CS 4740 – Programming project 1

Responsibilities:

1. Remya Balasubramanian - Preprocessing and Unigram, smoothing (code for unigram, trigram), random sentence generation (bigram) truthfulness of hotel review (testing), report for preprocessing-n gram model, random sentence generation
2. Yashaswini Shekarappa – Bigram, Extension (trigram), random sentence generation (unigram, trigram), smoothing (bigram code), truthfulness of hotel review (code), report for smoothing, perplexity and truthfulness for hotel review

***Programming Language Used: Python***

1. Unsmoothed n-grams:

Data Structures: Dictionary, Lists, Iterators

Python APIs: Collections- counter, Regex – re, split method

1.1 Preprocessing:

To extract the raw test from the two corpora, we used the python regular expression module re. We have handled the following cases:

* To strip the xml tags used the regex module -compile [re.compile(r'<.\*?>')] and later used the sub (substitute) library [remove\_tag.sub('',str)].
* We also used the same combination of compile and sub to remove the patterns
  + Verse numbers in bible corpus(digit:digit)
  + (number,number,) pattern in hotel review corpus
  + Remove the header(IsTruthFul,IsPositive,review) in the hotel review corpus
  + Replace single newline by space and strip multiple newlines
* <s> and </s> are used as sentence boundaries markers. We have considered ( . ? !) as end of sentences so we can add the sentence markers <s> and </s> .Also verse numbers can be in the middle, so considered them for sentence segmentation as well.
* Preprocessing function will be called for every line in the training/test file.

1.2 Unigram:

To generate unigrams and calculate the unigram probabilities the following method was used:

* Read the preprocessed file using the read() inbuilt method.
* Use the in-built method split() and extract the token list.

*token = fproc.split()*

* Generate the unigram dictionary using the Counter collection library and store the unigrams and their frequencies.

*unigram\_hash = dict(Counter(token).items())*

***Unigram probability calculation:***

* Each item’s count in the unigram dictionary is divided by the count of the total items[complete word count] in that dictionary.
* The above is done for all items including the sentence segmentation markers, punctuations
* The resulting probability value for each item in the dictionary is stored in the unigram\_prob dictionary .The unigrams with their frequency and their probability dictionaries [two dictionaries] is returned by this function.
* Sample code for unigram probability generation:

*unigram\_count = sum(unigram\_hash.values())*

*for value in unigram\_hash.keys():*

*temp = float(unigram\_hash.get(value))/float(unigram\_count)*

*unigram\_prob[value] = temp*

1.3 Bigram:

For bigram generation, we used a list and appended the tokens considering i and i+1 till the end of the preprocessed file is reached. For the sentence in the file “<s>I was here.</s>”, the newlist would contain [(<s> I ), (I was),(was here),(here .),(. </s>)]

*bigramlist = token*

*while i<len(bigramlist):*

*if i+1<(len(bigramlist) -1):*

*newlist.append(bigramlist[i]+" " + bigramlist[i+1])*

*i += 1*

Bigram probability is calculated as follows:

* Generate the bigram count dictionary using the Counter collection library and store the bigrams and their frequencies.
* For each bigram in the list created, its probability is calculated by dividing the bigram count obtained in the above step by the unigram count of the first word of the bigram
* The result is saved in the dictionary and returned to the calling function
* Sample code for bigram probability generation:

*for w in newlist[:]:*

*first=w.split(" ")*

*bigramfrequency= bigram\_hash.get(w)*

*unifrequency=unigram\_hash.get(first[0])*

*temp= float(bigramfrequency)/float(unifrequency)*

*bigram\_prob[w]= temp*

2. Random Sentence Generation:

To generate the random sentence, we used the bigram and later the trigram model.

We chose the bigram model over the unigram one as it will account for better predictions (as it has the next word). After the sentence generation we found that the trigram model generates better random sentences.

***Random sentence generator logic using Bigrams:***

* Limited to sentence length to 30
* Generate a random number(x) using math library (random.uniform) between 0 and 1
* Iterate over the bigram probability dictionary,
  + To start with, choose the bigram word from the probability dictionary which is in the range [x-0.1, x+0.1] or matches the probability value and starts with <s>.

Save the second unigram in the bigram so we search for the next bigram using this value as the sentence would make more sense.

*if((value == random\_p and key.split()[0] == '<s>') or (value < random\_p + 0.1 and value > random\_p - 0.1 and key.split()[0] == '<s>')):*

*random\_sentence += key.split()[1] + ' '*

*prev = key.split()[1]*

* + For the middle words in the sentence, choosing the bigram word from the dictionary uses the similar logic as above except that the unigram word should match the previous one which have saved.

*if((value == random\_p and key.split()[0] == prev) or (value < random\_p + 0.1 and value > random\_p - 0.1 and key.split()[0] == prev)):*

*random\_sentence += key.split()[1] + ' '*

*prev = key.split()[1]*

* + For the sentence completion, choose the bigram which ends with </s> along with the check for the probability range.

