Unit III: N-gram Language Model

The role of language models; Simple N-gram models. Estimating parameters and smoothing; evaluating language models. Part of Speech Tagging and Sequence Labeling: Lexical syntax. Hidden Markov Models (Forward and Viterbi algorithms)

The role of language models:

Language models determine word probability by analyzing text data. They interpret this data by feeding it through an algorithm that establishes rules for context in natural language. Then, the model applies these rules in language tasks to accurately predict or produce new sentences.

Language models determine word probability by analyzing text data. They interpret this data by feeding it through an algorithm that establishes rules for context in natural language. Then, the model applies these rules in language tasks to accurately predict or produce new sentences.

There are several different probabilistic approaches to modeling language, which vary depending on the purpose of the language model. From a technical perspective, the various types differ by the amount of text data they analyze and the math they use to analyze it. For example, a language model designed to generate sentences for an automated Twitter bot may use different math and analyze text data in a different way than a language model designed for determining the likelihood of a search query.

Importance of language modeling

Language modeling is crucial in modern NLP applications. It is the reason that machines can understand qualitative information. Each language model type, in one way or another, turns qualitative information into quantitative information. This allows people to communicate with machines as they do with each other to a limited extent.

It is used directly in a variety of industries including tech, finance, healthcare, transportation, legal, military and government. Additionally, it's likely most people reading this have interacted with a language model in some way at some point in the day, whether it be through Google search, an autocomplete text function or engaging with a voice assistant.

The roots of language modeling as it exists today can be traced back to 1948. That year, Claude Shannon published a paper titled "A Mathematical Theory of Communication." In it, he detailed the use of a stochastic model called the Markov chain to create a statistical model for the sequences of letters in English text. This paper had a large impact on the telecommunications industry, laid the groundwork for information theory and language modeling. The Markov model is still used today, and n-grams specifically are tied very closely to the concept.

Uses and examples of language modeling

Language models are the backbone of natural language processing (NLP). Below are some NLP tasks that use language modeling, what they mean, and some applications of those tasks:

- Speech recognition -- involves a machine being able to process speech audio. This is commonly used by voice assistants like Siri and Alexa.
- Machine translation -- involves the translation of one language to another by a machine. Google Translate and Microsoft Translator are two programs that do this. SDL Government is another, which is used to translate foreign social media feeds in real time for the U.S. government.
- Parts-of-speech tagging -- involves the markup and categorization of words by certain grammatical characteristics. This is utilized in the study of linguistics, first and perhaps most famously in the study of the Brown Corpus, a body of composed of random English prose that was designed to be studied by computers. This corpus has been used to train several important language models, including one used by Google to improve search quality.

- Parsing -- involves analysis of any string of data or sentence that conforms to formal grammar and syntax rules. In language modeling, this may take the form of sentence diagrams that depict each word's relationship to the others. Spell checking applications use language modeling and parsing.
- Sentiment analysis -- involves determining the sentiment behind a given phrase. Specifically, it can be used to understand opinions and attitudes expressed in a text. Businesses can use this to analyze product reviews or general posts about their product, as well as analyze internal data like employee surveys and customer support chats. Some services that provide sentiment analysis tools are Repustate and Hubspot's ServiceHub. Google's NLP tool -- called Bidirectional Encoder Representations from Transformers (BERT) -- is also used for sentiment analysis.
- Optical character recognition -- involves the use of a machine to convert images of text into machine encoded text. The image may be a scanned document or document photo, or a photo with text somewhere in it -- on a sign, for example. It is often used in data entry when processing old paper records that need to be digitized. In can also be used to analyze and identify handwriting samples.
- Information retrieval -- involves searching in a document for information, searching for documents in general, and searching for metadata that corresponds to a document. Web browsers are the most
- common information retrieval applications.

Simple N-gram models:

N-gram Language Model:

N-grams are defined as the **contiguous sequence of n items** that can be extracted from a given sample of text or speech. The items can be letters, words, or base pairs according to the application. The N-grams typically are collected from a **text or speech corpus** (Usually a corpus of long text dataset).

- N-grams can also be seen as a **set of co-occurring words** within a given window computed by basically moving the window some k words forward (k can be from 1 or more than 1).
- The co-occurring words are called "n-grams" and "n" is a number saying how long a string of words we have considered in the construction of n-grams.
- Unigrams are single words, bigrams are two words, trigrams are three words, 4-grams are four words, 5-grams are five words, etc.
- When correcting for spelling errors, sometimes dictionary lookups will not help. For example, in the phrase "in about fifteen mineuts" the word 'minuets' is a valid dictionary word but it's incorrect in this context. N-gram models can correct such errors.
-

An N-gram language model predicts the probability of a given N-gram within any sequence of words in the language. A good N-gram model can predict the next word in the sentence i.e the value of p(w|h)

Example of N-gram such as unigram (“This”, “article”, “is”, “on”, “NLP”) or bi-gram (‘This article’, ‘article is’, ‘is on’,’on NLP’).

Now, we will establish a relation on how to find the next word in the sentence using

. We need to calculate p(w|h), where is the candidate for the next word. For example in the above example, lets’ consider, we want to calculate what is the probability of the last word being “NLP” given the previous words:

After generalizing the above equation can be calculated as:

$p(w_5|w_1, w_2, w_3, w_4)$ *or* $P(W) = p(w_n|w_1, w_2 ... w_n)$

But how do we calculate it? The answer lies in the chain rule of probability:

$$P(A|B) = \frac{P(A,B)}{P(B)}$$
$$P(A,B) = P(A|B)P(B)$$

Now generalize the above equation:

$$P(X_1, X_2, ..., X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)....P(X_n|X_1, X_2, ...X_n)$$
$$P(w_1w_2w_3...w_n) = \prod_i P(w_i|w_1w_2...w_n)$$

Simplifying the above formula using Markov assumptions:

$$P(w_i|w_1, w_2, ...w_{i-1}) \approx P(w_i|w_{i-k}, ...w_{i-1})$$

- **For unigram:**

  $$P(w_1w_2, ...w_n) \approx \prod_i P(w_i)$$
- **For Bigram:**
- $$P(w_i|w_1w_2, ..w_{i-1}) \approx P(w_i|w_{i-1})$$

EXAMPLES:

Consider two sentences: "There was heavy rain" vs. "There was heavy flood". From experience, we know that the former sentence sounds better. An N-gram model will tell us that "heavy rain" occurs much more often than "heavy flood" in the training corpus. Thus, the first sentence is more probable and will be selected by the model.

