6.864 Advanced Natural Language Processing[1]

Lecture 1: Introduction

10 September 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Narasimhan, Tianheng Wang.
Scribes: Brandon Carter, Sarah Vente.

This first lecture introduces Natural Language Processing (NLP). It presents the core concepts of NLP, some of its main applications as well as challenges, a succinct history of the field, and machine learning approaches.

What is Natural Language Processing?

Language has no independent existence apart from the people who use it. It is not an end in itself; it is a means to an end of understanding who you are and what society is like.

- David Crystal

Natural Language Processing (NLP) is a sub-field of Artificial Intelligence focused on developing machines that understand human languages. NLP is quite a broad area covering tasks such as Machine Translation, Question Answering and Information Extraction. Recent advances in NLP have led to the development of ubiquitous applications used every day ranging from web-based apps like Google search to mobile personal assistants such as Siri. In spite of these successes, the field still remains a very active area of research with enormous potential, drawing ideas from computer science, linguistics, psychology, mathematics and information theory among others.

NLP is not just about string matching. Though some applications like spell checking and spam filtering can derive most of their benefits from variants of text matching, many tasks like summarization and dialogue generation require more sophisticated approaches. In fact, the notion of language understanding has often been associated with the idea of artificial intelligence. In 1950, Alan Turing wrote in his famous book *Computing Machinery and Intelligence*: "I believe that in about fifty years it will be possible, to program computers, with a storage capacity of about 10^9, to make them play the imitation game so well that an average interrogator will not have more than 70 per cent chance of making the right identification after five minutes of questioning."

This famous quote formed the basis for what is known as the Turing Test, taken to be an evaluation of a machine's ability to exhibit human-like intelligent behavior. Based on this, an annual competition known as the Loebner Prize is held where judges converse simultaneously with a human and a chatbot (with no prior knowledge of their

identities) and must decide which one is human. Current advances, however, have shown that it is possible to fool a human using simple techniques such as reformulating the user's input into conversation-continuing questions.

Applications of NLP

Recent years have seen rapid advancement in language technologies including the advent of large-scale web search, real-time machine translation, and even intelligent agents capable of understanding voice commands on a mobile phone. The impact of NLP research on the real world is very visible and will continue to be significant in the years to come.

To get a sense of where the field stands at present, the traditional NLP sub-tasks can be divided into three groups (the group names were taken from Dan Jurafsky's NLP course):

Mostly solved: Spam detection, part-of-speech tagging, named entity recognition. These tasks are relatively low-level and current systems are very good at performing them. Accuracies range from the upper 80 percents to the upper 90 percents.

Making good progress: Sentiment analysis, coreference resolution, word sense disambiguation, syntactic parsing, machine translation, information extraction. These are harder problems that don't have highly accurate systems but do have considerable performance.

Still really hard: Question-answering, paraphrase generation, summarization, dialogue management. These tasks are still in their infancy in terms of having systems that can robustly handle them.

Question-Answering

Let us consider the task of question answering. One example of a system performing this task is IBM's Watson, which can answer many trivia questions better than humans. For example, Watson is able to correctly answer a factual question such as:

William Wilkinson's "An account of the principalities of Wallachia and Moldavia" inspired this author's most famous novel.

Trivia questions like these are relatively straightforward to answer for Watson whose strengths lie in its capability to index and retrieve huge amounts of information. The inability to remember huge amounts of information remains an inherent weakness for humans in this task.

However, certain types of questions are difficult for computers to answer. For example, given a short story, computer programs still have trouble answering "basic" questions about the story. Questions that a kid in middle school would answer with ease. For example, consider the following snippet from a short story:

Sally liked going outside. She put on her shoes. She went outside to walk. Missy the cat meowed to Sally. Sally waved to Missy the cat.

Given the question *Why did Sally put on her shoes?*, current question-answering systems cannot reliably give the correct answer: *because she wanted to go outside*. This is because such a task invovles aspects of reasoning and understanding that is beyond current systems. State-of-the-art accuracy on such tasks is only around 70%, so there is still a lot of work to be done.

Challenges

The example in this section was adapted from (Lee, 2003). Consider this sentence from an old advertisement:

At last, a computer that understands you like your mother.

This sentence nicely illustrates the range of difficulties present in understanding natural language. Upon some consideration, we can come up with at least three different ways of interpreting this sentence:

1. The computer understands you as much as your mother understands you.
2. The computer understands that you like your mother.
3. The computer understands you as well as it understands your mother.

This is the notion of *ambiguity*, which is present in human languages. Although humans seem to be exceptionally good at discarding alternatives and homing in on a single meaning (usually the intended one), resolving ambiguities is a core technical challenge in NLP. We can roughly classify them into three categories.

Syntactic ambiguity: This type of ambiguity relates to the parsed structure obtained by processing a sentence. Given the above sentence, there can be two equally plausible syntactic parses. We can have a parse where the phrase *like your mother* is attached to *understands* (it understands you as well as your mother does). Alternatively, another parse could have *like* treated as a verb and hence *like your mother* would be just the verb phrase complement to the noun *you* (you like your mother).

Semantic ambiguity: Another type of ambiguity that we can find in this sentence is due to two conflicting definitions of mother: *a female parent* or *a slimy substance*. Though the second definition in this case might seem far-fetched, one can easily think of more common examples such as *I drank the orange lemonade* where *orange* could be the fruit or the color.

Discourse ambiguity: This type of ambiguity relates to elements of discourse in language. Consider the sentences

Alice says they've built a computer that understands you like your mother but she
... doesn't know any details.
... doesn't understand me at all.

Depending on which ending you see, the pronoun *she* could refer to either *Alice* or *your mother*. Again, resolving this is easy for humans but extremely difficult for automatic systems.

Ambiguity also varies across languages. For example, tokenization is easy in English since words are separated by spaces, but it is more challenging in Semitic languages such as Hebrew. Thus, developing NLP systems that work across different languages is an additional challenge.

A brief history

Any successful NLP system needs to have two components: *linguistic knowledge* and *knowledge about the world*. There are many approaches to solving this knowledge bottleneck problem, but we can broadly classify them into two categories:

1. Symbolic systems: The required knowledge is encoded into computers by human experts.
2. Statistical systems: Empirical methods are used to infer required knowledge from observed data.

The choice between these two approaches has often been the subject of debate in the field of NLP. Empirical methods were not in favor in the 1960s and 70s. One major factor for this is an argument by eminent linguist Noam Chomsky, who believed that statistical techniques would never be sufficient to gain a deep understanding on human language. This led to the dominance of *knowledge-based* approaches, requiring humans experts to encode knowledge into computers.

Emphasis was placed on deeper models of language, and on syntax, often using small focused domains. For instance, SHRDLU was

a system that could follow instructions to manipulate several different objects in a virtual world, as well as answer questions about them. However, SHRDLU and other similar systems required extensive manual injection of knowledge, and only worked in their specific domains. These systems had difficulty in scaling up to larger domains because it would require substantial human effort.

In the 1980s, an empirical revolution took place. The field of speech recognition in particular led the way with large-vocabulary real-time systems that were influenced by ideas from information theory. Inspired by this success, researchers began using probabilistic approaches to problems in natural language processing, leading to several breakthroughs in areas like search, information extraction and parsing. Nowadays, empirical methods are ubiquitous in all areas of NLP, often intertwined with developments in the field of Maching Learning. From Google Translate to curated news apps like Flipboard, statistical methods are key features in NLP components. For a more detailed overview of the history of NLP, see (Lee, 2003).

Applying Machine Learning Methods

The Determiner Placement Task

The determiner placement task is defined as follows: given a text with no determiner, add any missing determiner to the text. For example, the given text could be "*Scientists in United States have found way of turning lazy monkeys into workaholics using gene therapy.*". To simplify the task, assume that we only consider one sort of determiner, the definite article *the*. The decision is therefore binary, viz. whether *the* should precede a given noun in a given text.

For a native English speaker, performing this task is usually easy. However, there exist many grammar rules specifying determiner placement, based on linguistic knowledge such as the countability of the noun (i.e. whether it is countable or uncountable) or the plurality of the noun (i.e. whether it is singular or plural), as well as on world knowledge such as the uniqueness of reference (*the current president of the US*) or the situational associativity between nouns (*the score of the football game*). In addition to these generic grammar rules, there exist numerous exceptions: for instance, while the definite article is required in newspaper titles (e.g. *The Times*), it shall not be used for names of magazines and journals (e.g. *Time*).

Given the amount of rules and exceptions involved in determiner placement, it would be tremendously difficult to manually encode all of this information: a reliable symbolic system for the determiner placement task would subsequently be difficult to construct.

Determiner Placement as a Supervised Classification Task

As an alternative to symbolic systems, one could adopt a statistical approach to the determiner placement task by casting it as a supervised classification task.

A basic classifier could be trained as follows:

1. Gather a large collection of texts (e.g. a set of newspaper articles) and split it into a training set and a test set.
2. For each noun in the training set, compute the probability it follows a certain determiner, i.e. $p(\text{determiner}|\text{noun}) = \frac{\text{count(determiner, noun)}}{count(noun)}$.
3. Given a text from the test set where determiners have been removed, for each noun in the text, select the determiner with the highest probability as estimated in the previous step.

Unseen words cannot be classified

The quality of the predictions made by this classifier is not outstanding, but still surprisingly high for such a simple method: using the first 21 sections of the Wall Street Journal (WSJ) corpus as the training set, and the 23rd section as the test set, the prediction accuracy is 71.5%. This result can be used as a baseline for more elaborate classifiers.

The typical steps to train a classifier in a supervised fashion are as follows:

1. *Data collection*: Gather a large collection of texts, and split it into a training set and a test set. Because one can easily get a training set of sample sentences in English with correct determiner placement, no additional work is needed to obtain the labels: we assign a label -1 or $+1$ to each noun ($+1$ if the noun is preceded by a determiner, -1 if not).
2. *Feature extraction*: For each noun in the training and test sets, extract some features. E.g. whether the noun plural is plural (0 if no, 1 if yes), whether it is its first appearance in the text (0 if no, 1 if yes), and so on. We therefore obtain one feature vector for each noun. Figure 1 illustrates the process of feature extraction.
3. *Training*: Train a classifier using the extracted features as well as the labels. The training step is used to find the optimal feature weight vector to improve the final prediction. Figure 2 shows a decision boundary (in simplified two-dimensional space) based on the feature vectors and weight vectors.
4. *Testing*: Assess the classifier's prediction quality on the test set. Using this machine learning approach, one can reach an accuracy above 86%.

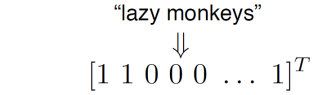


Figure 1: **Feature extraction** is the process of converting a text to a feature vector, before feeding it to a classifier. If there are n binary features, then the feature vector will be of dimension n.

Table 1 shows pairs of features and label that result from step 1 and 2. To train the classifier in step 3 using these features and labels, there exist many classification algorithms such as the perceptron algorithm. The perceptron algorithm iterates through the training examples and finds an optimal decision boundary to separate as many of the training points as possible: it places the decision boundary so that it optimizes the number of training points that are classified correctly, as shown in Figure 2.

Once the decision boundary is placed, one can plot any new vector into this space and determine on which side of the decision boundary it is located. A noun whose vector falls on the $+1$ side is assigned a $+1$ label, while a noun whose vector fall on the other side of the boundary is assigned a -1 label. This classification procedure therefore utilizes the training examples with known determiner placement to classify the unlabeled examples.

One of the most common features in NLP is word *n-grams*, or simply n-grams. A word n-gram is a sequence of n words. A unigram designates a 1-gram, a bigram is a 2-gram, and a trigram is a 3-gram. N-grams of larger sizes are less often used. An n-gram feature value is encoded as a *one-hot vector*, i.e. a vector in which all elements are 0 except one element that is 1. For example, $[0, 0, 0, 1, 0]$ is a one-hot vector. If the vocabulary is of size $|V|$, then there are $|V|$ possible unigrams, $|V|^2$ possible bigrams, and $|V|^3$ possible trigrams. This means that the unigram feature value will be a one-hot vector of dimension $|V|$, the bigram feature value will be a one-hot vector of dimension $|V|^2$, and the trigram feature value will be a one-hot vector of dimension $|V|^3$. Despite its simplicity, using n-gram as a feature often yields high performances: if the unigram feature is added to the classifier in the determiner placement task (i.e. the unigram feature is extracted from the noun, as shown in Figure 3), it significantly improves the accuracy of the classifier. However, n-gram features are very sparse, since $|V|$ is typically very large, often over 10^4.

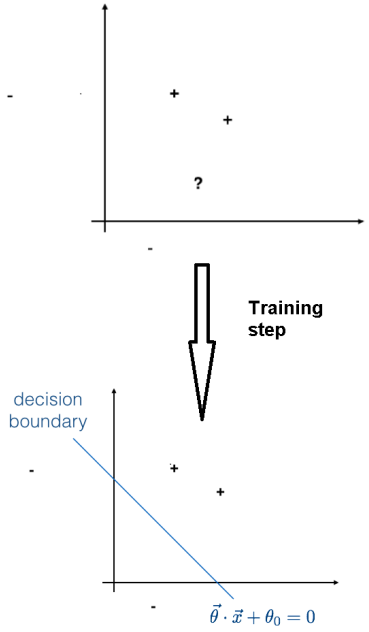


Figure 2: Learning the **decision boundary** based on the feature vectors and weight vectors. In this example (in simplified two-dimensional space), there are two positive points (with $+$ labels), and two negative points (with $-$ labels). The decision boundary correctly separates all of the training points.

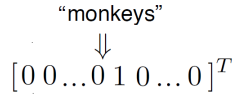


Figure 3: Extracting an unigram feature is the process of converting a word to a one-hot vector, which will be used as a feature.

	Features			
Noun	plural?	1st appearance?	unigram	label
Scientists	1	1	00100	null (-1)
United States	1	1	00010	the ($+1$)
lazy monkeys	1	1	00001	the ($+1$)
gene therapy	0	1	01000	null (-1)

Table 1: Example of feature values and labels given to the classifier during the training phase. This table results from data collection, labeling and feature extraction. The feature values for the unigram feature present in this table are much smaller than it is in reality: unigram feature values are one-hot vectors of dimension $|V|$, where V is the dictionary.

Limitations of Traditional Machine Learning Approaches

This traditional machine learning approach has several issues, the two main ones being the sparsity of the feature vectors and feature engineering:

- *Sparsity*: feature vectors are typically high-dimensional and sparse (i.e. most elements are 0). Figure 4 illustrates that sparsity often results in many feature vectors not being seen when the training set is small. Figure 5 shows the impact of sparsity on the accuracy of parsers for different languages.
- *Feature Engineering*: the second problem with using traditional machine learning approaches for classification is feature engineering. Early studies spent much time engineering features. Traditionally, features of compositional objects are manually-selected concatenations of atomic features, as Figure 6 illustrates. However, crafting these features by hand is difficult and time-consuming as they are combinatorial and the number of them is therefore extremely large.

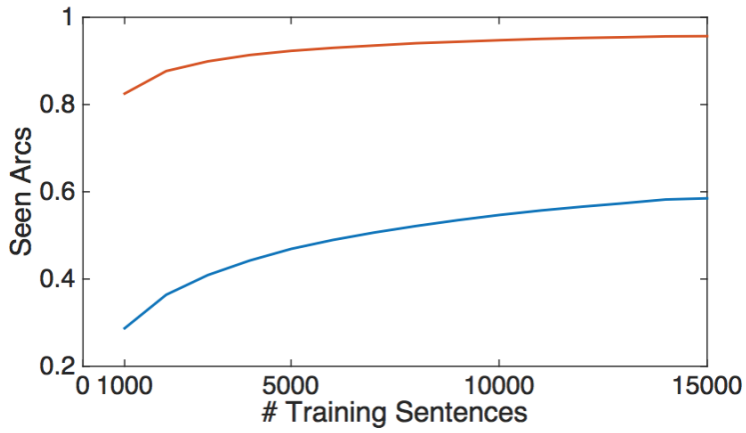


Figure 4: In this graph, the red line indicates the portion of unigrams (single words) in the unseen tests that has already been seen in training. The first 1000 training sentences contain over 80% of unigrams, and the first 10000 about 90% of them. The blue line in the graph indicates the portion of bigrams (pairs of words) in the unseen samples that has already been seen in the training samples: this number is a lot lower than that for unigrams. Thus, if we consider bigrams for classification, it becomes difficult to be able to accurately classify unseen examples when the classifier has not seen the majority of bigrams during the training phase.

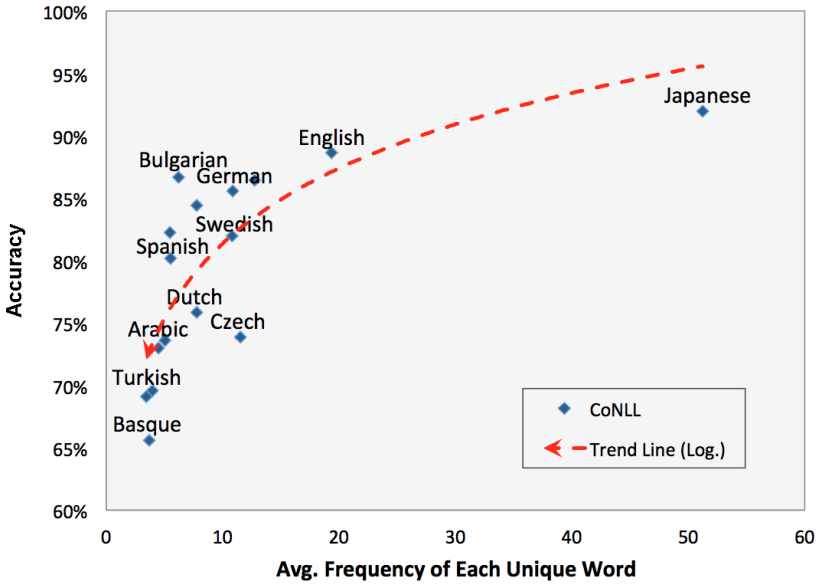


Figure 5: This graph shows the impact of sparsity on the accuracy of parsers for different languages. Japanese and English have the highest accuracies, and a good predictor of the parser accuracy is the average frequency of each unique word. This means that the parser has a much greater accuracy when it can be trained on larger subsets of the words that appear in the test set.

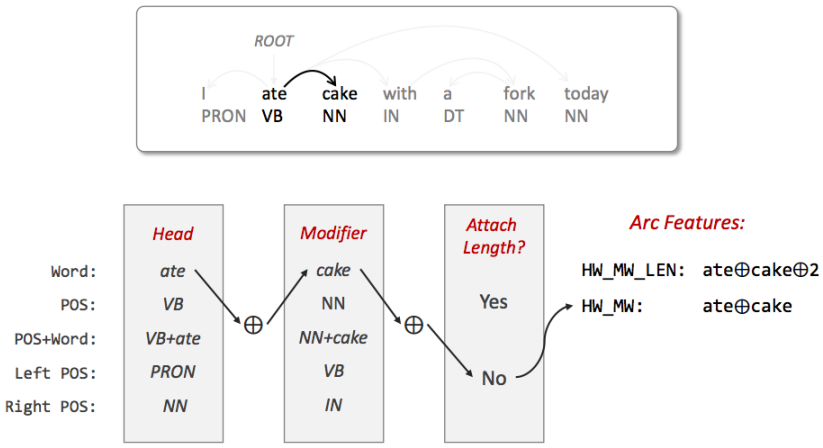


Figure 6: Selecting feature representation. Traditionally, features of compositional objects are manually-selected concatenations of atomic features.

Neural Models

Word embeddings are one solution to the sparsity problem. Embeddings are a set of machine learning techniques involving artificial neural networks that map discrete, one-hot vectors into low-dimensional continuous representations. They compress large sparse vectors into smaller, dense vectors, with semantically related words located close to each other in the word embedding space, as Figures 7 and 8 show. The word embedding space also contains interesting linear substructures, as Figure 9 illustrates: word embeddings are very powerful as they can capture some semantic relations between words.

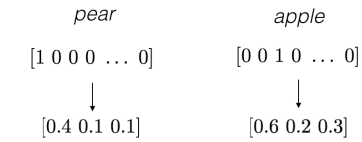


Figure 7: The words "pear" and "apple" are mapped from one-hot vectors to continuous, dense vectors of much lower dimension, where they are located close to each other.

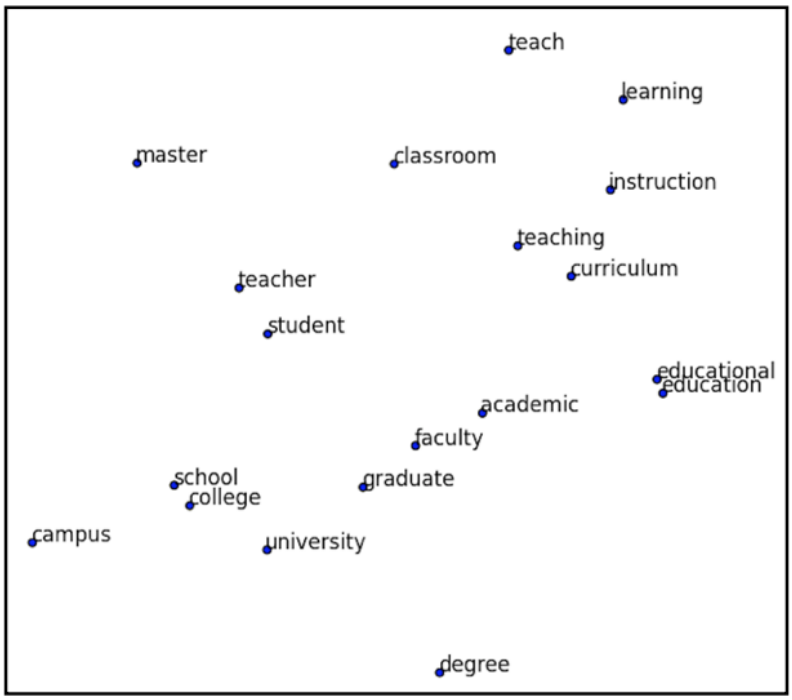


Figure 8: t-SNE visualization of word embeddings (t-SNE stands for t-Distributed Stochastic Neighbor Embedding). This plot illustrates a projection of embedded vectors into 2-D space. Semantically close words such as "school" and "college" are mapped closely to one another in the word embedding space, whereas they would appear unrelated when encoded as one-hot vectors. Source: `http://nlp.yvespeirsman.be`

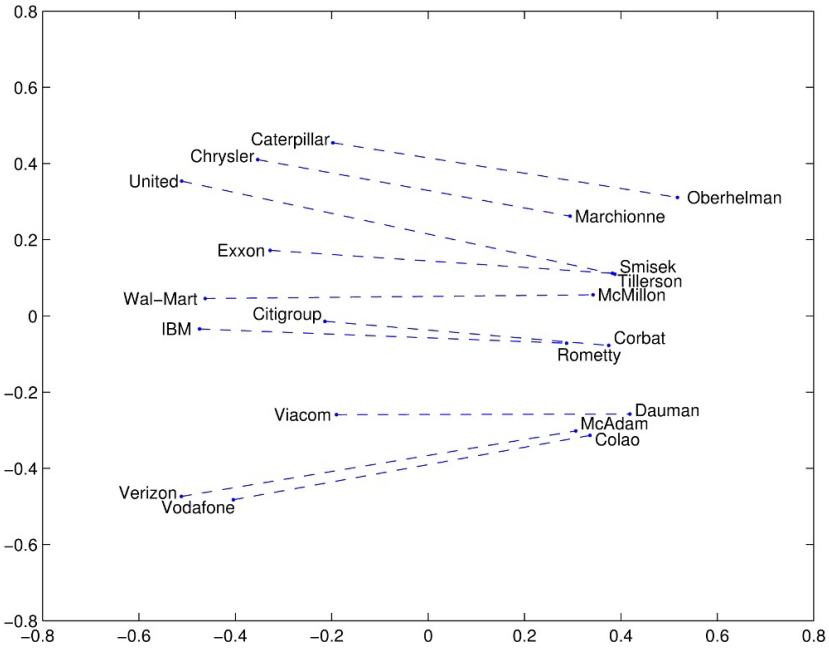


Figure 9: Linear substructures in word embeddings. In this example, if we take the embedding corresponding to the CEO of one company, subtract that company, and add another company, we will arrive at the CEO of the other company. Source: `http://nlp.stanford.edu/projects/glove`

Word embeddings are also one solution to the problem of feature engineering. Suppose we are trying to find a feature representation for the phrase "red apples." We could start with the word embedding vector for "red" and the word embedding vector for "apples". Then, we could add them, concatenate them, or combine them otherwise, as Figure 10 illustrates. Traditionally, we would just engineer features and use them for the main problem that requires these features. Using word embeddings, one can learn them jointly with the learning problem, for example by adding a hidden layer in the learning problem.

Neural networks are also used to learn compositionality, i.e. how to take two embeddings and create an aggregate representation for the phrase. A simple way to create this composition is to concatenate or average the vectors. Alternatively, one can learn vector representation for the phrase that optimizes performance on the task. This is achieved using neural networks, as shown in the Figure 10. The first layer takes as input the vector representation of the words, e.g., "red" and "apple", while the second layer represents a vector for the whole phrase (i.e. the phrase embedding). The model induces representation for this phrase while learning to perform some task such as document classification.

In general, one can tackle NLP tasks using traditional machine learning approaches, or newer techniques based on artificial neural networks. This course will cover both approaches.

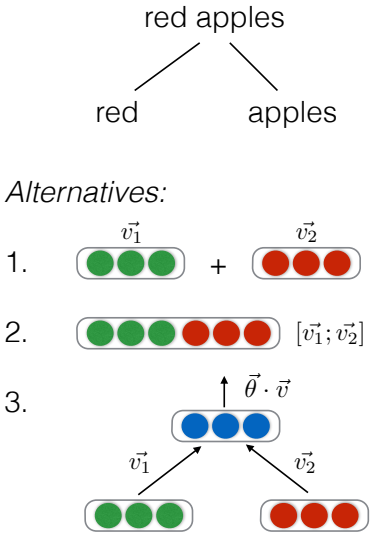


Figure 10: Three strategies amongst others to combine word embeddings.

References

Lee, L. (2003). "I'm sorry Dave, I'm afraid I can't do that": Linguistics, Statistics, and Natural Language Processing circa 2001. *arXiv preprint cs/0304027*.

6.864 Advanced Natural Language Processing[1]

Lecture 2: Language Modeling

15 September 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Narasimhan, Tianheng Wang.
Scribe: Nick Locascio

Statistical Language Model History

A statistical language model is a probability distribution over sequences of words. These probabilities should capture sentence plausibility, not necessarily its grammaticality. Clearly, plausibility will depend on the domain, genre, and other factors. Typically, language models are trained from a large corpus of raw sentences collected in the domain of interest. Since language models do not explicitly capture syntactic or semantic constructs, they faced some controversy at first. Noam Chomsky, in 1957, stated that statistical models were incapable of understanding sentences not seen before. As an example, he considered two sentences: "Colorless green ideas sleep furiously." and "Furiously sleep ideas green colorless." The first sentence is grammatical, the second is not. However, since neither of these sentences have ever appeared before, Chomsky argued that a statistical model would always fail to properly classify them. Eventually, this assertion was proven wrong. In 2000, Fernando Pereira demonstrated that a language model gives a significantly higher probability to the first sentence than to the second.

Of course, the real use of language models goes beyond evaluating fake sentences. Today, these models are widely used in NLP applications ranging from machine translation to speech recognition, and including spelling correction and summarization. In the next section we will illustrate their role in machine translation.

Noisy Channel Models

A machine translation system from French to English could be formulated as a noisy channel model, as illustrated in Figure 1.

The machine translation system will try to predict the English text given some French text that has the highest probability:

$$e* = \arg\max_{e} P(e|f) \tag{1}$$

However, modeling $P(e|f)$ directly is hard: it needs to capture both the grammaticality of the generated sentence, and account for the correctness of translation. From the estimation point of view, it will be easier if we can separate these decisions. For that reason, we apply

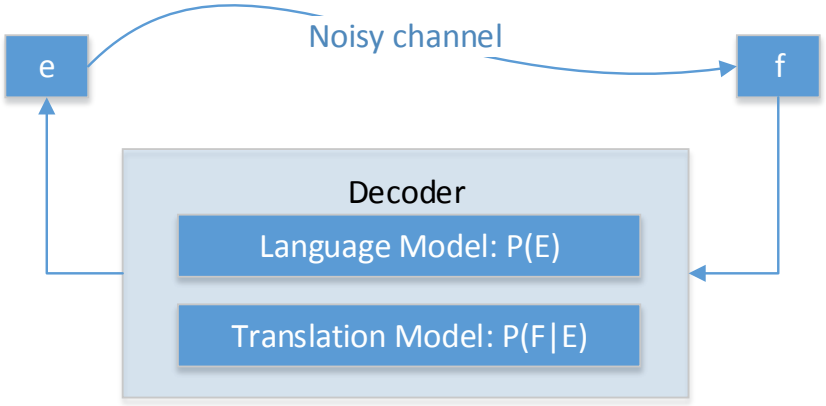


Figure 1: Machine Translation as a noisy channel model.

Bayes Rule, to inverse the probabilities:

$$e* = \arg\max_e \frac{P(e)P(f|e)}{P(f)} \tag{2}$$

Canceling out the denominator (since the max depends on e only, and the denominator does not contain e), we obtain:

$$e* = \arg\max_e P(e)P(f|e) \tag{3}$$

We will estimate $P(f|e)$ and $P(e)$ separately. The translation probability $P(f|e)$ is trained using parallel corpora (which consists of translation sentence pairs in French and English). We will talk about estimating translation probability later in the course. For now, we will focus on estimating language model $P(e)$. This model will be trained on English sentences, with no regard to French sources. We would like $P(e)$ to give low scores to ill-formed sentences that our translation model may like (e.g., sentences with broken word order, or wrong word selection as in "I went house").

Noisy-channel architecture (and subsequently language models) are used in a similar fashion in speech recognition, where we aim to predict the most likely transcription w for a given acoustic signal a.

$$w* = \arg\max_w P(w)P(a|w) \tag{4}$$

In this case, $P(w)$ captures the likelihood of the generated word sequence (viz. the transcription), and $p(a|w)$ models the translation between the transcription and the corresponding acoustic signal.

Building a Language Model

A **statistical language model** is a probability distribution over sequences of words. The model is trained using a corpus ($w_1, w_2, ..., w_{N-1}$, STOP), and a vocabulary V.

A Simple Model

Simple Model: count the number of occurrences of sentences in the corpus.

This model fails to generalize to novel, unseen sentences. It would certainly fail on Chomsky's example as well as any sentence that does not exactly appear in our training corpus. We need a model that can **generalize**.

Constructing a Model that can Generalize

The two keys elements to construct a language model that can generalize are:

1. **Decomposition**: the language model should break down the text into smaller components. N-gram models are a common decomposition technique.
2. **Smoothing**: the language model should gracefully deal with words that have never been seen before.

N-gram Models

N-grams models rely on the chain rule and the Markov assumption. The **chain rule** decomposes the probability of a sequence of words:

$$P(w_1, w_2, ..., w_{N-1}) = P(w_1)P(w_2|w_1)...P(w_N|w_1...w_{N-1}) \quad (5)$$

However, this formulation is impractical as we are conditioning on everything we've seen before in a sentence. To circumvent this issue, we make a **Markov assumption**: the probability of a given word only depends on the last k words. Common n-grams are formulated and compared below:

- **Unigram** ($k = 0$): $P(w_1, ..., w_N) = \prod_i P(w_i)$.
  Number of parameters: $O(|V|)$.
- **Bigram** ($k = 1$): $P(w_1, ..., w_N) = \prod_i P(w_i|w_{i-1})$,
  Number of parameters: $O(|V|^2)$.

- **Trigram** ($k = 2$): $P(w_1, ..., w_N) = \prod_i P(w_i|w_{i-2}, w_{i-1})$,
  Number of parameters: $O(|V|^3)$.

To apply these formulas in a real corpus, one needs to learn the probabilities $P(w_i)$, $P(w_i|w_{i-1})$ or $P(w_i|w_{i-2}, w_{i-1})$. One way to do so is to use the **Maximum Likelihood Estimate**.

In this context, the Maximum Likelihood Estimate simply means counting how many times a unigram, bigram or trigram occurred in the training set. Unigram probabilities are computed as follows (N is the number of words in the corpus):

$$P(w_i) = \frac{\text{count}(w_i)}{N} \tag{6}$$

Bigram probabilities are computed as follows:

$$P(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})} \tag{7}$$

For example, if the word w_i is "roses" and the word w_{i-1} is "pink":

$$P(\text{"}roses\text{"}|\text{"}pink\text{"}) = \frac{\text{count("pink", "roses")}}{\text{count}(\text{"}pink\text{"})} \tag{8}$$

Trigram probabilities are computed as follows:

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{\text{count}(w_{i-2}, w_{i-1}, w_i)}{\text{count}(w_{i-2}, w_{i-1})} \tag{9}$$

N-gram probabilities for larger n-gram sizes are computed in a similar fashion.

Smoothing

One obvious problem with the proposed estimation approach is that if an n-gram is not observed in the corpus, maximum likelihood estimate will assign zero to it, and thus the probability of our sentence will also be zero. How often do we encounter unseen ngrams? To answer this question, consider Europarl corpus. It has 86,700 distinct words (i.e. $|V| = 86700$). We previously saw that the number of parameters that need to be learnt for the bigram model is $O(|V|^2)$, i.e. $86,700^2 = 7,516,890,000$ in this example, which is order of magnitudes larger than the number of words the corpus contains. This problem of **sparse data** will cause most bigram or trigram probabilities to be zero, which results in any text containing an unseen ngram to be assigned a zero probability by the model.

Furthermore, in a given text, the most frequent word occurs approximately twice as often as the second most frequent word, three times as often as the third most frequent word, and so on. Figure 2 shows

the distribution of words in the novel Moby-Dick. This phenomena is called **Zipf's Law**. According to this law, any text corpora will contain a large fraction of low frequency words, which means that the sparsity problem will be a serious issue for language modeling.

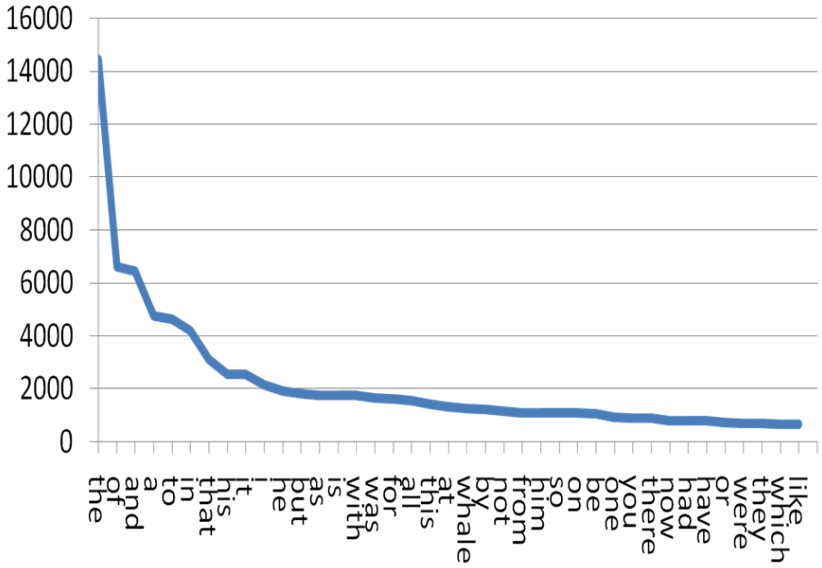


Figure 2: Distribution of words in Moby-Dick

Smoothing aims to alleviate the data sparsity issue. We will present two smoothing techniques:

1. **Discounting**: lowering probabilities of observed events to account for unseen events.
2. **Linear Interpolation**: combining unigrams with bigrams and trigrams, as unigrams suffer much less from data sparsity.

Discounting

The main idea behind discounting is lowering the non-zero probabilities to increase the zero probabilities so that no probability is zero.

Add-one discounting is an example of discounting methods. With add-one discounting, unigram probabilities are computed as follows:

$$P(w_i) = \frac{count(w_i) + 1}{N + |V|} \tag{10}$$

To get to this equation, we started with the maximum likelihood estimate for $P(W_i)$, i.e. the number of occurrences of the word, divided by the total number of words in the corpus, N:

$$P(w_i) = \frac{count(w_i)}{N} \tag{11}$$

And we artificially add 1 to the count of each word in the vocabulary, V, in order to boost uncommon words. To ensure the sum of unigram probabilities is one, we must add to the denominator the number of artificial word counts we added. Since we added 1 to every word in V, the denomenator should be $N + |V|$.

A straightforward generalization of add-one discounting is **add-α discounting**: as its name indicates, instead of adding 1 to each count, we add α:

$$P(w_i) = \frac{count(w_i) + \alpha}{N + \alpha|V|} \tag{12}$$

By the same token, the bigram and trigram probabilities using add-α-discount are as follows:

$$P(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i) + \alpha}{\text{count}(w_{i-1}) + \alpha|V|} \tag{13}$$

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{\text{count}(w_{i-2}, w_{i-1}, w_i) + \alpha}{\text{count}(w_{i-2}, w_{i-1}) + \alpha|V|} \tag{14}$$

α-discounting is simple to implement, but it is not very effective in practice. As Figure 3 shows, add one smoothing significantly overestimates the probability mass allocated for unseen events. Looking at this figure, you can also notice that lowering the value of *alpha* improves the estimates. How can we decide about the best value for α? For this purpose, we use a development set. We will estimate our language model with the different values of α on the training, and compare them on the development set. The optimal value selected on the development set will then be used during the test phase.

Count in 22M Words	Actual c* (Next 22M)	Add-one's c*	Add-0.0000027's c*
1	0.448	2/7e-10	~1
2	1.25	3/7e-10	~2
3	2.24	4/7e-10	~3
4	3.23	5/7e-10	~4
5	4.21	6/7e-10	~5
Mass on New	9.2%	~100%	9.2%
Ratio of 2/1	2.8	1.5	~2

Figure 3: Empirical performance of add-one smoothing.

Linear Interpolation

The larger the n-gram size, the more problematic data scarcity will become. However, longer n-grams contain more information, so they can still be very useful. In order to combine small n-grams with larger n-grams, one can use **linear interpolation**. In the following example, linear interpolation is used to combine unigrams, bigrams and trigrams:

$$\begin{aligned} P(w_i|w_{i-2}, w_{i-1}) =& \lambda_1 P_{ML}(w_i) + \lambda_2 P_{ML}(w_i|w_{i-1}) \\ &+ \lambda_3 P_{ML}(w_i|w_{i-2}, w_{i-1}) \end{aligned} \tag{15}$$

where $\sum \lambda_i = 1$ and $\lambda_i \geq 0$. We introduce these constraints on the values of λ to ensure that we are obtaining a valid probability distribution. These parameters can be estimated using the expectation maximization algorithm, which will be covered in later lectures.

There are several ways to refine linear interpolation. First, we don't have to use chains of n-grams, we can incorporate other models. Second, our λs can be separately estimated for different classes of n-grams. For instance, we may want to give a higher weight for frequent trigrams, and low weight for rarely observed ones. To this end, we can bucket the counts, and learn separate λs for each bucket.

Kneser-Ney Smoothing

Motivation:

Kneser-Ney is one of the most powerful smoothing techniques. It builds on two ideas: (1) absolute discounting, and (2) fertility. Let's start with discounting. As previously discussed, maximum likelihood estimates tend to overestimate the counts of observed events in the test data. We will remedy this issue by discounting actual counts by some small positive number d.

$$count^*(w_{i-1}, w_i) = \max(count(w_{i-1}, w_i) - d, 0) \tag{16}$$

which we then normalize to get the discounted bigram probability:

$$q_D(w_i|w_{i-1}) = \frac{count^*(w_{i-1}, w_i)}{count(w_{i-1})} \tag{17}$$

Discounting released some probability mass:

$$\alpha(w_{i-1}) = 1 - \sum_{w_i} q_D(w_i|w_{i-1}) \tag{18}$$

Then the brigram probability estimate is defined as:

$$P(w_i|w_{i-1}) = \begin{cases} q_D(w_i|w_{i-1}) & \text{if } count(w_{i-1}, w_i) > 0 \\ \alpha(w_{i-1}) \dfrac{P_{ML}(w_i)}{\sum\limits_{w \in \{w|count(w_{i-1},w)=0\}} P_{ML}(w)} & \text{otherwise} \end{cases}$$

The estimate of an unseen bigram is proportional to the weight of the corresponding unigram. For the seen bigrams, we compute estimates using discounted counts.

The second idea of Kneser-Ney smoothing is modeling word **fertility**. Given the sentence "I ate pasta with tomato _____", the most likely prediction is a food item (e.g. "sauce"). However if our training corpus contains many occurrences of "New York" and few occurrences of "sauce", the unigram model will prefer "York" over "sauce". We want our model to capture how likely it is for word w_i to appear in a new context, which is proportional to the number of bigrams the word w_i completes:

$$\widetilde{P}(w_i) \propto |\{w_{i-1}|count(w_{i-1}, w_i) > 0\}| \tag{19}$$

To get the exact probability, one needs to normalize it by the number of different bigrams that appear in the training set:

$$\widetilde{P}(w_i) = \frac{|\{w_{i-1}|count(w_{i-1}, w_i) > 0\}|}{|\{(w_{j-1}, w_j)|count(w_{j-1}, w_j) > 0\}|} \tag{20}$$

We interpolate the new $\widetilde{P}(w_i)$ with the original discounted bigram probabilities:

$$P(w_i|w_{i-1}) = q_D(w_i|w_{i-1}) + \alpha(w_{i-1})\widetilde{P}(w_i) \tag{21}$$

This is the final equation for Kneser-Ney smoothing. In practice, the Kneser-Ney smoothing is one of the best performing smoothing technique. It can be extended for other n-gram sizes, and improved using more fine-grained approaches, e.g. varying the discount d depending on the n-gram count.

Evaluation of Language Models

In order to assess the quality of a language model, one needs to define evaluation metrics. One evaluation metric is the **log-likelihood of a text**, which is computed as follows, assuming that the language model is a trigram model, and the text contains N words:

$$l_{\text{corpus}} = \log\left(\prod_{i=3}^{N} p(w_i|w_{i-2}, w_{i-1})\right) = \sum_{i=3}^{N} \log p(w_i|w_{i-2}, w_{i-1}) \quad (22)$$

In order to make this metric independent from the size of the corpus, one can compute the **average log-likelihood of the corpus on a per word basis**, i.e. the log-likehood of the corpus normalized by the number of words:

$$l_{\text{word\_average}} = \frac{1}{N} \sum_{i=1}^{N} \log p(w_i|w_{i-2}, w_{i-1}) \quad (23)$$

The most common evaluation metric for a language model is the **perplexity**, which can be computed directly from the average log-likelihood of the corpus on a per word basis:

$$\text{Perplexity} \;=\; 2^{-l_{\text{word\_average}}} \quad (24)$$

Note that in general, to make a meaningful comparison between two different language models, one needs to use the same vocabulary.

