CHAPTER .1

Language Models

Imagine a copy writer has produced several versions of a text advertisement. We want to investigate the theory that creative directors pick the most creative version for the ad campaign. We have the different version, we have the final campaign, but how do we measure creativity? This is a big question, but if we just want a simple measure to compare texts, we could look at which of the texts are most unusual, compared to some standard reference. This is what **language models** do: for any text we give them, they return a probability of how likely/unusual this text is (with respect to some standard). In our example, we can use them to assign each ad version a probability. A lower probability roughly corresponds to a more creative text (or at least one that is unlike most things we have seen). We can now check whether the texts with the lowest probabilities are indeed the ones that are picked most often by the creative director for the campaign, which would give us a way of measuring and quantifying "creativity".

language models

The interesting question is of course: "How do we compute the probability of sentence?" There are indefinitely many possible sentences out there: you probably have produced several today that have never been uttered before in the history of ever. How would a model assign a probability to something infinitely large? We cannot just count and divide: after all, any number divided by infinity is going to be 0. In order to get around this problem, we divide a sentence into a sequence of smaller parts, compute their probabilities, and combine them. Formally, a language model is a probability function that takes any document S and a reference corpus θ and returns a value between 0 and 1, denoting how likely S is to be produced under the parameters θ.

Language models have a long history in NLP and speech processing (Jelinek and Mercer, 1980; Katz, 1987), because they are very useful in assessing the output quality of a system: it gives us a fast and easy way to rank the different possible outputs produced by a chat bot, or the best translation from a machine translation system, or a speech recognizer. Today, the best language models are based on neural networks (Bengio et al., 2003; Mikolov et al., 2010, inter alia), but they tend to require immense amounts of data. For most of our purposes here, a probability-based n-gram language model will work well.

More specifically, we will build a trigram LM based on **Markov chains**. Markov chains are a special kind of **weighted directed graphs** that describe how how a system changes over time. Each state in the system is typically a point in discrete "time", so in the case of language, each state is a word. Markov chains have been used for weather

Markov chains

weighted directed graphs

prediction, market development, generational changes, speech recognition, and NLP, etc. They are also the basis of HMMs (Hidden Markov Models) and CRFs (Conditional Random Fields), which are widely used in speech recognition and NLP. Lately, they have been replaced in many of these applications by neural language models, which produce better output. However, probabilisitic language models based on Markov chains are a simple and straightforward model to assess the likelihood of a document, and to generate short texts. In general, they assume that the probability of seeing a sequence (e.g., a sentence) can be decomposed into the product of a series of smaller probabilities:

$$P(w_1, w_2, ..., w_n) = \prod_i^N P(w_i|w_1, ...w_{i-1})$$

I.e., the probability of seeing a sentence is the probability of seeing each word in the sentence preceded by all others before it.

1. The Markov Assumption

In reality, this is a good idea. Each state or word depends on a long history of previous states: the weather yesterday and the day before influences the weather today, and the words you have read up to this one influence how you interpret the whole sentence. Ideally, we want to take the entire history up to the current state into account.

However, this quickly leads to exponentially complex models as we go on (just imagine the amount of words we would have to keep in memory for the last words in the previous sentences). The **Markov assumption** limits the history of each state to a fixed number of previous states. This limited history is called the **Markov order**. In a 0th order model, each state only depends on itself. In a first order model, each state depends on its direct predecessor. In a second order model, each state depends on its two previous states.

Markov assumption
Markov order

Of course this is a very simplifying assumption, and in practice, the higher the order, the better the model, but higher orders also produce sparser probability tables.

2. Trigram LMs

In a trigram LM, we use a second order Markov chain. We first extract trigram counts from a reference corpus and then estimate trigram probabilities from them. In practice, it is enough to store the raw counts and compute the probabilities "on the fly".

A trigram (i.e., an n-gram with $n = 3$) is a sequence of 3 tokens, e.g., "eat more pie". Or, more general: $u\ v\ w$. A *trigram language model* (or trigram model) takes a trigram and predicts the last word in it, based on the first two (the history), i.e., $P(w|u, v)$. In general, the history or order of an n-gram model is $n - 1$ words. Other common n for language models are 5 or 7. The longer the history, the more accurate the probability estimates.

