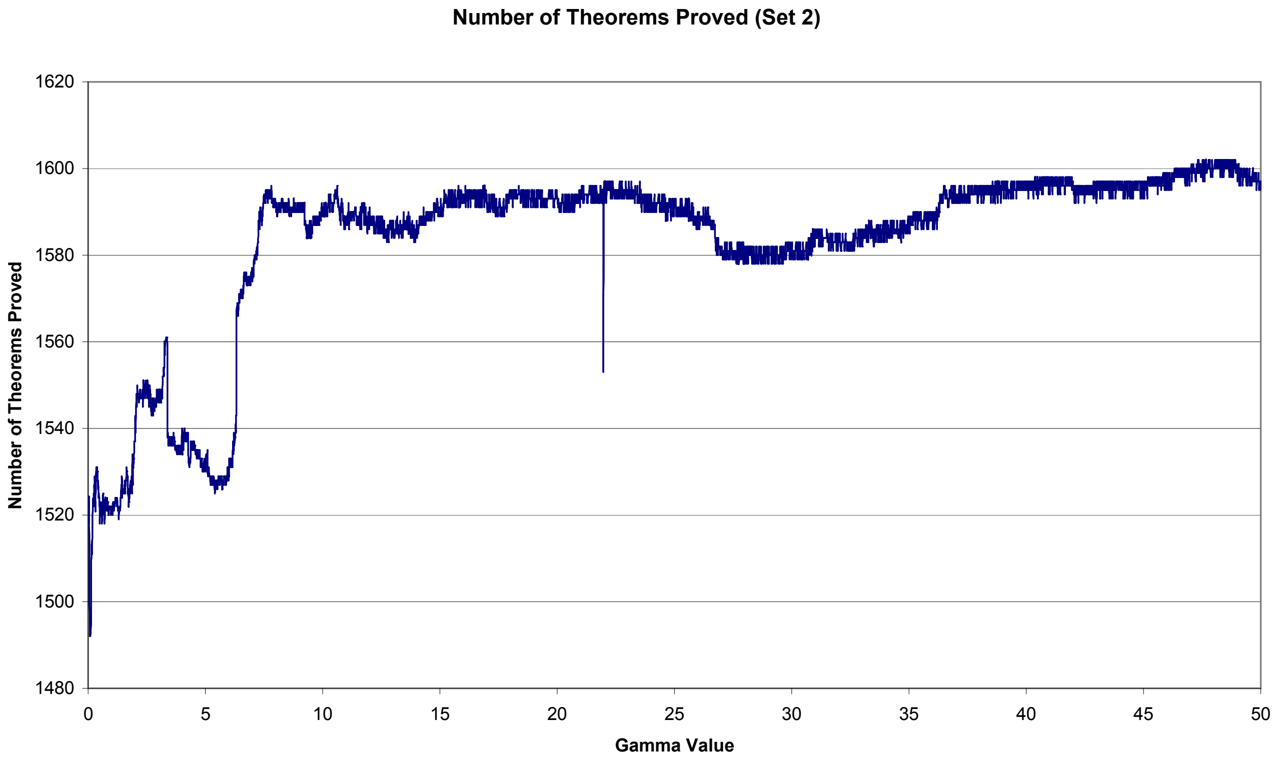

Figure 6.33: Theorems proved versus gamma value for second best feature subset

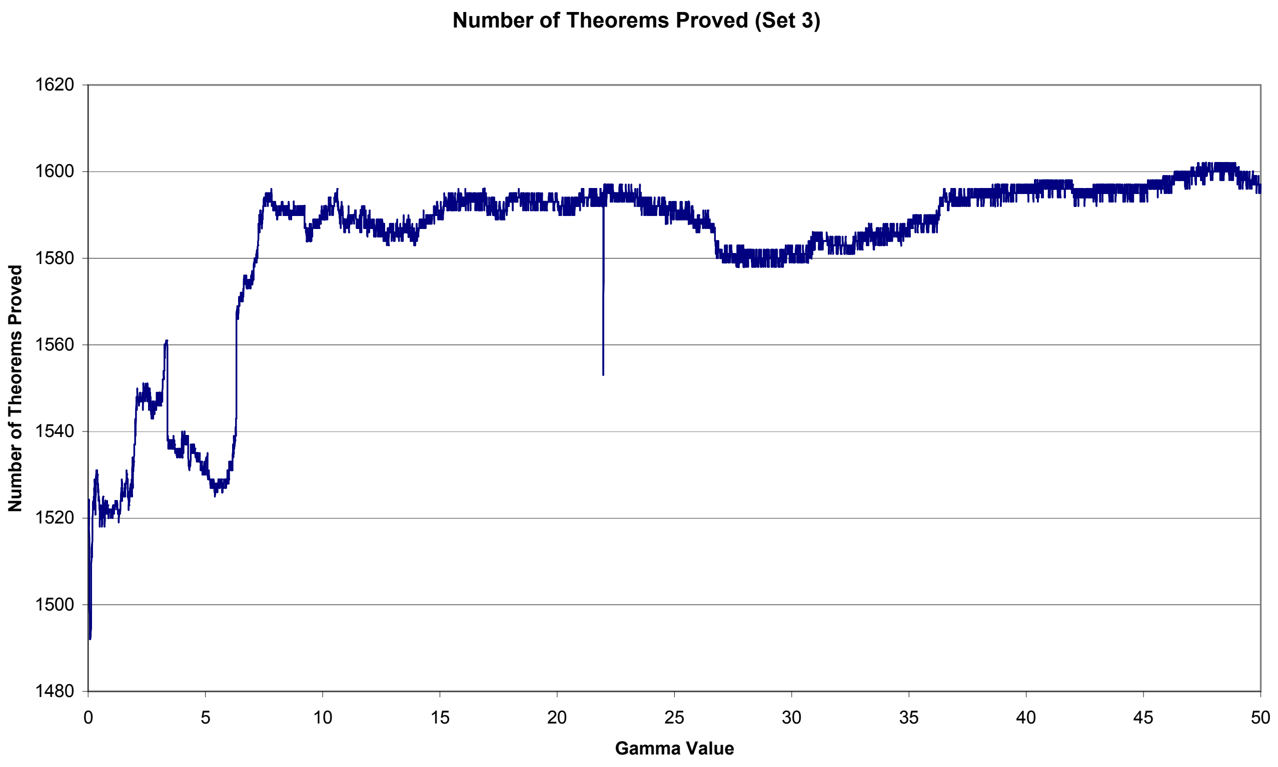

Figure 6.34: Theorems proved versus gamma value for third best feature subset

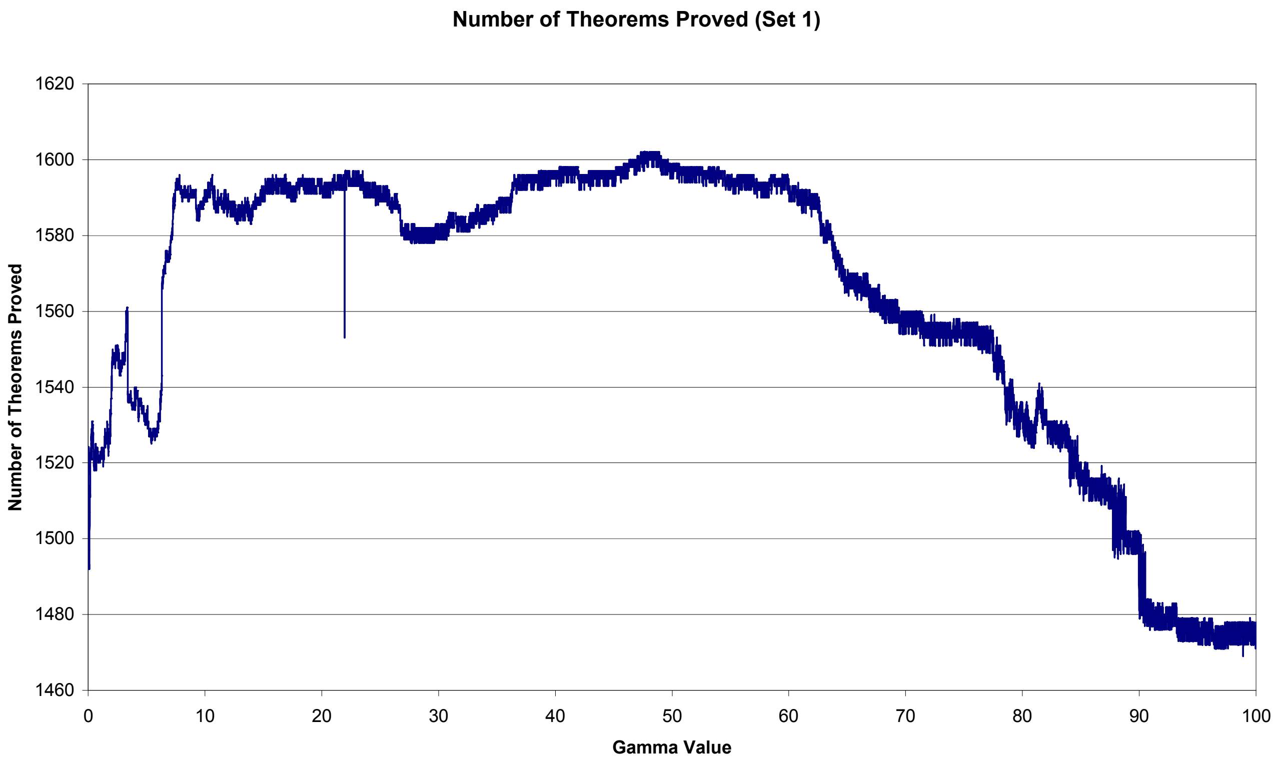

Figure 6.35: Theorems proved versus gamma value extended to 100 for best feature subset