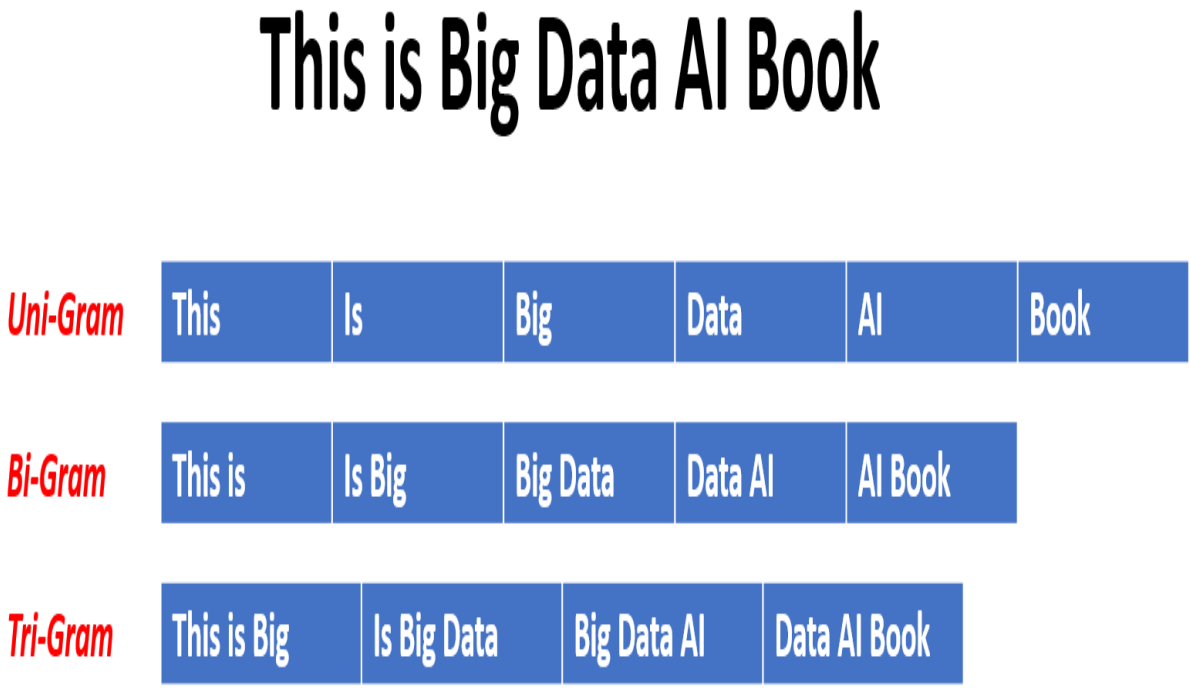
An n-gram model for the above example would calculate the following probability:

P('There was heavy rain') = P('There', 'was', 'heavy', 'rain') =
P('There')P('was'|'There')P('heavy'|'There was')P('rain'|'There was heavy')

Since it's impractical to calculate these conditional probabilities, using *Markov assumption*, we approximate this to a bigram model:

P('There was heavy rain') ~
P('There')P('was'|'There')P('heavy'|'was')P('rain'|'heavy')

A model that simply relies on how often a word occurs without looking at previous words is called **unigram**. If a model considers only the previous word to predict the current word, then it's called **bigram**. If two previous words are considered, then it's a **trigram** model.



- **What are some limitations of N-gram models?**

  A model trained on the works of Shakespeare will not give good predictions when applied to another genre. We need to therefore ensure that the training corpus looks similar to the test corpus.

  There's also the problem of **Out of Vocabulary (OOV)** words. These are words that appear during testing but not in training. One way to solve this is to start with a fixed vocabulary and convert OOV words in training to `UNK` pseudo-word.

  In one study, when applied to sentiment analysis, a bigram model outperformed a unigram model but the number of features doubled. Thus, scaling N-gram models to larger datasets or moving to a higher N needs good feature selection techniques.

N-gram models poorly capture longer-distance context. It's been shown that after 6-grams, performance gains are limited. Other language models such cache LM, topic-based LM and latent semantic indexing do better.

- **What is smoothing in the context of N-gram modelling?**

  It's quite possible that some word sequences occur in test data that were never seen during training. When this happens, the probability of the sequence equals zero. Evaluation is also difficult since perplexity metric becomes infinite.

  The usual way to solve this is to give non-zero counts to N-grams that are seen in testing but not in training. We can't just add 1 to all the zero counts since the overall probability distribution will not be normalized. Instead, we remove some probability mass from non-zero counts (called **discounting**) and add them to the zero counts. The overall process is called **smoothing**.

  The simplest technique is **Laplace Smoothing** where we add 1 to all counts including non-zero counts. An alternative is to add k, with k tuned using test data. Other techniques include Good-Turing Discounting, Witten-Bell Discounting, and Kneser-Ney Smoothing. All of these try to estimate the count of things never seen based on count of things seen once. **Kneser-Ney Smoothing** provides a good baseline and it's based on **absolute discounting**.

