AdvNLP Lab 4: Language Modelling

Julie Weeds

February 16, 2024

1 Getting started

During the next couple of labs this term, you are going to be exploring language models and applying them to the Microsoft Research Sentence Completion Challenge (Zweig and Burges, 2011). Today, you are going to start by building an **n-gram language model** based on the training corpus for the Challenge. All of the data for the Sentence Completion Challenge is contained in this `Lab4resources` directory in the sub-directory titled `sentence-completion`. You may need to unzip `training_data.tgz` so that the `Holmes_Training_Data` sub-directory is visible.

You can use the following code to list the directory and select 50% of the files (holding back the other 50% so that you can evaluate your language model). This code is also contained in the notebook in the resources directory.

```
import os,random,math
TRAINING_DIR="lab3resources/sentence-completion/Holmes_Training_Data" #this needs to be the parent
    directory for the training corpus

def get_training_testing(training_dir=TRAINING_DIR,split=0.5):

    filenames=os.listdir(training_dir)
    n=len(filenames)
    print("There are {} files in the training directory: {}".format(n,training_dir))
    #random.seed(53) #if you want the same random split every time
    random.shuffle(filenames)
    index=int(n*split)
    return(filenames[:index],filenames[index:])

trainingfiles,heldoutfiles=get_training_testing()
```

2 A Unigram Model

The code below implements a simple unigram model. The class stores the unigram probabliity distribution and provide methods for training and lookup.

```
from nltk import word_tokenize as tokenize
import operator

class language_model():

    def __init__(self,trainingdir=TRAINING_DIR,files=[]):
        self.training_dir=trainingdir
        self.files=files

        self.train()

    def train(self):
        self.unigram={}
        self._processfiles()
        self._convert_to_probs()

    def _processline(self,line):
        tokens=["_START"]+tokenize(line)+["_END"]
        for token in tokens:
            self.unigram[token]=self.unigram.get(token,0)+1


    def _processfiles(self):
```

```
        for afile in self.files:
            print("Processing {}".format(afile))
            try:
                with open(os.path.join(self.training_dir,afile)) as instream:
                    for line in instream:
                        line=line.rstrip()
                        if len(line)>0:
                            self._processline(line)
            except UnicodeDecodeError:
                print("UnicodeDecodeError processing {}: ignoring file".format(afile))


    def _convert_to_probs(self):

        self.unigram={k:v/sum(self.unigram.values()) for (k,v) in self.unigram.items()}

    def get_prob(self,token,method="unigram"):
        if method=="unigram":
            return self.unigram.get(token,0)
        else:
            print("Not implemented: {}".format(method))
            return 0
```

2.1 Training

You can train the model by initialising it with a list of text files. I suggest you work with a small portion of the files in trainingfiles (up to 5), at least whilst you are just testing functionality.

```
MAX_FILES=5
mylm=language_model(files=trainingfiles[:MAX_FILES])
```

Look through the code and you should see the following steps in the training process.

- Each of the subset of datafiles is processed in turn.
- In each datafile, each line is tokenized using `nltk.word_tokenize()`
- `_START` and `_END` tokens are added to the beginning and end of each line.
- A count is kept of how many times each token occurs. This is stored in a dictionary `self.unigram`
- Once all of the datafiles have been processed, the counts are converted into probabilities (by dividing all of them by the sum of all of the values in the dictionary).

Test your model by looking up the probabilities of some (known) words. At this stage, if the word was not seen in the training data, the probability returned should be zero (we will worry about smoothing later!)

2.2 Generation

Add some functionality to your class so that you can generate a string of highly probable words. You can do this by sorting the dictionary of unigram probabilities and then randomly choosing 1 of the top k words. Repeatedly choose words until a particular token is generated (e.g., '.') or a maximum length is exceeded.

As an extension, you could try to sample from the distribution. Assign each of the words a range of numbers - the size of which is proportional to its probability. You can do this by considering the cumulative probability distribution (iterate through the words in the distribution, adding the probability of the current word to the sum of all probabilities seen so far). Then you just need to pick a random number and select the word which has this number in its assigned range. Of course, there are library methods which can do this for you to — check out random.choices() at https://www.kite.com/python/answers/how-to-sample-a-random-number-from-a-probability-distribution-in-python

3 Adding Bigrams

3.1 Training

Extend your class so that, at training time, you also store bigram counts. I suggest that you use a dictionary of dictionaries for the bigram counts. The key for the outer dictionary should be the previous word (which may be a `_START` token) and the key for the inner dictionary should be the current word. The value is then how many times the current word has occurred after the previous word.

