Kvasir: Scalable Provision of Semantically Relevant Web Content on Big Data Framework

Liang Wang[1], Sotiris Tasoulis[2], Teemu Roos[3], and Jussi Kangasharju[3]

University of Cambridge, UK[1] Liverpool John Moores University, UK[2] University of Helsinki, Finland[3]

Abstract—The Internet is overloading its users with excessive information flows, so that effective content-based filtering becomes crucial in improving user experience and work efficiency. Latent semantic analysis has long been demonstrated as a promising information retrieval technique to search for relevant articles from large text corpora. We build Kvasir, a semantic recommendation system, on top of latent semantic analysis and other state-of-the-art technologies to seamlessly integrate an automated and proactive content provision service into web browsing. We utilize the processing power of Apache Spark to scale up Kvasir into a practical Internet service. In addition, we improve the classic randomized partition tree to support efficient indexing and searching of millions of documents. Herein we present the architectural design of Kvasir, the core algorithms, along with our solutions to the technical challenges in the actual system implementation.

Index Terms—Web application, information retrieval, semantic search, random projection, machine learning, Apache Spark.

1 Introduction

Currently, the Internet is overloading its users with excessive information flows. Therefore, smart content provision and recommendation become more and more crucial in improving user experience and efficiency in using Internet applications. For a typical example, many users are most likely to read several articles on the same topic while surfing on the Web. Hence many news websites (e.g., The New York Times, BBC News and Yahoo News) usually group similar articles together and provide them on the same page so that the users can avoid launching another search for the topic. However, most such services are constrained within a single domain, and cross-domain content provision is usually achieved by manually linking to the relevant articles on different sites. Meanwhile, companies like Google and Microsoft take advantage of their search engines and provide customizable keyword filters to aggregate related articles across various domains for users to subscribe. However, to subscribe a topic, a user needs to manually extract keywords from an article, then to switch between different search services while browsing the web pages.

In general, seamless integration of intelligent content provision into web browsing at user interface level remains an open research question. No universally accepted optimal design exists. Herein we propose Kvasir[1], a system built on top of latent semantic analysis (LSA).[2] We show how Kvasir can be integrated with the state-of-art technologies (e.g., Apache Spark, machine learning, etc.). Kvasir automatically looks for the similar articles when a user is browsing a web page and injects the search results in an easily accessible panel within the browser view for seamless integration.

The ranking of results is based on the cosine similarity in LSA space, which was proposed as an effective information retrieval technique almost two decades ago [5]. Despite some successful applications in early information systems, two technical challenges practically prevent LSA from becoming a scalable Internet service. First, LSA relies on large matrix multiplications and singular value decomposition (SVD), which become notoriously time and memory consuming when the document corpus is huge. Second, LSA is a vector space model, and fast search in high dimensional spaces tends to become a bottle-neck in practice.

We must emphasize that Kvasir is not meant to replace the conventional web search engines, recommender systems, or other existing technologies discussed in Section 2. Instead, Kvasir represents another potential solution to enhance user experience in future Internet applications. In this paper, by presenting the architectural components, we show how we tackle the scalability challenges confronting Kvasir in building and indexing high dimensional language database. To address the challenge in constructing the database, we adopt a rank-revealing algorithm for dimension reduction before the actual SVD. To address the challenge in high dimensional search, we utilize approximate nearest neighbour search to trade off accuracy for efficiency. The corresponding indexing algorithm is optimized for parallel implementation.

In general, this paper focuses on the system scalability perspective when utilizing big data frameworks to scale up Internet services, and we present our solutions to the technical challenges in building up Kvasir. Specifically, our contributions are:

1) We present the architecture of Kvasir, which is able to seamlessly integrate LSA-based content provision in web browsing by using state-of-art technologies.

1. Kvasir is the acronym for *Knowledge ViA Semantic Information Retrieval*, it is also the name of a Scandinavian god in Norse mythology who travels around the world to teach and spread knowledge and is considered extremely wise.

2. We have introduced the basic Kvasir system framework in [51] and demonstrated its seamless integration into web browsing in World Wide Web Conference (WWW'15) in May, 2015.

2) We implement the first stochastic SVD on Apache Spark, which can efficiently build LSA-based language models on large text corpora.
3) We propose a parallel version of the randomized partition tree algorithm which provides fast indexing in high dimensional vector spaces using Apache Spark.[3]
4) We present various technical solutions in order to address the scalability challenges in building Kvasir into a practical Internet service, such as caching, parallel indexing and reducing index size.

The paper is structured as follows: Section 2 gives a brief overview over the various topics covered in this paper. Section 3 presents the architecture of Kvasir. In Section 4, we illustrate the algorithms of the core components in details. Section 5 presents our evaluation results. Finally, Section 6 and 7 extend the discussion then conclude the paper.

2 Background and Related Work

Inside Kvasir, the design covers a wide range of different topics, each topic has numerous related work. In the following, we constrain the discussion only on the core techniques used in the system design and implementation. Due to the space limit, we cannot list all related work, we recommend using the references mentioned in this section as a starting point for further reading.

2.1 Intelligent Content Provision

Our daily life heavily relies on recommendations, intelligent content provision aims to match a user's profile of interests to the best candidates in a large repository of options [43]. There are several parallel efforts in integrating intelligent content provision and recommendation in web browsing. They differentiate between each other by the main technique used to achieve the goal.

The initial effort relies on the semantic web stack proposed in [4], which requires adding explicit ontology information to all web pages so that ontology-based applications (e.g., Piggy bank [28]) can utilize ontology reasoning to interconnect content semantically. Though semantic web has a well-defined architecture, it suffers from the fact that most web pages are unstructured or semi-structured HTML files, and content providers lack of motivation to adopt this technology to their websites. Therefore, even though the relevant research still remains active in academia, the actual progress of adopting ontology-based methods in real-life applications has stalled in these years.

Collaborative Filtering (CF) [9], [33], which was first coined in Tapestry [22], is a thriving research area and also the second alternative solution. Recommenders built on top of CF exploit the similarities in users' rankings to predict one user's preference on a specific content. CF attracts more research interest these years due to the popularity of online shopping (e.g., Amazon, eBay, Taobao, etc.) and video services (e.g., YouTube, Vimeo, Dailymotion, etc.). However, recommender systems need user behavior rather than content itself as explicit input to bootstrap the service, and is usually constrained within a single domain. Cross-domain recommenders [15], [34] have made progress lately, but the complexity and scalability need further investigation.

Search engines can be considered as the third alternative though a user needs explicitly extract the keywords from the page then launch another search. The ranking of the search results is based on multiple ranking signals such as link analysis on the underlying graph structure of interconnected pages (e.g., PageRank [41] and HITS [32]). Such graph-based link analysis is based on the assumption that those web pages of related topics tend to link to each other, and the importance of a page often positively correlates to its degree. The indexing process is modelled as a random walk atop of the graph derived from the linked pages and needs to be pre-compiled offline.

As the fourth alternative, Kvasir takes another route by utilizing information retrieval (IR) technique [20], [36]. Kvasir belongs to the content-based filtering and emphasizes the semantics contained in the unstructured web text. In general, a text corpus is transformed to the suitable representation depending on the specific mathematical models (e.g., set-theoretic, algebraic, or probabilistic models), based on which a numeric score is calculated for ranking. Different from the previous CF and link analysis, the underlying assumption of IR is that the text (or information in a broader sense) contained in a document can very well indicate its (latent) topics. The relatedness of any two given documents can be calculated with a well-defined metric function atop of these topics. Since topics can have a direct connection to context, context awareness therefore becomes the most significant advantage in IR, which has been integrated into Hummingbird – Google's new search algorithm.

Despite of the different assumptions and mechanisms, aforementioned techniques are not mutually exclusive, and there is no dominant technique regarding intelligent content provision in general [7], [11]. Intelligence depends on a specific context and can be achieved by combining multiple other techniques such as behavioural analysis. [1], [7] provide a broad survey and thorough comparisons of different technologies adopted in various recommender systems.

2.2 Topic Models and Nearest Neighbour Search

Topic modelling [2], [5], [6], [19], [27] is a popular machine learning technology that utilizes statistical models to discover the abstract topics in a collection of documents. Topic models are widely used in many domains such as text mining, network analysis, genetics and etc. In this paper, we constrain our discussion in Vector Space Model (VSM) due to both space limit and the fact that Kvasir is built atop of VSM. The initial idea of using linear algebraic technique to derive latent topic model was proposed in *Latent Semantic Analysis*, i.e., LSA [5], [19]. As the core operation of LSA, SVD is a well-established subject and has been intensively studied over three decades. Recent efforts have been focusing on efficient incremental updates to accommodate dynamic data streams [8] and scalable algorithms to process huge matrices.