Estimating parameters and smoothing:

By estimating parameters and smoothing, language models can better capture the underlying patterns and structures in natural language, resulting in more accurate predictions and better performance on NLP tasks.

Estimating Parameters:

In an N-gram model, the parameters to be estimated are the probabilities of each n-gram (i.e., a sequence of n words) occurring in the training corpus. For example, to estimate the probability of a bigram (i.e., a sequence of two words), we count the number of times that bigram appears in the corpus and divide it by the number of times the first word of that bigram appears in the corpus. Similarly, to estimate the probability of a trigram (i.e., a sequence of three words), we count the number of times that trigram appears in the corpus and divide it by the number of times the first two words of that trigram appear in the corpus.

Smoothing:

N-gram models suffer from the problem of sparsity, where many possible n-grams have not been observed in the training data. To address this issue, smoothing techniques are used to assign non-zero probabilities to unseen n-grams. One of the most commonly used smoothing techniques in N-gram models is the add-k smoothing method, where a small constant k is added to the count of each n-gram. Another popular smoothing technique is the Good-Turing smoothing, which estimates the probability of unseen n-grams by using the frequencies of n-grams that occur only once in the training corpus.

Evaluating Language Models:

Language models are very useful in a **broad range of applications** like speech recognition, machine translation part-of-speech tagging, parsing, Optical Character Recognition (OCR), handwriting recognition, information retrieval, and many other daily tasks

- One of the main steps in the usage of language models is to **evaluate the performance beforehand** and use them in further tasks.

- This lets us build **confidence in the handling** of the language models in NLP and also lets us know if there are any places where the model may behave uncharacteristically.

In practice, we need to decide on the dataset to use, the method to evaluate, and also select a metric to evaluate language models. Let us learn about each of the elements further.

How to Evaluate a Language Model?

- **Evaluating a language model** lets us know whether one language model is better than another during experimentation and also to choose among already trained models.
- There are two ways to evaluate language models in NLP: Extrinsic evaluation and Intrinsic evaluation.
    - Intrinsic evaluation captures how well the model captures what it is supposed to capture like probabilities.
    - Extrinsic evaluation (or task-based evaluation) captures how useful the model is in a particular task.
- **Comparing among language models:** We compare models by collecting a corpus of text which is common for models which we are comparing for.
    - We then divide the data into training and test sets and train the parameters of both models on the training set.
    - We then compare how well the two trained models fit the test set.

What Does Evaluating a Model Mean?

- After we train models, Whichever model assigns a higher probability to the test set is generally considered to accurately predicts the test set and hence a better model.
- Among multiple probabilistic language models, the better model is the one that has a tighter fit to the test data or that better predicts the details of the test data and hence will assign a higher probability to the test data.

Issue of Data Leakage or Bias in Language Models

- Most evaluation metrics for language models in NLP are based on test set probability, so it is important **not to let the test sentences into the training set**.

- Example: Assuming we are trying to compute the probability of a particular test sentence, and if our test sentence is part of the training corpus, we will **mistakenly assign** it an **artificially high probability** when it occurs in the test set.
    - We call this situation training on the test set.
    - Training on the test set introduces a bias that makes the probabilities all look too high, and causes huge inaccuracies metrics like in perplexity.

Extrinsic Evaluation

Extrinsic Evaluation is the best way to evaluate the performance of a language model by **embedding** it in an application and measuring how much the application **improves**.

- It is an **end-to-end evaluation** where we can understand if a particular improvement in a component is really going to help the task at hand.
- Example: For speech recognition, we can **compare the performance of two language models** by running the speech recognizer twice, once with each language model, and seeing which gives the more accurate transcription.

Intrinsic Evaluation

We need to take advantage of intrinsic measures because **running big language models** in NLP systems end-to-end is often very **expensive** and it is easier to have a metric that can be used to quickly evaluate potential improvements in a language model.

An intrinsic evaluation metric is one that measures the **quality of a model-independent** of any application.

- We also need a test set for an intrinsic evaluation of a language model in NLP
- The probabilities of an N-gram model training set come from the corpus it is trained on, the training set or training corpus.
- We can then measure the quality of an N-gram model by its performance on some unseen test set data called the test set or test corpus.
- We will also sometimes call test sets and other datasets that are not in our training sets held out corpora because we hold them out from the training data.

Good scores during intrinsic evaluation do not always mean better scores during extrinsic evaluation, so we need both types of evaluation in practice.

Perplexity

Perplexity is a very common method to evaluate the language model on some held-out data. It is a measure of **how well a probability model predicts a sample**.

- Perplexity is also an **intrinsic measure** (without the use of external datasets) to evaluate the performance of language models which come under NLP.
    - Perplexity as a metric quantifies **how uncertain a model is about the predictions it makes**. Low perplexity only guarantees a model is confident, not accurate.
    - Perplexity also often **correlates** well with the **model's final real-world performance** and it can be quickly calculated using just the probability distribution the model learns from the training dataset.

The Intuition

- The basic intuition is that the **higher the perplexity measure** is, the **better** the language model is at modeling unseen sentences.
- Perplexity can also be seen as a **simple monotonic function of entropy**. But perplexity is often used instead of entropy due to the fact that it is **arguably more intuitive to our human minds than entropy**.

Calculating Perplexity

- Perplexity of a probability model like language models in NLP: For a model of an **unknown probability distribution**, and a proposed probability model, we can evaluate perplexity measure mathematically as $b^{-\frac{1}{N}\sum_{i=1}^{N}\log_b q(x_i)}$
    - We can choose b as 2.
    - In general, better models assign higher probabilities to the test events, hence good models will have lower perplexity values and are less surprised by the test sample.

    - If all the probabilities were 1, then the perplexity would be 1 and the model would perfectly predict the text. Conversely, the perplexity will be higher for poorer language models.