Using unigram, bigram and trigram models trained on 38 million words from the Wall Street Journal, and using a vocabulary of size 19,979 one obtains a perplexity 962, 170, and 109, respectively, when tested on 1.5 million words from the same journal.

What is the intuitive meaning of perplexity? This measure can be interpreted as an actual branching factor of the model. Let's explore this intuition using a simple uniform model for unigrams: $P(w) = \frac{1}{|V|}, \forall w \in V$. This means that:

$$P(w_1, ..., w_N) = \prod_{i \in [\![1,N]\!]} P(w_i) = \left(\frac{1}{|V|}\right)^N$$

$$\text{perplexity} = 2^{-\frac{1}{N} P(w_1,...,w_N)} = 2^{-\frac{1}{N} \log\left(\left(\frac{1}{|V|}\right)^N\right)} = |V| \quad (25)$$

Under this uniform language model, the perplexity is equal to the size of the vocabulary. Generally, perplexity captures the effective vocabulary size under the model. For instance, a trigram model described above has a factual branching factor of 109, even though it operates over the vocabulary of 19,979.

6.864 Advanced Natural Language Processing[1]

Lecture 3: Neural Networks

September 30, 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Narasimhan, Tianheng Wang.
Scribes: Nicholas Hynes, Harini Kannan, Liang Zhou

This third lecture introduces neural networks. In this lecture, we will present the structure of feedforward neural networks, examples of how they are used, the usefulness of having a hidden layer, and applications of these networks (including a neural language model, which will be continued in the next lecture).

Motivation and history

Neural networks have become a state-of-the-art technique in computer vision as they have improved in accuracy and speed over the past few decades. However, neural networks have not had the same impact yet in NLP: developing neural network models for NLP is still an open area with many ideas to explore.

Although basic training algorithms for neural networks were developed decades ago, they have risen in popularity in the last few years mainly for the following three complementary reasons. First, the availability of GPUs have made it much faster to run large parallel computations. Second, large and deep neural networks are highly flexible models and require substantial training data which have become increasingly available in recent years. The third reason is that small neural networks of the past tended to be difficult to learn whereas large networks today are easier to estimate with simple gradient based algorithms.

Feedforward neural network

Structure of a neural network

A neural network is made up of decision-making **units**. A unit is an individual node that applies a function to a weighted sum of input values. Intuitively, a unit can be thought of as a node that makes a decision based on its input. In a **feedforward neural network** the units are typically arranged in layers where the input to the next layer of units consists of activations of units in the preceding layer. Figure 1 is an example of a feedforward neural network with three layers.

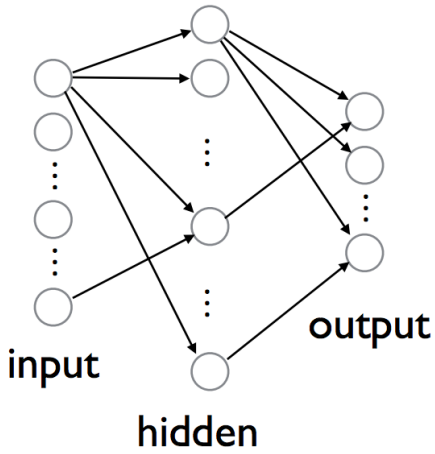


Figure 1: Feedforward neural network with three layers: input, hidden, and output.

1. First layer - Input layer of units, where each unit represents one of the values provided for the prediction task. For example, assume we want to perform sentiment analysis on a document. We could represent this document as a vector with as many coordinates as

there are words in the vocabulary and each coordinate could store the count or frequency of the corresponding word in the document. If we were to feed this vector through the neural network, each unit of the input layer would represent one element of the vector.

2. Second layer - Hidden layer of nodes. Intuitively, this layer is used to transform the original input by extracting features from the original input. We will see the importance of hidden layers in a later section.

3. Third layer - Output layer, which serves to classify the input. Below are three possible classifiers that the output layer could be, and the intuitive differences between each.

   (a) Binary classifier

      i. Example question it could answer: "Is the sentiment of this document positive?"

      ii. Number of units in output layer: 1

   (b) Multi-way classifier (a.k.a. multi-class classifier)

      i. Example question it could answer: "Is the sentiment of this document happy, neutral, or sad?"

      ii. Number of units in the output layer: as many as there are categories. If happy, neutral, and sad are the categories, then the number of units would be 3. The categories are assumed to be exclusive, i.e., we select one for each document.

   (c) Multi-label classifier

      i. In this case the categories are non-exclusive, i.e., we could select more than one. Example question it could answer: "Is this document about politics? Is the sentiment positive?".

      ii. Number of units in the output layer: as many as there are simple questions. In the example question, the number of units would be 2.

Example: Two-layer neural network with no hidden layer

Let us examine an example neural network with no hidden layer, just an input layer and an output layer. In Figure 2, $x_1, x_2, ...x_n$ are each of the inputs to the units in the input layer. We compute z, the input to the unit in the output layer, as the weighted sum of the inputs:

$$z = \sum_{i=1}^{d} x_i w_i + w_0$$

$w_1, w_2, ..., w_n$ are the *weights* and w_0 is the *offset* (also called *bias*). The unit in the output layer of our neural network takes z as input and

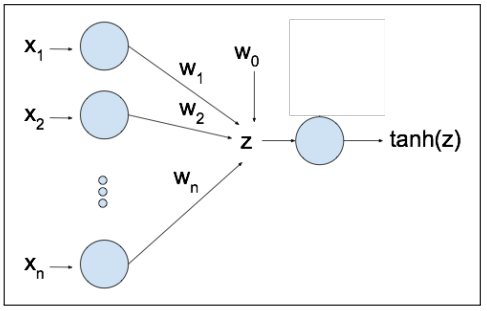


Figure 2: Neural network with just one input layer and one output unit

outputs $tanh(z)$. As Figure 3 illustrates, $\lim_{z\to-\infty} tanh(z) = -1$ and $\lim_{z\to+\infty} tanh(z) = 1$. $tanh$ is smooth and differentiable, which will be important for performing computations later on.

Let us simplify this example further and suppose that there are only 2 input units, which take in x_1 and x_2. What does the decision boundary look like (i.e. when $z = 0$)? If $z = 0$, then $w_1x_1 + w_2x_2 + w_0 = 0$. As shown in Figure 4, this defines a line whose normal vector is $[w_1, w_2]^T$. As we move in the direction of $[w_1, w_2]^T$, z increases, which means that $tanh(z)$ moves closer to 1. If we move away from the direction of $[w_1, w_2]^T$, then z decreases and $tanh(z)$ moves closer to -1. As shown in Figure 4, this intuitively means is that if the input pair (x_1, x_2) lies above the blue line, the unit will output a value between 0 and 1. If (x_1, x_2) lies below the line, the unit will output a value between -1 and 0.

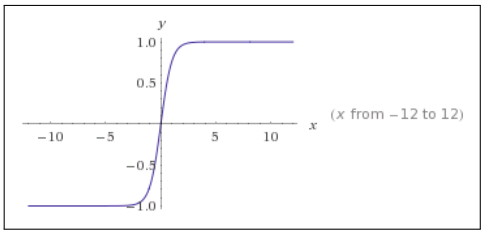


Figure 3: Graph of tanh(x). Source: Wolfram Alpha.

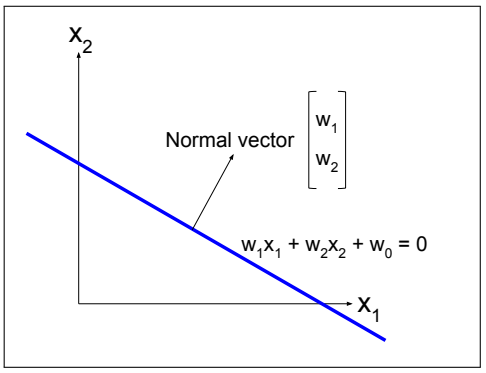


Figure 4: Decision boundary for 2 input units.

Why we need hidden layers

Let's try using our two-layer neural network to classify movie reviews. In Figure 5, we have three groups of movie reviews: negative reviews (left), unhelpful reviews (middle), and positive reviews (right). Suppose that we want a decision boundary that clearly separates unhelpful reviews. Can we do it with our two-layer neural network? Recall the linear boundary $w_0 + w_1x_1 + w_2x_2 = 0$ which comes from the expression for z:

$$z = \sum_{i=1}^{d} x_i w_i + w_0$$

Note that a single unit can only produce a linear boundary. Two possible linear boundaries are marked in Figure 5 as b_1 and b_2.

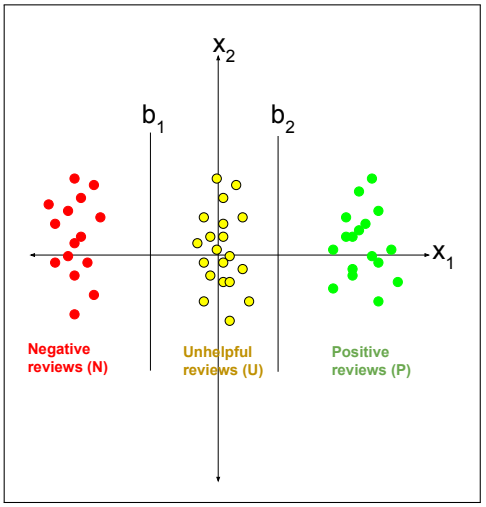


Figure 5: Graph of 3 groups of movie reviews: positive, negative, and unhelpful

The boundary b_1 separates the reviews into {negative} and {unhelpful, positive}. The boundary b_2 separates the reviews into {negative, unhelpful} and {positive}. However, there is no single linear boundary that would separate the reviews into {unhelpful} and {negative, positive}. In other words, given the nature of the input, we cannot use a single output unit operating on the input x that would adequately separate unhelpful reviews from the rest.

We can add hidden units to our network to correctly separate unhelpful reviews. In Figure 6, z_1 and z_2 are the input signals to the hidden nodes. The hidden unit activations, f_1 and f_2, are given by

$f_1 = tanh(z_1) = tanh(w_{1,1}x_1 + w_{1,2}x_2 + w_{1,0})$

$f_2 = tanh(z_2) = tanh(w_{2,1}x_1 + w_{2,2}x_2 + w_{2,0})$

Intuitively, f_1 and f_2 are **transformations** of x_1 and x_2 - we essentially have a new coordinate system in terms of f_1 and f_2, which means we can think of the original input in terms of f_1 and f_2 as well.

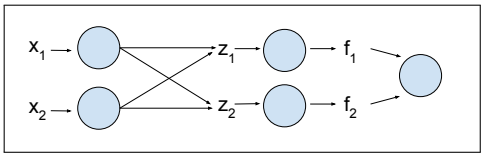


Figure 6: Neural network with 2 input units, 2 hidden units, 1 output unit

Now, if we were to graph the negative reviews, unhelpful reviews,

and positive reviews on a graph with f_1 and f_2 as axes, we get Figure 7. Notice how the input points have been transformed, and that we can now draw a decision boundary, marked as b_2, that linearly separates unhelpful reviews from negative and positive reviews. In other words, this boundary can be obtained with a single output unit that sees f_1 and f_2 rather than the original coordinates x_1 and x_2. This is the power of the hidden layer. We can learn to non-linearly transform the input signal into a coordinate system where the classification problem becomes simple.

Now, after the transformation (hidden layer), we can easily add an output unit to answer any of the following three possible questions

1. Is this review positive or not?
2. Is this review negative or not?
3. Is this review unhelpful or not?

Without a hidden layer, we could only answer the first two questions with a simple output unit.

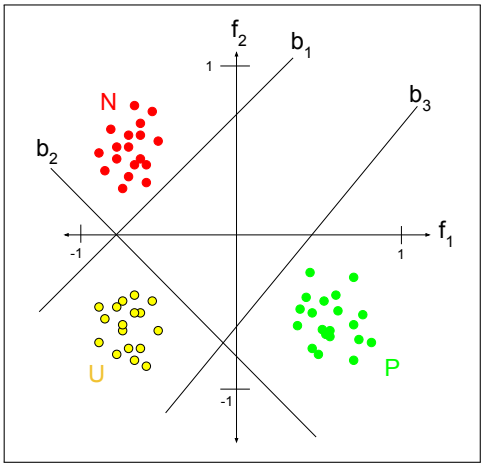


Figure 7: Graph of positive, negative, and unhelpful movie reviews on new axes

Multi-way vs Multi-label

We can now include three output units, one unit for each of the binary questions. This would be multi-label classification since more than one (or none) of the units may be active. For example, the point labeled X in Figure 8 would be classified as both positive and unhelpful.

Let's see how this works algebraically. From Figure 9, we have that:

$$z_p = w_{0,P} + f(z_1)w_{1,P} + f(z_2)w_{2,P}$$

$$z_u = w_{0,U} + f(z_1)w_{1,U} + f(z_2)w_{2,U}$$

$$z_n = w_{0,N} + f(z_1)w_{1,N} + f(z_2)w_{2,N}$$

(weights are not shown in the figure.) Each of z_p, z_u, z_n is linear function of $f_1 = f(z_1)$ and $f_2 = f(z_2)$, therefore defining a linear decision boundary in f_1, f_2 coordinates. In this example, z_n is represented by b_1, z_u by b_2, and z_p by b_3. We can use the tanh activation function for multi-label classification so that $P = tanh(z_p)$, $U = tanh(z_u)$, $N = tanh(z_n)$. In order to predict actual labels (rather than graded responses pertaining to the labels), we can simply label each example (give it three labels) based on the sides of the decision boundaries that the example falls in. In other words, we would label it as "positive" whenever $P > 0$ ($z_p > 0$), and "negative" when $N > 0$ ($z_n > 0$) as well as "unhelpful" if $U > 0$ ($z_u > 0$).

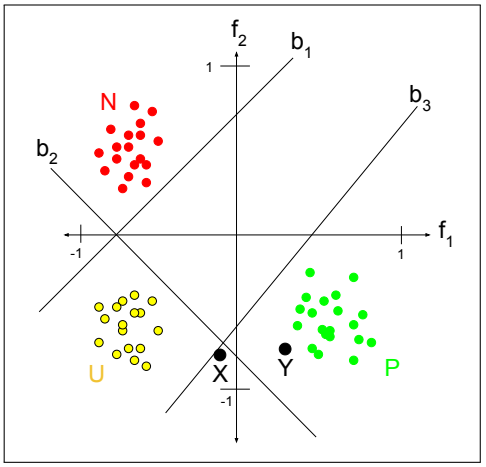


Figure 8: Multi-label classification of points X and Y

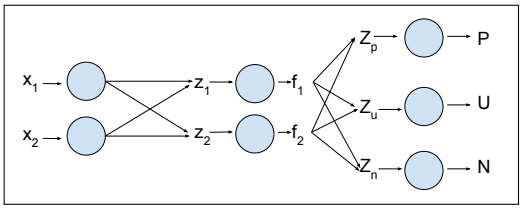


Figure 9: Neural network with 2 input units, 2 hidden units, 3 output units

Multi-way classification

In contrast, if we formulate the problem as a multi-way classification problem, the categories become exclusive in the sense that a review can only be positive, negative, or unhelpful. In this case, instead of using tanh activation functions for the output units, we tie them together via the **softmax** function. The softmax function transforms a set of real numbers to a probability scale so that the outputs will be predicted probabilities for each of the exclusive labels. Mathematically, for a given review x, we can compute the probability that this review is negative with the following expression:

$$P(y = n|x) = \frac{e^{z_n}}{\sum_{i \in \{P,U,N\}} e^{z_i}}$$

Recall that z_p, z_u, and z_n are the inputs to the units of the output layer as shown in Figure 9. When we use the softmax function, the output units will output $P(y = p|x)$, $P(y = n|x)$, and $P(y = u|x)$, respectively. If we are forced to predict a label rather than probability values, we would simply take the label with the highest probability.

Application: Neural language model for trigrams

We conclude this lecture with a discussion of an application of neural networks to NLP: a neural language model for trigrams. The neural language model for trigrams aims to answer the following question: given two consecutive words as input (w_{t-2} and w_{t-1}), can we predict the next word in the sentence? If we can answer this question, we can create whole sentences from only two initial words, using the neural network to generate the next word each time.

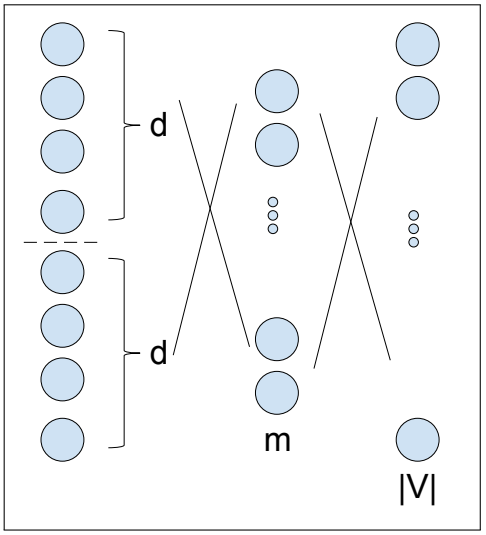


Figure 10: A neural network for trigrams.

First, we create our vocabulary of words (with the END symbol added to the vocabulary, as we also want to be able to predict when a sentence should end). Next, as shown in Figure 10, we will represent w_{t-2} and w_{t-1} as vectors $v(w_{t-2})$ and $v(w_{t-1})$. Each of these vectors will be of size d, and we can concatenate them to form an input vector x of size $2d$. We then have a hidden layer of size m in order to transform the input as discussed in the movie reviews example. Lastly, we have an output layer of the same size as vocabulary V, where we can use multi-way classification to find the output word with the highest probability.

By tuning the weights of this neural network using back propagation, we obtain a probability model over the words in the sentence. Indeed, the model can create a sentence, one word at a time, selecting next word probabilistically based on the two preceding words, w_{t-2}

and w_{t-1}. We will describe this model as well as the back propagation algorithm in more detail in the next lecture.

6.864 Advanced Natural Language Processing[1]

Lecture 4: Neural Networks, Application to Language Models

22 September 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Narasimhan, Tianheng Wang.
Scribes: Quan Minh Nguyen, Anil Shanbhag, Min Zhang

Introduction

The last lecture covered the basics of neural networks. This lecture presents their use in language models, and explains how to learn the parameters of neural networks with backpropagation.

Neural Language Model

A trigram language model predicts the next word given two previous words. A sentence in a trigram language model takes the form

$$\langle S\text{-}2\rangle \ \langle S\text{-}1\rangle \ w_0 \cdots w_{t-2} \ w_{t-1} \ w_t \cdots w_T \ \langle END\rangle \tag{1}$$

in which w_i is a word in the text, $\langle S\text{-}2\rangle$ and $\langle S\text{-}1\rangle$ are start symbols, and $\langle END\rangle$ is the end symbol. A bigram model requires one start symbol, whereas a trigram model requires two.

We will introduce a trigram language model in the form of a neural network. The network we will study can be divided into three sets of layers: the *input*, *hidden*, and *output* layers. The input layer takes in vector encodings of the two preceding words in the sentence. Each unit in the hidden layer performs a non-linear transform of the input by passing a weighted combination of the inputs through a non-linear activation function (here the tanh function). Finally, the output layer turns a weighted combination of hidden layer activations into a probability distribution over the next word.

Word vectors

The input layer consists of two previous words represented by two d-dimensional word vectors concatenated together. The input vector x is therefore a $2d$-dimensional real vector.

How are the words mapped to vectors? A common (direct) way to represent words as vectors is as one-hot vectors. That is, a particular word $w = k$ would be expressed as a $|V|$-dimensional vector with all zeros except for the k^{th} position, which is set to 1. However, in this high dimensional sparse representation, word vectors corresponding to any two distinct words are always orthogonal. Instead, we will use much lower dimensional real valued dense vectors that are learned together with the network. Our goal is to obtain vectors that map related words to similar vectors. Indeed, training our neural network

will create word feature vectors that yield similar vectors for words that tend to function the same as part of the language model. That is, w_{t-2}, w_{t-1} become similar to w'_{t-2}, w'_{t-1} if they are typically followed by the same words.

Model Architecture

The hidden layer contains m units while the output layer has $|V|$ units, one for each possible word in the vocabulary. We can change m but $|V|$ is fixed by the task. Weights W^h_{ij} mediate the signal from the input layer (unit i) to the hidden layer (unit j). Similarly, W^o_{jk} represents the weight between a hidden unit j activation and the output unit k.

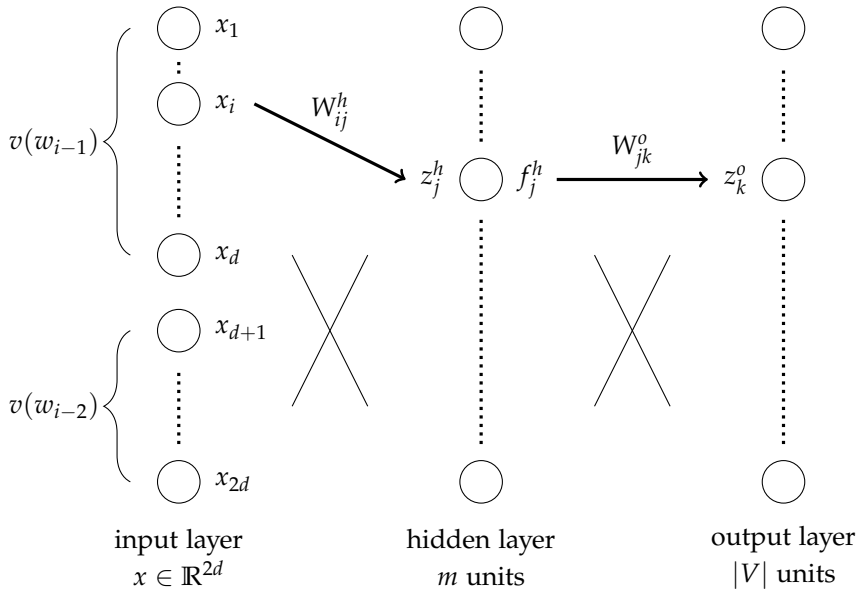


Figure 1: Structure of a neural network for a language model.

Specifically, each hidden unit j takes as its input a weighted combination of word vector coordinates

$$z^h_j = W^h_{0j} + \sum_{i=1}^{2d} x_i W^h_{ij} \tag{2}$$

Note that the unit also has an adjustable bias W^h_{0j} that does not depend on the input. The aggregate input to the hidden unit, z^h_j, is turned into its activation f^h_j by passing the input through the non-linear activation function

$$f^h_j = \tanh(z^h_j) \tag{3}$$

For small inputs, the *tanh* function is roughly linear but saturates to -1 or 1 for large negative or positive inputs, respectively.

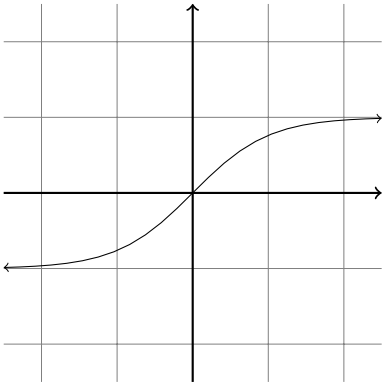


Figure 2: The *tanh* function.

Each output unit k takes in a weighted sum of hidden unit activations

$$z_k^o = W_{0k}^o + \sum_{j=1}^{m} f_j^h W_{jk}^o \tag{4}$$

with an adjustable bias w_{0k}^o as before. However, unlike in the hidden layer, these aggregate inputs z_k^o, $k \in \{1, \ldots, |V|\}$, are mapped to a probability distribution over the words by means of the softmax function

$$p_k = \frac{e^{z_k^o}}{\sum_{l=1}^{|V|} e^{z_l^o}} \tag{5}$$

Here $p_k = P(w_i = k | w_{i-2}, w_{i-1})$ represents the probability that the neural network predicts $w_i = k$ given two preceding words w_{i-2} and w_{i-1} in the sentence. Note that, by construction, $\sum_{k=1}^{|V|} p_k = 1$.

Structural Parameters

We can also view d (dimensionality of word vectors), m (number of hidden units), and n (n-gram) as parameters to optimize. These are, however, parameters we should learn differently from the weights. For example, increasing d, m and n will simply make the model more flexible, able to fit any training data. We should therefore set these parameters on the basis of the development data rather than the training data. For now, we will assume that these parameters remain fixed.

Learning the Neural Network

The neural network model involves a number of adjustable parameters that need to be learned. Specifically, we need to find W_{ij}^h, W_{0j}^h, W_{ij}^o, W_{0j}^o as well as the d-dimensional word vectors $v(w), w \in V$. Our goal is to make p_k approximate the conditional probability that word $w = k$ appears after w_{i-2}, w_{i-1}. In the following section, we will first define a log-likelihood criterion to evaluate how well the neural network model performs with a given set of parameters. We will then discuss stochastic a gradient ascent method to optimize the log-likelihood objective with respect to the parameters. The gradient ascent method requires us to evaluate derivatives with respect to the parameters. Given that the neural network output is a complicated function of its parameters, these derivatives are evaluated in stages via back-propagation.

Log Likelihood

The log-likelihood function is simply the log-probability of predicting each word in the sentence, across the sentences in the corpus. For a

sentence s with words w_i^s, $i = 0, \ldots, n_s$, where n_s is the length of the sentence and $w_{n_s}^s =$ <END>, the log-likelihood is given by

$$l_s(\theta) = \sum_{i=0}^{n_s} \log P(w_i^s | w_{i-2}^s, w_{i-1}^s) \tag{6}$$

where $P(w_i^s | w_{i-2}^s, w_{i-1}^s) = p_{w_i^s}$ is evaluated by passing the two preceding words through the network. Note that two initial words in the sentence, w_{-2}^s, w_{-1}^s, are always fixed to be the start symbols. θ is used here to collectively refer to all the adjustable parameters in the network, excluding structural parameters m, d, and n (n-gram) that remain fixed for now. The overall log-likelihood function that we aim to maximize is then just $l(\theta) = \sum_s l(\theta)_s$.

Stochastic Gradient Ascent

Stochastic gradient (SG) algorithm is an incremental algorithm for maximizing the log-likelihood function $l(\theta)$. The algorithm breaks the training corpus into a collection of instances, here trigrams, and selects a trigram, e.g., (w_{i-2}, w_{i-1}, w_i), at random or in a sequence, adjusting the parameters each time slightly in the direction that maximizes the chosen term

$$\log P(w_i | w_{i-2}, w_{i-1}^s) = \log p_{w_i}$$

The idea is that multiple such small steps will increase $l(\theta)$ as desired. Specifically, we adjust the parameters θ according to

$$\theta \leftarrow \theta + \eta \nabla_\theta \log p_{w_i} \tag{7}$$

where η is the learning rate and θ is a long vector of parameters that contains $v(w), w \in V$, W_{ij}^h, W_{0j}^h, $i = 1, \ldots, 2d$, $j = 1, \ldots, m$, W_{jk}^o, W_{0k}^o, $j = 1, \ldots, m$, $k = 1, \ldots, |V|$. Since many of the trigrams in the corpus will push the parameters in a similar direction, the stochastic gradient method is a simple and efficient way to increase the log-likelihood. Note that evaluating the full gradient $\nabla l(\theta)$ would require us to pass through the corpus for each parameter update. Moreover, the randomness inherent in the stochastic updates is actually beneficial in optimizing complex models.

The learning rate η determines the size of the step that the algorithm takes along the gradient. For small η, many iterations are needed before getting to a good solution while large η could result in oscillations. Usually we want η to satisfy the following conditions:

$$\sum_t \eta_t = \infty, \sum_t \eta_t^2 < \infty \tag{8}$$

where t is the iteration number. The first condition ensures that the algorithm will be able to find a locally optimal solution regardless of

the starting point and the second one controls oscillations. We will talk later about AdaGrad, which adjusts the learning rate dynamically depending on the gradient.

Backpropagation

It remains to evaluate the gradient $\nabla_\theta \log p_w$, i.e., derivatives of the log-probability with respect to each of the parameters. Intuitively, we need to understand how changing the weights and biases in a network changes the log-probability.

For $* \in \{o, h\}$:

$$\frac{\partial \log p_w}{\partial W^*_{ij}} = \left(\frac{\partial \log p_w}{\partial z^*_j}\right)\left(\frac{\partial z^*_j}{\partial W^*_{ij}}\right), \tag{9}$$

$$\frac{\partial \log p_w}{\partial W^*_{0j}} = \left(\frac{\partial \log p_w}{\partial z^*_j}\right)\left(\frac{\partial z^*_j}{\partial W^*_{0j}}\right). \tag{10}$$

For the output layer, from (2),

$$\frac{\partial z^o_k}{\partial W^o_{jk}} = f^h_j, \quad \frac{\partial z^o_k}{\partial W^o_{0k}} = 1. \tag{11}$$

For the hidden layer, from (4),

$$\frac{\partial z^h_j}{\partial W^h_{ij}} = x_i, \quad \frac{\partial z^h_j}{\partial W^h_{0j}} = 1. \tag{12}$$

We need to still evaluate

$$\delta^*_j = \frac{\partial \log p_w}{\partial z^*_j}, \tag{13}$$

for $* \in \{o, h\}$. These can be evaluated by working backwards from the output layer towards the input layer. For example,

$$\delta^h_j = \frac{\partial \log p_w}{\partial z^h_j} = \frac{\partial f^h_j}{\partial z^h_j}\frac{\partial \log p_w}{\partial f^h_j} = \frac{\partial f^h_j}{\partial z^h_j}\sum_{k=1}^{|V|}\frac{\partial z^o_k}{\partial f^h_j}\frac{\partial \log p_w}{\partial z^o_k} \tag{14}$$

$$= (1 - (f^h_j)^2)\sum_{k=1}^{|V|} W^o_{jk}\delta^o_k \tag{15}$$

since $\partial f^h_j / \partial z^h_j = 1 - \tanh^2(z^h_j) = 1 - (f^h_j)^2$ and $\partial z^o_k / \partial f^h_j = W^o_{jk}$. This is referred to as the *backpropagation* algorithm, easily generalized to any number of layers in the network. Lastly, we see that

$$\delta^o_k = \frac{\partial \log p_w}{\partial z^o_k} = \frac{\partial}{\partial z^o_k}\left[z^o_w - \log\sum_{l=1}^{|V|}\exp(z^o_l)\right] = \begin{cases} 1 - p_k & \text{if } k = w \\ -p_k & \text{o.w.} \end{cases}$$

AdaGrad

In the previous section we discussed how to use stochastic gradient ascent to learn the parameters of the neural network and backpropogation to evaluate the required gradient. Setting the learning rate in following the stochastic gradient can make a real difference in how fast the algorithm learns. We discus here an adaptive method for setting the learning rate known as AdaGrad.

Consider two situations:

1. If the objective looks like in Figure 3, the gradient would typically (at most points) have a small magnitude. As a result, we would need a large learning rate to quickly reach the optimum.

2. In contrast, in Figure 4 the objective changes rapidly as a function of the parameter, thus the gradient would typically be very large. Using a large learning rate would result in too large steps, oscillating around but not reaching the optimum.

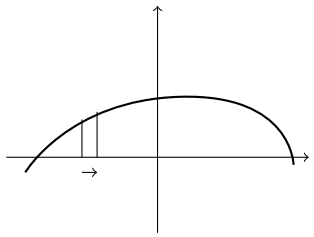


Figure 3: Slowly varying log-likelihood objective (small gradient)

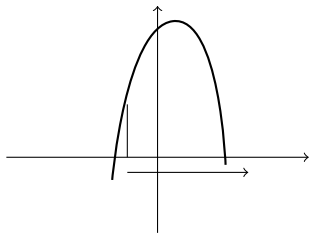


Figure 4: Log-likelihood that changes rapidly in response to parameter changes (large gradient)

These two situations occur because the learning rate is set independently of the gradient. To remedy the issue, AdaGrad maintains a variable G which just accumulates squared norms of the gradients seen so far. Note that it increases slowly when the gradients are small (at any point in the algorithm) while response quickly to large gradients. The learning rate should therefore be inversely related to G as in

$$\begin{aligned} G &\leftarrow G + \|\nabla_\theta \log p_w\|^2 \\ \theta &\leftarrow \theta + \frac{\eta}{\sqrt{G}} \nabla_\theta \log p_w \end{aligned}$$

The resulting algorithm is not very sensitive to η any more. In fact, a constant η such as $1/2$ would typically be fine. Note that G implies a decreasing learning rate even if the gradients remain constant (in terms of magnitude) for a while since

$$G_t = \sum_{i=1}^{t} \|\nabla_\theta \log p_{w_i}\|^2 = O(t)$$

so that that the actual learning rate ($\eta/\sqrt{G}$) used in the update is $O(1/\sqrt{t})$.

There are many ways to apply AdaGrad to the network models. For example, it is typically beneficial to "bundle" the adaptive gradients differently, per node in the network rather than using the same accumulator G across the network. For example, each hidden or output unit could have their own G and the associated gradients would be those pertaining to the incoming weights to these units.

Recurrent Neural Network

Until now we studied feed forward neural networks where the flow of computation proceeds from one layer to the next. As a language model, our n-gram feed-forward model requires n as a parameter, i.e., how many preceding words we incorporate into the prediction of the next word. But the length of this dependence often varies from sentence to sentence requiring potentially a large n. However, by increasing n we are rapidly making the model more flexible and therefore requiring more data to learn properly.

Recurrent neural networks is a class of neural networks that have memory in the form of a hidden state. This state is updated from one step to the next, i.e., the current state is fed back to the network along with the input for the next step. There are many ways to maintain this state.

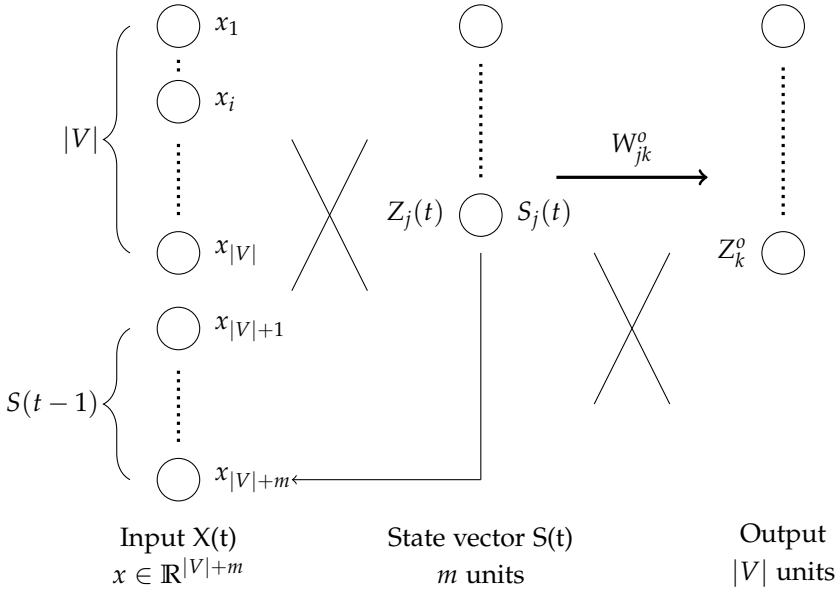


Figure 5: Using a recurrent neural network with a trigram model.

A simple recurrent neural network language model is shown in Figure 5. The coordinates $x_1 ... x_{|V|}$ could represent a one-hot vector for the preceding word while $x_{|V|+1} ... x_{|V|+m}$ represents the hidden state vector from the previous step. The state vector is fed back as an input. Note that, as the model is applied sequentially to the words in the sentence, the state vector is affected by all the preceding words, not just the last two. Of course, the model can also quickly forget (wash out) the memory from earlier words. The key point is that there is no fixed n step dependence as in typical n gram models.

6.864 Advanced Natural Language Processing[1]

Lecture 5: EM Algorithm and Topic Model

24 September 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Dongyoung Kim.

Earlier, we saw cases where the full data was observable and we could apply a straightforward maximum-likelihood estimator for the parameters. In Lecture 4, we introduce a Hidden Variable Model where data is partially observable. Then, we provide basic intuitions for the EM algorithm which can be used to optimize the likelihood on such Hidden Variable Models. We explore further by demonstrating the EM algorithm on a toy example with biased coins. Later, we connect our concept to more relevant topics in NLP such as topic modeling.

Hidden Variable Models

Motivation

Hidden Variable Models are very common in NLP applications. To gain intuition, see the following examples.

1. *Topic Model*: Consider the following traditional unigram model. Suppose document D is comprised of N words such that $D = \{w_1, w_2, ..., w_N\}$. Given a unigram distribution $p(w_i)$, the log-likelihood of any document will be given by the same form

$$\log p(D) = \sum_{i=1}^{N} \log p(w_i)$$

   However in practice, each documents are different and therefore we may want to use different distributions over the words depending on the context. This idea can be easily implemented in Topic Models. A topic is denoted by z where $z \in \mathcal{Z}$, and $\mathcal{Z}$ is our pre-defined universe of topics of k dimension. Each word w_i is sampled from a corresponding topic z_i.

   The problem in Topic Models arise from the fact that the topic information is not provided in the training corpus. If the entire data had been observed, we could simply estimate the conditional distribution as the empirical count,

$$\hat{p}(w|z) = \frac{count(w, z)}{count(z)} \tag{1}$$

   However, such straightforward approach seen in (1) is no longer applicable if the variables z_i are hidden. Since it is unlikely to have each word in a document explicitly tagged with its related topics, topic models naturally arise as Hidden Variable Models.

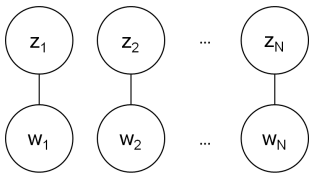


Figure 1: Topic model is where each word is associated with a conditional topic. For example, it is more likely to observe word $w_i = "Cat"$ if the topic is $z_i = "Pets"$.

2. *Machine Translation Alignment*: In Machine Translation, one is given two sentences in different languages with the same meaning. As seen in Figure 2, when the word *"I"* is paired together many times with the corresponding Korean word, then it is probably a good indicator that *"I"* should be translated to that word. Therefore, the first step in Machine Translation is to align corresponding translations. However, the alignments are not explicitly given in real life applications and should be treated as hidden variables one has to estimate.

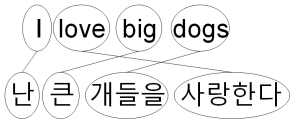


Figure 2: In machine translation, each word from one language has a hidden alignment to its counterpart.

3. *Part-of-Speech Tagging*: Part-of-speech tagging, which will be covered in the next lecture, matches each word in a sentence with its part of speech in English. For the example in Figure 3, *"I"* is tagged with *"pronoun"*, *"love"* with *"verb"*, *"big"* with *"adjective"*, and *"dogs"* with *"noun"*. When the full observed data observable, one can apply a supervised learning algorithm. However when the parts of speech are not observed, then they are treated as hidden variables.

Figure 3: Each word is associated with a hidden part-of-speech in English.

Observed Case vs. Unobserved Case

In previous lectures we looked at models where the full data was observed. We use

$$D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$$

to denote the given dataset in such cases, which is represented as a list of sampled tuples. Each data point (x, y) is independently sampled from some distribution $p_{X,Y}(x, y)$ parameterized by θ. The objective function we are trying to maximize is given by the following

$$\theta^* = \underset{\theta}{\text{argmax}} \sum_{i=1}^{N} \log p_{X,Y}(x_i, y_i|\theta) \tag{2}$$

Now, let's look at the unobserved case where $\mathbf{y}$ is unseen. In other words, $\mathbf{x}$ is the observed variable and $\mathbf{y}$ is the hidden variable. In this case, the observed data is given by

$$D = \{x_1, x_2, ..., x_N\}$$

In this case, we replace the objective function with the log-likelihood of the *partially observed data*, which is

$$\begin{aligned} \theta^* &= \underset{\theta}{\text{argmax}} \sum_{i=1}^{N} \log p_X(x_i|\theta) \\ &= \underset{\theta}{\text{argmax}} \sum_{i=1}^{N} \log \sum_{y \in \mathcal{Y}} p_{X,Y}(x_i, y|\theta) \end{aligned} \tag{3}$$

where the second equality is due to marginalization over y over $p_{X,Y}(x, y)$. The key point is to note that the criteria for the observed case and the unobserved case is different, and we should use a different objective function when we are trying to optimize for the parameter θ. In the following sections we will demonstrate how the EM algorithm finds the optimum parameter for (3).

Before we see how the EM algorithm can optimize (3), let's first how we can compute the solution for (2) through a simple example.

Example 1. Assume a simple unigram model where there are two words in the universe such that $V = \{\text{"dog"}, \text{"cat"}\}$. There is one free parameter given by

$$\begin{aligned} \theta &= p(\text{"dog"}) \\ 1 - \theta &= p(\text{"cat"}) \end{aligned}$$

Given document $D = \{w_1, w_2, ..., w_N\}$, the log-likelihood of the document is given by

$$\begin{aligned} l(D|theta) &\triangleq \log p(D|\theta) \\ &= \log(\theta^d (1-\theta)^c) \\ &= d \cdot \log \theta + c \cdot \log(1-\theta) \end{aligned}$$

Where d is a constant equal to the number of occurrence of "dog" in the document, and c is equal to that of "cat". To compute θ^*, we take the partial derivative of the log-likelihood with respect to θ and set it to zero.

$$\begin{aligned} \frac{\partial l(D|theta)}{\partial \theta} &= \frac{d}{\theta^*} - \frac{c}{1-\theta^*} \\ &= 0 \end{aligned}$$

Multiplying $\theta^*(1-\theta^*)$ to both sides,

$$\begin{aligned} d(1-\theta^*) - c\theta^* &= 0 \\ d - d\theta^* - c\theta^* &= 0 \\ (d+c)\theta^* &= d \\ \rightarrow \qquad \theta^* &= \frac{d}{d+c} \end{aligned}$$

We can easily derive that the Maximum Likelihood Estimator is given by the fraction of the empirical counts.

EM Algorithm

Example 2. We will now set up a simple example to walk through how the EM algorithm works. Suppose we have a card and two different coins, where all probabilities are biased. We will take the following steps to sample a sequence of coin flips.

1. Flip the card, which either returns side A or side B.
2. If A was drawn from step 1, toss coin 1 three times. Otherwise if B was drawn, toss coin 2 three times.
3. repeat steps 1-2

Expressing this more formally, y represents the side of the card and x represents the outcome of three consecutive coin tosses. H and T denotes the head and tail of a coin, respectively.

$$y \in \mathcal{Y} = \{A, B\}$$
$$x \in \mathcal{X} = \{HHH, HTH, ..., TTT\}$$

Furthermore, let us assume that we did not see the card values y, and we could only observe the sequence of the coin flips x. Since the data is partially observed, this example is an instance of a Hidden Variable Model.

