Machine Learning for Language Processing
Lecture 1: Classification

Stephen Clark

October 3, 2015

The ML Revolution in NLP The plot on the slide shows the percentage of papers at the main ACL conference which report research on statistical NLP. Today, in 2015, the figure would be close to 100%. Before 1990, research in NLP was rule-based, where the rules were written by domain experts (for example translators, for machine translation). The limitations of rule-based systems are well-documented: large and complex rule sets which are difficult to modify and maintain; expensive human input needed to create the rule sets; and difficulty in adapting rule sets designed for one language domain, e.g. finance, to another, e.g. biomedical text.

However, there are advantages to rule-based systems, and it is important to maintain some balance in the debate regarding rule-based vs. statistical. Many commercial NLP systems still use rule-based approaches, for the following reasons. One, they are typically *high precision*, meaning that, if a rule, or set of rules, does fire, the rule will probably give the right answer; and two, the rules are easy to inspect manually to determine why a system made a particular decision. The second reason is especially valued in the context of safety-criticial systems: if an automatic air traffic control system instructs two planes to land at the same time on the same runway, explaining that ten layers of a deep neural network made the decsision is going to provide little comfort.

So why did machine learning take over NLP (and other related disciplines such as speech recognition and computer vision)? The main reason is that ML provides a principled means of learning the required knowledge for intelligent behaviour (*assuming we can provide appropriate training data and an appropriate learning objective*). The amount of knowledge required is huge, and varies for each domain, so it is unlikely that we could ever solve this problem through hand-written rules.

Some History In the 1950s, the disciplines related to the cognitive sciences, such as psychology and linguistics, were dominated by ideas from emprícism, the philosophical doctrine which states that all knowledge is based on experience derived from the senses. Shannon had also recently published his theory of communication, based on information theoretic ideas grounded in probability theory.

In 1957, Chomsky effectively founded the modern discipline of linguistics, with the publication of Syntactic Structures. This book took a more rationalist approach, with the key data source for linguistic theories being the introspective judgements of native speakers. In 1969, Minsky and Papert published Perceptrons, which was a criticism of Rosenblatt's perceptron (a simple form of neural network), arguing for more traditional, knowledge-based approaches to AI. Both books were enormously influential, and rationalist approaches took over from empiricism.

The tide began to turn again in the 1980s. The IBM group, led by Fred Jelinek, had already been successful at applying statistical methods to automatic speech recognition.[1] Neural networks were back on the agenda with the work of the PDP group, although it would take many more years, and the availability of large datasets and powerful machines, before neural networks could become the dominant force they are in 2015. Finally, the IBM group started to apply the same statistical techniques that had been successful for speech recognition to the machine translation problem (before many of that group, including Peter Brown and Bob Mercer, went to Wall Street to found Renaissance Technologies[2]).

For an interesting perspective on machine learning research in NLP, see [1]. This paper takes the position that perhaps empiricism has been *too* successful in NLP, and some combination of empiricism and rationalism is required.

Statistical NLP not Science? The rejected COLING 1988 submission referred to on the slide is the original conference paper describing IBM's approach to statistical machine translation, which is now the dominant approach to MT.[3] The rather hostile tone taken in the review was not uncommon at the time, and can be interpreted as reflecting a general misunderstanding of how probabilistic and machine learning approaches are used in science and engineering. For a recent defence of statistical methods in NLP, see [2].

Text Classification Text classification is the problem of, given a linguistic unit as input (word, sentence, document), assign a discrete label from a finite set to that input. The classic text classification problem is, given a document as input, assign a label to it which specifies the document's topic. For example, we may wish to classify newspaper reports according to the section of the newspaper they are in, in which case the labels would be *politics, finance, sport, gardening, travel, cookery*, and so on. Another text classification problem is spam detection. Here the inputs are emails, and the output is a single label from the set { *spam, not-spam* }. A final example is sentiment analysis, where the input could be a document, an email, a tweet, or a smaller input such as a sentence or word (in context). The labels in this case would be { *positive, negative* }.

[1] "Every time I fire a linguist ..."
[2] http://cs.jhu.edu/∼post/bitext/
[3] http://www.statmt.org/

Machine Learning Framework Any machine learning framework relies on the notion of features, which can be thought of as the way that the input is represented mathematically in order to perform the classification. We'll be more precise later about how features are defined mathematically, but for now a useful intuition is to think of features as patterns in the input which we think will be useful for making the classification decision. If the input is a sentence, a feature could be a word in the sentence, or a sequence of words, or a ⟨word, part-of-speech tag⟩ pair, or a more complex pattern such as part of a parse tree. The set of features for a particular input will be represented as a *feature vector*.

Training and Test Data Training data is used to train the classifier. We'll mostly be concerned with the *supervised* setting, where each instance of the data also comes with a manually assigned label (for example a set of documents where the correct topic label for each document has been given). *Unsupervised* training is the general problem of finding structure in the input data, often framed as a clustering task, where the cluster labels — or even the number of clusters in the most general case — are not provided in advance.

Finally, there is also *reinforcement learning*, where the training signal is provided in the form of a reward after a number of actions have been taken by the system being trained. One application where reinforcement learning is used in NLP is dialogue modelling. Here, the problem is to learn a dialogue system which knows how to respond to utterances from the user, for example in order to book airline tickets, but the only signal provided during training is whether the required flight was successfully booked, after a number of interactions between the user and system have taken place.

The goal of the training process is to learn a classifier that generalises well to unseen data, i.e. instances not observed in the training set. Hence it is important that the training data and test data — the latter used to evaluate the accuracy of the classifier — are distinct. It is also important that the test data is not used "too often", or more directly to tune the value of a hyperparameter. A separate test set, called a development set, is often held out for this purpose.

Machine Learning-based Decisions One way to partition the options available for building a classifier is as follows. First, there are the generative models: *joint* probability distributions over the input and output. Second, there are the discriminative models, *conditional* probability distributions of the output given the input. And third, there is a more general class of discriminative functions, which do not attempt to learn probability distributions, but learn the mapping from inputs to outputs more directly (support vector machines are an example).

We'll encounter all three approaches during the course. All three have advantages and disadvantages, and which one to apply depends on various factors.

Naive Bayes Classifier Suppose we are interested in the conditional probability of a class given a document. Rewrite this using Bayes theorem, so that it is proportional to the product of the conditional probability of the document given the class (the *likelihood*) and the marginal probability of the class (the *prior*). The advantage of writing it this way is that there is an easy way to factor the likelihood.

Bag of Words Model Suppose we model a document as just a *set* of words, i.e. using a vector of indicator functions, where each component of the vector (a feature) corresponds to a word.[4] Now assume that each word is conditionally independent given the class. Then the conditional probability of the words (the features) given the class is just the product of the probabilities of each word given the class. The advantage of this formulation is that the component probabilities are easy to estimate.

Probability Estimation Estimating the prior probability of $P(c_j)$ is easy: just count the number of documents in the training data that have been given label c_j and divide by the total number of documents.

Estimating the likelihood probability of $P(f_i|c_j)$ is equally easy: just count the number of times that word f_i appears in documents assigned class c_j and divide by the number of documents assigned class c_j.

These relative frequency estimates can be theoretically justified as *maximum likelihood* estimates; we'll see more on maximum likelihood estimation later in the course.

What if word f_i has not appeared in any documents with class c_j? In this case the relative frequency estimate is zero, and the probability for the whole document will be zero (since the zero will propagate through the product). The way to solve this problem is to use a *smoothing* technique, where some of the probability mass is taken from the seen events in the training data and given to unseen events. There is a large literature on how to reassign the mass, since the *sparse data problem* in NLP means that we are often dealing with unseen events at test time. One trivial method is to simply replace each zero count with ϵ, where ϵ is a small real value greater than zero (being careful to renormalise the relevant distributions so that they sum to 1). We'll see more sophisticated smoothing methods later in the course.

Sentiment Classification The paper which began the now huge literature on sentiment analysis was [3]. The models used in the paper are straightforward, as well shall see, but what was noteworthy about the paper was the insight that manually annotated training data could be freely obtained from online movie review sites. For the experiments reported in the paper, they had a dataset consisting of 752 negative reviews and 1,301 positive ones.

Bag of Words Model The baseline results show the accuracy that can be achieved by simply counting the number of occurrences of "sentiment bearing" words in a document, where the lists of sentiment bearing words were created

[4]The extension to the multiset, or *bag*, case, where the words are counted rather than simply being present or absent, is straightforward.

manually (in the second case by also inspecting the test data). Machine learning methods can be applied using a similar idea, but this time the positive or negative sentiment of a word is learned automatically from the data.

