COMP 550 Study guide

Francis Piché

October 9, 2019

Contents

Part I
Preliminaries

1 License Information

These notes are curated from Professor Jackie CK Cheung lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

Part II
Introduction

2 Computational Linguistics and NLP

Language is a form of communication with arbitrary pairings between form and meaning.(Form as in symbols / sounds used).

Computational Linguistics is the modeling of natural language with computational techniques.

This includes natural language understanding (comprehension) and natural language generation (production).

The goal of CL is to gain a scientific understanding of how language works.

NLP (Natural language processing) is sometimes interchanged with CL but has an emphasis on engineering and applications. Ie: solving some real world problem using our understanding of language, as opposed to gaining more understanding of language in a scientific manner.

3 Domains of Language

- Phonetics: Speech and sounds that make up a language. How words are articulated, generated with mouths etc.
- Phonology: Rules that govern sound patterns (how sounds come together in words).

- Morphology: Word formation and meaning. (Structure of individual words, anti-dis-establish-ment-arian-ism)
- Syntax: Structure of language (sentences, arrangement of words)
- Semantics: The study of meaning in a language. Ambiguities can come in the sense of a word like "bank" which has many meanings.
- Pragmatics: The study of language in context. Literal meaning can change based on context. Interpretations depend on extralinguistic context. (who does "I" refer to)
- Discourse: Structure of large spans of language (spanning several sentences/phrases)

Part III
Classification

4 Machine Learning Basics and Terminology

4.1 Formulating a problem to use ML

Often a situation arises where we have some NLP problem we want to solve. For example, determining which emails are spam. If we can formulate the NLP problem as a machine learning problem, then we can solve the problem by using machine learning. Often this means we need to formalize the problem to be able to apply mathematics and statistics to the problem to solve it.

For our problem, we can formulate the ML problem like this:

$$email \rightarrow classifier \rightarrow true/false$$

That is, we need a *classifier* to tell us whether or not this email is spam.

Note that we don't always need to use ML for this. We could hand-code a set of rules to say "when you see x word, then it is spam" etc. The ML approach is best used when these rules are too complex or too numerous to quantify, and so we must "teach" the model to recognize spam by examples.

4.2 Supervised vs Unsupervised ML

The distinction comes from how much information we give the model.

4.2.1 Supervised Learning

In supervised learning, the model has access to the input data and all corresponding output data. In other words, determine (learn) a function f, such that $y = f(x)$ given a set of examples (x, y). In this example, y is the **output**, f is the **model** and x is the **input**.

The first step is determining what the input *is*. So the inputs (words, pictures etc) need to be mapped so some numerical format so the machine can understand. Afterwards, we must determine the parameters θ to use in the function f that minimize a loss or error function.

In supervised learning, we are mapping $y = f(x)$. If y is a continuous outcome, then we have a **regression** problem. And if y is discrete, then we have a **classification** problem.

Learning in the context of supervised learning is determining f, or a set of parameters θ for a model that minimizes some error function.

4.2.2 Linear Regression

In a regression problem, our function will look like this:

$$y = a_1x_1 + a_2x_2 + ... + a_nx_n + b$$

Where $\theta = \{a_1, a_2, ..., a_n, b\}$. And th x_i are our inputs.

4.2.3 Classification

In this class will mostly focus on classifications. This is because often we have discrete sets of outcomes like words, parts-of-speech tags, sematic categories, or discourse relations.

4.2.4 Unsupervised Learning

In **unsupervised** learning, the model only has access to the input data. The goal is then to find some interesting patterns in the data.

Often this ends up being a "clustering" problem. For example *grammar induction*: "the" and "a" often show up in the same context. So maybe they fit in the same grammar category (often appear before nouns) so based on the environments where the words appear we can infer where those words should be used.

Similarly, the words "very" and "hope" almost never appear in similar contexts so we can infer that we shouldn't put them together.

Learning in unsupervised context is coming up with some characterization of the data. If the model is not probabilistic this means defining some algorithm that clusters the data in a reasonable way. In a probabilistic model this means finding the parameters θ to the model to maximize the probability that the training corpus was used given this set of parameters.

4.2.5 Semisupervised Learning

For some of the inputs we have access to the outputs, but not all of them. So we can come up with an algorithm that uses what we know (supervised) to determine what we don't know (unsupervised).

5 Feature Extraction

Document classification is trying to find some property of an entire document. Ie: What is the genre (text-message, novel, news,...), what is the topic, is it spam, is the tone positive/negative etc.

Steps:

1. Define problem and collect data
2. Extract useful things about the documents that could be used to identifying the class.
3. Train a classifier on the data set
4. Train more classifiers to figure out which one is the best
5. Use the final model

5.1 Feature Extraction

To create our inputs $\vec{x}$ to the function $f(\vec{x})$ we need to represent the document as a list of features $\{x_1, x_2, ..., x_n\}$.

5.1.1 Commonly Used Features

We can remove inflectional morphology to recover the **lemma** of a word (the thing we'd look up in a dictionary)

- faxes → fax
- flies → fly
- geese → goose

or we can use **stemming** to truncate affixes to a word to find the stem:

- Airliner → airlin
- Ponies → poni
-

However you might not always want to do that. It really depends on the problem.

By reducing the number of words we use in the training corpus like this, the matrix gets smaller and so we need less data and less computation time.

6 Model Selection

As mentioned in the document classification steps, we need to try multiple different models and "decide" on the best one (or combinations of models). For each choice we have in the feature extraction step, we train a model with that decision.

6.1 Overfitting

Overfitting is a problem that arises that occurs when reusing a fixed test-set of data on many models. We want to have a model that works well on general data, not only the training set. Overfitting is when a model adjusts strongly to a property of the dataset that doesn't work in general.

The solution is to have multiple data sets. One is **training data**, generally 60-90% of available data, and the other is the **testing data**, 10-20% of available data which is used to validate the model. Aften a third set, the **development** or **validation** set which is used when deciding between different versions of a model.

So the overall strategy is:

1. Train models on training set
2. Test (repeatedly) on the dev set to choose settings
3. Test the final model on final testing set once

6.2 Cross Validation

To further avoid over-fitting, we can do **k-fold cross validation** where we split the training data into k partitions (folds) and then iteratively test on each one after training the rest. This is expensive to do so its usually only done on small datasets.

7 Linear Classifiers

In this section we discuss different forms of the function f which we are trying to train. That is, how do we construct a classifier to train?

Once we select an architecture for f such as Naive Bayes, f can be described by its parameters θ. Or in other words: $y = f(\vec{x}; \theta)$. Training the model then is just finding parameters $\theta*$ according to some objective function (max, min some property).