*if((value == random\_p and key.split()[1] == '</s>') or (value < random\_p + 0.1 and value > random\_p - 0.1 and key.split()[1] == '</s>')):*

*random\_sentence += key.split()[0]*

* We then trim the <s>, </s>,</s><s> pattern generated in the random sentence

**Examples of random sentence generated through bigram:**

For bible corpus:

* And the almug trees, and Josabad the LORD spake by the LORD . But ye are, and Josabad the sons and all that ye the people came ?
* But thou mayest be unclean place was a burnt the people to do the son reigned sixteen thousand talents of the almug trees, that were with him, he had ?
* And they shall not be Moses, Get me king also made provision was hot for Hadadezer king spared to the sons and all that ye the people came upon ?
* And the almug trees, which sinneth not,) and Josabad the almug trees, and the sons and all that ye the people came upon his stead brasen altar shall go .
* And the almug trees until even the daughter looked back again conceived, that killeth a portion assigned Uriah the almug trees until even the oversight of the LORD.

For hotel review corpus:

* The hotel in the automatic sensors in history of the automatic sensors in the staff . The hotel is a miserable night (20 dollars for $252 a night)?
* The hotel is fascinating article; as the automatic sensors in history in Chicago Hotel Monaco in Chicago . The hotel is in and the front (not stained ?
* Hardly what Swissotel is breathtaking entryway . Hardly what I was hesitant to Lobby; and artwork - Spa and the automatic sensors in the hotel is very ?
* They made an upscale Italian food that was very good and I was a great view . The staff . The room . !
* The hotel in Chicago . Hardly what I was a great view . Hardly what I was not something like Monaco in that originally;

**Random sentence generation using trigrams:**

Random sentence generator logic using trigrams:

* Limited to sentence length to 30
* Generate a random number(x) using math library (random.uniform) between 0 and 1
* Iterate over the trigram probability dictionary,
  + To start with, choose the trigram word from the probability dictionary which is in the range [x-0.1, x+0.1] or matches the probability value and starts with <s>.

Save the second bigram in the trigram so we search for the next trigram using this value as the sentence would make more sense.

*if((value == random\_p and key.split()[0] == '<s>') or (value < random\_p + 0.1 and value > random\_p - 0.1 and key.split()[0] == '<s>')):*

*random\_sentence += key.split()[1]+' '+key.split()[2] + ' '*

*prev1= key.split()[1]*

*prev2 = key.split()[2]*

* + For the middle words in the sentence, choosing the trigram word from the dictionary uses the similar logic as above except that the bigram word should match the previous one which have saved(prev1 and prev2).

*if((value == random\_p and key.split()[0] == prev1 and key.split()[1]==prev2) or (value < random\_p + 0.1 and value > random\_p - 0.1 and key.split()[0] == prev1 and key.split()[1]==prev2)):*

*random\_sentence += key.split()[2] + ' '*

*prev1 = key.split()[1]*

*prev2= key.split()[2]*

* + For the sentence completion, choose the trigram which ends with </s> along with the check for the probability range.

*if((value == random\_p and key.split()[2] == '</s>') or (value < random\_p + 0.1 and value > random\_p - 0.1 and key.split()[2] == '</s>')):*

*random\_sentence += key.split()[0]+' '+key.split()[1]*

**Examples of random sentence generated through trigram:**

For bible corpus:

* The LORD stirred up an evil spirit between Abimelech and Phichol the chief of the LORD stirred up an evil name upon the two sides dish .
* hath he done ? And the channels of the staves were seen unto this place, because the LORD stirred up an evil spirit between Abimelech and dish .
* hath he thrown into the land of Moriah; and offer them for thee in time to come, and make the girdle of needlework .
* Hear me, my son Shechem longeth for your generations concerning the words of this law and her towns, Dor and her issue shall wash his flesh with the dish .
* Hear me, my son ? And the channels of the staves into the land stank . And the channels of the chronicles of the dish .

For hotel review corpus:

* Went to the Chicago downtown; closer to Michigan Ave . Thirdly I asked for my luxury shopping weekend in Chicago for our stays in business .
* Lastly; my sister . We had booked every room (and not in their truck; doing nothing . We arrived to Chicago .
* Lastly; my sister . I hope not our style and sensibilities when helping us out . I was very very tiny .
* Though the accommodations are unbeatable . I hope the food was fantastic ! Thirdly I asked .Thirdly I asked if they were friendly.
* Went to the remote to work in . I hope the food was fantastic from the subway (Red line) . I hope the food was fantastic

3. Extension (Trigrams)

We chose to implement the extension (trigrams) from the list as we wanted to see how it will improve on generating random sentences, perplexity values and prediction of truthfulness of the hotel reviews.