Two Naive Bayes models were used, one where only the presence or absence of a word is taken into account (the binary model described earlier), and one where the words are also counted. In the latter case the representation of the document as a feature vector is different, since each word *token* in the document is effectively treated as a random variable, with the value being a word from the vocabulary.

Results There are two noteworthy aspects of the results. First, the Naive Bayes binary model performs surprisingly well, on a par with the frequency-based model, and almost as good as the more sophisticated models using support vector machines and maximum entropy classifiers. Second, the unigram model which only looks at single words performs surprisingly well, with little value in adding additional feature types such as bigrams or parts-of-speech.

More Recent Work on Sentiment Richard Socher's work from Stanford on using (recursive) neural networks for sentiment analysis has gained a lot of interest. However, note that this work also involves a large training set collected using crowd sourcing, so how much is being gained from the fancy neural network is not entirely clear.

Readings for Today's Lecture

- Chapter 16 of Manning and Schütze's Foundations of Statistical NLP – Text Categorization (perhaps glossing over some of the more technical descriptions of the various models at this stage)

References

[1] Ken Church. A pendulum swung too far. *Linguistic Issues in Language Technology LiLT*, 2(4), 2007.

[2] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24:8–12, 2009.

[3] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up? sentiment classification using machine learning techniques. In *Proceedings of EMNLP*, pages 79–86, 2002.

Machine Learning for Language Processing
Lecture 2: POS Tagging with HMMs

Stephen Clark

October 6, 2015

The POS Tagging Problem We can't solve the problem by simply compiling a tag dictionary for words, in which each word has a single POS tag. Like most NLP problems, ambiguity is the souce of the difficulty, and must be resolved using the context surrounding each word. Try inputting *I can can a can* into a POS tagger.

Probabilistic Formulation We'll formalise the problem as finding the most probable tag sequence, given the word sequence as input. In early debates regarding statistical methods in AI, the rule-based adherents would always say *sure, but where do the probabilities come from?* We now know the answer to this question: probabilities are estimated from data. However, in the general case, the difficulty of finding appropriate data should not be underestimated. In the POS tagging case, creating manually annotated data, in order to do supervised machine learning, is not too expensive, and is a task a trained linguist can do relatively quickly.

The second question is how to find the arg max, or *what's the search algorithm?* For HMMs there is a very nice dynamic programming solution to the problem of finding the highest scoring tag sequence: the Viterbi algorithm (for sequences).

HMM Tagging Model We'll rewrite the conditional probability of tag sequence given word sequence using Bayes theorem, thereby reversing the direction of the conditional. The reason for doing this is that the likelihood $P(W|T)$, and the prior $P(T)$, are easier to factorise into probabilities that we can estimate than the original conditional, $P(T|W)$. The denominator $P(W)$ can be ignored since W is a constant for the elements of the set we're finding the arg max over.

In order to factorise the two probabilities we first apply the chain rule. Note that the application of the chain rule is *exact*, so no advantage has been gained at this point in terms of ease of estimation: Estimating $P(t_n|t_{n-1}, \dots, t_1)$ is no easier than estimating the full joint probability $P(t_1, \dots, t_n)$. However, the application of the chain rule allows us to make independence assumptions.

For the tag sequence probability, we assume that the probability of a tag is only dependent on the previous $n-1$ tags; such a model is called an n-

gram model. For the equation on the slide, the probability of a tag is only dependent on the previous tag, giving a *bigram* model. Note that $P(t_i|t_{i-1})$ is much easier to estimate than the full conditional probability. For the word emission probabilities, we assume that the probability of a word only depends on its corresponding tag.

Clearly both independence assumptions are somewhat drastic. However, relaxing the independence assumptions leads to probabilities which are a) harder to estimate; and b) less efficient to compute with. There is a large literature on exploring this trade-off, for example investigating whether accuracies can be improved by increasing the size of the n-gram for the tag sequence probabilities. Here we find that moving beyond a trigram does not typically help for POS tagging (at least with the sizes of supervised training data available).

N-gram Generative Taggers Note that the HMM is a *generative* model, since we're effectively using the joint distribution: $P(W|T)P(T) = P(W, T)$. Intuitively the model generates both words and tags according to some stochastic process (even though the words are given as input).

Another solution is to use a conditional, or discriminative, model. The advantage of such models is that they are more flexible and do not rely so heavily on independence assumptions. One example of such a model is a maximum entropy tagger, or a tagger based on conditional random fields. We'll encounter such models later in the course. The downside is that the parameters of such models are typically harder to estimate, often requiring numerical methods rather than having a closed-form solution.

Parameter Estimation For an HMM tagger there are two sets of probabilities that need estimating: the tag transition probabilities and the word emission probabilities. Note that the model does not contain paramaters of the form $P(t_i|w_i)$ (because of the application of Bayes theorem), which could be considered counter-intuitive given that the goal is to estimate $P(T|W)$.

There has been lots of work on attempting to build taggers using raw data, treating the problem as an unsupervised machine learning problem, effectively trying to learn clusters of words which correspond to linguistically plausible parts of speech. However, by far the most popular approach to POS tagging is to treat it as a supervised machine learning problem, in which we assume the existence of a manually labelled training set.

Relative Frequency Estimation Suppose I toss a coin 1000 times, and it comes up heads 480 times. An intuitive, and statistically reasonable, estimate of the probability of heads for this coin would be $480/1000 = 0.48$. It turns out that, for the coin tossing case, the relative frequency estimate is also the *maximum likelihood* estimate: 0.48 is the value which makes it most likely that, out of 1000 tosses, heads will appear 480 times. (It is easy to show this formally with some high school calculus.)

The tagging scenario is similar, but with a move from the Bernoulli case, where the value of the relevant random variable (the coin toss) is zero or one, to the Categorical case, where the random variable can take a number of values from a finite set (for the tag transition probabilities it's the tag set; for the word emission probabilities it's the word vocabulary). It turns out that the same mathematics applies to the more general case, and relative frequency estimates for the HMM parameters are maximum likelihood estimates here also. Collins' thesis [2] (Section 2.3) has a very nice presentation of this result.

Smoothing for Tagging There is one aspect of the language estimation problem which is crucially different from the coin tossing scenario. It is easy to toss a coin many times, in order to get a reliable estimate. If a coin has been tossed 1,000 times, we'd be reasonably confident of the relative frequency estimate. However, suppose the coin has only been tossed 5 times, and it came out heads once. How confident would you be now in an estimate of 1/5 for the probability of heads?

What makes statistical modelling difficult in the language case is that we often find ourselves in this *sparse data* position. Since many words are observed rarely, relative frequency estimates are often unreliable. This unreliability is an instance of the machine learning problem of *overfitting* where, intuitively, we're making the probability estimates look *too* much like the data; we're trusting the data too much.

An extreme version of the overfitting problem occurs when events are entirely unseen in the training data, giving frequencies of zero. In the case when the *conditioning* event has not occurred, the relative frequency estimate is undefined, since the denominator is zero ($f(t_{i-1}) = 0$ for estimating $P(t_i|t_{i-1})$). When the event being generated has not occurred, the relative frequency estimate is zero ($f(t_{i-1}, t_i) = 0$ leads to $\hat{P}(t_i|t_{i-1}) = 0$).[1] Zero estimates are problematic because they propagate through the product, giving a zero probability for the whole sequence.

The solution to the overfitting problem is to use *smoothing* techniques, where we take some of the probability mass from the seen events in the training data, and give it to rare and unseen events. Many of the smoothing techniques used in NLP were originally developed for language modelling for speech recognition [1].

The solution on the slide is an example of the *linear interpolation* smoothing technique. The idea is to use progressively more general conditioning events, so that the chances of obtaining zero estimates is reduced. There are various methods available for estimating the λ's. One useful intuition is that we'd like higher values for the relevant λ when the corresponding conditioning event has occurred frequently (the analogue of having tossed the coin many times).

Another related solution is *backing off.* Here, we will use just one of the estimates, rather than mix them, with the decision of which level to use based

[1] $\hat{P}$ is used to denote a (maximum likelihood) estimate of P.

on the frequencies in the relative frequency estimate (e.g. use the first estimate which is not zero).

