NLP Report 1: Language Modeling

Lylla Younes and Minae Kwon

September 2015

1 Pre-Processing

For pre-processing, we decided to use our own parser instead of using a pre-existing platform such as NLTK. We did this for several reasons. First of all, when we attempted to pre-process with NLTK and another platform called Spacy, we noticed that some of the words were glued together. For example "I can" would be parsed to "Ican." This did not occur terribly frequently, but we still felt that it would somewhat affect our results.

Secondly, we felt that by using our own parser, we would have greater control over what we would consider to be "words." For example, we did not want to separate contractions from their respective words, like many pre-existing platforms do. For example, we did not want the word "didn't" to be parsed to "did" and "n't." Ultimately, we parsed out six different types of punctuation from the text: periods, question marks, exclamation points, colons, semicolons, and commas. We also appended a sentence marker to the end of each sentence ('<s>'). A snippet of the pre-processing code is shows below:

```
for i in range(len(corpus)):
       if(len(corpus[i]) >= 1):
           if((corpus[i][-1] == '.') | (corpus[i][-1] == '?') |
               (corpus[i][-1] == '!')):
               temp = corpus[i][-1]
               new_corpus.append(corpus[i][0:-1])
               new_corpus.append(temp)
               new_corpus.append('<s>')
           elif((corpus[i][-1] == ',') | (corpus[i][-1] == ';') |
               (corpus[i][-1] == ':')):
               temp = corpus[i][-1]
               new_corpus.append(corpus[i][0:-1])
               new_corpus.append(temp)
           else:
               new_corpus.append(corpus[i])
return new_corpus
```

2 Random Sentence Generation

a Unigram Generator

The unigram model was quite simple. We reasoned that if we place all the words in the corpus into one large array, and randomly select from that array, we would automatically be accounting for the probability of the any given word in the entire corpus. For example, if the word “tower” appears 4 times in the corpus, and we randomly select from the array of all the words in the corpus, the probability of selecting “tower” is 4 over the length of the corpus. If c denotes the count of a word w, then:

$$Pr(w) = \frac{c_w}{len(corpus}$$

A few of the unigram model’s randomly generated sentences are shown below:

from crime corpus:

Dmitri surprise as own this him laughed said grey your the flat stealing of always we , ”your

from history corpus:

George result for goods real thousands

b Bigram Generator

The bigram sentence generator algorithm was implemented using a dictionary. We chose to use this data structure because it is such an efficient way to keep track of not only all the individual bigrams, but also all the bigrams of a given word. To do this, we used nested dictionaries. The key of the outer dictionary is just a word, say “you.” The value of this key is another dictionary, maintaining all the words that come after some you in the text and the count of that pair’s occurrence. Below we show a small section of our nested dictionary:

```
{ ,carried’: {’a’: 1, ’them’: 2, ’her’: 2, ’away’: 2, ’up’: 1, ’it’:
    2, ’everything’: 1, ’the’: 1}, ’getting’: {’a’: 1, ’on’: 2, ’up’:
    1, ’this’: 1, ’some’: 1, ’into’: 2, ’M’: 1, ’it’: 1, ’quite’: 1,
    ’to’: 3, ’the’: 2, ’out’: 1, ’you’: 1, ’Jacques’: 1}, }
```

In the algorithm, we randomly selected words in according to a probability distribution until we hit a sentence marker. We first randomly chose a start word from all the capitalized words in the corpus. This is because except for a few cases, capitalized words will be at the start of a sentence. We then generated a random number and created a probability distribution for all the words that could follow. The random number was generated from 0 to the maximum value of the words that follow. From the probability distribution, we picked the word that corresponded with the random number we generated. We applied the same algorithm to pick the words following the start word.

A snippet of our code is shown below:

```
#generating random number
    rand = random.randint(0,max(b_dict[word].values()))
    print "rand number:" + str(rand)

#choosing next word
    if (rand <= min(values)):
        next_word = b_dict[word].keys()[0]
    else:
        for i in range(len(b_dict[word])-1):
            if ((b_dict[word].values()[i] < rand) and (rand <=
                 b_dict[word].values()[i+1])):
                next_word = b_dict[word].keys()[i+1]

    print "next word:" + next_word
    return next_word
```

```
#this function generates random sentences from a bigram model
def b_sent_gen(books):
    b_dict = write_bigrams(books)
    sentence = ''
    start_word = bigram_start_word(b_dict)
    next_word = bigram_next_word(start_word, b_dict)
    sentence = start_word + " " + next_word
    while(next_word != '<s>'):
        next_word = bigram_next_word(next_word, b_dict)
        if next_word == '<s>':
            print sentence
        else:
            sentence = sentence + " " + next_word
    print sentence
```

A few of the bigram model's randomly generated sentences are shown below:

```
{'<unk>': 0.0003250270855904659}, 'Theodoric': {'would':
    0.00021673168617251842, 'followed': 0.00021673168617251842,
    'returned': 0.00021673168617251842, 'cried':
    0.00021673168617251842, 'to': 0.00021673168617251842, 'started':
```

```
    0.00021673168617251842, 'some': 0.00021673168617251842,}
```

from crime corpus:

The knocking at all," murmured softly moved to the faint , and in to blackmail me .

from children's corpus:

Mustapha replied the slope to happen , she could .

