JRL336, QW79
CS 4740 Project 1

Final Report

1. Preprocessing

We first concatenate all the books in each category(folder) into one consolidated corporus using command line (resulting corpora are aggregated_children_train.txt, aggregated_crime_train.txt, and aggregated_history_train.txt). We then tokenize each corpus using Natural Language Toolkit (NLTK) 3.0. We keep and ignore all the special characters in this process. The tokenized list of words are stored in the “tokens” list object.

We then use “tokens” in the following random sentence generation.

2. Random Sentence Generation

We developed our random sentence generator using a bigram language model. We made a hashtable of all the bigrams in the corpus, which has a bigram tuple and its counts as the key and value pair (e.g. (“I”, “see”), 10).

```
#Create hashtable of bigrams and counts
def make_hashtable():
    hash_tokens = dict()
    for i in range(0, len(tokens) - 1):
        if ( hash_tokens.has_key( (tokens[i],
tokens[i+1])) ):
            hash_tokens[ (tokens[i], tokens[i+1])] +=
1
        else:
            hash_tokens[(tokens[i], tokens[i+1])] = 1
    return hash_tokens

#dicionary of all bigrams only with count >=1
bigrams_dict = make_hashtable()
```

We then use this bigrams dictionary to generate the next word. Our generate_next(given_word) method uses MLE bigrams probabilities as a basis. We loop through the bigrams dictionary to compute the number of bigrams that start with the given word and keep track of the different possible 2nd words along with their counts. We then use MLE to compute probabilities for the next word, based on the previous data. We then create a uniform distribution and random number generation based on these values to mimic “picking” a word based on their probabilities.

```
#Generate next word based on bigrams
def generate_next(given_word):
    temp_list = []
    for key,value in bigrams_dict.iteritems():
        if (given_word == key[0]):
            temp_list.append((key[1], value))

    total = 0
    for i in temp_list:
        total += i[1]

    rand = random.uniform(0, total)
    counter = 0.0
    for j in temp_list:
        if ( (rand <= counter + j[1] ) & (rand >
counter) ):
            return j[0]
        counter += j[1]
    return ''
```

If there is a given seed sentence, we call the generate_next(given_word) method on the last word of the sentence since that’s the only word that matters in our bigram language model. Otherwise, we use generate_seed() method that randomly selects a word from the corpus. We then have two last functions that generate random sentences based on either a given sentence and a given length or just simply a given length.

The results of our bigram language model on random sentence generator is as follows:
Without Seed:

- `enter , `` Your kindness , for this great hooks on a matter . Then ,`
- `the wind was laid myself , like ? '' Aladdin then are obliged to the mice`
- `once more meat , as you , and as we had evidently to command of Don`

With Seed:

- Given “when he started”
    - `when he started to find some suicidal pedantry , I must have asked Portia was made the world`

- Given “I was going”
  - `I was going back the Carrot , falling perpendicularly down before my way '' replied , by email`
- Given “I will say”
  - `I will say that it is a class of face , and peopled as if you 're not our`
  - `I will say that of his son-in-law , and in land a sheath-knife . And the sawdust and threw`

The sentences don’t really make any sense since this is a rudimentary bigram language model. We experiment with trigram language models in section 6 to see if it can generate random sentences that make more sense.

3. Good-Turing Smoothing

Our good turing implementation is in the goodTuring.py file. We attempted two methods to carry out good-turing smoothing.

In the first method, we create a smoothed bigrams dictionary by adding in counts for unseen bigrams (N0) and revising counts for N1 to N5 bigrams in the bigrams dictionary that we generated from each train corpus. We also create a smoothed unigrams dictionary that contains counts for unseen unigrams and revised counts for N1 to N5 unigrams. We do this in order to calculate perplexity in the next section.

To create a smoothed bigrams dictionary, we first assume that the number of unseen bigrams equals the number of seen once bigrams in the bigrams dictionary. We then apply the good turing formula to revise counts for N1 to N5 bigrams.

```
good_turing_count = i * count / prev_count
```

After that, we insert tuple (‘unk’, ‘unk’) as the key of unseen bigrams and the estimated N0 as the value into our bigrams dictionary. We thought that our model would work well because of its consistency.

```
#Modify bigrams_dict table with new counts from N_0 to
N_5
def good_turing_bigrams_dict_count():
    new_counts = revise_counts()
    for i in range (1,6):
        for bi in bigrams_dict:
            if (bigrams_dict.get(bi) == i):
                bigrams_dict[bi] = new_counts[i]
    bigrams_dict[('unk', 'unk')] = new_counts[0]
```

```
        return bigrams_dict
    good_turing_bigrams_dict_count()
```

