Language Models: *n*-grams and PCFGs

Final Project for LIN 538: Statistics for Linguists (Fall 2020)

Derek Andersen and Joanne Chau
Stony Brook University

In this project we will train and compare the results of two different types of language models: a **trigram** model and a **PCFG** (probabilistic context-free grammar). In both cases, we will use **perplexity** as a metric for evaluation, and we will compare the two models according to their perplexity values using the Wall Street Journal corpus.

Trigram Model

A trigram model predicts the probability, *p*, of a word, *w*, occurring next in a sentence, given a history, *h*, of *n – 1* words (in this case, *h* = *2*). So our trigram model will be able to predict *p(w | h)*, or more specifically, *p(w3 | w1 w2)*.

First, we import `nltk` and other necessary libaries. Then we define the functions we'll need: `trigram_model` to train our model, `generate_sentence` to demo sentence generation with our model, and `perplexity` to calculate the perplexity of a model according to a test sentence.

In [1]:

```
import nltk
from nltk import trigrams
from collections import Counter, defaultdict
import random
from pathlib import Path
import os

# Path to wsj corpus
corpus_path = Path("C:/Users/Derek/Documents/wsj_corpus")

def trigram_model(corpus_path):
    """Builds a trigram model trained on a training corpus."""
    # Smoothing of 0.01 to handle unattested words in test data
    model = defaultdict(lambda: defaultdict(lambda: 0.01))
    # Training set of 80% of the Wall Street Journal corpus (first 1963 fi
les)
    for file in os.listdir(corpus_path)[:1964]:
        with open(corpus_path / file, 'r') as current:
            sents = current.readlines()
            for sentence in sents:
                if ('.START' in sentence) or (sentence == '\n'):
                    continue
                else:
                    sentence = sentence.split()
                for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_l
eft=True):
                    model[(w1, w2)][w3] += 1
```

```
    # Transform the counts into probabilities
    for w1_w2 in model:
        total_count = float(sum(model[w1_w2].values()))
        for w3 in model[w1_w2]:
            model[w1_w2][w3] /= total_count

    return model

def generate_sentence(model):
    """Generates a sentence according to a trigram model."""
    text = [None, None]
    sentence_finished = False

    while not sentence_finished:
        r = random.random()
        accumulator = .0
        for word in model[tuple(text[-2:])].keys():
            accumulator += model[tuple(text[-2:])][word]
            if accumulator >= r:
                text.append(word)
                break
        if text[-2:] == [None, None]:
            sentence_finished = True
    print(' '.join([t for t in text if t]))

def perplexity(test_sent, model):
    """Computes the perplexity of a trigram model on a test sentence."""
    test_sent = test_sent.split()
    perplexity = 1
    N = 0
    for w1, w2, w3 in trigrams(test_sent, pad_right=True, pad_left=True):
        N += 1
        perplexity = perplexity * (1/model[(w1, w2)][w3])
    perplexity = pow(perplexity, 1/float(N))
    return perplexity
```

Training the model

First, we train our model using 80% of the Wall Street Journal corpus and save it as `model`.

In [2]:

```
# Create a trigram model according to wsj corpus
model = trigram_model(corpus_path)
```

Testing the model

Now, as a test, let's look at the probability that a sentence will start with 'The' according to our model.

In [3]:

```
# Print the probability that a sentence will start with 'The'
print(model[None, None]["The"])
```

```
0.16637225876894499
```

To illustrate how the model will behave when it's asked to predict a word unattested in the training set, let's run this same test with a made-up word following an *attested* history: 'political concerns'. We can see that the model predicts 0.01, which is treated as 0. This 0.01 is a result of the smoothing we applied when training our model, so that it can handle unattested words in testing.

In [4]:

```
print(model['political', 'concerns']['madeupword'])
```

```
0.01
```

Now we can use our model to generate a new sentence, to demo its nativeness. Not bad!

In [5]:

```
generate_sentence(model)
```

```
Consider, for instance, is forecasting growth in the oil-patc
h state of Alagoas.
```

Perplexity

The metric we're using for model comparison is **perplexity**. Perplexity is a measure of how good, or in the case of language, how native a model is. A lower perplexity is a correlate of higher nativeness, and ideally a model trained on English data should be able to recognize English sentences well, and thus score lower on the perplexity spectrum.