In comparison, we implemented the bigram random sentence generator without the probability distribution; we simply selected the next word based on the highest probability. This was problematic because we would get sentences that have repeated parts.

Some examples are shown below:

from history corpus:

Hiero , and the people , and the people , and

Ever since the people , and

3 Good Turing and Unknown Words

In order to account for unknown words, we took each word that occurred once in the training corpus and replaced it with the token '<unk>.'

To implement the Good Turing Smoothing on the training corpora, we first iterated through our bigram data structure (the dictionary of nested dictionaries). For all the bigrams with counts less than 5, we revised the counts for Good Turing smoothing using the formula below:

$c* = (c+1)\ \frac{N_c+1}{N_c}$

For all counts equal or greater to 5, we left the counts the same. We then iterated through the data structure again and for *all total* counts we computed the probability using the following formula:

Linguistically speaking, the total amount of what could be considered "words" is essentially infinite. Thus, when encountered with some training corpus, it is very likely that a given test corpus will contain words that do not occur in the training corpus, even if the sizes of the two corpora are the same. In other words, we can always expect to see new words. Furthermore, when performing certain NLP tasks such as machine translation, the accuracy of the task is significantly improved if it is able to account for unseen words.

It is essential, then, that no word's probability computes to zero. We use Good Turing Smoothing to compute the probabilities of the unseen words in a corpus. This smoothing is necessary for the accuracy of Parts 4 and 5. If we obtain accurate probabilities for bigrams containing unknown words, we are able to account for a much larger quantity of bigrams in our model. The perplexity, which is essentially a measurement of tightness of fit between two corpora, will be calculated with greater accuracy, and it is therefore reasonable to suggest that genre classification would also be more accurate.

Below we show a snippet of the code we used to implement Good Turing. We only show code for counts equal to one (the rest is in the source code in function 'trygrams.' We iterated through our large dictionary and modified bigram counts smaller than 5. After modifying the counts, we converted them to Good Turing probabilities using the formula above:

```
print "Modifying Small Counts With Good Turing"
    for key in bi_dict.keys():
        print bi_dict[key]
        for key2 in bi_dict[key].keys():
            count = bi_dict[key][key2]
            if (count < 5):
                if (count == 1):
                    Nc = Nc1
                    Nc_plus = Nc2
                    modified_count_b = (((count+1.0) * Nc_plus) / Nc)
                        *1000.0
                    unseen_bigrams = (V-(t_dict[key])) * (Nc1 / Nc0)
                    prob = ((modified_count_b/1000.0) / (t_dict[key] +
                        unseen_bigrams))
                    bi_dict[key][key2] = prob
```

4 Perplexity

The greatest challenge with calculating the perplexity of the training and test corpora was deciding what to do with words that appear in the test corpus but not in the training corpus. As stated previously, we first went through our corpus and changed all the words that occur once to the symbol '<unk>.' Our first few implementations yielded poor results, thus causing our genre classification function to be completely incorrect. Our initial mistake was to, when encountering a bigram with a first word in the training corpus and a second word *not* in the training corpus, to set the probability equal to zero. We soon realized this was a huge mistake. Out of curiosity, we created a variable to check how many probabilities we were arbitrarily defining to be zero, and the output was over 8,000 - obviously causing the problems we were experiencing in our genre classification.

On our final and most successful iteration, we created a series of conditions. We checked to see if a bigram in the test corpus existed in the training. If so, we used the already computed Good Turing probability. We then checked for cases where the first word in the training corpus but the second word is not. In such cases, we took the probability of the bigram in the training corpus with the same first word and the second word '<unk>. If this condition is not satisfied, we simply define the probability to be an arbitrarily small value. We reasoned that since the function will not enter this third case much, it will not significantly affect the accuracy of our perplexity calculation.

We used the formula below for our perplexity calculation:

For a test corpus W = w_1 w_2 ... w_N

PP (W) = $P(\frac{c_w}{len(corpus})^{-1/N}$

5 Genre Classification

Our final and most successful implementation of the genre classification algorithm using our perplexity calculations worked for the majority of the test books with a few exceptions.

Computed Perplexities for Genre Classification

Testing Book	Crime Corpus	History Corpus	Childrens Corpus
The Daffodil Mystery (Crime)	9.0638	12.366	10.430
The Moon Rock (Crime)	9.965	13.515	10.733
Bacon (History)	10.184	10.761	10.924
The Magic City (Childrens)	9.970	12.947	10.6414

As evident from table above, we experienced the most success when testing on the crime corpus. In the first two rows of the table, the test books (which were of the crime genre) corresponded to the lowest perplexity calculations in the training crime corpus. We experienced moderate success with the next two rows. With the history test book, for example, the lowest perplexity value was obtained from the training crime corpus, and the second lowest perplexity value was from the history corpus. Similarly, with the children's genre test book, the training children's corpus yielded the second-lowest perplexity - not the first.