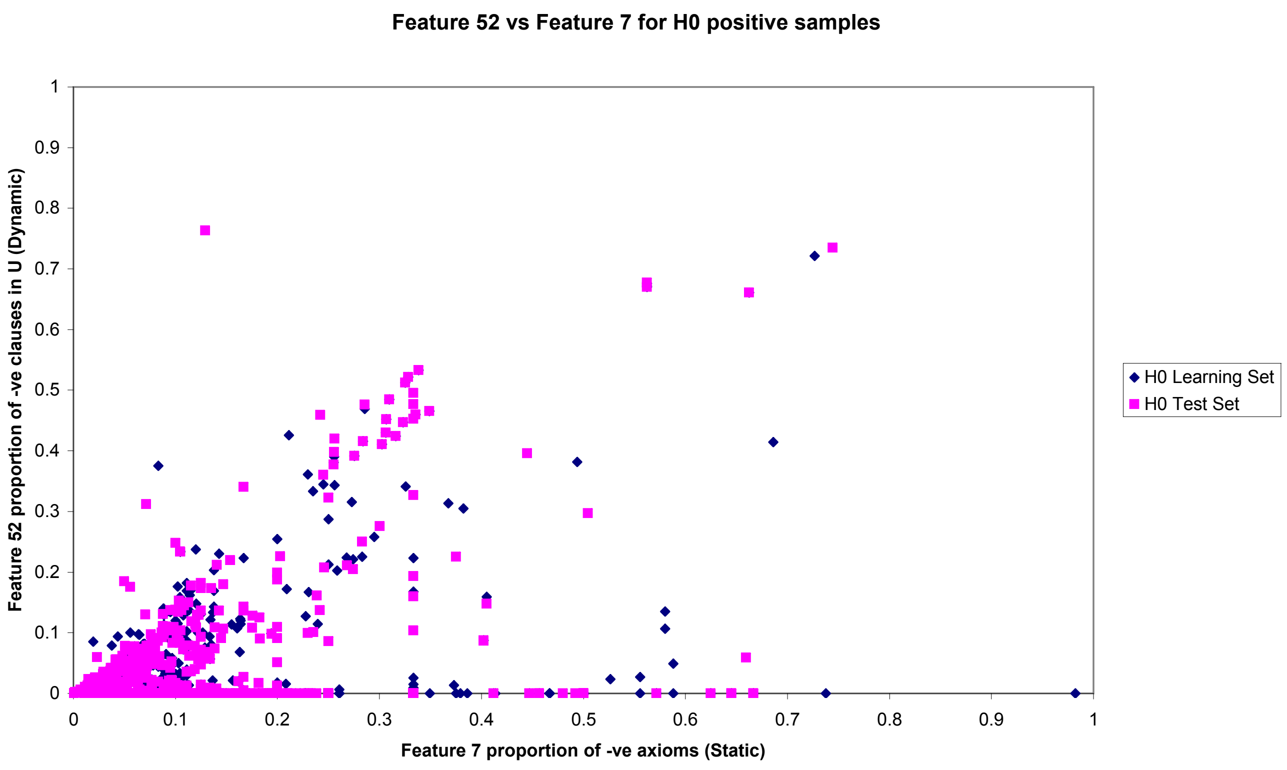

Figure 6.36: Feature 52 (Dynamic Feature 38) versus Feature 7 for H0

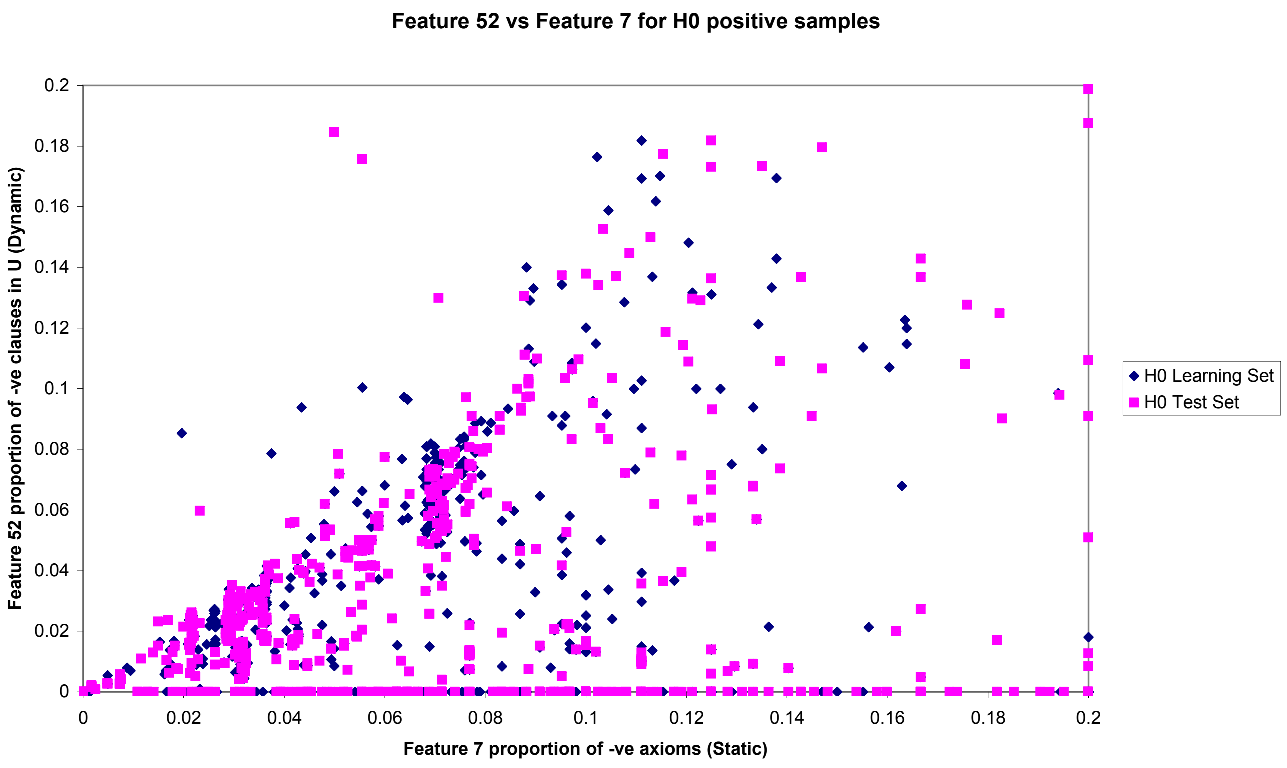

Figure 6.37: Detail of Feature 52 (Dynamic Feature 38) versus Feature 7 for H0

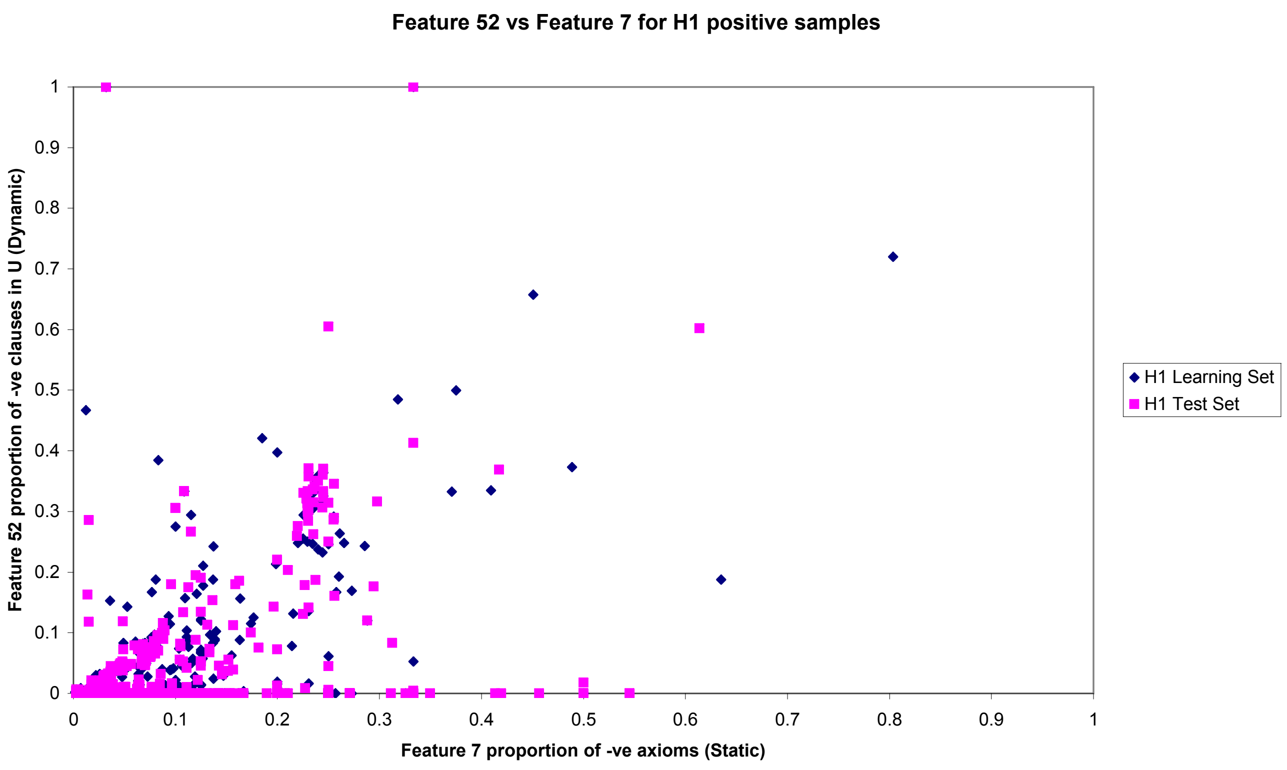

Figure 6.38: Feature 52 (Dynamic Feature 38) versus Feature 7 for H1

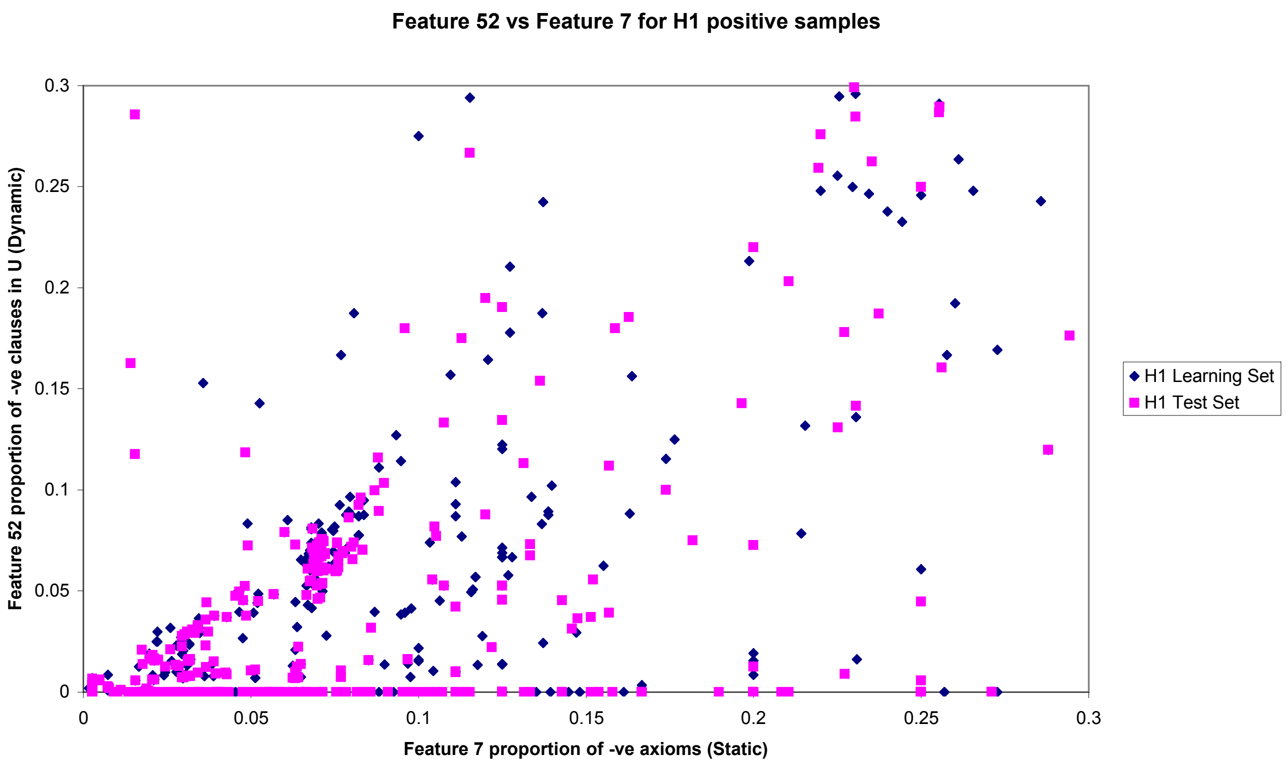

Figure 6.39: Detail of Feature 52 (Dynamic Feature 38) versus Feature 7 for H1

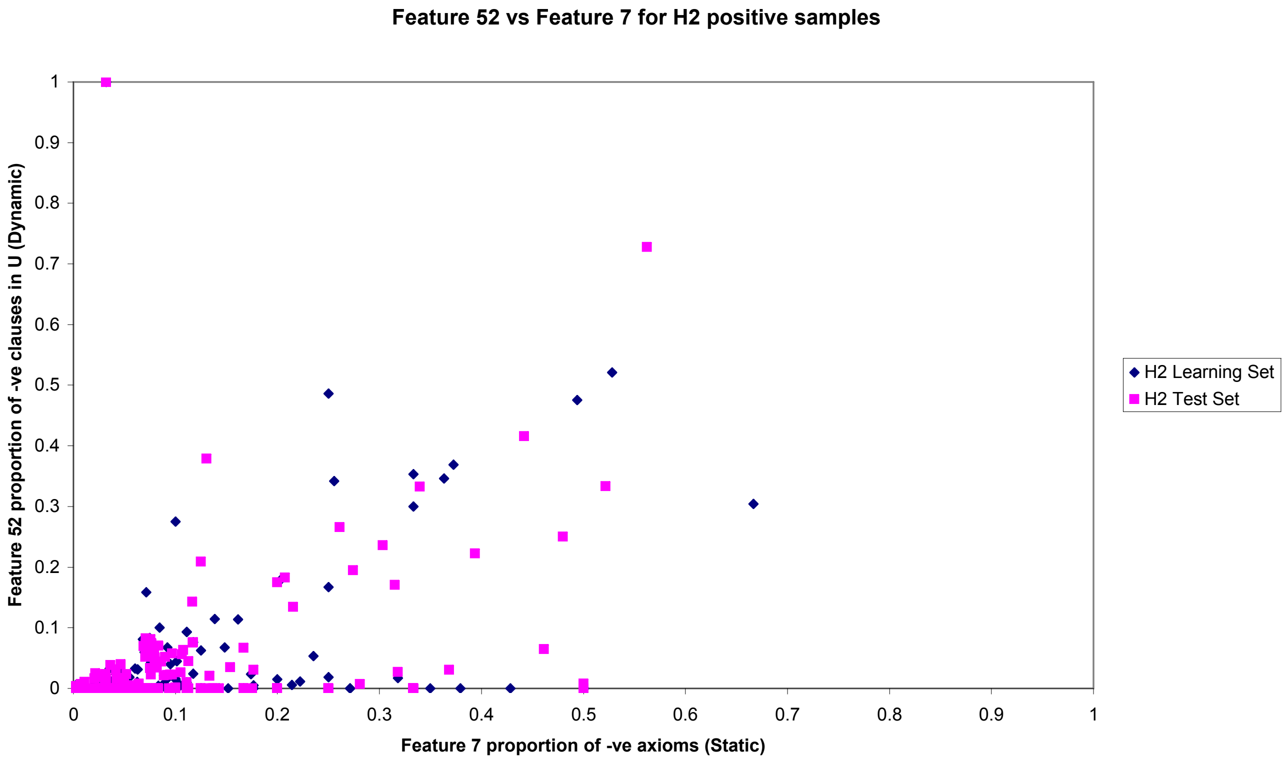

Figure 6.40: Feature 52 (Dynamic Feature 38) versus Feature 7 for H2

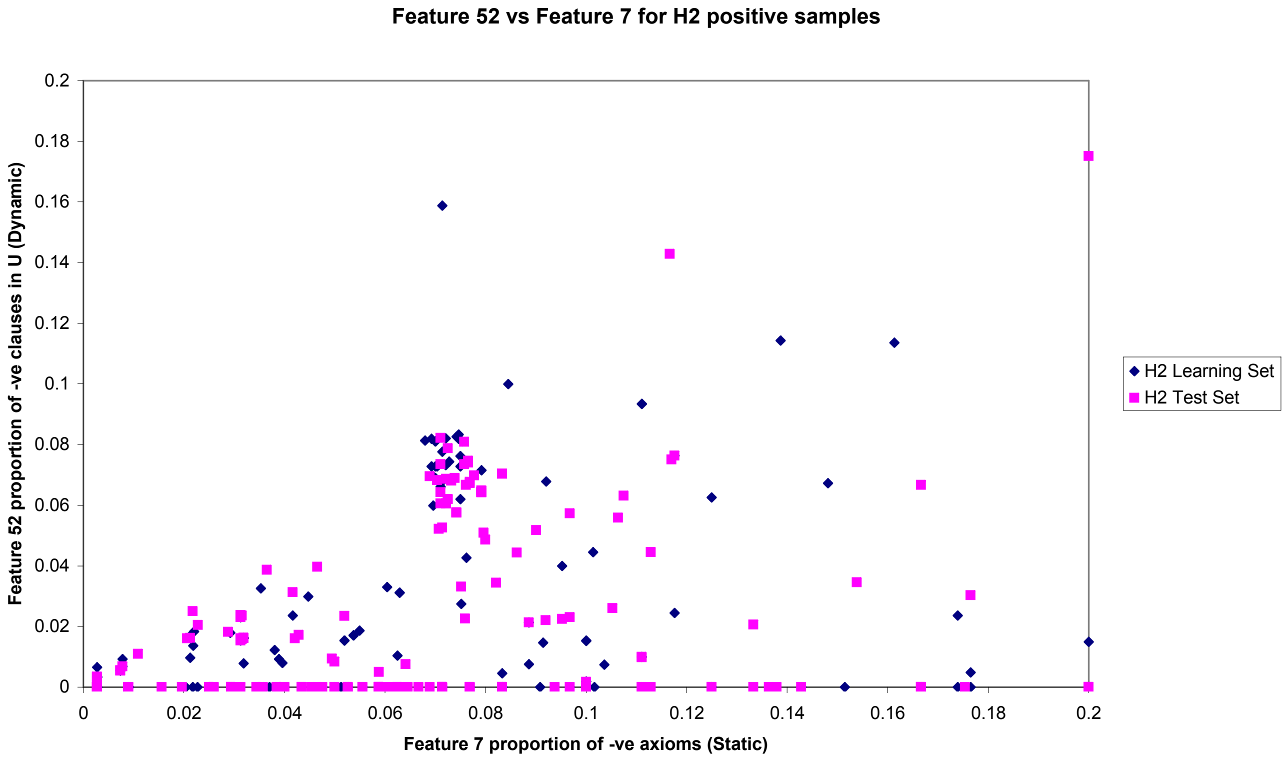

Figure 6.41: Detail of Feature 52 (Dynamic Feature 38) versus Feature 7 for H2

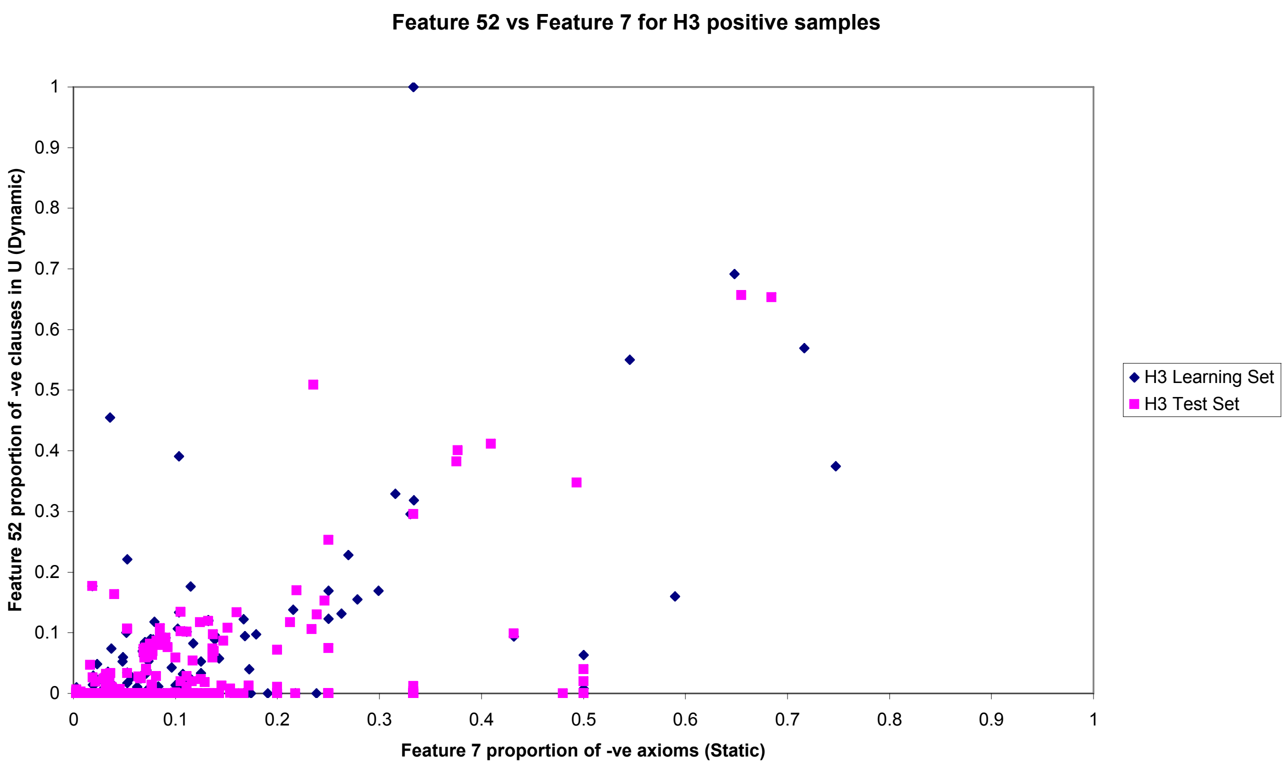

Figure 6.42: Feature 52 (Dynamic Feature 38) versus Feature 7 for H3

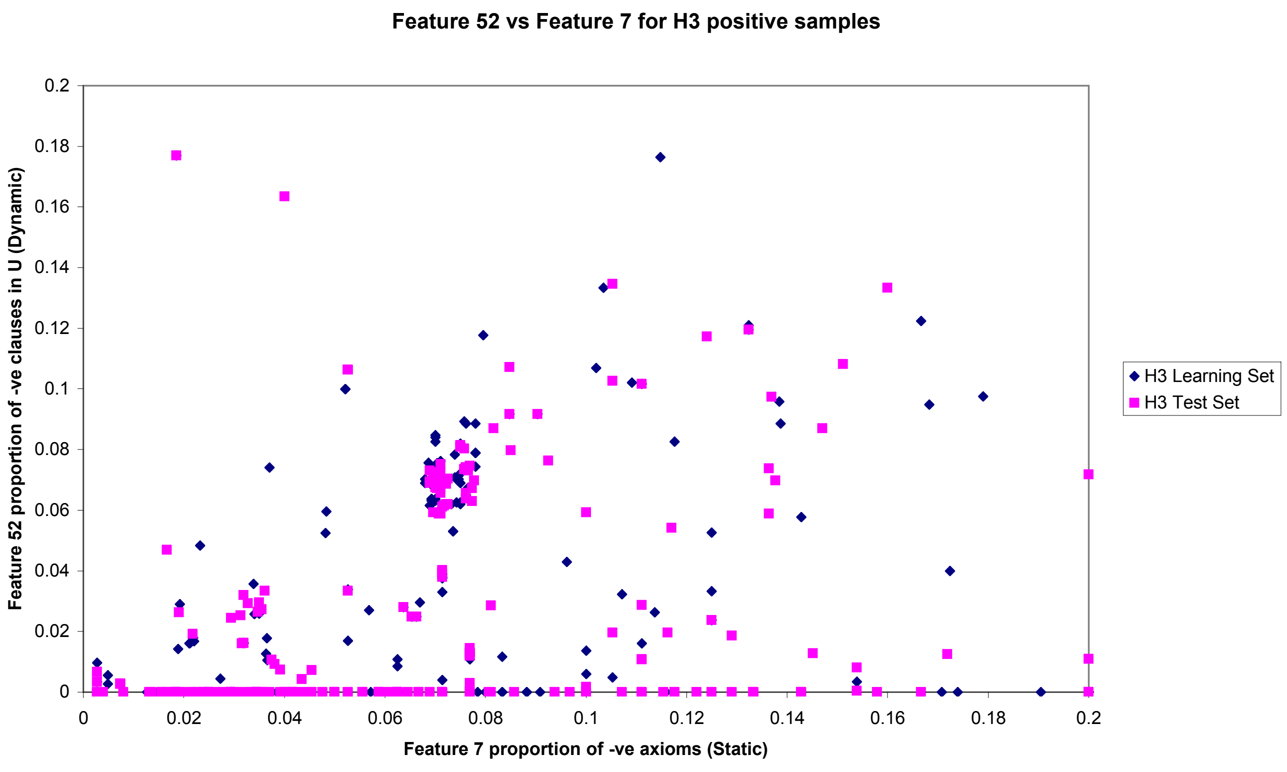

Figure 6.43: Detail of Feature 52 (Dynamic Feature 38) versus Feature 7 for H3

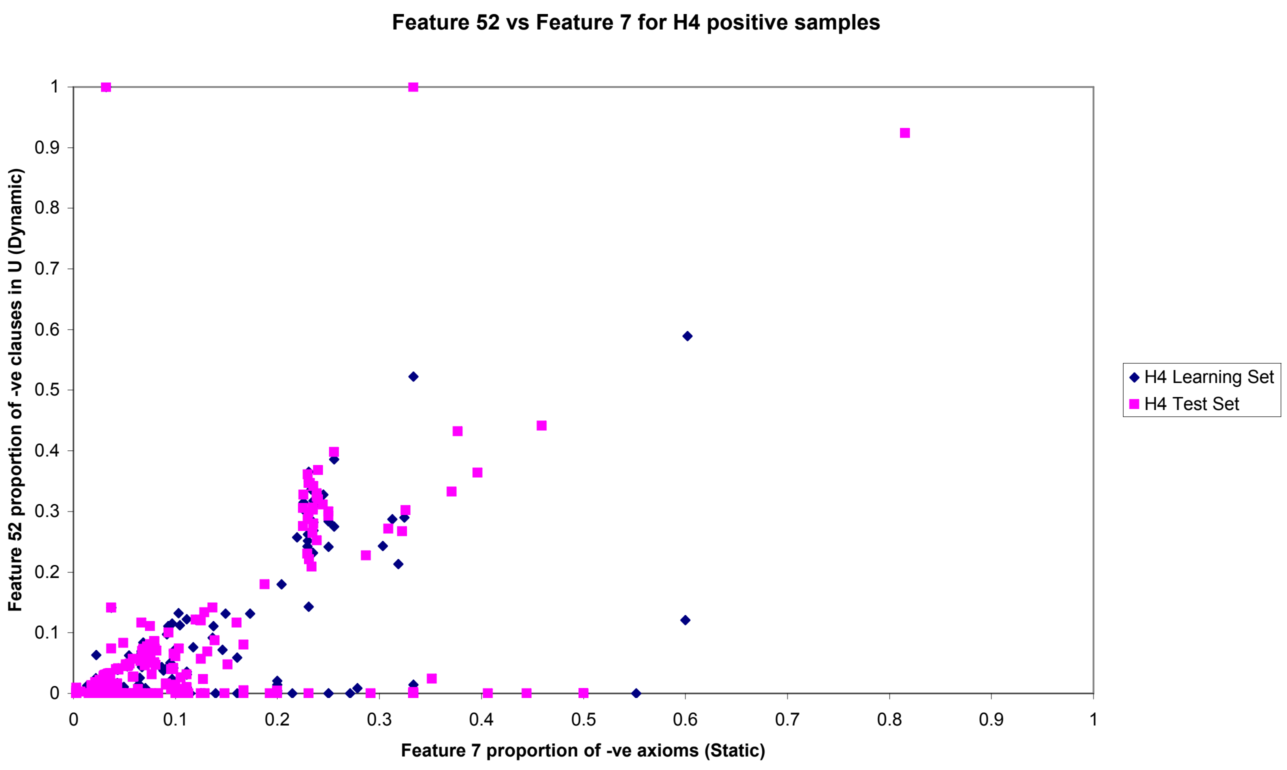

Figure 6.44: Feature 52 (Dynamic Feature 38) versus Feature 7 for H4