There are three free parameters in this setup, given by $\theta = \{\alpha, p_A, p_B\}$. The parameters are described as follows

$$\alpha = p(Card = A)$$
$$p_A = p(Coin1 = H)$$
$$p_B = p(Coin2 = H)$$

It is easy to see that

$$p(x, y|\theta) = p(y|\theta)p(x|y, \theta)$$
$$p(y|\theta) = \begin{cases} \alpha & \text{if } y = A \\ 1 - \alpha & \text{if } y = B \end{cases}$$
$$p(x|y, \theta) = \begin{cases} p_A^{h(x)}(1 - p_A)^{t(x)} & \text{if } y = A \\ p_B^{h(x)}(1 - p_B)^{t(x)} & \text{if } y = B \end{cases}$$

Where we define $h(x)$ to be a function that is equal to the number of heads in sample x, and $t(x)$ to be the number of tails. For example, if $x_0 = HHT$, then $h(x_0) = 2$ and $t(x_0) = 1$.

Observed Case

Let's go back on our assumption that the card values $\mathbf{y}$ were hidden, and say the card flips and the coin tosses were entirely observable. Each card flip is associated with three coin tosses. Suppose after three iterations, the observed data was given as follows

$$D = \{(A, HHH), (A, HHH), (B, TTT)\}$$

where $(y, x) = (A, HHH)$ denotes that the observed card flip was A, followed by three heads. Earlier in Example 1, we have seen that

applying partial derivative to the log-likelihood given in (2), the MLE parameters are simply given by the empirical counts. Using this fact for granted, the parameter values for these specific samples are given by

$$\left.\begin{aligned} \alpha^* &= \frac{\text{count(A)}}{3} = \frac{2}{3} \\ p_A^* &= \frac{\text{count}(A,H)}{\text{count}(A,H)+\text{count}(A,T)} = \frac{6}{6} \\ p_B^* &= \frac{\text{count}(B,H)}{\text{count}(B,H)+\text{count}(B,T)} = \frac{0}{3} \end{aligned}\right\} \quad \text{MLE parameters}$$

Shortly we will see that this analytic solution has a very similar form to the solution for each iteration in the EM algorithm, which is covered in the next section.

Unobserved Case

Continuing with the problem setup in Example 2, suppose the card flips are no longer observable. In other words, we will treat the card flip y as a hidden variable. Consider the following sample data

$$D = \{HHH, HHH, TTT\} = \mathbf{x}$$

Let us make an initial guess on the parameter values. Assign arbitrary prior values such that

$$\alpha^{(0)} = 0.1, \quad p_A^{(0)} = 0.8, \quad p_B^{(0)} = 0.5 \tag{4}$$

The initial guess in (4) is denoted with a superscript 0 to mark that this is the belief of the parameters at time 0. We will see how the belief is updated over each iteration of the EM algorithm.

For each x_i in the dataset, we compute $p_{Y|X,\theta}(y|x_i, \theta^{(0)})$ over all possible values of $y \in \mathcal{Y} = \{A, B\}$. For example, for our For $x_1 = HHH$ in the given example dataset,

$$\begin{aligned} &p(y_1 = A|x_1 = HHH, \theta^{(0)}) \\ &= \frac{p(x_1 = HHH, y_1 = A|\theta^{(0)})}{p(x_1 = HHH, y_1 = A|\theta^{(0)}) + p(x_1 = HHH, y_1 = B|\theta^{(0)})} \\ &= \frac{\alpha p_A^3}{\alpha p_A^3 + (1-\alpha)p_B^3} \\ &= \frac{0.1 \times 0.8^3}{0.1 \times 0.8^3 + (1-0.1) \times 0.5^3} \approx 0.3 \end{aligned}$$

$$p(y_1 = B|x_1 = HHH, \theta^{(0)})$$

$$= \frac{p(x_1 = HHH, y_1 = B|\theta^{(0)})}{p(x_1 = HHH, y_1 = A|\theta^{(0)}) + p(x_1 = HHH, y_1 = B|\theta^{(0)})}$$
$$= \frac{(1-\alpha)p_B^3}{\alpha p_A^3 + (1-\alpha)p_B^3}$$
$$= \frac{(1-0.1) \times 0.5^3}{0.1 \times 0.8^3 + (1-0.1) \times 0.5^3} \approx 0.7$$

Recall from the previous section that in the fully observed case, each x_i was explicitly associated with a *single* count of either $y_i = A$ or $y_i = B$. In the unobserved case, we assume that each x_i is associated with *fractional counts* over all possible values of y_i, to which we assign the *expected count* of y_i given x_i and our prior knowledge of θ.

x_i	y_i	empirical count
HHH	A	1
HHH	A	1
TTT	B	1

(a) y is observed

x_i	y_i	fractional count
HHH	A	0.3
	B	0.7
HHH	A	0.3
	B	0.7
TTT	A	0.5
	B	0.5

(b) y is unobserved

Table 1: In the fully observed case (a), a single value of y (which is the observed value itself) is associated with each i^{th} data sample. In the unobserved case (b), we assume that a entire list over the alphabet $\mathcal{Y}$, is associated with x_i. Each element in the spectrum is assigned a weight, or the *fractional count*, which corresponds to the expected value of y given x_i. The fractional count can also be understood as the confidence in the sample (x_i, y_i) given a prior belief over the parameters.

Hereafter the remaining steps are almost identical to the MLE maximization of the parameters. Again we will derive our distribution by counting, but since the actual counts are no longer available, we will count the fractional counts instead. The parameters for the next iteration, or time 1, will be computed as follows

$$\alpha^{(1)} \leftarrow \frac{\text{fractional count of } A}{3} = \frac{0.3 + 0.3 + 0.5}{3} = \frac{1.1}{3}$$
$$p_A^{(1)} \leftarrow \frac{\text{fractional count of } (A, H)}{\text{fractional count of } (A, H) + \text{fractional count of } (A, T)}$$
$$= \frac{3 \times 0.3 + 3 \times 0.3}{3 \times 0.3 + 3 \times 0.3 + 3 \times 0.5} = \frac{1.8}{3.3}$$
$$p_B^{(1)} \leftarrow \frac{\text{fractional count of } (B, H)}{\text{fractional count of } (B, H) + \text{fractional count of } (B, T)}$$
$$= \frac{3 \times 0.7 + 3 \times 0.7}{3 \times 0.7 + 3 \times 0.7 + 3 \times 0.5} = \frac{4.2}{5.7}$$

The EM algorithm guarantees that the likelihood given by Equation (3) is increased over every iteration as we update our parameters $\theta^{(0)}, \theta^{(1)}, \theta^{(2)}, ..., \theta^{(T)}$. In summary, the EM algorithm is given by the following steps

1. Randomly initialize $\theta^{(0)}$

2. Repeat until convergence

   - **E-step:** Based on $\theta^{(t)}$, compute $p_(y|x, \theta^{(t)})$ and count the fractionals
   - **M-step:** Re-estimate $\theta^{(t+1)}$

Properties of EM Algorithms

The EM algorithm is guaranteed to converge to a local maximum. As with many local-maximum search methods, initialization plays a crucial role in the EM algorithm. See the Figures 4 and 5 to understand when the EM algorithm works well and when it can get stuck, depending on the initialization values.

Iteration	λ	p_1	p_2	$\tilde{p}_1$	$\tilde{p}_2$	$\tilde{p}_3$	$\tilde{p}_4$
0	0.3000	0.3000	0.6000	0.0508	0.6967	0.0508	0.6967
1	0.3738	0.0680	0.7578	0.0004	0.9714	0.0004	0.9714
2	0.4859	0.0004	0.9722	0.0000	1.0000	0.0000	1.0000
3	0.5000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000

Figure 4: The coin example for $\mathbf{y} = \{HHH, TTT, HHH, TTT\}$. The solution that EM reaches is intuitively correct: the coin-tosser has two coins, one which always shows up heads, the other which always shows tails, and is picking between them with equal probability ($\lambda = 0.5$). The posterior probabilities $\tilde{p}_i$ show that we are certain on coin 1 (tail-biased) generated y_1 and y_4, whereas coin 2 generated y_2 and y_3.

Iteration	λ	p_1	p_2	$\tilde{p}_1$	$\tilde{p}_2$	$\tilde{p}_3$	$\tilde{p}_4$
0	0.3000	0.7000	0.7000	0.3000	0.3000	0.3000	0.3000
1	0.3000	0.5000	0.5000	0.3000	0.3000	0.3000	0.3000
2	0.3000	0.5000	0.5000	0.3000	0.3000	0.3000	0.3000
3	0.3000	0.5000	0.5000	0.3000	0.3000	0.3000	0.3000
4	0.3000	0.5000	0.5000	0.3000	0.3000	0.3000	0.3000
5	0.3000	0.5000	0.5000	0.3000	0.3000	0.3000	0.3000
6	0.3000	0.5000	0.5000	0.3000	0.3000	0.3000	0.3000

Figure 5: The coin example for $\mathbf{y} = \{HHH, TTT, HHH, TTT\}$, with p_1 and p_2 initialized to the same value. EM is stuck at a saddle point.

On final note, when y is hidden, it is also likely that we do not have previous knowledge of the dimension of the alphabet space, or $|\mathcal{Y}|$, such as the number of topics in the Topic Model. The dimension value can be configured by optimizing the model on a development corpus.

Topic Model

Introduction to Topic Model

In this section, we will briefly introduce Topic Model. Consider a simple unigram model where the probability of a word w_i is given by $p(w_i) = \theta_i$. We will also model the topic z such that $z \in \mathcal{Z}$ and $|\mathcal{Z} = k|$, where $\mathcal{Z}$ is our universe of topics and the dimension k is treated as a

"Arts"	"Budgets"	"Children"	"Education"
NEW	MILLION	CHILDREN	SCHOOL
FILM	TAX	WOMEN	STUDENTS
SHOW	PROGRAM	PEOPLE	SCHOOLS
MUSIC	BUDGET	CHILD	EDUCATION
MOVIE	BILLION	YEARS	TEACHERS
PLAY	FEDERAL	FAMILIES	HIGH
MUSICAL	YEAR	WORK	PUBLIC
BEST	SPENDING	PARENTS	TEACHER
ACTOR	NEW	SAYS	BENNETT
FIRST	STATE	FAMILY	MANIGAT
YORK	PLAN	WELFARE	NAMPHY
OPERA	MONEY	MEN	STATE
THEATER	PROGRAMS	PERCENT	PRESIDENT
ACTRESS	GOVERNMENT	CARE	ELEMENTARY
LOVE	CONGRESS	LIFE	HAITI

Figure 6: For each given topic, some words are more likely to appear than others.

hyperparameter. There will be a distribution over topics z given by θ_z, where $\sum_{z \in \mathcal{Z}} \theta_z = 1$. Furthermore, each word in the document will be generated from the multinomial distribution $p(w_i|z) = \theta_{w_i|z}$, where $\sum_{i=1}^{|V|} \theta_{w_i|z} = 1$. Consider the following steps to generate a document

1. Sample a topic z, such that $z \sim \theta_z$
2. **For** $i = 1$ to N:
   Sample words w_i given the topic z, such that $w_i \sim \theta_{w_i|z}$

Then, the likelihood of the document d of size N can be expressed as

$$p(d|\theta) = \sum_{z \in \mathcal{Z}} \theta_z \prod_{i=1}^{N} \theta_{w_i|z}$$

However, this model is problematic because it assumes that the whole document comes from a single topic. To attain a more flexible model, we should allow multiple topics in the same document. This motivation leads us to the *mixture model* where each word can select its own topic.

In this alternative approach, we setup a new model where each word w_i is sampled from each corresponding topic z_i. In other words, every word is a mixture of topics. The sampling procedure will now become

1. **For** $i = 1$ to N:
   Sample a topic z_i, such that $z_i \sim \theta_{z|d}$
   Sample a word w_i given the topic z_i, such that $w_i \sim \theta_{w_i|z}$

The likelihood for the new model is

$$p_(d|\theta) = \prod_{i=1}^{N} \sum_{z \in \mathcal{Z}} \theta_{z|d} \theta_{w_i|z}$$

The distribution over the topics is unique for each document, represented by $\theta_{z|d}$. Across all documents, the word conditional on a given topic is sampled from a single "shared" distribution, represented by $\theta_{w|z}$. It is easy to see that if both the words and topics are fully observed, we can derive the MLE parameters by looking at the empirical distributions

$$\hat{\theta}_{w_i|z} = \frac{\text{count}(w_i, z)}{\text{count}(z)}$$
$$\hat{\theta}_{z|d} = \frac{\text{count}(z)}{N}$$

In application, however, topic information are not given out in the training data and the problem should be treated as a Hidden Variable Model. We leave it to the readers to think about how the EM algorithm can be applied to solve this specific unigram topic model.

6.864 Advanced Natural Language Processing[1]

Lecture 6: Topic Models and Part of Speech (POS) Tagging

29 September 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Simanta Gautam, Brian Sun, Zygimantas Straznickas.

EM for Topic Models

In the last lecture we covered topic models - a way to discover abstract "topics" in documents. In particular, we considered a mixture topic model where topic distributions were unique for each document instead of being common for all documents. The parameters θ of this model are

- $p(\omega|z) = \theta_{\omega|z}$, satisfying $\sum_{\omega \in W} \theta_{\omega|z} = 1$: the probability distribution of all words $\omega \in W$ given a topic z
- $p(z|d) = \theta_{z|d}$, satisfying $\sum_{z=1}^{K} \theta_{z|d} = 1$: the distribution of topics $z \in \{1, \ldots, K\}$ in a document d

In this model, we assume that different words can belong to different topics. The probability of generating a word can be expanded as

$$p(\omega, \theta) = \sum_{z=1}^{K} \theta_{z|d} \theta_{\omega|z}$$

Here the $\theta_{z|d}$ part is document-specific and represents the distribution of topic in document d. The $\theta_{\omega|z}$ part is cross-document, i.e., the conditional distribution of word given topic is assumed to be independent of the document.

We also assume that generating each word is independent of the words surrounding it. Then the probability of generating a document is just the product of probabilities of generating each word in it:

$$p(d, \theta) = \prod_{\omega \in d} p(\omega|\theta) = \prod_{\omega \in d} \sum_{z=1}^{K} \theta_{z|d} \theta_{w|z} \tag{1}$$

Observed topic case

When applying the EM algorithm to a problem, it is often useful to first think about a case where all the variables are actually observed. In this case, the input data would be a set of n documents $d_1, d_2, ..., d_n$. Each document t would contain N_t words together with their associated topics:

$$\begin{array}{cccc} z_{t_1} & z_{t_2} & \ldots & z_{t_{N_t}} \\ \omega_{t_1} & \omega_{t_2} & \ldots & \omega_{t_{N_t}} \end{array}$$

Here each topic z_{t_1} may take value from $\{1, 2, \ldots, K\}$, and it is observed. For notational convenience, let $d_t = \{\omega_{t_1}, \omega_{t_2}, ..., \omega_{t_{N_t}}\}$ and $z_t = \{z_{t_1}, z_{t_2}, ..., z_{t_{N_t}}\}$.

In order to find the maximum likelihood estimate (MLE) of the parameters $\{\theta_{z|t}, \theta_{w|z}\}$ in the model, we need to maximize the likelihood of the model generating the words and topics in the documents. Note that here t is used instead of d as the index of documents. We first provide the log-likelihood of the whole set of input documents (words and topics):

$$\sum_{t=1}^{n} \log P(d_t, z_t|\theta) = \sum_{t=1}^{n} \sum_{i=1}^{N_t} \log \theta_{z_{t_i}|t} \theta_{\omega_{t_i}|z_{t_i}}$$

where the n documents are considered to be independent, and the equality uses (1).

By taking the derivative over θ and let the derivative be zero, we can show that, with observed topics, the MLE of θ can be found by counting the occurrences of topics in our input data. In particular, the MLE of $\hat{\theta}_{z|t}$ is equal to the number of times a topic z has been observed in document t divided by the total number of words in the document:

$$\hat{\theta}_{z|t} = \frac{\text{count}(z, t)}{N_t}$$

where $\text{count}(z, t)$ is the number of times that z is observed in document t.

Similarly, the MLE of $\hat{\theta}_{\omega_i|z}$ can be estimated by counting the number of occurrences of the word ω_i with topic z, divided by the total number of occurrences of the topic z:

$$\hat{\theta}_{\omega_i|z} = \frac{\text{count}(\omega_i, z)}{\sum_{\omega \in W} \text{count}(\omega, z)} = \frac{\text{count}(\omega_i, z)}{\text{count}(z)}$$

where the counts are across all documents.

However, the topics are usually not observed in practice since words in our training data often do not come with topic annotations. This problem can then be treated as a Hidden Variable Model. In this case, we are only given the words in a document, not their topics. Each word can possibly be associated with any of the topics in $\{1, 2, \ldots, K\}$. For example, consider document t:

$$\begin{array}{lllll}
z_{t_1} = 1 & z_{t_2} = 1 & \ldots & \ldots & z_{t_{N_t}} = 1 \\
z_{t_1} = 2 & z_{t_2} = 2 & \ldots & \ldots & z_{t_{N_t}} = 2 \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
z_{t_1} = K & z_{t_2} = K & \ldots & \ldots & z_{t_{N_t}} = K \\
\omega_{t_1} & \omega_{t_2} & \ldots & \ldots & \omega_{t_{N_t}}
\end{array}$$

The topic associated with each word is unobserved and can take any value from $\{1, 2, \ldots, K\}$.

Since the topics are unobserved, the MLE of the parameters $\{\theta_{z|t}, \theta_{w|z}\}$ maximizes the log-likelihood function $\sum_{t=1}^{n} \log p(d_t|\theta)$, where $d_t = \{w_{t_1}, w_{t_2}, \ldots, w_{t_{N_t}}\}$. In this case, the problem of finding the MLE can be solved using the EM algorithm. The details of the EM algorithm for topic model with observed words and unobserved topics are introduced in the following.

Expectation step (E-step)

Consider the kth EM iteration. In the E-step of the EM algorithm, we are given the model parameters $\theta^{(k)}$ that were calculated in the previous iteration (for the first iteration, $\theta^{(1)}$ can be initialized at random). Our goal in the E-step is to calculate the posterior probability of the hidden variable z given the observed w based on $\theta^{(k)}$. The posterior probability can be expressed as

$$p(z|\omega, t, \theta^{(k)}) = \frac{p(z, \omega, t, \theta^{(k)})}{\sum_{z'=1}^{K} p(z', \omega, t, \theta^{(k)})} = \frac{\theta_{z|t}^{(k)} \theta_{\omega|z}^{(k)}}{\sum_{z'=1}^{K} \theta_{z'|t}^{(k)} \theta_{\omega|z'}^{(k)}}$$

The next step is to compute the fractional counts of occurrences using the above posterior probability. The fractional count of a topic $z \in \{1, 2, \ldots, K\}$ in document t is calculated by summing the conditional probabilities of the topic z given each word w_{t_i} in document t:

$$\overline{\text{count}}(z, t) = \sum_{i=t}^{N_t} p(z|\omega_{t_i}, t, \theta^{(k)})$$

Similarly, in order to compute the fractional count of a word ω occurring with the topic z, we go through every occurrence of the word ω in all the documents, and sum the probabilities of the topic z given w for each appearance of (w, z) in the documents. The result can be expressed as

$$\overline{\text{count}}(\omega, z) = \sum_{t=1}^{n} \sum_{i=1}^{N_t} p(z|\omega_{t_i}, t, \theta^{(k)}) \delta(\omega_{t_i}, \omega)$$

where the inner sum goes through all the words in document t, and the outer sum goes through all the documents. Recall that N_t is the number of words in document t, w_{t_i} is the word in position i of document t, and δ is a function returning 1 if and only if its two arguments match:

$$\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

The δ function picks up the positions where the words are w from each document.

Maximization step (M-step)

After the fractional counts are found, the new parameters $\theta^{(k+1)}$ for step $(k+1)$ can be calculated similar to the case where topics z were observed, with discrete counts being replaced by fractional counts:

$$\theta_{z|t}^{(k+1)} = \frac{\overline{\text{count}}(z,t)}{N_t}$$

$$\theta_{\omega|z}^{(k+1)} = \frac{\overline{\text{count}}(\omega,z)}{\sum_{\omega' \in W} \overline{\text{count}}(\omega',z)}$$

After repeating these steps, $\theta^{(k)}$ will eventually converge to a value that (locally) maximizes the log-likelihood $\sum_{t=1}^{n} p(d_t|\theta)$.

LDA model

While the topic model we discussed is simple to understand and implement, it is not often used in practice. In particular, the model assumes that each document has a specific topic probability distribution associated with it and that the distributions of topics are independent - the topics in one document have absolutely no connection with the topics of another. A more useful alternative is the Latent Dirichlet Allocation (LDA) model.

In LDA, it is assumed that the topic distribution in each document is a mixture of several global topic distributions. Then, using the distribution of these mixtures as another parameter, we can express the likelihood of a document by marginalizing over the distributions of the mixtures. Suppose that there are M global topic distributions, the LDA model can be expressed as

$$p(\omega_1,...,\omega_N \,|\, \theta) = \sum_{j=1}^{M} p_j \left[\prod_{i=1}^{N} \underbrace{\sum_z \theta_z^j \theta_{\omega_i|z}}_{p(\omega_i \,|\, \theta)} \right]$$

where θ_z^j is the jth global topic distribution, and p_j is the probability that the document's topics are generated from the jth topic distribution. Note that $\{p_j\}_{j=1}^{M}$ is a "distribution over distribution (of topic)," and in this case, $\{p_j\}_{j=1}^{M}$ is a discrete distribution.

The distribution of global topic distributions does not have to be discrete. In the continuous case, the likelihood can be expressed by integrating over the parameter space:

$$p(\omega_1,...,\omega_N) = \int p(\theta) \left[\prod_{i=1}^{N} \sum_z \theta_z \theta_{\omega_i|z} \right] d\theta$$

where different values of θ specify different global topic distributions.

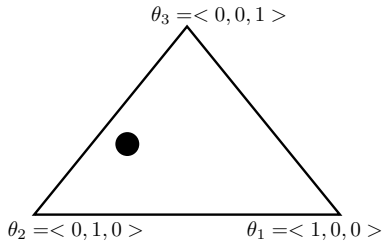


Figure 1: In the LDA model, each document's topic distribution is a mixture of multiple global distributions. In this picture, the combined distribution is mostly a combination of θ_2 and θ_3.

POS Tagging

A **part of speech** categorizes a group of words with similar grammatical properties. Examples of parts of speech labels include noun, verb, adjective, etc., but computational applications often make use of more granular part-of-speech labels, such as the ones defined in Figure 2.

POS tagging is the process of assigning a part-of-speech label to each word in a sentence. More precisely, given sentence $S = (w_1, \ldots, w_n)$ consisting of n words, a POS tagger maps S to tags $T = (t_1, \ldots, t_n)$ by assigning a part-of-speech label t_i to each word w_i. For example, the input sentence below is tagged with POS labels defined in Figure 2.

- *INPUT* : They refuse to permit us to obtain the refuse permit
- *OUTPUT* : They/*PRP* refuse/*VBP* to/*TO* permit/*VB* us/*PRP* to/*TO* obtain/*VB* the/*DT* refuse/*NN* permit/*NN*

tag	definition
PRP	personal pronoun
VBP	verb, non-3rd person singular present
TO	to
VB	verb, base form
DT	determiner
NN	noun, singular or mass

Figure 2: Examples of POS tags with definitions.

A set of POS tags is called **universal** if it contains only tags that exist in similar form across different languages. Otherwise, the set of POS tags is **language specific**.

In this lecture, we motivate the task of POS tagging with applications, address the challenges, and explore several methods for tagging.

Motivation

POS tagging finds applications in many subfields of NLP, often by simplifying or improving the accuracy of various tasks. The following examples are some subfields of NLP that make use of POS tagging.

1. *Text-to-speech*: A single word can have multiple different pronunciations based on its POS tag. For example, the word *record* can be a noun or a verb. When it's a noun, the first two letters are emphasized, but when it's a verb, then the last four letters are emphasized. As elucidated by this example, determining the pronunciation of a word can be improved by knowing its POS label.

2. *Lemmatization*: Lemmatization is the process of grouping together different forms of the same word to its canonical form. For example, *run*, *runs*, *ran*, *running* should all be grouped together under the word *run*. The fact that a single word can have multiple definitions is a source of ambiguity for lemmatization. Knowing the POS tag for the word, however, can remove this ambiguity. For example, the lemmatization of the word *saw* in the verb form involves mapping *saw* to *see*. In the noun form, however, *saw* is mapped to *saw*.

3. *Named-Entity Recognition*: This is a subtask of information extraction that involves tagging words with pre-defined categories such as

names of person, organization, and location. For example, suppose we want to extract the person and organization from the following sentence.

Devin works at Goldman Sachs.

A named-entity recognizer will tag the words in this sentence into those categories, giving the following output.

$$[\text{Devin}]_{\text{Person}} \text{ works at } [\text{Goldman Sachs}]_{\text{Organization}}.$$

The POS tag provides additional information about the word that could be useful in improving the accuracy of this task. Furthermore, the methods used for POS tagging could be applied to named-entity recognition given the similarity of these tasks.

4. *Parsing*: While POS tagging finds applications in many subfields of NLP, perhaps the most popular application is as a pre-processing step for parsing. Tagging each word of a sentence with a part of speech resolves the POS ambiguity, which makes the task of determining an optimal parse tree for the sentence much easier by reducing the number of parses.

Challenges

Due to variations in language and corpus, the size of the set of POS tags can vary from less than a dozen to hundreds. Using a small set of tags is often more difficult because it introduces more ambiguity in the prediction.

Ambiguity arises from the fact that a particular word can be tagged with several different parts of speech. Figure 3 shows an example of a sentence where each word could have several different tags. This is quite common, and serves as a challenge for POS tagging, which consists of tagging each word with a single part of speech.

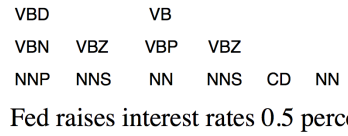


Figure 3: Sentence where the possible part-of-speech labels are shown for each word. A POS tagger must deal with this ambiguity when assigning each word a single label.

The challenge of ambiguity arises not just from a **coarse** set of tags, but also from the nature of the data corpus. For example, POS tagging on a corpus extracted from Twitter is generally more difficult because of unusual conventions such as hashtags, non-standard spellings, abbreviations, and emoticons. To tackle this challenge, one approach could be to expand the set of POS labels to include Twitter-specific tags.

Lastly, **tokenization**, which involves breaking a sentence into words, is especially challenging with a corpus composed of tweets. For example, the phrase *I am going to find* could be written as *imma find* in a tweet, which makes this task difficult. Since tokenization is the first step for POS tagging, the unconventional nature of the corpus would certainly make POS tagging more difficult.

Supervised Learning

We can train a model to predict POS tags to words from a **training set**, with the goal of having this model **generalize** well on unseen sentences, i.e., coming up with good tags to unseen sentences.

The training set consists of sentences that have been tagged by human annotators. We assume some noise exists in these human transcriptions, and set the **upper bound** word accuracy for POS tagging at 98%. First, we assess the accuracy of a simple tagging method based on frequency counts. Then we compare the method to the accuracy of supervised classification models that use word features and context features.

Frequency-Based Tagging

As we saw earlier, a single word could have several possible tags. The **Most Frequent Tag** approach learns the most common tag t_i for each word w_i in the training data. The prediction tag for a word is simply the most common tag for that word, as seen in the training data. In predicting the tag for an unseen word, the method simply assigns it the most common POS tag in the whole training corpus.

While this approach gets a decent accuracy on the seen data, it does not generalize well to unseen data (Figure 4). The reason is that the approach **over-fits** the training data.

Word and Context Features

We can use a **multiclass classifier** that predicts a part of speech label given features that describe the word. With this approach, we can construct **word features**, and learning would consist of appropriately weighting these features to better predict POS tags. Examples of word features are shown in the following table.

word feature	**example**
Prefixes	unfathomable: un- → adjective
Suffixes	surprisingly: -ly → adverb
Capitalization	Meridian: CAP → proper noun

When determining the POS tag of a particular word, it may be useful to also consider the preceding or following words. In particular, **context features** can provide useful information not only about the word, but also about how it fits in with the rest of the sentence. We will discuss these techniques in more detail in the next lecture. In Figure 4, the accuracy of the POS tagging methods discussed so far is shown alongside the upper bound.

Method	Seen word accuracy	Unseen word accuracy
Most Frequent Tag	**90%**	**50%**
Log-linear Model with word features	**93.7%**	**82.6%**
Log-linear Model with context features	**96.6%**	**86.8%**
Upper bound	**98%**	**98%**

Figure 4: Per-word accuracies of supervised POS tagging methods on Seen and Unseen corpora.

Generative PoS model

We can model the tagging problem with a joint probability distribution of words and tags. Intuitively, this allows us to generate pairs of observations and labels. We define a sequence of words and their corresponding tags as follows:

$$S := \{w_1, w_2, \ldots, w_n\}$$

$$T := \{t_1, t_2, \ldots, t_n\}$$

where the number of words is n, and the tags are chosen from a set $\mathcal{T}$.

We wish to find

$$T^* = \arg\max_T(P(S, T))$$

where we find the sequence of tags T^* that gives the highest probability for the observed sequence of words S. In effect, we are defining the tagging problem as a *hidden Markov model*, where the hidden states are the sequence of tags $T = \{t_1, t_2, \ldots, t_n\}$.

The joint probability of S and T is given by:

$$P(S, T) = P(w_1, \ldots, w_n, t_1, \ldots, t_n)$$

By the chain rule we can write,

$$\begin{aligned} P(S, T) &= \prod_{i=1}^{n} P(t_i, w_i | w_1, \ldots, w_{i-1}, t_1, \ldots, t_{i-1}) \\ &= \prod_{i=1}^{n} P(t_i | w_1, \ldots, w_{i-1}, t_1, \ldots, t_{i-1}) P(w_i | w_1, \ldots, w_{i-1}, t_1, \ldots, t_i) \end{aligned}$$

However, in the above expression, the probability of each tag still depends on all preceding words and tags. We can simplify the expression by making Markov assumption. Assuming that the current tag only depends on the previous tag, and that a word depends only on its tag, we have

$$P(S, T) = \prod_{i=1}^{n} P(t_i | t_{i-1}) * P(w_i | t_i)$$

This means that we only need to consider a single preceding tag when calculating the probabilities. We can interpret the above expression as the probability that a tag t_i is followed by a tag t_{i-1} times the probability that a word w_i is labeled by a tag t_i. This is done for each position i and then multiplied together.

The parameters of this model are the probabilities

$$p(u|v) = P(t_i = u | t_{i-1} = v)$$

$$q(w|t) = P(w_i = w | t_i = t)$$

The first one represents the context part - how likely a certain ordering of tags is. For example, in English, a combination *noun-verb* is much more likely than *verb-noun*. The second part represents relationships between words and tags.

In order to find the estimate of these parameters given input data, we can use the counting method. The parameter $p(u|v)$ can be estimated based on the number of times that $(t_{i-1} = v, t_i = u)$ appears, and the number of times $t_i = u$ appears in the document. In particular, the MLEs for bigram and unigram models are combined via linear interpolation to model $p(u|v)$, given by

$$P(u|v) = \lambda_1 \frac{\text{count}(v,u)}{\sum_u \text{count}(v,u)} + \lambda_2 \frac{\text{count}(u)}{n}$$

where the weights λ_1 and λ_2 satisfy $\lambda_1 + \lambda_2 = 1$. The choice of λ_1 and λ_2 can be found by applying the EM algorithm.

Similarly, $P(w|t)$ can be estimated by counting the occurrences of that word with the tag:

$$P(w|t) = \frac{\text{count}(t,w)}{\sum_{w'} \text{count}(t,w')}$$

Viterbi Algorithm

In the previous section, we wanted to find the sequence of tags T that were most likely associated with the given sequence of words S, i.e., to find $\arg\max_T(P(S,T))$. The naive solution is to simply calculate the probabilities of all the possible tag sequences and find the sequence that maximizes it. However, because the number of possible sequences grows exponentially with the length of the sentence n, this solution is almost always impractical.

Instead, we can redefine the problem as a dynamic programming (DP) problem using the Viterbi algorithm - a widely used algorithm for finding the most likely sequence of hidden states of a *hidden Markov model*.

Rethinking the brute-force approach by searching over all tag sequences, we notice that we are recomputing many sequences of log probability combinations. Rather than enumerate sequences, we can store previous computations in a lookup table. In particular, we can store the log probability that a sequence of tags ends at an index i, where i has range $1, \ldots, N$ with a certain tag t.

Given a word sequence S, we can find the most likely tag sequence

by recursively calculating

$$\pi[i,t] = \max_{t_1,\ldots,t_{i-1}} \log p(t_1,\ldots,t_{i-1},t_i = t, w_1, w_2, \ldots, w_i)$$
$$= \max_{t_1,\ldots,t_{i-1}} \left\{ \sum_{j=1}^{i-1} \Big[\log p(t_j|t_{j-1}) + \log p(w_j|t_j) \Big] + \log p(t|t_{i-1}) + \log p(w_i|t) \right\}$$

where i is the last position of a sequence of words that ends with tag t. The values of the table will be the corresponding log probabilities. The index pair (i,t) corresponds to a sequence of words that ends in position i with tag t. The value of $\pi[i,t]$ is the highest log-likelihood for such a sequence.

The table can be computed by applying a recursive algorithm called the Viterbi algorithm

Base Step:

$$\begin{aligned} &\pi[0, < S >] = \log 1 = 0 \\ &\pi[0,t] = \log 0 = -\infty, \; if \; t \neq < S > \end{aligned} \tag{2}$$

where $< S >$ is the start symbol.

Recursive Step:

$$\pi[i,t] = \max_{t'} \{ \pi[i-1,t'] + logP(t|t') + logP(w_i|t) \}$$

The base step is correct because the first word of the sentence has to be the start symbol. The recursive step can be interpreted as follows. Given the maximum log probabilities of sequences ending at position $i-1$ with tag t', we want to find the maximum log probabilities of sentences ending at position i with tag t. In particular, the recursive step can be derived as follows

$$\pi[i,t] = \max_{t_1,\ldots,t_{i-1}} \left\{ \sum_{j=1}^{i-1} \Big[\log p(t_j|t_{j-1}) + \log p(w_j|t_j) \Big] + \log p(t|t_{i-1}) + \log p(w_i|t) \right\}$$
$$= \max_{t'} \left\{ \max_{t_1,\ldots,t_{i-2}} \left\{ \sum_{j=1}^{i-2} \Big[\log p(t_j|t_{j-1}) + \log p(w_j|t_j) \Big] + \log p(t'|t_{i-2}) \right.\right.$$
$$\left.\left. + \log p(w_{i-1}|t') \right\} + \log p(t|t') + \log p(w_i|t) \right\}$$
$$= \max_{t'} \{ \pi[i-1,t'] + \log p(t|t') + \log p(w_i|t) \}$$

In other words, the optimal log-likelihood of tag sequence of length i depends on the optimal log-likelihood of the preceding subsequence of length $(i-1)$. If we had those log-likelihoods precomputed, we could calculate the new likelihood quickly by trying out all the tags

	0 <S>	1 I	2 love	3 big	4 dogs
<S>	0	$-\infty$	$-\infty$		
Noun	$-\infty$	-0.5	-0.12		
Adj	$-\infty$	-0.5	-0.25		
Verb	$-\infty$	-0.5			
Pronoun	$-\infty$	-0.5			

A partially generated table π for a sentence *I love big dogs.* In the next step of the algorithm, the value at $\pi[2, \text{Verb}]$ would be found. In order to do that, we would apply the recursive step: consider all of the values in column 1 of the table, estimate how likely they are to generate "Verb" as the next tag, and find the preceding tag that produces the maximum likelihood.

and choosing the one that maximizes the log-likelihood as in the recursive step. The insight is that we can define the problem as finding the last tag in a subsequence and then iterating forwards from 1 to N since we are only considering immediate predecessors. In other words, in order to find each tag in the tag sequence, we simply ask what is the best tag at index 1, then at index 2, etc. until we have generated the table.

Reconstructing T

There are several ways to reconstruct the actual tag sequence using the precomputed table. One of them is to keep another table with references to which word was chosen at each step and then following those links back. The other option is to just use the table π. We know that the last tag t_N of the optimal sequence is the one with the largest log-likelihood, i.e.,

$$t_N = \arg\max_t \pi[N, t]$$

After computing t_N, we can reconstruct the tag t_{N-1} before it by going through all the possibilities and choosing the one that made us choose t_N as the last tag:

$$t_{N-1} = \arg\max_t \{\pi[N-1, t] + \log P(t_N|t) + \log P(w_N|t_N)\}$$

This process can be repeated each time going one step back, until the first tag of the optimal sequence is reached. While the second method does not require an extra table, using it will cause some calculations to be needlessly repeated.

Performance Analysis

Since we are only storing the table π, the space complexity is $O(N|T|)$ where N is the number of words and T is the set of tags.

When building the table, we have need to compute $O(N|T|)$ values - one for each combination of tag and word position. Each value is computed by searching over $|T|$ tags. Overall, the run time for generating the table π is $O(N|T|^2)$.

6.864 Advanced Natural Language Processing[1]

Lecture 7: Discriminative POS tagging models

1 October 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Tej Chajed, Alexander Forsyth, William Spitzer.

In the previous lecture, we developed a generative Hidden Markov Model for the Part-of-Speech (POS) tagging problem. The goal of the POS tagging problem is to tag each word in a sentence with its part of speech (for example, noun, verb or adjective). The motivation for studying POS tagging is twofold: POS tags are themselves used in other NLP tasks (for example, parsing and machine translation); and, similar approaches are used for other problems, such as named entity recognition and other sequence learning tasks.

Hidden Markov Model (HMM)

We first note that the notation used in this lecture is slightly different than that in the last lecture. The setup of the problem is that we have a sentence of n words $x_1, \ldots, x_i, \ldots, x_n$, with a corresponding list of tags $y_1, \ldots, y_i, \ldots, y_n$. For convenience, we also define $y_0 = \text{START}$, a fixed start tag that every tag sequence begins with. This is a notational convenience so we can say things like $P(y_i \mid y_{i-1})$ for all $1 \leq i \leq n$.

The Hidden Markov Model (HMM) is a joint model over the set of words. It assumes a bigram model over tags so that each tag y_i is allowed to depend on the previous tag y_{i-1}. In addition, each tag y_i is related to a the corresponding word x_i. These two points are captured in the probability over the tags and words.

$$P(y_0 ... y_n, x_1 ... x_n) = \prod_{i=1}^{n} P(y_i \mid y_{i-1}) P(x_i \mid y_i) \tag{1}$$

To simplify the probability, we look at the log-likelihood:

$$\log P(y_0 ... y_n, x_1 ... x_n) = \sum_{i=1}^{n} \log(P(y_i \mid y_{i-1}) P(x_i \mid y_i)) \tag{2}$$

$$= \sum_{i=1}^{n} score(y_i, y_{i-1}) \tag{3}$$

The log likelihood converts the product of pairwise probabilities of the tags into a sum of log-probabilities. This in turn can be represented as a sum of scores where each term depends only on a pair of successive tags. The sequence of most likely tags, i.e., $y_1, \ldots, y_n$ that maximize $\sum_{i=1}^{n} score(y_i, y_{i-1})$ can be computed via the Viterbi algorithm.

Discriminative Models

The previous model is a so-called *generative model* because it gives a probability distribution over the data, modeling enough to (in principle) generate the data. However, one might question why we need to model how the sentences are generated ($P(x)$) since in our use case the sentence is given to us. To that end, we now develop a model taking advantage of the fact that we usually possess all of the words x_1 ... x_n. We will allow the model to take advantage of the whole sentence, rather than just the current word. Such a model is now *discriminative* rather than generative, because it allows us to distinguish between different outputs (tag sequences) for a given sentence and determine the best one, but does not explicitly model how the sentences are generated.

The Discriminative Model (DM) considers sequences of tags in which each pair of tags y_{i-1}, y_i does not relate with just the words x_{i-1} and x_i but potentially the entire sentence $x = x_1, \ldots, x_n$. The advantage of DM compared to HMM is that we can easily use more information from the sentence to model dependencies between the tags. This power doesn't come without a cost. Since DM does not represent $P(x)$, the distribution over the sentences, it also cannot easily deal with missing values (tags, words). At one extreme, if all the tags are missing for some sentences, HMM can still evaluate $P(x)$ which depends on all the parameters in the model, including the transition probabilities. It is therefore possible to estimate a tagger in an unsupervised (or semi-supervised) manner where some sentences are tagged and others are not. This is harder to do with a DM since it does not predict anything about the sentence and thus cannot use x alone to adjust its predictions.

DM specifies a new conditional probability model over the tags

$$P(y_0 ... y_n \mid x) = \prod_{i=1}^{n} P(y_i \mid y_{i-1}, i, x) \tag{4}$$

where, again, we use $x = x_1, \ldots, x_n$ to represent the entire sentence. In this probability, i represents the position in the sentence. We notice that each probability on the right hand side includes x as a parameter which reflects the fact that each pair of tags (y_i, y_{i-1}) can depend on the entire sentence x, not just x_i. We include i because we treat y_i and y_{i-1} as variable names and therefore need to isolate, in particular, the context around x_i that (y_i, y_{i-1}) is most closely associated with. The probabilities *do not* typically depend on the position directly in the sense that a particular tag pair would be more likely at the beginning rather than the end of the sentence. Instead, position i is used to isolate the relevant part of x (x_i, nearby words).

Maximum-Entropy Markov Model (MEMM)

We now move on to specifying the conditional probabilities $P(y_i \mid y_{i-1}, i, x)$. To this end, we introduce features that relate possible values of y_i to the previous tag and to the words in the sentences. Features are indicator functions that take as an input y_i and $h_i = (y_{i-1}, i, x)$, i.e., the available history that the conditional probability over y_i can depend on. The indicator functions yield 1 if the property they check is true and 0 otherwise.

We give some examples of features that are all binary indicator features. In the following, we use some example part-of-speech tags, including DT for determiners ("a", "the"), VBG for verbs in gerund form, and JJ for adjectives. These are standard abbreviations for POS tags from the Penn treebank, which includes hundreds of thousands of manually tagged English sentences.

1. *Bigram Features*: $\phi_k(h_i, y_i) = 1$ if $y_{i-1} = \text{DT}, y_i = \text{JJ}$. We would introduce an indicator feature for each possible successive tags (here (DT, JJ)). Taken together, these features encapsulate the bigram model in the HMM.