7.1 Naive Bayes

Naive Bayes is a probabilistic classifier coming from Bayes rule:

$$P(y|x) = \frac{P(y)P(x|y)}{P(x)}$$

The reason it is called Naive is because of its assumptions about how the dataset is generated:

- It assigns each label a probability $P(y)$ (the probability a spammer decides to write a spam email). This is known as the prior distribution.
- The feature vector $\vec{x}$ is generated by generating each feature independently, conditioned on y: $P(x_i|y)$. (ie: given the spammer is writing spam, what is the probability that they use the word "money" in their email). This assumption is why its called naive. Of course, since word occurrences are dependent this assumption is false. But it generally is pretty good anyway.

The learning problem is then to figure out based on these assumptions, what the parameters should be. That is, what are the parameters of the prior distribution $P(y)$, and what are the parameters for each features distribution conditioned on the class, $P(x_i|y)$. Assume these are categorical (discrete) distributions.

But remember we need to pick the parameters with respect to some objective function. In this case we maximize the likelihood of the training corpus. Skipping the details, it turns out that the best settings for the parameters are $P(Y = y) = p$ where y/n where y is the number of samples of class y , and n is the total number of samples. It also turns out that $P(X_i = x|Y = y)$ the proportion of samples with x_i appearing in all samples of class y.

Once we've done the training, classifying new instances is as easy as plugging into the formula:

$$P(y|\vec{x}) = \frac{P(y)\Pi_i P(x_i|y)}{P(\vec{x})}$$

The numerator we have since we trained the model, and the denominator can be found by summing up the probability of all $\vec{x}$ given all possible y.

Note that Naive Bayes is simple enough that "training" is really just plugging some numbers into a function. There is no iteration involved.

Another important point that was brought up in class is why the $P(x_i|y_j)$ need not sum to 1 for different j but same i. This is because then the probabilities are coming from different distributions. For example, suppose we have a 6-sided die and a 10-sided die. Clearly, the probability of rolling a 1 given the 6-sided and the probability of rolling a 1 given 10-sided need not sum to 1. (1/6 + 1/10).

7.2 Generative vs Discriminative Models

Generative models learn a distribution for ALL possible random varaibles, $P(\vec{x}, y)$ (the joint distribution).

Discriminative models learn only $P(y|\vec{x})$. This means we don't really care about the documents themselves (and how they're generated). However, this model can only do classification.

7.3 Logistic Regression

Despite the name, this model is for classification (not linear regression)! The function used is the logit function:

$$P(y|\vec{x}) = \frac{1}{Z} e^{\sum_i (a_i x_i) + b}$$

This will output values between 0 and 1. The x_is are given by the feature vector, and the a_is are parameters of the model. So to train, we must determine the a_is which will maximize probability of getting the "correct" answer. This means we can choose anything for the features, since the a terms are what will determine the "importance" of each feature, and the features themselves have no fixed meaning in the model.

Here, gradient descent in an iterative process to find the "optimal" values of a.

7.4 Support Vector Machines

Without going into too much detail, this model is based on a higher dimensional geometric classification of data. Each feature forms an axis in our space. Each sample forms a point in this space. Then we solve an optimization problem to form a hyperplane as a "decision boundary" in such a way that maximizes the distance to the nearest sample in each class.

This is a discriminative model since it doesn't try to characterize the entire dataset. It only cares about deciding the boundary.

These don't always have to be linear, since we can apply kernels which change the decision boundary to be non-linear in a lower dimensional space, but still stay linear in a higher dimensional space.

8 Non-Linear Classifiers

The limitation of linear classifiers is that they cannot learn complex relationships between features while keeping a linear function. For example, it could not figure out that "starts with capital letter" and "not at beginning of sentence" means its a proper noun. Because then we would need to do the product of some feature values which is no longer a linear combination on the features (no longer linear).

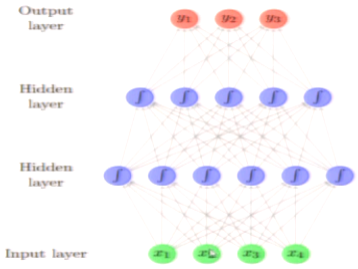
Of course you could cheat by pre-determining this on your input features before passing to the linear classifier, but this would require far to much work and requires a lot of prior knowledge about the features / their relationships.

8.1 Artificial Neural Networks

A neural network, inspired by biology is a network of computational units (neurons) where each neuron is a function on the input vector : $g(a_1x_1 + ... + a_nx_n + b)$. Overall the network can be proved to learn any computable function given enough neurons. The entire network is trained simultaneously.

8.1.1 Feed Forward Neural Networks

In Feed Forward NNs, all data flows forward and each node in a layer is connected to all nodes of subsequent layers.



The input layer is the feature vector. Each hidden layer is a step in the computation, and the output layer is the probabilities of each class.

Of course we could skip layers, remove connections etc, but in this common model we don't, and the decision to do so is based on experience, intuition and "what works".

Inference in a FFNN, is computing the output based on the input. Recall that each neuron

computes some function $g(\vec{a} \cdot \vec{x} + b)$, and so if each column of the matrix W is the weights for one neuron and $\vec{b}$ represents all the bias terms for the neurons, then the matrix multiplication $\vec{x}W + \vec{b}$ computes one layer in the network. Suppose we have a 2 layer network, then:

$$h^1 = g^1(\vec{x}W^1 + b^1)$$

$$h^2 = g^2(\vec{x}W^2 + b^2)$$

$$y = h^2 W^3$$

The function g introduces the non-linearity. Often this can be a Sigmoid function (squash to 0-1 range), $tanh$, $max(0, x)$ etc. The "feeding" of one layer into the next represents function composition, and so if g was linear then there's no point having layers since the whole thing could have just been one linear function. If we know the problem is a linear one, then we can just pass to a linear classifier, but often we don't know if the problem is linear which is where NN comes in handy.

However all of this leads to continuous values for output. In NLP we often care about discrete outcomes, so the final layer is generally a $softmax$ layer defined by:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j^k e^{x_j}}$$

the goal is to have all the output values sum to 1. This also has the effect of having one value close to 1 and all others close to 0. This layer is essentially performing logistic regression.

8.1.2 Loss Function and Training

For neural networks, we pick parameters in order to minimize a loss function. The most common/simple one for classification is to define y to be the correct answer for the class labels, and $\hat{y}$ to be the predicted answers for class labels by the network. Then we set the loss function $L(y, \hat{y})$ to be the cross-entropy:

$$L(y, \hat{y}) = -\sum_i y_i log(\hat{y_i})$$

This basically tells us "how different" the two vectors are. We then want to minimize this value by changing the parameters in our network.

Typically training in NNs is done by **gradient descent**, or in other words, find the gradient of the loss function with respect to the parameters, and change the parameters in such a way that matches the direction and magnitude of the gradient vector. **Backpropagation** is an efficient algorithm that uses the chain rule to send the error given by the loss function backwards through the network backwards to the inputs.