Probabilistic LSA (pLSA) [27] relates to non-negative matrix factorization (NMF), it explicitly models topics as latent

3. The source code of the key components in Kvasir are publicly accessible and hosted on Github. We will release all the Kvasir code after the paper is accepted.

variables based on the co-occurrences of terms and documents in a text corpus. Comparing to LSA that minimises the L2 norm (a.k.a Frobenius norm) of an objective function, pLSA relies on its likelihood function to learn the model parameters by minimizing of the cross entropy (or Kullback-Leibler divergence) between the empirical distribution and the model using Expectation Maximization (EM) algorithm. Projecting a query to a low dimensional space often requires several iterations of EM algorithm in pLSA, which in general is much slower than a simple matrix and vector multiplication in LSA. *Latent Dirichlet Allocation* (LDA) [6] is a generative model for topic modelling. LDA is similar to pLSA but replaces maximum likelihood estimator with Bayesian estimator, hence it is sometimes referred to as the Bayesian version of pLSA. Namely, LDA assumes that the topic distribution has a Dirichlet prior. All three aforementioned VSM are related to some extent but are different in actual implementations, which one can generate the best topic model still remains as an open question in academia. In Kvasir, we choose to extend the basic LSA because of its simplicity, good performance, and ease to parallelize.

Efficient nearest neighbour search in high dimensional spaces has attracted a lot of attention in machine learning community. There is a huge body of literature on this subject which can be categorized as graph-based [23], [49], hashing-based [26], [48], [52], and partition tree-based solutions [16], [31], [38], [46]. The graph-based algorithms construct a graph structure by connecting a point to its nearest neighbours in a data set. These algorithms suffer from the time-consuming construction phase. As the best known hashing-based solution, locality-sensitive hashing (LSH) [3], [35] uses a large number of hash functions with the property that the hashes of nearby points are also close to each other with high probability. The performance of a hashing-based algorithm highly depends on the quality of the hashing functions, and it is usually outperformed by partition tree-based methods in practice [38]. In particular the Randomized Partition tree (RP-tree) [16] method have been shown to be very successful in practice regarding its good scalability [18], while it was also recently shown [17] that its probability to fail is bounded when the data are documents from a topic model. RP-tree was initially introduced as an improvement over the *k-d* tree method that is more appropriate for use in high dimensional spaces, drawing inspiration from LSH [35]. In this work, inspired by a recent application of the random projection method [47], we take advantage of the simplicity of the RP-tree method to further develop its parallel version.

2.3 Popular Software Toolkits

There are abundant software toolkits with different emphasises on machine learning and natural language processing. We only list the most relevant ones like WEKA [25], scikit-learn [45], FLANN [39], Gensim [42], and ScalaNLP [44].

Both WEKA and scikit-learn include many general-purpose algorithms for data mining and analysis, but the toolkits are only suitable for small and medium problems. Gensim and ScalaNLP have a clear focus on language models. ScalaNLP's linear algebra library (Breeze) is not yet mature enough, which limits its scalability. On the other hand, Gensim scales well on large corpora using a single machine, but fails to provide efficient indexing and searching. Though FLANN provides fast nearest neighbour search, it requires loading the full data set in to memory therefore severely limits the problem size [30]. None of the aforementioned toolkits provides a horizontally scalable solution on big data frameworks. However, horizontal scalability promised by major frameworks plays a key role in achieving Internet-scale services. The default machine learning library MLlib in Apache Spark misses the stochastic SVD and effective indexing algorithms [54]. Generally speaking, an effective and efficient content-based recommender system needs to be explicitly designed, tailored, and optimized according to the specific application scenario. Despite of many off-the-shelf software toolkits, very few (if not none) practical systems were built directly atop of a general purpose machine learning package, because most toolkits are plagued with severe performance and scalability issues when they are used in a web service. Especially for large-scale LSA applications, based on our knowledge, Kvasir made the very first attempt and contribution in demonstrating the feasibility of scaling up a complex IR technique to Internet scale by exploiting the power of big data frameworks [51]. Comparing to Kvasir, other similar services we can find on the Internet such as *www.similarsites.com* and *Google Similar Pages* work only at domain level instead of page level to avoid scalability issues from high computation complexity.

3 Kvasir Architecture

At the core, Kvasir implements an LSA-based index and search service, and its architecture can be divided into two subsystems as *frontend* and *backend*. Figure 1 illustrates the general workflow and internal design of the system. The frontend is currently implemented as a lightweight extension in Chrome browser. The browser extension only sends the page URL back to the KServer whenever a new tab/window is created. The KServer running at the backend retrieves the content of the given URL then responds with the most relevant documents in a database. The results are formatted into JSON strings. The extension presents the results in a friendly way on the page being browsed. From user perspective, a user only interacts with the frontend by checking the list of recommendations that may interest him.

To connect to the frontend, the backend exposes one simple *RESTful API* as below, which gives great flexibility to all possible frontend implementations. By loosely coupling with the backend, it becomes easy to mash-up new services on top of Kvasir. Line 1 and 2 give an example request to Kvasir service. `type=0` indicates that `info` contains a URL, otherwise `info` contains a piece of text if `type=1`. Line 4-9 present an example response from the server, which contains the metainfo of a list of similar articles. Note that the frontend can refine or rearrange the results based on the metainfo (e.g., similarity or timestamp).

```
1 POST
2 https://api.kvasir/query?type=0&info=url
3
4 {"results": [
5   {"title": document title,
6    "similarity": similarity metric,
```

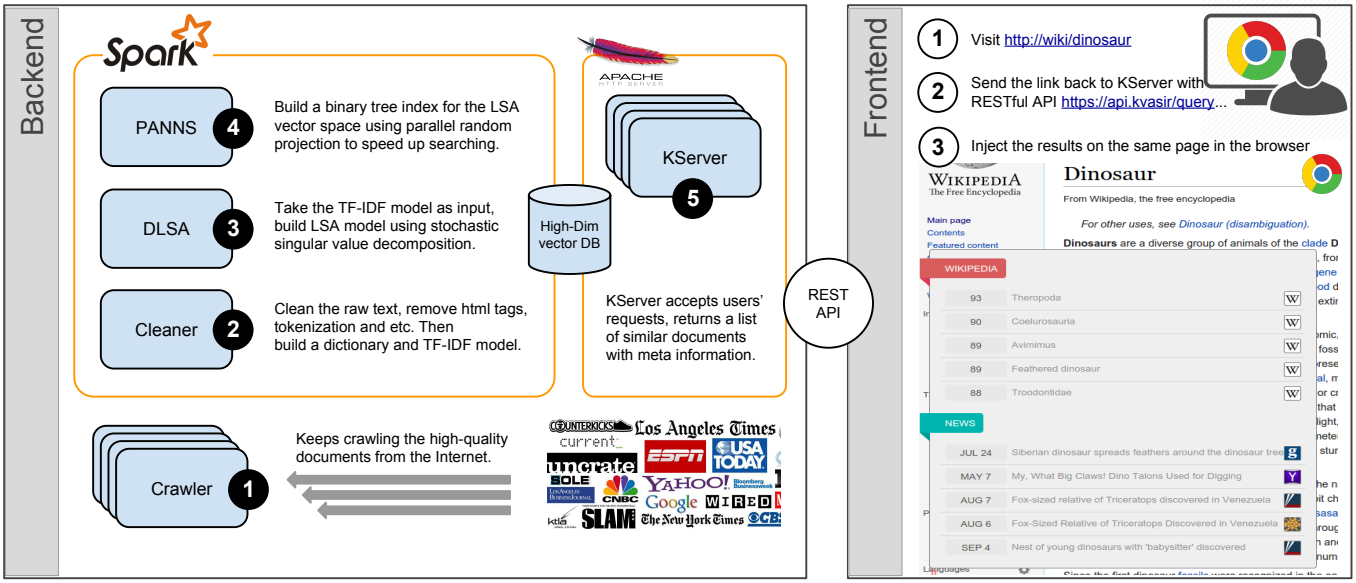

Fig. 1: Kvasir architecture – there are five major components in the backend system, and they are numbered based on their order in the workflow. Frontend is implemented in a Chrome browser, and connects the backend with a simple RESTful API. A screen shot of the user interface of the frontend is also presented at the bottom right of the figure.

```
    "page_url": link to the document,
    "timestamp": document create date}
]}
```

The backend system implements indexing and searching functionality which consist of five components: Crawler, Cleaner, DLSA, PANNS and KServer. Three components (i.e., Cleaner, DLSA and PANNS) are wrapped into one library since all are implemented on top of Apache Spark. The library covers three phases as text cleaning, database building, and indexing. We briefly present the main tasks in each component as below.

Crawler collects raw documents from the Web then compiles them into two data sets. One is the English Wikipedia dump, and another is compiled from over 300 news feeds of the high-quality content providers such as BBC, Guardian, Times, Yahoo News, MSNBC, and etc. Table 1 summarizes the basic statistics of the data sets. Multiple instances of the Crawler run in parallel on different machines. Simple fault-tolerant mechanisms like periodical backup have been implemented to improve the robustness of crawling process. In addition to the text body, the Crawler also records the timestamp, URL and title of the retrieved news as metainfo, which can be further utilized to refine the search results.