2. *Word-Tag Feature*: $\phi_{k'}(h_i, y_i) = 1$ if $x_i =$ "the"$y_i = \text{DT}$, i.e., we have a lexicalized feature of relating the current tag y_i to the word x_i underneath. These are features that the HMM also depends on. Together with the bigram features, the word-tag features lead to a discriminative model that is as expressive as the HMM model.

3. *Prefix-Suffix Feature*: $\phi_{k''}(h_i, y_i) = 1$ if x_i ends in "ing", $y_i = \text{VBG}$. These features use prefixes and suffixes of words to distinguish different forms of words and they relate to the tags. These might be particularly useful for new or rare words for which we might not have introduced (or have rarely seen) a word-tag feature.

4. *Context Features*: $\phi_{k'''}(h_i, y_i) =$ if $x_{i-1} =$ "the", $y_i = \text{JJ}$. These features use the context around x_i to relate to tag y_i. HMM cannot directly model such features.

In our notation $\phi_k(h_i, y_i)$ the subindex k enumerates all instances of the features. For example, the bigram features will have indices $k \in \mathcal{T} \times \mathcal{T}$, where $\mathcal{T}$ is the set of tags. If we include all bigram features, then the first word-tag feature would have subindex $k' = |\mathcal{T}|^2 + 1$, and so on. There are other features that we could include beyond those listed above.

We would like to construct the conditional probability distribution $P(y_i \mid h_i)$ on the basis of these features. One way to do this is to use the *maximum entropy* principle. In other words, we rely on all the information in the features but do not introduce any further dependences not

specified in the features. In this sense, the distribution is made maximally uncertain about y_i above and beyond what cannot be directly controlled with the given features. To this end, we consider all the features as one set $\phi_k(h_i, y_i)$ for $1 \leq k \leq M$, where M is the total number of features (a large number). The generic form of the maximum entropy distribution is:

$$P(y_i \mid h_i, \theta) = P(y_i | y_{i-1}, i, x, \theta) = \frac{e^{\sum_{k=1}^{M} \theta_k \phi_k(h_i, y_i)}}{\sum_{y'} e^{\sum_{k=1}^{M} \theta_k \phi_k(h_i, y_i')}} \tag{5}$$

In this probability distribution, ϕ_k represents each of the features that are being used in this model, θ_k represents the weight given to each feature (how important it is). Note that the resulting probability distribution is the softmax if we set $z_{y_i} = \sum_{k=1}^{M} \theta_k \phi_k(h_i, y_i)$ for each y_i.

The distribution leaves us with adjustable parameters $\theta_1, \ldots, \theta_M$. We can learn the optimal values for these weights by maximizing the log-likelihood of generating the tags in the training set. Since the annotated training set is typically not very large, we will instead maximize the *penalized log-likelihood*, which is the log likelihood minus a *regularization penalty*. The regularization penalty encourages the parameters θ_k to remain small so as to avoid overfitting. For example, without any regularization, if a feature appeared on (that is, took the value 1) for only a single training sentence, then to minimize the log likelihood we could set its weight to $\pm\infty$ depending on whether it would be useful for including or excluding tag settings. With regularization, we need to see the feature multiple times in order to give it higher weight. More formally, we maximize

$$\tilde{l}(\theta) = \sum_{s} \sum_{i=1}^{|s|} \log(P(y_i^s \mid y_{i-1}^s, i, x^s, \theta) - \frac{\lambda}{2} \sum_{k} \theta_k^2 \tag{6}$$

where the sum over is over all sentences s in the training corpus and positions within the sentences. The $\frac{\lambda}{2} \sum_k \theta_k^2$ is the regularization penalty term; λ determines how much we penalize complex feature weights. We can set the best value of λ based on the validation set.

Structured Prediction

The goal of tagging is to produce the correct sequence of tags for each sentence. The models we have discussed so far, HMM and the maximum entropy discriminative model (DM), can each give a (log-likelihood) score for each possible sequence of tags for the given sentence. The most likely sequence of tags according to the model is then obtained by maximizing this score (Viterbi). However, neither model is actually trained to produce the correct sequence. For example, the

DM was trained to correctly predict the next tag given the previous one (and the sentence). These goals are related but not the same.

In *structured prediction*, we formulate the problem such that we directly predict the full sequence of tags rather than its parts. In order to do this, we define a scoring function over sequences of tags and directly learn to adjust the parameters in this scoring function. Specifically, we define

$$score(y_1 ... y_n) = \sum_{i=1}^{n} \sum_{k=1}^{M} \theta_k \phi_k(y_i, h_i) \tag{7}$$

which is just the sum of scores used to define the conditional probabilities in our DM. But there's no normalization constant. We will use this scoring function as a way of ranking sequences of tags. If we re-arrange the terms a bit

$$score(y_1 ... y_n) = \sum_{k=1}^{M} \theta_k (\sum_{i=1}^{n} \phi_k(y_i, h_i)) \tag{8}$$

we see that the inner sum is simply the count of how often the feature is active along the sentence for the given tag sequence. Giving it a name, we substitute $\Phi_k(x, y) = \sum_{i=1}^{n} \phi_k(y_i, h_i)$ so that

$$score(y_1 ... y_n) = \sum_{k=1}^{M} \theta_k \Phi_k(x, y) \tag{9}$$

$$= \theta \cdot \Phi(x, y) \tag{10}$$

Now, the score for a sequence of tags is simply the dot product of the parameter vector θ and the vector of feature counts $\Phi(x, y)$.

How do we train the parameters θ? Our criterion is that the scoring function always gives the highest score for the correct tag sequence as opposed to any alternative. If $y^* = (y_1^*, \ldots, y_n^*)$ is the correct tag sequence for sentence x, then we would like to find parameters θ such that

$$\theta \cdot \Phi(x, y^*) > \theta \cdot \Phi(x, y),\ y \neq y^*$$

The same parameters should work for all the sentences and correct tag sequences in our training corpus. If these conditions hold then clearly

$$y^* = \arg\max_y\ \theta \cdot \Phi(x, y)$$

i.e., we would predict the correct sequence of tags.

Let's assume for a moment that there exists some parameter vector θ that works for the training corpus. In order for this to be true, we should have specified rich enough collection of features (introduced large numbers of parameters θ_k). Under this assumption, a simple *perceptron* algorithm will find one possible setting of the parameters

that lead to correct predictions. The algorithm works as follows. We cycle through pairs of sentence and target tag sequences (x, y) in the training corpus $(x^i, y^i), i = 1, \ldots, N$. If we make an error, i.e., if

$$y \neq \hat{y} = \arg\max_{y'} \theta \cdot \Phi(x, y')$$

then we update the parameters $\theta \leftarrow \theta + \Phi(x, y) - \Phi(x, \hat{y})$. In other words, we nudge the parameters in the direction of the count vector corresponding to the correct tag sequence, and subtract the counts related to the incorrect prediction. The algorithm is guaranteed to converge (no mistakes) after a finite number of runs through the training corpus.

If there is no parameter vector θ that works for all the sentences in the training corpus, the algorithm clearly will not converge. Instead, it will continue to correct the parameters one way or another (correcting one mistake, introducing another). In this case, the parameter vector θ will continue to bounce around some reasonable "mean" value. By averaging the parameters as we cycle through the training examples (whether we update or not), we obtain this mean after just a small number of runs through the training corpus. The resulting algorithm is called the *averaged perceptron algorithm* and works quite well even in this *unrealizable* case (no perfect parameters exist).

6.864 Advanced Natural Language Processing[1]

Lecture 8: Recurrent Neural Networks

6 October 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Joseph Kim, Michael Traub, Samira Bazuzi.

In this lecture we introduce recurrent neural networks (RNNs) for modeling sequences with longer term dependencies. In n-gram feed-forward language models the distribution over the next work depends on a fixed pre-determined number of previous steps. While hidden Markov Models can store longer term information in the underlying hidden state, the discrete representation of the state is often limiting. RNNs maintain a continuous vector representation of the state and can, in principle, store complex persistent information about the history. But they are challenging to learn with simple gradient methods.

Review of FeedForward Neural Network for Language Modeling

In Figure 1 we see an illustration of a neural network that takes w^{t-1} as input and provides a distribution over the possible words w^t. The vector $\mathcal{V}(w)$ representing each word in the input layer could be a $|V|$ dimensional one hot vector, some pre-processed vector in $\mathbb{R}^d$, or a vector representation that we will learn as part of the language model. Each output unit is associated with a word and encodes $P(w^t = w|w^{t-1})$ after the softmax. As a result, the model is no more powerful than a bi-gram language model. However, the fact that it employs potentially dense low dimensional word vectors as inputs (rather than words directly) can help it generalize better than a standard bi-gram model.

It is useful to briefly consider the role of word vector dimension d in this simple feed-forward model. If $d < m$, the input to hidden layer weights linearly expand each word to m dimensions. Increasing d would permit us to make use of the extra dimensions in the hidden layer to (potentially) better differentiate between similar words. However, when $d > m$, the input to hidden layer weights would linearly project word vectors back to m dimensions. The resulting model is no more powerful than one with exactly m dimensional learnable word vectors.

Recurrent Neural Networks for Language Modeling

Recurrent neural networks maintain a continuous representation of the state that is fed back into the model (in some manner). Figure 2 illustrates one technique where we append the hidden layer activations directly back in as another input vector. In particular, when we try to find the probability distribution for w^t, we take as inputs both

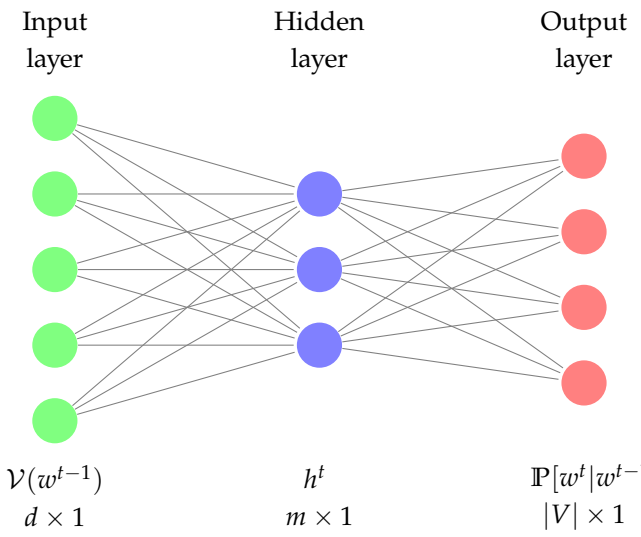


Figure 1: Illustration of a feedforward neural network with one hidden layer that tries to predict the next word based on the previous word. The input layer is a d dimensional vector representing the previous word, the hidden layer takes a weighted combination of the input components and applies a non-linear transform, possibly sigmoid. The output layer takes a weighted combination of this state to construct a score for each word in the vocabulary. After a soft max is applied, this produces a distribution over the vocabulary of the next word w^t. Below each layer the name of the vector and its dimension are included.

This architecture makes our output distribution conditional only on the previous word. In order to change this, we have to include the word vectors for a few more of the previous words.

the word vector $\mathcal{V}(w^{t-1})$ and the hidden state vector h^{t-1} from the previous step. Since h^{t-1} was updated on the basis of $\mathcal{V}(w^{t-2})$, the prediction for w^t depends on w_{t-2} as well. Going back further, unraveling how h^{t-2} depends on w_{t-3}, and so on, we see that the distribution for w^t is adjusted based on the entire history of previous words $w^1, w^2, \ldots, w^{t-1}$. The RNN output layer at time t therefore represents an estimate of $p(w^t|w^1, \ldots, w^{t-1})$. See the unravelled model in Figure 3

Learning RNNs

We can try to learn recurrent models with stochastic gradient ascent analogously to feed-forward models. The only difference is that the back-propagation step that evaluates the gradients with respect to the parameters now must proceed backwards in time to model the impact of the parameters on the hidden states along the earlier steps as well. Perhaps the easiest way to understand how this works is to look at the unravelled model in Figure 3. This is just a feed-forward neural network with the difference that the parameters mediating the transformations from one layer to another are shared across the time steps. But the gradient ascent works the same. During back-propagation, you can think of having different parameters at each time point (as you would in a feed-forward model), just that they are set to be the same. When updating the shared parameters, we simply add the gradients that correspond to the same parameters. Note that, for clarity, we have omitted the output distributions for the intermediate words in the figure, pretending that the only prediction happens at the end.

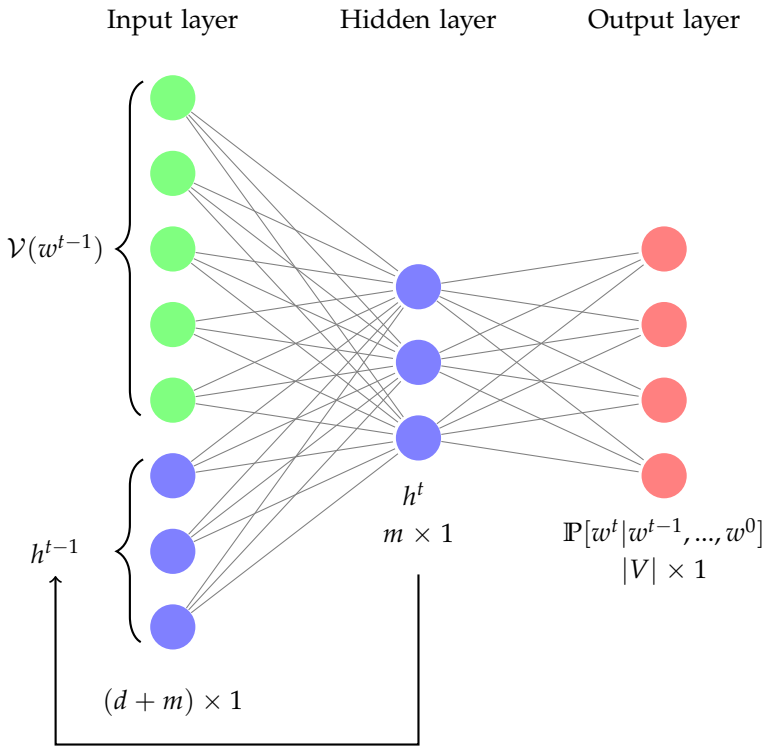


Figure 2: **Recurent Neural Network:** We have taken the network in Figure 1 and identified the hidden layer h^t as the ***state*** at time t. This network takes the activation values of that layer at $t-1$, and appends them to the input vector to predict t. This makes our prediction of t dependent on all previous inputs, instead of just the few in the recent past like an n-gram model.

Varnishing and Exploding Gradients

Here are the equations that describe a RNN with a single hidden layer as in Figure 2, where $f(x)$ is a non-linear function that could be a hinge or sigmoid function. For simplicity, we omit the offset parameters.

$$x_i^t = \begin{cases} \mathcal{V}(w^{t-1})_i & \text{if } i \in [1,d] \\ h_{i-d}^{t-1} & \text{if } i \in [d+1, m+d] \end{cases}$$

$$h_j^t = f\left(\sum_{i=1}^{d+m} W_{ij}^h x_i^t\right) \qquad z_k = \sum_{j=1}^{m} W_{jk}^o h_j^t$$

$$P\left(w^t = k | w^0, ..., w^{t-1}\right) = \frac{e^{z_k}}{\sum_{k'=1}^{|V|} e^{z_{k'}}}$$

Let's try to unravel these equations to see the recursion more clearly:

$$\begin{aligned} h_j^t &= f\left(\sum_{i=1}^{m+d} W_{ij}^h x_i^t\right) \\ &= f\left(\sum_{i=1}^{d} W_{ij}^h \mathcal{V}(w^{t-1})_i + \sum_{i=d+1}^{d+m} W_{ij}^h h_{i-d}^{t-1}\right) \end{aligned}$$

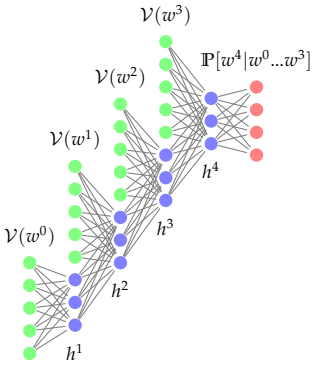


Figure 3: **Unravelled RNN:** When we determine the gradient update of a RNN, we can imagine unravelling back in time to make an extended feed forward net. The difference is that the connections are all governed by the same weight matrix W_{ij}. The derivatives are all executed the same way, but we run into a chain rule mess. Also notice no nodes for h^0 are included. In general you can include h^0 as some initial state, but in this discussion and in general this initial state is set so $h^0 = \vec{0}$.

We can continue to expand this until we get to h^0. However, what's more important than unraveling the exact function—which is highly nonlinear —is finding the gradient. Let's consider the gradient step for one of the weights. For $i \in [1, d]$,

$$\frac{\partial}{\partial W^h_{ij}} h^t_j = f'\left(\sum_{i'=1}^{d+m} W^h_{i'j} x^t_{i'}\right)\left[\mathcal{V}(w^{t-1})_i + \sum_{i'=d+1}^{d+m} W^h_{i'j} \frac{\partial}{\partial W^h_{ij}} h^{t-1}_{i'-d}\right].$$

We eventually get to the base case that $\frac{\partial}{\partial W^h_{ij}} h^0_{i'} = 0$. The key point to note is that as we calculate the gradient one step back-wards, the terms are multiplied with weights $W_{i'j}$ (and eventually other weights $W_{i'j'}$ as well). If the weights are small (as in the beginning of learning), the gradient that travels back will quickly vanish beyond a few steps. Note that this does NOT mean that the RNN wouldn't learn anything. It means that it wouldn't learn how weights impact the state several steps into the future. It would still learn fine the immediate effects from observations at the same (or around the same) time point. It is this balance between short term and long term information that is referred to as the vanishing gradient issue. The gradients can also explode if the weights are large. This can happen later during learning.

Note: the i' and τ' indicate that the the sum is not related to any i that has been "instantiated" outside that sum. These index the same vectors, and in general if we sum over all i', then one of them will equal i. We could use a notation with a completely different letter instead of priming the original, but this hides that we are indexing over the same vector.

These vanishing and/or exploding gradient phenomena make it challenging to learn RNNs. However, there have been architectures that are designed to circumvent these problems. One example is LSTM, which we will see soon.

Applications to Sentence Tagging

Now we try to predict a sequence of tags, like "noun", "verb", et al., for a sequence of words. These tags will have transition probabilities, so we want our model to be able to take previous tags into account.

We might consider doing the same as in Figure 2, but predict labels instead of the next word. However, the resulting architecture will produce a model of the tags y^t's in the form of $P(y^t|w^0, ...w^t)$. As a result, the dependency of y^t on previous tags is not captured. Instead, we would prefer a structure that produce a model in the form of $P(y^t|w^0, ..., w^t, y^0, ..., y^{t-1})$, so that we can use all previous information for predicting y^t. We can accomplish this with the architecture in Figure 4.

Here we sample $\hat{y}^t$ from $P(y^t = y|w^0, ..., w^t, y^0, ..., y^{t-1})$ in order to induce a distribution over a sequence of tags (different runs yield different sequences). A simple deterministic setting $\hat{y}^t = \arg\max_y P(y^t = y|w^0, ..., w^t, y^0, ..., y^{t-1})$ is myopic and may lead to lower probability sequences after a few steps. The sampled tag is then converted to $\widetilde{\mathcal{V}}(\hat{y}^t)$ with dimension $\tilde{d}$, and appended it to the input for predicting the next

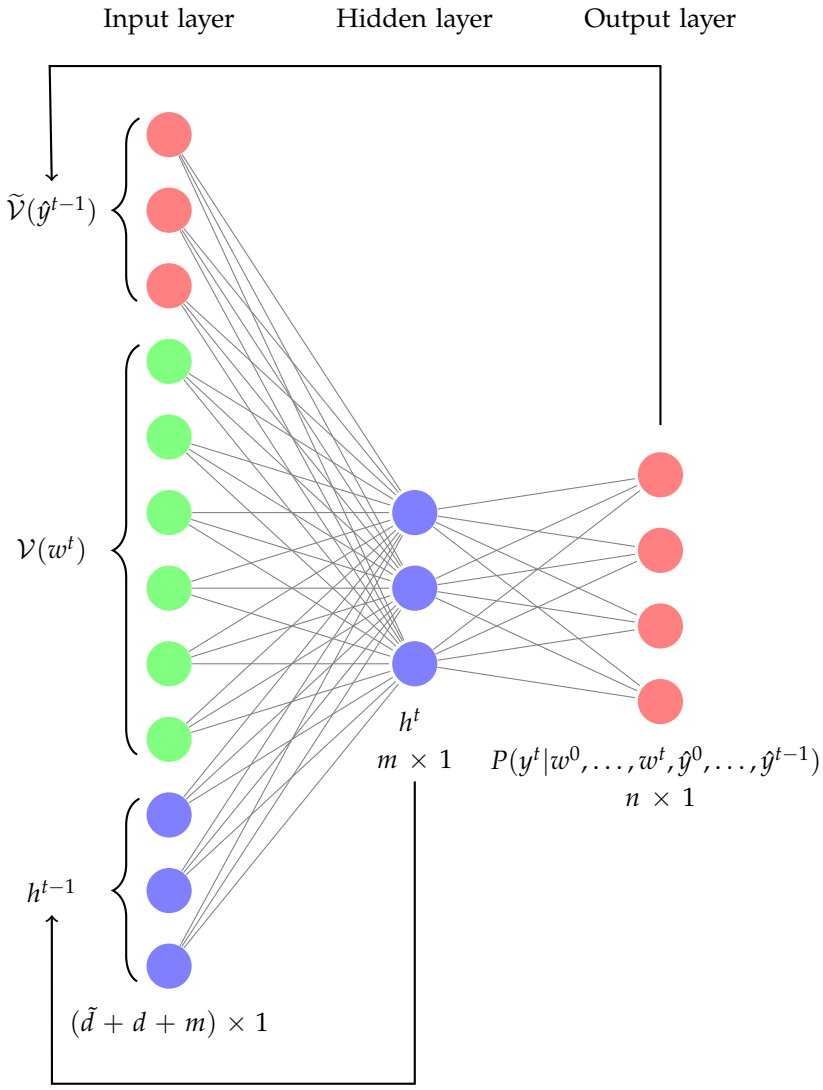
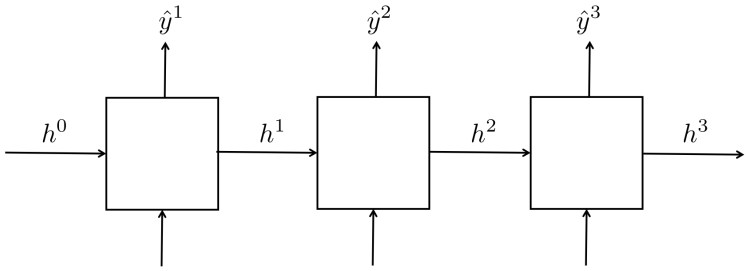


Figure 4: **RNN with outputs fed back in:** This architecture allows future predictions to depend on the previous predicted or observed labels. We can see how this makes our final distribution dependent on all the inputs and outputs in sentence tagging. The vector $\widetilde{\mathcal{V}}(y)$ is a vector representation of a label y and is separate from $\mathcal{V}(w)$. The label of the previous word y^{t-1} may be known (training), in which case it can be given as an input instead of the predicted label. During testing, a predicted (sampled) $\hat{y}^{t-1}$ is used instead.

label. We also could use the actual y^t if it is available to us (as in during training). By concatenating the architecture in Figure 4, the RNN for tagging is illustrated in Figure .



Long Short-Term Memory (LSTM)

To tackle the problem of vanishing/exploding gradients, a model known as Long Short-Term Memory (LSTM) has been developed. LSTM models are used to capture the following notions:

- What and when to forget (Forget gate)
- What and when to read from the input (Input gate)
- What and when to output for the next time step (Output/Exposure gate).

Intuitively, LSTM attempts to find balance between not capturing dependency information and too much dependency on irrelevant information.

For all of these *gates*, we will be using the sigmoid activation function $\sigma(z) = \frac{1}{1+e^{-z}}$ (computed element-wise). We can gain intuition of an LSTM model by going through each of the following stages:

1. **New memory cell generation.** A new memory cell, $\widetilde{c}^t$ is generated given the new input word x^t and the past hidden state h^{t-1}. This *summarizes* the new word with respect to the contextual past.

$$\widetilde{c}^t = \tanh(W^c x^t + U^c h^{t-1} + W_o^c) \tag{1}$$

   x^t here is the current word vector of size d x 1. h^{t-1} is of size m x 1. W^c (m x d), U^c (m x m), and W_o^c (m x 1) are parameters associated with the new memory generation stage.

2. **Input gate.** New memory generation stage does not check if the new word is important (or which aspects of it are important). This

choice is controlled by the input gate. It produces i^t (a vector of coordinate-wise gates), which serves as an indicator whether or not the input coordinate is worth reading.

$$i^t = \sigma(W^i x^t + U^i h^{t-1} + W_o^i) \tag{2}$$

Note that W^i (m x d), U^i (m x m), and W_o^i (m x 1) are parameters specifically associated with the input gate.

3. **Forget gate.** Similar to the input gate, instead of dealing with the current *word*, it assesses whether or not the past memory cell is useful or should be erased. Similarly to the input gate, we combine f^t element-wise with the past memory cell vector c^{t-1}. With the addition of new memory $\tilde{c}^t$, we get the final memory cell for the current time step c^t (Equation 4).

$$f^t = \sigma(W^f x^t + U^f h^{t-1} + W_o^f) \tag{3}$$

$$c^t = f^t \circ c^{t-1} + i^t \circ \tilde{c}^t \tag{4}$$

4. **Output gate.** The output gate controls how much of the information in the final memory cell c^t should be saved in the hidden state.

$$o^t = \sigma(W^o x^t + U^o h^{t-1} + W_o^o) \tag{5}$$

$$h^t = o^t \circ c^t \tag{6}$$

Note that the parameters W, U, and W_o are different across gate types. Since $\sigma(z)$ does not output exactly 0 or 1, you may still have a bit of vanishing gradients problem. But the time horizon is much improved over the simple RNN.

6.864 Advanced Natural Language Processing[1]

Lecture 9: Parsing and Probabilistic Context Free Grammar

8 October 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Clare Liu, Evan Pu, Kalki Seksaria

This lecture first discusses the evaluation of and the limitations of Hidden Markov Models for unsupervised part-of-speech (POS) tagging. The lecture then introduces the concepts of parse trees, context-free grammars (CFGs), and probabilistic context-free grammars (PCFGs). The CYK algorithm is then presented as an efficient method that uses dynamic programming to find the best parse tree for a sentence with a PCFG. Finally, the limitations of probabilistic context-free grammars are discussed and evaluated.

Evaluation of Unsupervised POS tagging

In the previous lecture we have shown how to solve the part-of-speech (POS) tagging problem by using a Hidden Markov Model (HMM). The hidden states of the HMM are supposed to correspond to parts of speech. However, the model is not given any linguistic information. Furthermore, we do not have the correspondence between numeric hidden states and actual POS tags, such as "noun" or "verb". To evaluate the accuracy of the HMM, we can use a matching algorithm, which attempts to construct a correspondence between hidden states and POS tags that maximizes the POS tagging accuracy. The goal of this matching is to see what POS each hidden state corresponds to.

Let's assume that the HMM has two possible hidden states, y_1 and y_2, and that there are three possible POS tags, N, V, and ADJ. Consider the two example sentences "Strong stocks are good" and "Big weak stocks are bad" (Figure 1). The HMM has assigned a hidden state (y_1 or y_2) to each word. In addition, we also know the correct POS tag for each word. Our goal is to use a matching algorithm to match each hidden state to the POS tag that would give us the highest score.

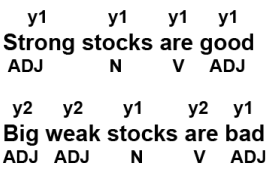


Figure 1: Here are two example sentences annotated with the hidden states generated by an HMM above and the gold-standard POS tags below. We want to find a matching between the hidden states and POS tags to maximize the number of words tagged correctly.

Matching

A matching is a bipartite graph between part-of-speech tags and hidden states. Two types of matchings can be performed: a 1-to-many matching or a 1-to-1 matching. The definitions of 1-to-many and 1-to-1 matchings are given below. Both types of matchings have different advantages and disadvantages when they are used to evaluate the performance of an unsupervised model.

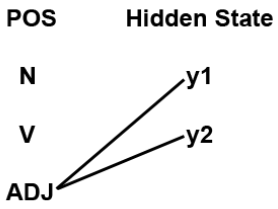


Figure 2: Example of a 1-to-many matching. Multiple hidden states are matched to the ADJ tag.

1-to-Many Matching In a 1-to-many matching, each hidden state is assigned to with the POS tag that gives the most correct matches. Each

POS tag can be matched to multiple hidden states, hence the name "1-to-many".

1-to-1 Matching In contrast, in a 1-to-1 matching, each POS tag can only be matched to a maximum of one hidden state. In this case, if the number of hidden states is greater than the number of tags, then some hidden states will not correspond to any tag. This will decrease the reported accuracy.

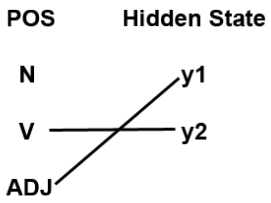


Figure 3: Example of a 1-to-1 matching. Each part-of-speech tag has a maximum of one hidden state that is matched with it.

Evaluation

Given a matching, the accuracy of the HMM tagging can be computed against a gold-standard corpus, where each word has been assigned the correct POS tag (based on the Penn WSJ Treebank). The correct tag for a word is compared with the tag that is matched with the word's hidden state. If these two tags match, this means that the tagging is correct for this word.

To find a matching, we can create a table that contains the counts of each correct POS tag for each hidden state. Here is an example evaluation table for the example sentences from Figure 1.

Hidden State	POS Tag	Count
y1	N	2
y1	V	1
y1	ADJ	3
y2	N	0
y2	V	1
y2	ADJ	2

Figure 4: An example evaluation table for the sentences from Figure 1. State y_1 has been matched with 2 nouns, 1 verb, and 3 adjectives. State y_2 has been matched with 0 nouns, 1 verb, and 2 adjectives.

The matching between hidden states and POS tags can then be found based on this table for both 1-to-many and 1-to-1 matchings.

Finding a good matching

As we can see, the accuracy of the HMM depends on the type of matching that is chosen (1-to-many or 1-to-1). To find a good matching, we can use a greedy algorithm that chooses individual state to tag matchings one at a time and picks the new matching that maximally increases the accuracy at each step. The algorithm terminates when no more matchings can be made. In a 1-to-1 matching, the algorithm terminates when either each POS tag has been assigned a hidden state or each hidden state has received a POS tag (whichever happens first). In a 1-to-many matching, the algorithm terminates when each hidden state has received a POS tag.

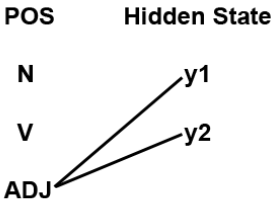


Figure 5: The matching that we get from maximizing the scores from the one-to-many evaluation table.

Example: 1-to-many Matching Consider the table from Figure 4. The highest count we see in the table is 3, which corresponds to y_1 being assigned the ADJ tag. Thus, we greedily assign state y_1 to ADJ. Since the highest count for y_2 is also ADJ, state y_2 is also assigned to the ADJ tag. y_1 tags 3 words correctly, y_2 tags 2 words correctly, and there are 9 words in total, so the best score from a 1-to-many matching is $\frac{5}{9}$. The bipartite matching is illustrated in Figure 5.

Hidden State	POS Tag	Count
y1	N	2
y1	V	1
y1	**ADJ**	**3**
y2	N	0
y2	V	1
y2	**ADJ**	**2**

Figure 6: This table corresponds to a 1-to-many matching. As we can see, ADJ is assigned to multiple hidden states because it leads to the highest score for both states

Example: 1-to-1 Matching We know that the highest count for y_1 is 3, which corresponds to an adjective, so we again greedily assign state y_1 to ADJ. Even though state y_2 also has the highest count for adjectives, ADJ has already been matched to state y_1 so we cannot use it again. Thus, y_2 must be assigned to V, which has the highest count that is not yet assigned to any state. y_1 tags 3 words correctly and y_2 tags 1 word correctly, so the total score for a 1-to-1 matching is $\frac{4}{9}$. See Figure 8 for the matching.

Hidden State	POS Tag	Count
y1	N	2
y1	V	1
y1	**ADJ**	**3**
y2	N	0
y2	**V**	**1**
y2	ADJ	2

Figure 7: This table corresponds to a 1-to-1 matching. Even though choosing ADJ would give the highest score for state y_2, we cannot repeat POS tags, so it must be assigned to V instead.

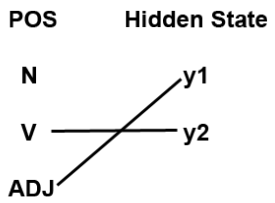


Figure 8: The matching that we get from maximizing the scores from the 1-to-one evaluation table.

Pitfalls of the HMM

Fully unsupervised tagging models generally perform very poorly. The HMM computed with the EM algorithm tends to give a relatively uniform distribution of hidden states, while the empirical distribution for POS tags is highly skewed since some tags are much more common than others. As a result, any 1-to-1 matching will give a POS tag distribution that is relatively uniform, which results in a low accuracy score.

A 1-to-many matching can create a non-uniform distribution by matching a single POS tag with many hidden states. However, as the

number of hidden states increases, the model could overfit by giving each word in the vocabulary its own hidden state. If each word has its own hidden state, we would achieve 100 percent accuracy, but the model would not give us any useful information. For more details on the evaluation of HMM for POS tagging, see "Why doesn't EM find good HMM POS-taggers?" (hyperlinked) by Mark Johnson.

Improving Unsupervised Models

There are several modifications we can make to improve the accuracy of unsupervised models:

1. Supply the model with a dictionary and limit possibilities for each word: For example, we know that the word "train" can only be a verb or a noun. This limits the set of possible tags for each word, which will push the model in the correct direction.

2. Prototypes: Provide each POS tag with several representative words for the tag. Even just a few words for each tag will help push the model in the right direction.

Parse Trees

We now move from POS tagging to sentence parsing. A parse tree is a rooted tree that represents the syntactic structure of a sentence according a grammar. In syntactic parsing, each word in a sentence is tagged with a POS tag, and groups of POS tags and/or phrases are labeled with a single phrase tag. For example, the root of a parse tree is the tag "S", describing the whole sentence. The sentence "S" is commonly broken into an "NP" (noun phrase) and "VP" (verb phrase). The lowest level of the parse tree consists of the words of the sentence. We will now informally describe several applications of parse trees in NLP. Figure 9 gives an example of a syntactic parse tree.

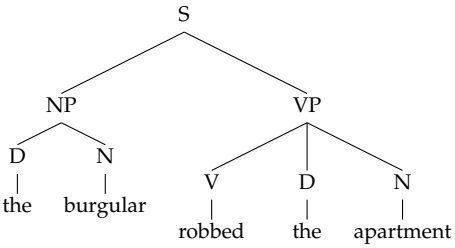


Figure 9: An example of a parse tree for the sentence "The burgular robbed the apartment"

Applications of Parse Trees

Grammar Checking Syntactic parse trees can be used for grammar checking/correction. If a sentence cannot be parsed against a given grammar, this could indicate a grammar error. Grammar suggestions can also be generated by comparing the sentence against the given grammar.

Machine Translation Parse trees can also be used in machine translation, where a source language is first parsed against the source lan-

guage's grammar, and the resulting tree is transformed to produce a sentence in the target language.

Semantic Role Labeling In semantic role labeling, a sentence is decomposed into a *predicate*, which represents a key concept of a sentence, and *arguments*, which take on different roles according to the predicate. For example, in the sentence "Mary sold the book to John", the verb "sold" is the predicate. The arguments are as follows: "Mary" is the seller, "the book" is the good, and "John" is the recipient. Note the sentence "The book was sold by Mary to John", while syntactically different, has the same semantic role labeling as before. Figure 10 shows another example tree for semantic role labeling.

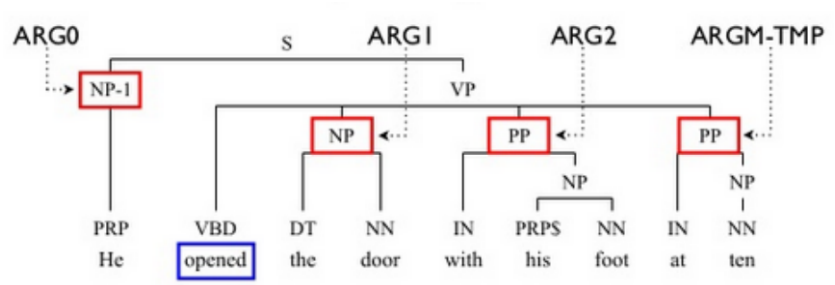


Figure 10: An example of Semantic Role Labeling, where the word "opened" is labeled as the predicate.

Discourse Parsing In discourse parsing, groups of related sentences, which are called discourses, are parsed as a group. This reveals the structure of a piece of text, and also shows us the relationships between the different sentences. The techniques used for discourse parsing are similar to those used for syntactic parsing. While words form the leaves of a syntactic tree, whole phrases or sentences form the leaves of a rhetorical structure tree. Figure 11 shows an example rhetorical structure tree for discourse parsing.

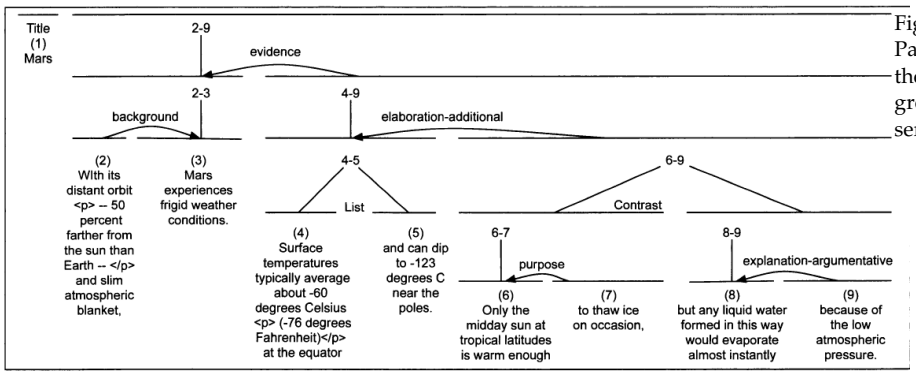


Figure 11: An example of Discourse Parsing that explains the structure of the given text. Sentence 2 acts as background for sentence 3, and sentences 4-9 serve as evidence for sentences 2-3.

Information Extraction In information extraction, structured information is extracted from a sentence. Information extraction can be used to populate a database directly from sentences instead of having to manually fill in the required fields. Information extraction is used to produce calendar notifications and contact cards directly from email text. Figure 12 shows an example tree for information extraction.

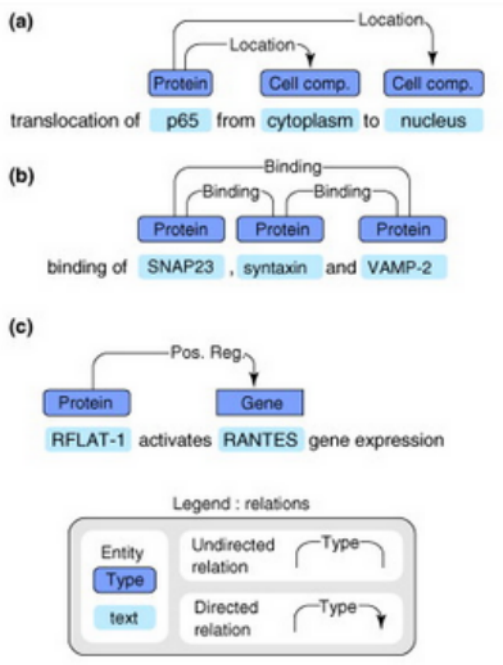


Figure 12: An example of information extraction where in a), the parse tree annotates that cytoplasm and nucleus are possible locations for the protein p65.

Datasets for Training Syntactic Parse Trees

In order to train a syntactic parser, annotated training data is required. Datasets for training parse trees include:

1. The Penn WSJ (Wall Street Journal) Treebank is a collection of 50,000 sentences with their associated trees. It is commonly used in training and test sets for parsing. However, the Penn WSJ Treebank only contains English sentences, so it cannot be used for other languages. Furthermore, the WSJ does not contain the full range of possible English sentences, as it is a newspaper focused on business. There is some overfitting in training and testing a parser on WSJ data only.

2. Universal Dependencies is a more recent data source for multilanguage parsing. It is available at `https://universaldependencies.github.io/docs/`.

Context-Free Grammar

In order to generate a parse tree, a grammar is required. In order to reduce the problem complexity and save computation time, We will make the assumption that the grammar is context-free. This assumption is very good for English, as no counter-examples have been found. In addition, across languages, only a few rare constructions violate this assumption. A context-free grammar (CFG) contains a set of symbols and rules. Each symbol can be either terminal or nonterminal. Terminal symbols correspond to words in the vocabulary and nonterminal symbols correspond either to parts of speech or to groupings of words, such as noun phrases and verb phrases. Each rule maps a nonterminal symbol to a set of nonterminal or terminal symbols. A CFG formalizes how a string (sentence) may be generated from its rules. The application of these rules forms a tree structure, which is the syntax tree (parse tree) for the string. In this section we will only discuss the generation of strings for a given CFG. The inverse operation, parsing, which takes in a string and produces a parse tree for the string, will be discussed later.

Formal Definition

A context-free grammar is defined by $G = (\mathcal{N}, \Sigma, R, S)$ where:

$\mathcal{N}$ = set of nonterminal symbols

Σ = set of terminal symbols

R = set of production rules of the form $X \rightarrow Y_1 ... Y_n$, where X is a nonterminal symbol and each Y_i can be a terminal or a nonterminal symbol.

S = starting symbol

Example Context-Free Grammar

$\mathcal{N} = \{S, NP, VP, PP, DT, VI, VT, NN, IN\}$

$S = S$

$\Sigma =$ {sleeps, saw, man, woman, telescope, the, with, in}

Generation of Strings with a CFG

We can generate strings with symbols in the terminal set Σ from a CFG by using a *left-most derivation* strategy. Here is a non-deterministic algorithm describing the left-most derivation strategy, where the input A is a symbol, and the output is a string:

$R =$

S ⇒ NP VP
VP ⇒ VI
VP ⇒ VT NP
VP ⇒ VP PP
NP ⇒ DT NN
NP ⇒ NP PP
PP ⇒ IN NP

VI ⇒ sleeps
VT ⇒ saw
NN ⇒ man
NN ⇒ woman
NN ⇒ telescope
DT ⇒ the
IN ⇒ with
IN ⇒ in

Figure 13: The rules for the example context-free grammar

```
left_most_derivation(A):
     if terminal(A):
         return A
     rule = choose (rules(A))
     rhs = right_hand_side(rule)
     return concatenate([left_most_derivation(A') for A' in rhs])
```

If the symbol is a terminal, the algorithm terminates. If the symbol is non-terminal, the non-deterministic operator *chooses* one rule where the left-hand-side is A. The algorithm then recursively expands the list of symbols on the right-hand-side, starting with the left-most symbol and concatenates the result. //

To generate a string from a CFG, we call this algorithm on the starting symbol S, which represents an entire sentence. The non-deterministic derivations used in generating the string will form the parse tree of the string.