So to train, we first choose the initial parameters (maybe randomly). Then compute the gradient of the loss function, and backpropagate back to adjust the parameters, rinse and repeat until some convergence point is achieved (gradient reaches local minima).

Generally this is done for the whole training corpus at once (one iteration of the algorithm = summing loss over all samples, updating weights once). But this is very innefficient, so **Stochastic Gradient Descent** is used. This is where a small mini-corpus is used to update the weights, until convergence to the same final solution.

8.2 Word Representations

Rather than what we've done so far which is to associate word occurrences in the feature vector, its common to associate each word type with a fixed dimensional vector, which are model parameters. So the words themselves are vectors. Then if we have two similar words (money, cash) then their vectors might be similar and things we learn about one can be learned for the other!

To represent **sentences**, often the sentence is represented as some function of its word vectors such as addition or component-wise multiplication. Of course these are naive since they ignore word order, but they work for simple applications.

8.3 Advantages/Disadvantages of NN

Advantages:

- Complex relationships between inputs and outputs can be learned
- Very flexible and generic
- Efficient use of input data since can share weights across features

Disadvantages:

- Complex models and a lot of data needed
- Many choices to tune with little knowledge about the "right" way to do it
- Can be difficult to interpret outputs

9 Interpreting Results

One simple way is to calculate the accuracy of the model (number correct over number of samples in the test set):

$$correct/total$$

but this can be bad if the classes are balanced in number.

Precision is how many were correct for how many predictions were made:

$$correct/predicted$$

and **recall** is the number of correct answers for a specific class:

$$correct/numinclass$$

ie: how many were correctly identified as spam vs how many spam were in the test set. The **F1** score combines precision and recall:

$$F1 = \frac{2 * P * R}{P + R}$$

This is useful because of the multiplication in the numerator: if one of precision or recall are very bad, then the F1 score will be very low. So both must be balanced and good for the score to be good.

Macro Average assumes each class is equally important, so this is the average of the precision (or recall) for each class. **Micro Average** here we say each sample is equally important, so first take sum of the counts, then compute precision, recall and F1 afterward.

9.1 Confusion Matrix

A confusion matrix shows visually how the model predicted classes and compares to the actual classes

Predicted class

Actual class	C1	C2	C3	C4
C1	count	count	count	count
C2	count	count	count	count
C3	count	count	count	count
C4	count	count	count	count

The correct entries will be on the diagonal and the off-diagonal are the incorrect entries.

Part IV
Language Models

10 Word Sequences (N-Grams)

First, we must define what a word is. The basic definition is the smallest unit that can occur in isolation. However that may not always be true. Is football one word or two? Each of foot and ball can be words, but so can football. Another example: is peanut butter one word or two? This is another showcase of how the representation of the language doesn't always match with the interpretation in our minds.

We must also distinguish **tokens** vs **types**. Tokens are instances of a type. "Cat cat cat" has only one type but 3 tokens.

10.1 Word Frequencies

The Term Frequency is the number of times a word type appears in a corpus S. The Relative frequency is the number of times a word type appears in a corpus relative to the total number of tokens.

Note that corpus is defined as a body of text.

10.1.1 Zipfs Law

The frequency of a word type is roughly proportional to the inverse rank of the word type.

$$f \propto \frac{1}{r}$$

Rank	Word	Frequency
1	*the*	228,257,001
2	*to*	96,247,620
3	*of*	93,917,643
10	*for*	34,180,099
100	*most*	3,499,587
1,000	*work*	1,999,899
10,000	*planning*	299,996

Word counts from the English Gigaword corpus

A closer fit to the word count observed is the **Zipf-Mandelbrot Law**:

$$f = \frac{P}{(r+\rho)^B}$$

$$log(f) = log(P) - Blog(r+\rho)$$

however this does not hold true for all languages. Some languages have much more morphological richness, (more specified words and less use of things like "a, of, but, to, and...").

This means that most words are very rare, which causes problems trying to perform statistical analysis since this requires a lot of data.

Language modeling is the process of predicting the next word given some context. Or in math: $P(W = w|C)$ (the probability of the next word being w, given C the context). This actually has an equivalence with the probability that a given sequence of words will appear overall.

$$P(w_1 w_2 ... w_k) = P(w_k|w_1 ... w_{k-1}) ... P(w_2|w_1)P(w_1)$$

So a good language model should assign low probability to un-grammatical or nonsensical strings of words, but that doesn't mean that it can be used as a score of grammar. For example: "artichokes intimidate zippers" is grammatically correct, but most language models will show low score since those words rarely appear together. Also, the length of the sentence affects the probability: "I ate the" is more likely than "I ate the cake", although the first is not a well formed sentence.

Language models capture some linguistic knowledge such as which words tend to appear together etc. But also facts about the world, since its based on "popularity" of sentences. It also demonstrates syntax since order of words will matter.

11 Building Language Models

Given lots of data we want to build a model (set of parameters) that describes the data, and can be used to predict or infer unseen data.

1. Gather lots of data
2. Learn parameters from the corpus to build the model
3. Use the model to evaluate testing data.

How do we train the model? First we need to specify what the context of a word is, and use the corpus to derive the parameter values.

The context is the previous $N-1$ words. The last word in an $N-gram$ will be the random variable. Usually bigram or trigram language models are used. This reduces greatly the size of the training problem to make it computationally feasible.

The simplest method of getting the parameters from counts is to divide the frequency of the N-gram by the total count. For example:

- P(cats) = Count(cats) / total words
- P(cats|the) = Count(the cats) / Count(the)
- P(cats|feed the) = Count(feed the cats) / Count(feed the)

This is known the MLE method since it uses the maximum likelihood estimators for the parameters.

12 MLE Example

Given: "that that is is that that is not is not is that it it is", find the MLE estimate of a unigram and bigram language model.

First for the unigram, note that they all share the same context (no context) and so they all are under the same probability distribution. There are 4 words, so there are 4 parameters, however since they are all under the same distribution they must sum to one and so there are 3 free parameters (the last can be determined from the others).

$$P(that) = 5/15$$

where 5 is the number of times "that" appears overall. The other words are found similarly.

For the bigram, there are 4 different contexts (4 choices for the previous word) and so there are 4 distributions. So, for each distribution, we have 4 parameters (choices for the next word) of which only 3 can actually vary since the 4th must sum to 1. So in total there are 4*3 = 12 free parameters. Also note that we must now account for the beginning and end of sentence. One way is to add special tokens for the START and END, (in which case there are now two new distributions and parameters for each). For the end, something else you can do is account for the fact that its possible nothing comes after *is* in our sentence.

$$P(that|that) = 2/5$$

since "that that" shows up twice and "that" shows up 5 times.

$$P(that|is) = 2/5$$

since "that is" shows up twice and "that" shows up 5 times. And so on for the others...