Better Handling of Unknown Words Suppose we wish to tag a word not seen in training data, e.g. *Trumpington* or *googling*. The capital T in Trumpington — assuming it's not the first word in the sentence, and we're tagging English — is a strong clue the tag should be NNP. Similarly, the *ing* suffix in *googling* is a strong clue the tag should be VBG. One way to adapt the HMM to incorporate such clues is to modify the generative process so that, for unknown words, various features of the word are generated, rather than the word itself. However, making sensible independence assumptions, and deciding which features to generate, is difficult. This difficulty is one of the main motivations for using discriminative models, which allow us to model such features directly.

The Search Problem for Tagging When performing the arg max, the number of tag sequences being searched over grows exponentially with the length of the sentence; simply enumerating all sequences and picking the highest scoring one is not going to work. For a *unigram* model, there is a trivial solution which is efficient and optimal: pick the most probable tag for each word, according to the probability $P(t)P(w|t)$. This algorithm is linear in both the length of the sentence and the size of the tagset.

A Non-Trivial Search Problem Consider now a bigram tagger. Suppose we try and use a similar solution to that for the unigram tagger, picking the most probable tag for each word w_i, according to the probability $P(t_i|t_{i-1})P(w_i|t_i)$ (where t_{i-1} was the tag chosen for the previous word). The example on the slide is designed to show that this algorithm is not optimal. The intuition is that, because of the dependence on the previous tag, the highest scoring tag at any particular point in the left-to-right tagging process could be overtaken by a lower scoring tag when the rest of the sentence is considered. (This intuition does not hold in the unigram case: there is no way that the highest-scoring tag for a word can be overtaken by another tag, because the probability does not depend on the previous tag.)

The Viterbi Algorithm The Viterbi algorithm is a dynamic programming (DP) algorithm, so like all DP algorithms requires the optimal sub-problem property. What this means in practice is that we need to break the larger problem into smaller sub-problems, such that the sub-problems can be solved in the same way (eventually "bottoming out" in a base case which can be solved trivially).

A useful intuition is the following: suppose we want to find the highest-scoring sequence ending at word w_n. Suppose further that we know the highest-scoring (sub-)sequences ending at w_{n-1} for each tag. So we know the highest-scoring sequence ending at w_{n-1} ending in DT; and the highest-scoring sequence ending at w_{n-1} ending in VBG; and the highest-scoring sequence ending at w_{n-1}

ending in NNP; and so on for all tags. From there we need $O(T^2)$ calculations (where T is the size of the tagset), based on the tag transition probabilities between w_{n-1} and w_n, and the word emission probabilities at w_n, to calculate the highest-scoring sequence ending at w_n.

Now apply this idea recursively to find the highest-scoring sequence ending at word w_{n-1} (for each tag). For the base case, calculating the highest-scoring sequence ending at w_1 for each tag is trivial, since there are no previous tags.

Viterbi for a Bigram Tagger The recursion on the slide formalises the intuition given above. In practice, the recursion is implemented by keeping track of the highest-scoring sub-sequences, for each tag and each position, as the tagger moves from left to right (by recording pointers to previous tags); and then the final step is to follow one set of these pointers from the end of the sentence to the start, tracing out the highest-scoring sequence for the whole sentence.

The algorithm is linear in the length of the sentence and quadratic in the size of the tagset. For a trigram tagger, the algorithm is cubic in the size of the tagset (and quartic for a 4-gram tagger, and so on).

Practicalities Taggers based on HMMs can be made highly accurate; e.g. the TnT tagger, which is now over 15 years old, is still competitive, and very fast. In particular, the training can be performed extremely efficiently on large datasets, since the training is essentially just counting. However, the ease of including rich, overlapping features into taggers using discriminative models — the topic of the next lecture — mean that HMM-based taggers are no longer the method of choice.

Readings for Today's Lecture

- Chapters 9 (Markov Models) and 10 (Part-of-Speech Tagging) of Manning and Schütze's Foundations of Statistical NLP; and Chapters 5 (Part-of-Speech Tagging) and 6 (Hidden Markov and Maximum Entropy Models) of Jurafsky and Martin (2nd. Ed.)

References

[1] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. Technical report, 1998.

[2] Michael Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University of Pennsylvania, 1999.

Machine Learning for Language Processing
Lecture 3: Maximum Entropy Models

Stephen Clark

October 6, 2015

Discriminative Models With a discriminative model $P(w_j|\boldsymbol{x})$ there is no need to model the joint distribution $P(w_j, \boldsymbol{x})$. For many NLP tasks this makes sense, since the input $\boldsymbol{x}$ is given and constant across the set of possible outputs, so why model $\boldsymbol{x}$? Modelling the conditional distribution directly is also arguably closer to the classification process, which assumes a fixed input. Finally, there is no need to specify a generative process, with all its associated independence assumptions, which may not be correct.

NER as a Tagging Problem We'll use Named Entity Recognition (NER) as an example of a tagging problem, as well as POS tagging, since NER is a problem that benefits from rich, overlapping features. The main motivation for MaxEnt models is to enable the use of such feature sets. Here we're using a standard NER tagset: { LOC, PER, ORG, DAT, TIME } (location, person, organisation, date, time).[1] I-PER, for example, should be read as "inside person". Sequences of such tags pick out named entities, with B-PER being used if two separate person entities are contiguous.

Feature-based Models Features in a MaxEnt tagging model should be thought of as pairs: elements of the context paired with a particular tag. For example, if the word being POS tagged starts with a capital letter, then that's positive evidence for the NNP tag. The feature which encodes this evidence is ⟨ title_caps, NNP ⟩; and the associated weight, which is learnt during the training process, encodes the strength of evidence.

Complex Features The key point about Maximum Entropy models is that the features can be arbitrarily complex, and they don't have to be statistically independent (unlike in the Naive Bayes model). For example, when doing NER tagging, it might be useful to know that the topic of the document is cricket, rather than, say, hill walking, in which case *Lancashire* might be tagged as an organisation (the cricket club) rather than location (the county). Incorporating

[1] It's possible to use much larger tagsets, depending on the application.

such information into an HMM tagger would be non-trivial; in a MaxEnt tagger it can easily be incorporated into the features.

Feature-based Tagging The downside of MaxEnt models, compared to the HMM, is that a more complex estimation method is required – although by current machine learning standards in NLP, the estimation is still relatively easy, since the objective function is convex (at least for MaxEnt models for supervised taggers).

Features in MaxEnt Models Mathematically a feature is a pair, or equivalently, a binary-valued indicator function which takes the value 1 when the pair is given as input to the function, and 0 otherwise.[2] The part of the feature definition which picks out the relevant part of the context is often called a *contextual predicate*.

The Model The form of the MaxEnt model is *log linear*, a general form of probabilistic model which is very popular in machine learning. Here we're using a conditional model: the probability of a tag, t, given a context, C. For the tagging problem, C is usually defined as a fixed-word window either side of the word being tagged, but in general C can contain any information deemed relevant to the tagging decision.

For a particular tag, context pair, we say that a feature "fires" or is "on" when it has the value 1. The total number of features n can be very large, even in the millions. However, only a small proportional of these features will fire for any particular tag, context pair.

Tagging with MaxEnt Models The model on the previous slide is a classification model. (In fact, Pang et. al used such a model for sentiment classification; see Lecture 1.) A simple way to use a classification model for sequence tagging is to chain the decisions together, multiplying the probabilities [2]. Crucially, the context for a particular decision will include the tags assigned to the previous word, much like the n-gram HMM tagger, and features will be defined in terms of those previous decisions. The search problem is very similar to that for an HMM tagger, and a variant of the Viterbi algorithm can be used to find the highest-scoring sequence of tags according to the probability of the tag sequence given the word sequence, $p(t_1 \dots t_n | w_1 \dots w_n)$.

There is a theoretically more pleasing approach to conditional sequence modelling, which is the Conditional Random Field (CRF) [1]. CRFs have the appealing property that the sequence probability is *globally* optimised, during training and testing. In contrast, the training for the MaxEnt model is *local*, in the sense that it effectively treats each tagging decision as a separate training instance (whilst still conditioning on the previous tags). There are a number of

[2]Entending MaxEnt models to handle count-based, or even real-valued, features is straightforward.

DP algorithms defined for CRFs, including the Viterbi algorithm for finding the highest-scoring tag sequence at test time.

Whether the theoretical niceties of the CRF lead to better performance over MaxEnt taggers is unclear. In practice, MaxEnt models can usually be made to perform as well as CRFs by using a large, rich feature set.