Now we have a revised bigrams dictionary with smoothed bigram counts. We applied the same procedure to the unigrams in the corpus.

```
    def smooth_unigram_table():
        new_counts = revise_unigram_counts()
        for i in range (1,6):
            for u in unigram_dict:
                if (unigram_dict.get(u) == i):
                    unigram_dict[u] = new_counts[i]
        unigram_dict['unk'] = new_counts[0]
        return unigram_dict
```

We used Good-Turing Smoothing for unigrams and bigrams in a consistent way (let N0 = N1 and use the correction formula from there). The downside of this however, is that you inherently group unseen words with unknown words. This downside is limited though because of the following example. Say your corpus is: “He runs at full speed.” We would argue that the “unseen” bigrams: “run He”, “at run”, “speed He”, etc should have just as much probability mass as “unknown” bigrams such as “Joseph runs.” Just because two words appear in the training set, does not mean that their bigram form should be likely. We understand the use of certain words in certain genres is skewed (Children genre: “unicorn”, “once upon a time”, “fairy”, etc) however this problem would better be handled by a bag of words fix rather than this probability shift towards “unseen bigrams.”

We were curious about the effects of differentiating unseen bigrams from unknown ones and compare the difference. We tried a new method of smoothing. In the second method, we differentiate unseen bigrams from unknown bigrams.

The first step was to go through the training set and insert an ‘unk’ token whenever we hit an “unknown” word. We modeled an unknown word as seeing a word for the first time. (“Joseph is mean to Joseph” -> “unk unk unk unk Joseph”)

```
    def insertUnknowns():
        found = dict()
        for i in range(0,len(tokens)):
            if not found.has_key(tokens[i]):
                found[tokens[i]] = 1
                tokens[i] = 'unk'
```

We then applied Good-Turing smoothing in a different manner. Instead of letting N0 be estimated by N1 (words that appeared once), we let N0 equal the number of unseen bigrams (all permutations of the vocab that doesn’t appear in the training set) and we compute the adjusted count using N1/N0. We then applied

Good-Turing starting from N1. This allows us to distinguish between an unseen word (count should be related to N0) and an unknown word (count should be related to an ‘unk’ entry).

```
#Compute revised Good-Turing counts for N_0 to N_5
def revise_counts():
    no_of_seen_once_bigrams =
get_no_of_seen_once_bigrams()
    no_of_unseen_bigrams = get_no_of_unseen_bigrams()
    adjusted_unseen_bigrams = no_of_seen_once_bigrams
/ no_of_unseen_bigrams
    new_counts = [adjusted_unseen_bigrams]
    prev_count = no_of_seen_once_bigrams
    for i in range(2,7):
        count = 0
        for bi in bigrams_dict:
            if( bigrams_dict.get(bi) == i):
                count = count + 1
        good_turing_count = i * count / prev_count
        prev_count = count
        #print prev_count #original bigram counts
        new_counts.append(good_turing_count)
    return new_counts
```

You can see our results for the different perplexities below in Table 2.

Finally, we create 6 pickle files of smoothed bigrams and unigrams dictionaries for each training corpus (aggregated_train_children, aggregated_train_crime, and aggregated_train_history) so that we can apply the trained language models on the test corpora in sections 4 and 5.

We needed to use some sort of smoothing method in order to eliminate the inevitable 0 probabilities in the test set. When calculating perplexity of a test set, we will inevitably run into unigrams or bigrams that we have not seen in the training set. We needed a way to handle this.

4. Perplexity

Our implementation of calculating perplexity and genre classification can be found in the perplexity_genreClassification.py file.

To calculate perplexity using our first method, we first assume that the smoothed bigrams probability of unknown bigrams are calculated by:

```
prob_unseen_bigram = train_bigrams_dict.get(('unk',
'unk')) / total
```

We didn't differentiate between ('seen word', 'unk') and ('unk, 'unk')
We then use the smoothed bigrams dictionary and the smoothed unigrams dictionary to compute other smoothed bigrams probabilities. And we apply the perplexity formula given to compute perplexities:

```
prob = 0
    for i in range(0, len(tokens)-1):
        count_tokeni = 0
        if ((tokens[i], tokens[i+1]) in
train_bigrams_dict):
            bigram_count =
train_bigrams_dict.get((tokens[i], tokens[i+1]))
            count_tokeni =
train_unigrams_dict.get(tokens[i])
            prob = prob - math.log((bigram_count /
count_tokeni))
        else:
            prob = prob - math.log(prob_unseen_bigram)
    return math.exp(1/len(tokens) * prob)
```

We used the perplexity formula for the average log to avoid underflow errors. By doing this, we're able to compute the perplexity for each of the language models (trained on each of the corpora) on each of the test corpora.