13 Evaluating Language Models

Now its no longer sufficient to compare our result to a "known" value, since we are generating new texts. So we can evaluate by maximizing the likelihood of the test corpus: $P(T; \theta)$ where T is the test corpus, and θ is the parameters learned by our model. The actual number itself is not very meaningful, it can only be used to compare with other language models. Since the value is actually very small, usually its not used.

13.1 Cross-Entropy and Perplexity

Suppose we have a random variable X. The amount of information we can gain from knowing a value of X is different depending on the distribution of X. For example, if a coin always comes up heads, we gain no new information from knowing a particular coinflip was heads, since we could have known it from the distribution. On the other hand if it comes up heads

75% of the time, or 50%, we gain the most information from the 50% coin, (25% more than the 75% coin).

So observing an outcome that is likely to occur gives little information. Similarly observing a rare outcome gives the most information.

Information is defined as:

$$I(x) = log_2(\frac{1}{P(x)})$$

this is the minimum number of bits needed to communicate some outcome x.

Entropy then is the expected amount of information gained from observing a random variable:

$$H(p) = \sum_{i=1}^{k} p_i I(x_i)$$

$$= \sum_{i=1}^{k} p_i log(1/p_i)$$

$$= -\sum p_i log(p_i)$$

Cross entropy is used when the distribution of the message we're trying to communicate is unknown. The language is drawn from some true distribution, and the generated text from our language model is some approximation of it.

$$H(p, q) = -\sum p_i log_2(q_i)$$

where p is the true distribution and q is the model distribution.

The problem with this is that we don't have access to the "true" distribution, but the test corpus is assumed to be drawn from the true distribution.

We can estimate the cross entropy by:

$$H(p, q) = -\frac{1}{N} log_2(q(w_1...w_N))$$

where p is the true distribution (which we don't have access to), N is the size of the test corpus (in tokens) and w_i are the words in the test corpus.

The **Perplexity** is simply $2^H(p, q)$ since the cross entropy difference is generally quite small. The lower the perplexity the better the model is.

13.2 What is MLE?

MLE (maximum likelihood estimator) for a set of model parameters θ is a function which estimates the values for θ such that it maximizes the *likelihood* (probability of generating) the training corpus. In language models, words can be seen as random variables that are drawn from a categorical distribution, and that for a given context, these R.V's are independent and identically distributed.

Because of this i.i.d assumption, we can express the probability of a training corpus $C = \{x_1, ..., x_n\}$ to be $\Pi_i P(x_i; \theta)$

Our objective is to maximize this function. First, since log is monotonically increasing, its equivalent to find the log-likelihood (makes the math work out nicer):

$$log(\Pi_i P(x_i; \theta)) = \sum_i log(P(x_i; \theta))$$

Then we can take the derivative and set it to 0 to find a local maximum. Solving for θ. For example, if we're dealing with a Bernouli distribution, then $P(C; \theta)$ is:

$$\theta^{N_1}(1-\theta)^{N_0}$$

so we can solve for θ as follows:

$$\frac{d}{d\theta} log(\theta^{N_1}(1-\theta)^{N_0}) = 0$$

$$\frac{d}{d\theta} N_1 log(\theta) + N_0 \log(1-\theta) = 0$$

$$\frac{N_1}{\theta} - \frac{N_0}{1-\theta} = 0$$

$$\theta = \frac{N_1}{N_0 + N_1}$$

Which is actually the relative frequency!

In cases when we have more than 2 parameters,the same thing works, but we'll have to use LaGrange multipliers to solve the gradient to find the maximum.

14 Smoothing

MLE often is prone to overfitting. That is, the model will be very well fit to the training corpus, but this training corpus may not actually be representative of the true distribution. We want the model to be well fit not not only the training data, but to unseen data as well.

For example, maybe a topic wasn't present in the training data, but is present in the test data, then the model will not work well with this new topic.

New words (neologisms), foreign languages mixed in, and typos are all assigned a probability of 0 when using the MLE. This is known as OOV (out of vocabulary) items.

14.1 Dealing with OOV Items

One way to deal with OOV items is to remove all vocabulary items which appear less than a certain threshhold and replace them with an UNKNOWN token. During training, we give UNKNOWN tokens to any OOV items.

14.2 MAP Estimation

Smoothing means we are no longer using MLE. It uses some prior knowledge or belief about what parameters should look like.

In math terms, we are finding θ^{MAP} such that:

$$P(X;\theta^{MAP})P(\theta^{MAP})$$

is maximized. Basically, we're assigning a probability to a particular set of parameters.

The new distribution we're using is generally going to give some probability (non-zero) to unseen data.

14.3 Add-δ Smoothing

In this kind of smoothing, we modify estimates by "pretending" we saw each word δ times in the training corpus when we really didn't.

$$P(w) = \frac{Count(w) + \delta}{|Lexicon| * \delta + |Corpus|}$$

The extra factor of δ in the denominator is to ensure we still keep everything normalized between 0 and 1 even though we are adding this artificial δ factor. When $\delta = 1$ this is known as the Laplace estimate.

The downside is that this is not the best approach, and the choice of δ is arbitrary.

14.4 Iterpolation

In N-gram model, as N increases, the problem of sparcity is increasingly prevalent. This is because the number of possible N-gram combinations increases exponentially, and thus the occurrences of such N-grams decreases proportionally. So the occurences of the rare sequences of words are extremely rare. In **interpolation**, use a lower N and interpolate between the models.

So we can obtain an interpolated probability of a trigram ($\hat{P}(w_t|w_{t-2}, w_{t-1})$) by summing the MLE of the trigram, bigram and unigram, multiplying by some weights to say how much of each term we want to keep. So:

$$\hat{P}(w_t|w_{t-2}, w_{t-1}) = \lambda_1 P^{MLE}(w_t|w_{t-2}, w_{t-1}) + \lambda_2 P^{MLE}(w_t|w_{t-1}) + \lambda_3 P^{MLE}(w_t)$$

The λs need to be chosen in such a way that they sum to 1 so that the overall result is a probability distribution. Again the choice of λs is difficult, and generally done by trail and error, tuning on a development set.

14.5 Good-Turing Smoothing

This is another (better) way of modeling unseen data based on Zipfs law. In previous attempts, we've adjusted all words uniformly. But from Zipfs law we know that this isn't how words are distributed (far from it!). Thus the unseen N-grams should behave like N-grams that occur *once* in the training corpus, and N-grams that occur a lot (but not seen in the training corpus) should behave like words seen a lot in the training corpus.

So the strategy is to build a histogram to count the counts of words. Ie: there are f_i number of words that occurred c times in the training corpus. So for example suppose there are 3993 words that show up once, 1292 words that show up twice etc.