- **Perplexity** denoted by PP of a discrete probability distribution p is mathematically defined as

$$PP(p) := 2^{H(p)} = 2^{-\sum_x p(x) \log_2 p(x)} = \prod_x p(x)^{-p(x)}$$

    - Where H(p) is the entropy (in bits) of the distribution and x ranges over events which we will learn about further.
    - Perplexity of a random variable X may be defined as the **perplexity of the distribution over its possible values x**.
- One other formulation for Perplexity from the perspective of language models in NLP: It is the **multiplicative inverse of the probability** assigned to the test set by the language model normalized by the number of words in the test set.
    - We can define perplexity mathematically as:
    - $PP(W) = P(w_1 w_2 ... w_N)^{-\frac{1}{N}}$

    - We know that if a language model can predict unseen words from the test set if the P(a sentence from a test set) is highest, then such a language model is more accurate.

Pros and Cons

- **Advantages of using Perplexity**
    - Fast to calculate and hence allows researchers to select among models that are unlikely to perform well in real-world scenarios where computing is prohibitively costly and testing is time-consuming and expensive.
    - Useful to have an estimate of the model uncertainty/information density
- **Disadvantages of Perplexity**

- **Not good for final evaluation** since it just measures the model's confidence and not its accuracy
- Hard to make comparisons across different datasets with different context lengths, vocabulary sizes, word vs. character-based models, etc.
- Perplexity can also end up rewarding models that mimic outdated datasets.

Introduction to POS Tagging

Part-of-speech (POS) tagging is a process in natural language processing (NLP) where each word in a text is labelled with its corresponding part of speech. This can include nouns, verbs, adjectives, and other grammatical categories.

POS tagging is useful for a variety of NLP tasks, such as information extraction, named entity recognition, and machine translation. It can also be used to identify the grammatical structure of a sentence and to disambiguate words that have multiple meanings.

POS tagging is typically performed using machine learning algorithms, which are trained on a large annotated corpus of text. The algorithm learns to predict the correct POS tag for a given word based on the context in which it appears.

There are various POS tagging schemes that have been developed, each with its own set of tags and rules. Some common POS tagging schemes include the Penn Treebank tag set and the Universal Dependencies tag set.

Let's take an example,

Text: "The cat sat on the mat."

POS tags:

- The: determiner
- cat: noun
- sat: verb
- on: preposition
- the: determiner
- mat: noun

In this example, each word in the sentence has been labelled with its corresponding part of speech. The determiner “the” is used to identify specific nouns, while the noun “cat” refers to a specific animal. The verb “sat” describes an action, and the preposition “on” describes the relationship between the cat and the mat.

POS tagging is a useful tool in natural language processing (NLP) as it allows algorithms to understand the grammatical structure of a sentence and to disambiguate words that have multiple meanings. It is typically performed using machine learning algorithms that are trained on a large annotated corpus of text.

Identifying part of speech of word is not just mapping words to their respective POS tags. Same word might have different part of speech tag based on different context. Thus it is not possible to have common mapping for parts of speech tags.

When you have a huge corpus manually finding different part-of-speech for each word is a scalable solution. As tagging itself might take days. This is why we rely on tool-based POS tagging.

But why are we tagging these words with their parts of speech?

Use of Parts of Speech Tagging in NLP

There are several reasons why we might tag words with their parts of speech (POS) in natural language processing (NLP):

- **To understand the grammatical structure of a sentence:** By labelling each word with its POS, we can better understand the syntax and structure of a sentence. This is useful for tasks such as machine translation and information extraction, where it is important to know how words relate to each other in the sentence.
- **To disambiguate words with multiple meanings:** Some words, such as “bank,” can have multiple meanings depending on the context in which they are used. By labelling each word with its POS, we can disambiguate these words and better understand their intended meaning.
- **To improve the accuracy of NLP tasks:** POS tagging can help improve the performance of various NLP tasks, such as named entity recognition and text classification. By providing additional context and information about the words in a text, we can build more accurate and sophisticated algorithms.

- **To facilitate research in linguistics:** POS tagging can also be used to study the patterns and characteristics of language use and to gain insights into the structure and function of different parts of speech.

Steps Involved in the POS tagging

Here are the steps involved in a typical example of part-of-speech (POS) tagging in natural language processing (NLP):

- **Collect a dataset of annotated text:** This dataset will be used to train and test the POS tagger. The text should be annotated with the correct POS tags for each word.
- **Pre-process the text:** This may include tasks such as tokenization (splitting the text into individual words), lowercasing, and removing punctuation.
- **Divide the dataset into training and testing sets:** The training set will be used to train the POS tagger, and the testing set will be used to evaluate its performance.
- **Train the POS tagger:** This may involve building a statistical model, such as a hidden Markov model (HMM), or defining a set of rules for a rule-based or transformation-based tagger. The model or rules will be trained on the annotated text in the training set.
- **Test the POS tagger:** Use the trained model or rules to predict the POS tags of the words in the testing set. Compare the predicted tags to the true tags and calculate metrics such as precision and recall to evaluate the performance of the tagger.
- **Fine-tune the POS tagger:** If the performance of the tagger is not satisfactory, adjust the model or rules and repeat the training and testing process until the desired level of accuracy is achieved.
- **Use the POS tagger:** Once the tagger is trained and tested, it can be used to perform POS tagging on new, unseen text. This may involve preprocessing the text and inputting it into the trained model or applying the rules to the text. The output will be the predicted POS tags for each word in the text.