Language of a CFG

Given a CFG, one can define the language generated by the CFG as follows: $L = \{s \in \Sigma^* | s = left\_most\_derivation(S)\}$ where Σ^* is the set of strings formed by terminals in Σ. In other words, a string belongs to the language of a CFG if there exists a sequence of left-most derivations that can generate the string.

Example of left-most derivation

We can visualize how the grammar defined earlier generates the sentence "The woman saw the man with the telescope." in Figures 14-16, along with their parse trees.

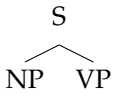


Figure 14: First, the starting symbol S is expanded into NP and VP.

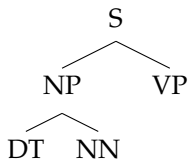


Figure 15: Next, *S*'s left child, NP, is expanded into DT and NN.

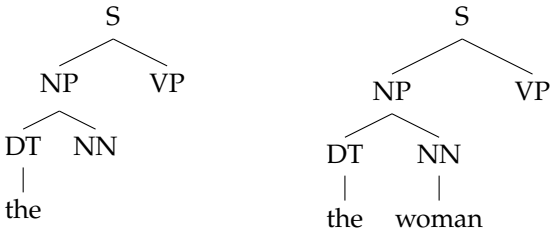


Figure 16: Next, the left child, DT, is expanded into "the", which is a terminal. Then we recursively go back to NP and expand its right child (NN), which gives us the terminal symbol "woman". Since the right side of NP has reached a terminal, we have now completed the left-most derivation for the symbol NP. The next step is to expand the right child of *S*.

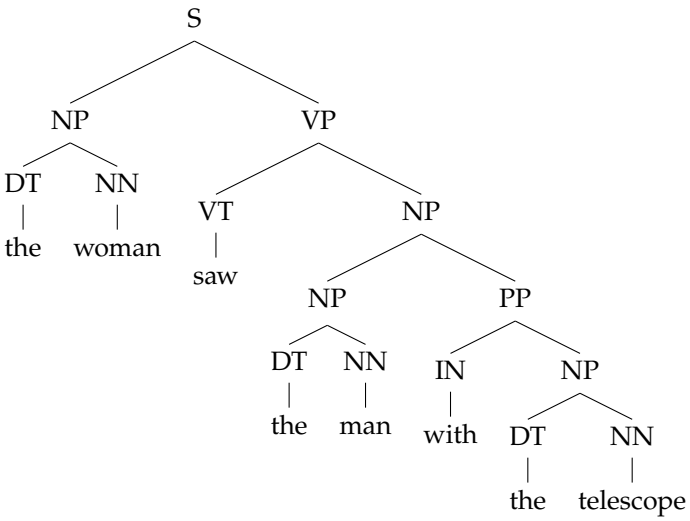


Figure 17: By completing this process of left-most derivation, we obtain a full parse tree. This tree shows one possible derivation and the resulting parse tree for the sentence "The woman saw the man with the telescope". Under this parsing, we interpret the sentence as stating that *the man who the woman saw had a telescope*.

Ambiguities

Given a CFG, there may be multiple possible derivations using the left-most-derivation strategy that generate the same string, but give different parse trees. These sentences have an ambiguous meaning, as each parse tree represents a different meaning. Figure 18 shows another way of parsing the example sentence "The woman saw the man with the telescope". Ambiguities are troublesome because

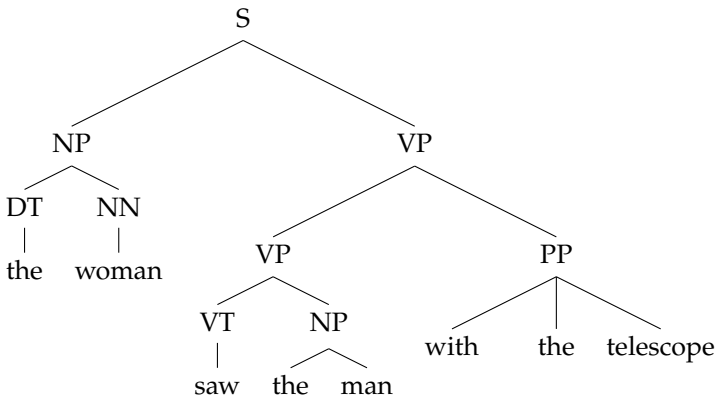


Figure 18: Another possible parse tree. Under this parsing, the woman is using a telescope to see the man.

one would like to have a way of preferring one parse tree over another. We can partially address this issue by augmenting the rules in R with probabilities, making certain derivations more likely than others.

Probabilistic Context-Free Grammar

A Probablistic Context Free Grammar (PCFG) is a CFG where each rule is augmented with a conditional probability of using that rule given the left-hand-side non-terminal. The conditional probabilities are written as $P(\alpha \rightarrow \beta|\alpha)$, where α is a non-terminal, and β is the right-hand-side derivation of the rule. Given a non-terminal, all the probabilities of the rules that apply to this non-terminal must add up to 1, i.e. $\sum_\beta P(\alpha \rightarrow \beta|\alpha) = 1$. This forms a probability distribution over the possible derivations.

Estimating Probabilities in PCFG

One way of learning these probabilities is to use a gold-standard corpus such as the Penn Treebank, and use the empirical distributions of the derivations. For example, if we have the rule $S \rightarrow NP\ VP$, the estimated conditional probability is equal to:

$$P(S \rightarrow NP\ VP|S) = \frac{count(S \rightarrow NP\ VP)}{count(S)}.$$

PCFG as a Language Model

The probability that a particular tree T (representing sentence s) is generated from the PCFG can be calculated by finding the product of the probabilities of all of the rules used in the tree:

$$P(T, s) = \Pi_i P(\alpha_i \rightarrow \beta_i | \alpha_i)$$

Furthermore, the probability of the sentence s being generated is equal to the sum of the probabilities of all of its possible parse trees:

$$P(s) = \sum_{T \in T(s)} P(T, s)$$

where $T(s)$ is the set of all parse trees that can generate s.

PCFG Example:

Here is a PCFG where we augmented the CFG of the previous example with conditional probabilities.

S ⇒ NP VP	1.0
VP ⇒ VI	0.4
VP ⇒ VT NP	0.4
VP ⇒ VP PP	0.2
NP ⇒ DT NN	0.3
NP ⇒ NP PP	0.7
PP ⇒ IN NP	1.0

VI ⇒ sleeps	1.0
VT ⇒ saw	1.0
NN ⇒ man	0.7
NN ⇒ woman	0.2
NN ⇒ telescope	0.1
DT ⇒ the	1.0
IN ⇒ with	0.5
IN ⇒ in	0.5

Figure 19: The rules for the context-free grammar from Figure 13, with added conditional probabilities

We can now determine the probabilities of the two ambiguous parse trees for the sentence "The woman saw the man with the telescope".

Tree 1:

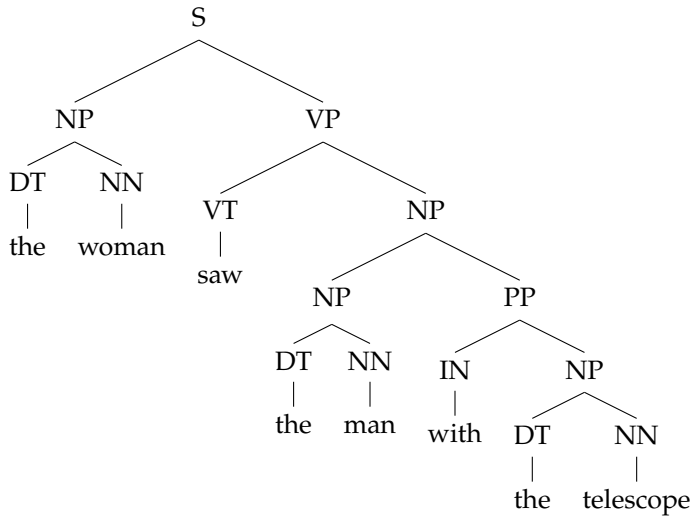


Figure 20: The parse tree from Figure 17 with probabilities for the derivations.

$$P(T_1, S) = 1 \cdot 0.3 \cdot 1 \cdot 0.2 \cdot 0.4 \cdot 1 \cdot 0.7 \cdot 0.3 \cdot 1 \cdot 0.7 \cdot 1 \cdot 1 \cdot 0.3 \cdot 1 \cdot 0.1 \approx 1 \times 10^{-4}$$

Tree 2:

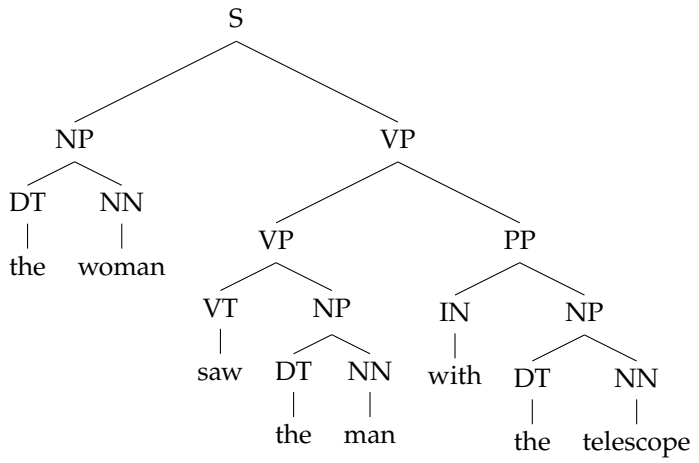


Figure 21: The parse tree from Figure 18 with probabilities for the derivations.

$$P(T_2, S) = 1 \cdot 0.3 \cdot 1 \cdot 0.2 \cdot 0.2 \cdot 0.4 \cdot 1 \cdot 0.3 \cdot 1 \cdot 0.7 \cdot 1 \cdot 1 \cdot 0.3 \cdot 1 \cdot 0.1 \approx 3 \times 10^{-5}$$

We see that the first parse tree is more likely under this PCFG. However, to compute the probability of the sentence "The woman saw the man with the telescope", we need to sum over all the possible parses of it, which can quickly become intractable as there can be an exponential number of parses. In the next section we resolve this issue with the CYK algorithm.

The CYK Algorithm

In this section, we explain the CYK parsing algorithm, that can achieve the following tasks:

1. Language modeling: Given a sentence s, find the probability of the PCFG generating the sentence. Formally:

$$P(s) = \sum_{T \in T(s)} P(T, s)$$

2. Syntactic parsing: Given a sentence s, find the parse tree T that derives the sentence with the highest probability. Formally:

$$T^* = argmax_{T \in T(s)} P(T, s)$$

We will describe the CYK algorithm that solves the first task in detail, and show a simple modification that allows it to solve the second task.

To find the probability of generating a particular sentence s given the PCFG, we calculate the probability of every possible derivation $T \in T(s)$, and sum them:

$$P(s) = \sum_{T \in T(s)} P(T, s)$$

The CYK algorithm solves this summation over a set of exponential size using Dynamic Programming. However, it requires the CFG to be in Chomsky Normal Form (CNF).

Chomsky Normal Form

Chomsky Normal Form restricts the set of allowable rules. A grammar is in Chomsky Normal Form if each rule either converts a nonterminal symbol into two nonterminal symbols or a single terminal symbol:

$$X \to Y_1 Y_2$$

$$X \to y$$

where $X, Y_1, Y_2 \in N$ and $y \in \Sigma$.

It can be shown that every CFG can be converted to CNF. See `http://www.cs.nyu.edu/courses/fall07/V22.0453-001/cnf.pdf` for the details. For example, the rule $NP \rightarrow DT\ ADJ\ NN$ can be converted into the following two rules that satisfy CNF:

$$NP \rightarrow DT\ ADJP$$

$$ADJP \rightarrow ADJ\ NN$$

The Algorithm

The CYK algorithm is a dynamic programming algorithm. Like any other dynamic programming algorithm, it has 2 components: The sub-problems, and the recurrence relation between the sub-problems.

Given a sentence s, let $s[i, j]$ denote the sub-sentence starting at the ith word and ending at the jth word, inclusive.

Sub-problems

The sub-problems of the CYK algorithm are of the form:

$$\pi(i, j, N)$$

Where i and j are the indexes of the words in the sentence, given $i \leq j$, and N is a non-terminal. The value of $\pi(i, j, N)$ corresponds to the probability that $s[i, j]$ can be generated by the non-terminal N.

Recurrence Relations

The recurrence relations of the CYK algorithm are defined by the base case and the inductive case.

Base Case: $\pi(i, i, N) = P(N \rightarrow w_i | N)$

Inductive Case:

$$\pi(i, j, N) = \sum_{k,P,Q} P(N \rightarrow P\ Q | N) \cdot \pi(i, k, P) \cdot \pi(k+1, j, Q)$$

where $k \in \{i, \ldots, j-1\}$, $P \in \mathcal{N}$, and $Q \in \mathcal{N}$.

The base case gives the probability where a non-terminal N would generate the word w_i, which forms the sub-sentence $s[i, i]$.

To understand the recurrence, notice we're trying to find the value for $\pi(i, j, N)$, i.e. the probability which $s[i, j]$ can be generated by N.

We can find this probability by sub-dividing $s[i,j]$ at index k into two pieces: $s[i,k]$ and $s[k+1,j]$, and pick two non-terminals, P and Q to generate each piece. Notice the importance of Chomsky Normal Form, because each rule has to produce exactly 2 non-terminals, this is the only form of decomposition we have to consider. The probability of P generating $s[i,k]$ is $\pi(i,k,P)$, and the probability of Q to generate $s[k+1,j]$ is $\pi(k+1,j,Q)$. Therefore, given a particular index k to split, and a particular P and Q to generate each split sub-sentence, the probability of this particular derivation for $s[i,j]$ is $P(N \rightarrow P\ Q|N) \cdot \pi(i,k,P) \cdot \pi(k+1,j,Q)$. This decomposition can be visualized in Figure 22.

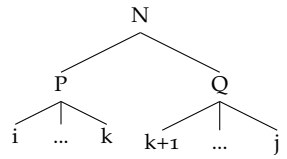


Figure 22: N generates two other non-terminals, P and Q. P spans words i through k and Q spans words $k+1$ through j. At this point, the CYK algorithm has already calculated the total probability of the trees generated by P and Q covering their respective sub-sentences. We can use these values to find the total probability of the tree generated by N.

By summing k, P, and Q over their respective domains, we obtain the total probability of generating $s[i,j]$ by non-terminal N.

The result of the CYK parser is $\pi[1,n,S]$, which is the probability that the start symbol S generates the whole sentence s. This value will be 0 if the sentence cannot be generated by our PCFG.

All that's left is to define an ordering to compute the sub-problems, for the CYK algorithm, we can order it by sub-sentence length, r.

```
# base case
for i in 1..n:
  for N in 𝒩:
    pi[i,i,N] = Pr(N->w_i|N)

# induction
for r in 2..n:
  for i in 1..(n-r):
    for N in 𝒩:
      j = i + r - 1
      pi[i,j,N] = 0

      # over all possible decompositions
      for k in i..j-1:
        for P in 𝒩:
          for Q in 𝒩:
            pi[i,j,N] += Pr(N->PQ|N)*pi[i,k,P]*pi[k+1,j,Q]

return pi[1,n,S]
```

We can also use the CYK algorithm to find the most probable parse tree for a given string. This is the syntactic parsing problem and can be solved by replacing the sum operator in the inductive case of the CYK algorithm with the max operator.

$$\pi(i,j,N) = \max_{k,P,Q} Pr(N \rightarrow P\ Q|N) \cdot \pi(i,k,P) \cdot \pi(k+1,j,Q)$$

where $k \in \{i, \ldots, j-1\}$, $P \in \mathcal{N}$, and $Q \in \mathcal{N}$. The reasoning for correctness is analogous.

Example: Running CYK with a simple PCFG

To illustrate the computation steps for CYK, let's consider a simple artificial PCFG. Consider the following example PCFG:

$\mathcal{N} = \{A, B\}$

$\Sigma = \{a, b, c\}$

$S = \{A\}$

$R =$

$A \rightarrow AB$	0.8
$A \rightarrow a$	0.2
$B \rightarrow BB$	0.7
$B \rightarrow b$	0.1
$B \rightarrow c$	0.2

Figure 23: The grammar rules of the example PCFG.

Now we will compute the probability that the string *abc* is generated by the given grammar:

Base Case:

$$\pi[1,1,A] = P(A \rightarrow a|A) = 0.2$$
$$\pi[1,1,B] = P(B \rightarrow a|B) = 0$$
$$\pi[2,2,A] = P(A \rightarrow b|A) = 0$$
$$\pi[2,2,B] = P(B \rightarrow b|A) = 0.1$$
$$\pi[3,3,A] = P(A \rightarrow c|A) = 0$$
$$\pi[3,3,B] = P(B \rightarrow c|A) = 0.2$$

Recursive Case:

$$\begin{aligned}\pi[1,2,A] =& P(A \rightarrow AA) \cdot P(A \rightarrow a|A) \cdot P(A \rightarrow b|A) \\ &+ P(A \rightarrow AB) \cdot P(A \rightarrow a|A) \cdot P(B \rightarrow b|B) \\ &+ P(A \rightarrow BA) \cdot P(B \rightarrow a|B) \cdot P(A \rightarrow b|A) \\ &+ P(A \rightarrow BB) \cdot P(B \rightarrow a|B) \cdot P(B \rightarrow b|B) \\ =& 0 + P(A \rightarrow AB) \cdot \pi[1,1,A] \cdot \pi[2,2,B] + 0 + 0 \\ =& 0.8 \cdot 0.2 \cdot 0.1 = 0.016\end{aligned}$$

Using similar calculations, we obtain:

$$\pi[1,2,B] = 0$$

$$\pi[2,3,A] = 0$$
$$\pi[2,3,B] = 0.014$$

Solution:

$$\begin{aligned}\pi[1,3,A] =& P(A \to AB) \cdot P(A \to a|A) \cdot P(B \to bc|B) \\ &+ P(A \to AB) \cdot P(A \to ab|A) \cdot P(B \to c|B) \\ =& P(A \to AB) \cdot \pi[1,1,A] \cdot \pi[2,3,B] + P(A \to AB) \cdot \pi[1,2,A] \cdot \pi[3,3,B] \\ =& (0.8 \cdot 0.2 \cdot 0.014) + (0.8 \cdot 0.016 \cdot 0.2) \\ =& \mathbf{0.0048}\end{aligned}$$

Some components of the sum are omitted as they must be 0 since A can only go to AB or a.

Weaknesses of PCFGs

PCFGs do not always provide accurate probability estimates for sentences. Two major weaknesses are:

1. Lack of sensitivity to lexical information
2. Lack of sensitivity to structural frequency

Lack of Sensitivity to Lexical Information

Consider the verb phrase: "drove down the street in the car". Below are two possible parse trees for this phrase:

a)

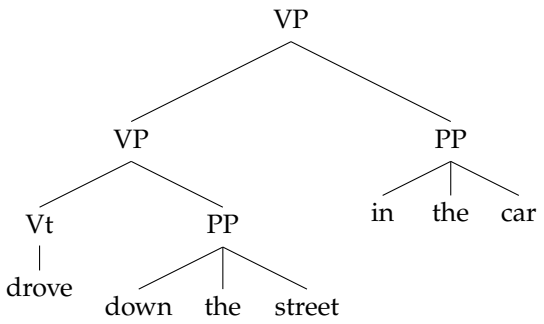


Figure 24: Parse tree a) for the verb phrase "drove down the street in the car". Here, the car is (logically) driving down the street.

A human can easily see that the phrase groupings in parse tree a) make much more sense than those in parse tree b). However, a PCFG considers each nonterminal expansion independently without taking into account the positions and semantics of other words and phrases in the sentence. Therefore, if $P(NP \to NP\ PP|NP) > P(VP \to VP\ PP|VP)$, then the second tree will have a higher probability according to the PCFG. The PCFG does not know that cars move on streets and streets should not be inside cars. This type of information is termed lexical information.

b)

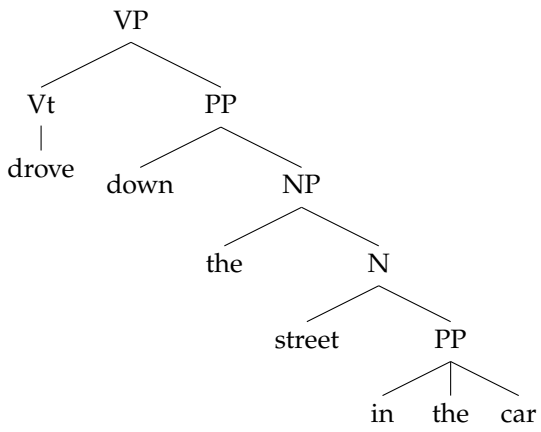


Figure 25: Parse tree b) for the verb phrase "drove down the street in the car". Here, the street is somehow inside the car.

Lack of Sensitivity to Structural Frequency

Another problem of PCFGs is the lack of sensitivity to structural frequency. Consider the following parse trees for the noun phrase "president of a company in Africa":

a)

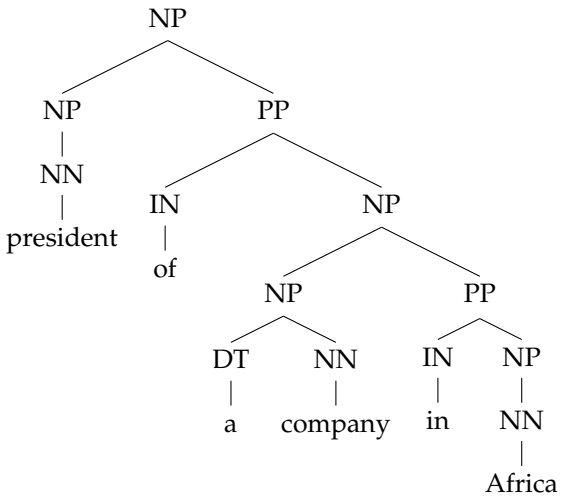


Figure 26: This parse tree has close attachment because "in Africa" describes the company and not the president.

b)

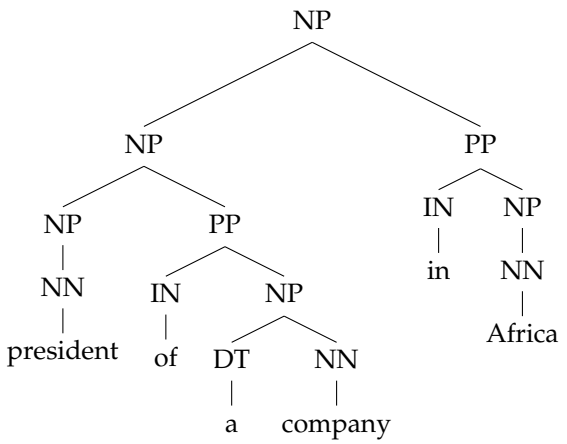


Figure 27: This parse tree does not have close attachment because "in Africa" describes the president and not the company.

As both parse trees use the same rules, they have the same probability under a PCFG. However, it has been shown that the "close attachment" interpretation occurs more often in Wall Street Journal (WSJ) texts. Therefore, the PCFG incorrectly assigns the same probabilities to both trees, even though tree a) should be assigned a higher probability. This error is also due to the PCFG's decisions to optimize at the edge-level rather than at the tree-level. In the next lecture, we will see how to construct better parsers that correct the errors made by PCFGs.

Rules
NP → NP PP
NP → NN
NP → DT NN
PP → IN NP
NN → president
NN → company
NN → Africa
IN → of
IN → in
DT → a

Figure 28: The rules used in trees a) and b). Both trees use the same rules. Therefore, they have the same probability under a PCFG.

6.864 Advanced Natural Language Processing[1]

Lecture 10: Dependency Parsing

15 October 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Surya Bhupatiraju, Sunoo Park, Amy Zhang.

Enhancing Probabilistic Context-Free Grammars

In the last lecture, we discussed *constituency parsing* for probabilistic context-free grammars (PCFGs). In this paradigm, the probability of a given (sentence, parse tree) pair is as follows:

$$p(S,T) = \prod_{i=1}^{m} p(\alpha_i \to \beta_i | \alpha_i) \tag{1}$$

where m is the number of transitions in the tree T, and $\alpha_i \to \beta_i$ is the i^{th} transition in T. Given a corpus of sentences annotated with parse trees, we can easily compute the maximum likelihood estimate (MLE):

$$p_{ML}(\alpha \to \beta | \alpha) = \frac{\text{count}(\alpha \to \beta)}{\sum_{\beta'} \text{count}(\alpha \to \beta')} \tag{2}$$

However, we concluded the last lecture by observing a number of shortcomings of PCFGs, which stem from the fact that they do not take into account *contextual information*,[2] but instead model production rules locally (i.e. one derivation step at a time). In particular, we observe the following two limitations of PCFGs:

[2] Indeed, this is precisely the reason for the term *context-free*.

1. Lack of sensitivity to lexical information
2. Lack of sensitivity to structural frequency

Some natural approaches to address the above limitations are:

1. Add lexical information to the tree (e.g. by labeling each node with associated words)
2. Look at bigger portions of the tree at a time (e.g. subtrees)

In fact, there are problems associated with applying these techniques naïvely: for example, a lexically annotated transition or a subtree will occur much more rarely than a simple unannotated transition, so *sparsity of data* becomes an issue. Another problem that can arise is *computational complexity*: computing probability estimates on larger and/or more complex tree components can be computationally much more intensive than standard PCFG computations.

Today's state-of-the-art parsers are based on PCFGs but with some modifications that are designed to take into account certain contextual information, while keeping data sparsity and computational complexity manageable. We now proceed to some detailed examples.

Lexicalized Trees

In PCFGs, the leaves of a parse tree are *lexicalized*, i.e. they are associated with *words* from the sentence. However, higher (non-leaf) nodes of a parse tree carry no lexical information.

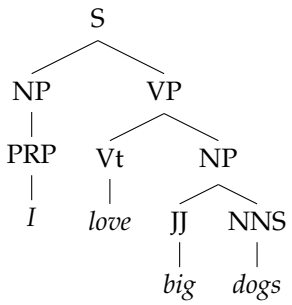


Figure 1: Example of a PCFG parse tree. Throughout these scribe notes, we use the standard part-of-speech tags from the Penn Treebank Project.

In order to address the lack of sensitivity of PCFGs to lexical information, it is desirable to *propagate* lexical information up through the parse tree. That is, nodes corresponding to *nonterminals* of the context-free grammar should also be associated with lexical information.

Recall that production rules in Chomsky normal form map a nonterminal to either two nonterminals (like NP $\rightarrow$ NP VP), or one terminal symbol (like JJ $\rightarrow$ *big*). For the latter type of production rule, we can simply annotate the nonterminal with the associated terminal symbol, as shown in Figure 2.

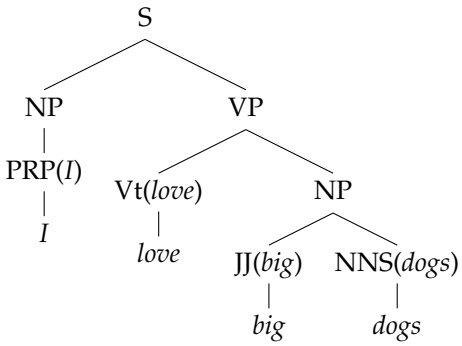


Figure 2: A parse tree partially annotated with lexical information.

However, when annotating higher levels of the tree, it is necessary to deal with production rules that map a nonterminal to two nonterminal. Since we would like to annotate each node with just one word,[3] and each child node is annotated with one word, we need to decide which child node will propagate its annotation to the parent.

Intuitively, we would like the more *meaningful* or *important* word to be the one that is propagated to the parent. For example, the more meaningful word from the phrase "big dogs" seems to be "dogs", rather than "big". More formally, the *head* of a phrase is defined to be the most meaningful word from that phrase, and there are a set of standard deterministic rules by which the head of any sequence of

[3] Annotating each node with many words would have several disadvantages. One disadvantage is sparsity: our parser is unlikely to encounter multiple subtrees containing exactly the same set of words. Another disadvantage is exponential growth: if we annotate each node with the words associated with all of its children, then the number of words per node would double at each level of the tree.

nonterminals can be decided. For example, the **bold** component is the head of the following pairs of nonterminals:

- JJ **NNS** (e.g. "big **dogs**")
- DT **NN** (e.g. "the **poodle**")
- **VP** NN (e.g. "**eats** cake")
- PRP **VP** (e.g. "he **laughs**")
- **VP** RB (e.g. "**laugh** loudly")
- IN **NN** (e.g. "in **class**")

In the examples given so far, it is quite unambiguous which word is the more meaningful one, and this is usually the case. However, in certain cases, it may not be clear which word is the head. For example, which word in the phrase "could see" is the head? In this example, both words seem important. The convention in such cases is to follow the standard set of deterministic rules for identifying the head of a phrase, and it turns out that parsers will perform well as long as the rules are applied consistently across such ambiguous cases.

Figure 3 shows a parse tree that is fully annotated with lexical information by propagating up the head at each level of the tree. Each node receives its *headword* from its *head child*. Note that each node is also annotated with the part-of-speech tag associated with its headword.

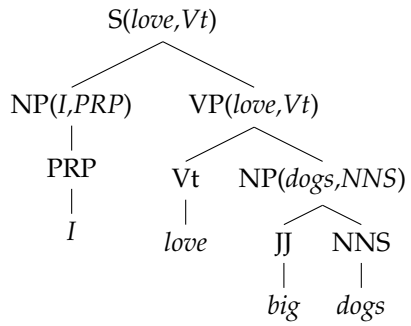


Figure 3: A parse tree annotated with lexical information

With this annotated tree representation, if we try to derive maximum likelihood estimates based on all the available information, then data sparsity will be a problem. While the rule $NP \rightarrow NP\ VP$ is likely to have occurred many times in our training data set, the *lexicalized rule* NP(*dogs*) $\rightarrow$ JJ(*big*) NNS(*dogs*) is far less likely to occur in the training corpus. To deal with this problem, we model the sequence of production rules as a Markov process where each production is conditioned on the head annotation of the starting node.

Here is an example of a production rule in our lexicalized grammar:

$$\text{S}(told,V) \rightarrow \text{NP}(yesterday,NN)\ \text{NP}(Hillary,NNP)\ \text{VP}(\underline{told},V) \qquad (3)$$

This production rule corresponds to the top-level derivation of a sentence beginning with "Yesterday Hillary told...". The nonterminal on the left-hand side of the rule (in this example, S(*told*,*V*)) is called the *parent*. A single nonterminal on the right-hand side of the rule (in the example, VP(*told*,*V*)) is designated as the *head* of the rule. Each nonterminal to the left or right of the head is called a *left-modifier* or *right-modifier* (respectively) of the rule: in rule (3), there are two left-modifiers and no right-modifiers. In contrast, rule (4) exhibits two right-modifiers and no left-modifiers. In other cases, rules can have both left-modifiers and right-modifiers.

$$\text{VP}(told,V) \rightarrow \text{V}(\underline{told},V)\ \text{NP}(Hillary,NNP)\ \text{SBAR}(that,COMP) \quad (4)$$

We now show how to derive probabilities based on this model, using the "Yesterday Hillary told..." sentence as an example. The parse tree corresponding to rule (3) is given in Figure 4.

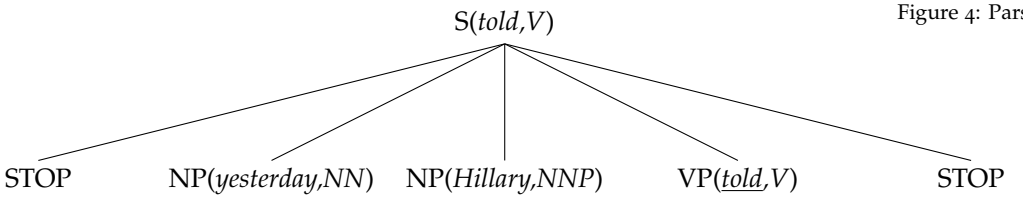


Figure 4: Parse tree for rule (3).

The Markov process proceeds as follows. First, the head node is generated; then the left-modifiers (starting from the left neighbor of the head child, and proceeding leftward); and finally the right-modifiers (starting from the right neighbor of the head child, and proceeding rightward). A special nonterminal (STOP) is introduced into the model so that the Markov process generates left-modifiers until STOP is produced (and likewise for right-modifiers).

Thus, the probability p_h of the head child in this model depends on the parent node only. The probabilities p_d of dependent (i.e. non-head) children are modeled as depending on the parent node, the head child, and the side (i.e. left or right) of the head node on which the dependent is located.

Returning to our example sentence: the Markov process generates the parse tree shown in Figure 4 in five steps, which are depicted in Figures 5 through 9. In each figure, the most recently generated nonterminal is shown in red.

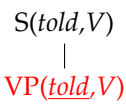


Figure 5: Step 1 of the Markov process that generates the parse tree of Figure 4.

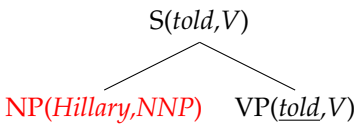


Figure 6: Step 2 of the Markov process that generates the parse tree of Figure 4.

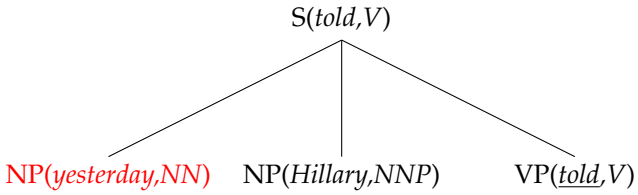


Figure 7: Step 3 of the Markov process that generates the parse tree of Figure 4.

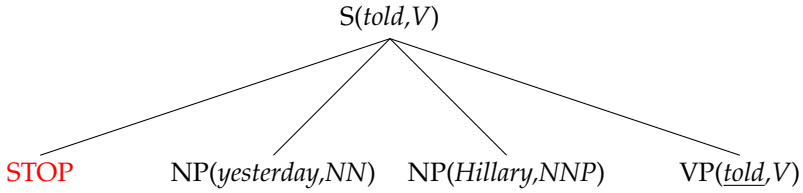


Figure 8: Step 4 of the Markov process that generates the parse tree of Figure 4.

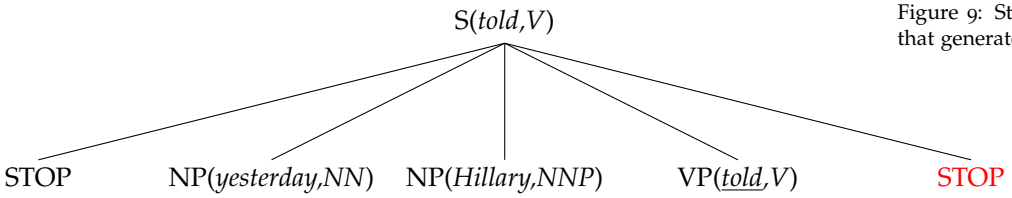


Figure 9: Step 5 of the Markov process that generates the parse tree of Figure 4.

The probability of the sequence of transitions depicted in Figures 5 through 9 is given by the following expression:

$$\begin{aligned} p_h(\text{VP}|\text{S}, \textit{told}, \text{V}) &\cdot p_d(\text{NP}(\textit{Hillary,NNP})|\text{S}, \text{VP}, \textit{told}, \text{V}, \text{left}) \\ &\cdot p_d(\text{NP}(\textit{yesterday,NN})|\text{S}, \text{VP}, \textit{told}, \text{V}, \text{left}) \\ &\cdot p_d(\text{STOP}|\text{S}, \text{VP}, \textit{told}, \text{V}, \text{left}) \\ &\cdot p_d(\text{STOP}|\text{S}, \text{VP}, \textit{told}, \text{V}, \text{right}) \end{aligned}$$

Lexicalized trees are just one example of a modification to PCFGs that results in better performance, and is used in state-of-the-art parsers. They are an example of a general paradigm: take PCFGs, add richer details (to take into account more than one production rule at a time), and then deal with the added complexity by Markovian assumptions.

Latent variable grammars

In the previous subsection, we saw a particular way of *annotating* the nodes of a parse tree with additional (lexical) information, to improve the performance of PCFG. Another possible modification to the parse trees is to make the grammar richer, by subdividing each nonterminal into a number of *categories*. This can be considered an alternative form of annotation: in addition to being labeled with a nonterminal, each

node is annotated with the *category* of its nonterminal.

To see why this makes sense, we give some examples of natural categories into which certain nonterminals can be subdivided.

- NP: NP_1 = subject vs. NP_2 = object
- DT: DT_1 = determiners vs. DT_2 = demonstratives
- IN: IN_1 = sentential vs. IN_2 = prepositional

It turns out that parsers which use reasonable manually generated categories (such as those in the above list) perform better than plain PCFGs. However, it can be tricky to determine what categories to divide each nonterminal into, in order to achieve best performance.

Latent variable grammars address this problem by modeling the categories with latent variables that need to be learned during an initial training phase. Suppose that each of the nonterminals S, NP, and VP are split into two categories. Then the grammar rules become as shown in Figure 10.

Old rules		**New rules**	**Parameters**
$\text{S} \to \text{NP VP}$	$\longmapsto$	$\text{S}_0 \to \text{NP}_0\ \text{VP}_0$	θ_0
		$\text{S}_0 \to \text{NP}_0\ \text{VP}_1$	θ_1
		$\text{S}_0 \to \text{NP}_1\ \text{VP}_0$	θ_2
		$\text{S}_0 \to \text{NP}_1\ \text{VP}_1$	θ_3
		$\text{S}_1 \to \text{NP}_0\ \text{VP}_0$	θ_4
		$\cdots$	$\cdots$
		$\text{S}_1 \to \text{NP}_1\ \text{VP}_1$	θ_7

Figure 10: Grammar rules change when more categories are added.

With each new rule is associated a parameter θ_i that models how likely the annotated transition is. The parameters θ_i can be learnt, for example, using the EM algorithm which was covered in previous lectures, provided that the number of categories is given.

This leads us to an important question when learning categories: how to determine the number of categories to split each nonterminal into? The ideal number of categories is usually different for different nonterminals: we would like more complex nonterminals to consist of more categories, but simple nonterminals should not be split into too many categories.[4] One technique to address this is *split and merge*: i.e. recursively try dividing each category into two subcategories, then see if the likelihood of the training set is improved, and keep the new categories only if a significant improvement was made.

After the categories have been learned, the parser is trained with the resulting enriched set of nonterminals encompassing all the learned categories. Latent variable grammars are used by the Berkeley parser, which is the best-performing constituency parser today.

[4] The first condition is desirable in order to get an expressive grammar which captures the different aspects of complex nonterminals. The second condition is desirable both to avoid overfitting, and to avoid adding unnecessary computational complexity by having too many categories.

Dependency Parsing

The context-free grammars we have learned determine syntactic structure based on the *constituency relation* (such as $S \rightarrow NP\ VP$), which uses phrase structure rules of the form $A \rightarrow B\ C$ to determine relationships from the constituent A to sub-constituents B and C. We now focus on a different type of grammar that uses instead the *dependency relation* to determine syntactic structure. In a dependency grammar, the structure is composed of lexical items, usually words, that are linked by binary asymmetric relations called *dependencies*. The concept of dependency comes from the theory that each word in a sentence directly relates to or is modified by another word in the sentence, with all words having a direct or indirect relation to the main verb of the sentence. Each dependency relation $A \rightarrow B$ means that A governs B or that B depends on A. There are many different kinds of dependency grammars but they all share these core concepts.

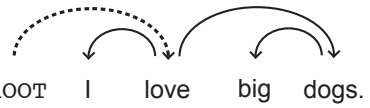


Figure 11: **Dependency parsing** assigns binary asymmetric dependency relations between words in a sentence. The ROOT node points to the main verb, which is the word that all the rest of the words in the sentence depends on, directly or indirectly.

For instance, in Figure 11, we have the example sentence "*I love big dogs*." Each word is a node in the structure, with one extra node for ROOT. An extra ROOT node always points to the main verb of the sentence, which in this case is the verb *love*. From there, *love* is modified by the subject *I* and also modified by the object *dogs*. Finally, *dog* has the noun modifier *big*.

The complete dependency structure can explicitly represent the following items.

1. Head-dependent relations: These are the directed arcs demonstrating dependency relationships, as shown in Figure 11.
2. Functional categories: Each dependency relation arc in the structure can have a label which describes what category of dependency relation it is.
3. Structural categories (optional): Each word in the sentence can have a part-of-speech label.

In contrast to dependency structures, constituency structures use more nodes to represent the parts of the sentence at different levels. They explicitly represent higher-level phrases as non-terminal nodes (with labels such as NP and VP), structural categories also as non-terminal nodes (with labels such as PP and JJ), as well as potentially some functional categories. As can be seen, constituency structures generally encode a great deal more information about the sentence than dependency structures.

For many languages, dependency parsers are trained with dependency structures that are hand-coded from scratch. For languages like English, where corpora of hand-coded constituency structures already exist, such as the Penn TreeBank project, dependency structures

are often automatically built from constituency structures using lexicalization. In many cases, the dependency parsers trained from dependency structures generated in this way outperform dependency parsers trained from hand-coded dependency structures. This is because of the greater information encoded in constituency structures as mentioned earlier.

Criteria for Heads and Dependents of a Dependency Relation

We describe the lexical items in a single dependency relation from $A \rightarrow B$ as the *head* and the *dependent*, respectively. There are many criteria for determining what constitutes a dependency relation. We mention some main established criteria below; however, keep in mind that some dependency relations do not readily satisfy all criteria. Given a head H, dependent D, and dependency construction C:

- H determines the syntactic category of C and can often replace C. For instance, "*big dogs*" is a noun phrase that could simply be replaced with "*dogs*" in our example sentence.
- H is the term that carries semantic meaning, so it determines the semantic category of C. D modifies H. For instance, in the phrase "*willow tree*", "*tree*" is a broad semantic category while "*willow*" specifies the type of tree. Thus "*tree*" is the head.
- H is obligatory while D sometimes is optional. The prior examples of "*big dogs*" and "*willow tree*" demonstrate this criteria.
- H selects D and determines whether D is obligatory. For example, intransitive verbs do not require a direct object ("*Sarah sneezed.*") while transitive verbs do ("*Antonio kicked the chair.*"). In these cases, the verb is H and the object is D.
- The form of D depends on H (agreement or government). This is more common in languages other than English. For instance, in languages where grammatical gender plays a role, there is often agreement between a noun and its modifiers ("*le grand homme*" versus "*la grande chaise*"). The gender of the H noun specifies the form of the D modifiers.
- The linear position of D is specified with reference to H. For instance, in English, adjectives describing a noun generally need to come directly before the noun and not elsewhere in the sentence ("*big dogs*").