Model Estimation There are two ways of approaching the estimation problem for MaxEnt models. One is to simply assume the log-linear form given earlier, and use maximum likelihood estimation (i.e. write down the probability of the data assuming a log-linear model, and find the weight values which maximise that probability). Another, which derives the log-linear form, is to start with a natural set of constraints, and then choose the maximum entropy model from the set of models which satisfy those constraints.[3] Remarkably, both approaches lead to the same estimates for the weight values.

The Constraints A natural choice for the constraints is to say that the expected value of each feature according to the model should be equal to the empirical expected value. The empirical expected value is just the number of times the feature turns up in the data. The expected value according to the model is the prediction the model makes about how often it thinks the feature should turn up.

Choosing the MaxEnt Model These constraints do not pick out a single model, so we need a method of selecting from the set of models that do satisfy the constraints. A natural choice is the model with maximum entropy. The entropy H of a distribution p is a measure of how uniform the distribution is.[4] The expression given on the slide for $H(p)$ is the *conditional* entropy, since we're dealing with a set of conditional probability distributions.

The Maximum Entropy Model One intuitive motivation for selecting the maximum entropy model is that it's the most uniform model which satistfies the constraints provided by the data, so in some sense, when selecting it, we are making no assumptions outside of what is observed in the data.

One simple method for estimating the weights of a MaxEnt model is Generalised Iterative Scaling (GIS).

Generalised Iterative Scaling (GIS) We're much more sophisticated in NLP now regarding numerical optimisation than we were in the late 90s, when MaxEnt models were starting to become popular. Hence a standard method now for estimating the weights would be to just optimise for (log-)likelihood, using any standard gradient-based optimisation technique. The gradients are easy to calculate — in fact they're just the differences between the expected

[3]By model we just mean the set of conditional probability distributions we're trying to estimate.

[4]$H(p) = -\sum_x p(x) \log p(x)$

and empirical feature values — and, since the likelihood is convex, global optimisation is easy.

However, it is worth considering GIS since it has a clear intuition, is easy to implement, and, for the tagging problem at least, training is relatively quick. (Training on the Penn Treebank for the C&C POS tagger takes roughly 10 minutes for 100 iterations.)

POS Tagger Features The features on the slide, which are a fairly standard set for a POS tagger, look at the words in a 5-word window centred on the word being tagged. These also include the previous two tags, assuming a left-to-right tagging process.

POS Tagger Features for Rare Words The more interesting features, and the ones which display the flexibility of MaxEnt models, are those that only apply to the rare words (here defined as words that occur less then 5 times in the training data). The idea is that, if a word occurs enough times in the training data, we have good evidence for its tag by simply looking at the word itself and the words (and tags) in the local window. However, for rare words it's useful to look at the internals of the word itself, for example its prefixes and suffixes, whether it contains a digit, uppercase character or hyphen.

The range of features that can be defined in this way is only restricted by the imagination of the model developer. Whether such features help with accuracy is an empirical question, of course, and needs to be tested using the training and (development) test data.

Performance MaxEnt taggers still offer competitive performance, although taggers based on RNNs may turn out to be superior, especially if you are interested in maintaining accuracy across domains (since RNNs tend to generalise better).

Contextual Predicates for NER The final set of slides are designed to show the range of features that can be added to a tagger, in this case the C&C NER tagger.

Readings for Today's Lecture

- Chapter 6 (Hidden Markov and Maximum Entropy Models) of Jurafsky and Martin (2nd. Ed.)

References

[1] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289, Williams College, MA, 2001.

[2] Adwait Ratnaparkhi. A maximum entropy part-of-speech tagger. In *Proceedings of the EMNLP Conference*, pages 133–142, Philadelphia, PA, 1996.

Machine Learning for Language Processing Lecture 4: The Perceptron for Structured Prediction

Stephen Clark

October 8, 2015

Local and Global Features The local features we have already seen for the maxent tagger can be extended to global features by simply summing up the values of the indicator functions for each position in the sentence. In other words, whilst the local features are indicator functions over local contexts, the global features become *count* functions over the whole sequence.

These global features are used in CRF taggers. The perceptron tagger that we describe here can be thought of as an alternative to the CRF: a global model, but without a probabilistic interpretation and with a different training algorithm. (The decoding algorithm remains the same: the Viterbi algorithm for sequences.)

A Linear Tagging Model The linear tagging model is similar to a MaxEnt model, but without the exponentiation and the normalisation constant. You may be asking: is a linear model powerful enough to solve complex NLP problems? (And didn't Minsky and Papert demonstrate a fundamental weakness of linear perceptrons back in the 60s?) There are two possible responses. The first response is to say that it all depends on the features. With a rich enough feature set, a linear model can always be made powerful enough. Indeed, the whole point of kernels, which we'll encounter later in the lecture, is to allow the use of linear models with extremely complex feature sets.

The second response is to say it's an empirical question. There is a large volume of current work applying non-linear neural networks to various tasks, some of them using many layers in the network; but whether these will turn out to "solve NLP", as some researchers are claiming, remains to be seen. In my experience, the structured perceptron with a large number of features usually provides a very strong baseline for structured prediction tasks [2].

Decoding for a Global Linear Model Decoding is performed in the same way as it was for the HMM and MaxEnt taggers, using the Viterbi algorithm for sequences. The restriction on features, in order that the decoding is efficient, is the same as it was for the MaxEnt tagger: features defined over any part of the input sentence are fine, but features which look over large parts of the generated tag sequence increase the complexity (in the same way as for any n-gram tagger, where the complexity is $O(T^n)$ for a tagset of size T).

The Perceptron Training Algorithm The algorithm is extremely simple: perform a number of passes over the training data, decoding the training instances. If the current model makes a mistake on a training instance, do a simple update. The fact that the weights (potentially) get updated at each instance means that the training is *online*. The fact that no update is performed if the training instance is decoded correctly means that the training is *passive*.

The Training Algorithm (with words) The update works as follows: for each local feature instance in the correct, gold sequence, add 1 its weight value; for each local feature instance in the output given by the tagger, take 1 from its weight value. I use the term *local feature instance* here since the overall update happens globally; i.e. the count of a feature has to be correct across the whole sequence in order for the corresponding feature weight to receive no update.

Averaging Parameters Intuitively, the perceptron training scheme forces the model to perform as well as possible on the training data; but this can lead to overfitting and a lack of generalisation on unseen data. One simple and effective method to combat overfitting is parameter averaging [1]. The set of weights is recorded for each training instance, for each pass over the data, and then averaged. (So for 40,000 training instances and 10 passes, for example, we'd have 400,000 weight vectors to be averaged.) The idea is to mix the earlier weight vectors, which haven't yet converged, with the later ones, which may be overfitted to the training data.

The averaging parameters idea is closely related to the voted perceptron, where, instead of averaging the weights, each instance of the weight vector is used for decoding at test time, and the output is determined through a voting procedure [1]. The obvious disadvantage with this scheme is that, for 40,000 training instances and 10 passes over the training data, the test data has to be decoded separately 400,000 times. The voted perceptron has some theory associated with it, explaining why it is effective.

Theory of the Perceptron The theory of the perceptron [1] shows that, *if the data is separable with some margin*, then the converged model will perform perfectly on the training data. Having (linearly) separable data in this case means that there is some setting of the weights values which will score each correct output higher than all the incorrect ones (by some margin), for each training instance. What if the training data is not linearly separable? There are guarantees here as well. As long as the data is "close" to being separable, then the number of mistakes on the training data will be small.

Note that, so far, the theoretical discussion has only been in terms of the training data, whereas what we care about is generalising to unseen, test data. There is a pleasing story here as well: if the number of mistakes on the training data is small, then there is a good chance that the number of mistakes on the test data will also be small (making the usual assumptions about the training and test data being drawn from the same distribution, and so on).

A *Ranking* Perceptron The rest of the lecture is geared towards the use of kernels, allowing extremely rich and complex feature sets to be computed with efficiently. We will use the ranking perceptron to motivate the use of kernels, where we assume that a relatively small number of possible outputs have already been generated. The reason to focus on the ranking problem is that we'd like the decoding problem to be trivial, rather than requiring dynamic programming (which would add an extra layer of complexity).