Application of POS Tagging

There are several real-life applications of part-of-speech (POS) tagging in natural language processing (NLP):

- **Information extraction:** POS tagging can be used to identify specific types of information in a text, such as names, locations, and organizations. This is useful for tasks such as extracting data from news articles or building knowledge bases for artificial intelligence systems.
- **Named entity recognition:** POS tagging can be used to identify and classify named entities in a text, such as people, places, and organizations. This is useful for tasks such as building customer profiles or identifying key figures in a news story.
- **Text classification:** POS tagging can be used to help classify texts into different categories, such as spam emails or sentiment analysis. By analysing the POS tags of the words in a text, algorithms can better understand the content and tone of the text.
- **Machine translation:** POS tagging can be used to help translate texts from one language to another by identifying the grammatical structure and relationships between words in the source language and mapping them to the target language.
- **Natural language generation:** POS tagging can be used to generate natural-sounding text by selecting appropriate words and constructing grammatically correct sentences. This is useful for tasks such as chatbots and virtual assistants.

Types of POS Tagging in NLP

Rule Based POS Tagging

Rule-based part-of-speech (POS) tagging is a method of labeling words with their corresponding parts of speech using a set of pre-defined rules. This is in contrast to machine learning-based POS tagging, which relies on training a model on a large annotated corpus of text.

In a rule-based POS tagging system, words are assigned POS tags based on their characteristics and the context in which they appear. For example, a rule-based POS tagger might assign the tag “noun” to any word that ends in “-tion” or “-ment,” as these suffixes are often used to form nouns.

Rule-based POS taggers can be relatively simple to implement and are often used as a starting point for more complex machine learning-based taggers. However, they can be less accurate and less efficient than machine learning-based taggers, especially for tasks with large or complex datasets.

Here is an example of how a rule-based POS tagger might work:

- Define a set of rules for assigning POS tags to words. For example:

- If the word ends in "-tion," assign the tag "noun."
- If the word ends in "-ment," assign the tag "noun."
- If the word is all uppercase, assign the tag "proper noun."
- If the word is a verb ending in "-ing," assign the tag "verb."

- Iterate through the words in the text and apply the rules to each word in turn. For example:

- "Nation" would be tagged as "noun" based on the first rule.
- "Investment" would be tagged as "noun" based on the second rule.
- "UNITED" would be tagged as "proper noun" based on the third rule.
- "Running" would be tagged as "verb" based on the fourth rule.

- Output the POS tags for each word in the text.

This is a very basic example of a rule-based POS tagger, and more complex systems can include additional rules and logic to handle more varied and nuanced text.

Statistical POS Tagging

Statistical part-of-speech (POS) tagging is a method of labeling words with their corresponding parts of speech using statistical techniques. This is in contrast to rule-based POS tagging, which relies on pre-defined rules, and to unsupervised learning-based POS tagging, which does not use any annotated training data.

In statistical POS tagging, a model is trained on a large annotated corpus of text to learn the patterns and characteristics of different parts of speech. The model uses this training data to predict the POS tag of a given word based on the context in which it appears and the probability of different POS tags occurring in that context.

Statistical POS taggers can be more accurate and efficient than rule-based taggers, especially for tasks with large or complex datasets. However, they require a large amount of annotated training data and can be computationally intensive to train.

Here is an example of how a statistical POS tagger might work:

- Collect a large annotated corpus of text and divide it into training and testing sets.
- Train a statistical model on the training data, using techniques such as maximum likelihood estimation or hidden Markov models.
- Use the trained model to predict the POS tags of the words in the testing data.
- Evaluate the performance of the model by comparing the predicted tags to the true tags in the testing data and calculating metrics such as precision and recall.
- Fine-tune the model and repeat the process until the desired level of accuracy is achieved.
- Use the trained model to perform POS tagging on new, unseen text.

There are various statistical techniques that can be used for POS tagging, and the choice of technique will depend on the specific characteristics of the dataset and the desired level of accuracy.

Transformation-based tagging (TBT)

Transformation-based tagging (TBT) is a method of part-of-speech (POS) tagging that uses a series of rules to transform the tags of words in a text. This is in contrast to rule-based POS tagging, which assigns tags to words based on pre-defined rules, and to statistical POS tagging, which relies on a trained model to predict tags based on probability.

In TBT, a set of rules is defined to transform the tags of words in a text based on the context in which they appear. For example, a rule might change the tag of a verb to a noun if it appears after a determiner such as “the.” The rules are applied to the text in a specific order, and the tags are updated after each transformation.

TBT can be more accurate than rule-based tagging, especially for tasks with complex grammatical structures. However, it can be more computationally intensive and requires a larger set of rules to achieve good performance.

Here is an example of how a TBT system might work:

- Define a set of rules for transforming the tags of words in the text. For example:

- If the word is a verb and appears after a determiner, change the tag to “noun.”
- If the word is a noun and appears after an adjective, change the tag to “adjective.”

- Iterate through the words in the text and apply the rules in a specific order. For example:

- In the sentence “The cat sat on the mat,” the word “sat” would be changed from a verb to a noun based on the first rule.
- In the sentence “The red cat sat on the mat,” the word “red” would be changed from an adjective to a noun based on the second rule.

- Output the transformed tags for each word in the text.

This is a very basic example of a TBT system, and more complex systems can include additional rules and logic to handle more varied and nuanced text.

Hidden Markov Model POS tagging

Hidden Markov models (HMMs) are a type of statistical model that can be used for part-of-speech (POS) tagging in natural language processing (NLP). In an HMM-based POS tagger, a model is trained on a large annotated corpus of text to learn the patterns and characteristics of different parts of speech. The model uses this training data to predict the POS tag of a given word based on the probability of different tags occurring in the context of the word.

An HMM-based POS tagger consists of a set of states, each corresponding to a possible POS tag, and a set of transitions between the states. The model is trained on the training data to learn the probabilities of transitioning from one state to another and the probabilities of observing different words given a particular state.