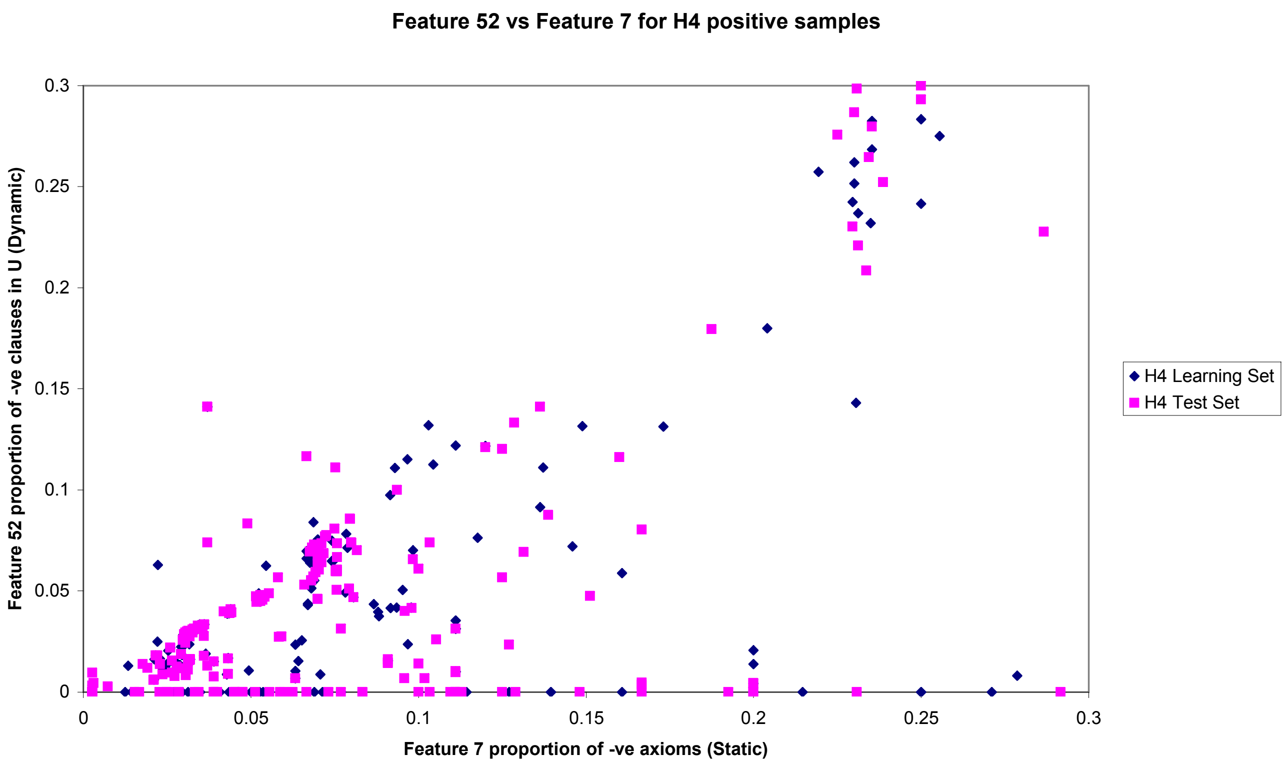

Figure 6.45: Detail of Feature 52 (Dynamic Feature 38) versus Feature 7 for H4

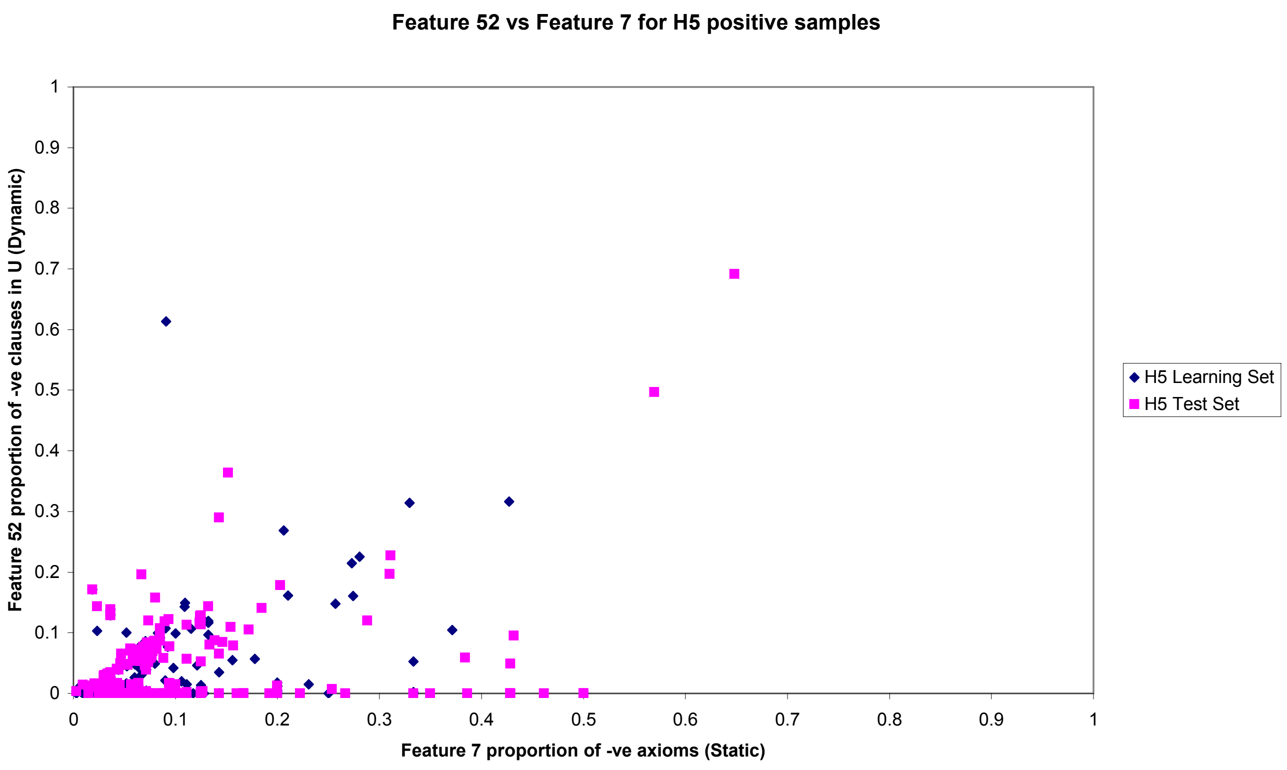

Figure 6.46: Feature 52 (Dynamic Feature 38) versus Feature 7 for H5

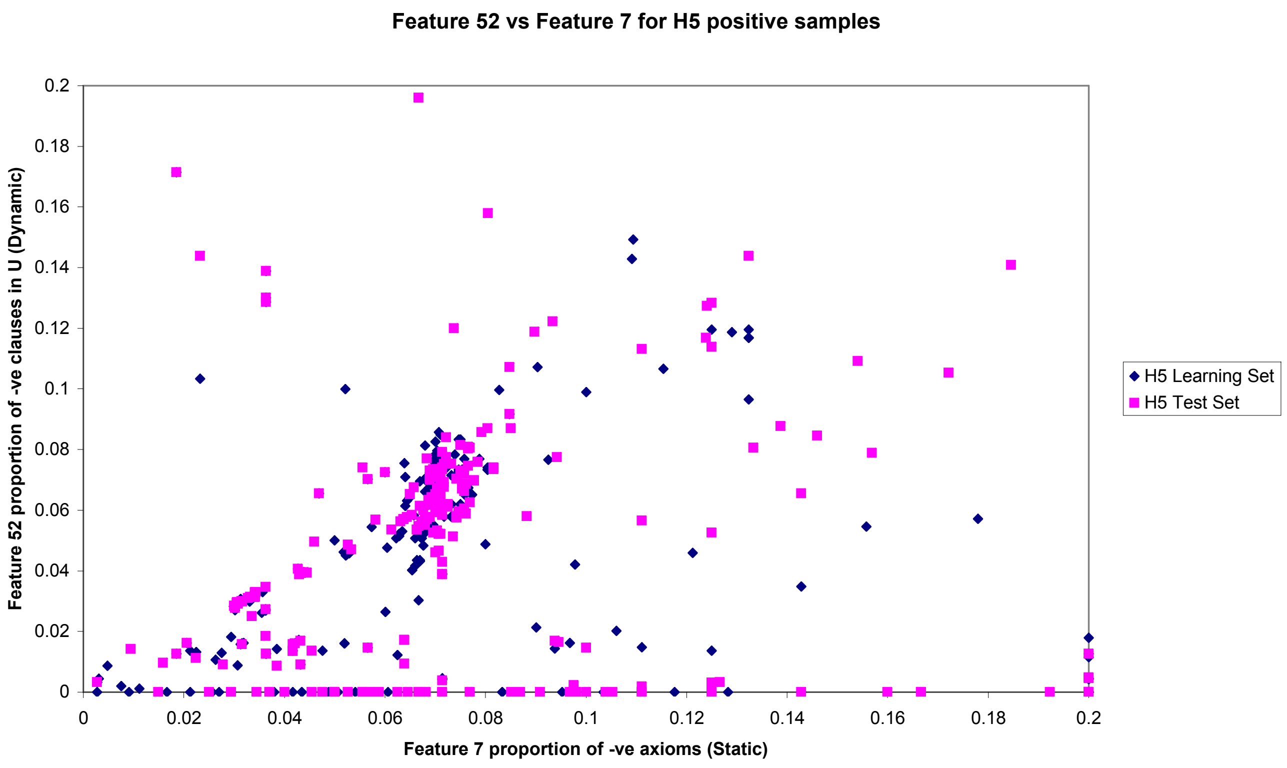

Figure 6.47: Detail of Feature 52 (Dynamic Feature 38) versus Feature 7 for H5

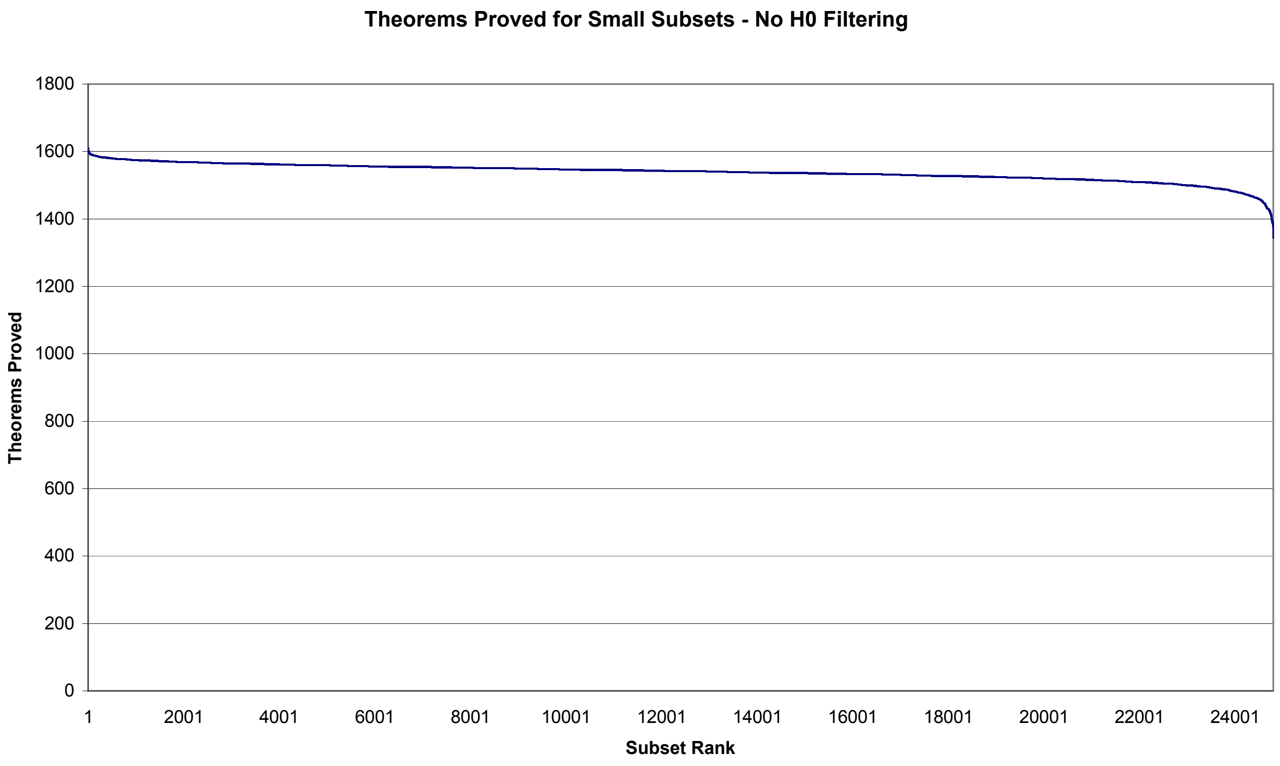

Figure 6.48: Theorems proved vs subset rank for small subsets without H0 filtering

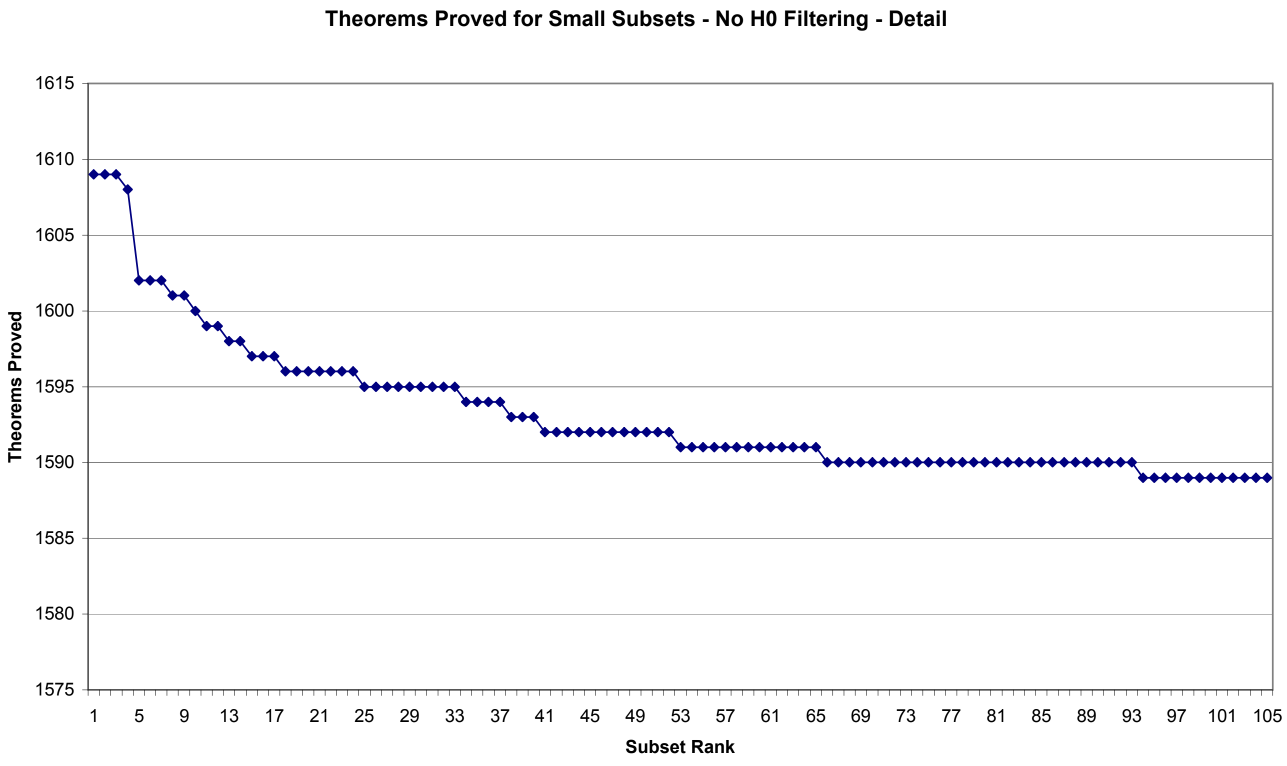

Figure 6.49: Theorems proved vs subset rank for small subsets without H0 filtering (Detail)

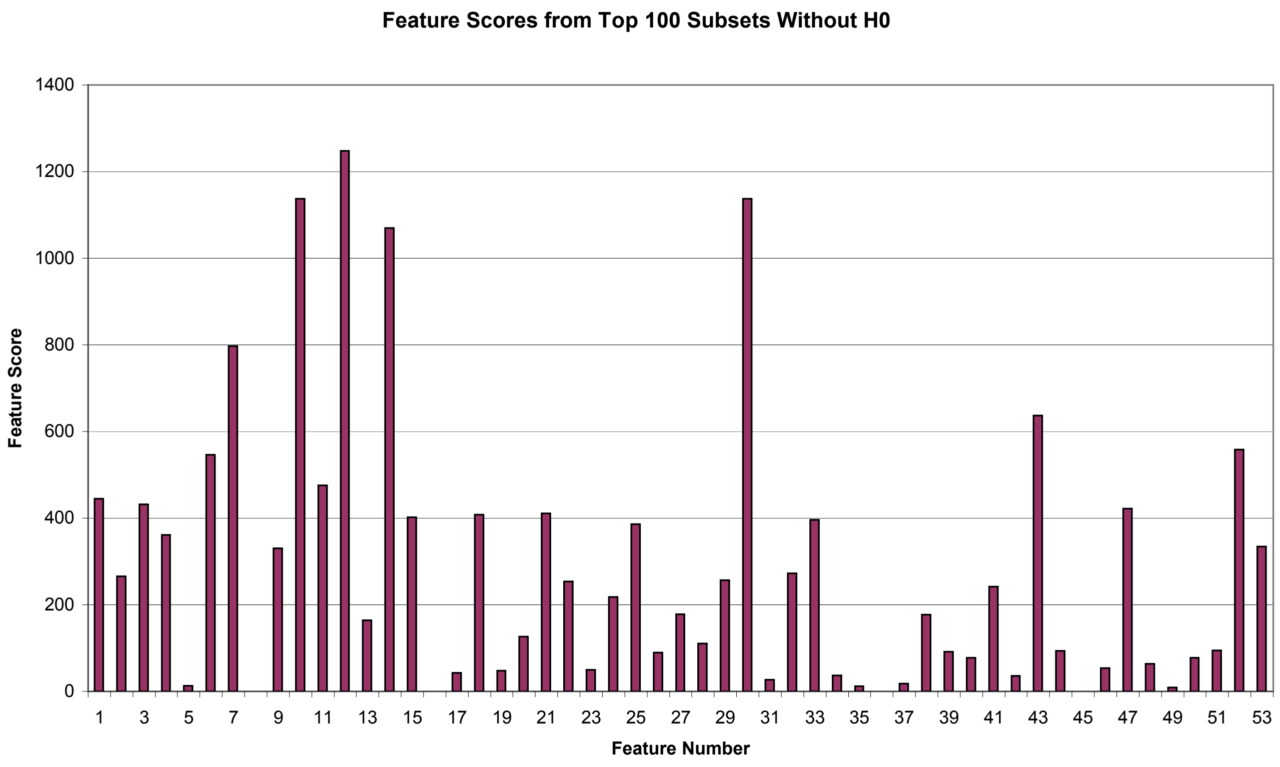

Figure 6.50: Scores for each feature based on 100 best subsets with no H0 filtering

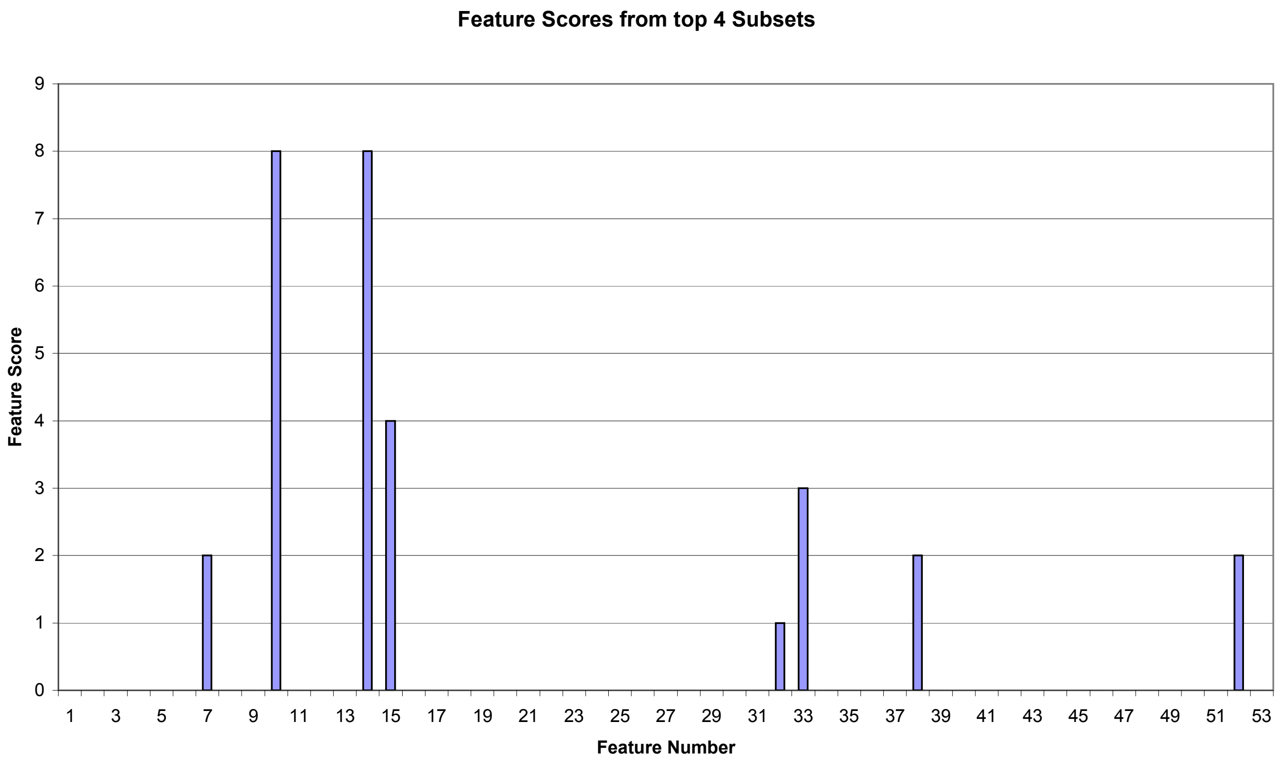

Figure 6.51: Scores for each feature based on top 4 subsets only with no H0 filtering

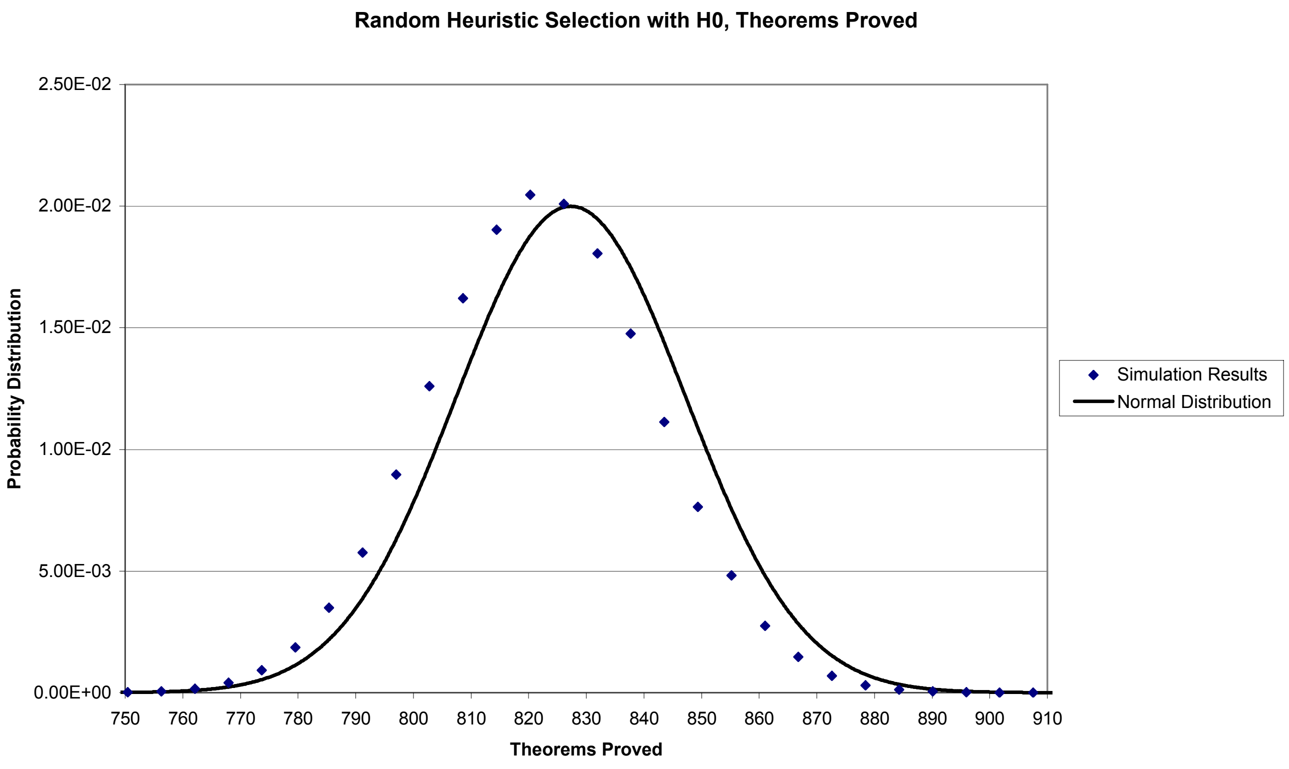

Figure 6.52: Probability distribution for number of theorems proved with H0 filtering