Determining dependency relations can be tricky. In Figure 12, we show the dependency graph for a more complex sentence.

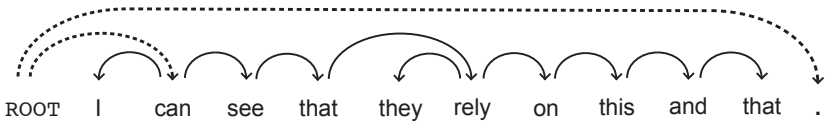


Figure 12: The dependency parsing for a more complex sentence.

When building the dependency structure, one should keep in mind the following potentially tricky cases.

- *Complex verb groups*: When it comes to verb groups such as "*can see*", the auxiliary verb ("*can*") should point to the main verb ("*see*").
- *Subordinate clauses*: In these types of clauses ("*after Amy sneezed*") the complementizer ("*after*") should point to the verb in question ("*sneezed*"). Likewise, in our example in Figure 12, "*that*" points to "*rely*".
- *Coordination*: When it comes to coordinating conjunctions joining two or more words, clauses, or sentences, the dependency relations go in the order of the sentence, so that the item to the left of the coordinator points to the coordinator and the coordinator points to the item on the right side. This can be seen with the phrase "*this and that*" in the example sentence.
- *Prepositional phrases*: In these phrases, such as "*on this*", the preposition ("*on*") points to the nominal ("*this*").
- *Punctuation*: Usually dependency parsers model punctuation as individual nodes. Thus the punctuation mark at the end of the sentence, in this case a period, would be its own node and have an dependency relation coming from the ROOT node.

Requirements of a Dependency Graph

Defined more formally, a dependency structure or graph can be defined as a direct graph G, consisting of a set of nodes V, a set of arcs (edges) E, and a linear precedence order on V. For the case of labeled graphs, there are additionally labels on the nodes V with the corresponding word forms, as well as labels on the arcs E with the dependency types.

Some conditions on dependency graphs include:

- Syntactic structure is complete (connectedness).[5]

$$\forall i \; \exists j \text{ s.t. } i \rightarrow j \text{ or } j \rightarrow i$$

- Syntactic structure is hierarchical (acyclic).

$$\text{if } i \rightarrow j \text{ then } \neg(j \xrightarrow{*} i)$$

[5] Enforced by the addition of the special ROOT node as seen in previous figures.

- Every word has a single head.

$$\text{if } i \rightarrow j, \;\; \forall (k \neq i) \; \neg (k \rightarrow j)$$

Projectivity

Finally, we discuss the concept of *projectivity*. Many algorithms assume this characteristic within dependency graphs, including the shift-reduce algorithm described in the next section. The overall concept of projectivity is that the edges or arcs drawn on the dependency graph do not cross. More specifically, if we put the words in the sentence in their linear order, preceded by ROOT, the edges can be drawn above the words without crossings. This has been the case with all the examples shown previously. Equivalently, a word and its descendants form a contiguous substring of the sentence.

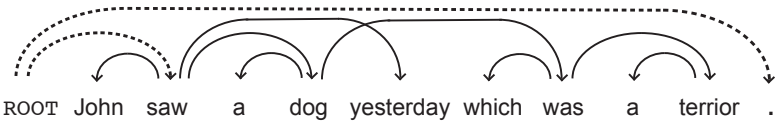


Figure 13: The dependency parsing for a sentence that exhibits non-projectivity.

There are some cases in English where projectivity is violated but these cases are rare and can sound awkward. For instance, as seen in Figure 13, the sentence "*John saw a dog yesterday which was a terrier.*" has a crossing because "*saw*" goes to "*yesterday*", while "*dog*" goes to "*was*". In other languages with more flexible word order such as German, Dutch, and Czech, non-projective dependencies are more frequent. Most theoretical frameworks assume non-projectivity. However, as mentioned earlier, there are many algorithms that do assume projectivity.

Shift-Reduce Parser

We now discuss the shift-reduce algorithm, a popular parsing method that can be used to construct dependency graphs. Conceptually, shift-reduce parsing is a technique that uses a stack and queue data structure and sequentially adds edges representing dependency relations between words in a sentence. Some of the features of this algorithm include the following:

- The algorithm is inherently greedy. It parses the sentence from left to right and decides on dependencies, but the algorithm does not backtrack or change any of its previous decisions.

- Because the algorithm only takes one pass through the sentence, the algorithm's time complexity is $O(n)$; this is a large improvement from the algorithms used in constitueny parsing, which can have time complexities of $O(n^3)$. The success of this algorithm can largely be attributed to its speed.
- Shift-reduce requires that the underlying language and dependency graph is projective as discussed above. This algorithm will therefore work well in English, but perform poorly for non-projective languages.
- The algorithm is closely related to how humans parse sentences; instead of looking at the sentence as a whole, we sequentially parse the sentence one word at a time, deciding on the dependency (and even the constituency) relations.

Description of Algorithm

The algorithm uses a stack $[\ldots, w_i]_S$ of partially processed tokens from the sentence and a queue $[w_j, \ldots]_Q$ of remaining input tokens. At the beginning of the algorithm, the sentence is pushed to the queue and the `ROOT` is added to the stack. Words are then considered one-by-one off the top of the queue. As the words are processed, the parsing actions are built from the atomic actions of adding arcs, or edges in the dependency graph: $w_i \rightarrow w_j$, $w_i \leftarrow w_j$, and stack and queue operations. These four parsing actions are as follows:

- Shift: This operation pops the first element off of the queue and pushes it onto the stack:

$$\begin{aligned} & [\ldots]_S \ [\mathbf{w_i}, \ldots]_Q \\ \Rightarrow\ & [\ldots, \mathbf{w_i}]_S \ [\ldots]_Q \end{aligned}$$

- Reduce: This operations pops the first element off the stack.

$$\begin{aligned} & [\ldots, \mathbf{w_i}]_S \ [\ldots]_Q && \exists\ w_k : w_k \rightarrow w_i \\ \Rightarrow\ & [\ldots]_S \ [\ldots]_Q \end{aligned}$$

- Left-Arc$_r$: An arc, or edge, is made between the top element in the queue and stack, and the top element from the stack is popped.

$$\begin{aligned} & [\ldots, \mathbf{w_i}]_S \ [\mathbf{w_j}, \ldots]_Q && \neg \exists\ w_k : w_k \rightarrow w_i \\ \Rightarrow\ & [\ldots]_S \ [\mathbf{w_j}, \ldots]_Q && w_i \overset{r}{\leftarrow} w_j \end{aligned}$$

- Right-Arc$_r$: An arc is made between the top element in the stack and queue, and the top element from the queue is pushed onto the

stack.

$$[\ldots, \mathbf{w_i}]_S\ [\mathbf{w_j}, \ldots]_Q \qquad \neg \exists\ w_k : w_k \to w_j$$
$$\Rightarrow [\ldots, \mathbf{w_i}, \mathbf{w_j}]_S\ [\ldots]_Q \qquad w_i \xrightarrow{r} w_j$$

These parse actions show that the algorithm processes the right-dependents in an 'arc-eager' manner, always trying to link the next unprocessed input token to something partially processed. In addition, each action takes constant time, and because each token has a constant number of actions done to it, the overall runtime is $O(n)$, linear in the size of the input.

An example will better demonstrate how a sentence is parsed. We start with the example sentence "Economic news had little effect on financial markets." To begin, we initialize the queue with the sentence and the stack with the ROOT node:

[ROOT]$_S$ [Economic news had little effect on financial markets .]$_Q$

In this situation, because the stack is effectively empty, we perform a shift operation:

[ROOT Economic]$_S$ [news had little effect on financial markets .]$_Q$

At this point, we can choose to make a left arc, a right arc, or reduce. We choose to make a left arc, resulting in:

[ROOT]$_S$ Economic [news had little effect on financial markets .]$_Q$

This decision seems arbitrary, but for now, we assume we have access to an oracle that tells us which parse action to take. Because the stack is empty, we shift again and decide to do another left-arc operation (while it is omitted for brevity, in doing any of these operations, we label each arc with a particular semantic label):

[ROOT$_S$ Economic news] [had little effect on financial markets .]$_Q$

[ROOT]$_S$ Economic news [had little effect on financial markets .]$_Q$

Next, operation we do is a right-arc, which results in the following:

[ROOT$_S$ Economic news had] little effect on financial markets .]$_Q$

We continue in this fashion, continuing to make decisions to make left-arc or right-arc, or to shift or reduce the tokens. By the time we reach the end of the sentence, we will have a dependency graph on the sentence computed in linear time.

Decision-Making in Shift-Reduce Parsing

A major question in this algorithm is how we are able to pick one of the four parse actions given the state of the stack and queue at any point. We are implicitly asking some sort of *oracle* for an decision, but we note that we can approximate this oracle by a *classifier*, which can consequetly be trained using *treebank* data. This observation allows us to black-box the process of decision-making, and we can use any sort of learning method or classifier to make decisions. These learning methods can be support vector machines, memory-based learning, maximum entropy modeling, or even deep neural networks.

Stated more formally, this problem of learning approximates a function from parser states, represented by feature vectors, to parser actions (i.e. shift, left-arc, right-arc, reduce), given a training set of gold standard derivations. To get the training data, we take the given dependency trees and break these structures into a derivation, or sequence of decisions. In doing so, we can associate a probability distribution over derivations:

$$P(c_0, \ldots, c_m) = \prod_{i=1}^{m} P(c_i | c_0, \ldots, c_{i-1}) = \prod_{i=1}^{m} P(c_i | c_{i-1})$$

As described above, PCFGs can be regarded as models at the derivations but they are context-free, so most information in c_{i-1} is ignored. However, and as is the idea in shift-reduce, we can think of $P(c_i|c_{i-1})$ as classifiers, which take features of c_{i-1} (and the input words x) and return a score for every configuration c_i:

$$\theta^T \Phi(x, c_i, c_{i-1}).$$

This method of using shift-reduce parsing and training some classifier is often called *data-driven deterministic parsing*. In practice, while data-driven deterministic parsers do not perform as well as spanning tree parsers with online training and more sophisticated parsers, their performance is very comparable, while being considerably faster. One of the best performing neural parsers was indeed based on the shift-reduce algorithm and was popularized by the Stanford NLP group and Google.

6.864 Advanced Natural Language Processing[1]

Lecture 11: Dependency Parsing (Transition, MST)

20 October 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Oleg Grinchuk, Aizhan Ibraimova, Ji Young Lee.

In the last lecture, we saw transition-based dependency parsing which derives a dependency parse tree by predicting a transition sequence (which can be either shift, left-arc, right-arc, or reduce) from an initial configuration to the terminal configuration. In this lecture we will discuss another transition-based dependency parser based on neural network, as well as a maximum spanning tree parser.

Motivation

A lot of methods used in NLP can be easily illustrated in the context of parsing because this problem is well studied and many algorithms have been developed for parsing. The problem is challenging for many reasons, including limited annotated training data (sparsity due to unseen word combinations) and algorithmic complexity. The insights behind the solutions are not particular to parsing but extend to other NLP tasks as well. Therefore, understanding the existing parsing algorithms thoroughly will help us design algorithms for other NLP tasks.

Properties of parsing algorithms

Let's try to understand parsing algorithms along three significant dimensions. We label these Scoring, Inference, and Feature representations as shown in Figure 1.

Scoring describes the way we assess how good a particular tree would be for a given sentence. A scoring function is parameterized and these parameters must be learned from annotated data. A rich global scoring function includes many parameters for controlling how combinations of dependency arcs relate to the sentence. A local scoring function, in contrast, has much fewer parameters and merely assesses the importance of each arc in isolation, independent of how the other dependencies are drawn.

Inference pertains to how well we optimize the tree so as to maximize the scoring function. A local (myopic) inference algorithm operates on the tree sequentially (e.g., greedily modifying arcs) so as to improve the score but without any guarantees to find the global optimum. A global inference algorithm, on the other hand, finds the highest scoring tree but may be algorithmically complex. The difficulty of the inference problem relates to the richness of the scoring function. For example, easily finding the highest scoring tree is (roughly speaking) possible only when the scoring function is local (per arc).

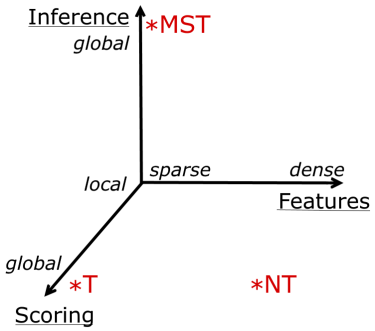


Figure 1: Parsing methods comparison

Feature representations could be sparse or dense, depending on what types of features are considered. For example, we may introduce bigram features and corresponding weights to score each dependency arc. These features are sparse, captured by a $|V|^2$ dimensional one-hot binary vector. Dense features condense the dimensionality before taking combinations, greatly reducing the number of parameters that must be learned. In doing so, we can no longer separately score all possible combinations. However, we can learn how the features are aggregated into continuous vectors.

Let's consider a transition dependency parser (T) in terms of properties mentioned above. It walks through the sentence looking at current configuration and makes simple decisions about which transition to use. Inference is therefore local. On the other hand, each decision can be based arbitrarily on the configuration at that point (partial trees) so introducing a rich global scoring function is easy. A standard transition based parser casts each configuration in terms of a large number of indicator features, making the feature representation very sparse.

We will illustrate two parsing algorithms in detail. The first one is Neural Transition-based parser (NT) which uses dense feature representations. The second algorithm is Maximum Spanning Tree parser (MST) which employs sparse local features but solves the highest scoring tree globally.

Transition-based dependency parsing

One of the popular transition systems is the arc-standard system, which we use for the parser. The configuration $c = (s, b, A)$ in this system consists of a stack s, a buffer b and a set of dependency arcs A. In the initial configuration, stack contains `ROOT`, buffer contains all the words from the sentence in the same order, the set of arcs A is empty. Let's define stack s as $s = \{s_1, s_2, ...\}$ where s_i is i-th top element on the stack, and buffer b as $b = \{b_1, b_2, ...\}$ where b_i is the i-th element on the buffer.

The arc-standard system uses following types of transitions:

- SHIFT: moves the first element in the buffer b_1 to the stack s.
- LEFT_ARC: adds an arc from s_1 to s_2 and removes s_2 from the stack.
- RIGHT_ARC: adds an arc from s_2 to s_1 and removes s_1 from the stack.

Every arc also has a label characterizing the type of dependency (e.g., subject, object). If we suppose that there are N different labels of the arcs, then we have $2N + 1$ possible transitions in total. Note that arcs

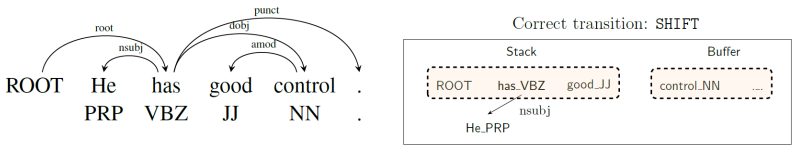


Figure 2: An example of transition-based dependency parsing. Top left: a desired dependency tree, top right: an intermediate configuration, bottom: a transition sequence of the arc-standard system.

#	Transition	Stack	Buffer	A
0		[ROOT]	[He has good control .]	$\emptyset$
1	SHIFT	[ROOT He]	[has good control .]	
2	SHIFT	[ROOT He has]	[good control .]	
3	LEFT-ARC(nsubj)	[ROOT has]	[good control .]	$A\cup$ nsubj(has,He)
4	SHIFT	[ROOT has good]	[control .]	
5	SHIFT	[ROOT has good control]	[.]	
6	LEFT-ARC(amod)	[ROOT has control]	[.]	$A\cup$amod(control,good)
7	RIGHT-ARC(dobj)	[ROOT has]	[.]	$A\cup$ dobj(has,control)
...	...	...	...	...
10	RIGHT-ARC(root)	[ROOT]	[]	$A\cup$ root(ROOT,has)

can be drawn only between the top two words in the stack, from s_1 to s_2, or vice versa.

An example of transition-based Dependency Parsing for the sentence 'He has good control' is shown in Figure 2. The top left sub-figure shows the desired dependency tree for the sentence including POS-tags and label-tags. Top right sub-figure shows one possible configuration. Note the sub-tree that is hanging off of the word 'has' in the stack.

Example.
Let's look at the configuration in the top right sub-figure of Figure 2, where s={Root has good} is in the stack, in the buffer we have b={control.}. In such configuration s_1={good}, s_2={has}, b_1 ={control}. Note that depending on the left-right arcs drawn in earlier steps, the words in the stack such as s_1 can have a left child ($lc(s_1)$) or a right child ($rc(s_1)$) and possible more than one left/right child. The word $\{He\}$ represents a left-child of {has} (the arc is drawn slanted to the left). In case of more than one left or right children, we use $lc(s_1)$ to specify the left-most child. The table at the bottom of Figure 2 shows transition sequence of the arc-standard system from the initial configuration to the terminal one. The stack and buffer are those that follow the transition indicated on the same line. The configuration discussed above (in the top right sub-figure of Figure 2) corresponds to the step 4 in the table.

- *Configuration 5.* After the configuration 4 we do transition SHIFT which adds {control} to the stack, i.e. s_1={control}, s_2={good} and b_1={.}.
- *Configuration 6.* Since s_2 depends on s_1, the next transition is LEFT_ARC, which removes {good} from the stack and put it as a left-child of {control}.

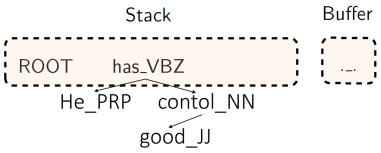


Figure 3: Configuration 7.

Now we have s={Root has control}, where s_1={control} depends on s_2={has}.

- *Configuration 7*. We now do the transition RIGHT_ARC which removes {control} from the stack (becomes a right-child of {has}). And in this configuration s={Root has} and b={.}. The illustration of this configuration can be seen in Figure 3.

Continuing this algorithm, we can get the dependency tree. Note that ROOT always stays in a stack.

Classification problem

It is relatively simple to construct dependency tree if we know which transition to use for each possible configuration. Unfortunately, we don't have such information, but only training examples of configurations with labeled transitions. This can be posed as a multi-label classification problem where we can train a classifier that takes as input a configuration and returns the most probable transition out of $2N + 1$ possible variants. The remaining questions include: how can we represent configurations so that they can be fed into a classifier? Which features do we choose? We will use a neural network to address these questions.

Neural network based parser

Let's look at the neural network model, as proposed in [1].

[1] Chen, Danqi, and Christopher D. Manning. "A fast and accurate dependency parser using neural networks." *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Vol. 1. 2014.

Input layer

Given a configuration c, we have access to the word (w), POS tag (t) and arc label (l) for each position in the stack, buffer, and trees comprising dependency arcs (hanging from words in the stack). However, we do not necessarily have to use all this information. Often a reasonable subset of words, POS tags, arc labels based on the positions in the stack, buffer, and dependency trees are chosen to be used as features. We denote these sets as S^w, S^t, and S^l, respectively.

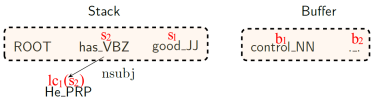


Figure 4: An example of configuration.

Example.
Let's look at the configuration shown in Figure 4. Suppose we represent the configuration as $[lc_1(s_2),\ s_2,\ rc_1(s_2),\ s_1,\ b_1]$, where $lc_1(s_i)$ and $rc_1(s_i)$ denote the leftmost and rightmost child of s_i, respectively. Then

the set of words S^w, POS tags S^t, and arc labels S^l are:

$$S^w = \text{[He, has, NULL, good, control]},$$
$$S^t = \text{[PRP, VBZ, NULL, JJ, NN]}, \quad \textit{and}$$
$$S^l = \text{[nsubj, NULL, NULL, NULL, NULL]}.$$

Missing features here are treated as non-existent element, *NULL*. Note that since the stack and the buffer change in size based on the transition operations, the number of NULLs changes. However, padding these sets with NULLs ensures that the classifier will maintain a consistent interpretation of what each coordinate means.

In order to use combinations of elements in S^w, S^t, and S^l to guide transition decisions without exploding the size of the feature representations, we resort to dense features. Specifically, we first represent each word as a d - dimensional word vector, map POS tags to tag vectors of lower dimension, and similarly for the arc labels in S^l. Note that NULLs also have an associated vector (e.g., all zeros). This allows us to map the configuration into a fixed length feature vector. In other words, each element in S^w, S^t, S^l is replaced by the corresponding vector and these vectors are concatenated into an input feature vector $[x^w,\ x^t,\ x^l]$.

Neural network architecture

Once we have the fixed length input vector for each configuration, the remaining classifier is a simple feed-forward neural network with one hidden layer. Each hidden layer unit takes in a weighted combination of coordinates of $x = [x^w,\ x^t,\ x^l]$ and passes it through the activation function. In this particular parser, a **cube activation function** is used:

$$h = (W_1 x + b_1)^3 = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^3$$

where $(\cdot)^3$ is applied element-wise so h is a vector of hidden unit activations. Note that the cubic activation function is able to capture interactions between any three elements from the input layer:

$$(w_1 x_1 + \cdots + w_m x_m + b)^3 = \sum_{i,j,k} (w_i w_j w_k) x_i x_j x_k + ...$$

where x_i, x_j, x_k could come from words, tags or label embeddings.

A softmax layer is finally added on the top of the hidden layer for modeling multi-class probabilities $p = \texttt{softmax}(W_2 h)$, where $p \in \mathbb{R}^{|\mathcal{T}|}$ and $\mathcal{T}$ is the set of possible transitions (note that arc labels increase the number of possible transitions).

Performance

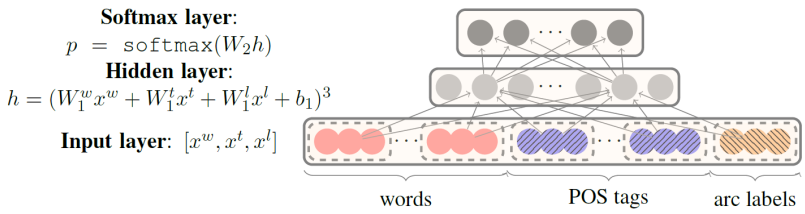


Figure 5: NN architecture for transition-based dependency parsing (Chen & Manning, 2014).

Figure 6 shows the results of different algorithms by computing unlabeled attachment scores (UAS) and labeled attachment scores (LAS) on the English Penn Treebank. Neural network performs better both in terms of speed and quality of labeling than the comparisons listed. However, the authors did not compare to the state-of-the-art algorithms (that would have higher accuracy). As a transition based parser, the main advantage of the neural parser is speed.

Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	89.9	88.7	89.7	88.3	51
eager	90.3	89.2	89.9	88.6	63
Malt:sp	90.0	88.8	89.9	88.5	560
Malt:eager	90.1	88.9	90.1	88.7	535
MSTParser	92.1	90.8	**92.0**	90.5	12
NN Parser	**92.2**	**91.0**	**92.0**	**90.7**	**1013**

Figure 6: Accuracy and parsing speed on English Penn Treebank (Chen & Manning, 2014).

Parsing as Structural Prediction

Neural transition-based parser uses a *local* inference algorithm. That is, the algorithm makes dependency decisions myopically step by step, deciding whether to add left arc, right arc, or shift. We consider here a parser at the other end of the spectrum where the predicted parse tree is solved directly as the highest scoring tree. In order to make this possible, we will have to settle for a simpler (per-arc) scoring function.

Let $\mathbf{x} = x_0 x_1 x_2 \dots x_n$ represent a generic input sentence where x_0 is ROOT, and $\mathbf{y}$ represent a dependency tree for sentence $\mathbf{x}$. Here, $\mathbf{y}$ is represented as the set of arcs that comprise a dependency tree; that is, $\mathbf{y} = \{(i,j) | \text{word } x_j \text{ modifies } x_i\}$. For example, for the input sentence

$$\begin{array}{lcccccc} \mathbf{x} = & \text{ROOT} & \text{I} & \text{saw} & \text{a} & \text{good} & \text{lecture} \\ & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \end{array}$$

one possible dependency tree could be

$$\mathbf{y} = \{(0,2),(2,1),(2,5),(5,3),(5,4)\},$$

which can be equivalently represented as a directed tree.

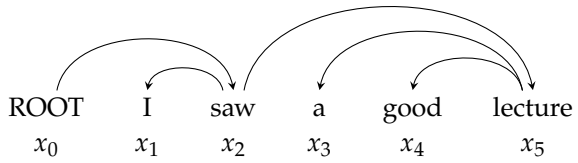


Figure 7: A graphical representation of dependency tree

$\mathbf{y} = \{(0,2),(2,1),(2,5),(5,3),(5,4)\}$

for input sentence

$\mathbf{x}$ = ROOT I saw a good lecture

Let $\mathcal{Y}(\mathbf{x})$ be the set of all valid dependency trees for input sentence $\mathbf{x}$. In other words, this is the set of directed spanning trees over the words in the sentence. For each dependency tree $\mathbf{y} \in \mathcal{Y}(\mathbf{x})$, let us define a feature vector $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$ consisting of indicator features that tie the arc choices to the sentence. These features are weighted by a weight/parameter vector $\boldsymbol{\theta}$ whose dimension is the same as the feature vector. The score of each dependency tree $\mathbf{y} \in \mathcal{Y}(\mathbf{x})$ can be then defined as $\boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$.

Now, the dependency parsing problem can be formulated as finding the dependency tree $\hat{\mathbf{y}} \in \mathcal{Y}(\mathbf{x})$ that maximizes the score:

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y} \in \mathcal{Y}(\mathbf{x})} \boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}). \tag{1}$$

In order to train and use such a parser, we need to address three problems. First, we need to specify exactly what the features (how we tie the trees to the sentence). Second, we need to solve for the highest scoring tree for any given parameter vector. And, third, we need to learn the parameters from annotated sentences.

First Order Maximum Spanning Tree Parser

As an example parser we will consider the *first order maximum spanning tree parser* implemented in [2]. This parser uses a simple per-arc (arc factored) scoring function. In order words, the score for each tree for a given sentence is represented by a sum of scores associated with each arc. This scoring function is first order since the score for each arc does not depend on what other arcs are draw. The advantage of this simple scoring function is that we can solve for the highest scoring tree efficiently using the maximum weight directed spanning tree algorithm (Chu-Liu-Edmonds). We will elaborate on these further, including an on-line algorithm for estimating the weights.

[2] McDonald, Ryan, Koby Crammer, and Fernando Pereira. "Online large-margin training of dependency parsers." *Proceedings of the 43rd annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2005
Figure 8: [**??**]

Edge Based Features

We introduce here a per-arc scoring function. To this end, we introduce a feature vector for each possible arc (i, j) in the tree. Formally,

$$s(i, j) = \boldsymbol{\theta} \cdot \boldsymbol{\phi}(i, j, \mathbf{x}) = \sum_k \theta_k \phi_k(i, j, \mathbf{x}),$$

where $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_K)$ is the weight/parameter vector and $\boldsymbol{\phi}(i, j, \mathbf{x}) = (\phi_1(i, j, \mathbf{x}), \ldots, \phi_K(i, j, \mathbf{x}))$ is a binary indicator feature vector corresponding to arc (i, j) in sentence $\mathbf{x}$. An example bi-gram indicator (a coordinate of the feature vector) is shown in Figure 9. The features are typically built from templates. For example, a bi-gram feature vector

for (i, j) arc is a one-hot sparse vector with 1 in position that corresponds to the word pair (x_i, x_j). A similar vector can be constructed based on tags rather than words, and so on. All these vectors are then concatenated together into a long and sparse feature vector $\boldsymbol{\phi}(i, j, \mathbf{x})$.

Taken together, the score for a dependency tree $\mathbf{y} \in \mathcal{Y}(\mathbf{x})$ is the sum of the associated arc scores

$$s(\mathbf{x}, \mathbf{y}) = \sum_{(i,j)\in\mathbf{y}} \boldsymbol{\theta} \cdot \boldsymbol{\phi}(i, j, \mathbf{x}) = \boldsymbol{\theta} \cdot \underbrace{\sum_{(i,j)\in\mathbf{y}} \boldsymbol{\phi}(i, j, \mathbf{x})}_{\triangleq \boldsymbol{\phi}(\mathbf{x},\mathbf{y})} = \boldsymbol{\theta} \cdot \boldsymbol{\phi}(x, y),$$

where we can think of $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y}) = \sum_{(i,j)\in\mathbf{y}} \boldsymbol{\phi}(i, j, \mathbf{x})$ as the feature vector associated with tree $\mathbf{y}$. Note that since arc features are indicator features, the feature vector for the tree represents counts, e.g., how many times a particular pair of tags appear as the end points of the arcs.

$$\phi(i, j, \mathbf{x}) = \begin{cases} 1, & \text{if } x_i = \text{'saw'}, x_j = \text{'lecture'} \\ 0, & \text{otherwise.} \end{cases}$$

Figure 9: An example of a binary feature of arc (i, j) for input sentence $\mathbf{x}$. In the dependency tree of Figure 7, this feature would have a value of 1 for arc $(2, 5)$.

Maximum Weight Directed Spanning Tree

We are now ready to address the second sub-problem: given the per-arc scoring function, how do we find the dependency tree that maximizes the score? We can reformulate this problem as the *maximum weight directed spanning tree* (MST) problem (also known as the maximum arborescence problem).

For a given input sentence $\mathbf{x} = (x_0, x_1, \ldots, x_n)$ with $n + 1$ words (where $x_0 = \text{ROOT}$), we define a weighted directed graph $G_\mathbf{x} = (V_\mathbf{x}, E_\mathbf{x})$ where $V_\mathbf{x} = \{0, 1, \ldots, n\}$ and $E_\mathbf{x} = \{(i, j) | i \neq j, i \in \{0, 1, \ldots, n\}, j \in \{1, 2, \ldots, n\}\}$ are the sets of vertices and arcs (i.e., directed edges), respectively. In words, $G_\mathbf{x}$ is a graph with each vertex $i \in \{0, \ldots, n\}$ corresponding to each word x_i in the input sentence $\mathbf{x}$, and a directed edge between every pair of distinct vertices (excluding arcs into the root vertex). To each possible arc (i, j) in $G_\mathbf{x}$, we assign a weight which is just the arc score $s(i, j) = \theta \cdot \boldsymbol{\phi}(i, j, \mathbf{x})$.

It is clear that there is a bijection between the set of dependency trees $\mathcal{Y}(\mathbf{x})$ and the set of spanning trees of $G_\mathbf{x}$. Moreover, the score of each dependency tree is exactly the weight of the corresponding spanning tree in the graph, defined as the sum of weights of all the arcs. We can use the well-known algorithm for directed MST problem, called *Chu-Liu-Edmonds algorithm*. See Figure 10 for a pseudocode of the algorithm. While we explain it here, tthe algorithm itself is not a required part of the course material.

Chu-Liu-Edmonds algorithm is a recursive algorithm, which takes as input a directed graph $G = (V, E)$ rooted at r and an arc weight function $s : E \to \mathbb{R}$, and outputs a maximum (weight) spanning tree $E^* \subseteq E$. In each recursive call of the algorithm, for each node x other

Chu-liu-Edmonds $(G = (V, E), r, s)$
Input: Directed graph $G = (V, E)$ with root at r and edge weight function $s : E \rightarrow \mathbb{R}$
Output: Maximum spanning tree $E^* \subseteq E$ of G

Let $M = \{(\pi(x), x) : x \in V \setminus \{r\}\}$ where $\pi(x) \triangleq \arg\max_{x'} s(x', x)$
Let $G_M = (V, M)$
If G_M has no cycles
  return M
Else
  Choose a cycle C in G_M
  Let $(G', s', t) =$ **Contract**(G, C, s)
  Let $E' =$ **Chu-Liu-Edmonds**(G', r, s')
  $E^* = 0$
  Add $t(u', v')$ to E^* for each $(u', v') \in E'$
  Let (u, c) be the unique incoming edge to c in E'
  Let $(u, v) = t(u, c)$
  Add each edge in C except for $(\pi(v), v)$ to E^*
  return E^*

Contract $(G = (V, E), C, s)$
Input: Directed graph $G = (V, E)$, a cycle C and edge weight function $s : E \rightarrow \mathbb{R}$
Output: Directed graph $G' = (V', E')$, edge weight function $s' : E' \rightarrow \mathbb{R}$, and edge traceback function $t : E' \rightarrow E$

Let $V' = (V \setminus C) \cup \{c\}$
Let $E' = \emptyset$
For each $(u, v) \in E$ such that $u \notin C$ and $v \notin C$
  $E' \leftarrow E' \cup (u, v)$; $s'(u, v) = s(u, v)$; $t(u, v) = (u, v)$
For each $x \in V'$
  If there is any edge in E from a node in C to x
    $E' \leftarrow E' \cup (c, x)$; $s'(c, x) = \max_{v \in C:(v,x) \in E} s(v, x)$
    $t(c, x) = (v^*, x)$ where $v^* = \arg\max_{v \in C:(v,x) \in E} s(v, x)$
  If there is any edge in E from x to a node in C
    $E' \leftarrow E' \cup (x, c)$
    $s'(x, c) = \max_{v \in C:(x,v) \in E}\ s(x, v) - s(\pi(v), v)$
    $t(x, c) = (x, v^*)$ where $v^* = \arg\max_{v \in C:(x,v) \in E}\ s(x, v) - s(\pi(v), v)$
return $G' = (V', E'), s', t$

Figure 10: Chu-Liu-Edmonds algorithm for finding maximum spanning trees in directed graph

than the root, we denote the head (tail end of the arc) with maximum weight (ties broken arbitrarily) as $\pi(x)$. We consider the set of the edges $M = \{(\pi(x), x) : x \in V \setminus \{r\}\}$. If $G_M = (V, M)$ does not contain a cycle, then G_M is the maximum spanning tree of G.

Otherwise, artibtrarily choose a cycle C of G_M and define a new

weighted directed graph $G' = (V', E')$ in which cycle C is "contracted" into a single node c. We include all edges of E' that are not in the cycle C with original edge weight from G.

For each node x outside of C that has an incoming arc from any node in C we add an edge from c to x to the new graph G' whose weight is the maximum weight among all edges from C to x in the original graph G. The edge with maximum weight is recored in edge traceback function t.

For each node x outside of C that has an outgoing arc to a node in C we add an edge from x to c in the new graph G' and its weight is set to the maximum of $s(x, v) - s(\pi(v), v)$ among all $v \in C$ such that $(x, v) \in E$. Again, the edge (x, v) that maximizes $s(x, v) - s(\pi(v), v)$ is recorded in edge traceback function t.

We can run recursively call Chu-Liu-Edmonds algorithm with this new graph G' to get a maximum spanning tree E' of the graph G' with cycle C "contracted" to a single node c. From this, a maximum spanning tree of the original graph G can be constructed by combining the original edges of each edge in E' and all edges in C except for $(\pi(v), v)$ where v is the node in C that corresponds to the head of the unique incoming edge to c in the maximum spanning tree E' of G'.

Online Learning of Weight Vector

We are left with the problem of learning the parameters of the first order scoring function. We can do this efficiently by using an *online learning algorithm with averaging*. An online learning algorithm considers a single training instance (sentence and its gold tree) at a time, updating the weight vector so as to better (or correctly) parse the sentence. Figure 11 shows a pseudocode for a generic online algorithm with averaging, suitable for learning with any scoring function in a form of $\theta \cdot \phi(\mathbf{x}, \mathbf{y})$ (first order or not).

Input: training set of size N containing input-output pairs $(\mathbf{x}, \mathbf{y})$
Output: parameter $\overline{\boldsymbol{\theta}}$
1 : $\boldsymbol{\theta} = \mathbf{0}$; $\overline{\boldsymbol{\theta}} = \mathbf{0}$
2 : for $t = 1, \ldots, T$
3 : for $(\mathbf{x}, \mathbf{y})$ in the training set (size N)
4 : update $\boldsymbol{\theta}$ based on $(\mathbf{x}, \mathbf{y})$
5 : $\overline{\boldsymbol{\theta}} \leftarrow \overline{\boldsymbol{\theta}} + \boldsymbol{\theta}/(N \cdot T)$
6 : return $\overline{\boldsymbol{\theta}}$

Figure 11: Generic online learning algorithm with averaging

It remains to specify a suitable online update to line 4 in Figure 11. Denoting $\boldsymbol{\theta}^c$ as the current parameter before the update in line 4, we

define the online update as the parameter $\boldsymbol{\theta}$ that is the solution to the following problem.

$$\begin{aligned} \arg\min_{\boldsymbol{\theta}} \quad & \|\boldsymbol{\theta} - \boldsymbol{\theta}^c\|^2 \\ \text{s. t.} \quad & \boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}) \geq \boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}, \hat{\mathbf{y}}) + \delta(\mathbf{y}, \hat{\mathbf{y}}) \end{aligned} \tag{2}$$

where $\hat{\mathbf{y}} \triangleq \arg\max_{\mathbf{y}' \in \mathcal{Y}(\mathbf{x})} \{\boldsymbol{\theta}^c \cdot \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}') + \delta(\mathbf{y}, \mathbf{y}')\}$ is the highest-scoring tree for the current weight vector if we add an incentive to select trees other than the gold tree. $\delta(\mathbf{y}, \mathbf{y}')$ here is a margin that counts the number of arcs that are different in $\mathbf{y}$ and $\mathbf{y}'$. Clearly $\delta(\mathbf{y}, \mathbf{y}) = 0$.

Let's understand the on-line update intuitively. First, we wish to remain close to the current parameters θ^c so that we do not unnecessarily change predictions for other sentences. Moreover, if the gold tree is already the highest scoring tree (even with the incentive to choose others), then we do not change the parameters at all. If this is not the case, we will modify the parameters only to the extent that the gold tree would score better than $\hat{\mathbf{y}}$ with at least margin $\delta(\mathbf{y}, \hat{\mathbf{y}})$, i.e.,

$$\boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}) \geq \boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}, \hat{\mathbf{y}}) + \delta(\mathbf{y}, \hat{\mathbf{y}})$$

(note that $\hat{\mathbf{y}}$ is set as the worst offending tree on the right hand side if the parameters are $\boldsymbol{\theta}^c$). The purpose of introducing the margin is to ensure that the scoring function would prefer the gold tree with a substantial margin.

The above algorithm is similar to the averaged perceptron algorithm, where the single highest scoring tree is used to update the weight vector without any notion of margin. In fact, both algorithms result in updates of the form

$$\boldsymbol{\theta} = \boldsymbol{\theta}^c + \eta[\boldsymbol{\phi}(\mathbf{x}, \mathbf{y}) - \boldsymbol{\phi}(\mathbf{x}, \hat{\mathbf{y}})]$$

They differ in terms of how $\hat{\mathbf{y}}$ is computed and what the learning rate η is. For the algorithm presented above,

$$\eta = \frac{\boldsymbol{\theta}^c \cdot \boldsymbol{\phi}(\mathbf{x}, \hat{\mathbf{y}}) + \delta(\mathbf{y}, \hat{\mathbf{y}}) - \boldsymbol{\theta}^c \cdot \boldsymbol{\phi}(\mathbf{x}, \mathbf{y})}{\|\boldsymbol{\phi}(\mathbf{x}, \mathbf{y}) - \boldsymbol{\phi}(\mathbf{x}, \hat{\mathbf{y}})\|^2}$$

Note that $\eta = 0$ if $\hat{\mathbf{y}} = \mathbf{y}$.

6.864 Advanced Natural Language Processing[1]

Lecture 12: Higher Order Parsing, Approximate Learning

22 October 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Adam Yala, UNKOWN NAMES.

In the last lecture, we introduced parsing as structured prediction and MST parsers. In this lecture, we continue our conversation on parsing and extend parsers we've seen before to create state of the art parsers. First, we discuss extending the MST parser with a global scoring function. Then we explore other methods to extend simple parsers, namely Reranking and Tritraining. We close with a discussion on creating dense feature representations with a Neural Network Parser. These methods are powerful and broadly applicable outside of parsing.

Extending the MST Parser

The MST parser is a simple, but global parser. It uses highly local, per edge scoring, to calculate maximum scoring trees. The power of the MST parser comes from predicting whole trees at time.

But is using local scoring powerful enough? Is the context of our two words large enough? Consider the following examples:

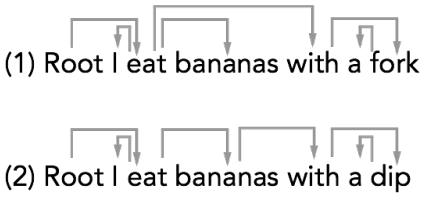


Figure 1: An example sentences with different dependency trees that cannot be differentiated with local scoring. The key difference here is the head of *with*. In the local scoring case, all we have to compare is the score of $(eat, with)$ vs. $(bananas, with)$. We do not capture the relation of $fork$ and dip in the decision.

These two sentences have different dependency structures. In (1), *with a fork* modifies how we *eat*, not the *bananas*. In this case, *eat* governs *with*. In (2), *with a dip* modifies the *bananas*, not *eat*. In this case, *bananas* governs *with*.

This distinction is difficult to differentiate with just local scores. We need a larger sense of context for the rest of the sentence. We would like to encompass grand-parent and sibling dependencies with in our contexts. To motivate this, consider the example again; $banana \rightarrow with \rightarrow dip$ is much more likely than $banana \rightarrow with \rightarrow fork$

We have just introduced a global scoring function. We now have two challenges, learning the parameters and finding the best tree for given the parameters. We learn the parameters to ensure the largest gap

between the gold tree and other candidate trees. Given the complexity of our new features, we can no longer rely on the Viterbi algorithm to find the highest scoring tree. The problem of finding the best tree with global features is NP-Hard, and so we use the randomized greedy algorithm to find a local optimal solution.

Learning the Parameters

We learn the parameters of our extended MST parser via an online algorithm.

Let (x, y) be our training pair, where x is the sentence, and y is the gold parse tree. The score of the training pair is as follows:

$$score(x, y, \Theta) = \Theta \cdot \Phi(x, y)$$

Where $\Phi(x, y)$ is our vector of global features, Θ is our vector of weights, and their dot product is our score.

Let $Y(x)$ be the set of possible tree of a sentence x. Note, this set is exponentially large in the length of x. In training, we choose our tree as follows:

$$y^* = argmax_{y' \in Y(x)} \{\Theta \cdot \Phi(x, y') + \delta(y, y')\} \quad (1)$$

where $\delta(y, y')$ is our margin. This is the number of different arcs between y and y'. Note, we use the margin to bias the parser towards trees that are high scoring, but very different from gold. This allows us to use our training to increase the distance in score between our gold tree and other options. This idea is illustrated in Figure 2. The margin is only used in training!