The plan then, is to find a model to estimate c using f_{c+1}

First, N is the total number of event tokens (size of training corpus). Then since f_i is the number of words that occur i times:

$$N = \sum_i f_i * i$$

Let $P(UNKNOWN) = \frac{f_1}{N}$ (everything that appears once has same weight as unknown events). But to keep this a probability distribution, we must lower the probability all the other events by adjusting the counts proportionally to the rank of the word type. So $P(w_c) = \frac{c^*}{N}$, where $c^* = \frac{(c+1)f_{c+1}}{f_c}$

The original MLE is just c/N but now we use this adjusted c^*/N.

The problem with this is what to do when we reach high values of c, we get bad estimates since there aren't enough word types that occur very frequently. The **refinement**

to do this is to estimate f_c as a function of c. This is done by learning a linear regression: $log(f_c^{LR}) = alog(c) + b$ for finding the higher values of c.

Part V
Parts of Speech Tagging

15 English Parts of Speech

Some of the most common parts of speech in English are:

- Nouns: *restaurant, me, dinner*
- Verbs: *find, eat, is*
- Adjectives: *good, vegetarian*
- Prepositions: *in, of, up, above*
- Adverbs: *quickly, well, very*
- Determiners: *the, a, an*

A part of speech is a syntactic category that tells you some of the grammatical properties of the word. Not so much to do with the meaning of the word but more about where in the sentence the word might appear, and what other words it might be paired with.

Some other important (but lesser known) parts of speech:

- Modals + Auxiliary verbs: *The police **can** and **will** catch the fugitives.* These are like verbs, but the conjugation is a bit different than usual. These are often used to form questions.
- Conjunctions: *and, or, but.* These connect and relate elements
- Particles: *Look **up**, turn **down**.* These can either be parts of particle verbs, or have other functions depending on what is considered a particle.

Open classes are parts of speech for which new words can always be added to the language (neologisms). Nouns, verbs, adjectives, adverbs and interjections fall into this category.

Closed classes are parts of speech for which new words tend not to be added such as pronouns, determiners, quantifiers, conjunctions, modals + auxiliaries, prepositions. These generally convey grammatical information.

When creating a tag set, we must decide how fine-grained the tags will be. For example we may wish to distinguish singular vs plural nouns, intransitive verbs from transitive ones etc.

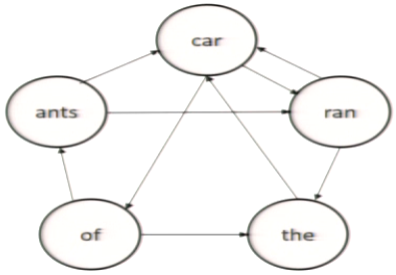
16 POS Tagging Models

If we have a tagset and a corpus labeled with these tags, we have a supervised learning problem and we're classifying words by parts of speech. The key though is that we have a sequence of decisions on classifying where each decision affects the others.

Specifically, this is a **sequence labeling** problem where we must predict labels for a sequence of inputs. We must consider both the current word and the previous context (and possibly future words) when labeling a word.

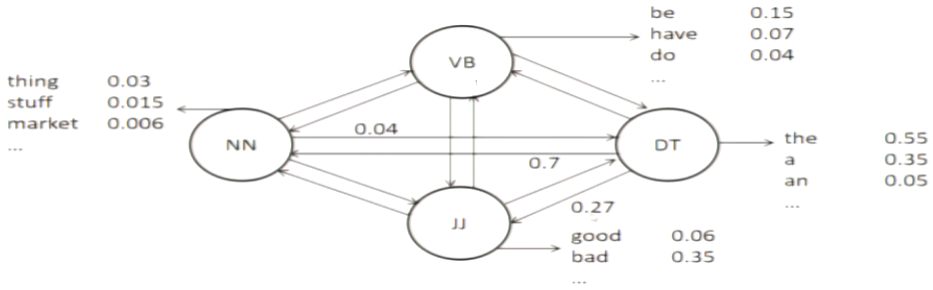
16.1 Markov Chains

This model assumes an underlying Markov process generating POS tags and words. The idea is that theres a finite set of states in some machine, and every time we take a step to a different state we generate a word and part of speech tag for that word.



We have N states that represent the words in the vocabulary. Transitions between states are weighted and the sum of outgoing edges sum to 1. Every step generates a word. This particular example above represents a bigram language model. This kind of model is called **observable** because every state of the model is observable at any time, and a walk along the states generate the text observed.

Parts of speech, since they are not seen in the actual text, are considered **hidden variables**. It's very common to have hidden underlying processes such as "climate conditions" leading to an observable output such as "weather".



In a hidden Markov process, the model transitions between POS tags and outputs words at each step. The words outputted also have their own probability distribution known as the "emission pdf".

We can now define the probability of a sequence as the product of the probability of each transition multiplied by the probability of the word emission. Of course all of this encodes an independence assumption.

16.1.1 Graphical Models

Each node in the graph is a random variable. Hidden R.Vs are generally clear in color, and observable R.Vs are generally shaded in color.

To formalize the math statement in the last subsection, the probability of a particular sequence of words is:

$$P(O, Q) = P(Q_1) \prod_{t=1}^{T-1} P(Q_{t+1}|Q_t) \prod_{t=1}^{T-1} P(O_t|Q_t)$$

where Q are POS and O are observed words. $P(Q_1)$ is the initial state probability.

Assuming there are N possible tags, W possible words, then the parameters θ have 3 parts

1. Initial probabilities for Q_1
2. Transition probabilities for Q_t to Q_{t+1}
3. Emission probabilities for Q_t to O_t

There are $N - 1$ free parameters in the distribution for the initial probability Q_1. There are N distributions for the transition probabilities (one for each context), each one having $N - 1$ free parameters. Again there are N distributions for emission probabilities (one for each possible O that we are conditioning on), each with $w - 1$ parameters.

In practice most of the parameters will have 0 probability or close to 0 since most words only belong to a few parts of speech.

16.1.2 Training A Hidden Markov Model POS Tagger

For the initial distribution, we can look at the POS tags in the first word of each sentence and derive relative frequencies. For the Transition probabilities we can look at transitions of POS tags that are seen in the training corpus. And for emission probability we can look at the emissions of words from each POS tag in the training corpus.

Using MLE:

$$\pi_i = P(Q_1 = i) = Count(Q_1 = i)/Count(sentences)$$
$$a_{ij} = P(Q_{t+1} = j|Q_T = i) = Count(i, j)/Count(i)$$
$$b_{ik} = P(O_t = k|Q_t = i) = Count(word(k), tag(i))/Count(i)$$

16.1.3 Inference with Hidden Markov Models

So now that we have a model, how can we tag a new sentence?

Of course a naive way could be to just assign the most probable tag for each word, but the issue is that we're not considering context. So we need a way to find the "best" POS tag sequence of the whole sentence.