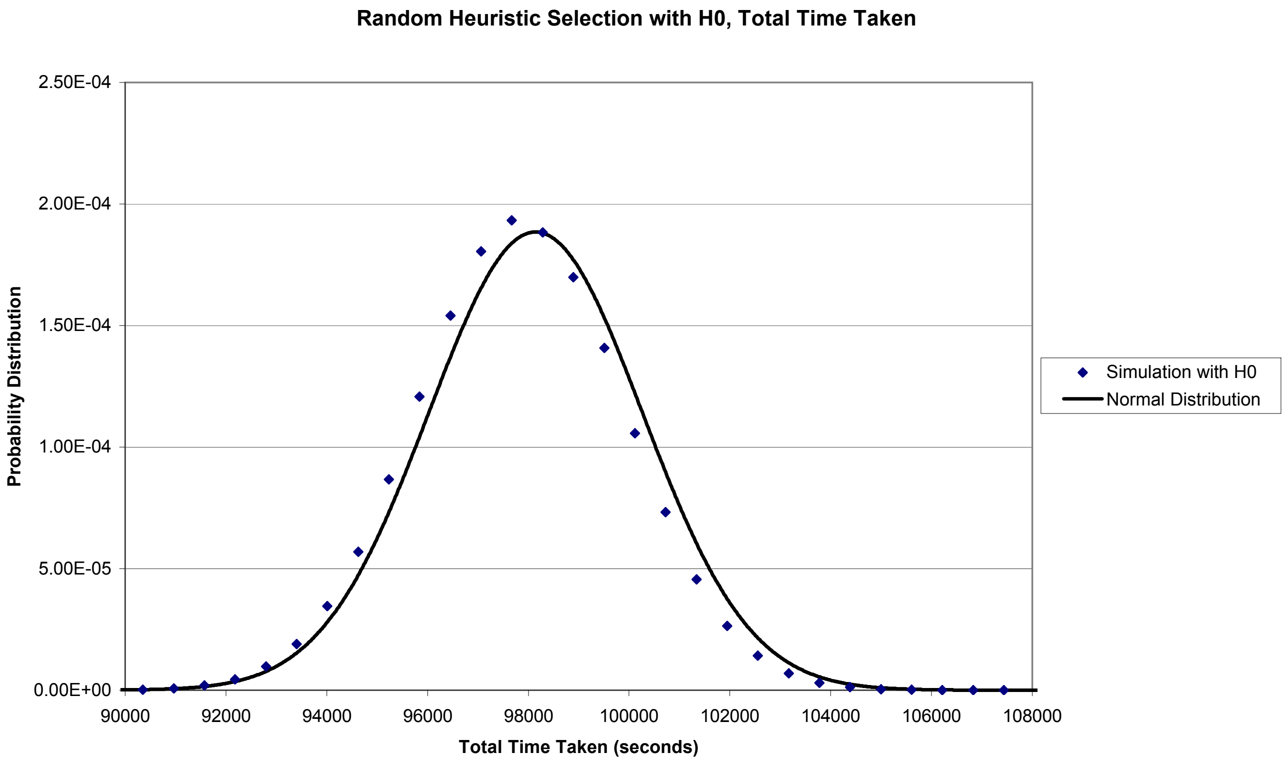

Figure 6.53: Probability distribution for total time taken with H0 filtering

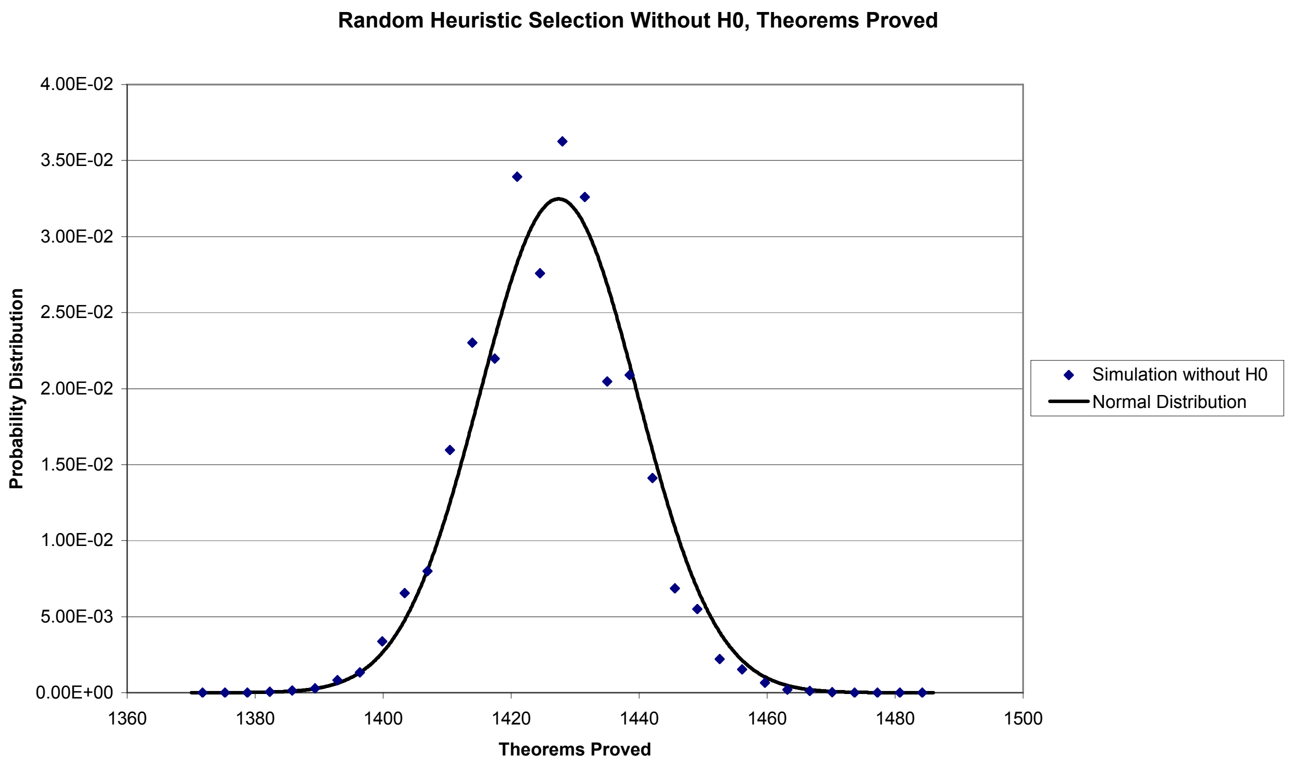

Figure 6.54: Probability distribution for number of theorems proved with no H0 filtering

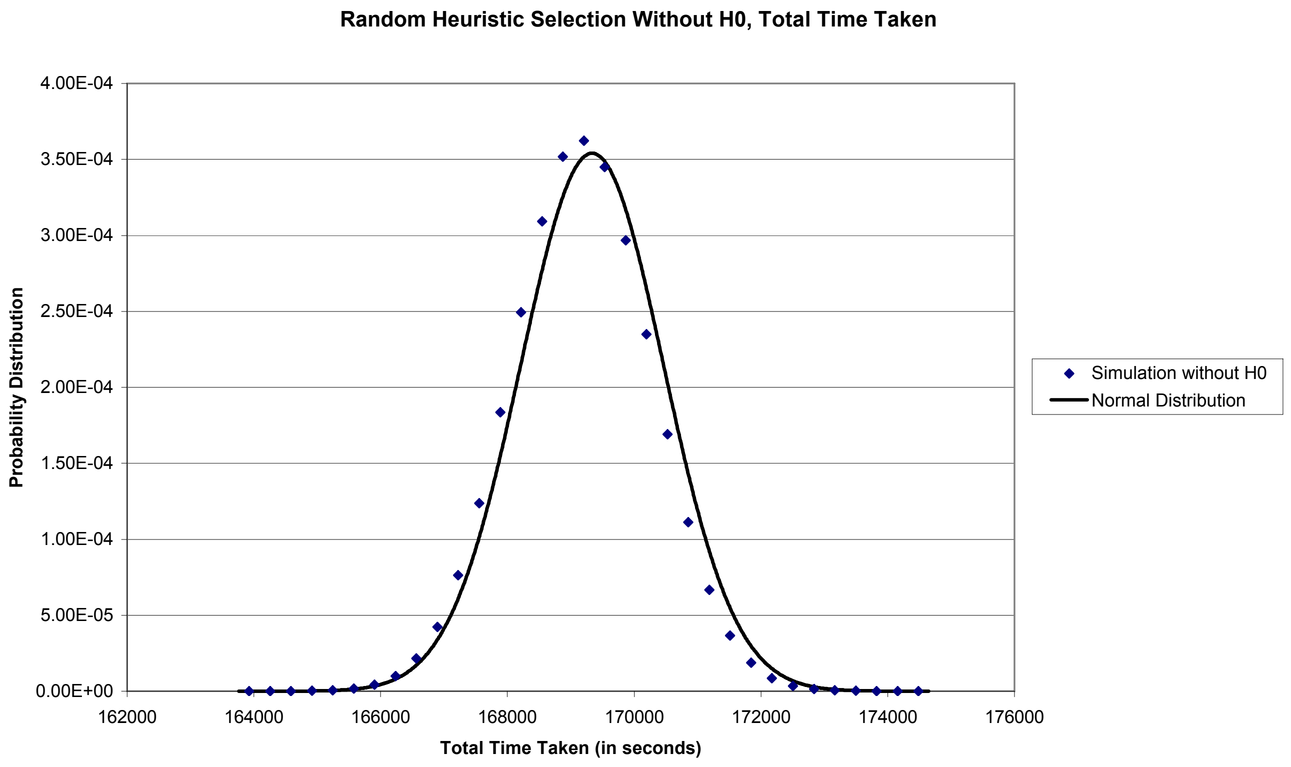

Figure 6.55: Probability distribution for total time taken with no H0 filtering

Chapter 7

Conclusions

Theorem provers based on first order logic with equality should be capable of automatic operation, but to be effective different heuristics need to be used for different problems. The choice of the best heuristic to be used often requires the intervention of a human expert.

The work described in this dissertation has shown that machine learning based on simple features measured on conjectures and associated axioms is effective in determining a good choice of heuristic to use in the proof search. This was demonstrated by the learned heuristic selection routine doing better than any single heuristic, and doing much better than random heuristic selection.

The work was generic - no significance was attached to function and predicate symbols. Even so, there is still a large number of features that may be measured and fifty three were investigated. It was found that effective learning required a combination of only a very few features. In fact for the case which included H0 filtering, using just two features gave optimal results.

The remainder of this chapter covers related work, summarises what was learned about applying machine learning to a theorem prover, reaches some conclusions, and suggests where the work could be taken further.

7.1 Related work

Other work in the application of machine learning to theorem proving has concentrated on learning heuristics rather than learning how to choose between established heuristics. There are a number of drawbacks to learning heuristics and success has been limited. See chapter 2 of this dissertation for more details.

Additionally, the E theorem prover (Schulz [78]) has an auto mode for heuristic selection from features of the conjecture to be proved. The method used is not based on machine learning in the normal meaning of the term. A set of binary or ternary feature values is used to divide conjectures into classes. Shulz applied this classification to the TPTP library [85] and then tested over one hundred heuristics to find which was best for each class as a whole. For new conjectures the auto mode uses the features to determine the class and then uses the previously stored "best" heuristic for that class. The choice of features to use was based on Shulz's experience with the constraint that the features can only take one of two or three discrete values.

The work described in this dissertation is very different to that behind the E auto mode. Rather than assume which features are important, machine learning and systematic feature selection methods were used to determine which were optimal. There was no constraint on the type of feature to use and the support vector machine classifiers are able to model a much more sophisticated relationship between feature values and heuristic choice. A direct comparison in terms of performance is not useful as the E auto mode is tailored to the TPTP library and uses over a hundred heuristics while the work described in this dissertation was limited to five heuristics.

7.2 What was learned about machine learning

The field of machine learning is extensive and no attempt was made in the work described in this dissertation to be comprehensive in the investigation of the efficacy of different approaches when applied to theorem provers. In particular the decision was made to use established support vector machine (SVM) software Joachims [36] and not to attempt to program other learning schemes. SVMs are widely used, and possible improvements from using other methods are unlikely to materially alter the results concerning whether machine learning works for heuristic selection in a theorem prover. The work was also limited to a single theorem prover, but within the area of first order logic with equality most modern theorem provers use very similar algorithms. A range of kernel models were looked at, a wide range of features were investigated including dynamic as well as static measures and extensive feature selection experiments done.

7.2.1 Kernel function

One outcome of the first, preliminary experiment, was the finding that the best results were obtained with the radial basis functions kernel. As discussed earlier, this is consistent with other workers [25, 24] in the field as it relates to a nearest neighbour approach.

7.2.2 Static and dynamic features

A novel aspect of the work reported in this dissertation is the use of dynamic as well as static features. Dynamic features are measured after a fixed number of steps of the proof search process.[1]

Though the initial experiments didn't show any great advantage arising from the use of dynamic features, the feature selection experiments produced the interesting, and unexpected, result that the key two features that gave the best learning outcome with H0 filtering were *static and dynamic measures of the same feature.*

[1]Subsequent to undertaking the work, Hoos *et al* [100] published work which used dynamic features in the field of SAT solvers. SAT solvers are very different from first order logic theorem provers so a close comparison is not possible.

7.2.3 Comparative versus absolute margins

As with many combinatorial problems, investigating all the possibilities of different features and heuristics involves very large numbers of trials so it is necessary to impose restrictions to make the problem tractable.

One finding of the feature selection experiments is that care must be taken as to how the problem is restricted. The first approach of treating each heuristic classifier separately and scoring features according to the number of correct classifications led to different results to those obtained when the whole heuristic selection process was considered.

The implication of this is that it is not just the sign or even absolute magnitude of the margin obtained with any *single* heuristic classifier that is of ultimate importance, rather it is the relative margin magnitudes from *all* the heuristic classifiers. It is the relative magnitudes that determine which heuristic is chosen and thus are of more significant. For example, a classifier for one heuristic might correctly produce a positive margin to indicate that that heuristic is the best but this correct classification will be undone at the heuristic selection stage if another heuristic's classifier incorrectly produces a positive margin and that margin is more positive than that of the correct heuristic classifier.

The importance of the relative margin magnitudes from heuristic classifier to heuristic classifier means that optimising the parameter values of individual classifiers to maximise the number of correct classifications may be counter-productive. Better individual heuristic classifier performance does not necessarily imply better overall heuristic selection performance.

7.3 Conclusions

7.3.1 Machine learning is applicable to theorem provers

Though the improvement was not by a large margin, the machine learned heuristic selection did better than any of the heuristics by themselves. This result was demonstrated to be learned intelligence rather than random chance by the fact that random selection of heuristics led to results that were considerably worse.

Table 7.1 summarises the key results.

7.3.2 Very few features are needed

Though different experiments led to different rankings of features, all the feature selection and ranking experiments indicated that only a few features are needed. Where features were removed one at a time the results did not deteriorate until the last few features were removed. The examination of small subsets gave results that were an improvement over those obtained using the full set of fifty three features in the learning process.

7.3.3 H0 filtering saves time at the expense of theorems proved

Significant savings in total time may be obtained by applying a pre-filter which rejects some conjectures as being too difficult to prove without spending time on seeking a proof.

Though the classifier that does this is quite effective, it is not perfect so whether or not such H0 filtering is performed depends on the overall task and the user's requirements. The trade off is between total time and total number of theorems proved. Overall time may also be reduced by reducing the CPU time limit allowed, this also reduces the number of theorems proved and is dependent on the problems in the test set - it is not an intelligent filtering method.

7.3.4 Matching features to heuristics

The feature selection experiments involving a single set of features for all heuristics led to the finding that just a small subset of two or three features gave optimal results. Additionally, the optimal feature sets differ between the case where H0 was included and where it wasn't. This implies some relation between the features best used for learning and the set of heuristics.

Additionally, the feature selection experiments involving individual heuristic classifiers showed that the feature numbers with the highest scores differed between classifiers (see figures 6.20 to 6.25). It is reasonable to hypothesise that each heuristic has a separate subset of features for which it is well matched.

7.4 Further work

Following up on the hypothesis that each heuristic has a separate subset of features for which it is well matched, it would be interesting to investigate the possibility of using separate feature sets for each heuristic classifier. To investigate this properly requires coverage of a very large combined search space. Individual heuristic classifiers cannot be treated in isolation because the best choice of features depends on the performance measure used. The feature subset investigation, based on overall heuristic selection performance, gave different results to the individual heuristic classifiers in terms of which features are of most significance. Additionally care would need to be taken to consider properly the relative scales of different features when different sets are used with different classifiers whose output margins are all to be compared.

An alternative hypothesis is that different features are useful in determining the comparative benefits of pairs of heuristics. For instance the best features to use when comparing heuristics H1 and H3 may differ from those to use when comparing H1 and H2 or H4 and H5. Investigating this on all pairings of the five heuristics would be possible, similar to running the subset experiment ten times, but still a major piece of work.

Heuristic Number	*Number of Theorems Proved*	*Total Time Taken in Seconds*
1	1,514	162,029
2	1,352	177,530
3	1,424	168,593
4	1,421	169,598
5	1,339	176,959
Random Heuristic Selection with H0 Filtering	827(mean)	98,145(mean)
Random Heuristic Selection without H0 Filtering	1,427(mean)	169,337(mean)
Learned Heuristic Choice (Best Subset $\gamma = 48.05$) (Includes H0 filtering)	1,602	149,323
Learned Heuristic Choice (Best Subset $\gamma = 10$) (No H0 Filtering)	1,609	150,700

Table 7.1: Summary of Key Results

Bibliography

[1] S. Abe. *Support vector machines for pattern classification.* Springer, 2005.

[2] E. Alpaydin. *Introduction to machine learning.* MIT Press, 2004.

[3] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[4] L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I, chapter 11, pages 353–397. Kluwer, 1998.

[5] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–100. North Holland, 2001.

[6] C. Baier and J. Katoen. *Principles of model checking.* MIT Press, 2008.

[7] C.M. Bishop. *Pattern recognition and machine learning.* Springer, 2006.

[8] M. Black. The identity of indiscernibles. *MIND*, LXI(242):153–164, April 1952.

[9] M. Blanchard, J. Horton, and D. MacIsaac. Folding architecture networks improve the performance of otter. *Short Paper at 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006) Tuesday 14 November 2006*, 2006.

[10] D. Bondyfalat, B. Mourrain, and T. Papadopoulo. An application of automatic theorem proving in computer vision. In *X,-S Gao, D.Wang and L.Yang (Eds) ADG'98, LNAI 1669*, pages 207–232. Springer-Verlag, 1999.

[11] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using first-order theorem provers in a data structure verification system. In *Proc. 8th Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, 2007.

[12] J. Brank, M. Grobelnik, N. Milic-Frayling, and D. Mladenic. Feature selection using linear support vector machines. *Technical Report MSR-TR-2002-63*, 2002.

[13] C.J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[14] K. Claessen, R. Hahnle, and J. Martensson. Verification of hardware systems with first-order logic. *PaPS2002*, 2002.

[15] E. Cohen. Taps: A first-order verifier for cryptographic protocols. In *Computer Security Foundations Workshop*, volume CSFW-13 Proceedings 13 IEEE 3-5 July, pages 144–158, 2000.

[16] S.C. Cook. The complexity of theorem-proving procedures. In *Third ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[17] I. Dahn, A. Haida, T. Honigmann, and C. Wernhard. Using mathematica and automated theorem provers to access a mathematical library. In *CADE-15 Workshop Integration of Deduction Systems*, 1998.

[18] E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal on Artificial Intelligence Tools*, 15(1):81–107, 2006.

[19] J. Denzinger and M. Fuchs. High performance atp systems by combining several ai methods. In *Proc. of the 15th IJCAI, Nagoya.* Morgan Kaufmann, 1997.

[20] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from previous proof experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999. (also to be published as a SEKI report).

[21] J. Denzinger, M. Kronenburg, and S. Schulz. Discount - a distributed and learning equational prover. *J. Autom. Reason.*, 18(2):189–198, 1997.

[22] Dershowitz and Nachum. Orderings for term-rewriting systems. *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 123–131, Oct. 1979.

[23] H.B. Enderton. *A mathematical introduction to logic.* Academic Press Inc, 1972.

[24] M. Fuchs. Automatic selection of search-guiding heuristics for theorem proving. Technical Report TR-ARP-2-1998, Research School of Information Sciences and Engineering and Centre for Information Science Research Australian National University, 1998.

[25] M. Fuchs and M. Fuchs. Applying case-based reasoning to automated deduction. In *ICCBR '97: Proceedings of the Second International Conference on Case-Based Reasoning Research and Development*, pages 23–32, London, UK, 1997. Springer-Verlag.

[26] J.H. Gallier. *Logic for computer science: foundations of automatic theorem proving.* Harper & Row Publishers, Inc., New York, NY, USA, 1985. Revised edition (2003) available for download from http://www.cis.upenn.edu/ jean/gbooks/logic.html.

[27] G. Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:405–431.

[28] C. Goller. Learning search-control heuristics for automated deduction systems with folding architecture networks. *ESANN'1999 proceedings - European Symposium on Artificial Neural Networks Bruges (Belgium) 21-23 April 1999*, pages 45–50, 1999.

[29] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[30] S.R. Gunn. Support vector machines for classification and regression. Technical report, University of Southampton, May 1998.

[31] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 2003.

[32] M. Huth and M. Ryan. *Logic in computer science, Second Edition.* Cambridge University Press, 2004.

[33] C. Quigley J. Meng and L.C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

[34] M. Jamnik, M. Kerber, and C. Benzmuller. Learning method outlines in proof planning. *CSRP-01-08*, 2001.

[35] Jaśkowski. On the rules of suppositions in formal logic. *Studia Logica*, 1, 1934. Reprinted in S. McCall (1967) Polish Logic 1920-1939, Oxford: Oxford Univ. Press pp. 232-258.

[36] T. Joachims. Making large-scale svm learning practical. In B.Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning.* MIT-Press, 1999.

[37] E. Katsiri and A. Mycroft. Knowledge representation and scalable abstract reasoning for sentient computing using first-order logic. In *Proc. 1st Workshop on Challenges and Novel Applications for Automated Reasoning in conjunction with CADE-19*, pages 73–82, 2002.

