Report: Project 1 Language Moeling

Hongfei Li (hl963) Shibo Zang (sz428)

1. Unsmoothed n-grams.

For this part, we implement the unsmoothed unigram and bigram language model using given text corpus. We train the language models based on different genres of books, considering every genre as a single corpus. We use Natural Language Toolkit 3.0 to implement word tokenization on the raw text:

```python
for file in files:
    print "Scanning through " + file + "..."
    f = io.open(directory+file, 'r', encoding="utf-8")
    for line in f.readlines():
        tokens += nltk.word_tokenize(line)
```

From above codes, we will have a list of tokens from corpus. To get unigram language model (unsmoothed), 2 values need to be computed for one token: the frequency of specific tokens, count(token), and the number of all words in the corpus, count(words).

```python
def unigram(tokens):
    counts = getWordCount(tokens)
    total_words_number = float(len(tokens))

    for word,count in counts.iteritems():
        counts[word] = count / total_words_number
```

To get bigram language model (unsmoothed), we use the method similar to implementing

unigram language model.

```python
def bigram(tokens):
    # wordCount is a dictionary. Key: tokens; Value: counts.
    wordCount = getWordCount(tokens)

    bigram_counts = {}
    for i in range(len(tokens) - 1):
        # Construct bigram tokens from single tokens
        bigram = tokens[i] + " " + tokens[i + 1]
        if bigram_counts.has_key(bigram):
            bigram_counts[bigram] += 1
        else:
            bigram_counts[bigram] = 1

    # Compute the bigram probabilities of each token.
    prob = {}
    for bigram, count in bigram_counts.iteritems():
        wn_1_count = wordCount[bigram.split()[0]]
        prob[bigram] = count / float(wn_1_count)

    # seeding is a dictionary.
    # Key: tokens; Value: set of words that may appear after the key token.
    seeding = {}
    for i in range(len(tokens) - 1):
        if not seeding.has_key(tokens[i]):
            seeding[tokens[i]] = set()
        seeding[tokens[i]].add(tokens[i+1])

    return (prob, seeding)
```

We count the occurrence of every bigram tokens in the corpus respectively, and then divide them by the whole occurrence of first word of a bigram. At first the corpus is processed line by line, although the result reaches our expectation, we find that this approach ignores some bigrams that cross between two lines. Thus, we decide to read the whole corpus for one time. This modification leads to a little bit improvement on the final genre classification

result.

Data Structure:
We choose dictionary to store token and their probabilities as key-value pairs for both unigram and bigram language model.

2. Random Sentence Generation

We experiment on both unigram language model and bigram language model. For unigram model, seed dosen't make any sense because every generated word is not related with preceding words. For bigram model, we use a dictionary to store all tokens (as key) and their corresponding following tokens (as value), which is stored in a set.

```python
def bigram_sentence_generator(seed, tokens):
    # prob is a dictionary.
    # Key: tokens; Value: probabilities.
    # seeding is a dictionary.
    # Key: tokens; Value: set of words that may appear after the key token.
    prob, seeding = bigram(tokens)

    # Get the last word of input as the original seed.
    returnString = seed.strip()
    while(not is_terminator(returnString[-1:])):
        wordsInSeed = seed.split()
        lastWordInSeed = wordsInSeed[-1]

        # If no following words after the seed, append '.' .
        if not seeding.has_key(lastWordInSeed):
            return returnString + ".\n"

        # possibleWord -> (word, probability)
        possibleWordSet = seeding[lastWordInSeed]
        l = []
        for item in possibleWordSet:
            a = prob[lastWordInSeed + " " + item]
            l.append((item, a))
```

```
        randomNumber = random.random()

        # Get the next word based on their possibilities
        for possibleWord in l:
            if randomNumber < possibleWord[1]:
                returnString += (" " + possibleWord[0])
                seed = possibleWord[0]
                break
            else:
                randomNumber -= possibleWord[1]

    return returnString + "\n"
```

While genrating random sentence, we first set an original seed. Every time generating a random word, we will first search for the candidate set of the seeding word in the dictionary, and then choose the generated word according to their probabilities.

How to choose words by probabilities? We would generate a random number between 0 to 1, and minus the probability of each words in candidate list one by one. When the result turns to negative, we know that this word is the one we look for. And we use the generated word as the new seed to generate next words.