Cleaner cleans the unstructured text corpus and converts the corpus into term frequency-inverse document frequency (TF-IDF) model. In the preprocessing phase, we clean the text by removing HTML tags and stopwords, deaccenting, tokenization, etc. The dictionary refers to the vocabulary of a language model, its quality directly impacts the model performance. To build the dictionary, we exclude both extremely rare and extremely common terms, and keep 10^5 most popular ones as *features*. More precisely, a term is considered as rare if it appears in less than 20 documents, while a term is considered as common if it appears in more than 40% of documents.

DLSA builds up an LSA-based model from the previously constructed TF-IDF model. Technically, the TF-IDF itself is already a vector space language model. The reason we seldom use TF-IDF directly is because the model contains too much noise and the dimensionality is too high to process efficiently even on a modern computer. To convert a TF-IDF to an LSA model, DLSA's algebraic operations involve large matrix multiplications and time-consuming SVD. We initially tried to use MLib to implement DLSA. However, MLlib is unable to perform SVD on a data set of 10^5 features with limited RAM, we have to implement our own stochastic SVD on Apache Spark using rank-revealing technique. Section 4.1 discusses DLSA in details.

PANNS[4] builds the search index to enable fast k-NN search in high dimensional LSA vector spaces. Though dimensionality has been significantly reduced from TF-IDF (10^5 features) to LSA (10^3 features), k-NN search in a 10^3-dimension space is still a great challenge especially when we try to provide responsive services. Naive linear search using one CPU takes over 6 seconds to finish in a database of 4 million entries, which is unacceptably long for any realistic services. PANNS implements a parallel RP-tree algorithm which makes a reasonable tradeoff between accuracy and efficiency. PANNS is the core component in the backend system and Section 4.2 presents its algorithm in details.

KServer runs within a web server, processes the users requests and replies with a list of similar documents. KServer uses the index built by PANNS to perform fast search in the database. The ranking of the search results is based on the cosine similarity metric. A key performance metric for KServer is the service time. We wrapped KServer into a

4. PANNS is becoming a popular choice of Python-based approximate k-NN library for application developers. According to the PyPI's statistics, PANNS has achieved over 27,000 downloads since it was first published in October 2014. The source code is hosted on the Github at https://github.com/ryanrhymes/panns .

Data set	# of entries	Raw text size	Article length
Wikipedia	3.9×10^6	47.0 GB	Avg. 782 words
News	4.6×10^5	1.62 GB	Avg. 648 words

TABLE 1: Two data sets are used in the evaluation. Wikipedia represents relatively static knowledge, while News represents constantly changing dynamic knowledge.

Docker[5] image and deployed multiple KServer instances on different machines to achieve better performance. We also implemented a simple round-robin mechanism to balance the request loads among the multiple KServers.

Kvasir architecture provides a great potential and flexibility for developers to build various interesting applications on different devices, e.g., semantic search engine, intelligent Twitter bots, context-aware content provision, and etc.[6]

4 Core Algorithms

DLSA and PANNS are the two core components responsible for building language models and indexing the high dimensional data sets in Kvasir. In the following, we first sketch out the key ideas in DLSA. Then we focus on the mechanisms of PANNS and present in details how we extend the classic RP-tree algorithm to improve the indexing and querying efficiency. The code of the key components in Kvasir are publicly accessible on Github, those who have interest can either read the code to further study the algorithmic details [50] or refer to our technical report [29].

4.1 Distributed Stochastic SVD

The vector space model belongs to algebraic language models, where each document is represented with a row vector. Each element in the vector represents the weight of a term in the dictionary calculated in a specific way. E.g., it can be simply calculated as the frequency of a term in a document, or slightly more complicated TF-IDF. The length of the vector is determined by the size of the dictionary (i.e., number of features). A text corpus containing m documents and a dictionary of n terms will be converted to an $A = m \times n$ row-based matrix. Informally, we say that A grows taller if the number of documents (i.e., m) increases, and grows fatter if we add more terms (i.e., n) in the dictionary. LSA utilizes SVD to reduce n by only keeping a small number of linear combinations of the original features. To perform SVD, we need to calculate the covariance matrix $C = A^T \times A$, which is a $n \times n$ matrix and is usually much smaller than A.

We can easily parallelize the calculation of C by dividing A into k smaller chunks of size $[\frac{m}{k}] \times n$, so that the final result can be obtained by aggregating the partial results as $C = A^T \times A = \sum_{i=1}^{k} A_i^T \times A_i$. However, a more serious problem is posed by the large number of columns, i.e., n. The SVD function in MLlib is only able to handle tall and thin matrices up to some hundreds of features. For most of the language models, there are often hundreds of thousands features (e.g., 10^5 in our case). The covariance matrix C becomes too big to fit into the physical memory, hence the native SVD operation in MLlib of Spark fails as the first subfigure of Figure 2 shows.

In linear algebra, a matrix can be approximated by another matrix of lower rank while still retaining approximately properties of the matrix that are important for the problem at hand. In other words, we can use another thinner matrix B to approximate the original fat A. The corresponding technique is referred to as rank-revealing QR estimation [24]. A TF-IDF model having 10^5 features often contains a lot of redundant information. Therefore, we can effectively thin the matrix A then fit C into the memory. Figure 2 illustrates the algorithmic logic in DLSA, which is essentially a distributed stochastic SVD implementation.

4.2 Parallel Randomized Partition Tree

With an LSA model at hand, finding the most relevant document is equivalent to finding the nearest neighbours for a given point in the derived vector space, which is often referred to as k-NN problem. The distance is usually measured with the cosine similarity of two vectors. However, neither naive linear search nor conventional *k-d* tree is capable of performing efficient search in such high dimensional space even though the dimensionality has been significantly reduced from 10^5 to 10^3 by LSA.

Nonetheless, we need not locate the exact nearest neighbours in practice. In most cases, slight numerical error (reflected in the language context) is not noticeable at all, i.e., the returned documents still look relevant from the user's perspective. By sacrificing some accuracy, we can obtain a significant gain in searching speed.

The general idea of RP-tree algorithm used here is clustering the points by partitioning the space into smaller subspaces recursively. Technically, this can be achieved by any tree-based algorithms. Given a tree built from a database, we answer a nearest neighbour query q in an efficient way, by moving q down the tree to its appropriate leaf cell, and then return the nearest neighbour in that cell. However in several cases q's nearest neighbour may well lie within a different cell.

Figure 3 gives a naive example on a 2-dimension vector space. First, a random vector x is drawn and all the points are projected onto x. Then we divide the whole space into half at the mean value of all projections (i.e., the blue circle on x) to reduce the problem size. For each new subspace, we draw another random vector for projection, and this process continues recursively until the number of points in the space reaches the predefined threshold on cluster size. We can construct a binary tree to facilitate the search. As we can see in the first subfigure of Figure 3, though the projections of A, B, and C seem close to each other on x, C is actually quite distant from A and B. However, it has been shown that such misclassifications become arbitrarily rare as the iterative procedure continues by drawing more random vectors and performing corresponding splits. More precisely, in [16] the authors show that under the assumption of some intrinsic dimensionality of a subcluster (i.e., nodes of a tree structure), its descendant clusters will have

5. Docker is a virtualization technology which utilizes Linux container to provide system-level isolation. Docker is an open source project and its webiste is https://www.docker.com/

6. We provide the live demo videos of the seamless integration of Kvasir into web browsing at the official website. The link is http://www.cs.helsinki.fi/u/lxwang/kvasir/#demo

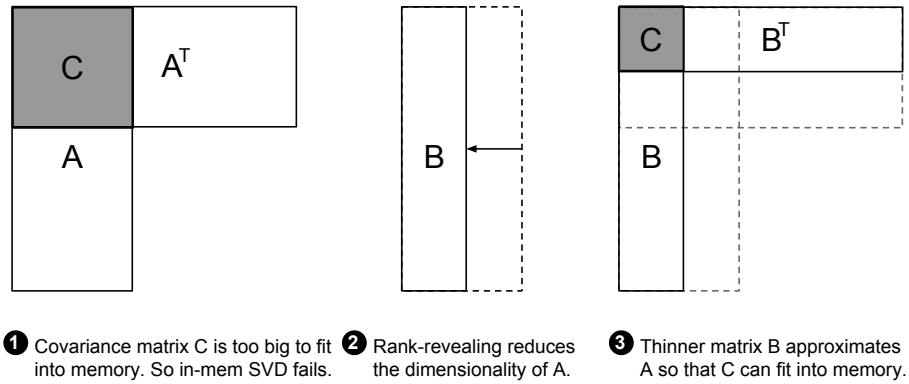

Fig. 2: DLSA uses rank-revealing to effectively reduce dimensionality to perform in-memory SVD. By converting the fat matrix A to the thinner matrix B, we can effectively reduce the size of the covariance matrix C. At the same time, the smaller matrix B remains as a good estimate of A.

a much smaller diameter, hence can include the points that are expected to be more similar to each other. Herein the diameter is defined as the distance between the furthest pair of data points in a cell. Such an example is given in Figure 3, where y successfully separates C from A and B.