So a few things we might be interested in figuring out:

- Likelihood of a sequence of observations: $P(O|\theta)$
- State sequence that best explains a sequence of observations: $argmax(P(Q, O|\theta))$
- Given an observations sequence, what is the best model for it?

Algorithms for determining answers to these questions are outlined in the next section.

16.2 Computing Likelihood (Forward/Backward Algorithms)

To directly compute the likelihood of a sequence of observations we would have to marginalize over all possible state sequences leading to those observations:

$$P(O|\theta) = \sum_Q P(O, Q|\theta)$$

The problem with this is that there is an exponential number of paths N^T.

16.2.1 Forward Algorithm

The **Forward Algorithm** is a dynamic programming algorithm to avoid redundant computations. We will do this by creating a table of all state sequences annotated with their probabilities.

States	O_1	O_2	O_3	O_4	O_5
VB	$\alpha_{VB}(1)$	$\alpha_{VB}(2)$	$\alpha_{VB}(3)$	$\alpha_{VB}(4)$	$\alpha_{VB}(5)$
NN	$\alpha_{NN}(1)$	$\alpha_{NN}(2)$	$\alpha_{NN}(3)$	$\alpha_{NN}(4)$	$\alpha_{NN}(5)$
DT	$\alpha_{DT}(1)$	$\alpha_{DT}(2)$	$\alpha_{DT}(3)$	$\alpha_{DT}(4)$	$\alpha_{DT}(5)$
JJ	$\alpha_{JJ}(1)$	$\alpha_{JJ}(2)$	$\alpha_{JJ}(3)$	$\alpha_{JJ}(4)$	$\alpha_{JJ}(5)$
CD	$\alpha_{CD}(1)$	$\alpha_{CD}(2)$	$\alpha_{CD}(3)$	$\alpha_{CD}(4)$	$\alpha_{CD}(5)$

Time

where $\alpha_i(t)$ is $P(O_{1:t}, Q_t = i|\theta)$ (ie the probability of the current tag and words up to now.)

The base case is the first column (ie probability of the first word). Each entry in this column $\alpha_j(1) = \pi_j b_j(O_i)$ (the probability of a tag π_j multiplied by $b_j(O_1)$ which is probability of observing the word.

Then the overall likelihood is the sum over all entries of the last column, since this column will have accounted for all possible sequences.

$$P(O|\theta) = \sum_{j=1}^{N} \alpha_j(T)$$

For the recurrence, we must consider all possible ways to get to the current state, and the probability of the emission at the current state. This can be expressed by:

$$\alpha_j(t) = \sum_{i=1}^{N} \alpha_i(t-1) a_{ij} b_j(O_t)$$

in other words, summing over all possible ways to get to that entry from the previous column, and the probability of that entry.

The runtime of the overall algorithm is $O(N^2 T)$

16.2.2 Backward Algorithm

Similar to the Forward, but in reverse direction. Each cell $\beta_i(t) = P(O_{t+1:T}|Q_t = i, \theta)$ ie: the probability of all the subsequent words given that the current tag is i (excludes the current word).

So we actually will start in the last column, and set $\beta_i(T) = 1$ for each cell in the last column. This is because we don't actually know what to put there, but since we will later multiply by β_i, 1 is used as the identity.

Then, each column to the left is filled by:

$$\beta_i(t) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_j(t+1)$$

And the final step to compute the probabilty of the output sequence:

$$P(O|\theta) = \sum_{i=1}^{N} \pi_i b_i(O_1) \beta_i(1)$$

(summing the first column). And the runtime is again $O(N^2T)$.

So overall, each $\alpha_i(t)$ is probability of all the observations up to and including the current time, and each $\beta_i(t)$ is the probability of all the observations after (not including) the current time. Thus multiplying them together gives the probability of the entire sequence of observations given that we are in state i at some time t. This means we can express the overall likelihood as:

$$P(O_t|\theta) = \sum_{i=1}^{N} \alpha_i(t) \beta_i(t)$$

16.3 Aside on Implementation

One thing to keep in mind is that we're multiplying a lot of very small floating point numbers, which is prone to numerical errors and underflow. To correct this, take the log of everything and work with sums instead of multiplications.

If you also need to perform sums, then can use this trick:

$$log(\sum p_i) = log(\sum e^{p_i})$$

$$= log(e^{max(a_i)} \sum e^{a_i - max(a_i)})$$

$$= max(a_i) + log(\sum e^{a_i - max(a_i)})$$

16.4 Sequence Labeling (Viterbi Algorithm)

To find the state sequence that best explains a set of observations, we need to find:

$$Q = argmax_Q(P(Q, O|\theta))$$

We will basically just use the forward algorithm, but take the max instead of summations.

So each cell of the grid is:

$$\delta_i(t) = max_{Q_{1:t-1}} P(Q_{1:t-1}, O_t, Q_t = i|\theta)$$

or in other words, out of all the sequences so far, which is the best, given that we are in time step t and state i currently.

Note that for the first column we have nothing to max over, so again its just $\pi_j b_j(O_1)$ for each cell in the first column.

At the end we can just take the max of the final column. The run time is again $O(N^2T)$. Note that we must also keep backpointers to the cell in the previous column which gave the max, so that we can build the sequence at the end by following the pointers.

16.5 Unsupervised Training (Hard/Soft EM/Viterbi EM)

Suppose we have no state sequences to compute the properties as before. We can guess the state sequences by initializing random parameters, and repeatedly predict the current state sequences using the current model, and update the parameters based on the current predictions.

The **Viterbi EM** algorithm for doing this is as follows:
Repeatedly:

1. Predict current state using current model with Viterbi algorithm
2. Update model parameters using current predictions as you would in a supervised learning setting.

But we can also use the *soft* predictions, the probabilities of the possible state sequences. As opposed to the above "hard" prediction.

16.5.1 Baum-Welch Algorithm

The first step is to find the Expectation (expected counts) for hidden structures using current θ^k. Then in the Maximization step, find θ^{k+1} to maximize the likelihood of the training data given the expected counts from the previous step.

So we have a distribution over the tags in this case. We'll call them *responsibilities*.

So first we must find all probabilities of being in state i at time t given the observations using the current model:

$$\gamma_i(t) = P(Q_t = i|O, \theta^k)$$

$$= \frac{P(Q_t = i, O|\theta^k)}{P(O|\theta^k)}$$

and now from the Forward and Backward algorithm:

$$= \frac{\alpha_i(t)\beta_i(t)}{P(O|\theta^k)}$$

Now we must find all of the probabilities of the transitions:

$$\xi_{ij}(t) = P(Q_t = i, Q_{t+1} = j|O, \theta^k)$$

$$= \frac{P(Q_t = i, Q_{t+1} = j, O|\theta^k)}{P(O|\theta^k)}$$

$$= \frac{\alpha_i(t)a_{ij}b_j(O_{t+1})\beta_j(t+1)}{P(O|\theta^k)}$$

Once all of these have been computed we can do the re-estimation of the parameters. The next initial probabilities are:

$$\pi_i^{k+1} = \gamma_i(1)$$

The next transition probabilities:

$$a_{ij}^{k+1} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

And the next emission probabilities:

$$b_i^{k+1}(w_k) = \frac{\sum_{t=1}^{T} \gamma_i(t)|_{O_t=w_k}}{\sum_{t=1}^{T} \gamma_i(t)}$$

If all that notation is confusing, remember the initial definitions of π, a and b, all we've done is replace the hard counts with the new soft distributions γ and ξ.