To perform POS tagging on a new text using an HMM-based tagger, the model uses the probabilities learned during training to compute the most likely sequence of POS tags for the words in the text. This is typically done using the Viterbi algorithm, which calculates the probability of each possible sequence of tags and selects the most likely one.

HMMs are widely used for POS tagging and other tasks in NLP due to their ability to model complex sequential data and their efficiency in computation. However, they can be sensitive to the quality of the training data and may require a large amount of annotated data to achieve good performance.

Challenges in POS Tagging

Some common challenges in part-of-speech (POS) tagging include:

- **Ambiguity:** Some words can have multiple POS tags depending on the context in which they appear, making it difficult to determine their correct tag. For example, the word "bass" can be a noun (a type of fish) or an adjective (having a low frequency or pitch).
- **Out-of-vocabulary (OOV) words:** Words that are not present in the training data of a POS tagger can be difficult to tag accurately, especially if they are rare or specific to a particular domain.
- **Complex grammatical structures:** Languages with complex grammatical structures, such as languages with many inflections or free word order, can be more challenging to tag accurately.
- **Lack of annotated training data:** Some languages or domains may have limited annotated training data, making it difficult to train a high-performing POS tagger.
- **Inconsistencies in annotated data:** Annotated data can sometimes contain errors or inconsistencies, which can negatively impact the performance of a POS tagger.

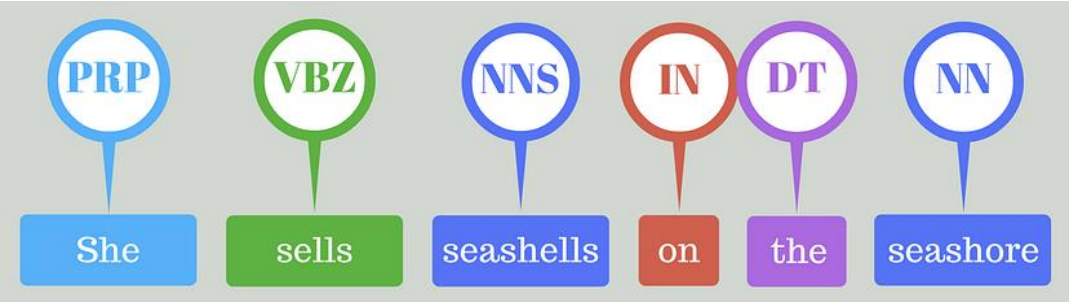
Sequence Labeling

Sequence labeling is a typical NLP task which assigns a class or label to each token in a given input sequence. In this context, a single word will be referred to as a "token". These tags or labels can be used in further downstream models as features of the token, or to enhance search quality by naming spans of tokens. In question answering and search tasks, we can use these spans as entities to specify our search query (e.g..,. "Play a movie by Tom Hanks") we would like to label words such as: [Play, movie, Tom Hanks]. With these parts removed, we can use the verb "play" to specify the wanted action, the word "movie" to specify the intent of the action and Tom Hanks as the single subject for our search. To do this, we need a way of labeling these words to later retrieve them for our query.

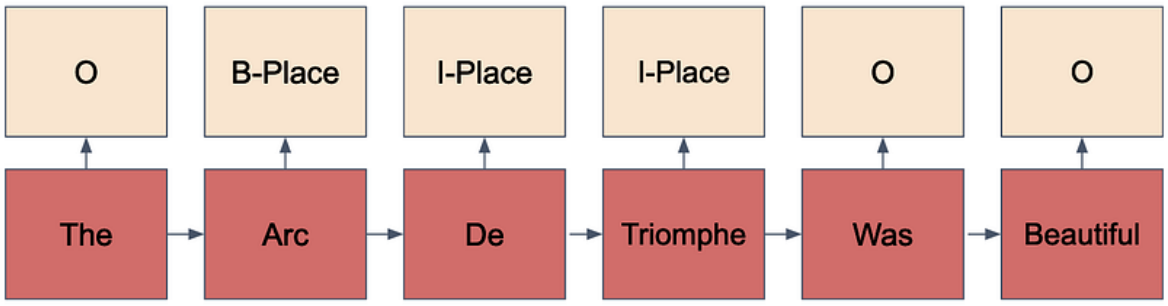
Two forms of sequence labeling are:

- Token Labeling: Each token gets an individual Part of Speech (POS) label and
- Span Labeling: Labeling segments or groups of words that contain one tag (Named Entity Recognition, Syntactic Chunks).

Raw labeling is a common task which involves labeling a single word unit with its respective tag. One common application of this is part-of-speech(POS) tagging. Given a dataset of tokens and their POS tags within their given context, it is possible to train a model that will learn from the context and generalize to other unseen texts and predict their POS.



Segmentation labeling is another form of sequence tagging, where we have a single entity such as a name that spans multiple tokens. To simplify this task, we write it as a raw labeling task with modified labels to represent tokens as members of a span. These spans are labeled with a **BIO** tag representing the **B**eginning, **I**nner, and **O**utside of entities:



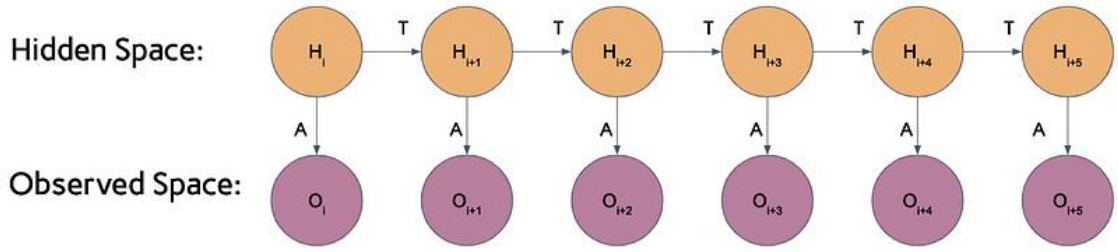
By further breaking down multiword entities into groups of **BIO** tags that represent the span of a single entity, we can train a model to tag where a single entity begins and ends. Here, “Arc de Triomphe” are three tokens that represent a single entity. By identifying the beginning and inner of the entity, they can be joined to form a single representation. This is important in tasks such as question answering, where we want to know the tokens “Tom” and “Hanks” refer to the same person, without separating them, thus allowing us to generate a more accurate query. For more details, please refer to the blog by Hal Daumé III in Getting Started in: Sequence Labeling

