L101: Machine Learning for Language Processing

Lecture 4

Guy Emerson

Today’s Lecture

- Decision boundaries
- Non-probabilistic classifiers
  - Support Vector Machine
  - Kernels
- Linguistic kernels

Decision Boundaries

$$\begin{aligned} & \operatorname*{argmax}_y P(y|x) \\ = & \operatorname*{argmax}_y \exp\left(\theta_y + \sum_i \theta_{y,i} x_i\right) \end{aligned}$$

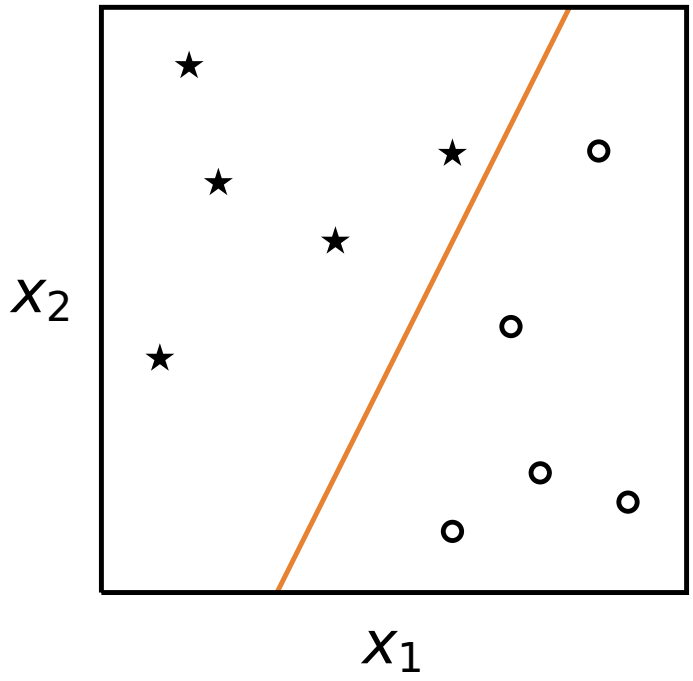
For two classes:

$$b + \sum_i a_i x_i > 0 \ ?$$

For any probabilistic model, we can consider the best decision under the model, $\operatorname{argmax}_y P(y|x)$.

The rest of the slide is specifically for logistic regression (and Naive Bayes). For any two classes, we can simplify the decision to checking whether $b + \sum_i a_i x_i$ is more or less than 0, for some values of a_i and b.

Decision Boundaries



If we consider each input x as a point in space, a classifier makes a decision at each point. We can then consider regions of space with the same decision, and the boundaries between regions.

For logistic regression and Naive Bayes, the decision boundary is a straight line.

Ideally, a classifier's decision boundaries should match the classes – e.g. the orange decision boundary correctly separates the stars and circles.

Support Vector Machines

- Non-probabilistic
- $\text{argmax}_y \left(\theta_y + \sum_i \theta_{y,i} x_i\right)$
- Linear: hyperplane boundary

A support vector machine is a non-probabilistic classifier – it directly learns a decision boundary.

The boundary is linear, just like for logistic regression and Naive Bayes. For an n-dimensional space, the boundary is $(n-1)$-dimensional, called a hyperplane. (Compared to logistic regression, we can drop the exponential, because we don't need to define a probability distribution.)

Since we only care about the argmax of the above expression, and not its magnitude, multiplying all parameters by a constant doesn't change the boundary – unlike for logistic regression, where this changes the probabilities. This means a support vector machine has fewer independent parameters than logistic regression does – for two classes, over an n-dimensional space, a support vector machine has n independent parameters, while logistic regression has $n+1$. (And Naive Bayes has $2n+1$.)