For example, if you see the sentence "When did the cat sit on the mat?", you should get the following dictionary:

```
bigram={_START:{When:1},
   When:{did:1},
   did:{the:1},
   the:{cat:1, mat: 1},
   cat:{sit:1},
   sit:{on:1},
   on:{the:1},
   mat:{?:1},}
```

You need to normalise each of the internal dictionaries to create probability distributions for $P(w_2|w_1)$. You do this by summing the values in each internal dictionary and then dividing each count by the sum of values for that dictionary.

3.2 Generation

Can you use your bigram probabilities to generate plausible sounding sentences? Initialise the current token as `_START` and then randomly choose one of the top k most probable words for the next token. Then use this as the current token to generate the next token and so on. Experiment with different values of k

4 Perplexity

Perplexity is a function of a model and some language data. If the observed language data is highly likely given the model then the perplexity will be low.

- Write a method or function that calculates the log probability (LP) of a corpus for a language model (e.g.,the unigram model and the bigram model).
- Compute the perplexity using the formula $p = e^{\frac{-LP}{N}}$, whre N is the number of words in the corpus

5 Dealing with Unseen Data

If you run your perplexity function on the `heldoutfiles` (rather than the `trainingfiles`), you will quickly run into the problem that certain unigrams and bigrams in the `heldoutfiles` were not present in the `trainingfiles`. This means that their probability will be zero (making the probability of the whole corpus to be zero).

5.1 Unknown words

In order to have some probability mass for unseen unigrams, we need to pretend that some were unseen in the training data. Therefore, we replace low frequency unigrams in the training data with the `_UNK` token. One way of implementing this is:

- Before converting counts to probabilities, iterate over the unigram probability distribution and find words which have occurred less frequently than a threshold (e.g., 2). Remove this key from the unigram distribution `del self.unigram[key]` and add the count to the entry for the `_UNK` token.
- Iterate over the bigram distribution.
  - For each inner dictionary, use the same process as for the unigram distribution.
  - If any outer keys are no longer valid unigram keys, delete that outer key and add the whole contents of the dictionary associated with it to an entry in the bigram distribution for `_UNK`

- When looking up a probability or a distribution for a word, if it is not found then use the probability or distribution for `_UNK` instead

You should now be able to compute the perplexity of the unigram model for the `heldoutfiles`. What do you notice about the perplexity values for this model?

5.2 Discounting for unseen combinations

Having dealt with unseen unigrams, we need to also deal with unseen bigrams i.e., bigrams where both words have been seen but not with each other.

We could use Laplacian (add-k) smoothing but this is both computationally inefficient and it tends to assign too much probability mass to unseen events. A more common approach is to use some form of **discounting** (or **backoff**).

First, we apply a discount to each bigram count. There are variations where the discount applied depends on the word or the frequency of the word or the frequency of the bigram, but applying a fixed or absolute discount of 0.75 to each bigram count is the simplest method and works very well. It is also necessary to store the total amount that has been discounted for each unigram (this will depend on how many words have been seen following it). You can store this in the bigram distribution (under `self.bigram[unigram]["_DISCOUNT"]`. This means the total counts will not have changed and therefore you have effectively reserved some probability mass for unseen bigrams.

Then, to estimate the smoothed probability of $P_e(w_2|w_1)$:

$P_e(w_2|w_1) = P_o(w_2|w_1) + P_o(\_\text{DISCOUNT } |w_1) \times P_o(w_2)$

You should now be able to compute the perplexity of the bigram model with respect to the `heldoutfiles`.

6 Extensions

Choose your own extension(s) from those listed below.

1. Implement the Kneser-Ney backoff method and compare this to the absolute discounting method introduced above.

2. Investigate how perplexity is affected by one or more of the following

   - amount of training data
   - amount of testing data
   - the threshold frequency for unknown words
   - the size of the discount applied to observed bigrams

3. Extend your model to trigrams.

References

Geoffrey Zweig and Christopher Burges. 2011. The microsoft research sentence completion challenge. Technical report, Microsoft Research, December.