In order to ground the discussion, suppose that we have 1,000 parse trees generated by a statistical parser, for a number of training examples, with the correct tree among those 1,000. The goal of the ranking perceptron is to use models with complex features in order to rank each set of 1,000 trees, so that the correct tree is scored higher than all the incorrect ones (for each training instance). The intention is that the rich models will perform better than the original parsing model (which was likely constrained in terms of the features it could exploit due to the use of DP, or other methods, for navigating the original, very large search space).

Training (with the new notation) The pseudocode on the slide shows just one pass over the training data. It's the same algorithm as before, but using the notation introduced on the previous slide. The purpose of the new notation will become clear.

Note that the calculating the $\arg\max$ in the ranking scenario is trivial: just exhaustively loop through the candidates $C(s)$ for training sentence s, calculating the score for each.

Perceptron Training (a duel form) The key to the duel form perceptron training is the scoring function $G(\mathbf{x})$, where $\mathbf{x}$ is a feature vector (in our example the features associated with a parse tree). The new scoring function works by taking $\mathbf{x}$ *and comparing it with every other parse tree feature vector in the training data.* $G(\mathbf{x})$ will be high if $\mathbf{x}$ is more similar to the correct parse trees in the data than the incorrect ones.

The training procedure also has a set of weights, but this time a weight α_{ij} for each parse tree vector $\mathbf{x}_{ij}$ in the training data. Note that α_{ij} gets increased if the tree $\mathbf{x}_{ij}$ is returned as the highest scoring one for sentence s_i. The idea is that, if an incorrect parse tree is getting returned as the correct tree for a sentence, then we'd like that tree to have more impact in the training process (in order to fix the incorrect decision).

Equivalence of the two Forms It turns out the two scoring functions $F(\mathbf{x})$ and $G(\mathbf{x})$ are equivalent. (Exercise left for the reader.) So comparing a parse tree feature vector with all the other examples in the training data, and calculating a dot product between the feature vector and (original) weight vector, lead to the same score. It may seem counter-intuitive that there are situations where the first comparison is more efficient than the second, but this is the case if the feature vector is exponentially large. Note that, in the duel form training, *the feature vector does not need to be explicitly represented, as long as the inner product between two parse tree vectors can be calculated.*

Complexity of the two Forms One pass over the data for the standard perceptron learning algorithm takes $O(Td)$ time, where T is the total number of parse trees in the training data (e.g. 1,000 trees for each sentence $\times$ 40,000 sentences), and d is the size of the feature vector (since each dot product calculation between feature and weight vector takes $O(d)$ time).

One pass over the data for the duel form perceptron takes $O(Tnk)$ time, where n is the number of sentences: for each sentence, each of the 1,000 trees needs comparing with all T trees (in order to score each tree). Hence if each tree comparison (dot product between tree vectors) takes $O(k)$ time, then we have $O(n \times 1,000 \times T \times k) = O(Tnk)$.

Hence there can be efficiency gains with the dual form if nk, the number of sentences times the time taken to compute the similarity of objects (e.g. trees), is much smaller than d, the size of the feature vector.

Complexity of Inner Products Can nk ever be much smaller than d? Yes, if the feature vector representation is exponentially large, and if the dot product between such vectors can be calculated efficiently (i.e. without ever explicitly representing the feature vector).

Two examples of feature representations which are exponentially large are representations which a) track all sub-sequences in a sequence (since the number of sub-sequences grows exponentially with the length of the sequence); and b) representations which track all sub-trees in a tree (since the number of sub-trees grows exponentially with the size of the tree).

Tree Kernels A *tree kernel* is a function which returns the number of subtrees in common between two trees. The feature vector representation is one where the basis elements of each vector correspond to a subtree, and the corresponding value is the number of times the subtree appears in the whole tree.

Computation of Subtree Kernel The number of subtrees in common between trees $\mathcal{T}_1$ and $\mathcal{T}_2$ can be calculated efficiently, without ever explicitly representing all subtrees. The algorithm on the slide is a DP algorithm which runs over all pairs of nodes (n_1, n_2) from $\mathcal{T}_1$ and $\mathcal{T}_2$, recursively counting the number of subtrees in common where the subtrees are rooted at n_1 and n_2, $f(n_1, n_2)$.

Two cases are straightforward. If the context-free productions with parents n_1 and n_2 are different, then the number of subtrees in common rooted at n_1 and n_2 has to be zero. For the base case, i.e. when n_1 and n_2 are pre-terminals, $f(n_1, n_2) = 1$ if the productions rooted at n_1 and n_2 are the same.

For the recursive case, if the productions rooted at n_1 and n_2 are the same, then we get a count for the production itself, which combines with a recursive call on each of the child pairs from n_1 and n_2. (Exercise for the reader: run through some examples to convince yourself that the recursive case is correct.)

Readings for Today's Lecture

- Michael Collins and Nigel Duffy. New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. ACL 2002.

References

[1] Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of EMNLP*, pages 1–8, Philadelphia, USA, 2002.

[2] Yue Zhang and Stephen Clark. Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics*, 37(1):105–151, 2011.

Machine Learning for Language Processing
Lecture 5: Topic Modelling and LDA

Stephen Clark

October 20, 2015

Probabilistic Topic Modelling Probabilistic topic modelling provides a suite of techniques for automatically finding structure in documents. Latent Dirichlet Allocation (LDA) is one prominent example of an approach to topic modelling, based on Bayesian probabilistic modelling. LDA attempts to find themes, or topics, in a set of documents, with the idea that these themes can be used for some downstream searching or browsing task. For example, a scientific search engine could organise its search results by topic (e.g. *genetics, evolution, disease, computation*), rather than simply provide a ranked list.

Crucially, the set of topics is not fixed in advance, nor does LDA rely on any human manual annotation. This is attractive because LDA can be applied to any document set, and the user does not require any knowledge in advance of what topics might be present; nor does the user have to perform any expensive annotation. Essentially we can think of LDA as performing clustering: grouping documents (probabilistically) into clusters, each document cluster corresponding to a topic; and grouping words (probabilistically) into clusters, each word cluster defining a topic.

One downside of LDA is that the topics are not automatically provided with labels – they're simply clusters. Hence how to label, or interpret, clusters after they have been produced is an important research problem.

Key Assumptions behind LDA LDA makes the following modelling assumptions. First, each document is a *mixture* of topics. However, it is typically assumed that each document contains only a few topics. (The latter assumption is controlled by one of the model's parameters.) LDA is a probabilistic model with a corresponding *generative process*, and each document is assumed to be created, or generated, by that process. The structure of the process, defined by a graphical model which encodes some random variables and independence assumptions, is given in advance; hence the inference problem, given a set of documents, is to infer the parameters which determine the various probabilities associated with the graphical model.

A *topic* is defined as a probability distribution over a fixed vocabulary of words. Note that each topic is a distribution over the same vocabulary, so it's not strictly correct to say that LDA groups words into topics (at least hard topics); however, the words which appear "at the top" of the distribution with high probability will be different for each topic. Hence LDA does perform a probabilistic, soft clustering of the vocabulary.

One feature of LDA which is important to grasp is that it's a Bayesian modelling framework, in that the probability distributions themselves are also generated as part of the generative process. Topics are generated first, before the documents. Hence we require distributions over distributions, which is where the Dirichlet distribution comes in. The Dirichlet distribution allows us to encode a prior expectation that each topic will be dominated by a relatively small number of words (in terms of the probability mass assigned to those words); and also that each document will be made up of only a few topics.

The only parameters that require specifying in advance are the number of topics, and the parameters of the Dirichlet distributions (which encode how sparse we'd like the topic and document distributions to be).

Example Topics The clusters on the slide are the 15 most probable words taken from a set of topics (with the corresponding probabilities not shown). So here we've effectively created a hard clustering by applying a cut-off after the top 15 words in each topic. In this example, taken from David Blei, the topics have been obtained from a set of scientific documents. The topics are relatively coherent and meaningful. It is interesting to consider what label you might assign to each topic to best describe it.

Documents and Topics The example on the slide is designed to show how a document can be thought of as a small set of topics, with each word in the document assigned to a single topic. The words in yellow appear to be related to genes; the words in purple to life and evolution; and the words in blue to computational analysis. Other topics may be less meaningful, or contain words with less semantic content (such as function words). How to subsequently use, or interpret, the topics is a separate research question.

The next slide, again from David Blei, shows the generative process applied to a single document.