3. Smoothing and unknown words.

- Good-Turing smoothing:

The Good-Turing algorithm is based on computing N_c, the number of N-grams that occur c times. The MLE count of N_c is c. The Good-turing estimate replaces this with a smoothed count c^*, as a function of N_{c+1}.

$$N_c = \sum_{x.count(x)=c} 1$$

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

Based on the formula above, we implement Good-Turing smoothing algorithm.

```python
def __good_turing_discounting(self, counts):
        self.numberOfCounts = {}

        # Get Nc
        for word, count in counts.iteritems():
            if str(count) not in self.numberOfCounts:
                self.numberOfCounts[str(count)] = 0
            self.numberOfCounts[str(count)] += 1

        # Get c*
        smoothedCount = {}
        for word, count in counts.iteritems():
            if count < 5:
                smoothedCount[word] = (count + 1) \
                * self.numberOfCounts[str(count + 1)] / \
                * float(self.numberOfCounts[str(count)])

            else:
                smoothedCount[word] = counts[word]

        return smoothedCount
```

- Handling Unknown Words.

While training bigram language model, we treat every bigram tokens in corpus that occurs for the first time as unknown words, which are marked as <unk>, and count their frequency from the second time they appear.

```python
# Change seen words to <UNK>
inputTokens = nltk.word_tokenize(line)
for i in range(len(inputTokens)):
    token = inputTokens[i]
    if token not in seenTokens:
        seenTokens.add(token)
```

```
        inputTokens[i] = "<UNK>"
    tokens += inputTokens
```

```python
def get_probability(self, string):
        # Get known words probability
        if string in self.probability:
            return self.probability[string]
        else:
            # This part handles unknown words
            if not self.usingGoodTuring:
                return 0
            else:
                # Get the number of all possible bigram combination
                numberOfAllPossibleBigramCombination = len(
                    self.bagOfSeeds) ** 2

                # N1: the number of tokens that appear for only one time
                N1 = self.numberOfCounts["1"]

                if 'numberOfSeenBigram' not in locals():
                    self.numberOfSeenBigram = len(self.bigramCounts)

                # N0: the number of tokens that never appear
                N0 = numberOfAllPossibleBigramCombination - \
                    self.numberOfSeenBigram
                leadingWord = string.split()[0]
                if leadingWord not in self.wordCount:
                    leadingWord = "<UNK>"

                return N1 / float(N0) / float(self.wordCount[leadingWord])
```

4. Perplexity

Suppose we want to compare different language models, there is a useful metric for how well a given statistical model matches a test corpus, called perplexity (PP).

$$PP = (\prod_{i}^{N} \frac{1}{P(W_i|W_{i-1}, \ldots, W_{i-n+1})})^{-\frac{1}{N}}$$

$$PP = exp\frac{1}{N}\sum_{i}^{N} -logP(W_i|W_{i-1}, \ldots, W_{i-n+1})$$

While computing perplexity based on the formula above, we encounter the problem that the first formula would be out of bound because the result is too large. Thus, we use the second formula to reduce the overflow, which works pretty well.

```python
def run_perplexity(model, testTokens):
    pp = 0
    for i in range(len(testTokens)-1):
        bigram = testTokens[i] + " " + testTokens[i+1]
        print "Evaluating \t" + bigram+""
        prob = model.get_probability(bigram)
        print "Probability \t " + bigram + " \t " + str(prob)
        pp += -math.log(prob)
    pp /= (len(testTokens) - 1)
    print "PP is " + str(pp)
    return math.exp(pp)
```

5. Genre Classification

For the three bigram language models we have got from previous procedures, we run each test corpus on every models. Then, we choose the lowest perplexity genre as the classification result of the test corpus since perplexity is used to tell how different language models match a test corpus.

```python
# Langugae Model Training
bigramModels = {}
for genre in genres:
    print "\nGenerating model for " + genre + " genre"
    tokens = get_tokens_from_directory(
```

```
        trainingDirectory + "/" + genre + "/", useUNK=True)
    bigramModels[genre] = Bigram(tokens, usingGoodTuring=True)

testDirectory = currentDirectory + "/books/test_books/"
genres = os.listdir(testDirectory)
testResults = {}

# Run each test corpus on different genre language model
for genre in genres:
    genreDirectory = testDirectory + genre
    for f in os.listdir(testDirectory + genre):
        print "\nAnalysing book " + f + " under genre " + genre
        fileDirectory = genreDirectory + "/" + f
        bookTokens = get_tokens_from_file(fileDirectory)
        minPP = sys.maxint
        resultGenre = ""
        # Iterate different genre language model
        for key, model in bigramModels.iteritems():
            pp = run_perplexity(model, bookTokens)
            print "Perplexity for " + key + " model is " + str(pp)
            # Choose the model that has the minimum pp
            if pp < minPP:
                minPP = pp
                resultGenre = key
        testResults[f] = resultGenre

print "----------------------------------------"
print "Here are the results"
pp = pprint.PrettyPrinter(depth=6)
pp.pprint(testResults)
```