The perplexity, *PP*, of a sentence, *s*, can be calculated with the following:

PP(s) = p(w1, ..., wn)^(–1/n)

We will use a test set of 20% of the Wall Street Journal corpus to evaluate the perplexity of the model.

In [6]:

```
# Construct a test set of 20% of the Wall Street Journal corpus (files 196
4 - 2454)
testset = []
for file in os.listdir(corpus_path)[1964:2455]:
    with open(corpus_path / file, 'r') as current:
        sents = current.readlines()
        for sentence in sents:
            if ('.START' in sentence) or (sentence == '\n'):
                continue
            else:
                testset.append(sentence)

# Calculate the perplexity of the model with the entire test set
PP = 0
perplexities = []
i = 0
for sentence in testset:
    p = perplexity(sentence, model)
    # ignore infinity cases
    if p == float("inf"):
```

```
        continue
    i += 1
    PP += p
# average of perplexities
PP = PP / i


print('Model perplexity on test set:', PP)
```

```
Model perplexity on test set: 55.434798009501684
```

For reference, let's test the same model's perplexity instead using a Jabberwocky sentence (a sentence with several made-up words). We can see that the perplexity is much higher, as we get a value of 100. This is what we would expect considering the lower perplexity value with the English test set of unattested sentences we used earlier.

In [7]:

```
jabberwocky = "Twas brilig, and the slithy toves Did gyre and gimble in th
e wabe; All mimsy were the borogoves, And the mome raths outgrabe."
print(perplexity(jabberwocky, model))
```

```
100.00000000000001
```

PCFG Model

A PCFG model takes a set of training sentences and examines all context free grammar rules. It then saves it as a full list of all productions.

In order to have an adequate number of trees to train and test the model, we saved all `.mrg` files from the Wall Street Journal treebank in the `nltk_data` directory for easier access, since there are already built-in functions in `nltk` for dealing with treebanks.

Instead of the 80/20 split we did with the trigram model, the PCFG model will run on a small portion of the training set (60 files) as it takes too long for the full training set to get all the PCFG rules. We will save our model in `grammar`.

In [8]:

```
import nltk
from nltk import Tree, PCFG, treetransforms, induce_pcfg, Nonterminal
from nltk.corpus import treebank
from nltk.parse import pchart, ViterbiParser

def make_PCFG_grammar():

  '''
  Trains a PCFG grammar on the WSJ treebank.
  '''

  # Save a list of all produced PCFG rules given the test data
  PCFG_rules = []

  # We'll utilize the Wall Street Journal corpus
  for item in treebank.fileids()[:61]:
```

```
        # We want to first get rid of all non-binary branchings of the tree
        for tree in treebank.parsed_sents(item):
            tree.collapse_unary(collapsePOS = False)
            tree.chomsky_normal_form(horzMarkov = 2)
            PCFG_rules += tree.productions()

    # Induce the PCFG grammar
    S = Nonterminal('S')
    PCFG_grammar = induce_pcfg(S, PCFG_rules)

    return PCFG_grammar


# save our model
grammar = make_PCFG_grammar()
```

Test set

With our PCFG grammar, we can use the built-in `ViterbiParser` function available in `nltk` to build a CKY parser applicable to our PCFG rules. Using this, we can run test trees through it to find their most probable parses.