The Generative Process The first task, before any documents are generated, is to generate the topic distributions (topics shown to the left on the pictorial slide, but not in the pseudocode); how this is done will be explained later. The first task when generating a document is to generate a distribution over topics. Intuitively, we want to say something like: 45% of this document will be made up of the "gene" topic, 20% of the "life" topic, and so on. Now each word is generated. This is done by first choosing a topic (according to the distribution just created), and then choosing a word from that topic (which is a distribution over words, remember).

Note that, as a statistical model of text, the generative process is extremely simple: it's a bag-of-words model where each word is generated independently of every other word. Other than a "set-of-words" model, the bag-of-words model is as simple as it gets when it comes to modelling text (and from a linguistic perspective totally stupid).[1] However, decades of work in NLP, and IR in particular, has shown that it's possible to build effective NLP applications using this model.

The (Formal) Generative Process The joint distribution at the bottom of the slide is the probability of a set of documents, from 1 to D, together with the associated probability distributions. First choose the topics, β_i, from 1 to K, each one independently; these will be chosen according to a Dirichlet distribution, with parameter η. For each document d, choose the topic proportions, θ_d, again independently and again according to a Dirichlet distribution (with parameter α). Then, for each word position n in document d, independently choose a topic $z_{d,n}$. Finally, choose a word $w_{d,n}$ for that position, given the topic (distribution), where each word is conditionally independent given the topic.

LDA as a Graphical Model The *plate diagram* on the slide is a diagrammatic notation for representing graphical models. The arrows represent conditional independence, as in standard graphical model notation, and the boxes represent *repetition*. For example, the box with the D in the bottom corner means that the θ_d distribution has to be generated D times, once for each document; and the box with the N in the bottom corner means that the θ_d distribution has to be sampled from N times, once for each word position (to give the $z_{d,n}$ distibutions). Note that the N box is within the D box, meaning that N seperate draws have to be made for each d. Finally, the $z_{d,n}$ distribution (topic) has to be sampled from N times to generate each word, $w_{d,n}$.

There is a crucial difference between the unshaded white nodes and the nodes shaded grey. The grey nodes are *observed* in the data; i.e. their values are known. The white nodes are unobserved, and their values have to be inferred. The plate diagram neatly expresses the significant amount of information we're attempting to infer, given only the words in the documents.

The Dirichlet Distribution The key point to remember about the Dirichlet distribution is that it's a *distribution over distributions*. Hence the "observations" $\boldsymbol{x}$ on the slide are vectors of values, with the values x_i summing to 1. The parameter $\boldsymbol{\alpha}$ is also a vector, with one parameter α_i for each x_i.

[1]There are many extensions to the basic LDA model, for example one which takes some word order into account.

The formula for the Dirichlet distribution is provided for completeness on the slide, with the comment that the Dirichlet is *conjugate prior* to the multinomial (and categorical) distribution. If the binomial distribution provides the probability of the number of heads given a series of coin tosses, the multinomial is the extension of this to the many-sided die case. The idea of the conjugate prior is that the form of the posterior distribution (after we've seen the data) is the same as the form of the prior, which greatly simplifies the problem of statistical inference.

The key intuition underlying the use of the Dirichlet distribution is provided by the picture. Here we have "observations" which are probability distributions over 3 values, x, y and z. The plots show how likely a particular distribution is, given the $\boldsymbol{\alpha}$ values shown. So with $\boldsymbol{\alpha} = (6, 2, 2)$, for example, the chances of getting a distribution with most of the mass concentrated on x is high. In practice for topic modelling, it is typical to use the same value for all the α_is, so that there is a single α parameter. The effect of the α parameter is to determine how sparse the resulting distribution is likely to be. For an α value less than 1, there is pressure to choose distributions over topics which favour just a few of the topics.

Parameter Estimation Our main concern is to estimate the topic distributions β_k, i.e. the probability of each word given the topic. We are also concerned with the distributions $\theta_{d,k}$, i.e. the probability of each topic for each document in the collection. Attempts have been made to estimate these distributions directly, e.g. using EM, but a more common, and successful, approach has been to get at these distributions indirectly, via the probability of each word *token* in the collection belonging to a particular topic. Technically this is achieved by marginalising out, i.e. summing over, the distributions β and θ. Here we'll gloss over this aspect of the mathematics, and proceed directly to the estimates provided by *Gibbs sampling.*

Estimates using Gibbs Sampling The application of Gibbs sampling to a collection of documents is straightforward [1]. For each word token in turn, estimate the probability of that word token being assigned each topic, keeping fixed the topic assignments for all the other word tokens. Then using this distribution over topics, sample a topic, and assign it to the word token.

The Gibbs sampling algorithm stores count matrices C^{WT} and C^{DT}. C^{WT} has the counts for each word *type* and topic, and C^{DT} has the counts for each document and topic. Initially each word token is assigned a random topic, and the count matrices are calculated. Then, each word token is considered in turn. First, the count matrices are decremented by one for the entries that correspond to the current topic assignment to the token under consideration (since we want to calculate probabilities based on all the *other* topic-token assignments). Then the probability of this token being assigned each topic is given by the formula on the slide, which is used to generate a sample and the new assignment is made.

The two relative frequencies on the slide have intuitive interpretations. The one on the left measures how often w_i has been assigned topic j across all documents; and the one on the right measures how often topic j has been assigned to words in document d_i. Increasing either of these counts will increase the chance that the ith token will be assigned topic j. Notice also the effect of η and α, which are the parameters of the two Dirichlet distributions. Here they can be seen as acting as smoothing parameters in the relative frequency estimates (much like add-one smoothing).

Finally, note that the formula given on the slides is an unnormalised probability estimate. These estimates need to be divided by a normalising constant, which is a sum of the unnormalised values across all topics.

The Final Estimates Once the Gibbs sampler has been run for a number of iterations, the final sample can be used for calculating the final count matrices, which can then be used to calculate the (smoothed) relative frequency estimates on the slide.

Why does topic modelling work? Behind all the fancy mathematics lies a simple intuition: *words which tend to appear in the same documents get clustered in the same topics.*

Readings for Today's Lecture David Blei's topic modelling website has a host of useful material: http://www.cs.columbia.edu/∼blei/topicmodeling.html. A number of pictures on today's slides were stolen from David's 2012 ICML tutorial. A good place to start is his general introduction to topic modeling.

References

[1] Mark Steyvers and Tom Griffiths. Probabilistic topic models. In T. Landauer, D. Mcnamara, S. Dennis, and W. Kintsch, editors, *Latent Semantic Analysis: A Road to Meaning.* Laurence Erlbaum, 2006.

Machine Learning for Language Processing
Lecture 6: Vector Space Models of Semantics

Stephen Clark

November 5, 2015

Much of the content in these notes is taken from my book chapter on Vector Space Models of Lexical Meaning. (See Readings for Today's Lecture.)

VSMs in Document Retrieval The document retrieval problem in Information Retrieval (IR) [3] is as follows: given a query — typically represented as a set of query terms — return a ranked list of documents from some set of documents, ordered by relevance to the query. Terms here can be words or lemmas, or multi-word units, depending on the lexical pre-processing being used.

One of the features of most solutions to the document retrieval problem, and indeed Information Retrieval problems in general, is the lack of sophistication of the linguistic modelling employed: both the query and the documents are considered to be "bags of words", i.e. multi-sets in which the frequency of words is accounted for, but the order of words is not. However, this simplifying assumption has worked surprisingly well, and attempts to exploit linguistic structure beyond the word level have not usually improved performance. For the document retrieval problem perhaps this is not too surprising, since queries, particularly on the web, tend to be short (a few words), and so describing the problem as one of simple word matching between query and document is arguably appropriate.

Once the task of document retrieval is described as one of word overlap between query and document, then a vector space model is a natural approach: the basis vectors of the space are words, and both queries and documents are vectors in that space. The coefficient of a document vector for a particular basis vector, in the simplest case, is just the number of times that the word corresponding to the basis appears in the document. Queries are represented in the same way, essentially treating a query as a "pseudo-document".

Term-Frequency Model The figure on the slide gives a simple example containing two short documents. In this example the user is interested in finding documents describing the England cricketer, Matthew Hoggard, taking wickets against Australia, and so creates the query { *Hoggard, Australia, wickets* }. Here the query is simply the set of these three words. The vectors are formed by assuming the basis vectors given in the term vocabulary list at the top of the figure (in that order); so the coefficient for the basis vector *Hoggard*, for example, for the document vector $\overrightarrow{d1}$, is 2 (since *Hoggard* occurs twice in the corresponding document).