Perceptron Algorithm

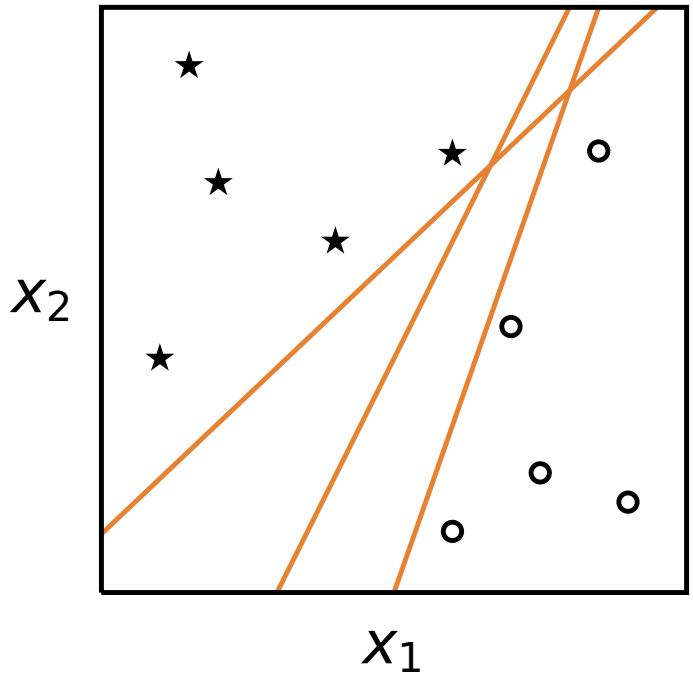
- Iterate through training data
- If correct, do nothing
- If incorrect, update:
  - Correct class y: $\theta_{y,i} \leftarrow \theta_{y,i} + x_i$
  - Incorrect class y': $\theta_{y',i} \leftarrow \theta_{y',i} - x_i$

The perceptron algorithm is a simple training algorithm.

Conceptually similar to gradient descent – we incrementally update the parameters, to improve the predictions of the model.

(Not to be confused with the term “multilayer perceptron”, which refers to a multilayer feedforward network, but which cannot sensibly be trained using the perceptron algorithm. I will avoid the term “multilayer perceptron” to avoid confusion.)

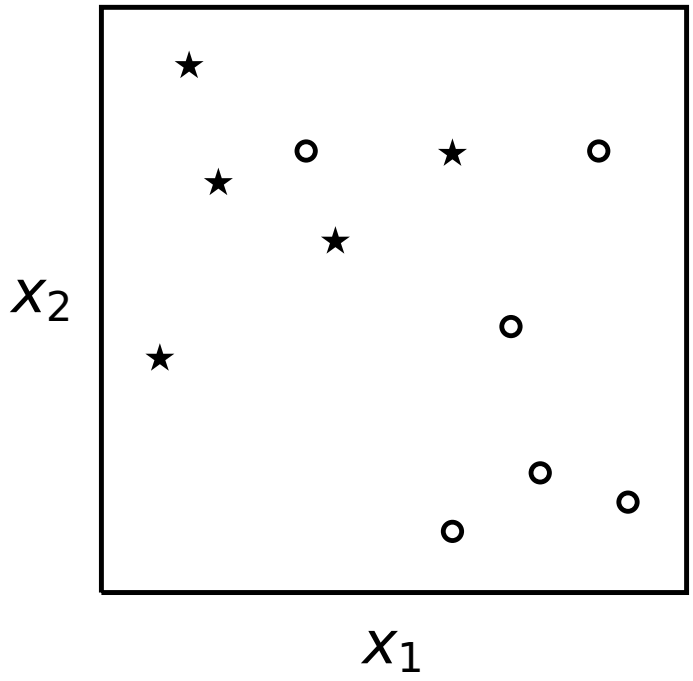
Linear Separability



A dataset is linearly separable if a linear decision boundary can perfectly separate the classes. There may be more than one such boundary.

For a linearly separable dataset, the perceptron algorithm is guaranteed to converge on a separating hyperplane.

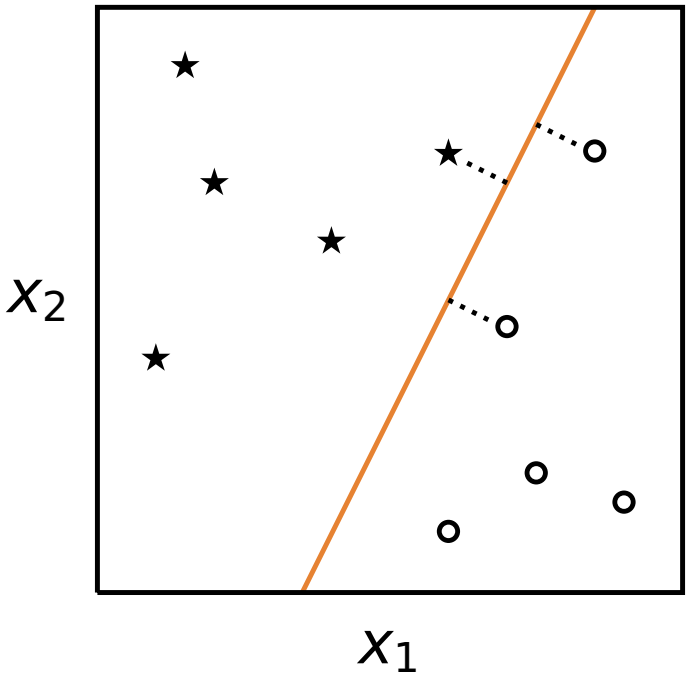
Linear Separability



This dataset is not linearly separable.