We will follow the notation in Mike Collins' lecture notes[1]. We are going to build a trigram language model that consists of a vocabulary $\mathcal{V}$ and a parameter $P(w|u,v)$ for each trigram $u\ v\ w$. We will define that the last token in a sentence X is STOP, i.e., x_n=STOP. We will also prepend $n-1$ dummy token to each sentence, so we can count the first word, i.e., in a trigram model, x_0 and x_{-1} are set to $\star$.

We first have to collect the necessary building blocks (the trigram counts) for the LM from a corpus of sentences. This corpus determines how good our LM is: bigger corpora give us better estimates for more trigrams. However, if we are only working within a limited domain (say, all ads from a certain agency), and want to rank sentences within that, the size of our corpus is less important, since we will have seen all the necessary trigrams we want to score.

Take the following sentence: `* * eat more pie STOP` The 3-grams we can extract are

```
[('*', '*', 'eat'),
 ('*', 'eat', 'more'),
 ('eat', 'more', 'pie'),
 ('more', 'pie', 'STOP')]
```

The $\star$ symbols allow us to estimate the probability of a word starting a sentence ($P(w|*,*)$), and the `STOP` symbol allows us to estimate what words end a sentence.

3. Maximum likelihood estimation (MLE)

The next step is to get from the trigram counts to a probability estimate for a word given its history, i.e., $P(w|u,v)$. We will here use the most generic solution and use **maximum likelihood estimation** (meaning: count and divide). We define:

maximum likelihood estimation

$$P(w|u,v) = \frac{c(u,v,w)}{c(u,v)}$$

where $c(u,v,w)$ is the count of the trigram and $c(u,v)$ the number of times the history bigram is seen in the corpus.

Instead of actually storing both trigram and bigram counts, we can make use of the fact that the bigrams are a subset of the trigrams. We can keywordmarginalize out the count of a bigram from all the trigrams it occurs in, by summing over all those occurrences, using the following formula:

$$P(w|u,v) = \frac{c(u,v,w)}{c(u,v)} = \frac{c(u,v,w)}{\sum_z c(u,v,z)}$$

E.g., in order to get the probability of $P(STOP|more,pie)$, we count *all* trigrams starting with "more pie".

[1] http://www.cs.columbia.edu/ mcollins/lm-spring2013.pdf

4. Probability of a Sentence

Now that we have the necessary elemnents of a trigram language model (i.e., the trigram counts and a way to use them to compute all probabilities $P(w|u,v)$), we can use the model to calculate the probability of an entire sentence based on the Markov assumption. The probability of a sentence is the product of all the trigram probabilities involved in it:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i|x_{i-2}, x_{i-1})$$

I.e., we slide a trigram window over the sentence, compute the probability for each of them, and multiply them together. If we have a long sentence, we will get very small numbers, so it is better to use the logarithm of the probabilities again:

$$\log P(x_1, ..., x_n) = \sum_{i=1}^{n} \log P(x_i|x_{i-2}, x_{i-1})$$

Note that when using logarithms, products become sums, and division becomes subtraction, so the logarithm of the trigram probability can be in turn computed as

$$\log P(w|u,v) = \log c(u,v,w) - \log \sum_{z} c(u,v,z)$$

5. Smoothing

If we apply the LM only to the sentences in the corpus we used to collect the counts, we are done at this point. However, sometimes we want to compute the probability of a sentence that is *not* in our original corpus (for example, if we want to see how likely the sentences of another ad agency are).