What does this tell us? First of all, the implementation of our perplexity algorithm has flaws in it. We could do a better job of accounting for words that do not satisfy the first two conditions that were detailed in the perplexity algorithm above. It may be useful to consider why our algorithm worked better on some of the test books than others. It is probably not a coincidence that the crime test books had a 100 percent success rate. It is likely that the crime corpus has a more unique and less standardized vocabulary than the other corpora. In other words, this could mean that crime books have a more specialized vocabulary than other genres of books.
Furthermore, it is

6 Extension

For our extension, decided to compare the results we got using Good Turing smoothing with those we would obtain with a simple Add-One Smoothing model. We were interested in seeing whether the difference in genre classification would be significant. We implemented for both a unigram and a bigram model. Here we will discuss the bigram model because the unigram model yielded very poor results with seemingly arbitrary perplexity values. While the bigram add-one smoothing algorithm was still a poor predictor of genre, we obtained some slightly more interesting results.

For the unigram model, we computed the probability as follows:

$\mathrm{P}(w_x) = \frac{count(w_x)+1}{N}$

A snippet of our code from the unigram smoothing is shown below:

```
#Implement add-one smoothing for unigrams
def uni_add1(books):
    corpus = write_bag(books)
    N = float(len(corpus))
    #generate a dictionary of word types and their counts
    t_dict = type_dict(books)
    for key in t_dict.keys():
        #compute unigram probabilities, modifying the #counts in t_dict
        new_val = (t_dict[key] + 1.0) / N
        t_dict[key] = new_val
    return t_dict
```

A snipped of the resulting dictionary of probabilities is shows below:

```
{ '"Awake': 2.5848476232326104e-05, 'rickety': 2.5848476232326104e-05,
    "Doesn't": 2.5848476232326104e-05, 'eagerness':
    2.5848476232326104e-05, '"Two': 3.8772714348489155e-05,
    'daylight,"': 2.5848476232326104e-05, }
```

We now turn to the add-one smoothing for bigrams, computing probabilities as follows:

P$(w_n \mid w_{n-1}) = \frac{count(w_n|w_n-1)+1}{count(w_n-1)}$

Below we show a snippet of our code for this smoothing process:

```
#Implement add-one smoothing for bigrams
def bi_add1(books):
    corpus = write_bag(books)
    N = float(len(corpus))
    bi_dict = write_bigrams(books)
    t_dict = type_dict(books)
    for key in bi_dict.keys():
        for key2 in bi_dict[key].keys():
            new_val = (bi_dict[key][key2] + 1.0) / t_dict[key]
            bi_dict[key][key2] = new_val
    print bi_dict
```

And a snippet of the resulting probabilities:

Our results from the add-one bigram model are shown in the table below:

Computed Perplexities for Genre Classification With Add-One Bigram Model			
Testing Book	Crime Corpus	History Corpus	Childrens Corpus
The Daffodil Mystery (Crime)	19.986	26.348	22.550
The Moon Rock (Crime)	19.986	26.349	22.550
Bacon (History)	21.013	23.580	22.673
The Magic City (Childrens)	23.080	21.202	23.079

These results were not as accurate as the Good Turing Model, as we expected. For the history test book, the training history corpus actually yielded the *hightest* probability. Unlike with the unigram add-one smoothing model, however, the perplexities are closer together and thus the algorithm is more precise (as expected). The Magic City, the children's test book, was a moderate failure, with the training children's corpus yielding the middle perplexity value.

The most interesting part of this extension, perhaps, was the results we generated from the crime test book. Once again, we experienced a 100 percent success rate in predicting both crime test books. This could mean several things.

First of all, it can mean that the power of our Good-Turing perplexity test is not as great as we had hoped, and we owe more to the actual specifics of this genre of literature than anything else. Conversely this could tell us something about when and how add-one smoothing models are more accurate. With more standardized vocabularies, we will see greater success with Laplacian probability models. This is somewhat obvious however, and overall, we do not recommend the add-one smoothing methods for genre classification.

7 Work Division

The work was divided fairly evenly between the two of us. We took turns coding and thinking about the algorithms, and we did research separately, on our own time. In terms of report writing, we created a shared account on Share Latex and collaborated an equal amount, filling out each section and checking the other's work until we were finished. Overall, Minae probably put more work into the perplexity algorithm, and Lylla put more work into the report writing and explanation. It is unlikely that the total work time was significantly different between the two of us.

In future exploration with NLP, we decided that we would be interested in implementing some more complex smoothing methods such as the Kneser-Ney Smoothing and Whitten-Bell Smoothing, and comparing these techniques to the Good Turing Algorithm, evaluating the different resutls.