Previous Methods : Hidden Markov Models (HMMs)

HMMs are “a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobservable (i.e. hidden) states”. They are designed to model the joint distribution *P(H , O)* , where *H* is the hidden state and *O* is the observed state. For example , in the context of POS tagging, the objective would be to build an HMM to model *P(word | tag)* and compute the label probabilities given observations using Bayes’ Rule:

$$P(H|O) = \frac{P(O|H)P(H)}{P(O)}$$

HMM graphs consist of a *Hidden Space* and *Observed Space,* where the hidden space consists of the labels and the observed space is the input. These spaces are connected via transition matrices {*T*,*A*} to represent the probability of transitioning from one state to another following their connections. Each connection represents a distribution over possible options; given our tags, this results in a large search space of the probability of all words given the tag.



The main idea behind HMMs is that of making observations and traveling along connections based on a probability distribution. In the context of sequence tagging, there exists a changing observed state (the tag) which changes as our hidden state (tokens in the source text) also changes.

Previous Methods : Maximum Entropy Markov Models (MEMMs)

However, there are two problems with HMMs. First, HMMs are limited to only discrete states and only take into account the last known state. Furthermore, it is hard to create a state as a function of multiple others, and the features allowed are limited. These problems limit the utilization of our context, where it would be preferable to consider our sequence as a whole rather than strictly assume independence as in HMMs. Hence, a new model was needed to overcome these problems.

Given an observation space, **Maximum Entropy Markov Models (**MEMMs) predict the state sequence. MEMMs use a maximum entropy framework for features and local normalization. In contrast to HMMs, MEMM's objective is to model $P(O|H)$ where O is our label as opposed to the HMM joint distribution objective of P(O, H)

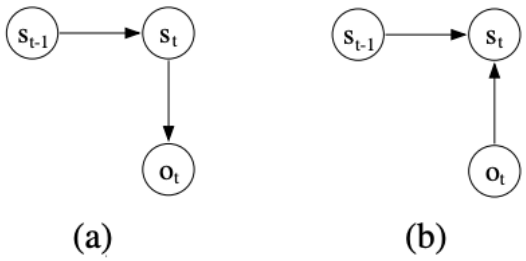


Figure 1. **(a)** The dependency graph for a traditional HMM; **(b)** for our conditional maximum entropy Markov model.

Previous Methods : Conditional Random Fields (CRFs)

MEMMs also have a well-known issue known as **label bias.** The label bias problem was introduced due to MEMMs applying local normalization. This often leads to the model getting stuck in local minima during decoding. The local minima trap occurs because the overall model favors nodes with the least amount of transitions. To solve this, Conditional Random Fields (CRFs) normalize globally and introduce an undirected graphical structure.

$$p(l|s) = \frac{exp[score(l|s)]}{\sum_{l'} exp[score(l'|s)]} = \frac{exp[\sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(s,i,l_i,l_{i-1})]}{\sum_{l'} exp[\sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(s,i,l'_i,l'_{i-1})]}$$

Lexical syntax:

Lexical syntax is a subfield of natural language processing (NLP) that deals with the study of the syntax or grammatical structure of words or lexical items in a language. It focuses on the relationships between the surface forms of words and their syntactic properties, including the grammatical categories, morphological properties, and syntactic dependencies.

Lexical syntax plays a crucial role in many NLP applications, such as part-of-speech tagging, parsing, and semantic analysis. For instance, in part-of-speech tagging, lexical syntax is used to determine the grammatical category or part of speech of each word in a sentence. In parsing, it helps to identify the syntactic structure of a sentence, such as the subject and object of a sentence. In

semantic analysis, it helps to determine the meaning of words and their relationships in a sentence.

One of the main challenges of lexical syntax in NLP is dealing with the ambiguity of natural language. Many words can have multiple syntactic and semantic interpretations depending on the context in which they appear. To overcome this challenge, lexical syntax relies on various techniques such as rule-based parsing, statistical parsing, and machine learning algorithms.

In summary, lexical syntax is a critical area of study in NLP that focuses on the analysis of the grammatical structure of words in a language. It plays a crucial role in many NLP applications and helps to address the challenge of ambiguity in natural language.

In natural language processing (NLP), lexical syntax refers to the rules and patterns that govern how words are used in a language. These rules include the structure of words, such as their inflection, tense, and grammatical category, as well as their relationships to other words in a sentence.

Some common examples of lexical syntax in NLP include:

Morphology: This refers to the study of the structure and formation of words, including inflection, derivation, and compounding. For example, in English, the word "walk" can be inflected to form "walked" (past tense), "walking" (present participle), and "walks" (third-person singular).

Parts of speech: This refers to the grammatical categories that words can be classified into, such as nouns, verbs, adjectives, and adverbs. Parts of speech play a critical role in determining the structure and meaning of a sentence.

Syntax: This refers to the rules that govern the arrangement of words in a sentence. For example, in English, the subject typically precedes the verb in a sentence ("the cat chased the mouse").