Another kind of misclassification is that two nearby points are unluckily divided into different subspaces, e.g., points B and D in the left panel of Figure 3. To get around this issue, the authors in [35] proposed a tree structure (i.e., spill tree) where each data point is stored in multiple leaves, by following overlapping splits. Although the query time remains essentially the same, the required space is significantly increased. In this work we choose to improve the accuracy by building multiple RP-trees. We expect that the randomness in tree construction will introduce extra variability in the neighbours that are returned by several RP-trees for a given query point. This can be taken as an advantage in order to mitigate the second kind of misclassification while searching for the nearest neighbours of a query point in the combined search set. However, in this case one would need to store a large number of random vectors at every node of the tree, introducing significant storage overhead as well. For a corpus of 4 million documents, if we use 10^5 random vectors (i.e., a cluster size of 20), and each vector is a 10^3-dimension real vector (32-bit float number), the induced storage overhead is about 381.5 MB for each RP-tree. Therefore, such a solution leads to a huge index of 47.7 GB given 128 RP-trees are included, or 95.4 GB given 256 RP-trees.

The huge index size not only consumes a significant amount of storage resources, but also prevents the system from scaling up after more and more documents are collected. One possible solution to reduce the index size is reusing the random vectors. Namely, we can generate a pool of random vectors once, then randomly choose one from the pool each time when one is needed. However, the immediate challenge emerges when we try to parallelize the tree building on multiple nodes, because we need to broadcast the pool of vectors onto every node, which causes significant network traffic.

To address this challenge, we propose to use a pseudo random seed in building and storing search index. Instead of maintaining a pool of random vectors, we just need a random seed for each RP-tree. The computation node can build all the random vectors on the fly from the given seed. From the model building perspective, we can easily broadcast several random seeds with negligible traffic overhead instead of a large matrix in the network, therefore we improve the computation efficiency. From the storage perspective, we only need to store one 4-byte random seed for each RP-tree. In such a way, we are able to successfully reduce the storage overhead from 47.7 GB to 512 B for a search index consisting of 128 RP-trees (with cluster size 20), or from 95.4 GB to only 1 KB if 256 RP-trees are used.

4.3 Using More Trees with Small Clusters

A RP-tree helps us to locate a cluster which is likely to contain some of the k nearest neighbours for a given query point. Within the cluster, a linear search is performed to identify the best candidates. Regarding the design of PANNS, we have two design options in order to improve the searching accuracy. Namely, given the size of the aggregated cluster which is taken as the union of all the target clusters from every tree, we can

1) either use *less trees with larger leaf clusters*,
2) or use *more trees with smaller leaf clusters*.

We expect that when using more trees the probability of a query point to fall very close to a splitting hyperplane should be reduced, thus it should be less likely for its nearest neighbours to lie in a different cluster. By reducing such misclassifications, the searching accuracy is supposed to be improved. Based on our knowledge, although there are no previous theoretical results that may justify such a hypothesis in the field of nearest neighbour search algorithms, this concept could be considered as a combination strategy similar to those appeared in ensemble clustering, a very well established field of research [40]. Similar to our case, ensemble clustering algorithms improve clustering solutions by fusing information from several data partitions. In our further study on this particular part of the proposed system we intend to extend the probabilistic schemes developed in [17] in an attempt to discover the underlying theoretical properties suggested by our empirical findings.

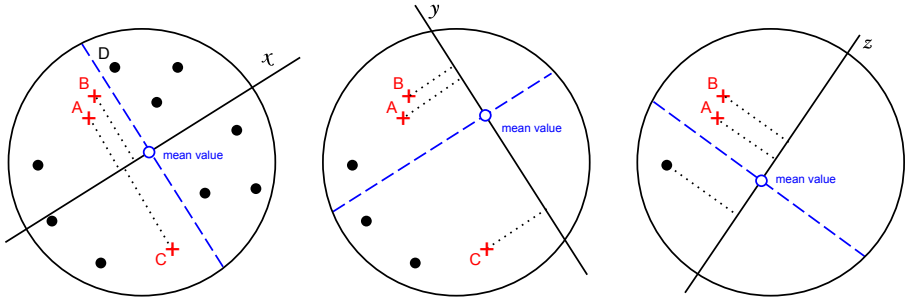

Fig. 3: We can continuously project the points on random vectors and use mean value to divide the space for clustering.

In particular, we intend to similarly provide theoretical bounds for failure probability and show that such failures can be reduced by using more RP-trees.

To experimentally investigate this hypothesis we employ a subset of the Wikipedia database for further analysis. In what follows, the data set contains $500,000$ points and we always search for the 50 nearest neighbours of a query point. Then we measure the searching accuracy by calculating the amount of actual nearest neighbours found.

We query $1,000$ points in each experiment. The results presented in Figure 4 correspond to the mean values of the aggregated nearest neighbours of the $1,000$ query points discovered by PANNS out of 100 experiment runs. Note that x-axis represents the "size of search space" which is defined by the number of unique points within the union of all the leaf clusters that the query point fall in. Therefore, given the same search space size, using more tress indicates that the leaf clusters become smaller.

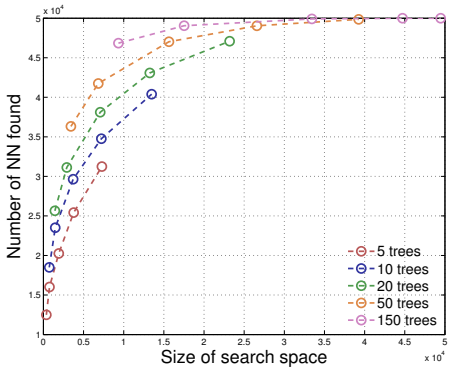

Fig. 4: The number of true nearest neighbours found for different number of trees. For a given search space size, more trees lead to the better accuracy.

As we can see in Figure 4, for a given x value, the curves move upwards as we use more and more trees, indicating that the accuracy improves. As shown in the case of 50 trees, almost 80% of the actual nearest neighbours are found by performing a search over the 10% of the data set.

To further illustrate the benefits of using as many RP-trees as possible, we present in Figure 5 the results where the size of search space remains approximately constant while the number of trees grows and subsequently the cluster size shrinks accordingly. As shown, a larger number of trees leads to the better accuracy. E.g., the accuracy is improved about 62.5% by increasing the number of trees from 2 to 18.

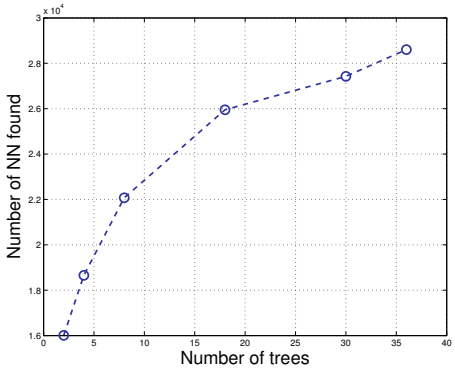

Fig. 5: The number of true nearest neighbours found as a function of the number of RP-trees used, while the search space size remains approximately constant.

Finally in Figure 6 similar outcome is observed when the average size of the leaf clusters remains approximately constant and the number of trees increases. In these experiments, we choose two specific cluster sizes for comparisons, i.e., cluster size 77 and 787. Both are just average leaf cluster sizes resulted from the termination criterion in the tree construction procedure which pre-sets a maximum allowed size of a leaf cluster (here 100 and 1000 respectively, selected for illustration purposes as any other relative set up gives similar results). In addition, we also draw a random subset for any given size from the whole data set to serve as a baseline. As we see, the accuracy of the random subset has a linear improvement rate which is simply due to the linear growth of its search space. As expected, the RP-tree solutions are significantly better than the random subset, and cluster size 77 consistently outperforms cluster size 787 especially when the search space is small.

Our empirical results clearly show the benefits of using more trees instead of using larger clusters for improving search accuracy. Moreover, regarding the searching performance, since searching can be easily parallelized, using more trees will not impact the searching time.

4.4 Getting Rid of Original Space

To select the best candidates from a cluster of points, we need to use the coordinates in the original space to calculate their relative distance to the query point. This however, first increases the storage overhead since we need to keep the original high dimensional data set which is usually huge;

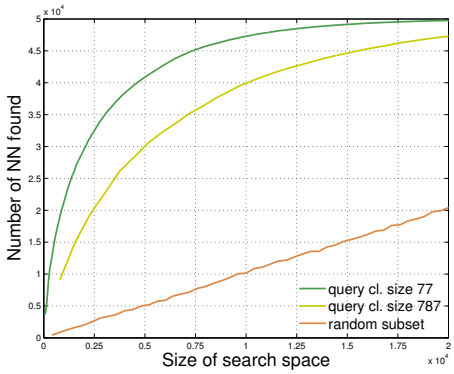

Fig. 6: The number of true nearest neighbours found as a function of the search space size, while the average leaf cluster size remains approximately constant. The random subset serves as the baseline for comparison.

second increases the query overhead since we need to access such data set. The performance becomes more severely degraded if the original data set is too big to load into the physical memory. Moreover, computing the distance between two points in the high dimensional space per se is very time-consuming.