[38] M. Kaufmann and J.S. Moore. Some key research problems in automated theorem proving for hardware and software verification. *RACSAM, Rev. R.Acad. Cien. Serie. A. Mat.*, 98(1):181–195, 2004.

[39] R. Kaye. *The mathematics of logic, a guide to completeness theorems and their applications.* Cambridge University Press, 2007.

[40] D.E. Knuth. *The art of computer programming, Volume 2, seminumerical algorithms.* 1998.

[41] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, Elmsford, N.Y., 1970.

[42] T. Lev-ami, N. Immerman, T. Reps, M. Sagiv, and S. Srivastava. Simulating reachability using first-order logic with applications to verification of linked data structures. In *In CADE-20*, pages 99–115, 2005.

[43] H. Li and Y. Wu. Automated theorem proving in incidence geometry - a bracket algebra based elimination method. In *J.Richter-Gebert and D.Wang (Eds.) ADG 2000 LNAI 2061*, pages 199–227. Springer-Verlag, 2001.

[44] Z. Liu, M. Tan, and F. Jiang. Regularized f-measure maximization for feature selection and classification. *Journal of Biomedicine and Biotechnology*, 2009:8, 2009.

[45] D.W. Loveland. Mechanical theorem-proving by model elimination. *J. ACM*, 15(2):236–251, 1968.

[46] M. Manzano. *Extensions of first order logic.* Number 19 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996.

[47] C. Mareco and A. Paccanaro. Using neural networks to improve the performance of autotmated theorem provers. *Computational Methods and Neural Networks, M.P.Bekakos, M.Sambandham Eds.*, pages 379–404, 1998.

[48] J. McCarthy. Programs with common sense. In *Mechanisation of Thought Processes: Proceedings of a symposium held at the National Physical Laboratory on 24th, 25th, 26th and 27th November 1958*, volume 1, pages 75–91. London HMSO, 1959.

[49] W.W. McCune. *OTTER 3.0 reference manual and guide.* Technical Report ANL-94/6. Argonne National Laboratory, January 1994.

[50] W.W. McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[51] J. Meng. Integration of interactive and automatic provers. In *Manuel Carro and Jesus Correas (editors) Second CologNet Workshop on Implementation Technology for Computational Logic Systems*, September 2003. FME2003.

[52] J. Meng and L.C. Paulson. Experiments on supporting interactive proof using resolution. In *Second International Joint Conference on Automated Reasoning, IJCAR 2004*, Cork, Ireland, 4-8 July 2004.

[53] J. Meng and L.C. Paulson. Lightweight relevance filtering for machine-generated resolution. In *Geoff Sutcliffe, Renate Schmidt and Stephan Schulz (editors), ES-Cor: Empirically Successful Computerized Reasoning (CEUR Workshop Proceedings)*, volume 192, pages 53–69, 2006.

[54] J. Mercer. Functions of positive and negative type and their connection with the theory of integral equations. *Philos. Trans. Roy. Soc. London*, A209:415–446, 1909.

[55] T.M. Mitchell. *Machine learning.* McGraw-Hill, 1997.

[56] K. Morik, P. Brockhausen, and T. Joachims. Combining statistical learning with a knowledge-based approach - a case study in intensive care monitoring. In *Proc. 16th Int'l Conf. on Machine Learning (ICML-99)*, 1999.

[57] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. pages 530–535, 2001.

[58] E.T. Mueller. *Commonsense reasoning.* Elsevier, 2006.

[59] J. Shawe-Taylor N. Cristianini. *Support vector machines and other kernel-based learning methods.* Cambridge University Press, 2000.

[60] M. Newborn and Z. Wang. Octopus: combining learning and parallel search. *J. Autom. Reason.*, 33(2):171–218, 2004.

[61] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving", booktitle = "handbook of automated reasoning", pages = "371-443", year = "2001", url = "citeseer.ist.psu.edu/nieuwenhuis01paramodulationbased.html".

[62] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic.* Number Tutorial 2283 in LNCS. Springer, 2002.

[63] A. Nonnengart and C. Weidenback. Computing small clause normal forms. In *Handbook of Automated Reasoning*, volume I, pages 335–367. Elsevier Science and MIT Press, 2001.

[64] A.B. Novikoff. On convergence proofs on perceptrons. *Symposium on the Mathematical Theory of Automata*, 12:615–622, 1962.

[65] F.J. Pelletier. A history of natural deduction and elementary logic textbooks. In *Logical Consequence: Rival Approaches, Vol. 1*, pages 105–138. Oxford: Hermes Science Pubs, 2000.

[66] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlem, in welchem, die addition als einzige operation hervortritt. *Sprawozdanie z I Kongresu Matematyków Krajów Stowiańskich*, pages 92–101 and 395, 1930.

[67] V. Prevosto and U. Waldmann. Spass+t. In *Geoff Sutcliffe, Renate Schmidt and Stephan Schulz, editors, ESCoR: Emperically Successful Computerized Reasoning*, August 2006.

[68] G. Priest. *An introduction to non-classical logic, Second Edition.* Cambridge University Press, 2008.

[69] C.E. Rasmussen and C.K.I. Williams. *Gaussian processes for machine learning.* MIT Press, 2006.

[70] A. Riazanov and A. Voronkov. Splitting without backtracking. *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, 1:611–617, 2001. B. Nebel, ed.

[71] B.D. Ripley. *Pattern recognition and neural networks.* CUP, 1996.

[72] G.A. Robinson. Automatic deduction with hyper-resolution. *International journal of computer mathematics*, 1:227–234, 1965.

[73] G.A. Robinson and L. Wos. Paramodulation and theorem proving in first order theories with equality. Presented at 4th Annual Machine Intelligence Workshop, Edinburgh, Scotland, Aug 1968.

[74] J.A. Robinson. Theorem-proving on the computer. *J. ACM*, 10(2):163–174, 1963.

[75] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[76] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[77] S. Schulz. *Learning search control knowledge for equational deduction.* Number 230 in DISKI. Akademische Verlagsgesellschaft Aka GmbH Berlin, 2000.

[78] S. Schulz. E – A brainiac theorem prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[79] S. Schulz and F. Brandt. Using term space maps to capture search control knowledge in equational theorem proving. In A.N. Kumar and I. Russel, editors, *Proc. of the 12th FLAIRS, Orlando*, pages 244–248. AAAI Press, 1999.

[80] J. Schumann. Automated theorem proving in high-quality software design. *Intellectics and Computational Logic (NASA report)*, 2000.

[81] W. Shakespeare. *Macbeth Act 5 Scene 5.*

[82] R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, 1979.

[83] T. Skolem. Logisch-kombinatorische untersuchungen über die erfüllbarkeit oder beweisbarkeit mathematischer sätze, nebst einem theoreme über dichte mengen. *Skrifter utgit av Videnskappsellkapet i Kristiania*, 4:4–36. See [89] for English translation.

[84] P. Smith. *An introduction to Gödel's theorems.* Cambridge University Press, 2007.

[85] G. Sutcliffe and C.B. Suttner. The tptp problem library: Cnf release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[86] A. Tarski. *A decision method for elementary algebra and geometry.* University of California Press, 1951.

[87] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence.* Springer, 2002.

[88] J. Urban. Malarea: a metasystem for automated reasoning in large theories. In *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, pages 45–58, 2007.

[89] J. van Heijenoort. *From Frege to Gödel a source book in mathematical logic 1879-1931.* Harvard University Press, 1967.

[90] Y. Venema. Review of extensions of first order logic by maria manzano. *The Journal of Symbolic Logic*, 63(3):1194–1196, Sep 1998.

[91] A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 13–28, London, UK, 2001. Springer-Verlag.

[92] C. Walther. A mechanical solution of schubert's steamroller by many-sorted resolution. *Artif. Intell.*, 26(2):217–224, 1985.

[93] C. Walther. *A many-sorted calculus based on resolution and paramodulation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[94] (Ed.) F. Wiedijk. *The seventeen provers of the World.* Number 3600 in LNAI. Springer, 2005.

[95] R. Wolf. *A tour through mathematical logic.* Number 30 in The Carus Mathematical Monographs. The Mathematical Association of America, 2005.

[96] L. Wos. Otter and the moufang identity problem. *Journal of Automated Reasoning*, 17:215–257, 1996.

[97] L. Wos. The flowering of automated reasoning. *D. Hutter and W. Stephan (eds) Mechanizing Mathematical Reasoning*, LNAI 2605:204–227, 2005.

[98] L. Wos, G.A. Robinson, and D.F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, 1965.

[99] L. Wos, G.A. Robinson, D.F. Carson, and L.Shalla. The concept of demodulation in theorem proving. *J. ACM*, 14(4):698–709, 1967.

[100] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of Artficial Intelligence Research*, 32:568–606, 2008.

Appendix A

Details of features

This appendix is not intended for detailed reading, but to provide a central reference for all the features used in the experiments.

A.1 Initial feature set

For the initial experiment a set of sixteen dynamic features was used, defined, as follows. The measurements were made after the solver had run for a number (100 in the first instance) of clause selections. In the following descriptions, the set U is that of the unprocessed clauses and the set P is that of the processed clauses. Clause length is a measure of the number of literals in the clause, clause depth is a measure of the degree of nesting of terms. Clause weight is based on the weighting scheme used in term ordering.

1. Proportion of the total number of generated clauses that are kept (i.e. are not discarded as being trivial).

2. The "Sharing Factor", a measure of the number of terms which are shared between different clauses (Stephan Schultz suggested that this measure seemed to correlate with the success or failure of the proof of some theorems.) The sharing factor is provided as a function within the E source code.

3. Proportion of the total clauses that are in P (*i.e.* have been processed).

4. The ratio of the size of multi-set U to its original size (the original size being the number of axioms in the original theorem).

5. The ratio of the longest clause in P to the longest clause in the original axioms.

6. The ratio of the average clause length in P to the average axiom clause length.

7. The ratio of length of the longest clause in U to the longest axiom clause length.

8. The ratio of the average clause length in U to the average axiom clause length.

9. The ratio of the maximum clause depth in P to the maximum axiom clause depth.

10. The ratio of the average clause depth in P to the average axiom clause depth.

11. The ratio of the maximum clause depth in U to the maximum axiom clause depth.

12. The ratio of the average clause depth in U to the average axiom clause depth.

13. The ratio of the maximum clause standard weight in P to the maximum axiom clause standard weight.

14. The ratio of the average clause standard weight in P to the average axiom clause standard weight.

15. The ratio of the maximum clause standard weight in U to the maximum axiom clause standard weight.

16. The ratio of the average clause standard weight in U to the average axiom clause standard weight.

Apart from the sharing factor measure (feature 2) all the measures are ratios which keeps the scales to a reasonable size.

A.2 Extended feature set

For the heuristic selection experiment the feature set was extended to 53 features including a 14 static features and 39 dynamic features.

A.2.1 Static feature set

There are fourteen static features as follows (all measured for the clauses in the negated conjecture and associated axioms prior to the proof search):

1. Fraction of Clauses that are Unit Clauses (i.e. clauses containing a single literal).

2. Fraction of Clauses that are Horn Clauses (i.e. clauses containing no more than one positive literal).

3. Fraction of Clauses that are Ground Clauses (i.e. clauses with no variables).

4. Fraction of Clauses that are Demodulators (see background chapter for a description of demodulation).

5. Fraction of Clauses that are Re-write Rules (see background chapter).

6. Fraction of Clauses that are purely positive

7. Fraction of Clauses that are purely negative

8. Fraction of Clauses that are mixed positive and negative

9. Maximum Clause Length

10. Average Clause Length

11. Maximum Clause Depth

12. Average Clause Depth

13. Maximum Clause Weight

14. Average Clause Weight

A.2.2 Dynamic feature set

There are thirty nine dynamic features. One reason for there being many more dynamic features than static ones is that during the proof process there are two sets of clauses in the proof state, processed clauses (P) and unprocessed clauses (U) whilst the static features are measured on the single initial clause set (the axioms).

Note, the dynamic features are measured at a point of the proof search when one hundred selected clauses have been processed.

The dynamic features are as follows:

1. Proportion of Generated Clauses that are kept (clauses that are subsumed or are trivial are discarded).

2. Sharing Factor (measure of the number of shared terms - the E theorem prover provides a function for calculating the sharing factor and Stephan Schulz, the author of E, has indicated in private correspondence that he'd noted that sharing factor seems to correlate with how quickly some proofs are found). Note that E does not store separate copies of shared terms, this increases efficiency as terms need only be rewritten once.

3. Ratio of Number of Clauses in P/Number in (P + U), i.e. the size of the saturated clause set relative to the total number of clauses in the current proof state.

4. Size of U/Original Size of U (ie the number of Axioms). This should be a measure as to how rapidly the number of generated clauses has grown given that the measure is taken after a fixed number of clauses has been selected as the given clause.

5. Ratio of Longest Clause Length in P to Longest Axiom Clause Length.

6. Ratio of Average Clause Length in P to Average Axiom Clause Length.

7. Ratio of Longest Clause Length in U to Longest Axiom Clause Length.

8. Ratio of Average Clause Length in U to Average Axiom Clause Length.

9. Ratio of Maximum Clause Depth in P to Maximum Axiom Clause Depth.

10. Ratio of Average Clause Depth in P to Average Axiom Clause Depth.

11. Ratio of Maximum Clause Depth in U to Maximum Axiom Clause Depth.

12. Ratio of Average Clause Depth in U to Average Axiom Clause Depth.

13. Ratio of Maximum Clause Standard Weight in P to Maximum Axiom Clause Standard Weight.

14. Ratio of Average Clause Standard Weight in P to Average Axiom Clause Standard Weight.

15. Ratio of Maximum Clause Standard Weight in U to Maximum Axiom Clause Standard Weight.

16. Ratio of Average Clause Standard Weight in U to Average Axiom Clause Standard Weight.

17. Ratio of the number of trivial clauses to the total number of processed clauses. (Trivial clauses are those that are trivially true, because they either contain a literal and its negation, or they contain a literal of the form t = t.)

18. Ratio of the number of forward subsumed clauses to the total number of processed clauses.

19. Ratio of the number of non-trivial clauses to the total number of processed clauses, this should be effectively the same as feature 17 above.

20. Ratio of the number of other redundant clauses to the total number of processed clauses.

21. Ratio of the number of non-redundant deleted clauses to the total number of processed clauses.

22. Ratio of the number of backward subsumed clauses to the total number of processed clauses.

23. Ratio of the number of backward rewritten clauses to the total number of processed clauses.

24. Ratio of the number of backward rewritten literal clauses to the total number of processed clauses.

25. Ratio of the number of generated clauses to total number of processed clauses.

26. Ratio of the number of generated literal clauses to the total number of processed clauses.

27. Ratio of the number of generated non-trivial clauses to the total number of processed clauses.

Note that in the following context_sr_count, factor_count and resolv_count are counters maintained by E which were embodied into features as described.

28. Ratio context_sr_count to the total number of processed clauses (clauses generated from a contextual or top level simplify-reflect also known as contextual literal cutting or subsumption resolution inference step - see the E user guide for details).

29. Ratio of paramodulations to the total number of processed clauses.

30. Ratio of factor_count (the number of factors found) to the total number of processed clauses.

31. Ratio of resolv_count (resolvant count) to the total number of processed clauses.

32. Fraction of total clauses in U that are Unit.

33. Fraction of total clauses in U that are Horn.

34. Fraction of total clauses in U that are Ground Clauses.

35. Fraction of total clauses in U that are demodulators.

36. Fraction of total clauses in U that are Re-write Rules.

37. Fraction of total clauses in U that contain only positive literals.

38. Fraction of total clauses in U that contain only negative literals.

39. Fraction of total clauses in U that contain both positive and negative literals.

Note, that in the above there is some redundancy between features, but the process of machine learning should automatically ignore irrelevant or redundant input.

Appendix B

Details of heuristics

This appendix gives details of the heuristic used in the initial experiment and the heuristics used in the working set for the heuristic selection experiments. There are a large number of options associated with each heuristic and the following does not provide an explanation of all, they are reproduced here to allow the same heuristics to be setup again if necessary. The E user manual (provided with the software) provides descriptions of such options as the different weighting functions for the clause selection. It should be noted though that the manual, at the time of writing, lags the software in the sense of not describing the exact weighting functions implemented in the heuristics.

B.1 Heuristic used in initial experiment

The heuristic used, on the suggestion of Stephan Schulz, was as follows (the strings beginning "–" or "-" are the parameters and brief notes are placed beneath each one). (The size of this heuristic in terms of the number of different parameters is an indication of the need for an automation process in heuristic selection.) Note that this heuristic was set using the parameters as shown rather than using one of the working set of heuristics described in the chapter on methodology. The working set of heuristics had not been programmed into the modified version of E at the stage at which the initial experiment was done.

–definitional-cnf=24

–split-aggressive –split-clauses=4

True case splitting involves considering both alternatives separately in a disjunction of literals and requires a great deal of work if back-tracking is required in the search for a proof. To get around this a more efficient method involving generation of new clauses is used, see Riazanov and Voronkov [70].

–simul-paramod

simultaneous paramodulation

–forward-context-sr

–destructive-er-aggressive –destructive-er

These two parameters are concerned with equational resolution. In equational resolution a clause containing an inequality literal is replaced by the same clause with the literal removed and a substitution made throughout the clause using the most general unifier of the left and right sides of the inequality. The destructive term means that the original clause is not kept.

–prefer-initial-clauses

Give priority to the axioms in clause selection.

-winvfreqrank -c1

Sort symbols by inverse frequency.

-Ginvfreq -F1

Sort symbols by inverse frequency.

-WSelectMaxLComplexAvoidPosPred

priority queues for clause selection

-H'(20*ConjectureRelativeSymbolWeight(ConstPrio,0.1, 100, 100, 100, 100, 1.5, 1.5, 1.5), 1*Refinedweight(PreferNonGoals,2,1,2,3,0.8), 1*FIFOWeight(ConstPrio))'

Main, round robin scheme for organizing queues from which the next selected clause is taken. In this case there are three queues, the first of which is used twenty times more often than the second or the third. The first two queues are clause weighting schemes whilst the third queue is based on how long the clause has existed (so every clause has some chance of being selected even if it has a low weight in both the other queues).

B.2 Heuristics used in working set

The heuristics were selected on the basis of the number of cases from the TPTP library for which the auto mode in E would select that heuristic. The auto mode in E classes problems according to a limited number of binary and ternary features. Stephan Schulz, the author of E, did extensive work testing different heuristics and finding the best for each class. The information regarding class size and best heuristic is contained in the source code to E and it was this information that was used to select a working set of heuristics for the work described in this dissertation.

B.2.1 Heuristic 1

Heuristic 1 was the global best heuristic (in 2442 cases in testing over conjectures from the TPTP library).

For heuristic 1 the labeling in E is

"G_E___021_K31_F1_PI_AE_S4_CS_SP_S2S"

which breaks down as follows:

"G_E___021" : simple label of no long term significance,
"_K31" : type of term ordering (KBO with the 31st method investigated by Stephan Schulz),
"_F1" : limited forward rewriting of new clauses,
"_PI" : Prefer Initial clauses (*i.e.* process the original problem clauses before any derived ones),
"_AE" : Aggressive Equality resolution ($X \neq Y \vee R$ is simplified to $R(X \leftarrow Y)$ even for newly generated clauses),
"_S4" : Split strategy 4,
"_SP" : Simultaneous Paramodulation,

Heuristic parameters are set (in che_X___________auto.c) as

prefer_initial_clauses = true;
forward_context_sr = true;
selection_strategy = SelectMaxLComplexAvoidPosPred;
split_clauses = 4;
split_fresh_defs = false;
er_varlit_destructive = true;
er_aggressive = true;
forward_demod = 1;
pm_type = ParamodAlwaysSim;

Ordering parameters are set as

to_weight_gen = WInvFrequencyRank;
to_prec_gen = PByInvFrequency;
to_const_weight = 1;

Clause selection queues are set with the following weightings:

(4*ConjectureGeneralSymbolWeight(SimulateSOS,100,100,100,50,50,10,10,1.5,1.5,1),
3*ConjectureGeneralSymbolWeight(PreferNonGoals,200,100,200,50,50,1,100,1.5,1.5,1),
1*Clauseweight(PreferProcessed,1,1,1),
1*FIFOWeight(PreferProcessed))

B.2.2 Heuristic 2

Heuristic 2 was the best heuristic in 437 cases.

For heuristic 2 the labeling in E is

“______081_B31_F1_PI_AE_S4_CS_SP_S0Y”

“H______081” : is a simple label,
“_B31” : corresponds to a Lexical Path term ordering option,
“_F1” : limited forward rewriting of new clauses,
“_PI” : Prefer Initial clauses (i.e. process the original problem clauses before any derived ones),
“_AE” : Aggressive Equality resolution ($X \neq Y \vee R$ is simplified to $R(X \leftarrow Y)$ even for newly generated clauses),
“_S4” : Split strategy 4,
“_SP” : Simultaneous Paramodulation,

Heuristic parameters are set (in che_X__________auto.c) as

prefer_initial_clauses = true;
forward_context_sr = true;
selectction_strategy = SelectMaxLComplexAvoidPosPred;
split_clauses = 4;
split_aggressive = true;
er_varlit_destructive = true;
er_aggressive = true;
forward_demod = 1;
pm_type = ParamodAlwaysSim;

Ordering parameters are set as

ordertype = LPO4;
to_prec_gen = PByInvFreqConstMin;

Clause selection queues are set with the following weightings:

(8*Refinedweight(PreferGoals,1,2,2,2,2),
8*Refinedweight(PreferNonGoals,2,1,2,2,0.5),
1*Clauseweight(PreferUnitGroundGoals,1,1,1),
1*FIFOWeight(ConstPrio))

B.2.3 Heuristic 3

Heuristic 3 was the best heuristic in 377 cases.