When training on such a dataset, the perceptron algorithm will never converge (for any linear boundary, there is some incorrectly classified point, and so there will be an update to the parameters). However, for a dataset that is close to separable, it will only make a small number of mistakes.

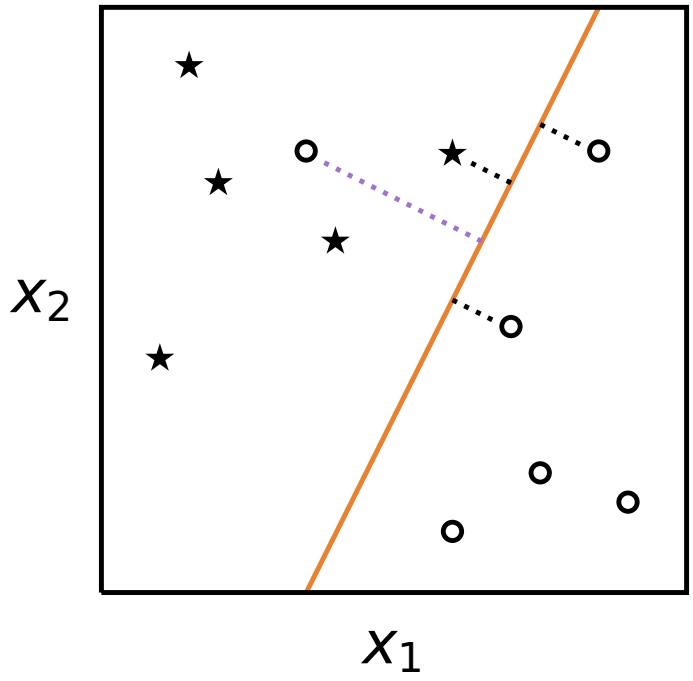
Maximum Margin



For a linearly separable dataset, a support vector machine chooses the hyperplane with maximum distance from any point.

The points closest to the boundary are called the “support vectors”, hence the name “support vector machine”.

Maximum Margin

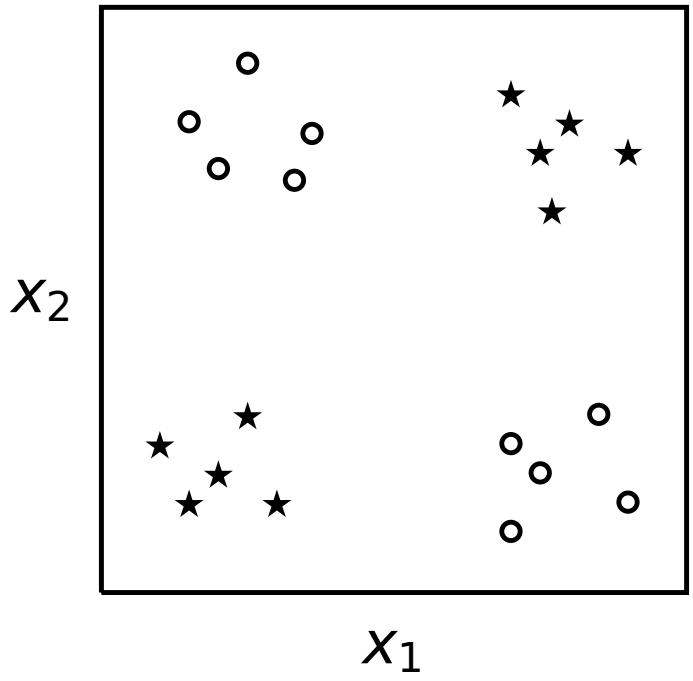


To deal with datasets that are not linearly separable, we can introduce a penalty for incorrectly classified points, where the penalty increases with distance from the boundary.

Training then involves finding a balance between maximising the margin for correctly classified points, and minimising the penalty for incorrectly classified points. We can introduce a hyperparameter to control the balance between these two objectives.