If our algorithm selects the gold tree despite the bias, i.e., $y = y^*$, then we do not update our parameters. Otherwise, we update our parameters as follows:

$$\Theta \leftarrow \Theta + \eta \nabla_\Theta (score(x, y, \Theta) - score(x, y^*, \Theta)) .$$

We perform gradient descent on max-margin loss. We have now discussed how to learn the parameters. The next challenge will be solving (1).

Figure 2: We bias our tree selection with $\delta(y, y^*)$ in training in order to increase the margin between our gold tree and other possibilities. We would like to make the gold tree a clear winner if possible

Solving the Maximization

We now work to solve (1). The structure of our scores might look something as following:

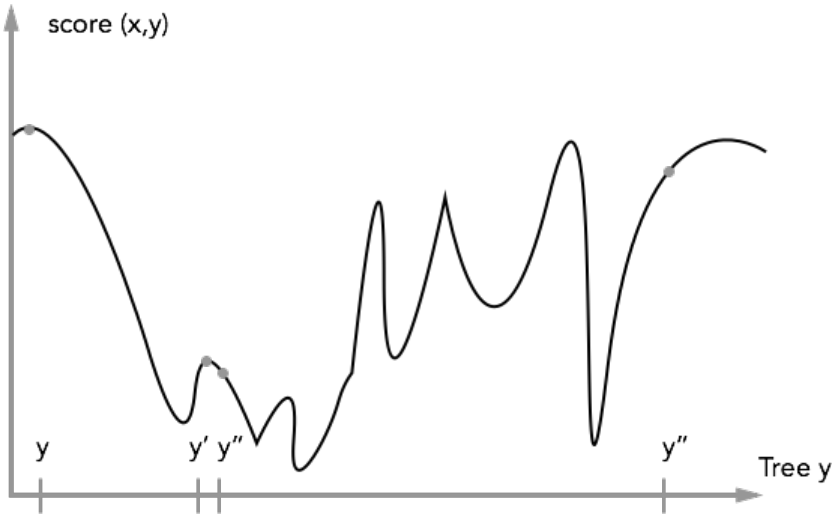


Figure 3: An example of what tree scores might look like as we move along the axis of valid trees. We can across this access using single *head* changes. An example of a head change is changing the *head* of *with* from *eat* to *bannana* in our previous example (See Figure 2).

It can be shown that we can enumerate through all valid trees using single head changes. However, the number of trees is exponentially large, and there is no general method of finding the global maximum efficiently. Finding the global maximum is NP-Hard. Instead, we introduce the Randomized Greedy algorithm to look for local optimums. First, let's discuss the Greedy Algorithm.

Greedy Algorithm

The Greedy Algorithm is simple. The pseudo code is as follows:

```
Greedy(y, x):
  y'= singleHeadChange(y)
  while (score(x,y') > score(x,y)):
    y = y'
    y' = singleHeadChange(y)
      return y
```

This algorithm discovers new trees by performing single head changes. Note that we make head changes bottom-up (from leaves and proceding upwards) in order to prevent cycles. The greedy algorithm halts when it finds a local maxima. However, depending on our initialization (i.e. the valid tree we start with), we might call $Greedy(y'', x)$, where y'' and x are shown in Figure 3, and find y'. In this case, we un-

luckily arrive at a poor local optimum. We mitigate the risk of getting unlucky with the following Randomized Greedy Algorithm.

Randomized Greedy Algorithm

The Randomized Greedy Algorithm is a simple extension of the previous algorithm. The pseudo code is as follows:

```
RandomizedGreedy(x, T):
  for i in range T:
    y_t = randomValidTree(x)
    y_t_best = Greedy(y_t, x)
  return argmax(y_t_best, score(x,y_t_best)
```

The Randomized Greedy Algorithm runs Greedy(y^t, x) T times and returns the maximum scoring tree. On each restart, we choose a new random tree, in hopes of finding a new local maximum. We essentially mitigate our risk of being unlucky by choosing many starting points, and choosing the best local maximum. The number of random restarts is generally on the order of hundreds. In this way, we have higher chance of achieving globally optimal.

Summary

To recap, we took the MST parser presented in the last lecture, and added global features. We learned the parameters using gradient decent on max-margin loss and found our best y for a given set of parameters using Randomized Greedy Algorithm. These techniques are broadly applicable outside of MST parsers. The notion of max-margin loss can be extended to training other parsers and other tasks. Similarly, the idea of Randomized Greedy Algorithm can be broadly applied to approximate difficult maximizations.

Other Methods

In the previous section, we discussed methods used to extend the MST parser. We now introduce three more broadly applicable methods to extend simple parsers. Namely, reranking, tritraining and dense-feature extraction with neural networks.

Reranking

We have already discussed the benefits of global scoring functions: they allow us to better differentiate between possible trees but come at a cost of computational complexity. The inference problem becomes

NP-Hard. One way to deal with this complexity is heavily pruning the set of trees we need to consider. We can use a simple parser, like MST or a Neural Shift Reduce Parser to generate the top couple of hundred candidate trees. We then score these trees globally and choose the best one.

To state this formally, let

$$Y(x) = \{y^1, y^2, y^3, ..y^k\}$$

where k is on the order of hundreds. We wish to solve

$$y' = argmax_{y \in Y(x)}\{score(x, y, \Theta)\}$$

The inference problem is now easy! We can quickly find the global maximum of the candidate set as it is very small.

This is an incredibly powerful method: it allows us to utilize much quicker and simpler parsers, and still leverages the power of global scoring. The key assumption here is that although the simple parser might not have enough power to guess the correct parse tree, the correct parse tree is likely to be in the top few hundred candidate trees. This concept is widely applicable outside of parsing, and is useful in extending simple tools.

Tritraining

One problem we haven't discussed before is the lack of a sufficiently large annotated training corpus. In a lot of these cases, the test corpus is abundant and large, but it is difficult to sufficiently train even simple parsers. Tritraining is a semi-supervised learning method to tackle that issue.

The idea is simple and powerful. We train 3 parsers on the training data we have. These parsers can all be of the same type or different. We then use their agreement/disagreement to create more training data. If 2 parsers agree on an annotation, and the 3rd disagrees, we use the agreed upon annotation as a training sample for the 3rd. We do this repeatedly until we have sufficiently training our parsers. This is illustrated in Figure 4.

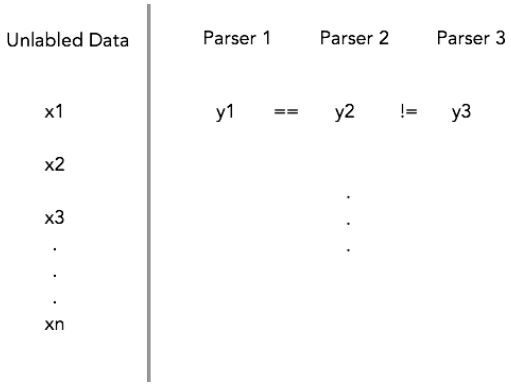


Figure 4: An illustration of Tritraining. In the parse of $x1$, Parse 1 and Parser 2 agree on an annotation, $y1$. This annotation, $y1$ is then used to train Parser 3. If none of the parsers agree, we do nothing.

The intuition is simple. If two parsers agree on a tree, denoted by y^1, then we take this as a strong signal that y^1 is a very good tree. The tree y^1 is either the gold annotation or something near it, and so we can confidently use it to train the third parser. If no parsers agree, then we do not have enough confidence to use the generated trees to train any parser.

Creating dense features

Many of the parsers we have discussed thus far use sparse indicator features. Each of these features is hardcoded and we cannot adjust what feature we are extracting through training. The set of indicator features can include things like word-features, prefix-suffix features and context features; the general approach is to use templates and generate thousands of them. The size of this set also makes it expensive to consider n-gram interaction of features.

Instead of sparse features, which are large and require a lot of human engineering, we would like to learn dense, low-dimensional feature vectors. The core idea is to leverage the power of word embeddings, which capture both syntactic and semantic meaning, to create sentence vectors. In short, we wish to learn dense vector for dependency structures.

To state this more explicitly, consider the example in Figure 5. Given length d word-embeddings for *eat* and *bananas*, we would like to compute a vector $v(eat\ bananas)$. We can then compare the score of $v(eat\ bananas)$ vs. $v(eat\ with)$ etc later on when choosing dependencies. We would like to keep the vector the same length so we can score $v(eat)$, $v(eat\ bananas)$ and $v(eat\ bananas\ with)$ with the same Θ. We have two problems, learning the sentence vector and learning the scoring function

Figure 5: Using the word embeddings $v(eat)$ and $v(bananas)$, we use a RNN to construct a a length d vector to represent the phase *eat bananas*. Note, since our dependencies are directed, $v(eat\ bananas)$ is not the same as $v(bananas\ eat)$

We can compute the vector $v(eat\ bananas)$ as follows,

$$v_{\in R^d}(eat\ bananas) = tanh(W_{d\ \times 2d} \begin{bmatrix} v(eat) \\ v(bananas) \end{bmatrix} + W_{0\ d\times 1})$$

We can do this recursively with a Recurrent Neural Network in order to capture full dependency structures. Now let's solve the scoring problem. We can do this via a simple dot product as follows,

$$score(eat\ bananas) = \Theta \cdot v(eat\ bananas)$$

6.864 Advanced Natural Language Processing[1]

Lecture 13: Embeddings, Word2Vec

9 September 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Hayk Saribekyan.

> You shall know a word by the company it keeps
>
> Firth J.R. 1957

In previous lectures we have consistently treated words as vectors but have not explicitly studied how to get them. This lecture discusses word embeddings, i.e., vector representations of words. A number of methods have been proposed to obtain dense low dimensional vectors from large corpora. We will focus here on a popular word2vec method.

Advantages of Using Word Vectors

We can think of word vectors as mappings from symbols (words) to vectors. Consider one-hot mapping given by $\phi : V \to \mathbb{R}^{|V|}$ and word vectors given by $v : V \to \mathbb{R}^d$, where $d << |V|$. Typically $|V|$ is on the order of $100,000$, whereas d can be just 100. There are many advantages of using dense low dimensional word vectors as opposed to sparse one-hot vectors. We will discuss two of them in detail.

Computational and Statistical Advantage

In many NLP tasks such as parsing, algorithms use lexicalized scoring functions that involve scores associated with observed combinations of words. Such combinations are much cheaper to include if we work with low-dimensional word vectors. To see this, consider first bigram vectors constructed from one-hot vectors. Specifically,

$$\phi_2(w, w') = vec(\phi(w)\phi(w')^T) \in \mathbb{R}^{|V|^2}$$

where *vec* concatenates the rows of matrix $\phi(w)\phi(w')^T$ into one (long) vector. Thus we can think of ϕ_2 as embedding a pair of words into $|V|^2$-dimensional space (on the order of 10^{10}). Even in this simple bigram case, we would need a large corpus to reliably estimate parameters associated with the coordinates of the bigram vector. In contrast, consider bigram vectors obtained from low dimensional word vectors:

$$v_2(w, w') = vec(v(w)v(w')^T) \in \mathbb{R}^{d^2}$$

Now v_2 embeds pairs of words into a d^2-dimensional space which is on the order of $10,000$ only.

So, word vectors give us a clear statistical advantage. Of course, by reducing the dimensionality, we are probably losing some information. Let's see what we can possibly capture with word vectors. Let's say that $v(w)$ is a one-hot d-dimensional vector of word clusters, where there are d clusters of words. In other words, the word vectors simply map each word to the closest cluster. The resulting vectors for pairs then capture how words interact across clusters. We get even more power by using *dense* low-dimensional vectors rather than just one-hot cluster indicators.

Semantic Similarity

Since the dimensionality of word vectors is much lower than that of one-hot vectors, it is possible that the compressed low dimensional vectors capture semantic similarities between words. In other words if words w and w' are semantically similar (e.g., based on human assessment) we would expect that $||v(w) - v(w')||$ is small or that the cosine similarity is close to one. Such grounded vector similarities are very useful for a variety of algorithms. For example, a parsing algorithm that encounters a new word would typically have trouble with deciding how to relate it to others. However, if we have a word vector for this new word and, as a vector, it is similar to a known word (from the training set), the resulting parsing decisions would also be related.

If word vector similarities capture semantic similarities the vectors can be easily used for all kinds of prediction tasks. For example, we could predict properties of words (see Figure 1) since most machine learning algorithms accept vectors as inputs.

Word	Vector	Properties
cat	$v(\text{"}cat\text{"})$	meow, agile, etc.
dog	$v(\text{"}dog\text{"})$	bark, etc.
tiger	$v(\text{"}tiger\text{"})$	?

Figure 1: An example of how word vectors can be used to predict properties

But the big question is whether word vectors do capture any semantic similarity. Most of word vectors are derived from large corpora based on how they co-occur in sentences or in small "contexts" around each word. We can ask then whether such co-occurrences relate to actual human assessments. Indeed they do, as show in Figure 2. The horizontal axis is a 1-10 similarity score that participants assigned to pairs of words (10 indicating that words are virtually the same). The vertical axis measures how often the words co-occur in the same context.

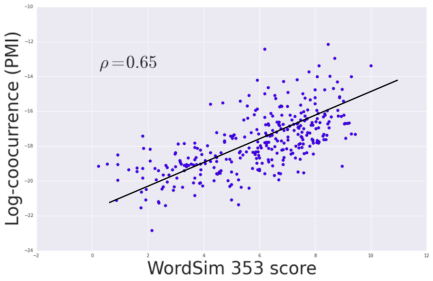


Figure 2: Semantic similarity assessments between pairs of words

Deriving Word Vectors from Context

Now that we know why word vectors can be useful, let's see how to estimate them. Assume that the only information we have is a large corpus of text (for example, all Wikipedia articles) absent any annotations.

All current approaches for creating word vectors rely on the idea

that words that appear in a similar context in the text are likely semantically similar, therefore should have similar vectors. Here we define *context* as the set of words that are within k words from the word in question (in each direction). For example consider the sentence in the slogan of the lecture.

"You shall **know** *the word by the company it keeps"*

The set of words {you, shall, the, word} form the context for "know" if $k = 2$. When k is small, the words in the neighborhood of a given word are likely to be syntactically constrained. As k increases, these constraints disappear (or are diluted), so the resulting vectors capture more semantic information about the words. The order of the context words is typically also not taken into account so as to avoid making the word vectors largely syntactic. Note that, in order to get a reasonable sense of what the distribution of contexts is for each word, the size of the corpus has to be quite large.

Skip gram model

This section describes skip gram models, which is a simple model that tries to relate words to those appearing in their contexts. Our goal is to parametrize this model using word vectors. We will write it as a simple feedforward neural network model (Figure 3).

Input layer: The input layer consists of one-hot vector representation of word w_t. So the input layer has $|V|$ nodes.

Hidden layer: The hidden layer consists of d hidden units. The values of these units are computed using the weight matrix W of size $|V| \times d$.

Output Layer: The output layer has units to predict context words. Like the input word w_t, the context words are also represented as one-hot vectors. In figure 3 has four output units that represent the context of w_t for $k = 2$. So, the output units represent words $w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}$. The number of output units in this case is $4|V|$.

The output layer is computed using another parameter matrix W' of size $d \times |V|$ instead of $d \times 4|V|$. The same W' is used for predicting each context word. In other words, the context words are treated as if they were given as a bag.

Let's begin by seeing how to extract word vectors if this network is already trained. For the i^{th} word of the vocabulary, we can take the i^{th} row of the matrix W as a vector for word i.

$$v(w_i) = W_i^T$$

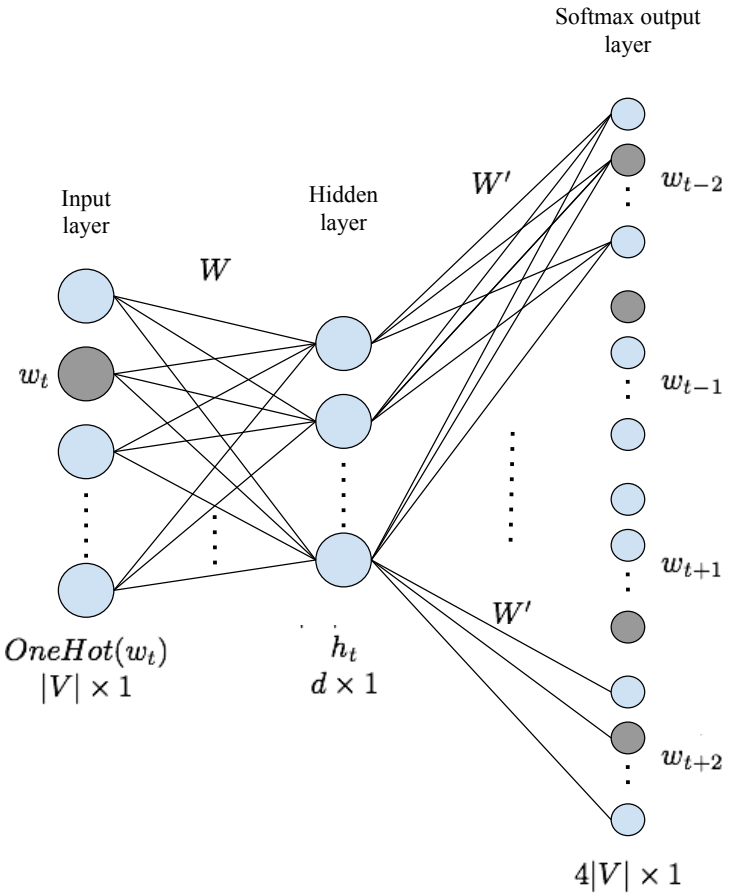


Figure 3: Neural network for skip gram model.

Alternatively, we can take the i^{th} column of the matrix W' as a word vector

$$v'(w_i) = W'_i$$

The difference between v and v' is that v describes a word when it's used as the input, whereas v' represents the word as a context word. Both v and v' produce d-dimensional word vectors. In practice, we typically take just $v(w_i)$ but the concatenation of v and v', $v^c(w) = [v(w)^T, v'(w)^T]^T$, is likely to be even better.

We can now write the probability of predicting a context word w_{t+j} given w_t in the neural network model in terms of the two types of word vectors. Specifically,

$$P(w_{t+j}|w_t) = \frac{e^{v(w_t)\cdot v'(w_{t+j})}}{\sum_{w\in V} e^{v(w_t)\cdot v'(w)}} \tag{1}$$

where the same distribution is used for $j = \pm 1, \pm 2$ ($k = 2$ in our example). In other words, all the context words are assumed to have come from the same distribution.

Training of skip gram model

To train this model our goal will be to find $v(w)$ and $v'(w)$, $w \in V$, so as to maximize

$$\max \sum_t \sum_{\substack{j=-k \\ j\neq 0}}^{k} \log P(w_{t+j}|w_t) \tag{2}$$

We could train the model by optimizing (2) using stochastic gradient ascent. Notice, however, that we would have to evaluate the normalization in (1) after each position (since the vectors change). Typically V is quite large (as the corpus is large) and using the above criterion is not feasible.

Hierarchical softmax

The hierarchical softmax is a binary tree that can be used to evaluate the probability of each word as in (1) potentially much faster than $O(|V|)$ per word. The key idea is that we represent words as leafs of a tree and model the path to each leaf as a sequence of (left-right) choices. Each choice specifies a branch to take and requires us to compare only two things (left and right). There are $O(\log |V|)$ such choices until we reach a particular leaf (word). Figure 4 shows the structure of the tree. The tree looks balanced here but in practice it should be optimized better so that typical words have short paths and rare words have longer ones (cf. Huffman coding).

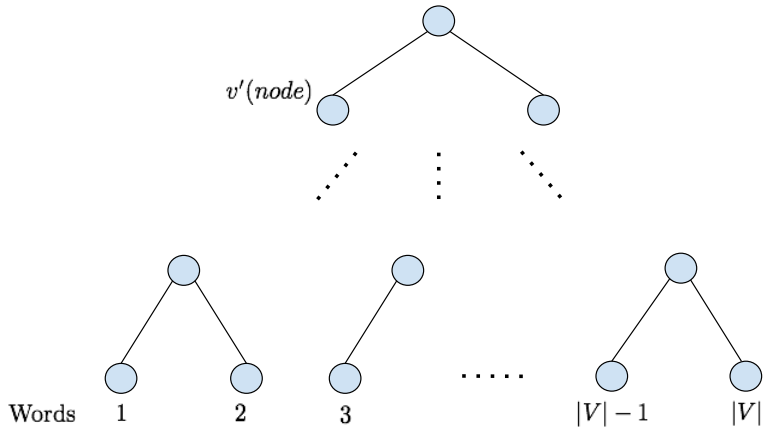


Figure 4: Hierarchical softmax tree. Each leaf of the tree represents one word in the vocabulary. Each node is equipped with a vector $v'(node)$ which represents a classifier about which branch (left or right) to take.

Suppose we wish to evaluate $P(w_{t+j}|w_t)$. Let $p = \{\text{root}, \ldots, w_{t+j}\}$ be the unique path of nodes in the tree that leads to the w_{t+j} leaf node. Let $s = \{+1, -1, \ldots, \}$ be the corresponding sequence of left (-1) and right (+1) choices following the nodes in path p. Note that $|s| = |p| - 1$. Each left-right choice is modeled with a classifier

$$P(\text{right}|\text{node } i, w_t) = \sigma(v(w_t) \cdot v'(\text{node}_i))$$

where non-leaf nodes in the tree are associated with parameters (node vectors) $v'(\text{node}_i)$. Taken together, the probability of seeing w_{t+j} given w_t is then evaluated as

$$P(w_{t-2}|w_t) = \prod_{i=1}^{|s|} \sigma(s_i \, v(w_t) \cdot v'(p_i))$$

Since $|s|$ is $O(\log |V|)$, the number of comparisons $v(w_t) \cdot v'(p_i)$ we need to evaluate is substantially smaller than $|V|$.

Word2Vec (with negative sampling)

Word2Vec is a popular method for estimating word embeddings. The motivation is the same as before but the typical implementation simplifies the problem even further so as to make it more efficient. word2vec casts the problem as a classification problem, i.e., trying to classify a word as appearing or not appearing in the context. The words in the same context (as defined before) therefore represent positive examples for this classifier while a randomly chosen other word from the vocabulary is (with high probability) a negative example.

The classifier is parameterized by word vectors. Specifically,

$$P(w \text{ in context}|w_t) = \sigma(v'(w) \cdot v(w_t))$$

where, again, each word such as w has two different vector representations depending on whether we condition on it (in which case we use $v(w)$) or whether it is part of the query (in context or not, represented by $v'(w)$).

The word vectors are trained by maximizing the log-likelihood of correctly classifying a context word as such and correctly classifying randomly chosen words as not. In other words, for each word w_{t+j} in the context of w_t, we replace the term $P(w_{t+j}|w_t)$ in Equation (2) by

$$\log \sigma(v'(w_{t+j}) \cdot v(w_t)) + \frac{1}{K} \sum_{l=1}^{K} \log \sigma(-v'(\tilde{w}_l) \cdot v(w_t))$$

where $\tilde{w}_l$ is a word sampled at random from a unigram distribution[2]. If we are estimating the vectors from a large corpus, the number of random negative examples, i.e., K can be quite small (e.g., 5).

[2] If $P_1(w)$ is the unigram distribution estimated from the same corpus, the negative words are actually sampled (heuristically) from $P(w) \propto P_1(w)^{3/4}$.

Evaluation

Once we have estimated the word vectors, we can try to evaluate what kind of semantic content their carry. Three possible evaluation methods are analogies, series completion and classification, visualized in Figure 5. We will primarily focus here on analogies.

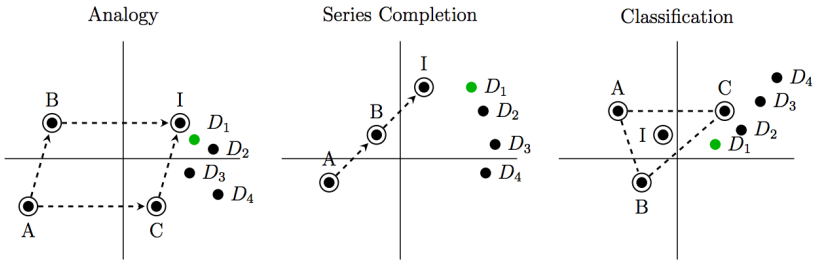


Figure 5: Three methods of evaluations: analogies, series completion and classification.

Analogies

A typical analogy question would ask: A is to B as C is to what? Assuming word vectors behave like vectors in some "semantic" vector space, we can try to answer such questions with simple vector operations. In other words, we would find word D such that its vector $v(D)$ is close to adding to $v(C)$ the part that takes us from $v(A)$ to $v(B)$, i.e., $(v(B) - v(A))$. More formally,

$$D = \arg\min_D \|v(D) - v(C) - (v(B) - v(A))\|$$

though other metrics are possible as well. For example, can sometimes get better results with cosine similarity (i.e., first normalizing word vectors).

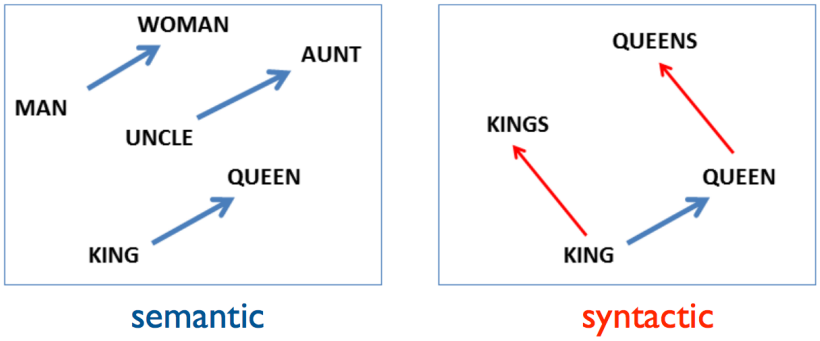


Figure 6: Two kinds of analogical reasoning: semantic and syntactic (Mikolov et al., 2013)

Analogies can be syntactic or semantic as shown in Figure 6. Figure 7 illustrates semantic analogies with word vectors. All the vectors in this case have been projected down to two dimensions (potentially with some distortions). The relationships between countries and their capitals is captured quite well by the word vectors.

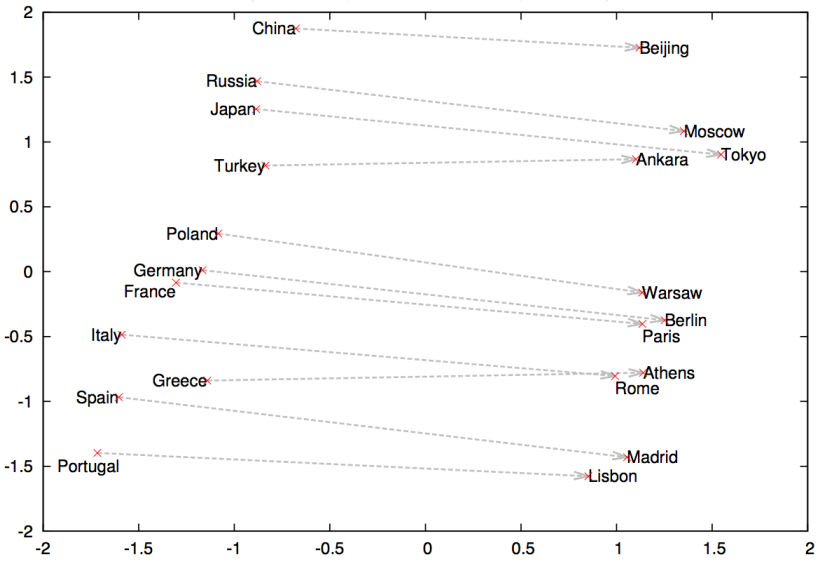


Figure 7: Word vector analogies (Mikolov et al., 2013)

The results in Figure 8 from Pennington et al. demonstrate that word vectors can capture syntactic analogies as well. The vectors here were not estimated via word2vec but using another common method GloVe (global vectors).

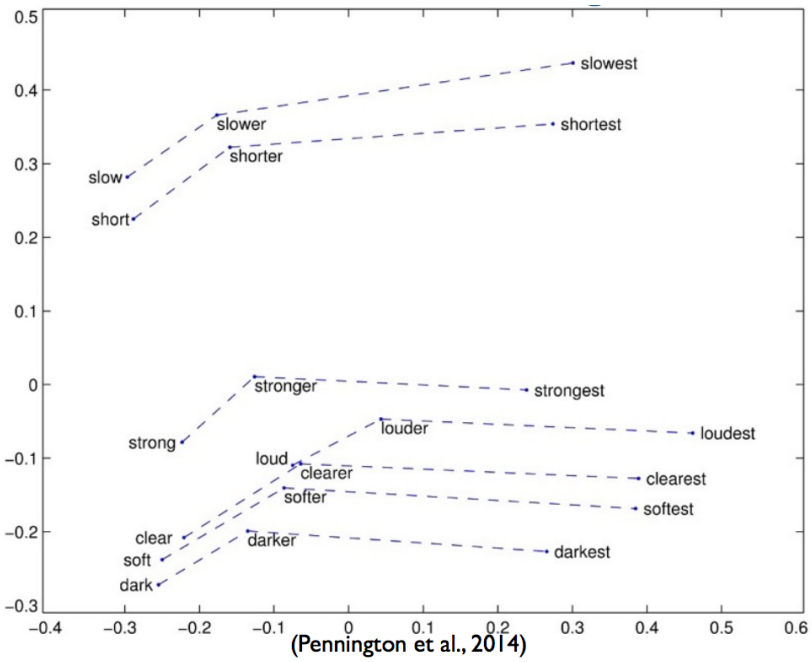


Figure 8: Adjectives and syntactic analogies (Pennington et al., 2014)

Comparison

There are a number of methods available for estimating word vectors. Table 9 provides some comparisons on typical semantic/syntactic and SAT analogies. The numbers represent percent correct. The semantic / syntactic analogies are open vocabulary results so random guessing would lead to a very low accuracy. For SAT questions, random guessing would get you 20% (1 in 5). However, an average student entering college gets 57% so word2vec's result of 42% is not bad at all.

	Google Analogies (cos)			Google Analogies (L_2)			SAT	
Method	Sem.	Synt.	Total	Sem.	Synt.	Total	L_2	Cosine
Regression	**78.4**	70.8	**73.7**	**75.5**	**70.9**	**72.6**	39.2	37.8
GloVE	72.6	71.2	71.7	65.6	66.6	67.2	36.9	33.6
SVD	57.4	50.8	53.4	53.7	48.2	50.3	27.1	25.8
Word2vec	73.4	**73.3**	73.3	71.4	**70.9**	71.1	**42.0**	**42.0**

Figure 9: Comparison of different methods (Hashimoto et al., 2015). The numbers in the table represent percent correct. The columns show results on Google Analogies dataset and SAT words.

Conclusion

This lecture covered how to build word vectors that can be used in many NLP tasks. The resulting low dimensional vectors capture semantic and syntactic meanings of the words. They are estimated from large corpora of text without any annotations, relying only on word

co-occurrences.

6.864 Advanced Natural Language Processing[1]

Lecture 14: Semantics of Language Grounding

3 November 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Vikas Garg.

In this lecture, we introduce the concept of grounding linguistic analysis using frameworks that have been shown to work well in three different settings. Our objective is to automate the process of mapping information from text instructions into a sequence of executable actions without requiring any annotated training examples. That is, our focus is on learning semantics as language 'programs'. For instance, we may want to operate simulators using instructions directly from the manuals, or design an autonomous game player.

General Setting

We will introduce ideas from three different papers that view the problem of understanding the semantics of text in a reinforcement learning setting. As we saw in homework 3, if annotations are available, we can map text into an abstract representation and then train simple models using these representations. However, the problem becomes considerable harder in tasks such as designing an autonomous game player because we need to model the interactions between the agent and the environment that it operates in without having human-created annotations to guide us. Reinforcement learning is a natural setting to study these kinds of problems; however, we need to resolve several challenges along the way as we shall see shortly.

Mapping Text Instructions to Actions

We will describe a reinforcement learning based approach to map text instructions to a sequence of executable actions.[2] The framework assumes having access to a reward function in order to measure the quality of the executed actions. The problem is set up as follows. Given a document d that comprises a sequence of sentences, we want to learn a mapping from d to a sequence of n actions $a = (a_0, a_1, a_{n-1})$ such that action a_i is predicted and executed sequentially before a_{i+1} is predicted. Note that this task is tailor-made for reinforcement learning with some differences from a typical setup such as dialogue management, where the learner finds a good policy through a trial-and-error process involving repeated interactions with the user. In our setting, since a human user is not involved, the cost of interaction is much lower than dialogue management. On the flip side, unlike dialog management, the emphasis is on learning the semantics by proactively

[2] S.R.K. Branavan, H. Chen, L. Zettlemoyer, and R. Barzilay. Reinforcement Learning for Mapping Instructions to Actions, *ACL*, 2009.

interacting with the environment. As a result, the state space is determined by the environment and thus can be much larger compared to the size a typical dialogue management task.

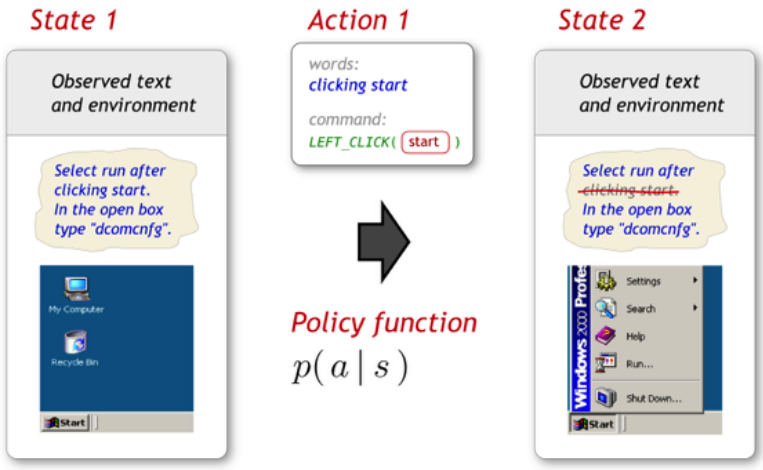


Figure 1: **Mapping from an instruction sequence to an action sequence in Windows 2000**. We show a state (observed text + environment) and possible actions (word selections + commands) that can be taken in this state. Some text is made available to the program, and it needs to figure out which object to select, which command to execute, and which words to select. Taking an action will result in transitioning to a new state with another set of objects and commands to execute, and only words not already mapped by previous actions will be available for selection.

Problem Setting

We first define the states, actions, and rewards to full characterize the reinforcement setting.

1. **Environment**: The environment $\mathcal{E}$ specifies the different objects that are available for interaction in a state, and the properties of these objects. $\mathcal{E}$ changes when an action is taken in a state according to a transition distribution $P(\mathcal{E}'|\mathcal{E}, c, R)$, where c is a command with parameters R that is executed as part of taking the action. R is a set of objects that are available in the environment. Note that this distribution is not specified to the system, and the learner uses a policy gradient based algorithm to avoid having to directly estimate this distribution.

2. **State**: Our state is a quadruple $(\mathcal{E}, d, j, W)$ where $\mathcal{E}$ is the current environment, j is the index of current sentence in the document d, and W is the set of words in the current sentences that have already been mapped by previous actions. The initial state s_0 for document d is $(\mathcal{E}_d, d, 0,)$: $\mathcal{E}_d$ is the unique initial environment for d.

3. **Action**: An action a is defined as a tuple (c, R, W), where the words W specify the command c with parameters R. To accommodate words that do not describe any actions, c is allowed to be a null command. Taking action a in state $s = (\mathcal{E}, d, j, W)$ results in transition to a new state s' according to the conditional distribution $p(s'|s, a)$: $\mathcal{E}$ transitions according to $p(\mathcal{E}'|\mathcal{E}, c, R)$, $W' = W \cup W_{a,j,d}$ ($W_{a,j,d}$ is the

set of words selected for a), and j increases by 1 when all the words in the current sentence have been mapped.

4. **Reward**: The reward $r(h)$ is a function of the history of the sequence of states and actions, $h = (s_0, a_0, \ldots, s_{n_1}, a_{n-1}, s_n)$, that are visited while processing a single document. The reward function provides a real-valued score that correlates with correct action selection. Reward may be immediately available following an action, or delayed until the feedback is available only after the last action, e.g., in applications such as Windows Help and Support.

Fig. 1 illustrates the setting in the context of a Windows 2000 system.

Training

The goal of training is to learn a *policy* to map states to actions. The input to training is a set D of documents, a blackbox that allows to sample from the transition distribution $p(\mathcal{E}'|\mathcal{E}, c, R)$, and a reward function $r(h)$. The output of training is to estimate the parameter vector θ of the policy distribution $p(a|s;\theta)$ such that the expected reward is maximized. The algorithm constructs a sequence of actions by repeatedly choosing the next action in the sequence given the current state. For a state $s = (\mathcal{E}, d, j, W)$, the set of possible actions is defined by the enumerating the unused words (the words that are not in W) in the current sentence j of document d under the environment $\mathcal{E}$.

The policy distribution is taken to be a log-linear model over the space of actions:

$$p(a|s;\theta) = \frac{e^{\theta \cdot \phi(s,a)}}{\sum_{a'} e^{\theta \cdot \phi(s,a')}},$$

with features represented as a joint state-action vector $\phi(s,a) \in \mathbb{R}^n$, as shown in Fig. 2. The features can characterize words, environment, and objects from the application etc. For example, we can have a feature corresponding to whether an object is bright or visible during a gaming application.

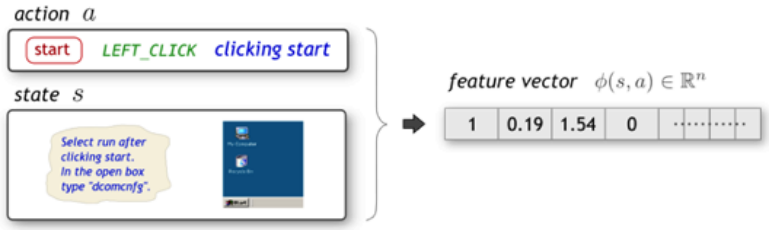


Figure 2: **Feature Vector**. Features are based on the state-action pair.

For any document $d \in D$, the expected reward is given by the value function $V_\theta(s) = \mathbb{E}_{p(h|\theta)} r(h)$, where h is the history of the document d,

and $p(h|\theta)$ is the probability of encountering history h and following a policy with parameters θ. Since the state space can be prohibitively large, we assume the policy distribution can be factorized as a product of simpler distributions over different time steps:

$$p(h|\theta) = \prod_{t=0}^{n-1} p(a_t|s_t;\theta)p(s_{t+1}|s_t,a_t).$$

We average the value function over all the documents in D. We are faced with an immediate issue in order to estimate the parameters θ that optimize the value function: directly computing the derivative of the value function is intractable since we need to compute an expectation over all the possible histories. To circumvent this, we apply a policy gradient algorithm that estimates a noisy estimate of the value function by performing stochastic gradient ascent using only a subset[3] of the histories, that are obtained by acting in the environment.[4] We perform the following steps (see Fig. 3) on each document $d \in D$ during each iteration $i \in \{1, 2, \ldots, T\}$.

1. Initialize θ to a small random value.
2. Sample history $h \sim p(h|\theta)$ as follows:
   - Sample action $a_t \sim p(a|s_t;\theta)$ for $t \in \{0, 1, \ldots, n-1\}$.
   - Execute a_t on state s_t: $s_{t+1} \sim p(s|s_t, a_t)$.
3. Observe the reward $r(h)$.
4. Update the parameters θ based on $r(h)$.

[3] In practice, even one history is known to work well.

[4] Policy gradient algorithms optimize a non-convex objective by performing stochastic gradient descent. Therefore, they are susceptible to yielding a local optimum. They scale to large state spaces, however, and are known to perform well in practice.

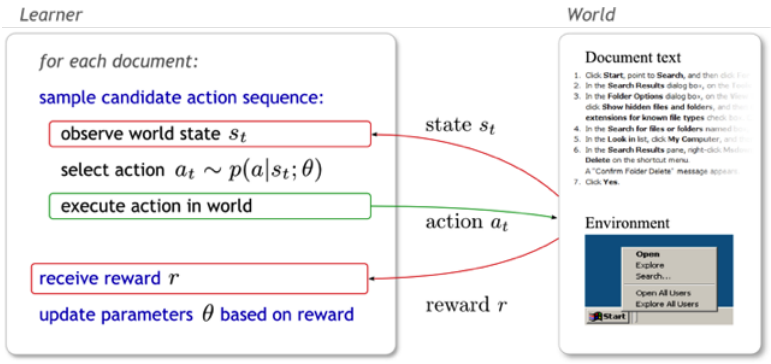


Figure 3: **The learning algorithm for mapping text instructions to a sequence of actions**.

Once the parameters θ are learnt, we make hard decisions (i.e. we choose only one action) during test and select the actions according to the mode of this distribution $p(a|s;\theta)$.

Winning games by extracting information from manuals

We will now describe another application,[5] where automatically extracted information helps improve the performance of control applications such as games. Consider, for instance, the complex strategy game Civilization II.[6] This game requires extensive planning against an active adversary, and the branching factor (number of possible moves) for multiplayer strategy is of the order $\approx 10^{20}$. As a result, finding a winning strategy is challenging even for humans, and the (human) players have to rely on some guide about promising tactics to narrow down the search space. Therefore, it makes sense to leverage text information from manuals to improve the performance of control algorithms used in games. However, since the state space is prohibitively large, we cannot manually annotate the information that is relevant to the different states. We address this challenge by studying the problem in a Monte-Carlo search framework, where we learn text analysis based on some feedback such as game score.

[5] S.R.K. Branavan, D. Silver, and R. Barzilay. Learning to Win by Reading Manuals in a Monte-Carlo Framework, *ACL*, 2011.

[6] Civilization II is a multi-player game set on a grid-based map, where each grid location represents some area of land or sea. Each location is associated with various resources and terrain attributes. The goal of each player is to control multiple cities as well as other units (e.g. as settlers and explorers). The game is won by gaining control of the entire map.

Problem Setting

We set up the game in a stochastic environment using a Monte-Carlo framework. The game is represented as a Markov Decision Process (MDP) quadruple (S, A, T, R), where S is the set of possible states, A is the set of valid actions, R is a utility or reward function, and T is a stochastic state transition function. For instance, let $s \in S$ be the current state and $a \in A$ be a valid action in s. Then, $T(s'|s, a)$ gives the probability of next state being s' when action a is taken in state s. More concretely, we model a game in terms of the following components.

1. **State**: The game state is defined to be the map of the world, the attributes of each map location, and the attributes of each player's cities and units. A state encodes the different attributes of the Civilization game world such as available resources and city locations.