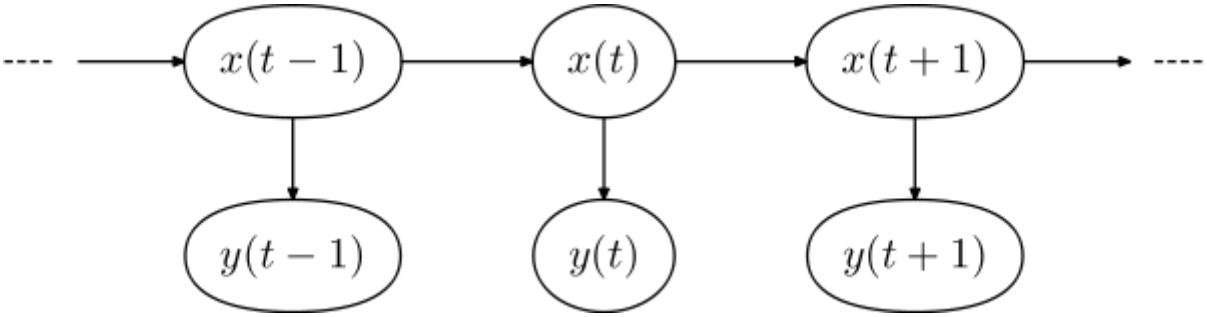
Hidden Markov Models (Forward and Viterbi algorithms):

The **forward algorithm**, in the context of a hidden Markov model (HMM), is used to calculate a 'belief state': the probability of a state at a certain time,

given the history of evidence. The process is also known as *filtering*. The forward algorithm is closely related to, but distinct from, the Viterbi algorithm.

The forward and backward algorithms should be placed within the context of probability as they appear to simply be names given to a set of standard mathematical procedures within a few fields. For example, neither "forward algorithm" nor "Viterbi" appear in the Cambridge encyclopedia of mathematics. The main observation to take away from these algorithms is how to organize Bayesian updates and inference to be efficient in the context of directed graphs of variables (see sum-product networks).

For an HMM such as this one:



this probability is written as $p(x_t|y_{1:t})$Here $x(t)$s the hidden state which is abbreviated as x_t and $y_{1:t}$ are the observations 1 to t.

The backward algorithm complements the forward algorithm by taking into account the future history if one wanted to improve the estimate for past times. This is referred to as *smoothing* and the forward/backward algorithm computes $p(x_t|y_{1:T})$ for $1 < t < T$. Thus, the full forward/backward algorithm takes into account all evidence. Note that a belief state can be calculated at each time step, but doing this does not, in a strict sense, produce the most likely state *sequence*, but rather the most likely state at each time step, given the previous history. In order to achieve the most likely sequence, the Viterbi algorithm is required. It computes the most likely state sequence given the history of observations, that is, the state sequence that maximizes $p(x_{0:t}|y_{0:t})$.

Algorithm:

1. Initialize

   $t = 0$,

   transition probabilities, $p(x_t|x_{t-1})$,

   emission probabilities, $p(y_j|x_i)$,

   observed sequence, $y_{1:T}$

   prior probability, $\alpha_0(x_0)$

2. For $t = 1$ to T

   $$\alpha_t(x_t) = p(y_t|x_t) \sum_{x_{t-1}} p(x_t|x_{t-1})\alpha_{t-1}(x_{t-1}).$$

3. Calculate $\alpha_T = \sum_{x_T} \alpha_T(x_T)$

4. Return $p(x_T|y_{1:T}) = \dfrac{\alpha_T(x_T)}{\alpha_T}$

Applications of the algorithm

The forward algorithm is mostly used in applications that need us to determine the probability of being in a specific state when we know about the sequence of observations. We first calculate the probabilities over the states computed for the previous observation and use them for the current observations, and then extend it out for the next step using the transition probability table. The approach basically caches all the intermediate state probabilities so they are computed only once. This helps us to compute a fixed state path. The process is also called posterior decoding. The algorithm computes probability much more efficiently than the naive approach, which very quickly ends up in a combinatorial explosion. Together, they can provide the probability of a given emission/observation at each position in the sequence of observations. It is from this information that a version of the most likely state path is computed ("posterior decoding"). The algorithm can be applied wherever we can train a model as we receive data using Baum-Welch[2] or any general EM algorithm. The Forward algorithm will then tell us about the probability of data with respect to what is expected from our model. One of the applications can be in the domain of Finance, where it can help decide on when to buy or sell tangible assets. It can have applications in all fields where we apply Hidden Markov Models. The popular ones include Natural language processing domains like tagging part-of-speech and speech recognition.[1] Recently it is also being used in the domain of Bioinformatics. Forward algorithm can also be applied to perform Weather speculations. We can have a HMM describing the weather and its relation to the state of observations for few consecutive days

(some examples could be dry, damp, soggy, sunny, cloudy, rainy etc.). We can consider calculating the probability of observing any sequence of observations recursively given the HMM. We can then calculate the probability of reaching an intermediate state as the sum of all possible paths to that state. Thus the partial probabilities for the final observation will hold the probability of reaching those states going through all possible paths.

Viterbi algorithm

The **Viterbi algorithm** is a dynamic programming algorithm for obtaining the maximum a posteriori probability estimate of the most likely sequence of hidden states—called the **Viterbi path**—that results in a sequence of observed events, especially in the context of Markov information sources and hidden Markov models (HMM).

The algorithm has found universal application in decoding the convolutional codes used in both CDMA and GSM digital cellular, dial-up modems, satellite, deep-space communications, and 802.11 wireless LANs. It is now also commonly used in speech recognition, speech synthesis, diarization,[1] keyword spotting, computational linguistics, and bioinformatics. For example, in speech-to-text (speech recognition), the acoustic signal is treated as the observed sequence of events, and a string of text is considered to be the "hidden cause" of the acoustic signal. The Viterbi algorithm finds the most likely string of text given the acoustic signal.