Further reading: Cortes and Vapnik (1995) "Support-vector networks" https://link.springer.com/article/10.1007/BF00994018

Nonlinear Decision Boundaries



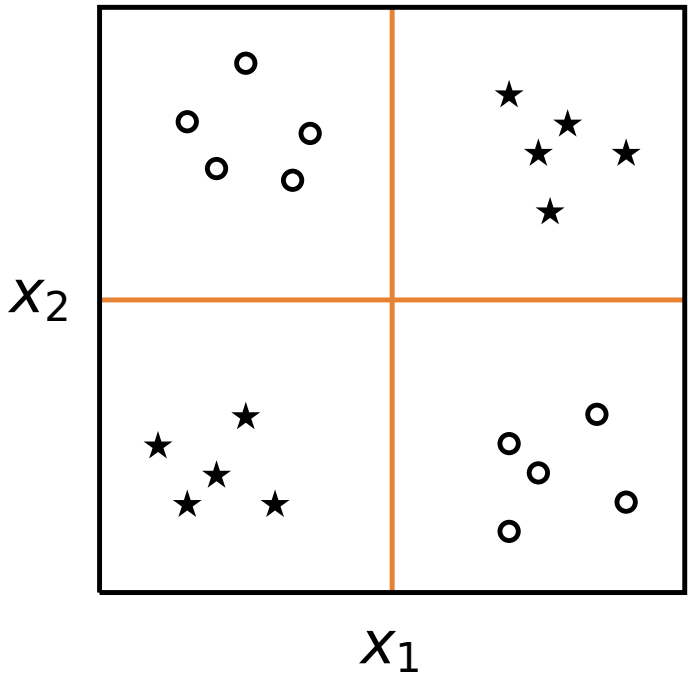
For some datasets, we need nonlinear boundaries.

Kernel Methods

- Input space → Feature space
- Linear boundary in feature space
- e.g. $(x_1, x_2) \mapsto (x_1, x_2, x_1 x_2)$

Linear decision boundaries are easy to work with. Rather than abandoning them completely, one option is to map the input space to a higher-dimensional feature space, and then calculate a linear boundary in that space. This is the first important idea for kernel methods.

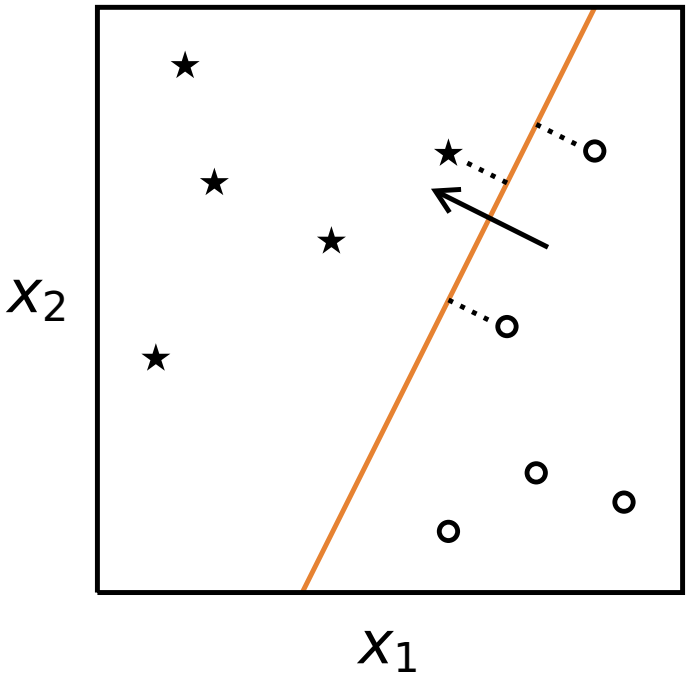
Kernel Methods



$$xy - x - y + 1 = 0$$

Mapping $(x_1, x_2) \mapsto (x_1, x_2, x_1 x_2)$ allows us to deal with this example.

Maximum Margin



The second important idea for kernel methods is to notice that only a few training points (the "support vectors") are necessary to define the hyperplane. (This is more extreme when there are many points and few dimensions.)