The results are:

	Children_Train	Crime_Train	History_Train
the_magic_city	38.81	31.76	17.82
the_daffodil_mystery	31.23	29.53	18.37
the_moon_rock	33.88	30.22	17.84
bacon	33.06	30.04	19.98

Table 1 Perplexities using method 1

As seen from the above results, the good turing smoothing on the bigram language model is not that accurate.

To calculate perplexity using our second method, we iterated over the test set and applied our second language model to it. We converted unknown words to the 'unk' token and then ran the bigrams through our tests. If the bigram was seen in our training corpus and was ('seen1', 'seen2') --recall 'unk' is included in seen -- then we simply calculated the probability of the second word given the first word as #seen1/#(seen1 seen2) as seen below.

```
if ((tokens[i], tokens[i+1]) in train_bigrams_dict):
```

```
            bigram_count =
train_bigrams_dict.get((tokens[i], tokens[i+1]))
            count_tokeni =
train_unigrams_dict.get(tokens[i])
            prob = prob - math.log((bigram_count /
count_tokeni))
```

However, we also had to check for unseen bigrams (words that appeared in the training set but never formed a bigram). We did this by:

```
elif ((train_unigrams_dict.has_key(tokens[i]) and
train_unigrams_dict.has_key(tokens[i+1])) and
            (train_unigrams_dict.has_key(tokens[i])
and tokens[i+1]=='unk') and
            train_unigrams_dict.has_key(tokens[i+1])
and tokens[i]=='unk'):
            prob = prob -
math.log((prob_unseen_bigram))
```

The results are:

	Children_Train	Crime_Train	History_Train
the_magic_city	28.9	23.72	15.16
the_daffodil_mystery	27.34	25.09	16.11
the_moon_rock	26.77	23.38	14.63
bacon	24.08	21.92	15.94

Table 2 Perplexities using method 2

According to the results of running the different language models, we can see that the second method was superior. Differentiating between unknown and unseen words was a beneficial factor for our language model. It was interesting to compare two different ways of generating a language model and then comparing them. Further interpretation of the results are included in the next section.

5. Genre Classification

We use the values of perplexity in genre classification. Since the lower the perplexity, the better the language model, our algorithm does genre classification based on the lowest perplexity.

```
def genre_classification():
    per_children =
get_perplexity(children_bigrams_dict,
children_unigrams_dict,
'books/test_books/children/the_magic_city.txt')
    per_crime = get_perplexity(crime_bigrams_dict,
crime_unigrams_dict,
'books/test_books/children/the_magic_city.txt')
    per_history = get_perplexity(history_bigrams_dict,
history_unigrams_dict,
'books/test_books/children/the_magic_city.txt')
    best_perplexity = min(per_children, per_crime,
per_history)
    if (best_perplexity == per_children): print
'children'
    if (best_perplexity == per_crime): print 'crime'
    if (best_perplexity == per_history): print
'history'
```

The test corpus is classified to the genre that has the lowest perplexity. The history language model clearly fit the test sets the best in all cases. We hypothesize this to be the case because history writing uses more general language. While the tone and action in children books is very specific to that plot, history writing is more generic. Crime is a little less specific to the plot than children. There are certain recurring themes and expressions. History is the most generic in context and expression, thus it fits most test corpora the best. So our results are reasonable.

6. Extension - Trigram Language Model

In order to generate sentences that make better sense, we experimented with a trigram language model. The implementation is included in extension.py.
You can see the results below. We realized that the with n-grams, as n increases, you begin to see sentences straight out of the text. This is because certain 4 word phrases only appear once and therefore will keep generating the next word in the specific sentence.

Results:

- Given “When the young one”
    - `When the young one of the poor little ones , looking up directly ahead . The child he had come with me , I`
    - `When the young one ; no , Angelica Maria saw a strange longing , as if he were the three were equally unsatisfactory when`
    - `When the young one of the boat where he lay down at the foot of the stream , the snake 's coming ! ''`

- Given “He turned his head”
    - `He turned his head with a growl like that ? '' continued Cardenio , `` and not to be otherwise than what it was`
    - `He turned his head , as I could make , '' said the beast . I will tell thee , where he complained they`
    - `He turned his head and took from his bed head , with its fragrant breath . It is well thought of the mountain .`
- Given “Every sight he beheld in the heavens”
    - `Every sight he beheld in the heavens , how stupid people were so slight as to carry out all at once in a pot of butter mamma`
    - `Every sight he beheld in the heavens and crashing his teeth . There he lay smoking on the road , apparently with the sunset : they beat`
    - `Every sight he beheld in the heavens , with a bright little eyes twinkled with greed and tyranny with which to learn what it was shorter than`

Responsibilities:
We worked on all aspects of this project together.