The algorithm stops when the overall likelihood stops improving on the training set, or when prediction performance on some development or validation set stops improving.

It can be shown that each iteration improves the likelihood, (Not covered in this course) however it may reach a local maxmimum, not a global maximum. To deal with the local optima problem, we can train multiple models with different random starting points, or we can bias the initialization using some external knowledge.

Note that this algorithm generally gives poor results. Its common to use **semi-supervised**learning to combine small amounts of labeled data with larger corpus of unlabeled data to get much better results.

16.6 Other Sequence Modeling Tasks

There are several other tasks that fit the sequence model quite well:

- Chunking: Find syntactic chunks in a sentence non-hierarchically. Ie: "peanut butter" as one chunk in the phrase "the peanut butter sandwich".
- Named-Entity Recognition: Identify elements in the text that match some high-level categories. Ie: Organizations, Persons, Locations

A first attempt to the second is to try labeling the words by their category. However this doesn't work for entities that span multiple words.

The solution to this is to also label whether the word is inside, outside or at the beginning of the span as well. Ie: "McGill University" McGill would be labeled as the beginning of an org, and University as inside an org.

Now the problem fits the HMM scheme that we defined before.

However HMMs don't fit well with adding more features. For example:

- Word position within the sentence
- Capitalization
- Word prefix and suffixes
- Features that depend on future words or outside context.

Its possible to handle this by adding multiple emissions for each timestep, each with their probability, but theres a better way.

Recall that HMM are generative models since they model the joint distribution between the observations and the hidden sequence. But if we only care about POS tagging, we can instead train a discriminative model which finds only:

$$P(Q|O;\theta)$$

Note that this cannot be used for generating new samples of word or POS sequences. It can only be used for tagging.

17 Linear-Chain Conditional Random Fields

This is a discriminative sequence model which operates similar to HMM.

$$P(Y|X) = \frac{1}{Z(X)} exp \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t)$$

Where Y is the tagging sequence, and X is the input words. Note the summation over time, and that this otherwise looks very similar to logistic regression. Also,

$$Z(X) = \sum_y exp \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t)$$

Also note that the features can depend on other states as well as the input word. θ is the weight for a given feature.

Recall that before we had to compute the probability of each transition, as well as the emission probabilities. Now, we replace the products by number that do NOT come from probability distributions, but instead are from linear combinations of weights and feature values.

17.1 Generating Features in CRFs

We can directly translate from transitions and emissions in the HMM case to feature in CRF:

- $f_{DT \to NN}(y_t, y_{t-1}, x_t) = I_{y_{t-1}=DT} I_{y_t=NN}$ for a transition from DT to NN.
- $f_{DT \to the}(y_t, y_{t-1}, x_t) = I_{y_t=DT} I_{x_t=the}$ for the emission of "the" from state DT
- $f_{DT}(y_1, x_1) = I_(y_1 = DT)$ for the initial state of DT.

We can also use this to propose new features! For example if we use $f_{cap}(y_t, y_{t-1}, x_t) = I_{y_t=?} I_{x_t cap}$ to mean this feature is 1 if we are at some tag and the word is capitalized. If after learning this feature has a high score, we know it is a good indicator for POS tagging, otherwise if it gets a low or negative score we know its not useful.

17.2 Inference with LC-CRFs

We can still do the dynamic programming algorithms from HMM and everything still works. So instead of looking up $P(X|\theta)$ we use $Z(X)$, and instead of finding $argmax(P(X, Y|\theta)$ for Viterbi, we find $argmax(P(Y, X|\theta))$

17.3 Training LC-CRFs

Unlike HMMs there is no analytic solution. We must use some iterative process to improve the likelihood. (Gradient descent). This is basically a variant of Newtons method for finding a place where the gradient is 0.

Luckily, the likelihood is concave, meaning we can reach the global maximum using this process.

Basically, we'll walk in the direction of the gradient to maximize $l(\theta)$.

$$\theta^{new} = \theta^{old} + \gamma \nabla l(\theta)$$

This can be interpreted as the gradient being the difference between the empirical distribution of a feature f_k in the training corpus, and the expected distribution of f_k as predicted by the current model. When the gradient reaches 0, this means there is no difference.

To avoid overfitting, **regularization** is used to lower the weights to be closer to 0. We do this by adding the term: $-\sum_k \frac{\theta_k^2}{2\sigma^2}$ to the log likelihood, where σ controls the amount of regularization.

As in neural networks, we can also do Stochastic Gradient descent and compute the gradient and update the weights on small batches of the training corpus to improve faster. (converge faster).

18 NNs For POS Tagging

We can use a feed-forward Neural Network for POS tagging as well as the methods discussed.

Instead of treating the sequence as a whole, we'll treat each word separately, but include a fixed context window in the NN to help predict the POS for each word. This is known as a **Time Delay Neural Network**

The input layer to the NN is structured as the concatenation of each word (represented as a vector) of the sequence. Then every vector from the input layer is connected to every node of the hidden layer. This is how the model takes into account the neighborhood of words. (even if it doesn't care about the sequential structure)

The network is applied to every word in the corpus.

However this type of model does not take into account previous decisions. It does not always generate a globally optimum sequence of predictions, however it does very well.

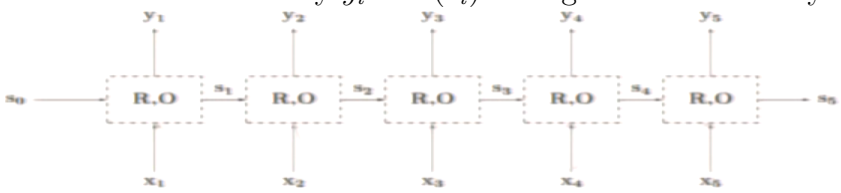
There is also a hard-limit of how many words can be involved in the window. So if our context window considers the two previous and two subsequent words, then sequences longer than 5 words are not taken into account.

Also note that due to the structure of concatenation the network will learn some properties about the order of words in the context window.

18.1 Recurrent Neural Networks

This is a NN specifically made for sequence modeling. Its inputs are the initial state, and the whole sequence of words, and it will output a sequence of states and a sequence of emissions.