The third important idea is that we can represent the classifier just in terms of dot products with the support vectors. This relies on the fact that we are working in a vector space, and the boundary is linear. We can represent a vector perpendicular to the boundary using a weighted sum of the support vectors ($\star - \frac{2}{3}\circ_{\text{top}} - \frac{1}{3}\circ_{\text{bottom}}$). Taking a dot product with this vector lets us check where a point is, relative to the boundary. This means, rather than thinking about what *boundary* to use, we can think about what *points* to use (and with what weights).

(Stephen Clark's notes from 2015/2016 give more details on this equivalence, describing a "dual form" of the perceptron algorithm. However, be aware that those notes use a different notation, and use the term "feature" in a slightly different way – for structured prediction tasks (such as parsing), it can be useful to talk about "features" of the output.)

Kernel Methods

- Can represent and train an SVM only using dot products in feature space
- Kernel function in input space = dot product in feature space

The final important idea is to think about the dot product, not in feature space, but in the input space. As a function over the input space, it is called a kernel function.

This is efficient when the feature space is sparse (for most points, values in most dimensions are 0), which means that calculating the kernel function can be much faster than calculating the mapping to feature space.

String Kernel

Number of shared substrings

The dog barked The dog slept

In this example, there are 3 shared substrings: “the”, “dog”, and “the dog”.

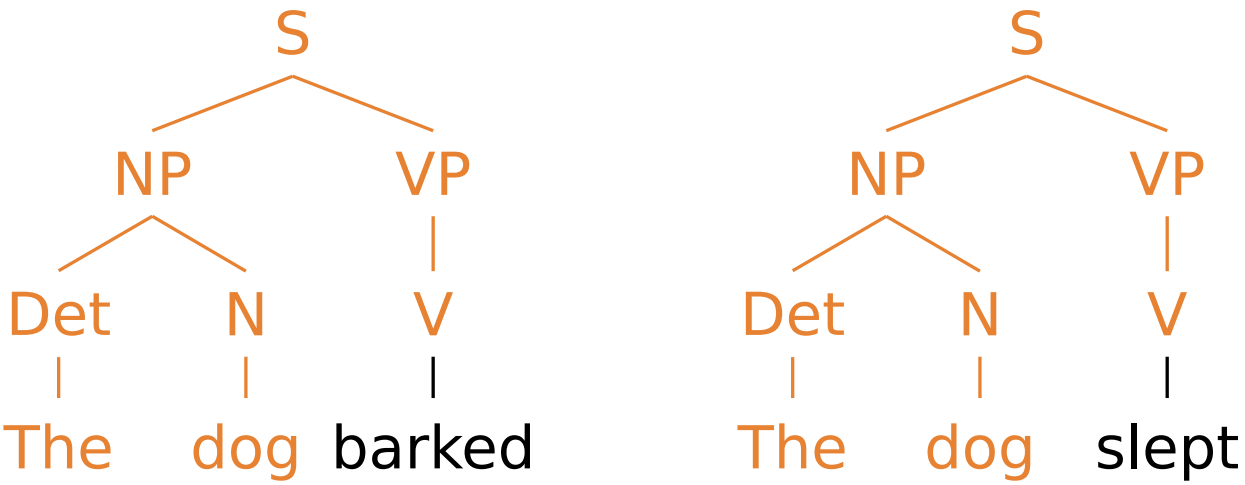
Each possible string is a dimension of the feature space – so this is an infinite-dimensional space! The value for each dimension is the number of times that substring appears – almost all dimensions will have value 0.

One method to calculate the string kernel is to use dynamic programming. A more efficient method, which runs in linear time, is to use a suffix tree.

Further reading:
Collins and Duffy (2002)
http://aclweb.org/anthology/P02-1034
Vishwanathan and Smola (2003)
http://papers.nips.cc/paper/2272-fast-kernels-for-string-and-tree-matching.pdf
Moschitti (2006)
http://aclweb.org/anthology/E06-1015

Tree Kernel

Number of shared subtrees



In this example, there are 48 shared subtrees (if allowing single-node subtrees, and allowing subtrees that include some but not all children of a node).

Each possible tree is a dimension of the feature space.

The tree kernel can be calculating using dynamic programming, or more efficiently using a suffix tree – see references on previous slide.

The tree kernel can be useful for parsing (e.g. see: Collins and Duffy, 2002).

Trees can be seen as a generalisation of strings. This can be further generalised, viewing strings as 1-dimensional, and trees as 2-dimensional, but I am not aware of work on "higher-dimensional trees" in NLP. Further reading: Rogers (2003) https://link.springer.com/article/10.1023/A:1024695608419 [paywall]