LIN 528 Final Project Report

Language Models: n-grams and PCFGs

Derek Andersen and Joanne Chau

Fall 2020

1 Objective

The objective of this project was to train, evaluate, and compare the efficiency of two different statistical language models: a trigram model and a probabilistic context-free grammar (PCFG). The underlying structure of these two types of models varies quite a bit, but they both hope to accomplish the same thing: to act as a representation (more accurately, an approximation) of a language (limited to the scope of a given corpus), and predict the likelihood of occurrence of some new unattested language data.

In Section 2, we will outline the techniques we used for training our models. In Section 3 and 4, we will outline the training / testing for the trigram model and PCFG model respectively. In section 5, we will discuss the results of the models and the evaluation metrics we used.

1.1 Trigram model

In the case of an n-gram model, its purpose is to approximate the probability $P(w|h)$, or the probability of a word w given some history h, where h is a number of previous words equal to $n-1$. With our trigram model, $h = 2$, so for a word being evaluated, its two-word history will be considered. It's important to note that the motivation behind the n-gram model is to settle for the *approximation* of the probability of some word given its recent history, since it isn't possible to calculate the exact probability of a word given its entire history. (Jurafsky and Martin 2019)

1.2 PCFG model

In the case of a PCFG model, its purpose is to determine the most likely parse of a sentence, and output the probability of a sentence's most likely parse. It is trained on a training set of fully parsed trees, from which the model takes constructions it sees and constructs a grammar of Chomsky normal form (CNF) rules:

- Start symbol (S)
- Non-terminals (NP, VP, etc.)
- Terminals (vocabulary items)

2 Techniques and Tools

Both of our models were implemented in Python via the `nltk` package. This package provides functions for extracting *n*-grams in the case of the trigram model, and implementing a CKY parser and constructing a PCFG grammar in the case of the PCFG model.

The Wall Street Journal corpus was chosen as the corpus for training and testing our models. This corpus contains 2,499 stories from a three-year period resulting in 38,785 English sentences. (Marcus 1999) For training and testing the models, we used an 80/20 split of the dataset respectively.

3 Trigram Model

The trigram model was trained according to the following process:

1. Construct an empty double-nested dictionary
2. Iterate over the data files (each containing multiple sentences)
3. Iterate over each sentence per data file
4. For each sentence, extract its trigrams, padded on the left and right if at the beginning or end of the sentence
5. For each trigram, add the first 2 words (because $h = 2$) as a dictionary key, and add the third word as a value in the sub-dictionary for that key. Its value is an integer (count), incremented whenever this particular trigram is attested
6. Transform the counts for each $(w3|w1w2)$ pair into probabilities taking into consideration the total number of $w3$s attested for a given bigram history

3.1 Smoothing

Since the model will be used to evaluate a test set (inevitably with words outside the scope of its knowledge), we have to apply smoothing to the model. This will assign a very low probability to words which it hasn't yet seen. To do this, when instantiating our model's double-nested dictionary, we apply a `lambda` value of 0.01.

3.2 Using the model

Once the model is trained, we can look at example probabilities for specific trigram constructions. For example, if we wanted to look at all of the attested $w3$s for a given bigram history, 'the profits', we can access them via our model:

```
print(model[('the', 'profits')])
> {'and': 0.22160970231532526, 'from': 0.22160970231532526,
'in': 0.11135611907386991, 'generated.': 0.11135611907386991,
'with': 0.11135611907386991, 'began': 0.11135611907386991,
'businessmen': 0.11135611907386991}
```

From this, we can see all of the words that follow the bigram 'the profits' along with their probability distribution. Importantly, all of these will total 1, since this output is a dictionary containing *all* of the attested $w3$s for the history we queried.

We can also use our model to check the probability of a sentence starting with 'The' according to our training data. In this case, `None` values represent padding.

```
print(model[None, None]["The"])
> 0.16637225876894499
```

4 PCFG Model

Similar to the trigram model, the PCFG model required a iteration over multiple data files. The building and training process of the model is documented below:

1. Construct an empty list to store all PCFG rules that would be created by the model
2. Iterate over data files in the `.mrg` format. In the case of the code written, we formed a code that added all of the Wall Street Journal files into the treebank corpora provided by the `nltk` package. This allowed easier access into the files.

3. As each data file had more than one parsed tree, iterate over each parsed tree of the data file.

4. Ensure that all trees are changed into binary branching.

5. Add every produced rule into the PCFG rule list.

6. After all production rules are retrieved from the given data, set the nonterminal node to start at `‘S’`.

7. Induce the grammar given the `nltk induce_pcfg()` function. When the grammar rules are formed, return the PCFG for future use.