Nonetheless, our results show that it is possible to completely get rid of the original data set while keeping the accuracy at a satisfying level. The underlying core idea is to replace the original space with the projected one. By so doing, we are able to achieve a significant reduction in storage and non-trivial gains in searching performance.

PANNS supports both *"searching with original space"* and *"searching without original space"* modes. To illustrate how PANNS searches for k-NN of a given query point $x \in \mathbb{R}^n$ without visiting the original space, we first introduce the following notations. We let T_i denote the i^{th} RP-tree in the search index file. Recall that a search index file is a collection of RP-trees. We then let C_i denote the target cluster at a leaf node of T_i which might contain the k-NN of x (or just part of them). Instead of storing the actual vector values of the original data points, each cluster C_i stores only the tuples of indices and projected values of the data points. I.e., $C_i = \{(j, y_{i,j})|i, j \in \mathbb{N}, y_{i,j} \in \mathbb{R}\}$ wherein j is the index of a data point in the data set and $y_{i,j}$ is its projected value in C_i. PANNS performs the following steps in order to find the approximate k-NN of x.

Step 1 For each RP-tree T_i in the index file, locate the target cluster C_i using the standard RP-tree algorithm. We then project the query point $x \in \mathbb{R}^n$ into the same space as those points in C_i, i.e. $x'_i \in \mathbb{R}$.

Step 2 For each C_i, make another set D_i which contains the relative projected distances. Namely, the relative distance between x'_i and each point $y_{i,j}$ in C_i in the projected space. More precisely,

$$D_i = \{(j, d_{i,j})|d_{i,j} = |y_{i,j} - x'_i|, \forall (j, y_{i,j}) \in C_i\}. \quad (1)$$

Step 3 Merge all the sets D_i into D and utilize the frequency information of each point to calculate its weighted distance. Namely,

$$D = \{(j, d_j)|d_j = \frac{\sum_{\forall D_i} d_{i,j}}{\sum_{\forall D_i} \mathbb{I}(j, D_i)}, \forall (j, \cdot) \in \bigcup_{\forall D_i} D_i\}. \quad (2)$$

$\mathbb{I}(j, D_i)$ is the identity function which returns 1 *iff* there is some $(j, \cdot) \in D_i$, otherwise returns 0.

Step 4 Sort the tuples in set D according to their weighted distances d_j in a non-decreasing order, then return the first k elements as the result of x's approximate k-NN.

As shown above, the core operations of PANNS are in the Step 2 and Step 3 which first calculate the relative projected distance between the query point and each potential candidates in the projected space, then weigh the distances based on the frequency in the target clusters. In fact, the weighting scheme in Step 3 simply calculates the average of the relative distance of every given point, hence we refer to it as *average weighting scheme* in the following discussion for convenience.

Although the average weighting scheme seem to present a viable solution, our evaluations show that the effectiveness of k-NN search is greatly affected by minor misclassification errors making the particular task much more complex. For example, given a query point x, even though the point y is actually very far away from x in the original space, it is still highly likely that y will be included in the final result as long as y's projected value appears to be very close to x'_i in just one of the RP-trees. To avoid such errors, we have to consider that the probability of a point to be an actual k-NN should increase accordingly if that point appears in several target clusters C_i.

We adopt another new weighting scheme called *cubic weighting scheme* in PANNS implementation. As such in the cubic weighting scheme, we empirically update the calculation of the weighted distance as follows:

$$D = \{(j, d_j)|d_j = \frac{\sum_{\forall D_i} d_{i,j}}{(\sum_{\forall D_i} \mathbb{I}(j, D_i))^3}, \forall (j, \cdot) \in \bigcup_{\forall D_i} D_i\}. \quad (3)$$

By so doing, we give much more weight on the points which have multiple occurrences in different C_i by assuming that such points are more likely to be the true k-NN.

To measure the effectiveness of the two weighting schemes, we perform the following experiments wherein only the projected space is used for searching for k-NN. The experiment set-up remains the same as that has been presented in Section 4.3. The results are illustrated in Figure 7 which corresponds to using the cubic weighting scheme and Figure 8 which corresponds to using the average weighting scheme. Note that both figures use the same scale on y-axis for the purpose of comparison. The dramatic effect of the different weighting schemes is exposed by comparing the results in both figures. The cubic weighting scheme has more predictable behaviours and is much more superior to the average weighting scheme regarding the searching accuracy.

Interestingly, by comparing Figure 7 to Figure 4, we notice that the accuracy does not improve as much as we

expected when the size of search space increases. Moreover, we can even observe a consistent decrease in all the cases in Figure 8 despite of the increased search space. We hypothesize that as the cluster size grows the amount of points with multiple occurrences in different clusters grows affecting the updated weighting distance. However, a more thorough investigation is reserved for our future research in order to gain a better understanding of such behaviours.

In general, the results of using cubic weighting scheme confirm that it is feasible to use only the projected space in the actual system implementation. Figure 7 shows that the accuracy is already able to reach over 30% by using only 30 trees, and 50% by using 150 trees.

Furthermore, the results above also indicate that using more trees with smaller clusters is arguably the more effective way (if it is not the only way) to improve the accuracy than using larger clusters whenever only the projected space is used. The reason is because enlarging search space does not necessarily lead to any significant improvement in accuracy in such context.

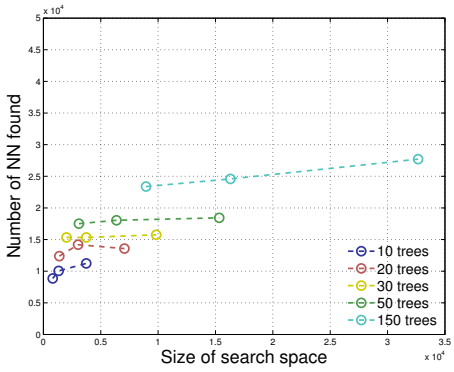

Fig. 7: The number of true nearest neighbours found in the projected space for different number of trees using the cubic weighting scheme. Using more trees improves the accuracy, but enlarging the search space by using larger clusters only brings marginal benefits. Note that only the projected space is used for searching for k-NN.

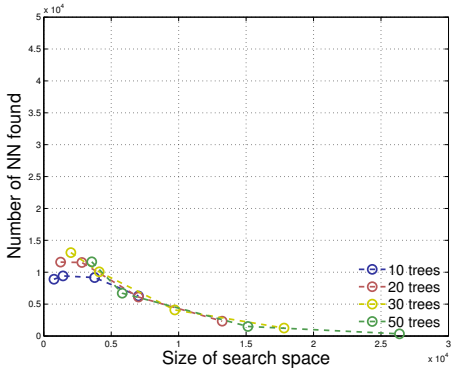

Fig. 8: The number of true nearest neighbours found in the projected space for different number of trees using the original average weighting scheme. Neither using more trees nor using larger clusters can improve the searching accuracy. Note that only the projected space is used for searching for k-NN.

4.5 Caching to Scale Up

Even though our indexing algorithm is able to significantly reduce the index size, the index will eventually become too big to fit into memory when the corpus grows from millions to trillions of documents. One engineering solution is using MMAP provided in operating systems which maps the whole file from hard-disk to memory space without actually loading it into the physical memory. The loading is only triggered by a cache miss event due to accessing a specific chunk of data which does not happen to be in the memory. Loading and eviction are handled automatically by the operating system.

With MMAP technology, we can effectively access large data sets with limited memory. However, the performance of using MMAP is subject to the data access pattern. E.g., search performance may degrade if the access pattern is truly random on a huge index. In practice, this is highly unlikely since the pattern of user requests follows a clear Zipf-like distribution [10], [13] with strong temporal and spatial locality [37]. In other words,

1) most users are interested in a relatively small amount of popular articles;
2) most of the articles that an individual user is reading at any given time are similar.

In vector-based language models, similar articles are clustered together in a small part of the whole space. These two observations imply that only a small part of the index trees is frequently accessed at any given time, which leads to the actual performance being much better than that of a uniformly distributed access pattern.

5 Preliminary Evaluation

Because scalability is the main challenge confronting Kvasir, the evaluation revolves around two questions as below.

1) How fast can we build a database from scratch using the library we developed for Apache Spark?
2) How fast can the search function in Kvasir serve users' requests?

In the following, we present the results of our preliminary evaluation.

The evaluation is performed on a small testbed of 10 Dell PowerEdge M610 nodes. Each node has 2 quad-core CPUs, 32GB memory, and is connected to a 10-Gbit network. All the nodes run Ubuntu SMP with a 3.2.0 Linux kernel. ATLAS (Automatically Tuned Linear Algebra System) is installed on all the nodes to support fast linear algebra operations. Three nodes are used for running Crawlers, five for running our Spark library, and the rest two for running KServer to serve users' requests as web servers. In this paper, we only report the results on using Wikipedia data set. News data set leads to consistently better performance due to its smaller size.