Because language is so creative and any corpus, no matter how big, is limited, some perfectly acceptable three-word sequences will be missing from our model. This means that a large number of n-grams we will see will have a probability of 0 under our model, even though they should really get a non-zero probability. This is problematic, because any 0 for a single trigram in our product calculation above would make the probability of the entire sentence 0 from thereon out.

smoothing
discounting
linear interpolation

Therefore, we need to modify the MLE probability computation to assign some probability mass to new, unseen n-grams. This is called **smoothing**. There are numerous approaches to smoothing, i.e., **discounting** (reducing the actual observed frequency) or **linear interpolation** (computing several n-grams and combining their weighted evidence). There is a whole paper on the various methods (Chen and Goodman, 1996), and it is still an area of active research.

Laplace-smoothing

The easiest smoothing, however, is **Laplace-smoothing**, also known as "add-1 smooth-

ing". We simply assume that the count of any trigram, seen or unseen, is 1, plus whatever evidence we have in the corpus. It is the easiest and fastest smoothing to implement, and while it lacks some desirable theoretical properties, it deals efficiently with new words. So, for a trigram that contains an unseen word x, our probability estimate becomes:

$$P(w|u,x) = \frac{c(u,x,w)+1}{c(u,x)+1} = \frac{0+1}{\sum_z c(u,x,z)+1}$$

Here, the resulting probability is 1, which seems a bit high for an unseen trigram. We can reduce that effect by choosing a smaller smoothing factor, say, 0.001, i.e., pretending we have seen each word only a fraction of the time. As a result, unseen trigrams will have a much smaller probability.

Putting everything above together in code, we can estimate probabilities for any new sentence. Smoothing can be efficiently implemented by using the `defaultdict` data structure, which returns a predefined value for any unseen key. By storing the trigrams in a nested dictionary, we can easily sum over them to compute the bigram counts:

```
from collections import defaultdict
import numpy as np
import nltk

smoothing = 0.001
counts = defaultdict(lambda: defaultdict(lambda: smoothing))

for sentence in corpus:
    tokens = ['*', '*'] + sentence + ['STOP']
    for u, v, w in nltk.ngrams(tokens, 3):
        counts[(u, v)][w] += 1

def logP(u, v, w):
    return np.log(counts[(u, v)][w]) - np.log(sum(counts[(u, v)].
   values()))

def sentence_logP(S):
    tokens = ['*', '*'] + S + ['STOP']
    return sum([logP(u, v, w) for u, v, w in nltk.ngrams(tokens, 3)])
```

CODE 1. A simple trigram LM.

This code can be easily extended to deal with higher-order n-grams.

6. Generation

generate text

Typically, Markov chain language models are used to predict future states, but we can also use it to **generate text** (language models and Markov Chains are generative graphical models). In fact, Andrey Markov first used Markov Chains to predict the next letter in a novel. We will do something similar and use a Markov chain process to predict the next word, i.e., to generate sentences. This technique has become a bit infamous, since it was used by some people to generate real-looking, but meaningless scientific papers (that got accepted!), and is still used by spammers, in order to generate their emails.

We can use the counts from our language model to compute a probability distribution over the possible next words, and then sample from that distribution according to the probability (i.e., if we sampled long enough from the distribution of a word $P(w|u, v)$, we would get the true distribution over the trigrams).

```
import numpy as np

def sample_next_word(u, v):
    keys, values = zip(*counts[(u, v)].items())
    values = np.array(values)
    values /= values.sum()
    return keys[np.argmax(np.random.multinomial(1, values))]

def generate():
    result = ['*', '*']
    previous_word = None
    next_word = sample_next_word(result[-2], result[-1])
    result.append(next_word)
    while next_word != 'STOP':
        next_word = sample_next_word(result[-2], result[-1])
        result.append(next_word)

    return ' '.join(result[2:-1])
```

CODE 2. A simple trigram sentence generator.

This generates simple sentences. A trigram model is not very powerful, and in order to generate real-sounding sentences, we typically want a higher order, which then also means a much larger corpus to fit the model on.

Bibliography

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Stanley F. Chen and Joshua Goodman. 1996. An Empirical Study of Smoothing Techniques for Language Modeling. In *34th Annual Meeting of the Association for Computational Linguistics*.

Fred Jelinek and Robert Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings Workshop Pattern Recognition in Practice*, pages 381–397.

Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.