2. **Action**: Performing an action in a state results in transition to a new state according to the stochastic function T.[7] Building a city, for instance, is one action. On average, each player controls many units and several actions are available in each unit, thereby resulting in a huge action space ($\approx 10^{21}$ actions for the experiments in the paper). The problem of searching in such a large space is mitigated by assuming that the actions of each unit are conditionally independent of the actions of all other units of the same player given the state.

3. **Utility**: The utility function is used to evaluate the outcomes. Each transition to a state s' fetches a reward $R(s) \in \mathbb{R}$. Since the game

[7] Since T is not known a priori, we estimate it by sampling transitions from the program that encodes the game.

result is known only at the end, $R(s)$ can be defined based on the likelihood of winning the game from state s; e.g., we can take $R(s)$ to be the intermediate game score that Civilization II provides for each player. This score is a noisy indicator of the current performance of a player. For a two-player game, we can simply use the ratio of their game scores as the utility function.

Training

The goal of the training is to learn a policy, i.e., to dynamically select the best action for the current state s_t. Training is done using a Monte-Carlo search algorithm (Fig. 4): multiple *roll-outs* are simulated[8] starting from the current state s_t and their outcome is observed. More specifically, the algorithm repeatedly selects and executes actions by sampling state transitions according to T, starting from s_t. At the completion of the game (at some later time τ), the final utility is measured and the actual action sequence is selected to be the one with the best final utility.

[8] for example, by playing against the game's built-in AI.

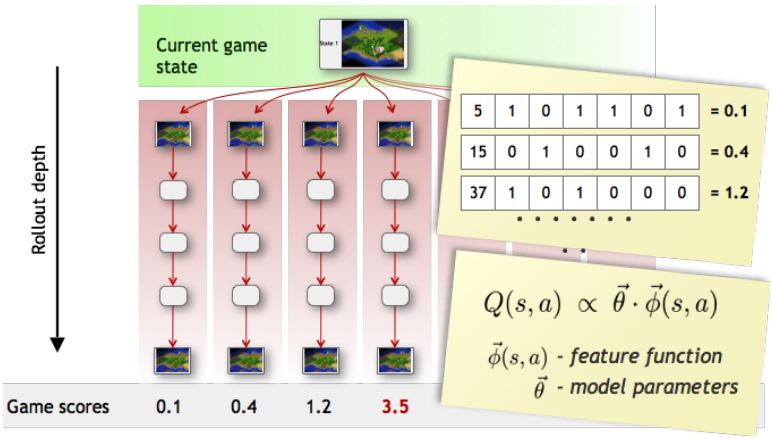


Figure 4: **Monte-Carlo search algorithm**. Multiple action sequences are simulated starting from the current state by making copies of the current state and environment settings, and sampling state transitions according to T for each copy. The action with the best corresponding final roll-out utility is selected as the actual game action.

The success of this procedure relies on its ability to make a fast, local estimate of the action quality at each step during a roll-out. An action value function $Q(s,a)$ is used as an estimate of the expected outcome of action a in state s. This function guides action selection during the roll-outs. Usually, actions are selected to maximize the $Q(\cdot,\cdot)$ function, however, sometimes other actions are also randomly explored[9] using an ϵ-greedy strategy[10] in the hope that they might be more valuable than the action with the highest $Q(s,a)$ value. As the accuracy of the $Q(s,a)$ improves, the quality of action selection also improves and vice-versa. Since the branching factor is huge, it is infeasible to enumerate $Q(s,a)$ for different values of s and a. Instead, the $Q(\cdot,\cdot)$ function is

[9] This tension between exploring a new strategy versus sticking to the current strategy is the classic *exploration-exploitation* tradeoff in the reinforcement learning literature.

[10] In the ϵ-greedy strategy, the action with highest value in the current state is chosen with probability $(1-\epsilon)$, while a random action is chosen with the remaining probability.

modeled as a linear combination of the state and active attributes:

$$Q(s,a) = w \cdot \phi(s,a),$$

where $\phi(s,a)$ is a real valued feature vector and w is a weight vector. The parameters w can be learned based on the feedback from the roll-out simulations; for instance, stochastic gradient descent may be used to compare the current predicted $Q(s,a)$ against the roll-out utility that is observed eventually.

Incorporating Textual Information

So far, we have not described how the information, which is automatically extracted from the text, can be used to inform both model structure and parameter estimation. Toward that goal, we modify the function $Q(s,a)$ to take into account the words of the document in addition to state and action information.[11] Since most of the text in the whole document will likely not be relevant to the current state, we model the relevance of each state as a hidden variable. Specifically, given a strategy document d, we want to find a sentence y_i that is most relevant to the current state s_t and action a_t. We model the relevance decision as a log-linear distribution over sentences using a feature function $\phi'(y_i, s_t, a_t, d)$.

[11] This information is not known a priori and therefore must be encoded during learning.

We also have hidden variables to separate the words that describe actions or state attributes from the rest of the sentence (called background). This task of predicate labeling is modeled as a sequence prediction problem since the word label assignments are likely to be mutually independent. Since there may be dependencies between the words that describe state and action attributes, the words are labeled in a dependency order (instead of the word order) to allow a word's label decision to depend on its parent in the dependency tree.

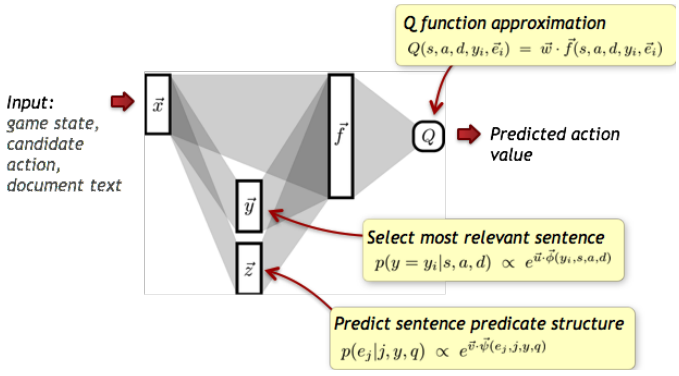


Figure 5: **Structure of the neural model**.

A four layer neural network is used as the model as shown in Fig. 5. The input layer represents the current state s, a candidate action

a, and document d. In the second layer, we have two separate sets of units, y to encode the state relevance and z to encode the predicate labeling. These units use the standard softmax activation, and thus effectively model the sentence relevance and predicate labeling decisions via log-linear distributions as described above. Each unit in the third layer corresponds to a fixed feature function $f(s, a, d, y_i, z_j) \in R$ that is deterministically computed using the active units y_i and z_j of the softmax layers. The last layer corresponds to the output, and encodes the action-value function $Q(s, a, d)$, which depends on the document d in addition to state s and action a. $Q(s, a, d)$ is simply a weighted linear combination of the output from the units in the third layer. The model parameters can be updated via backpropagation.

Deep Reinforcement Learning for Text-based Games

We now describe a deep reinforcement learning based approach for language grounding in text-based Multi-User Dungeon (MUD) games.[12] The idea is similar to the design of automatic game player to learn strategies for Civilization in that text analysis and control strategies are learned jointly by using the feedback obtained from roll-out simulations. However, unlike the previous setup (where states were fully observable), we no longer observe the state representation, which now has to be inferred from the textual description. That is, we now operate in a *Partially Observable Markov Decision Process (POMDP)* setting. The deep reinforcement learning framework jointly learns the state representations and action policies using the reward feedback.

[12] K. Narasimhan, T. Kulkarni, and R. Barzilay. Language Understanding for Text-based Games Using Deep Reinforcement Learning, *EMNLP*, 2015.

Problem Setting

The game is represented by a 5-tuple (H, A, T, R, ψ), where H is the set of all possible game states,[13] $A = \{(a, 0)\}$ is the set of all commands, $T(h'|h, a, o)$ is the stochastic transition function and $R(h, a, o)$ is the reward function. The player receives a varying text description produced by a stochastic function $\Psi : H \rightarrow S$. The reinforcement learning system has the following description.

[13] Note a slight departure from the previous notation: we use H to denote the set of states, in order to use the symbol S for the space of possible text descriptions.

1. **State**: The game state $h \in H$ keeps track of attributes such as player's location, and the time of the day etc. The function Ψ then converts this state h into a textual description s of the location the player is at.

2. **Action**: The action-object pair $(a, o) \in A$ is a command that can be executed and results in a transition to state h'.

3. **Reward**: The rewards are given to the player only at the completion of the in-game quests. The player takes an action a on seeing the

description s by evaluating the action's expected long-term reward using a state-action value function $Q(s, a)$,[14] which is learned as part of the training process.

[14] Note that the Q-learning is *model-free* technique for learning the Q-function. An alternative paradigm is called model-based learning, where all the state-action pairs are explicitly stored in tables. Model-based learning incurs a huge storage overhead, especially when the state space is large, as is the case in our setting.

Training

Starting from a random initialization, the player continuously updates its Q-values by playing the game and observing the resulting rewards. The Q function is updated iteratively using the Bellman equation:

$$Q_{i+1}(s, a) = \mathbb{E}(r + \gamma \max_{a'} Q_i(s', a') | s, a),$$

where γ is discount factor for future rewards, and the expectation is taken over all game transitions that involve the player while taking action a in state s.

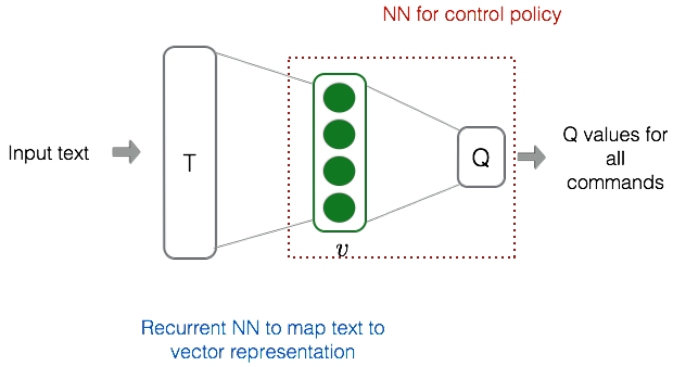


Figure 6: **Neural Model for Text-based Games**.

Again, an ϵ-greedy strategy can be employed to tradeoff exploitation for exploration. In order to learn the Q-value function, we can use a Deep Q-Network (DQN). The DQN approximates the Q-value function for all possible actions simultaneously given the state description.

The deep neural model is shown in Fig. 6. The model consists of two components. The first component, *representation generator*, converts the textual description of the current state into a vector. The second component, *action scorer*, takes input from the representation generator and outputs the Q-score for all actions. We now describe the two components in detail.

1. **Representation Generator**: The representation generator, ϕ_R, converts the raw text displayed to the player to a vector representation. Since a bag-of-words representation does not capture the higher order structures in text, an LSTM model is used as a representation generator. Recall from previous lectures that the LSTM is a recurrent neural network that can recognize long-range patterns in the text, and thus is able to encode the underlying semantics of sentences to an extent. The LSTM network takes in embeddings w_k of

the words in a description s and produces the output vectors x_k at each step. The final representation v_s is obtained by using a *mean pooling* layer that computes the element-wise mean over the output vectors x_k.

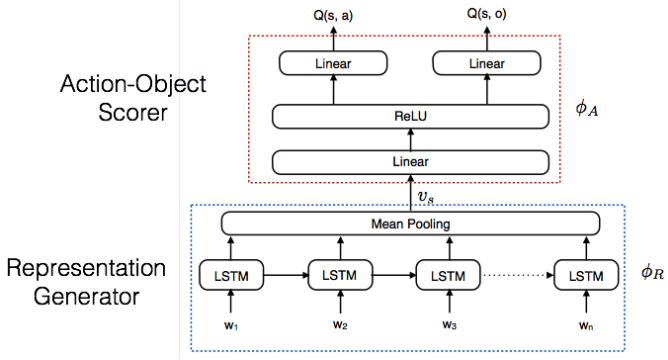


Figure 7: **Architecture of the LSTM-DQN model**. The representation generator is shown in the blue box, while the action scorer is shown in the red box.

2. **Action-Object Scorer**: The action scorer, ϕ_A, produces scores for the set of possible actions when given the vector representation v_s as input. A multi-layered neural network is used as the action scorer. All the actions are simultaneously scored; this is computationally more efficient that obtaining a score for each state-action pair separately (Fig. 7).

Therefore, the Q-score for the pair (s, a) can be approximated as the composition

$$Q(s, a) \approx \phi_A(\phi_R(s))[a].$$

The parameters θ_R of ϕ_R, and θ_A of ϕ_A can be learned using a stochastic gradient descent approach that reduces the discrepancy between the predicted Q-value of the current state, and the expected Q-value given the reward and the value of the next state.

6.864 Advanced Natural Language Processing[1]

Lecture 15: Machine Translation

5 November 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Austin Freel.

This lecture introduces Machine Translation, which is a subfield of computational linguistics that investigates the use of software to translate text (or speech) from one language into another. The lecture will introduce some of the history and challenges of this problem as well as some traditional machine translation approaches.

Challenges of Machine Translation

Below are just some of the challenges associated with Machine Translation. Although each problem is fairly straightforward, together they can give an idea of how different two languages may translate the same sentence.

Lexical Ambiguity

Words or phrases may have more than one meaning. For example, in the phrases "**book** the flight" and "read the **book**", the English word **book** may be translated into several different words in another language.

Differing Word Order

In English, phrases often follow the subject-verb-object structure, but in another language it can be different. Japanese, for example, is often subject-object-verb.

Syntactic Ambiguity

We have seen syntactic ambiguity before in the form of parse trees, where the same sentence may have several valid trees. For example, in the sentence "She saw the man on the mountain with a telescope", we do not know for sure who it is that has the telescope.

Pronoun Resolution

Pronoun resolution refers to resolving references to earlier or later items in the discourse. If I say "Michael helped Mary, he was kind", then most humans will unconsciously assume that Michael is the kind one, but it is harder for a computer to make this kind of assumption. In translation, if we are translating into a language which marks the gender of pronouns, for example, then this resolution is important.

Differing Treatments of Tense

Not every language has the same tenses. If one has ever taken a Spanish class, for example, he or she knows that verb conjugation is based on the tense. For example, in English if I said "Josh went to Korea", that could imply that he went and came back, or that he went and is still there. However, in other languages (like Spanish), the verb *went* translates differently in these two cases.

Introduciton to Statistical Machine Translation

The above problems elucidate the complexities that all languages have and begin to give insight into why statistical approaches might be more useful than trying to completely model each language. The translation problem began to become formalized after WWII, when people began to think about translation as a deciphering problem. For the next couple decades, people attempted to tackle the problem by trying to write rules for all the problems given above, and without large computation power, statistical approaches did not yet make sense. In 1966, as progress on the machine translation problem was slowing, the U.S. withdrew its funding. Although other countries still worked on the problem thereafter, it was not until the 1990's that machine translation came back in full swing in the U.S., this time in the form of statistical machine translation, which IBM had the computing power to facilitate.

Overview

The basic idea behind statistical machine translation is that we have parallel corpora for some set of language pairs. We can these use the parallel corpora as the training set for our model. The idea is that if we have a mapping of sentences from one language to another, then given a new sentence, we should be able to try and predict its translation just based on the samples we have. Most sentences, however, will not have been seen before verbatim, and so we must apply more complex techniques, a few of which are described below.

Phrase-Based Translation

Phrase-based translation can be thought as a happy medium between doing trivial word-to-word translations and trying to translate whole sentences at once. Our phrases are just sequences of words and can be of arbitrary length. It is worth noting that, in practice, these phrases are usually chosen to be idiomatic phrases over linguistic phrases.

More details of phrase-based translation are shown at the end of the lecture.

Syntax-Based Translation

Syntax-based translation is based on the idea of translating syntactic units rather than single words or sequences of words (phrases). In other words, we are translating partial parse trees of sentences from one language to another. One advantage to this approach is that we can now better model reordering. Phrase-based translation provides us with some local reordering, but only for phrases seen in training, and thus only for the length of phrases seen in training (which usually won't be that long since the number of phrases grows exponentially with allowed length). Note that this is still statistical MT since we are using annotated parallel-data.

Semantics-Based Translation

The idea behind semantics-based translation is that the more linguistic structure that is used, the better the translation will be. Examining our canonical machine translation pyramid shown below, we see that semantics are at the top, defining the fundamental differences of each language.

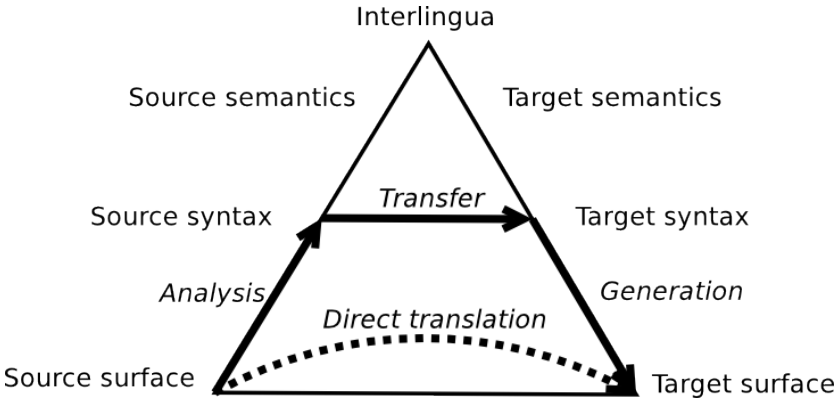


Figure 1: **Machine Translation Pyramid**. The bottom left corner represents the source surface, which in our case is the source language. The bottom right corner then represents the language we are translating into. As we move up the pyramid, we are moving away from pure statistical translation into more syntactic and semantic-based approaches.

The Abstract Meaning Representation (Knight et al., ongoing) is a popular semantic representation of sentences. There is an online AMR Bank of English sentences paired with their AMR representations that is constantly being added to with the hopes that people will use it to do more research into semantic-based translation.

Noisy Channel Model

We were first introduced to the noisy channel model and its relation to machine translation in Lecture 1. To review, pretend that we are translating from French (source language) to English (target language). The noisy channel model has two components:

1. *Language Model* - assigns a probability $p(e)$ for any sentence $e = e_1 \dots e_\ell$ in English. We can, for instance, use our favorite n-gram language model here.
2. *Translation Model* - assigns a conditional probability $p(f|e)$ to any French/English pair of sentences. The parameters of this model are estimated from our corpora of translation examples. For a given English sentence $e_1 \dots e_\ell$, we will predict both the length m of the French sentence and its words $f_1 \dots f_m$.

Our translation model is then defined as follows:

$$e^* = \underset{e}{argmax}\ p(e|f) = \underset{e}{argmax}\ \frac{p(f|e)p(e)}{p(f)}$$

where e^* represents the best English translation of the given French sentence f when iterating over all possible English translations e. However, we can note that the probability of the given French sentence $p(f)$ will be the same for any e, and therefore rewrite the equation as follows:

$$e^* = \underset{e}{argmax}\ p(f|e)p(e)$$

We can see that the score for a potential translation e is just the product of the outputs of the language and translation models. To give some insight into why we need both, we use an example from the Koehn and Knight tutorial. Specifically, we are translating the sentence "Que hambre tengo yo" from Spanish to English. Solely based on $P(Spanish|English)$, we get the following translations and their scores:

What hunger have	$P(S\|E) = 0.000014$
Hungry I am so	$P(S\|E) = 0.000001$
I am so hungry	$P(S\|E) = 0.0000015$
Have I that hunger	$P(S\|E) = 0.000020$

All this tells us is that given these four English sentences, the one most likely to produce the Spanish sentence above based on the translation model is the final sentence. However, the final sentence (and the first two for that matter) are clearly not correct valid English sentences. This is why we must also include the language model, to avoid translations that don't make much sense in the target language. Now, with

$P(Spanish|English) \times P(English)$, we have:

What hunger have	$P(S\|E)P(E) = 0.000014 \times 0.000001$
Hungry I am so	$P(S\|E)P(E) = 0.000001 \times 0.0000014$
I am so hungry	$P(S\|E)P(E) = 0.0000015 \times 0.0001$
Have I that hunger	$P(S\|E)P(E) = 0.000020 \times 0.00000098$

And we can see that the third translation now has the highest probability as expected. We show how one might estimate the parameters of the translation model later in this lecture, but first we look at how one might evaluate a machine translation system.

Evaluation of Machine Translation Systems

Human evaluations of machine translation take a long time and can be quite expensive. Therefore, research into automated machine translation system evaluation began. The system that we discuss is Bleu. In this system, we call machine translations of a sentence Candidates, and the respective human (correct) translations References. For example, we might have:

Candidate 1: some mat had on it cat
Candidate 2: the cat sat on the rug
Reference 1: the cat sat on the mat
Reference 2: there is a cat on the mat

It is clear that Candidate 2 is better, but how might a computer figure this out? We examine some possible methods below.

Attempt 1: Unigram Precision

The unigram precision of a candidate translation is given by:

$$C/N_c$$

where C is the number of words in the candidate which are in at least one reference translation, and N_c is the number of words in the candidate. Applying this to the example above, Candidate 2 gets a score of $5/6$ while Candidate 1 only gets a score of $3/6$. Seems pretty good, but there is a problem. Imagine I generate a new candidate translation "the the the the the the". It gets a score of $6/6$, so something must be changed.

Attempt 2: Modified Unigram Precision

To solve the above problem, we introduce “clipping", which just means setting a maximum score for any candidate word. For example, to avoid the above issue, we may try to set the cap of the word “the" to be 2. This means our trick sentence “the the the the the the" now only gets a score of 2/6, good!

It is also worth noting here that we can expand this model to other n-grams. For example, using bigram precision Candidate 1 gets a score of 0/5 since none of its five bigrams appear in any references, while Candidate 2 gets a score of 4/5.

But there is another problem. Imagine we have the candidate “on the". It will receive a perfect unigram and bigram score of 1, and clipping won't help. So we can see that precision isn't enough here, which means it might be a good idea to look at recall. We define recall to be

$$C/N_r$$

where C is the number of n-grams in the candidate that are correct, and N_r is the number words in references. The problem, however, is that the system considers multiple reference translations, each of which may use a different word choice to translate a given source word. Most good translations will only contain one of these choices, and so we might run into a problem like the following:

Candidate 1: I always invariably perpetually do

Candidate 2: I always do

Reference 1: I always do

Reference 2: I invariably do

Reference 3: I perpetually do

The first candidate recalls the most words, but is a bad translation.

So how do we avoid picking the “on the" candidate? We can observe that the reason this candidate gets high precision and is immune to clipping is that it is quite short in length. In fact, almost any short candidate composed of common words is likely to be picked as a correct translation, so we will now try to avoid really short candidates.

Attempt 3: Sentence Brevity Penalty

1. For each candidate, compute it closest matching reference **in length**. For example, if our candidate is length 17 and we have references

of length 14, 16, and 22, the best match is the reference with length 16.

2. We define ℓ_i to be the length of candidate i and r_i to be the length of the best reference match for candidate i, and then compute:

$$brevity = \frac{\sum_i r_i}{\sum_i \ell_i}$$

3. We now compute the brevity penalty:

$$BP = \begin{cases} 1 & \text{if } brevity < 1 \\ e^{1-brevity} & \text{if } brevity \geq 1 \end{cases}$$

For example, if $r_i = 1.1 \times \ell_i$ for all i (candidates are all 10% too short), then $BP = e^{-0.1} = 0.905$.

Final Bleu Score

We define the corpus precision for any n-gram to be:

$$p_n = \frac{\sum_{C \in \{Candidate\}} \sum_{ngram \in C} Count_{clip}(ngram)}{\sum_{C \in \{Candidate\}} \sum_{ngram \in C} Count(ngram)}$$

This is the number of correct n-grams in the candidates *after clipping* divided by the total number of n-grams in the candidates.

The final Blue score is then:

$$Bleu = BP \times (p_1 p_2 p_3 p_4)^{1/4}$$

This is the BP multiplied by the geometric mean of the unigram, bi-gram, trigram, and four-gram precisions. The Bleu score has been shown to be highly correlated with human judgments, and is thus considered a good benchmark score for a machine translation system. Take a look at the Bleu paper for more details.

The Sentence Alignment Problem

The problem is that for a pair of texts in two different languages, the respective number of sentences might not be the same in each. For example, say we have a sequence of 1003 English sentences and a translated sequence of 987 French sentences - which English sentences correspond to which French sentences?

Sentence Length Methods

We now describe a method developed by Gale and Church in 1993. The method assumes that paragraph alignment is known, but that sentence alignment is not known, which is a reasonable assumption that reduces the complexity of the problem quite a bit. We define:

$$\ell_e = \text{length of English sentence, in characters}$$
$$\ell_f = \text{length of French sentence, in characters}$$

We also assume that, given length ℓ_e, length ℓ_f has a gaussian/normal distribution with mean $c \times \ell_e$ and variance $s^2 \times \ell_e$, where c and s are some constants (in Gale and Church, these constants were determined empirically from the Union Bank of Switzerland translation corpus). Since we have this distribution, we can define a cost to any pair of lengths ℓ_e and ℓ_f:

$$Cost(\ell_e, \ell_f)$$

Note that this cost just depends on differences in lengths. We can also define a cost of matching i English sentences with j French sentences as:

$$C_{ij}$$

For example, if we said that 2 English sentences were aligned with 1 French sentence, the cost would be C_{21}. To understand what this might look like, consider the following possible alignment:

e_1	f_1
e_2	
e_3	f_2
e_4	f_3
e_5	f_4
	f_5
e_6	f_6
e_7	f_7

If we let ℓ_i be the length of e_i and also let m_i be the length of f_i, then the cost of the above alignment is:

$$\begin{aligned} Cost = & Cost(\ell_1 + \ell_2, m_1) + Cost_{21} + \\ & Cost(\ell_3, m_2) + Cost_{11} + \\ & Cost(\ell_4, m_3) + Cost_{11} + \\ & Cost(\ell_5, m_4 + m_5) + Cost_{12} + \\ & Cost(\ell_6 + \ell_7, m_6 + m_7) + Cost_{22} \end{aligned}$$

This is the cost of a single possible alignment. We can use dynamic programming to find the lowest cost alignment.

Now that we know how to take two parallel texts and align their sentences, we can return our focus to the translation aspect.

The IBM Translation Models

We now return to our original question of estimating the parameters of the translation model. Recall from earlier in the lecture that our noisy channel model depends on both a language model and a translation model. We now describe some approaches for generating the latter, specifically IBM Model 1 and IBM Model 2.

We define $e = e_1 \dots e_\ell$ to be an English sentence and $f = f_1 \dots f_m$ to be the Foreign translation (note that f is no longer French, but rather just some arbitrary foreign language). The goal of the translation model is to find $p(f|e)$, but as we said before, whole sentences do not often reoccur. This means we need to break the sentence down somehow, and the IBM models do this through alignments. An alignment is essentially a mapping of words in f to words in e. Specifically, for an alignment a, we define a_j to be the index of the word in e from which word f_j was derived from. For example, say we have the following:

$$\begin{aligned} e &= \text{I love big dogs} \\ f &= \text{evol sgod gib} \end{aligned}$$

Let the reverse of each foreign word f_j represent its English translation (for example, “evol" is translated to be “love" in English). The alignment for this example is thus given by $a = \{2, 4, 3\}$, since $f_1 \to e_2$, $f_2 \to e_4$, and $f_3 \to e_3$. It is worth noting here that we are not zero-indexing, and the reason for that is that there is always a chance that some foreign word f_j does not match one of the English words, in which case it will point to *null*, which we define to be at e_0. One should also observe that each f_j maps to only one e_i, but it is not

wrong for a given e_i to map to multiple f_j words. Another way to look at this is that a will always be the same length as f, but a itself may contain several duplicate entries if multiple words in f were derived from the same word in e.

We can now define our formulation of the translation probability as follows:

$$p(f|e) = \sum_{a \in A} p(f, a|e)$$

where A is the set of possible alignments.

Using the chain rule, we have that:

$$p(f, a|e) = p(a|e) \cdot p(f|a, e)$$

The latter term, $p(f|a, e)$, is simply the probability of the foreign sentence given the alignment and the English sentence, which is itself just a product of the probability of generating each foreign word f_j given the English word it points to in the alignment e_{a_j}:

$$p(f|a, e) = \prod_j p(f_j|e_{a_j})$$

The final step is to calculate $p(a|e)$, which is the probability of the alignment given the English sentence. There are several ways to do this, and this is actually what fundamentally differentiates IBM Model 1 from IBM Model 2.

Model 1

In this first model, we assume that there is a uniform probability distribution over the $\ell + 1$ possible English words (recall that the English sentence $e = e_1 \dots e_\ell$, and there is also the possibility that a foreign word f_j is aligned with $e_0 = null$). Since there are m words in the foreign sentence f, and each can map to any of the $\ell + 1$ English words, there are a total of $(\ell + 1)^m$ possible alignments. Therefore, we define $p(a|e)$ in the first model as:

$$p(a|e) = \frac{c}{(\ell + 1)^m}$$

where c is a number representing the probability of generating a string of length m. The reason we include this is because different translations will have differing lengths m, and we may want to weight certain translations lower, for example, if their difference in word lengths from

the original are quite large.

For IBM Model 1, we therefore have:

$$p(f|e) = \sum_{a \in A} \frac{c}{(\ell+1)^m} \prod_j p(f_j|e_{a_j})$$

Model 2

In the second model, we no longer assume a uniform probability over the English words. Instead, we now define parameters:

$$p(a_j|j, \ell, m)$$

as the conditional probability of foreign word f_j yielding alignment a_j given foreign sentence length m and English sentence length ℓ. We can now define $p(a|e)$ in this model as:

$$p(a|e) = \prod_j p(a_j|j, \ell, m)$$

Therefore, for IBM Model 2 we have:

$$p(f|e) = \sum_{a \in A} \prod_j p(a_j|j, \ell, m) \cdot p(f_j|e_{a_j})$$

It is worth noting that Model 1 is just a special case of Model 2. More specifically, if we have that:

$$p(a_j|j, \ell, m) = \frac{1}{\ell+1}$$

then Model 2 becomes Model 1. In practice, Model 2 is much harder to learn than Model 1, so people often initialize it with the probabilities generated by Model 1. To learn more about the details of these models, these notes are quite helpful.

If we are given a training corpus that is annotated with alignment variables, then we can simply use maximum-likelihood estimates to calculate the parameters $p(a_j|j, \ell, m)$. However, this is almost never the case, which means that the alignment variables are in fact hidden. Therefore, to estimate them we use Estimation Maximization (EM). To understand what this looks like, take a look at the example below.

Alignment Variables EM Example

In this example, we assume that we have two corpuses, each of which only contain a single sentence and its translation. Corpus 1 contains the sentence “b c" and its translation “x y". Corpus 2 contains the sentence “b" and its translation “c". In this example, b, c, x, and y are all

words.

In this example, we assume that there are no empty node alignments. That means that for Corpus 1, the possible alignments are $a = \{2,1\}$ and $a = \{1,2\}$, which we refer to as A1 and A2 respectively. Just to review, $a = \{2,1\}$ means that, in Corpus 1, y is generated from b and x is generated from c. In Corpus 2, the only possible alignment $a = \{1\}$, which we call A3, means that y is generated from b. From this setup, we can clearly see from Corpus 2 that y is associated with b, and thus that x should be associated with c. We now show how the EM algorithm figures this out.

We begin with some random initialization of our parameters. For our example, we will say:

$$p(x|b) = p(y|b) = p(x|c) = p(y|c) = 1/2$$

We know the probability of an alignment to be:

$$p(a|e,f) = \frac{p(f,a|e)}{\sum_{a \in A} p(f,a|e)}$$

So, to calculate the probability of alignment A1 for example, we would do:

$$\begin{aligned} p(A1|\text{“b c", “x y"}) &= \frac{p(\text{“x y", } A1|\text{“b c"})}{p(\text{“x y", } A1|\text{“b c"}) + p(\text{“x y", } A2|\text{“b c"})} \\ &= \frac{p(x|c)p(y|b)}{p(x|c)p(y|b) + p(x|b)p(y|c)} \\ &= \frac{(1/2)(1/2)}{(1/2)(1/2) + (1/2)(1/2)} \\ &= 1/2 \end{aligned}$$

We can calculate the probability of alignment A2 in the same way:

$$\begin{aligned} p(A2|\text{“b c", “x y"}) &= \frac{p(x|b)p(y|c)}{p(x|c)p(y|b) + p(x|b)p(y|c)} \\ &= \frac{(1/2)(1/2)}{(1/2)(1/2) + (1/2)(1/2)} \\ &= 1/2 \end{aligned}$$

And since A3 is the only choice for Corpus 2, its probability is trivially just 1:

$$p(A3|\text{“b", “y"}) = 1$$

To review, we have:

$$\begin{aligned} p(A1) &= 1/2 \\ p(A2) &= 1/2 \\ p(A3) &= 1 \end{aligned}$$

Now we can calculate the fractional counts of our parameters:

$$\overline{count}(x|b) = 1/2$$
$$\overline{count}(y|b) = 3/2$$
$$\overline{count}(x|c) = 1/2$$
$$\overline{count}(y|c) = 1/2$$

We got these values by counting the number of times each parameter showed up in each alignment multiplied by the respective alignment probability. For example, word x is only generated by word b in alignment A2, and A2 has a probability of $1/2$, so $\overline{count}(x|b) = 1/2$. Word y is generated by word b in alignment A1 and A3, which have probabilities of $1/2$ and 1 respectively, giving us $\overline{count}(y|b) = 3/2$.

Finally, we re-estimate our parameters based on the fractional counts:

$$p(x|b) = \frac{\overline{count}(x|b)}{\overline{count}(x|b) + \overline{count}(y|b)} = \frac{1/2}{(1/2) + (3/2)} = 1/4$$
$$p(y|b) = \frac{3/2}{(1/2) + (3/2)} = 3/4$$
$$p(x|c) = \frac{1/2}{(1/2) + (1/2)} = 1/2$$
$$p(y|c) = \frac{1/2}{(1/2) + (1/2)} = 1/2$$

Notice how $p(y|b)$ increased and $p(x|b)$ decreased. This is the model beginning to see that word y is likely generated from word b. The next step would be to plug these parameters back into our alignments and continue to re-estimate. Notice that even on the next iteration A1 becomes much more likely than A2, as expected. Eventually, the model will converge on a probability distribution where A1 is clearly the best alignment for the sentence in Corpus 1.

Decoding

Recall our noisy channel model equation from the beginning of lecture:

$$e^* = \underset{e}{argmax}\ p(f|e)p(e)$$

We now know how to estimate both the language model $p(e)$ and the translation model $p(f|e)$, so all that is left to do is take the argmax over all possible English sentences e to get the best translation e^*. So we are done right? Not quite. Knight (1999) actually showed that this final step of translation, which is often referred to *decoding*, is NP-complete

if we want to assure we find the globally optimum decoding. So, most decoding techniques today do not always find the global optimum, but can get close and within a reasonable amount of time. We will examine a greedy approach to decoding today.

First Stage of the Greedy Method

We start out very simply. For each French word f_j, we pick the English word e that maximizes $T(e|f_j)$, where $T(e|f)$ is an inverse translation table that is given. This gives us an initial alignment.

Next Stage: Greedy Search

The first stage gave us an initial (e^0, a^0) pair. To work towards a better translation, we define a set of transformations that map an (e, a) pair to a new (e', a') pair. More formally, let us say that $\prod(e, a)$ is the set of all (e, a) reachable from (e, a) by some transformation. Then, at each iteration we take:

$$(e^t, a^t) = argmax_{(e,a)\in\prod(e^{t-1},a^{t-1})} P(e)P(f, a|e)$$

In other words, we greedily take the current pair at time t to be the highest probability output from the results of all possible transformations from the pair at time $t-1$. We then iterate until this process converges.

Transformations

We now discuss the types of transformations between pairs that are possible.

1. $CHANGE(j, e)$ - changes translation of f_j from e_{a_j} into e. If e_{a_j} is aligned with exactly one word, then we can do a simple replace. If it is aligned with more than one word or is aligned with *null*, then we place e at a position in the string that maximizes the alignment probability. It is worth noting that we typically only consider (e, f) pairs such that e is in the top 10 ranked translations for f under $T(e, f)$.
2. $CHANGE2(j1, e1, j2, e2)$ - changes translation of f_j1 from e_{a_j1} into $e1$ and f_j2 from e_{a_j2} into $e2$. This is the same as performing $CHANGE(j1, e1)$ and $CHANGE(j2, e2)$ in sequence.
3. $TranslateAndInsert(j, e1, e2)$ - calls $CHANGE(j, e1)$ and then inserts $e2$ at the most likely point in the string.
4. $RemoveFertilityZero(i)$ - removes e_i provided that e_i is aligned to nothing in the alignment.

5. $SwapSegments(i1, i2, j1, j2)$ - swaps words $e_{i1} \dots e_{i2}$ with words e_{j1} and e_{j2}, provided that the two segments do not overlap.
6. $JoinWords(i1, i2)$ - deletes English words at e_{i1} and then takes all French words that were previously linked to e_{i1} and links them to e_{j2}.

Example

The following example is taken from Germann et. al 2001. Say we are given the following sentence in French:

Bien intendu , il parle de une belle victoire

and we want to translate it into English. Following the first step of our greedy approach, which just selects the highest probability translations for each words, yields us:

Well heard , it talking NULL a beautiful victory

We could then call $CHANGE(5, talks, 8, great)$, which will now yield:

Well heard , it **talks** NULL a **great** victory

then call $CHANGE(2, understood, 6, about)$:

Well **understood** , it talks **about** a great victory

then call $CHANGE(4, he)$:

Well understood , **he** talks about a great victory

then call $CHANGE(1, quite, 2, naturally)$:

quite naturally , he talks about a great victory

We are left with the correct translation: “Quite naturally, he talks about a great victory." At each timestep, we just greedily choose the best transformation. Note that although the number of possible transformations can be pretty large, it is still manageable because of certain restrictions we put in place (such as the fact that a $CHANGE$ only looks at the top 10 ranked translations). Therefore, the greedy approach can be run in a reasonable amount of time and yield good results. Click here to learn more about the details of this greedy decoding.

Phrase-Based Translation

Thus far, we have only looked at word-to-word translations. Specifically, our alignments only allow each French word to point to a single

English word (or *null*). However, as we stated earlier in this lecture, it is also possible to do phrase-based translation, where we align phrases in English with phrases in French. The idea here is that phrases, like words, are still common, reoccurring entities in a given corpus, but they can also capture more information than a single word, so translating at the phrase level might be useful.

Alignment Matrices

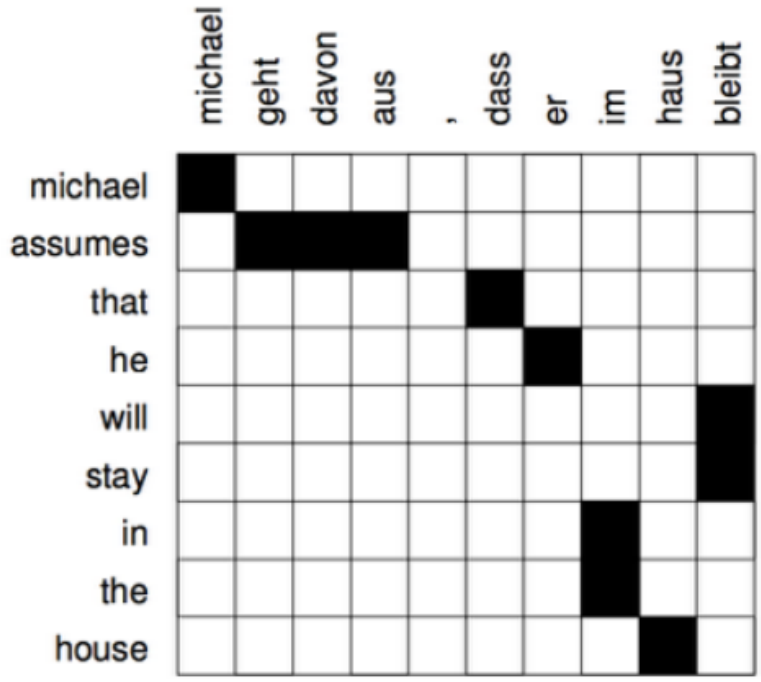


Figure 2: An alignment matrix between English and German

In the above alignment matrix, each row is an English word and each column is a German word. A filled in cell means that the words are aligned. Note, however, that it is not a one-to-one mapping. For example, the English word *assumes* maps to the German words *geht davon aus*. The nice thing about the alignment matrix is that it allows us to easily find *phrases* and their translations. For example, we can take the English phrase *in the house*, look at the alignment matrix to see what cells are filled in, and conclude that the German translation is *im haus*.

Finding Alignment Matrices

To find an alignment matrix like the one above, we first train IBM Model 4 for $p(f|e)$ and come up with the most likely alignment for

each (e, f) pair. We then do the same for $p(e|f)$ and come up with the most likely alignment for each (e, f) pair. Now that we have two alignments, we take their intersection as our starting point, as shown below.

Alignment from $P(f \mid e)$ model:

	Maria	no	daba	una	bof'	a	la	bruja	verde
Mary	●								
did						●			
not		●							
slap			●	●	●				
the							●		
green									●
witch								●	

Alignment from $P(e \mid f)$ model:

	Maria	no	daba	una	bof'	a	la	bruja	verde
Mary	●								
did		●							
not		●							
slap					●				
the							●		
green									●
witch								●	

Intersection of the two alignments:

	Maria	no	daba	una	bof'	a	la	bruja	verde
Mary	●								
did									
not		●							
slap					●				
the							●		
green									●
witch								●	

The Model

Below are the steps of the phrase-based model, which again is a probability model that models $p(f|e)$.

1. Choose a segmentation of e (all segmentations are equally likely).
2. For each English phrase e, choose a French phrase f with probability $T(f|e)$.

3. Choose positions for the French phrases. If the start position of the i'th French phrase is a_i and the endpoint of the $(i-1)$'th French phrase is b_{i-1}, then this position has a probability given by $R(a_i - b_{i-1})$.

Training the Model

Training the model is quite straightforward once we have the alignment matrices. In fact, we can just take the maximum-likelihood estimates:

$$\phi(f|e) = \frac{count(e,f)}{\sum_{f_i} count(e,f_i}$$

The Decoding Method

The goal is to find a high probability English string e under

$$p(e)p(f,a|e)$$

where

$$p(f,a|e) = \prod_{i=1}^{n} T(f_i|e_i)R(a_i - b_{i-1})$$

where i is iterating through the n *phrases* in the alignment and a_i and b_i are the respective start and end points of the i'th phrase.

Stack-Based Decoding

We define a *partial hypothesis* to be an English prefix aligned with some of the French sentence. S_m is a *stack* which stores the n most likely partial hypotheses that account for m French words. At each point in our algorithm, we simply pick a partial hypothesis and advance it by choosing a substring of the French sentence. To find out more details about this process, check out this link.