The genre classification result is shown below:

```
Generating model for children genre
Loading the_junior_classics_vol_1.txt...
Loading the_junior_classics_vol_4.txt...
Loading the_junior_classics_vol_5.txt...
```

```
Loading the_junior_classics_vol_6.txt...
Loading the_junior_classics_vol_7.txt...
Loading the_junior_classics_vol_8.txt...

Generating model for crime genre
Loading a_thief_in_the_night.txt...
Loading arsene_lupin.txt...
Loading crime_and_punishment.txt...
Loading the_adventures_of_sherlock_holmes.txt...
Loading the_extraordinary_adventures_of_arsene_lupin.txt...
Loading the_mysterious_affair_at_styles.txt...

Generating model for history genre
Loading famous_men_of_the_middle_ages.txt...
Loading julius_caesar_vol_1.txt...
Loading queen_victoria.txt...

Analysing book the_magic_city.txt under genre children
Perplexity for history model is 10958.2156894
Perplexity for children model is 1837.77667763
Perplexity for crime model is 1674.88627338

Analysing book the_daffodil_mystery.txt under genre crime
Perplexity for history model is 6501.76893237
Perplexity for children model is 1420.1544746
Perplexity for crime model is 777.910297553

Analysing book the_moon_rock.txt under genre crime
Perplexity for history model is 6896.81465931
Perplexity for children model is 1467.87519207
Perplexity for crime model is 1069.7520652

Analysing book bacon.txt under genre history
Perplexity for history model is 4915.8722572
Perplexity for children model is 3967.04764472
Perplexity for crime model is 3907.83250887
--------------------------------------
Here are the results
{'bacon.txt': 'crime'
```

```
{'bacon.txt': 'crime',
 'the_daffodil_mystery.txt': 'crime',
 'the_magic_city.txt': 'crime',
 'the_moon_rock.txt': 'crime'}
```

From above results, we can see that all test books are classified into "crime" genre, which seems incorrect. In addition, we can see that, for each test corpus, the perplexity of children genre and crime genre are similar, while the history language model have much higher perplexity compared with the other two genres.

We think that this is because the training set of History genre is relatively small, which leads to smaller size of bigram token set and less features for this genre. Thus, when computing perplexity, the probability of bigram tokens from test corpus matching tokens in language corpus would be relatively small, which causing the perplexity larger than other two models.

Extension

1. Trigram

Trigram is one of the extensions that we have implemented.
We copied code from the bigram class and modiefied the following some methods. Bellow are some of the essential ones.

```python
    def __generate_word_count(self):
        counts = {}
        for i in range(len(self.tokens) - 1):
            leadingTwoWord = self.tokens[i] + " " + self.tokens[i+1]

            if leadingTwoWord in counts:
                counts[leadingTwoWord] += 1
            else:
                counts[leadingTwoWord] = 1
        return counts
```

```python
    def __generate_trigram_counts(self):
        self.trigramsCounts = {}
        for i in range(len(self.tokens) - 2):
            trigram = self.tokens[i] + " "
                + self.tokens[i + 1] + " " + self.tokens[i + 2]

            if trigram in self.trigramsCounts:
                self.trigramsCounts[trigram] += 1
            else:
                self.trigramsCounts[trigram] = 1
        return self.trigramsCounts
```

```python
        # Probability of trigrams
        self.probability = {}
        for trigram, count in self.trigramsCounts.iteritems():
            leadingWord = trigram.split()[0] + " " + trigram.split()[1]
            if leadingWord != "<UNK> <UNK>":
                wn_1_count = self.wordCount[
                    trigram.split()[0] + " " + trigram.split()[1]]
                self.probability[trigram] = count / float(wn_1_count)
```