For heuristic 3 the labeling in E is

“H_____047_K18_F1_PI_AE_R4_CS_SP_S2S”

“_K18” : type of term ordering (KBO with the 18th method investigated by Stephan Shulz),
“_F1” : limited forward rewriting of new clauses,
“_PI” : Prefer Initial clauses (i.e. process the original problem clauses before any derived

ones),
"_AE" : Aggressive Equality resolution ($X \neq Y \vee R$ is simplified to $R(X \leftarrow Y)$ even for newly generated clauses),
"_R4" : Split strategy 4, but with re-use of old split definitions,
"_SP" : Simultaneous Paramodulation,

Heuristic parameters are set (in che_X___________auto.c) as

prefer_initial_clauses = true;
forward_context_sr = true;
selection_strategy = SelectNewComplexAHP;
split_clauses = 4;
split_aggressive = true;
split_fresh_defs = false;
er_varlit_destructive = true;
er_aggressive = true;
forward_demod = 1;
pm_type = ParamodAlwaysSim;

Ordering paramaters are set as

to_weight_gen = WInvFrequencyRank;
to_prec_gen = PByInvFrequency;
to_const_weight = 1;

Clause selection queues are set with the following weightings:

(10*PNRefinedweight(PreferGoals,1,1,1,2,2,2,0.5),
10*PNRefinedweight(PreferNonGoals,2,1,1,1,2,2,2),
5*OrientLMaxWeight(ConstPrio,2,1,2,1,1),
1*FIFOWeight(ConstPrio))

B.2.4 Heuristic 4

Heuristic 4 was the best heuristic in 329 cases.

For heuristic 4 the labeling in E is

"G_E____008_K18_F1_PI_AE_CS_SP_S0Y"

"_K18" : type of term ordering (KBO with the 18th method investigated by Stephan Shulz),
"_F1" : limited forward rewriting of new clauses,
"_PI" : Prefer Initial clauses (i.e. process the original problem clauses before any derived ones),
"_AE" : Aggressive Equality resolution ($X \neq Y \vee R$ is simplified to $R(X \leftarrow Y)$ even for newly generated clauses),
"_SP" : Simultaneous Paramodulation,

Heuristic parameters are set (in che_X__________auto.c) as

prefer_initial_clauses = true;
forward_context_sr = true;
selection_strategy = SelectMaxLComplexAvoidPosPred;
er_varlit_destructive = true;
er_aggressive = true;
forward_demod = 1;
pm_type = ParamodAlwaysSim;

Ordering parameters are set as

to_weight_gen = WInvFrequencyRank;
to_prec_gen = PByInvFrequency;
to_const_weight = 1;

Clause selection queues are set with the following weightings:

(10*ConjectureRelativeSymbolWeight(ConstPrio,0.1,100,100,100,100,1.5,1.5,1.5),
1*FIFOWeight(ConstPrio))

B.2.5 Heuristic 5

Heuristic 5 was the best heuristic in 321 cases.

For heuristic 5 the labeling in E is

"G_E___008_K18_F1_PI_AE_R4_CS_SP_S2S"

"_K18" : type of term ordering (KBO with the 18th method investigated by Stephan Shulz),
"_F1" : limited forward rewriting of new clauses,
"_PI" : Prefer Initial clauses (i.e. process the original problem clauses before any derived ones),
"_AE" : Aggressive Equality resolution ($X \neq Y \vee R$ is simplified to $R(X \leftarrow Y)$ even for newly generated clauses),
"_R4" : Split strategy 4, but with re-use of old split definitions,
"_SP" : Simultaneous Paramodulation,

Heuristic parameters are set (in che_X__________auto.c) as

prefer_initial_clauses = true;
forward_context_sr = true;
selection_strategy = SelectNewComplexAHP;
split_clauses = 4;
split_aggressive = true;
split_fresh_defs = false;
er_varlit_destructive = true;
er_aggressive = true;
forward_demod = 1;

pm_type = ParamodAlwaysSim;

Ordering parameters are set as

to_weight_gen = WInvFrequencyRank;
to_prec_gen = PByInvFrequency;
to_const_weight = 1;

Clause selection queues are set with the following weightings:

(10*ConjectureRelativeSymbolWeight(ConstPrio,0.1,100,100,100,100,1.5,1.5,1.5),
1*FIFOWeight(ConstPrio))

Appendix C

Results of varying parameter C

As covered in the background chapter, support vector machines designed for soft margin classification allow a trade-off between the size of margin and the training error. The parameter governing this trade-off is C. In the experimental work described in the main body of this dissertation the parameter C was left at its default value which is equal to the average value of

$$(\boldsymbol{x_i} \cdot \boldsymbol{x_i})^{-1}$$

For the sake of completeness this appendix presents the results of varying the value of C to compare with the default results. To keep the number of combinations manageable the value of C was kept the same for all six heuristic classifiers. The value of the parameter γ was fixed at 48 which corresponds to the optimal value from varying γ. (By changing C from its default setting the conditions under which γ was optimised are altered but the assumption made was that the optimal value of γ is not strongly affected by the value of C.)

C.1 Results for subset with features 7 and 52

In the subset experiments described in the main body of the dissertation the best subset with H0 filtering contained only the features 7 and 52.

Figure C.1 shows the effect of varying the parameter C on the total number of theorems proved with features restricted to the optimal subset consisting of features 7 and 52 only and H0 filtering. This was found to be the optimal subset where H0 filtering is included. Figure C.2 shows the plot of total time taken and comparing the two figures shows that the drop in theorems proved for values of C above 4 corresponds to a drop in time taken which implies a too drastic H0 filtering for these higher values of C.

To confirm that the drop in theorems and time were indeed due to the effects of H0 filtering, the experiment was repeated without H0 filtering and the results are shown in figures C.3 and C.4. It can be seen that without H0 filtering the number of theorems proved does not drop with higher C values but there is a corresponding increase in time taken.

For this subset the optimal value of C is around 2 but the results are not better than those obtained with C set to its default (the default setting of C with H0 filtering and a γ value of 48 led to 1,602 theorems being proved).

The results in this case are worse than those for the default setting of C with γ at 10. This implies that the optimal value of γ is not 48 for this feature subset.

C.2 Results for subset with features 10, 14 and 15

In the subset experiments described in the main body of the dissertation the best subset with no H0 filtering contained features 10, 14 and 15.

Figure C.5 shows the effect of varying parameter C on the number of theorems proved with the subset containing features 10, 14 and 15 and with H0 filtering included. Figure C.6 shows the corresponding graph of total time taken.

Figure C.7 shows the effect of varying parameter C on the number of theorems proved with the subset containing features 10, 14 and 15 and without any H0 filtering. Figure C.8 shows the corresponding graph of total time taken.

C.3 Results for the full feature set

Figure C.9 shows the effect of varying parameter C on the number of theorems proved for the full feature set with H0 filtering included. Figure C.10 shows the corresponding graph of total time taken.

Figure C.11 shows the effect of varying parameter C on the number of theorems proved for the full feature set without any H0 filtering. Figure C.12 shows the corresponding graph of total time taken.

Again, there is no improvement obtained by setting C rather than leaving it at its default setting.

C.4 Extending the range of C variation up to 10,000

The variation in the parameter C described so far has been limited to values up to 10 in linear steps. The experiments were repeated using logarithmic variation up to 10,000. Figures C.13 to C.24 show the extended curves. The plots are of $Log_{10}(C)$ so that 4 corresponds to a value of C of 10,000. It can be seen that for large values of C above about 500 there is a lot of noise with wide fluctuations from point to point. It can also be seen that extending the range of values of C does not reveal any new optimal points beyond those seen in the initial curves where the value of C ranged up to 10.

C.5 Conclusions

For completeness it is useful to have looked at the effect of varying parameter C but the results obtained are not better than those obtained when C is left at its default setting. The approach taken in the main body of this dissertation, of using the default setting for parameter C, is therefore justified.

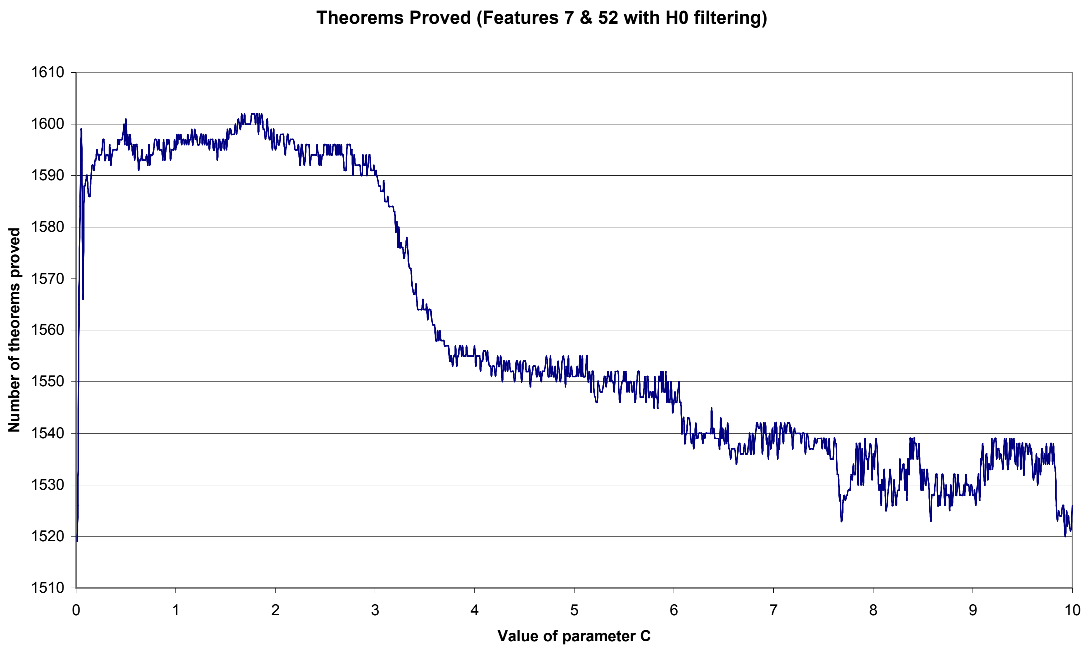

Figure C.1: Effect of varying parameter C for subset {7,52},$\gamma = 48$, H0 filtering used

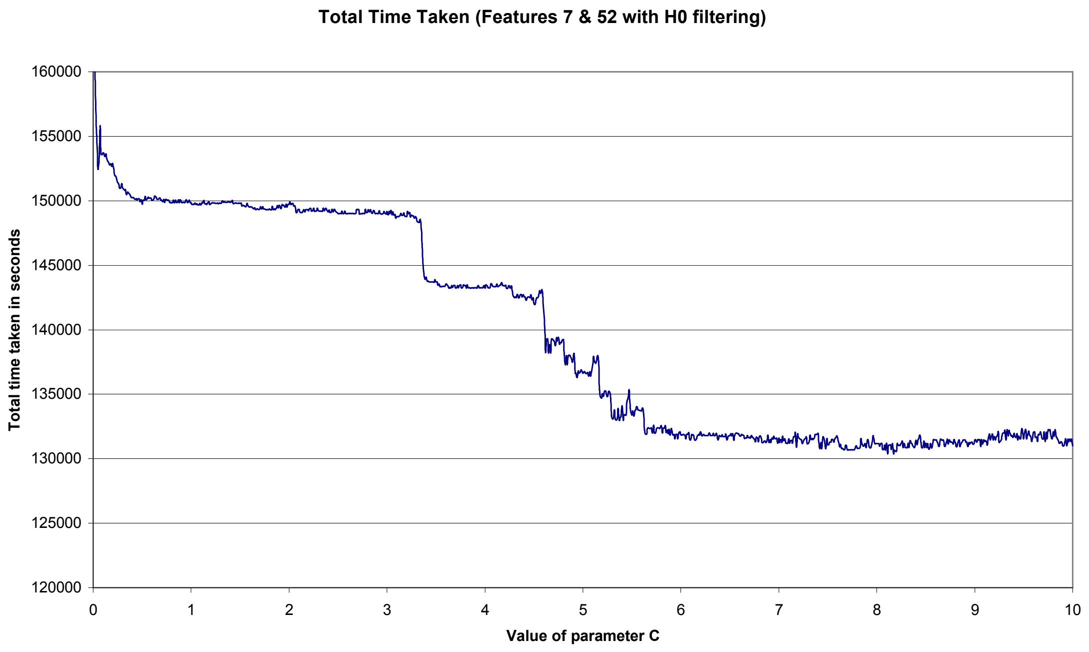

Figure C.2: Effect of varying parameter C for subset {7,52},$\gamma = 48$, H0 filtering used

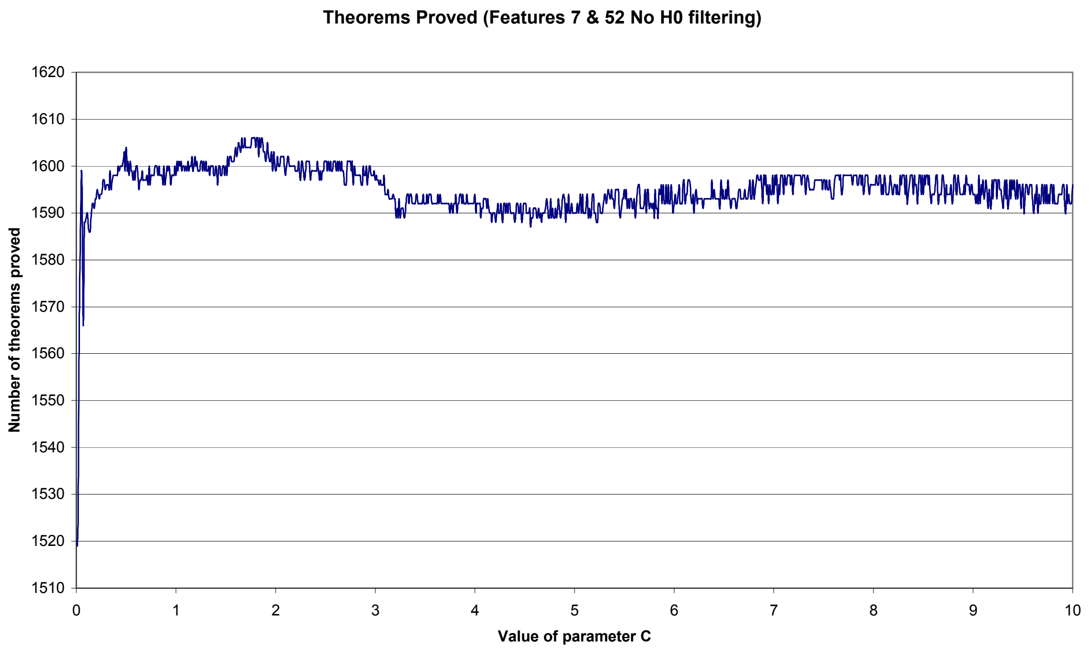

Figure C.3: Effect of varying parameter C for subset {7,52},$\gamma = 48$, No H0 filtering used

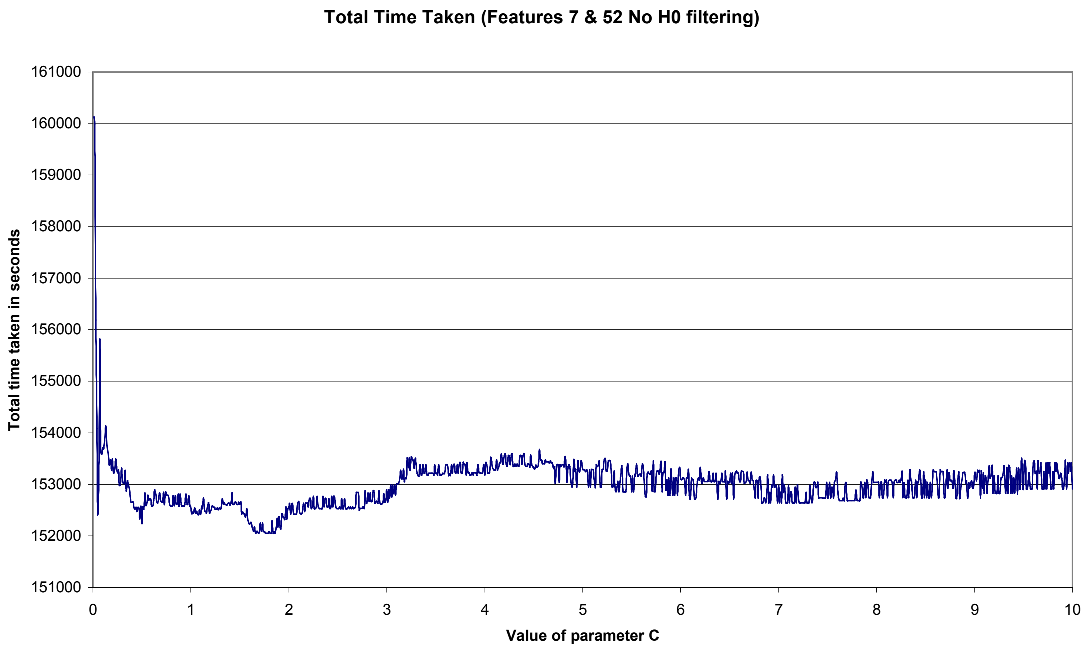

Figure C.4: Effect of varying parameter C for subset {7,52},$\gamma = 48$, No H0 filtering used

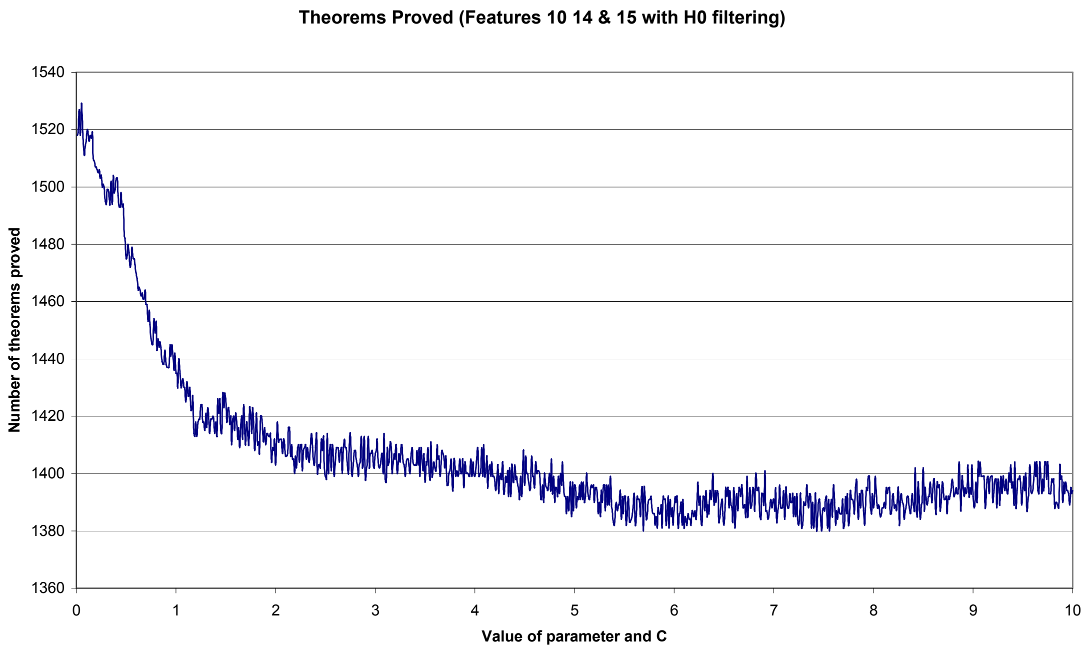

Figure C.5: Effect of varying parameter C for subset {10,14,15},$\gamma = 48$, H0 filtering used

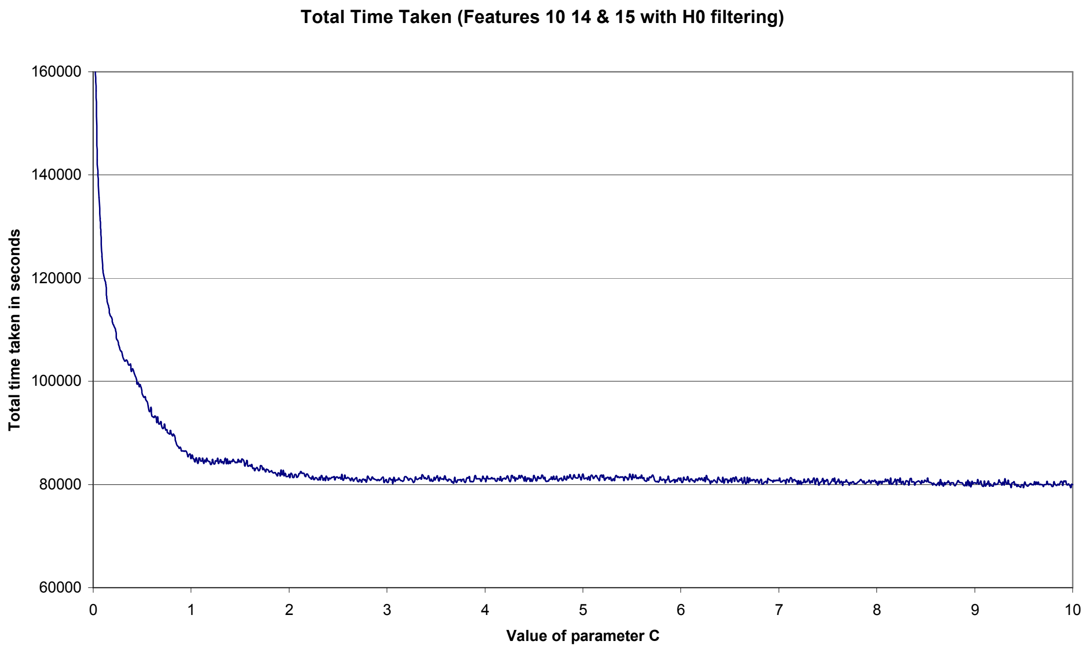

Figure C.6: Effect of varying parameter C for subset {10,14,15},$\gamma = 48$, H0 filtering used