5.1 Database Building Time

We evaluate the efficiency of the backend system using our Apache Spark library, which includes text cleaning, model building, and indexing the three phases. We first perform a

# of CPUs	Cleaner	DLSA	PANNS	Total
1	1.32	20.23	13.99	35.54
5	0.29	6.14	2.86	9.29
10	0.19	4.22	1.44	5.85
15	0.17	3.14	0.98	4.29
20	0.16	2.61	0.77	3.54

TABLE 2: The time needed (in hours) for building an LSA-based database from Wikipedia raw text corpus. The time is decomposed to component-wise level. Search index uses 128 RP-trees with cluster size of 20 points.

sequential execution on a single CPU to obtain a baseline benchmark. With only one CPU, it takes over 35 hours to process the Wikipedia data set. Using 5 CPUs to parallelize the computation, it takes about 9 hours which is almost 4 times improvement. From Table 2, we can see that the total building speed is improved sublinearly. The reason is because the overhead from I/O and network operations eventually replace CPU overhead and become the main bottleneck in the system.

By examining the time usage and checking the component-wise overhead, DLSA contributes most of the computation time while Cleaner contributes the least. Cleaner's tasks are easy to parallelize due to its straightforward structure, but there are only marginal improvements after 10 CPUs since most of the time is spent in I/O operations and job scheduling. For DLSA, the parallelizm is achieved by dividing a tall matrix into smaller chunks then distributing the computation on multiple nodes. The partial calculations need to be exchanged among the nodes to obtain the final result, and therefore the penalty of the increased network traffic will eventually overrun the benefit of parallelizm. Further investigation reveals that the percent of time used in transmitting data increases from 10.5% to 37.2% (from 5 CPUs to 20 CPUs). On the other hand, indexing phase scales very well by using more computation nodes because PANNS does not require exchanging too much data among the nodes.

5.2 Accuracy and Scalability of Searching

Service time represents the amount of time needed to process a request, which can also be used to calculate server throughput. Throughput is arguably the most important metric to measure the scalability of a service. We tested the service time of KServer by using one of the two web servers in the aforementioned testbed.

During normal operations, a KServer shall see a stream of requests consisting of both URLs and pieces of text from users, and the KServer shall be responsible for translating them into corresponding vectors using the LSA model. However, in our evaluation of the search scalability, content fetching via a URL is completely unnecessary and brings no insights at all in result analysis because the dominant factor becomes network conditions rather than the quality of a search algorithm. Therefore we let the client submit requests directly in their vector format. The KServer only performs the search operations using PANNS whenever a request vector arrives. The request set is generated by randomly selecting 5×10^5 entries from Wikipedia dataset.

We model the content popularity with a Zipf distribution, whose probability mass function is $f(x) = \frac{1}{x^\alpha \sum_{i=1}^{n} i^{-\alpha}}$, where x is the item index, n is the total number of items in the database, and α controls the skewness of the distribution. Smaller values of α lead to more uniform distributions while larger α values assign more mass to elements with small i. It has been empirically demonstrated that in real-world data following a power-law distribution, the α values typically range between 0.9 and 1.0 [10], [12].

We plug in different α to generate the request stream. The next request is sent out as soon as the results of the previous one is successfully received. Round trip time (RTT) depends on network conditions and is irrelevant to the efficiency of the backend, hence is excluded from total service time. In the evaluations, we still access the original high dimensional space in order to identify the true k-NN in the last step of linear search. But it is worth noting that Kvasir is able to work well by using only the projected space, the improved scalability comes at the price of degraded accuracy. Table 3 summarizes our results.

We also experiment with various index configurations to understand how index impacts the server performance. The index is configured with two parameters: the maximum cluster size c and the number of search trees t. Note that c determines how many random vectors we will draw for each search tree, which further impacts the search precision. The first row in Table 3 lists all the configurations. In general, for a realistic $\alpha = 0.9$ and index $(20, 256)$, the throughput can reach 1052 requests per second (i.e., $\frac{1000}{7.6} \times 8$) on a node of 8 CPUs.

From Table 3, we can see that including more RP-trees improves the search accuracy but also increases the index size. Since we only store the random seed for all random vectors which is practically negligible, the growth of index size is mainly due to storing the tree structures. The time overhead of searching also grows sublinearly with more trees. However, since searching in different trees are independent and can be easily parallelized, the performance can be further improved by using more CPUs. Given a specific index configuration, the service time increases as α decreases, which attests our arguments in Section 4.5. Namely, we can exploit the highly skewed popularity distribution and utilize caching techniques to scale up the system.

As we mentioned, given a fixed number of trees, increasing the cluster size is equivalent to reducing the number of random projections, and vice versa. We increase the maximum cluster size from 20 to 80 and present the results in the right half of Table 3. Though the intuition is that the precision should deteriorate with less random projections, we notice that the precision is improved instead of degrading. The reason is two-fold: first, large cluster size reduces the probability of misclassification for those projections close to the split point. Second, since we perform linear search within a cluster, larger cluster enables us to explore more neighbours which leads to higher probability to find the actual nearest ones. Nonetheless, also due to the linear search in larger clusters, the gain in the accuracy comes at the price of inflated searching time.

Nonetheless, given a fixed aggregated cluster size, we can improve both searching performance and accuracy at the same time by using a larger number of trees with smaller cluster size. E.g., both index configurations $(20, 64)$ and $(80, 16)$ lead to the similar aggregated cluster size after

(c,t)		(20,16)	(20,32)	(20,64)	(20,128)	(20,256)	(80,16)	(80,32)	(80,64)	(80,128)	(80,256)
Index (MB)		361	721	1445	2891	5782	258	519	1039	2078	4155
Precision (%)		68.5	75.2	84.7	89.4	94.9	71.3	83.6	91.2	95.6	99.2
$\alpha_1 = 1.0$	ms	2.2	3.7	4.5	5.9	6.8	4.6	7.9	11.2	13.7	16.1
$\alpha_2 = 0.9$	ms	3.4	4.3	6.0	6.8	7.6	7.2	9.5	14.9	15.3	17.1
$\alpha_3 = 0.8$	ms	4.3	4.9	6.7	7.9	8.4	9.1	11.7	15.2	17.4	17.9
$\alpha_4 = 0.7$	ms	5.5	6.3	7.4	8.5	9.3	11.6	13.4	16.1	17.7	18.5
$\alpha_5 = 0.6$	ms	6.1	6.7	7.9	8.8	9.8	13.9	16.0	18.5	19.8	21.1
$\alpha_6 = 0.5$	ms	6.7	7.3	8.2	9.0	10.3	16.6	17.8	19.9	20.4	23.1

TABLE 3: Scalability test on KServer with different index configurations and request patterns. (c,t) in the first row, c represents the maximum cluster size, and t represents the number of RP-trees. Zipf-(α, n) is used to model the content popularity. The results confirm our analysis in Section 4.3 that using a larger number of trees of smaller clusters improves both the searching performance and accuracy. E.g., compare $(20, 64)$ to $(80, 16)$, or compare $(20, 128)$ to $(80, 32)$.

taking the union of leaf clusters which is $20 \times 64 = 80 \times 16 = 1280$. However, the configuration $(20, 64)$ is more attractive due to the following reasons. First, the accuracy of $(20, 64)$ is higher than that of $(80, 16)$, i.e., 84.7% vs. 71.3%. Second, as we mentioned, searching in different trees can be performed in parallel, therefore using more trees will not degrade the searching performance. Moreover, our experiments (in Section 4.3) also show that using more trees in general renders smaller aggregated cluster due to the increased overlapping between the individual clusters, which further explains why configuration $(20, 64)$ is always faster than configuration $(80, 16)$. Similar results can also be observed by comparing the following configuration pairs: $(20, 128)$ to $(80, 32)$, or $(20, 256)$ to $(80, 64)$.

5.3 Scalability on Increasing Data Sets

To test the scalability of Kvasir, we use a much larger data set from Yahoo company called *L1 - Yahoo! N-Grams (version 2.0)* data set[7], we refer to it as L1 data set for short in the following discussion. L1 data set contains 14.6 million documents (126 million unique sentences, 3.4 billion running words). The data set was crawled from over $12,000$ news-oriented sites over a period of 10 months between February 2006 and December 2006. L1 data set does not contain any raw HTML documents but only has n-gram files ($n = 1, 2, 3, 4, 5$). This does not pose a problem in evaluations since the text cleaning phase only has marginal contribution to the total building time (please refer to Table 2). Moreover, we only use 1-gram file in our evaluation.

In order to gain an understanding on the performance of Kvasir on larger data sets, we randomly select certain number of documents from L1 data set to make a new text corpus, and gradually increase the corpus size from 4 million to 14 million. Then we study the total building time, query time, query accuracy respectively as a function of increasing data size, as Figure 9 shows.