8. Initialize a CKY parser via `nltk`’s `ViterbiParser()` function.

4.1 Accounting for unattested vocabulary items

Unlike the trigram model, the PCFG model’s attempt at smoothing is mainly to account for unattested vocabulary for incoming test data. To account for this, for every sentence from the data files, we used `grammar.check_coverage()` to first check all words in the sentence before it parses. In `nltk`, if a word is not in the grammar, it would raise a `ValueError: Grammar does not cover some of the input words: word`. As a hack and to keep track of all skipped sentences, we have a `skipped_sentences` counter and utilized `try, except` and `continue` to skip any errors that may rise due to unattested vocabulary. An example of the `ValueError` is shown below.

```
grammar.check_coverage(‘test’)
> ValueError: Grammar does not cover some of the input
> words: ‘"Poor \‘s", \‘contracts.\ ’’.)
```

Imagine a grammar that has only the words ‘a’, ‘man’, ‘eats’ and we would like to evaluate the sentence ‘a dog eats’. An example of how the code works and a quick sample counter of skipped words is shown below. The `try, except` allows us to catch the `ValueError` that may be raised in cases where vocabulary is unattested given a specific PCFG.

```
skipped_words = 0
try:
    grammar.check_coverage([‘a’, ‘dog’, ‘eats’])
except:
    skipped_words += 1
print(‘Number of skipped words:’, skipped_words)
```

```
> Number of skipped words: 1
```

This is particularly important for the testing of the PCFG model as some files in the testing data have multiple trees in each and not every word in the English vocabulary would be available in the PCFG rules.

4.2 CKY parser

The grammar is fed into the CKY parser and a syntactic parse of the sentence along with the probability of the parse is provided. The `nltk` package has a built-in function to generate this parser for us. After generation of the CKY parser, it iterates over any file of data and extracts the trees in the data. For every tree, it extracts just the leaves from the sentence, so it produces a list of words for each sentence. The list is then fed into the `PCFG_grammar.check_coverage()` and if all words are attested for in the grammar, it parses the sentence and prints the parse and probability of the parse.

```
parser = ViterbiParser(PCFG_grammar)
sentences = treebank.parsed_sents('wsj_1964.mrg')
skipped_sentences = 0
for sentence in sentences:
    sentence = sentence.leaves()
    try:
        PCFG_grammar.check_coverage(sentence)
        for parse in parser.parse(sentence):
            print(parse)
    except:
        skipped_sentences += 1
        continue
print('Total skipped sentences:', skipped_sentences)
```

It also keeps track of all skipped sentences, in the case one would like to average how many sentences are skipped in a specific dataset.

5 Results

In order to compare the PCFG model to the trigram model, we chose perplexity as our metric for evaluation for the models. Higher perplexity for a model correlates with worse performance. In the case of language models, this means that a lower perplexity corresponds to a more accurate approximation of the target language.

5.1 Trigram model perplexity

The perplexity, PP, for an n-gram model is defined as the inverse probability of the test set, normalized by the number of words. For a trigram model, this can be represented as:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(s_i|w_{i-1}, w_{i-2})}}$$

where the probability for one word on in the denominator depends on its preceding two words, and N is the number of words in the test set. (Jurafsky and Martin 2019)

Since our test set is a collection of sentences, to calculate the perplexity of the model on the test set, we calculated the perplexity of each sentence via the model, and then calculated the average perplexity. The perplexity of our model on the test set was 55.434798009501684.

5.2 PCFG model perplexity

In the case of the PCFG model, because the probabilities are output with each parse, we extract the probability from each calculated parse. The formula for calculating perplexity for the PCFG model is the same as the trigram model, with the difference being that the probability is per tree parse as opposed to per trigram.

The probability of each tree parse is provided by the CKY parser, which is then extracted from the parsed tree by a simple string extraction and saved into a list of all probabilities.

5.2.1 Issues

Assuming we used the full training set, our PCFG model would have over 94,000 rules. Since the CKY parser tests the sentence on every rule, it takes a very long time to parse. It was not possible to get the possibility of all the tree parses in the remaining 20% of the WSJ treebank. In our case, it was not feasible to train the model on the entire training set; the computational requirements were too high. For this reason, we decided to go with a training set of 60 and a test set of 2. The perplexity of the model with this configuration was calculated to be about 5e+45. Interestingly, when using more test files (e.g. +2), the perplexity seemed to get worse. Further tests would need to be performed in order to gauge the PCFG model's performance more accurately compared to the trigram model. Specifically, the full 80/20 split for train/test would be helpful.

6 Conclusion

The perplexity shows us that the PCFG model is worse compared to the trigram model. This makes sense, as there is a very large amount of rules that the parser must consider in order to find the most likely parse for each sentence. The trigram model seems to perform quite well, and is able to account for unseen sentences in the test set with a fairly low perplexity.

For a better understanding of how well the PCFG model could potentially perform, further work is necessary. In terms of minimizing the PCFG model's perplexity, more training/test data would be needed for a more accurate representation. We would also need to account for unattested vocabulary. In the current PCFG model code, all sentences with unattested vocabulary words are ignored, whereas smoothing took care of this issue for the trigram model. For a better comparison of the two, we would have to investigate how well the PCFG model handles such sentences. Some methods that may prove useful are utilizing wildcard terminals. These would allow unattested words to be treated as a separate class of node, ignoring its original type. But, there may also be downsides to using wildcards as the part of speech of a word would be lost and thus the syntactic structure as well.

References

Jurafsky, Daniel and James H. Martin (2019). *Speech and Language Processing.*

Marcus, Mitchell P. et al. (1999). *Treebank-3 LDC99T42.* Philadelphia: Linguistic Data Consortium. URL: `https://catalog.ldc.upenn.edu/LDC99T42` (visited on 12/15/2020).