In [9]:

```
def CKY_parser(PCFG_grammar):

    '''
    Given the PCFG grammar, we use the built in CKY praser function
    to get a sentence's most probable parse
    '''

    # Utilize the ViertabiParser given the PCFG grammar induction rules
    parser = ViterbiParser(PCFG_grammar)

    # Sample file parse for reference
    sentences = treebank.parsed_sents('wsj_0001.mrg')

    skipped_sentences = 0

    # A for loop to get all possible trees within the files
    for sentence in sentences:
        sentence = sentence.leaves()

        # To speed up the code, we'll check with the grammar first
        # And ensure that all words are accounted for
        # If it is not accounted for, skip the sentence
        # And increment skipped_sentences
        try:
            PCFG_grammar.check_coverage(sentence)

            # Print the final parse of the sentence
            for parse in parser.parse(sentence):
                print(parse)
        except:
            skipped_sentences += 1
            continue

    print("Total skipped sentences:", skipped_sentences)
```

```
# demo the parser
CKY_parser(grammar)
```

```
(S
  (NP-SBJ-27
    (NP (NNP Pierre) (NNP Vinken))
    (NP-SBJ-27|<,-ADJP>
      (, ,)
      (NP-SBJ-27|<ADJP-,>
        (ADJP (NP (CD 61) (NNS years)) (JJ old))
        (, ,)))))
  (S|<VP-.>
    (VP
      (MD will)
      (VP
        (VB join)
        (VP|<NP-PP-LOC>
          (NP (DT the) (NN board))
          (VP|<PP-LOC-NP-TMP>
            (PP-LOC
              (IN as)
              (NP
                (DT a)
                (NP|<JJ-NN>
                  (JJ nonexecutive)
                  (NN director))))
            (NP-TMP (NNP Nov.) (CD 29))))))
    (. .))) (p=2.39483e-50)
(S
  (NP-SBJ (NNP Mr.) (NNP Vinken))
  (S|<VP-.>
    (VP
      (VBZ is)
      (NP-PRD
        (NP (NN chairman))
        (PP
          (IN of)
          (NP
            (NP (NNP Elsevier) (NNP N.V.))
            (NP|<,-NP>
              (, ,)
              (NP
                (DT the)
                (NP|<NNP-VBG>
                  (NNP Dutch)
                  (NP|<VBG-NN> (VBG publishing) (NN grou
p))))))))))
    (. .))) (p=2.27472e-37)
Total skipped sentences: 0
```

We can see two output parses above, along with their probabilities (*p* values). These two sentences are present in the model's training set and thus all of their vocabulary is attested.

Perplexity

In order to calculate the perplexity of the PCFG model, run all testing set and then save all probabilities for the best parse.

We will save the probabilities into a list of probabilities for reference. Then we utilize that to calculate the perplexity of the model.

Issues with testing

Unfortunately, with our training set, the amount of time it takes to test on the model increases exponentially and it is not feasible to run all of the tests. In our case, we used 2 files from the test set.

In [10]:

```
import re

def all_parse_probabilities(PCFG_grammar):

  '''
  Given the PCFG grammar, we utilize the CKY parser to get
  the test set's parse probabilties.
  '''

  parser = ViterbiParser(PCFG_grammar)
  # Make a list to save all extracted parse probabilities
  all_p = []

  # 2 test files
  for item in treebank.fileids()[1964:1966]:
      trees = treebank.parsed_sents(item)
      for tree in trees:
        tree = tree.leaves()
        try:
          PCFG_grammar.check_coverage(tree)

          # Change the parsed tree from a tree to a string
          # Use regular expression to find the correct chunk
          # Delete the last character and then append to the all_p list
          for parse in parser.parse(tree):
            parse_string = str(parse)
            p = re.search(r"p=([^/]+)", parse_string).group(1)
            p = p[:-1]
            all_p.append(float(p))
        except:
          continue

  return all_p


# get the probabilities of our test trees
all_p = all_parse_probabilities(grammar)
```

In [11]:

```
def perplexity(all_p):

    '''
    Given a list of the probabilities of all parses from the testing set,
    this calculates the perplexity of the model.
```

```
    '''

    perplexity = 1

    # N is the total number of probabilities
    N = float(len(all_p))
    for p in all_p:
        # ignore infinity cases
        if perplexity * (1/p) == float("inf"):
            continue
        perplexity = perplexity * (1/p)
    perplexity = pow(perplexity, 1/float(N))
    return perplexity


# calculate perplexity for our test set
print("Model perplexity on test set:", perplexity(all_p))
```

```
Model perplexity on test set: 5.614793240381743e+45
```

The model perplexity on our small test set comes out to be about 5e+45. Unfortunately, this is not an accurate representation of the PCFG model's comparison to the trigram model, since the train/test sets had to be reduced. But from this evaluation, the PCFG model appears to perform worse than the trigram model, since its perplexity is much higher.

References

- https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-language-model-nlp-python-code/
- https://nlpforhackers.io/language-models/
- https://www.cs.bgu.ac.il/~elhadad/nlp16/NLTK-PCFG.html
- https://www.asc.ohio-state.edu/demarneffe.1/LING5050/material/structured.html