In Figure 9a, we can see that the total amount of building time grows accordingly as the corpus size increases, and the growth is basically linear with negligible variations. Error bars in the figure show the variations in the building time which are often caused by the stragglers [53] among worker processes. In general, such variations depend on both workload and traffic load in a cluster and are usually small in most cases. Doubling the number of CPUs used in index building (e.g., from 20 to 40) can significantly reduce

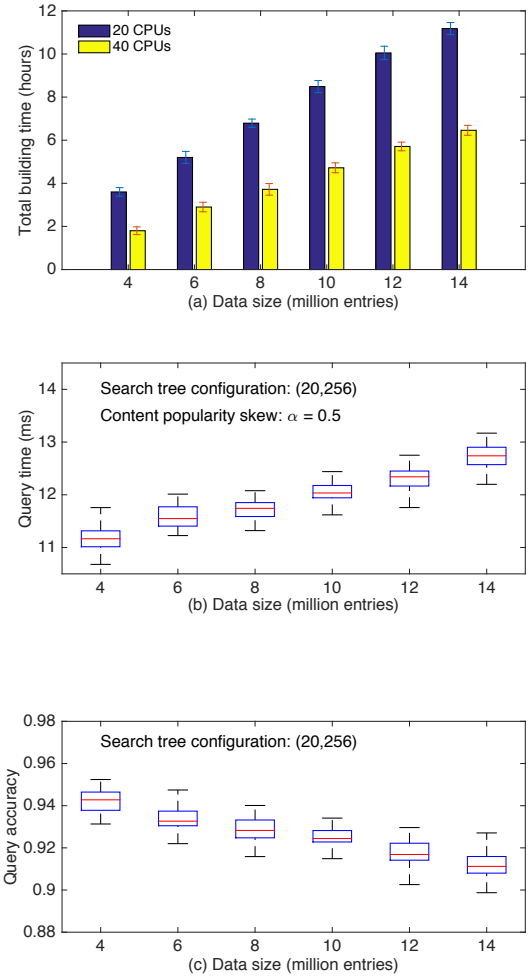

Fig. 9: Building time, query time, and accuracy as a function of increased data size. As we can see, building time increases linearly as data set grows, but using more CPUs decreases the building time effectively. Increasing data set appears to have marginal impacts on both query time and accuracy, which indicates Kvasir's scalability on large text corpora.

the total building time about 45%. Together with Table 2, the results show that Kvasir has a very good horizontal scalability over a resource pool.

In Figure 9b, we present the Whisker plot of query time as a function of increasing corpus size. The average query time is 11.2 ms for the smallest corpus size of 4 million documents, and this number increases to 12.7 ms for 14 million

7. http://webscope.sandbox.yahoo.com/#datasets

documents. Even though the corpus size increases almost three times, the actual query time only slightly increases about 13.3% (i.e., 1.5 ms in its absolute value). To explain such small degradation, first recall a search operation consists of "search in RP-trees" and "search in aggregated leaf clusters" two steps. The reason of the slow growth in query time can be explained in two fold: First, the depth of a binary search tree for 4 million documents is 17.6 levels while for 14 million documents it is 19.4 levels on average (given the leaf cluster size is 20). Two more search levels only introduce negligible overhead in practical searching time. Second, when the original high dimensional space is accessed, the linear search in the aggregated leaf clusters constitutes the most time-consuming step in PANNS. However, since the tree configuration remains the same $(20, 256)$ throughout the evaluations, namely a cluster size of 20 and 256 search trees, the aggregated set will also remain roughly the same size. Without accessing the original space, the linear search in the last step can be skipped. In either case, the impact on the searching time is marginal, which further shows the capability of Kvair in handling large text corpora.

Figure 9c plots the impact of corpus size on the accuracy of searching. Given a fixed search tree configuration (i.e., $(20, 256)$ used as before), the search accuracy slightly degrades from 94.2% to 91.2% even though the documents increase from 4 million to 14 million. In theory, increasing the number of data points (or documents) in a unit high dimensional ball leads to an increased density, which can further increase the probability of misclassification as explained in Section 4.2. However, since using both multiple projections and multiple search trees can effectively ameliorate the negative effects of misclassification, the actual impact of a large corpus becomes small, only 3% drop in search accuracy. The degradation in search accuracy is much slower than the increase in corpus size, which again indicates Kvasir's good scalability on large corpora. Moreover, using more RP-trees can certainly improve the accuracy. Our further investigation shows that using the RP-tree configuration $(20, 400)$ is able to bring the accuracy back to 95%.

Note that in this evaluation we are heavily constrained by our available computational resources in the lab (refer to Section 5). Larger data sets can certainly be handled when more computation and storage resources are pooled in a cluster. The preliminary results above already show that Kvasir possesses good vertical and horizontal scalability.

5.4 Empirical Work on User Satisfaction

A full study of user experience is already out of the scope of this paper due to our strong focus on architecture, data structure and algorithm, therefore we only briefly share some empirical results. In information retrieval, user satisfaction is measured by *Relevance* metrics [14]. Human assessors are required to manually label the relevant documents in the search results for each given information need. With the labelled data, multiple metrics are available to measure the relevance, from the well-known precision and recall to mean average precision, precision@k, R precision, and etc. [36] Note that the classic metric recall is no longer important for modern web scale IR systems because most users only consume the results on first several pages. Therefore, precision@k, which measures the percent of relevant documents among the top k search results, becomes an important measure especially for ranked results.

Query article from theguardian.com: *The 2010 'flash crash': how it unfolded*

Rank	Page Title	Source
#1	2010 Flash Crash	Wikipedia
#2	High-frequency Trading	Wikipedia
#3	Flash Crash, Could it happen again?	CNN
#4	Stock market crash	Wikipedia
#5	Algorithmic trading	Wikipedia
#6	US options craft rules to fend off turmoil	Thomson Reuters
#7	Are machines running exchanges	CNN
#8	A dark magic: The rise of the robot traders	BBC
#9	Market Structure: Perception Versus Reality	Markets Media
#10	Volatility Time Horizon Contracts	Markets Media

Query article from en.wikipedia.org: *Merge sort*

Rank	Page Title	Source
#1	Spreadsort	Wikipedia
#2	Insertion sort	Wikipedia
#3	Selection sort	Wikipedia
#4	Funnelsort	Wikipedia
#5	Quicksort	Wikipedia
#6	Divide and conquer algorithms	Wikipedia
#7	Bubble sort	Wikipedia
#8	Sorting algorithms	Wikipedia
#9	Best, worst and average case	Wikipedia
#10	Linked list	Wikipedia

TABLE 4: An illustration of Kvasir search results. The query is a link to an Internet article with its page title and domain specified on the first row of each table. Only page titles of the top 10 returned results are presented due to space limit.

precision@k	user #1	user #2	user #3	user #4	user #5
Average	0.9360	0.9260	0.9380	0.9560	0.9360
StdError	0.1005	0.0986	0.0805	0.0644	0.0663

TABLE 5: For each user (one column), the average and standard error of precision@k of 50 queries are presented. In total, 250 queries are performed and 2,500 articles are manually assessed and labelled by five test users.

We performed a preliminary user experiment in the Computer Lab at Cambridge University. Five users were invited to participate, and each was asked to perform 50 queries via Kvasir. The user can decide what to query but the query must be a link to an Internet article containing meaningful content. Kvasir returns top 10 results, and the user is required to manually check the content of every result then mark it either {*0: not relevant*} or {*1: relevant*}. Table 4 provides two example queries. The first one is a news article from *TheGuardian* on a famous stock market crisis in 2010. The first result links to a Wikipedia article which has a full description of the crisis, the rest come from various sources. The second example uses an article on "Merge sort" as query, and all the results are from Wikipedia itself. In both cases, all the results have strong relevance to the content of the query articles. Table 5 further presents the values of precision@k in each user test. Kvasir achieves promising results in all five user tests. On average, Kvasir's precision@k reached as high as 93.8%. We further investigated on the results marked as "non-relevant" by users. It turned out that 107 out of 154 negatives were due to the fact that a page was removed by the website so that users could not access the content to evaluate. By excluding such failure cases, the precision@k can increase to 98.0%.

6 Discussions and Future Work

Kvasir aims to provide a scalable platform of intelligent content provision. It is by no means constrained by browser frontends. In fact, many interesting applications can be developed as frontends (e.g., mobile phone apps, enterprise search engines, twitter bots, and etc.) thanks to Kvasir's flexible RESTful API and powerful backend. The recommended content come from the data sets maintained by the backend.

We mainly focused on addressing the implementation issues in the present paper, along with some empirical evaluations and discussions on the performance, scalability, and user satisfaction of Kvasir. It is worth noting that we can improve the current design in many ways. E.g., new content keeps flowing into Kvasir. However, we need not necessarily to rebuild the LSA model and index from scratch whenever new documents arrive. LSA space can be adjusted on the fly for incremental updates [8]. Then, the trees are updated by adding the new points to the corresponding leaf cell. In fact, we are actively optimising Kvasir to integrate aforementioned techniques to support dynamic data flows.

The results from KServer are ranked based on cosine similarity at the moment. However, finer-grained and more personalized re-ranking can be implemented by taking users' both long-term and short-term preferences into account. Such functionality can be achieved by extending one-class SVM or utilizing other techniques like reinforcement learning [21]. Kvasir provides a scalable Internet service via a RESTful API. Content providers can integrate Kvasir service on their website to enhance users' experience by automatically providing similar articles on the same page. Besides, more optimizations can be done at frontend side via caching and compression to reduce the traffic overhead. Furthermore, a thorough measurement on user satisfaction should be performed with the involvement of HCI experts after a larger deployment of Kvasir.