The function for calculating the similarity between query and document is the dot product between the corresponding vectors, which is essentially measuring term overlap between the two.

TF-IDF Model The point of the example is to demonstrate a weakness with using just term-frequency as the vector coefficients: all basis vectors count equally when calculating similarity. In this example, document $d2$ matches the query as well as $d1$, even though $d2$ does not mention Hoggard at all. The solution to this problem is to recognise that some words are more indicative of the meaning of a document (or query) than others. An extreme case is the set of function words: we would not want a query and document to be deemed similar simply because both contain instances of the word "the".

Continuing with the example, let us assume that the document set being searched contains documents describing cricket matches. Since *wicket* is likely to be contained in many such documents, let us assume that this term occurs in 100 documents in total. *Hoggard* is more specific in that it describes a particular England cricketer, so suppose this term occurs in only 5 documents. We would like to down-weight the basis vector for *wicket*, relative to *Hoggard*, since *wicket* is a less discriminating term than *Hoggard*. An obvious way to achieve this is to divide the term-frequency coefficient by the corresponding document frequency (the number of documents in which the term occurs), or equivalently multiply the term frequency by the *inverse document frequency* (IDF). The figure on the slide shows the simple example with IDF applied (assuming that the document frequency for *Australia* is 10 and *collapse* is 3). Document $d1$ is now a better match for the query than $d2$.

Finally, there is one more standard extension to the basic model, needed to counter the fact that the dot product will favour longer documents, since these are likely to have larger word frequencies and hence a greater numerical overlap with the query. The extension is to normalise the document vectors (and the query) by length, which results in the cosine of the angle between the two vectors.

Term-Document Matrix One useful way to think about the document vectors is in terms of a *term-document* matrix. This matrix is formed by treating each term or word vector as a row in the matrix, and each document vector as a column. The figure on the slide shows the term-document matrix for our simple running example.

The main reason for introducing the term-document matrix is that it provides an alternative perspective on the co-occurrence data, which will lead to vectors for terms themselves. But before considering this perspective, it is important to mention the potential application of *dimensionality reduction* techniques to the matrix, such as singular value decomposition (SVD). The use of SVD, or alternative dimensionality reduction techniques such as non-negative matrix factorisation or random indexing, will not be described here. However, it is important to mention these techniques since they are an important part of the linear algebra toolbox which should be considered by any researchers working in this area. Chapter 18 of [3] provides an excellent textbook treatment of matrix decompositions in the context of Information Retrieval.

The second term-document matrix example (with term frequencies as the values) has more documents and provides a better example for what follows. The first insight is that documents can be clustered based on the terms they contain. So in the example, the c's cluster together, because they tend to contain the same terms; likewise for the m's. But now an alternative suggests itself: rather than looking at the similarity of the columns, consider the similarity of the rows. So we can now say that terms are similar if they tend to occur in the same documents. How similar? The cosine measure can be applied here as well.

A Finer Notion of Context The term-document matrix introduced above gives us the basic structure for determining word similarity. There the intuition was that words or terms are similar if they tend to occur in the same documents. However, this is a very broad notion of word similarity, producing what we might call topical similarity, based on a coarse notion of context. The trick in arriving at a more refined notion of similarity is to think of the term-document matrix as a term-*context* matrix, where, in the IR case, context was thought of as a whole document. But we can narrow the context down to a sentence, or perhaps even a few words either side of the target word.

Once the context has been shortened in this way, then a different perspective is needed on the notion of context. Documents are large enough to consider which words appear in the same documents, but once we reduce the context to a sentence, or only a few words, then similar words will tend not to appear in the same *instance* of a contextual window. The new perspective is to consider single words as contexts, and count the number of times that a context word occurs in the context of the target word.

The example on the slide demonstrates the construction of a *word-word* (or *term-term*) co-occurrence matrix, using a single sentence as the context window. Note that the target words — the words for which context vectors are calculated — do not have to be part of the term vocabulary, which provide the context. Determining which words are similar can be performed using the cosine measure, as before.

Alternative Definitions of Context The previous example uses what is often called a *window* method, where the contextual words for a partiular instance are taken from a sequence of words containing the target word. In the example, the window boundaries are provided by each sentence. When the window is as large as a sentence — or a paragraph or document — the relation that is extracted tends to be one of topical similarity, for example relating *gasoline* and *car*. If the intention is to extract a more fine-grained relation, such as synonymy, then a smaller, and more fine-grained, notion of context is appropriate.

More fine-grained defintions are possible even for the window method. One possibility is to pair a context word with a direction. Now the vector coefficients are weighted counts of the number of times the context word appears to the left, or right, of the target word, respectively. A further possibility is to take position into account, so that a basis vector corresponds to a context word appearing a particular number of words to the left or right of the target word. Whether such modifications improve the quality of the extracted relations is not always clear, and depends on the lexical relations that one hopes to extract.

The next step in refining the context definition is to introduce some linguistic processing. One obvious extension to the window methods is to add part-of-speech tags to the context words. A more sophisticated technique is to only consider context words that are related syntactically to the target word, and to use the syntactic relation as part of the definition of the basis vector. The figure on the slide shows how these various refinements pick out different elements of the target word's linguistic environment. The pipe or bar notation (|) is simply to create pairs, or tuples — for example pairing a word with its part-of-speech tag. The term *contextual element* is used to refer to a basis vector term which is present in the context of a particular instance of the target word.

The intuition for building the word vectors remains the same, but now the basis vectors are more complex. For example, in the grammatical relations case, counts are required for the number of times that *goal*, say, occurs as the direct object of the verb *scored*; and in an adjective modifier relation with *first*; and so on for all word-grammatical relation pairs chosen to constitute the basis vectors. The idea is that these more informative linguistic relations will be more indicative of the meaning of the target word.

Weighting So far we have been assuming that all contextual elements are equally useful as basis vectors. However, this is clearly not the case: the word *the* provides very little information regarding the meaning of the word *goal*. Here we can adopt the idea of weighting from document retrieval. One simple method for weighting a basis vector is to divide the corresponding term frequency by the number of times that the term occurs in a large corpus, or the number of documents that the term occurs in (similar to IDF).

The figure on the slide demonstrates the effect of using IDF in this way for the extreme case of *the* as a basis vector. The effect is a little hard to capture in two dimensions, but the idea is that, in the vector space to the left of the figure, the vectors for *dog* and *cat* will be pointing much further out of the page — along the *the* basis — than in the vector space on the right. A useful intuition for the effect of IDF is that it effectively "shrinks" those basis vectors corresponding to highly frequent terms, reducing the impact of such bases on the position of the word vectors, and thereby reducing the amount of overlap on those bases when calculating word similarity.

One feature of IDF is that the shrinking effect applies in the same way to all target words (since IDF for a basis vector is independent of any target word). However, we may want to weight basis vectors differently for different target words. For example, the term *wear* may be highly indicative of the meaning of *jacket*, but less indicative of the meaning of *car*. Hence we would want to emphasise the basis vector for *wear* when building the vector for *jacket*, but emphasise other basis vectors — such as *gasoline* — when building the vector for *car*. Curran [1] uses collocation statistics, such as pointwise mutual information, to allow the weighting scheme to have this dependence on the target word. For example, *jacket* and *wear* will be highly correlated according to a collocation statistic because *jacket* co-occurs frequently with *wear* (relative to other basis vector terms).

Evaluation How to evaluate semantic vector space models is a hot topic at the moment. The standard approach is to use a set of human similarity judgements, and calculate a spearman correlation coeffient between the rankings of pairs induced by the human judgements, and those induced by the automatic similarity metric. However, most researchers agree this method is inadequate, not least because the notion of similarity, outside of any context, is not very meaningful. (This is especially true when moving from single words to phrases or even sentences.) The example judgements on the slide are from the frequently used wordsim 353 dataset.

Another problem is that semantic similarity can be interpreted in a variety of ways. Hence a number of researchers have created alternative datasets, being careful to distinguish semantic similarity — e.g. (*car, van*) — and semantic relatedness — e.g. (*car, petrol*). Examples include Simlex-999 and MEN.

One final comment is that the models introduced have a large number of parameters (window size, weighting scheme, and so on). How best to set these paramaters, and whether some settings are better than others for various semantic relations or evaluations, is another hot topic. For a recent paper attempting to investigate this question, see [2].

Readings for Today's Lecture A preprint of my book chapter is available online.