Using Trigram models dramatically increased the perpelixity of language models, as well as changed the classification result for every test book. We think one possible reason that the perplexity number increases is because trigram models have a much lower probability for each entry in the model. **Though result still wrong, we do not understand why the classification result is affected in this way.**

```
Using Bigram Model

Analysing book the_magic_city.txt under genre children
Perplexity for history model is 8860.8216519
Perplexity for children model is 1560.19431586
Perplexity for crime model is 1422.48116425
```

```
Analysing book the_daffodil_mystery.txt under genre crime
Perplexity for history model is 5276.49395515
Perplexity for children model is 1227.99519085
Perplexity for crime model is 669.821934126

Analysing book the_moon_rock.txt under genre crime
Perplexity for history model is 5590.96693877
Perplexity for children model is 1253.23871377
Perplexity for crime model is 912.984987405

Analysing book bacon.txt under genre history
Perplexity for history model is 3973.86620869
Perplexity for children model is 3385.16748929
Perplexity for crime model is 3314.36084177
---------------------------------------
Here are the results
{'bacon.txt': 'crime',
 'the_daffodil_mystery.txt': 'crime',
 'the_magic_city.txt': 'crime',
 'the_moon_rock.txt': 'crime'}
```

```
Using Trigram Model
Generating model for history genre
Loading famous_men_of_the_middle_ages.txt...
Loading julius_caesar_vol_1.txt...
Loading queen_victoria.txt...

Analysing book the_magic_city.txt under genre children
Perplexity for history model is 7.14531243e+11
Perplexity for children model is 3.07295422546e+12
Perplexity for crime model is 1.44406326509e+12

Analysing book the_daffodil_mystery.txt under genre crime
Perplexity for history model is 6.20530368808e+11
Perplexity for children model is 4.07026723447e+12
Perplexity for crime model is 1.45031259953e+12

Analysing book the_moon_rock.txt under genre crime
```

```
Perplexity for history model is 6.67609347175e+11
Perplexity for children model is 3.64720402324e+12
Perplexity for crime model is 1.4769648119e+12

Analysing book bacon.txt under genre history
Perplexity for history model is 5.77154465154e+11
Perplexity for children model is 4.81964303807e+12
Perplexity for crime model is 2.02585164205e+12
---------------------------------------
Here are the results
{'bacon.txt': 'history',
 'the_daffodil_mystery.txt': 'history',
 'the_magic_city.txt': 'history',
 'the_moon_rock.txt': 'history'}
```

2. Add One Smoothing

Apart from good-turing smoothing, we have also implemented add one smoothing. Bellow are the code changes. Turns out the add one smoothing methdolog decreases perplexity values, but does not change the result of classification.

```python
    def __add_one_smoothing(self, counts):
        self.numberOfCounts = {}
        smoothedCount = {}

        for word, count in counts.iteritems():
            smoothedCount[word] = count+1
        # # Get Nc
        for word, count in counts.iteritems():
            if str(count) not in self.numberOfCounts:
                self.numberOfCounts[str(count)] = 0
            self.numberOfCounts[str(count)] += 1

        return smoothedCount
```

Here is the result from good-turing smoothing

```
Bigram Good-turing smoothing classification result

Analysing book the_magic_city.txt under genre children
Perplexity for history model is 10958.2156894
Perplexity for children model is 1837.77667763
Perplexity for crime model is 1674.88627338

Analysing book the_daffodil_mystery.txt under genre crime
Perplexity for history model is 6501.76893237
Perplexity for children model is 1420.1544746
Perplexity for crime model is 777.910297553

Analysing book the_moon_rock.txt under genre crime
Perplexity for history model is 6896.81465931
Perplexity for children model is 1467.87519207
Perplexity for crime model is 1069.7520652

Analysing book bacon.txt under genre history
Perplexity for history model is 4915.8722572
Perplexity for children model is 3967.04764472
Perplexity for crime model is 3907.83250887
---------------------------------------
Here are the results
{'bacon.txt': 'crime',
 'the_daffodil_mystery.txt': 'crime',
 'the_magic_city.txt': 'crime',
 'the_moon_rock.txt': 'crime'}
```

Add one smoothing classification result

```
Bigram add one smoothing classification result

Analysing book the_magic_city.txt under genre children
Perplexity for history model is 8860.8216519
Perplexity for children model is 1560.19431586
```

```
Perplexity for crime model is 1422.48116425

Analysing book the_daffodil_mystery.txt under genre crime
Perplexity for history model is 5276.49395515
Perplexity for children model is 1227.99519085
Perplexity for crime model is 669.821934126

Analysing book the_moon_rock.txt under genre crime
Perplexity for history model is 5590.96693877
Perplexity for children model is 1253.23871377
Perplexity for crime model is 912.984987405

Analysing book bacon.txt under genre history
Perplexity for history model is 3973.86620869
Perplexity for children model is 3385.16748929
Perplexity for crime model is 3314.36084177
--------------------------------------
Here are the results
{'bacon.txt': 'crime',
 'the_daffodil_mystery.txt': 'crime',
 'the_magic_city.txt': 'crime',
 'the_moon_rock.txt': 'crime'}
```