Currently, Kvasir does not provide full-fledged security and privacy support. For security, malicious users may launch DDoS attacks by submitting a huge amount of random requests quickly. Though limiting the request rate can mitigate such attacks to some extent, DDoS attacks are difficult to defend against in general. For privacy, Kvasir needs tracking a user's browsing history to provide personalized results. However, a user may not want to store such private information on the server. Finer-grained policy is needed to provide flexible privacy configurations. Security and privacy definitely deserve more thorough investigations in our future work.

7 Conclusion

In this paper, we presented Kvasir which provides seamless integration of LSA-based content provision into web browsing. To build Kvasir as a scalable Internet service, we addressed various technical challenges in the system implementation. Specifically, we proposed a parallel RP-tree algorithm and implemented stochastic SVD on Spark to tackle the scalability challenges in index building and searching. The proposed solutions were evaluated on the testbed and scaled well on multiple CPUs. The results showed that Kvasir can easily achieve millisecond query speed for a 14 million document repository thanks to its novel design. Kvasir is an open-source project and is currently under active development. The key components of Kvasir are implemented as an Apache Spark library, and all the source code are publicly accessible on Github.

Acknowledgments

This research was co-supported by the *Academy of Finland* under "The Finnish Centre of Excellence in Computational Inference Research (COIN)". We thank Nik Sultana, Richard Mortier, and Jon Crowcroft for their valuable comments.

References

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, June 2005.

[2] E. M. Airoldi, D. M. Blei, S. E. Fienberg, and E. P. Xing. Mixed membership stochastic blockmodels. *J. Mach. Learn. Res.*, 2008.

[3] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. 47th Annual IEEE Symposium on*, 2006.

[4] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[5] M. Berry, S. Dumais, and G. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[7] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutirrez. Recommender systems survey. *Knowledge-Based Systems*, 2013.

[8] M. Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear Algebra and its Applications*, 2006.

[9] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, 1998.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99, IEEE*, volume 1, pages 126–134 vol.1, Mar 1999.

[11] R. Burke. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370, 2002.

[12] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *ACM IMC'07*, 2007.

[13] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. Analyzing the video popularity characteristics of large-scale user generated content systems. *IEEE/ACM Trans. Netw.*, 2009.

[14] C. L. Clarke, M. Kolla, G. V. Cormack, O. Vechtomova, A. Ashkan, S. Büttcher, and I. MacKinnon. Novelty and diversity in information retrieval evaluation. In *ACM SIGIR'08*, 2008.

[15] P. Cremonesi, A. Tripodi, and R. Turrin. Cross-domain recommender systems. In *Data Mining Workshops (ICDMW), IEEE*, 2011.

[16] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *ACM Theory of Computing*, 2008.

[17] S. Dasgupta and K. Sinha. Randomized partition trees for exact nearest neighbor search. *CoRR*, abs/1302.1948, 2013.

[18] C. M. De Vries, L. De Vine, S. Geva, and R. Nayak. Parallel streaming signature em-tree: A clustering algorithm for web scale applications. In *International Conference on World Wide Web*, 2015.

[19] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

[20] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Inc., USA, 1992.

[21] D. Glowacka, T. Ruotsalo, K. Konyushkova, K. Athukorala, S. Kaski, and G. Jacucci. Scinet: A system for browsing scientific literature through keyword manipulation. In *ACM International Conference on Intelligent User Interfaces Companion*, 2013.

[22] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, Dec. 1992.

[23] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *International Joint Conference on Artificial Intelligence*, IJCAI'11, pages 1312–1317. AAAI Press, 2011.

[24] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 2011.
[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *ACM SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.
[26] J. He, W. Liu, and S.-F. Chang. Scalable similarity search with optimized kernel hashing. In *ACM SIGKDD*, 2010.
[27] T. Hofmann. Probabilistic latent semantic indexing. In *ACM SIGIR*, pages 50–57, 1999.
[28] D. Huynh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the semantic web inside your web browser. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *The Semantic Web*, 2005.
[29] V. Hyvönen, T. Pitkänen, S. Tasoulis, L. Wang, T. Roos, and J. Corander. Technical report: Fast k-nn search. *arXiv preprint arXiv:1509.06957*, 2015.
[30] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, Jan 2011.
[31] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, pages 3392–3399, 2010.
[32] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, Sept. 1999.
[33] Y. Koren and R. Bell. Advances in collaborative filtering. In F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, editors, *Recommender Systems Handbook*, pages 145–186. Springer US, 2011.
[34] B. Li, Q. Yang, and X. Xue. Can movies and books collaborate?: Cross-domain collaborative filtering for sparsity reduction. In *International Joint Conference on Artificial Intelligence*, 2009.
[35] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems*, MIT Press, 2004.
[36] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, Cambridge University Press, 2008.
[37] L. Wang, S. Bayhan, J. Ott, J. Kangasharju, A. Sathiaseelan, and J. Crowcroft. Pro-Diluvian: Understanding Scoped-Flooding for Content Discovery in Information-Centric Networking. In *Information-Centric Networking (ICN'15)*, ACM, 9-18, 2015.
[38] M. Muja and D. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(11):2227–2240, Nov 2014.
[39] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application*, INSTICC Press, 2009.
[40] O. Okun, editor. *Supervised and Unsupervised Ensemble Methods and their Applications*. Springer-Verlag Berlin, Germany, 2008.
[41] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford Technical Report*, 1999.
[42] R. Rehurek and P. Sojka. Software framework for topic modelling with large corpora. In *LREC 2010 workshop New Challenges for NLP Frameworks*, 2010.
[43] P. Resnick and H. R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, Mar. 1997.
[44] Scalanlp, http://www.scalanlp.org/.
[45] scikit-learn toolkit, http://scikit-learn.org/.
[46] R. Sproull. Refinements to nearest-neighbor searching ink-dimensional trees. *Algorithmica*, 6(1-6):579–589, 1991.
[47] S. Tasoulis, L. Cheng, N. Valimaki, N. J. Croucher, S. R. Harris, W. P. Hanage, T. Roos, and J. Corander. Random projection based clustering for population genomics. In *IEEE International Conference on Big Data*, pages 675–682, 2014.
[48] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for scalable image retrieval. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3424–3431, June 2010.
[49] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, pages 1106–1113, 2012.
[50] L. Wang. Kvasir project, http://cs.helsinki.fi/u/lxwang/kvasir.
[51] L. Wang, S. Tasoulis, T. Roos, and J. Kangasharju. Kvasir: Seamless integration of latent semantic analysis-based content provision into web browsing. In *International Conference on World Wide Web Companion*, pages 251–254, 2015.
[52] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu. Complementary hashing for approximate nearest neighbor search. In *IEEE Computer Vision*, pages 1631–1638, 2011.
[53] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.
[54] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2010.

Liang Wang is a research associate in the Computer Laboratory at University of Cambridge, United Kingdom. In 2003, he received his BEng in Computer Science and Mathematics from Tongji University, Shanghai, China. Later, he received both his MSc and PhD degrees in Computer Science from University of Helsinki, Finland in 2011 and 2015 respectively. Liang's research interests include system and network optimisation, information-centric networks, machine learning, and big data frameworks.

Sotiris Tasoulis received his diploma in Mathematics and his MSc in Mathematics of Computers and Decision Making in 2009 from University of Patras, Greece. In 2013 he received his PhD from University of Thessaly, Greece. In the same year he joined Helsinki Institute for Information Technology (HIIT) and Department of Computer Science of the University of Helsinki as a post doctoral researcher in Knowledge discovery in Big Data. Since September 2015 Sotiris is a lecturer in the Department of Applied Mathematics at Liverpool John Moores University, UK. His research interests are machine learning in big data applications, large scale data mining, dimensionality reduction and unsupervised learning.

Teemu Roos is an assistant professor at the Helsinki Institute for Information Technology (HIIT) and the Department of Computer Science, the University of Helsinki, Finland. He received his MSc and PhD degrees in computer science from the same university in 2001 and 2007 respectively. Teemu's research interests include the theory and applications of probabilistic graphical models, information theory, and machine learning.

Jussi Kangasharju received his MSc from Helsinki University of Technology in 1998. He received his Diplome d'Etudes Approfondies (DEA) from the Ecole Superieure des Sciences Informatiques (ESSI) in Sophia Antipolis in 1998. In 2002 he received his PhD from University of Nice Sophia Antipolis/Institut Eurecom. In 2002 he joined Darmstadt University of Technology (TUD), first as post-doctoral researcher, and from 2004 onwards as assistant professor. Since June 2007 Jussi is a professor at the department of computer science at University of Helsinki. Between 2009 and 2012 he was the director of the Future Internet research program at Helsinki Institute for Information Technology (HIIT). Jussi's research interests are information-centric networks, content distribution, opportunistic networks, and green ICT. He is a member of IEEE and ACM.