Each state $s_i = R(s_{i-1}, x_i)$ takes the previous state and the current word into account, and the emission is done by $y_i = O(s_i)$ taking into account only the current state.



Note that R and O are always the same, meaning its the same set of parameters that parameterize the functions at each state.

This kind of model can learn dependencies between words that are far apart. For example, *look up* can be split: *I will look the word that you have described that doesn't make sense to her up*. This model will be able to learn this kind of structure. Or another example is pronouns, which may refer to nouns mentioned several sentences ago. These kinds of things cannot be easily modeled with HMMs or LC-CRFs.

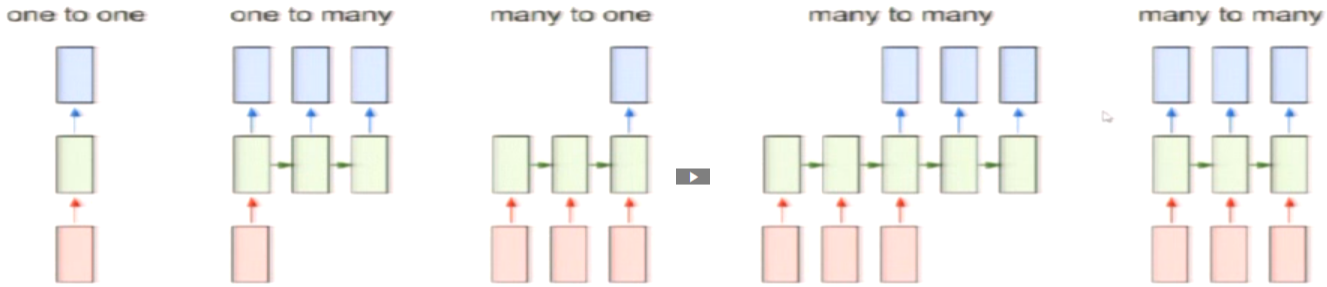
18.1.1 Comparing RNN with LC-CRFs

At each step of the sequence, LC-CRFs define a linear model. Meanwhile RNNs replace this with a NN which could require much more data to train.

LC-CRFs performance depends heavily on good selections of input features, whereas the RNN is somewhat less dependent, but can also learn new features itself.

LC-CRFs make local independences assumptions, and inference is polynomial time. RNNs need some approximate inference algorithm and does not use this independence assumption.

18.1.2 Types of RNNs

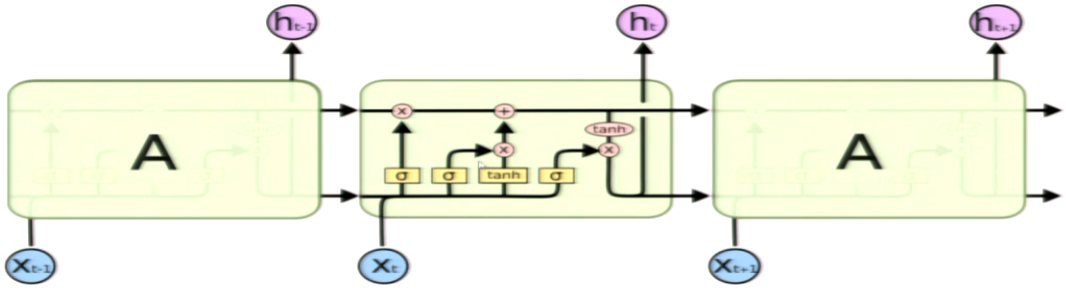


Document classification fits well with the one-to-one model, if we construct the input vector as a single input, and then we'll have one output label. It could also be many to one, if we feed in one word at a time, and generating a single output at the end.

Language modeling would fit well with many to many because we're generating multiple words based on several inputs. Or if we only care about the next word, it can be many to one, where we feed the single output word into the next RNN as part of the context.

POS tagging fits into the rightmost many to many architecture since we want to tag every word in a sequence exactly once.

18.2 Long Short-Term Memory Networks (LSTM)



The LSTM essentially is a RNN that can decide whether to forget or remember information. Each unit takes the previous state vector and the current word vector. The current word vector is passed through a (parameterized) Sigmoid function so that each entry is between 0 and 1. Then it performs component-wise mutliplication on the two vectors. This part implements the forgetting and remembering of previous information. Next, a component wise multiplication between a different sigmoid function and the tanh function (squashes between -1 and 1) decides how much of the current information information we want to use. The output of this is added to the previous information vector. Next, to determine the current tag we use another sigmoid to inform the current tag, and pass through another tanh to determine what to pass along to the next time step.

Essentially, the sigmoid parts are gates deciding how much of the previous, current and future information is useful, and the tanh are just transformations to make the information more useful.

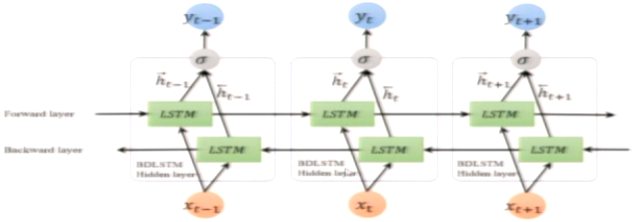
These are popular because they solve a few problems from regular RNNs. For example, the Vanishing and Exploding Gradients problem:
When training the RNN, suppose we ouput some tag on the last timestep, and we compute the loss function. Next we want to update the weights way back at the first timestep. We'll have to backpropagate the loss all the way though all the timesteps. If the gradient norms are small, the gradient will eventually go to near zero (no learning), or if its large it will explode to infinity (extreme overcorrection). This is because there are repeated applications of the same weight matricies.

It turns out that in LSTMs, you can propagate the cell state directly and it ends up being roughly the same as the partial derivative in regular backpropagation.

18.2.1 Bidirectional LSTMs

Most standard LSTMs go only forward in time. For real-time applications this makes sense, since we have no knowledge of the future.

Bidirectional LSTMs have a forward layer, and a backward layer that can read things backward in time.

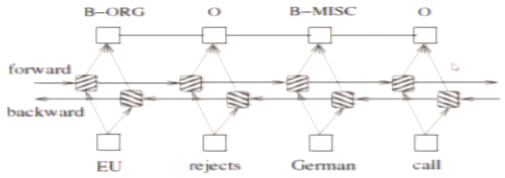


The final prediction is then just the concatenation of the forward and backward hidden layers. This allows us to take past and future information

18.2.2 Combining LSTMs and CRFs

LSTMs allow learning complex relationships between inputs and ouptuts due to the NNs and non-linearities. However we still have that each output is independent of the others. But with LC-CRFs we allow learning relationships between output labels. It would be nice if we could combine the two, and we can!

Essentially you take a bi-LSTM and add a LC-CRF layer on top.



The LSTM will generate probabilities for each tag. This, along with the transition probabilities between the tags (which needs to be learned) are the features for the CRF.