- Vector Space Models of Lexical Meaning. Stephen Clark. *Handbook of*

Contemporary Semantics second edition, edited by Shalom Lappin and Chris Fox. Chapter 16, pp.493-522. Wiley-Blackwell, 2015

- From frequency to meaning: Vector space models of semantics. Peter D. Turney and Patrick Pantel. *Journal of Articial Intelligence Research* 37:141-188. 2010

References

[1] James R. Curran. *From Distributional to Semantic Similarity*. PhD thesis, University of Edinburgh, 2004.

[2] Douwe Kiela and Stephen Clark. A systematic study of semantic vector space model parameters. In *Proceedings of the 2nd Workshop on Continuous Vector Space Models and their Compositionality (CVSC) at EACL 2014*, pages 21–30, 2014.

[3] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

Machine Learning for Language Processing
Lecture 7: Word Embeddings

Stephen Clark

November 17, 2015

Neural Distributional Models An alternative to deriving semantic word vectors through counting methods, as described in the previous lecture, is to effectively encode the distributional hypothesis as part of a probabilistic model, and then learn the parameters of that model using standard ML techniques. One way to achieve this is to set up the learning of word vectors as a language modelling problem.

For the continuous bag of words (CBOW) model, shown on the slide, the word being predicted by the language model is the target word, i.e. the word for which we are trying to learn a vector. CBOW predicts the target word by first adding the vectors of the context words, to give a single vector for the context, and then predicting the target on the basis of that context vector.

An alternative to CBOW, and the model we will focus on, is the skip-gram model. Here, instead of the context predicting the target, we'll use the target to predict the context. The picture makes it appear as though we're using a neural network to make the prediction, and in some respects we are, but the network architecture is trivial (and certainly not one that could be described as "deep", so don't think of this as an application of "deep learning").

The "projection" layer is a standard way of representing an input vector in a neural network. Think of the input as a *one-hot* vector, $w(t)$, where the vector is the size of the target word vocabulary, and each value is zero except for the basis vector corresponding to the target word, which has the value one. Then between the one-hot vector and the projection layer is a matrix, where each column of the matrix corresponds to a word vector. So multiplying the one-hot vector by the matrix gives the vector for the target word, which is the projection layer in the picture.

Once we have the projection layer, this is used to predict each of the context words. In the skip-gram model, the context is a fixed-word window either side of the target word, in this case a 2-word window either side. It is important to note that there are two types of vector here: vectors for the target word and vectors for the context words. So assuming that the target word and context word vocabularies are the same, each word has two separate vectors.[1]

Skip-Gram Language Modelling The expression on the slide is the quantity we'd like to maximise when estimating the vectors, where θ is the set of parameters that we're trying to learn, i.e. the values of the target and context vectors. *Text* is the set of instances containing each target word (token), and $C(w)$ is the set of context words for target word instance w. For the model on the previous slide, $C(w)$ contains four context words – the two words either side of the target word. Note that each context word is generated independently of the other context words in the window. Note also that each context word (token) will get generated a number of times: once for each target word whose context it appears in. Given these assumptions, the data D can be represented as a set of target, context word pairs, and the goal of the training process is to maximise the (conditional) probability of these pairs.

Parameterisation of Skip-Gram So far there has been no discussion of how a target vector and context vector can be combined to yield a probability. A natural way of predicting a context word (vector) given a target word (vector) is using the softmax probability function on the slide, where the denominator is a normalising constant so that the probabilities for all context words sum to 1 (given a target word). Note that the probability of a context word given a target word will be high if the dot product between the respective vectors is high.

During estimation, since the goal is to maximise the probability of the data, the target word vectors will be forced to be similar to the context vectors for words appearing in the contexts of the target word. Or to put it another way: words will have similar target vector representations if they tend to appear in the same contexts. So we've arrived at the distributional hypothesis via a language modelling objective.

How is such a model estimated in practice? We won't consider the optimisation problem in any detail, except to say that the neural networks community has recently got very good at estimating these sorts of models, using an optimisation technique called stochastic gradient descent (SGD). The problem with the current objective is that calculating the normalisation constant for each context word is expensive, since it requires a sum over the context word vocabulary, which could be very large. The following slides present an alternative objective which is much cheaper to calculate, but extremely effective in practice.

Negative Sampling The alternative objective arises from conceiving of the language modelling problem in a different way: rather than predicting a context

[1]As motivation for this choice, consider what happens when a word appears in its own context.

word given a target word, the goal will be to decide whether a particular context, target word pair is from the data, or has been artificially generated as a negative example. The advantage of this formulation is that we now have a binary classification task, and so the normalisation constant effectively involves a sum over only two values, rather than the whole vocabulary.

The derivation on the slide is for the new training objective. Note that D is being overloaded here: it serves as the set of positive training examples (as before), and also as a random variable indicating whether a word, context pair is positive or negative. D' denotes the set of negative training examples (how this set is obtained will be explained later).

The question arises again of how to obtain a probability from a word and context vector, this time the probability for the random variable D. A similar solution is adopted, using the sigmoid function as a natural way to encode the relevant probability. Note that, again, the probability $p(D = 1|c, w)$ will be high if the respective target and context word vectors are similar; likewise $p(D = 0|c, w)$ will be high if the target and context vectors are dissimilar.

Sampling Details The Goldberg and Levy paper points out that there are many details in the freely available implementation of the skip-gram model — word2vec — that are not explicitly mentioned in the accompanying papers; and crucially that many of these details can signifcantly affect performance.

The first question is how to create, or sample, the negative data. The solution is to produce a probability distribution over the context vocabulary, which will then be sampled from (k times) for each target, context word pair. The distribution used in word2vec is the unigram distribution, but where each value is raised to the power 0.75 (and then normalised overall). I assume that the 0.75 value is designed to downweight the highly frequent words, so that these do not overly dominate the negative examples.

How best to set all these various parameters, including k, is an interesting question. Currently the approach is to try various values and observe the effect on whatever evaluation is being used to test the models.

Another interesting feature of word2vec is that the window size is *dynamic*, i.e. the window size varies for each target word instance. The window size is chosen according to a uniform distribution between 1 and N. What this means in practice is that words closer to a target word are more likely to appear as contexts in the positive data D than words further away.

Finally, words whose overall frequency is less than some threshold M are discarded, and this discarding occurs before the sets D and D' are created, including the selection of the context windows. What this means in practice is that the window size is effectively increased, since many of the words that would have appeared in the context window have now been removed from the original data.

Linguistic Regularities? One of the claims for the word2vec software is that it induces semantic spaces where various linguistic regularities are encoded as part of the linear structure [3]. Some researchers have found these claims very exciting, for example the idea that a vector for *queen* can be obtained by taking the vector for *king*, taking out the *woman* vector (by subtraction), and adding back in the *man* vector. Similar results have been claimed for capital cities, e.g. that $\overrightarrow{\text{PARIS}} = \overrightarrow{\text{LONDON}} - \overrightarrow{\text{ENGLAND}} + \overrightarrow{\text{FRANCE}}$. Whether this linear structure really holds for a range of interesting semantic relations still requires careful empirical study, but the suggestion is undoubtedly intriguing.

Evaluation One paper reporting an extensive evaluation of various vector space models, in particular comparing the "count" models from the previous lecture, and the "predict" models from this lecture, is Baroni et al. [1]. The slides show the various evaluations carried out by Baroni et al., and give a good indication of the sorts of tasks currently being used to evaluate word vectors.

Results The headline result from the Baroni et al. paper is that the predict vectors performed extremely well across a range of tasks, despite the authors essentially using the word2vec vectors "out of the box". However, this conclusion needs tempering in the light of a subsequent paper by Levy and Goldberg [2] which showed that, by adapting some of the techniques described in the current lecture (e.g. a dynamic window), similarly competitive results can be obtained for the "count" models.

Readings for Today's Lecture

- **Efficient Estimation of Word Representations in Vector Space.** Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. In Proceedings of Workshop at ICLR, 2013.
- **word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method.** Yoav Goldberg and Omer Levy. arXiv:1402.3722

References

[1] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 238–247, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[2] Omer Levy and Yoav Goldberg. Neural word embeddings as implicit matrix factorization. In *NIPS*, 2014.

[3] Geoffrey Zweig Tomas Mikolov, Wen-tau Yih. Linguistic regularities in continuous space word representations. In *Proceedings of NAACL-HLT 2013*, pages 746–751, Atlanta, Georgia, 2013.