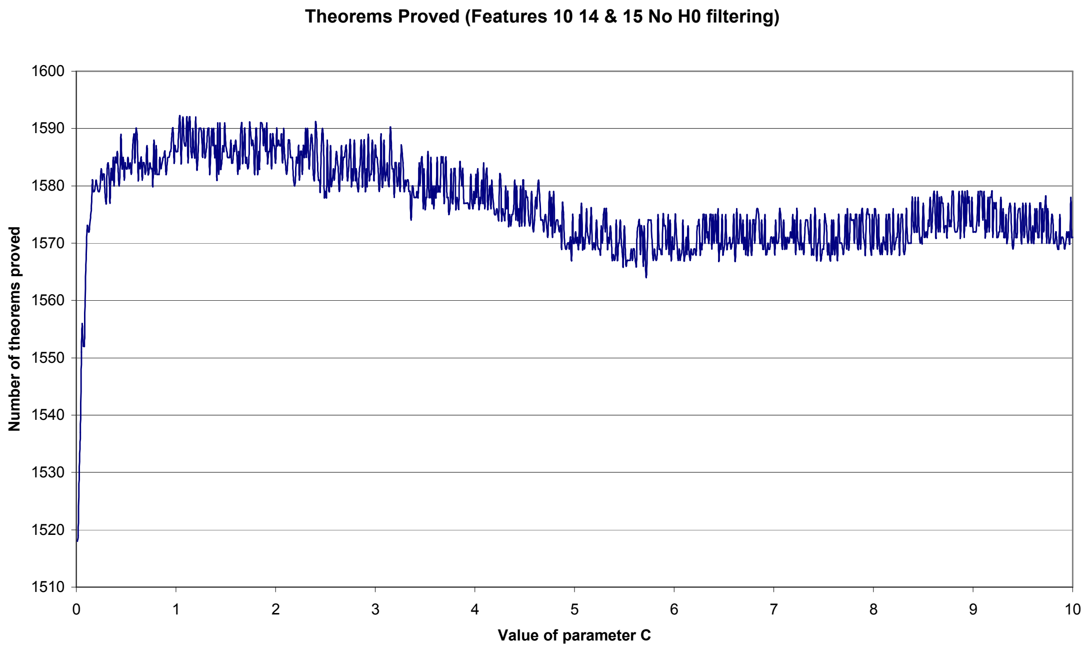

Figure C.7: Effect of varying parameter C for subset {10,14,15},$\gamma = 48$, No H0 filtering used

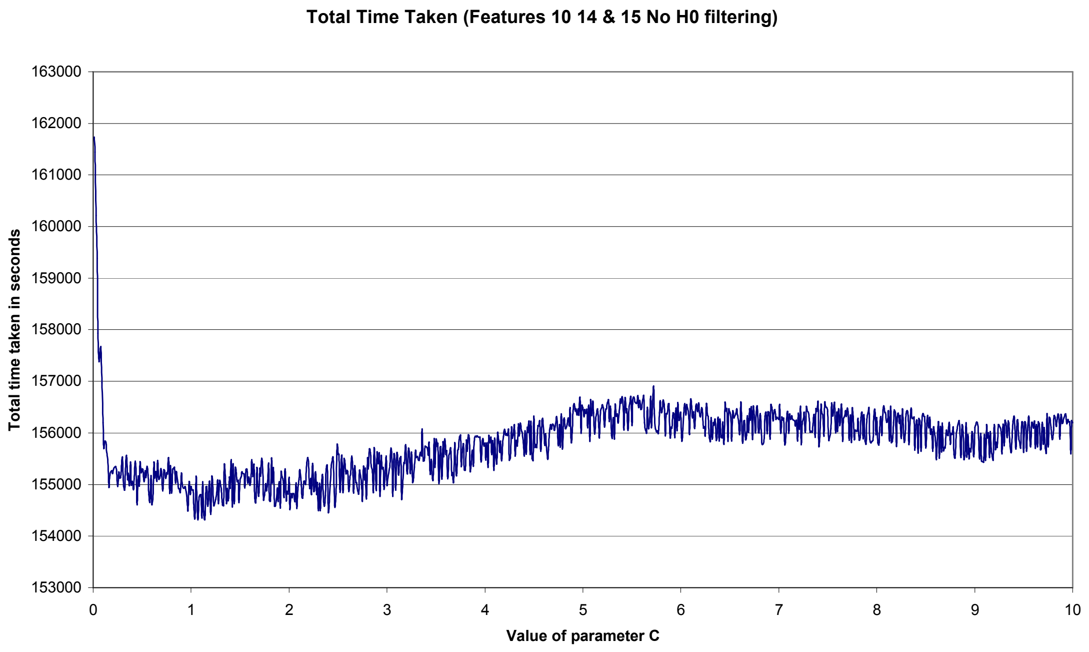

Figure C.8: Effect of varying parameter C for subset {10,14,15},$\gamma = 48$, No H0 filtering used

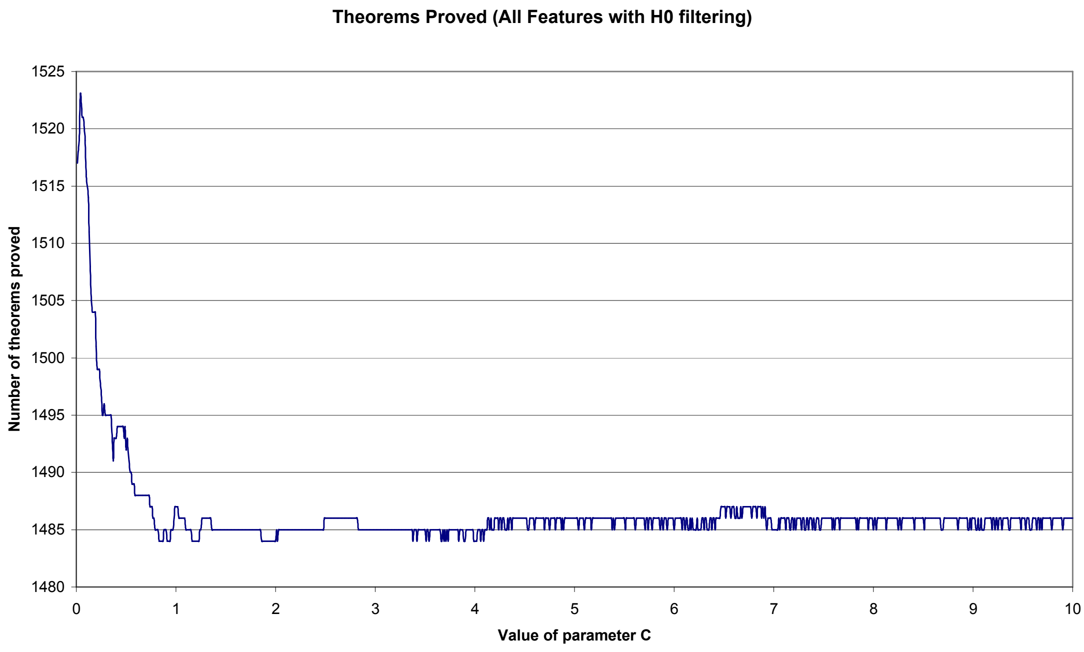

Figure C.9: Effect of varying parameter C for full feature set,$\gamma = 48$, H0 filtering used

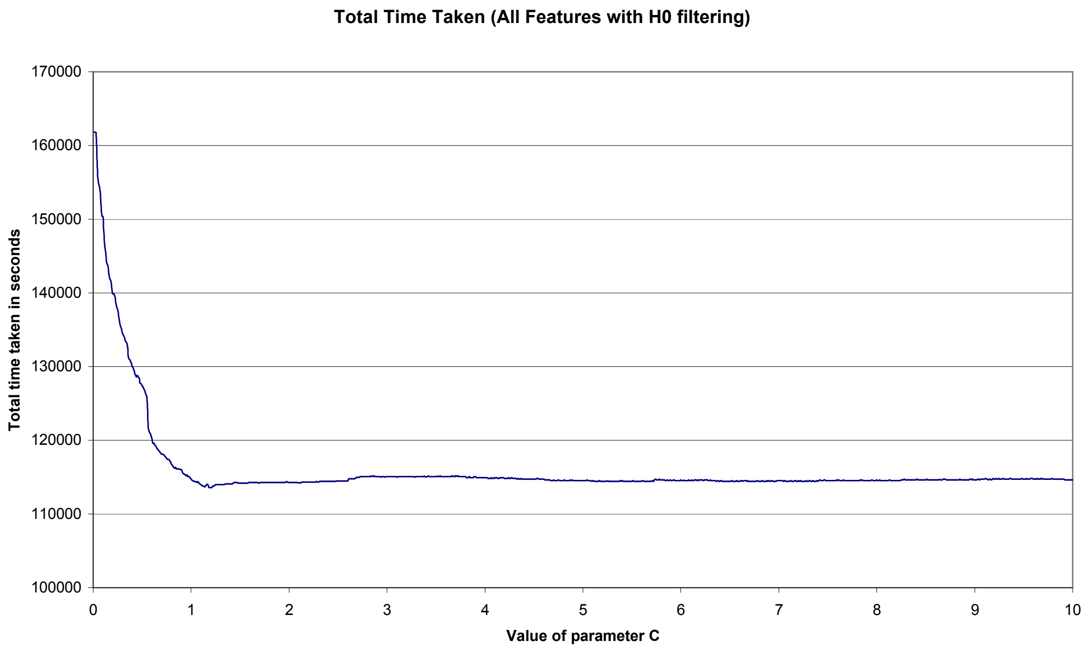

Figure C.10: Effect of varying parameter C for full feature set,$\gamma = 48$, H0 filtering used

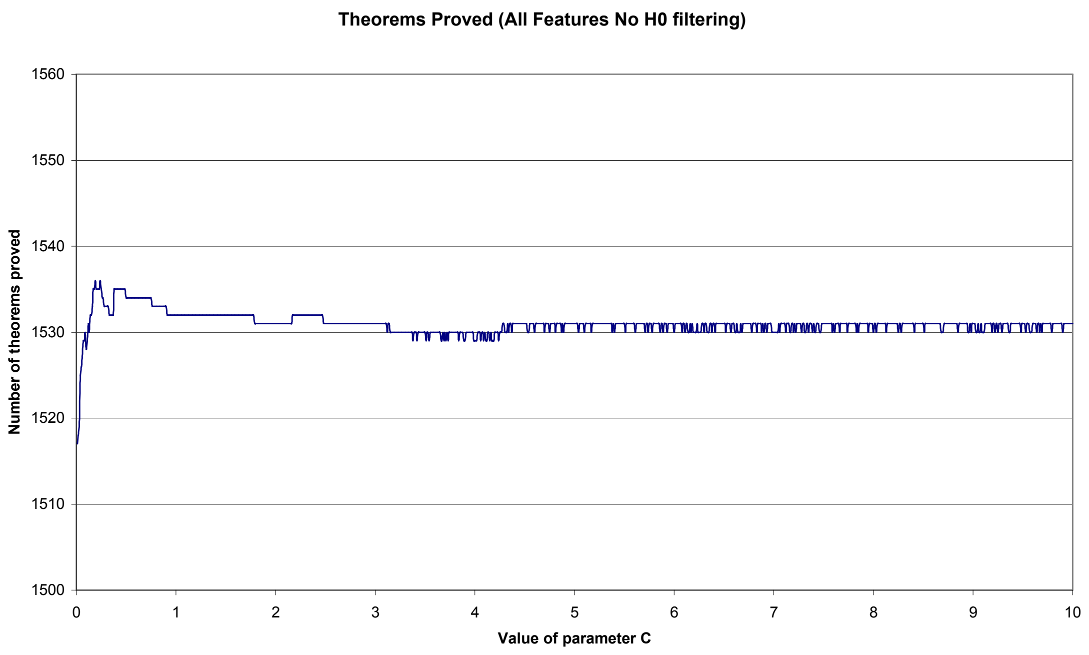

Figure C.11: Effect of varying parameter C for full feature set,γ = 48, No H0 filtering used

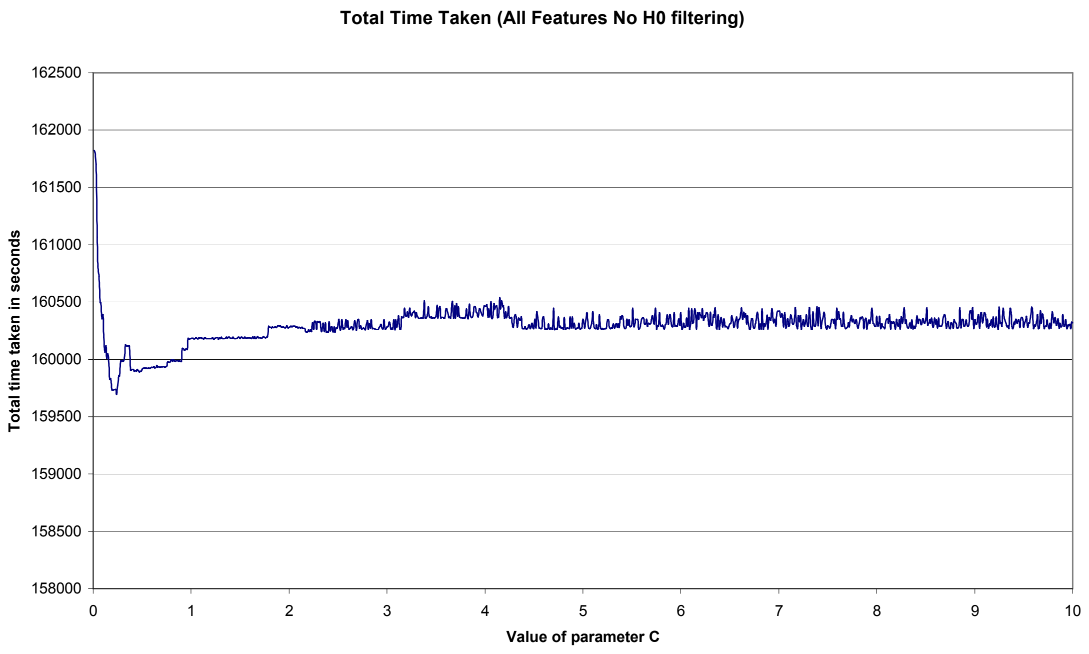

Figure C.12: Effect of varying parameter C for full feature set,γ = 48, No H0 filtering used

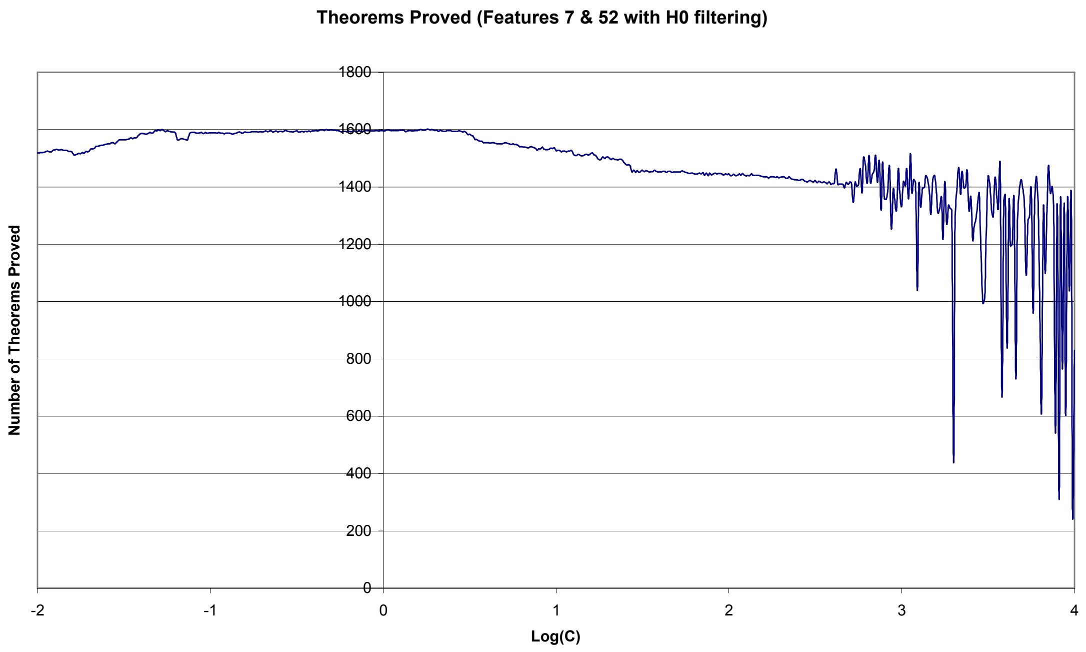

Figure C.13: Effect of varying $Log_{10}(C)$ for subset {7,52},$\gamma = 48$, H0 filtering used

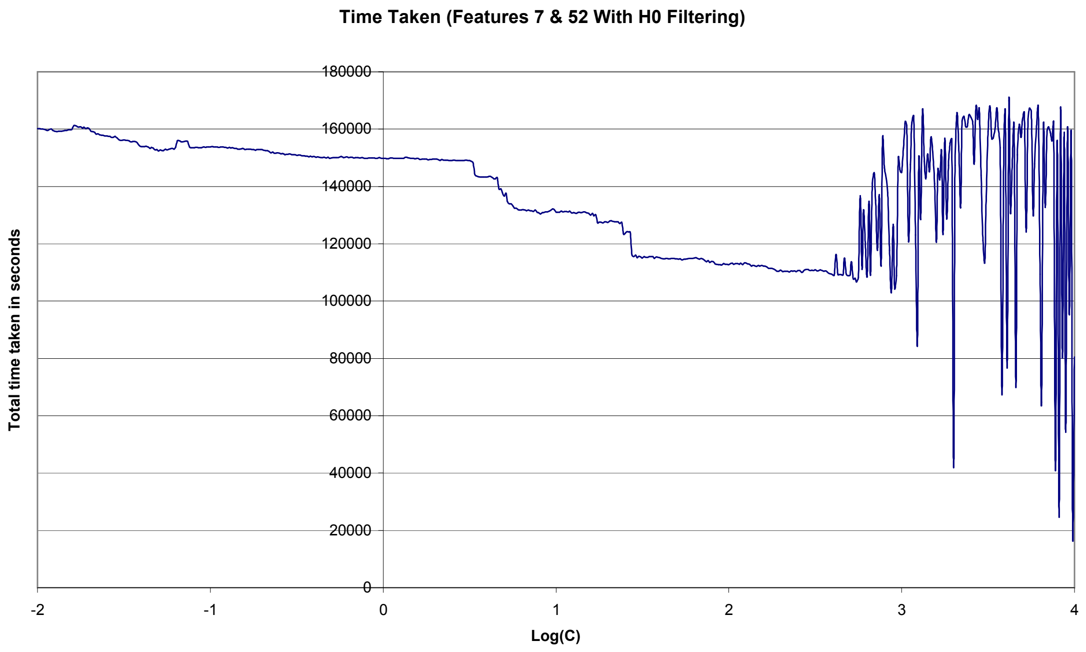

Figure C.14: Effect of varying parameter C for subset {7,52},$\gamma = 48$, H0 filtering used

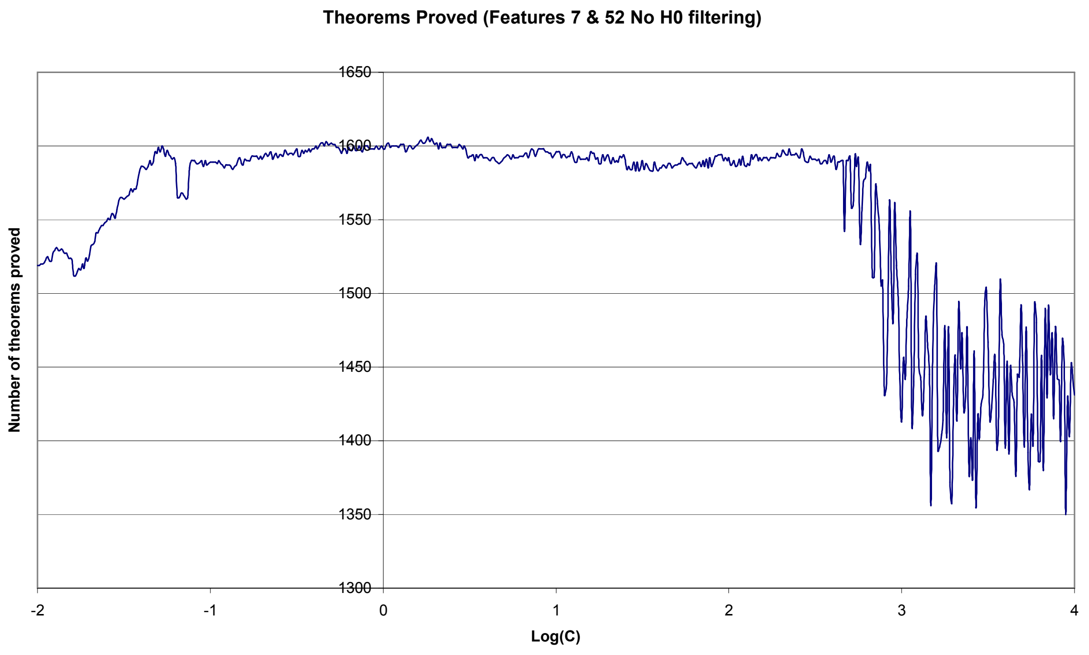

Figure C.15: Effect of varying $Log_{10}(C)$ for subset {7,52},$\gamma = 48$, No H0 filtering used