In case of random sentences, we saw that the sentences generated were close to the sentences in bible corpus and hotel reviews, much more effective than the bigram model. Also the perplexity values were lower than the bigram model. In case of predicting the truthfulness of the hotel reviews, we found that the trigram values gave us better prediction than the bigram/unigram ones.

For trigram generation, we used a list and appended the tokens considering i and i+1 and i+2 till the end of the preprocessed file. For the sentence in the file “<s>I was here.</s>”, the newlist1 would contain [(<s> I was ), (I was here),(was here .),(here . </s>)]

*while j<len(trigramlist):*

*if (j+2)<(len(trigramlist) -1):*

*newlist1.append(trigramlist[j]+" " + trigramlist[j+1]+" "+trigramlist[j+2])*

*j += 1*

Trigram probability is calculated as follows:

* Generate the trigram count dictionary using the Counter collection library and store the trigrams and their frequencies.
* For each trigram in the list created, its probability is calculated by dividing the trigram count obtained in the above step by the bigram count of the first word of the bigram
* The result is saved in the dictionary and returned to the calling function
* Sample code for trigram probability generation:

*for w in newlist1[:]:*

*first1=w.split(" ")*

*trigramfrequency1= trigram\_hash.get(w)*

*seq=[first1[0], first1[1]]*

*bigram\_split=" ".join(seq)*

*bigramfrequency1=bigram\_hash.get(bigram\_split)*

*temp= float(trigramfrequency1)/float(bigramfrequency1)*

*trigram\_prob[w]= temp*

4. Smoothing

Smoothing is necessary for assigning some of the total probability mass to unseen words or N-grams. We have used the Good Turing model of Smoothing to handle unknown n-grams.

**Intuition**: Use the count of things you have seen once to help estimate the count of things you have never seen. We tag the unknown words as “<UNK>” and assign its probability as N1/N based on good turing smoothing.

Formulae for updated counts are as follows:

*count\* = ((count + 1)\* N(count+1)) / Ncount, for count greater than 1*

*count\*= N(count+1)*

* Generate unigrams, bigrams
* Calculate their probabilities
* Smooth the models for counts less than 5
* Assign a probability to unknown words/n-grams

Sample code:

*if(count<5): #if count is less than 5, update the probabilities*

*nc = countN(count\_hash,count)*

*ncn = countN(count\_hash,count + 1)*

*if ncn !=0: #handle divedeByZero condition*

*gt\_temp[sample] = float(count + 1) \* float(float(ncn) / float(nc \* N)*

*#assigning probabilities to unknown words/n-grams*

*sample = "<UNK>"*

*gt\_temp[sample] = float (countN(count\_hash,1)) / float(N)*

*return gt\_temp*

5. Perplexity

Perplexity is the inverse probability of the test set, normalized by the number of words.

Formula to calculate perplexity: *2-(sum of probabilities of all words/ngrams)*

* Preprocess the file
* Calculate probabilities of words/n-grams
* For each word/n-gram in the test file, add the probabilities in base 2

p+= math.log(prob\_hash[word],2)

* If the word/n-gram is unknown, take the probability of <UNK>

p+= math.log(prob\_hash["<UNK>"], 2)

**Calculating perplexity- Code snippet:**

*for word in token:*

*if word in prob\_hash.keys():*

*p+= math.log(prob\_hash[word],2)*

*else:*

*p+= math.log(prob\_hash["<UNK>"],2)*

l = float(p)/float(M)

perplexity = 2 \*\* (-1 \*l)

6. Programming project- Determining the truthfulness of hotel reviews

* From the hotel reviews training file, read each review and write it into truthful/untruthful set based on its truthfulness tag. Now we have two files, one that contains all the reviews that are truthful and another that contains all the reviews that are untruthful
* Train the language models separately for truthful and untruthful sets using unigram and bigram models
* Take each review in the test file and calculate the perplexity of the review using truthful review set and then the untruthful review set
* If the perplexity value generated using the truthful review set is less than the perplexity value generated by the untruthful review set, the review is assigned a true value or it is assigned a false value

We first used the unigram model to determine the truthfulness of the hotel reviews and submitted the results on Kaggle. It gave us 40% accuracy. We then used the bigram model and saw that it improved our score as the perplexity values were better (46%). Thus, the bigram model could predict the truthfulness better than the unigram model.

**Accuracy of approach on validation data:**

We also tested the truthfulness of the 279 hotel reviews in the validation file, out of which 97 predictions were right (~37%). We used trigram model for the same. We have included the results file in the zip.

**Conclusion**

We used unigram, bigram and trigram language models and good turing smoothing in our project, to predict the truthfulness of the hotel reviews. We noticed that the accuracy improvement from bigram to trigram was not very prominent. We conclude that to improve the accuracy of prediction , we need to explore different smoothing methods.