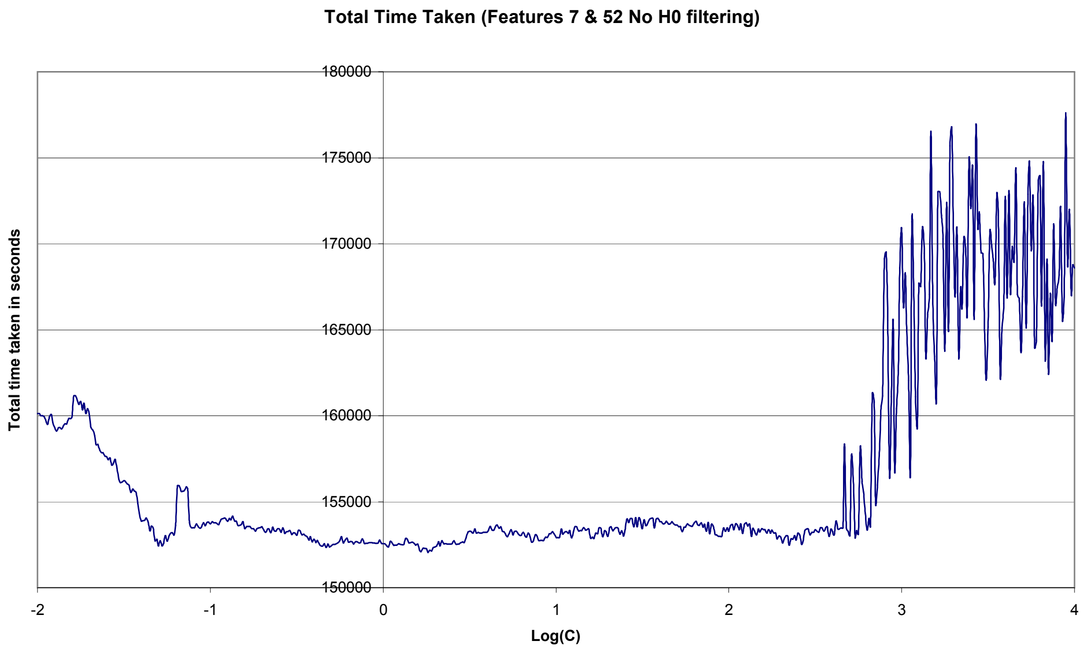

Figure C.16: Effect of varying $Log_{10}(C)$ for subset {7,52},$\gamma = 48$, No H0 filtering used

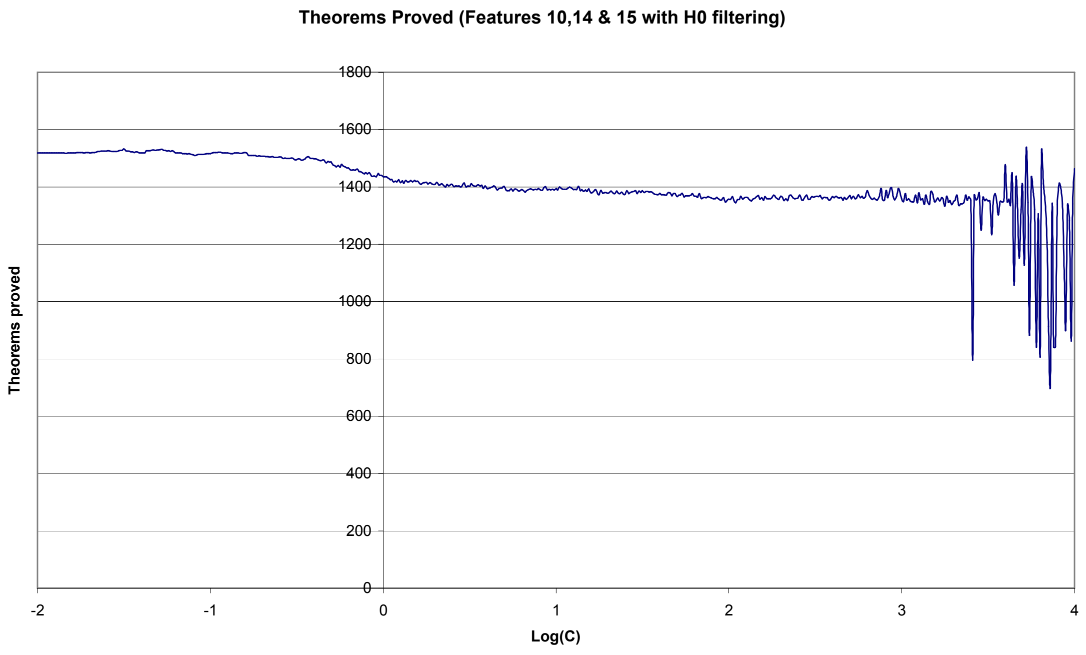

Figure C.17: Effect of varying $Log_{10}(C)$ for subset {10,14,15},$\gamma = 48$, H0 filtering used

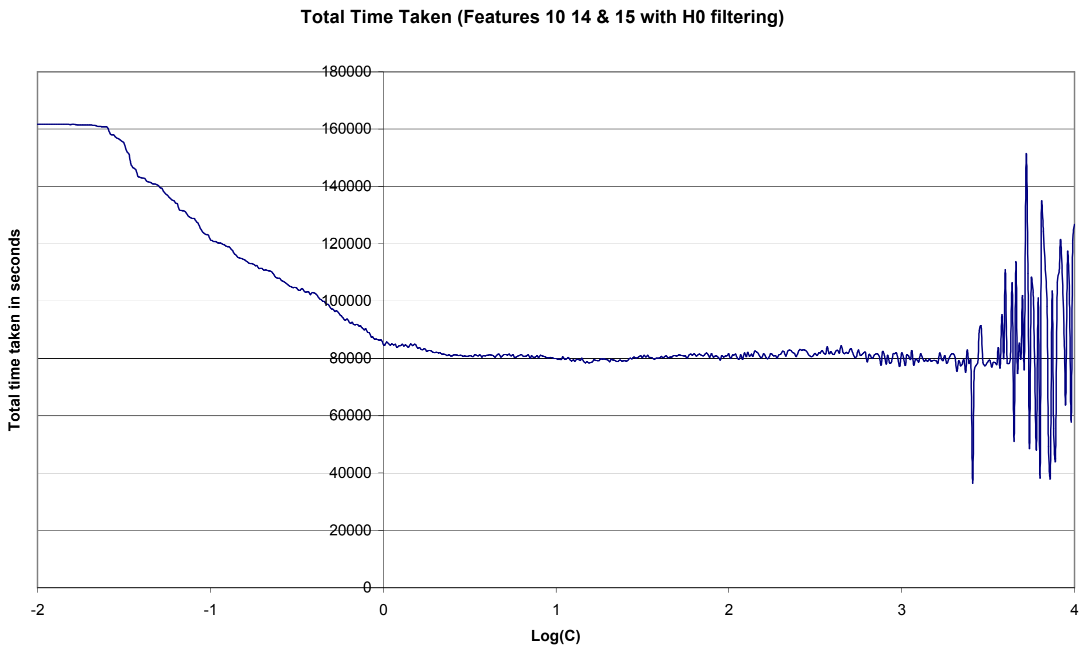

Figure C.18: Effect of varying $Log_{10}(C)$ for subset {10,14,15},$\gamma = 48$, H0 filtering used

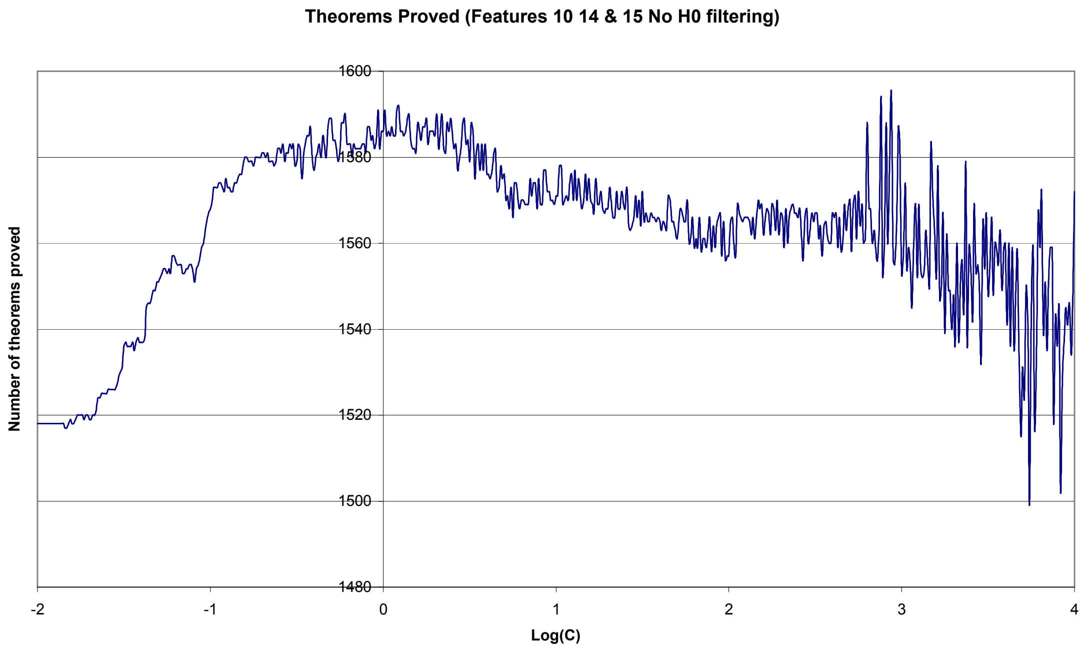

Figure C.19: Effect of varying $Log_{10}(C)$ for subset {10,14,15},γ = 48, No H0 filtering used

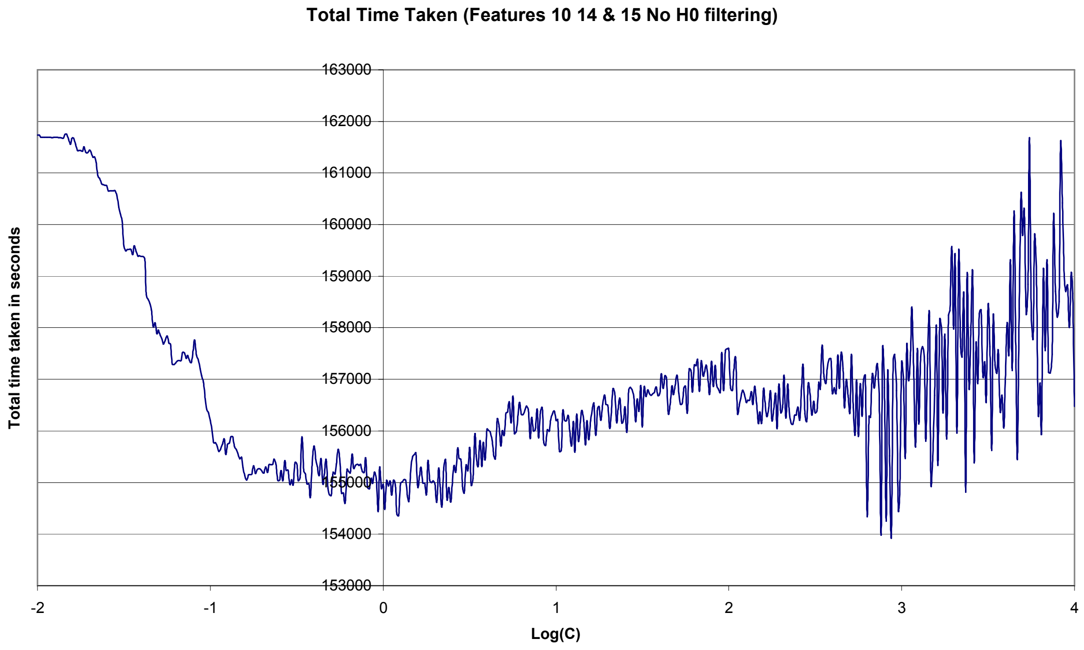

Figure C.20: Effect of varying $Log_{10}(C)$ for subset {10,14,15},γ = 48, No H0 filtering used

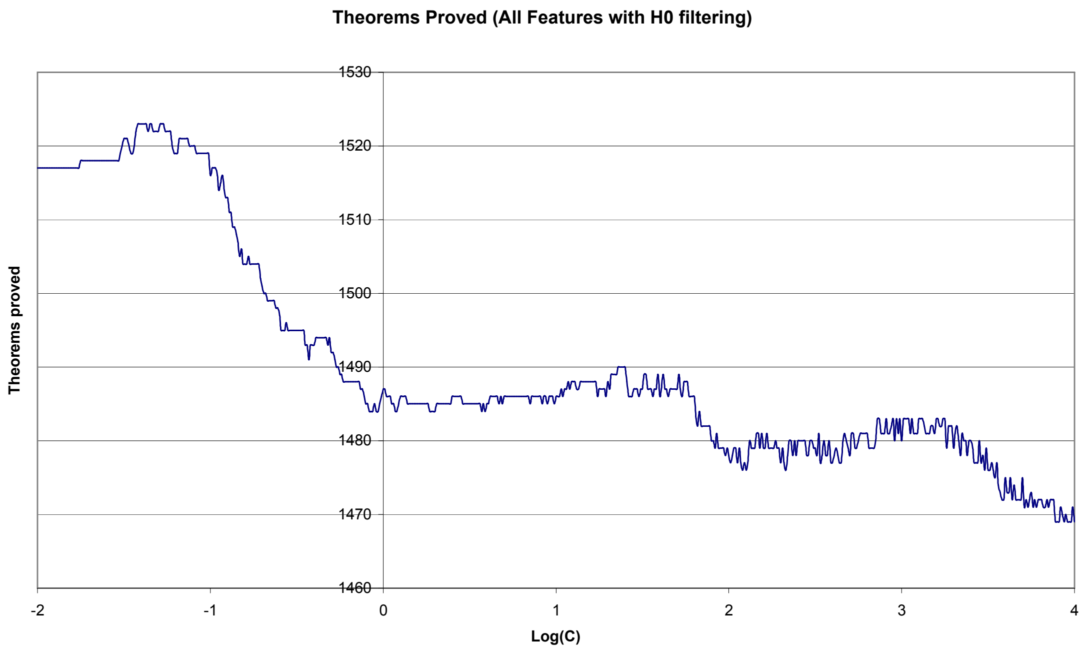

Figure C.21: Effect of varying $Log_{10}(C)$ for full feature set,$\gamma = 48$, H0 filtering used

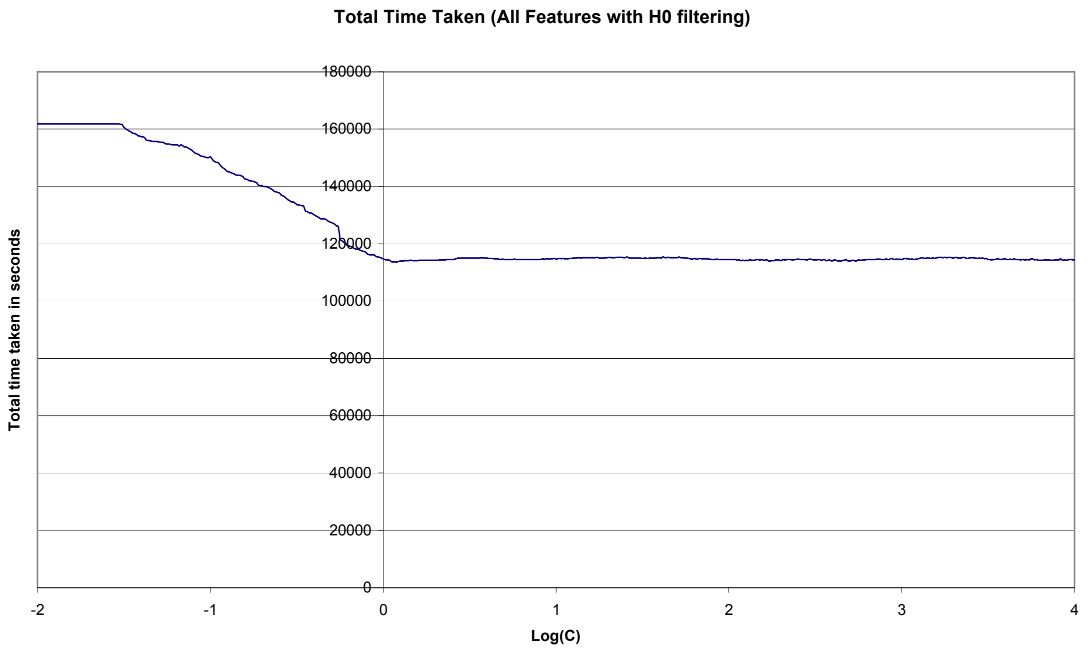

Figure C.22: Effect of varying $Log_{10}(C)$ for full feature set,$\gamma = 48$, H0 filtering used

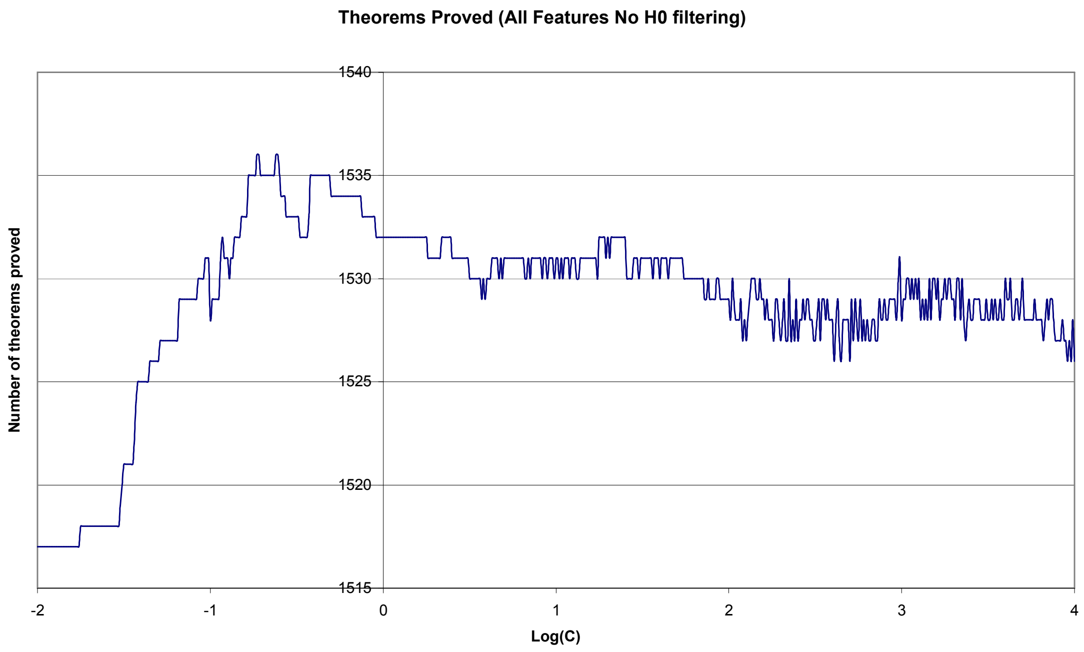

Figure C.23: Effect of varying $Log_{10}(C)$ for full feature set,$\gamma = 48$, No H0 filtering used

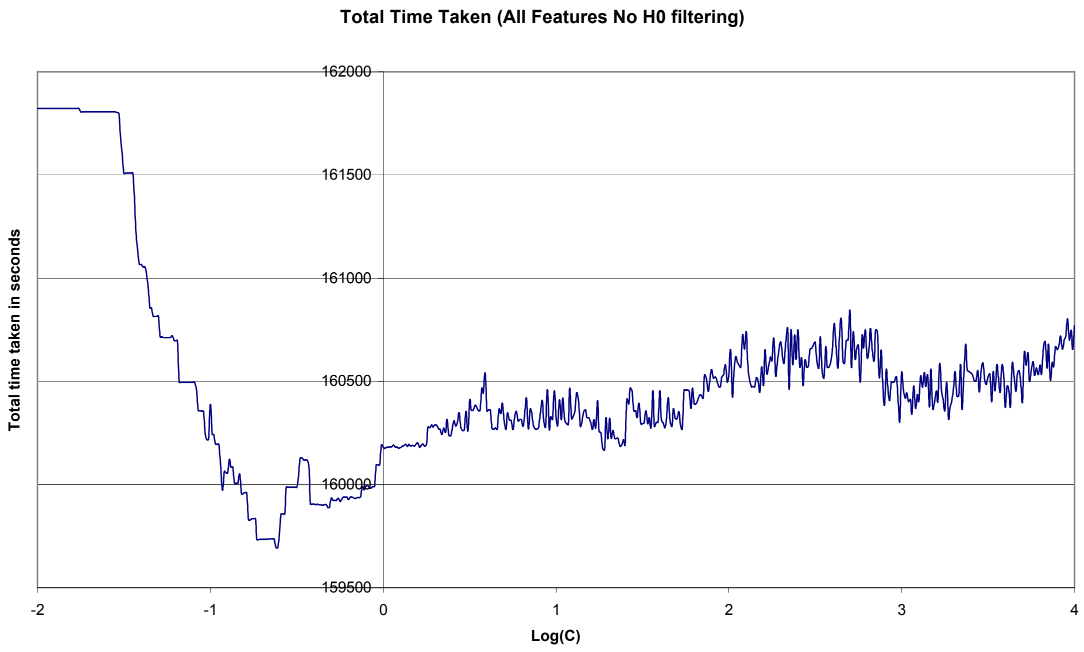

Figure C.24: Effect of varying $Log_{10}(C)$ for full feature set,$\gamma = 48$, No H0 filtering used

Index