## Responsibilities

* FNU Arpana Hosabettu (fa97) - Preprocessing the corpus, Unigram model, generation, Testing, Report write up, good turing, perplexity
* Harsh Shah (hs634) - Bigram model generation, Random text generation, perplexity testing, report write up, truthfulness of hotel reviews

## Programming Portion

### Unsmoothed ngrams

* Sentence segmentation tools - **nltk**
* Programming Language - **Python**
* Data Structures Used **- Lists, Dictionary, tuple, iterators, generators**

### Preprocessing:

The two corpora provided include labels, xml tags and require preprocessing to extract the raw text. The following steps are executed to preprocess the text.

* nltk's clean\_html() strips the xml tags
* sub (substitute) - Python's regular expression library - is used to strip away labels and some numbers that are specific to corpora.
* nltk's sentence tokenizer tokenizes the string based on sentence boundaries.
* <s> and </s> are used as sentence boundaries markers.
* nltk's PunkWordTokenizer is used to split sentence into words

### Random Sentence Generation

### Unigrams:

Using the list that holds all the words in the corpora, we use Python's Counter to count the number of occurrence of unique words. Python's Counter method returns a dictionary with the unique words and their counts. An example usage of Counter is shown below:

unigrams\_dict = Counter(word\_list)

Unigram probability is calculated as follows:

* Divide the word count with the count of all words in the corpora including the sentence segmentation markers
* Store the words and their probability in a dictionary

The code used is shown below:

for key, value in unigrams\_dict.iteritems():

unigrams\_probability\_dict[key] = round(value/float(unigrams\_len), 6)

### Bigrams

For bigram, python iterator zip is used to create a list of tuple of the word with its context word. The zip method takes two iterators and "zips" the values together to create a new list of tuples. An example of the output of zip is shown below:

[('<s>', 'in'), ('in', 'the'), ('the', 'beginning'), ('beginning', 'god'), ('god', 'created'), ('created', 'the'), ('the', 'heaven'), ('heaven', 'earth), ('earth.', '</s>')]

Bigram probability is calculated as follows:

* The bigram pair's count is obtained from the Counter method.
* For each bigram its probability is obtained by dividing its count by the unigram count of the first word of the bigram
* The result is saved in the dictionary

The code is shown below:

for key, value in bigrams\_dict.iteritems():

bigrams\_probability\_dict[key] = round(value/float(unigrams\_dict[key[0]]), 6)

### Random sentence generation logic

The design of the unigram random sentence generator is as described below:

1. Find the maximum probability of a word in the dictionary which contains unigrams and their probabilities.
2. Generate a random number between 0 and the max probability found in first step.
3. Find the unigram whose probability is closest to the random number generated in step 2.
4. If the difference in probabilities is too large, then reject the unigram. This is to make sure that the unigram chosen is close to the random number.
5. If the unigram's probability is reasonably close, then add the word to the sentence.
6. If the end sentence marker (</s>) is encountered, stop generating the sentence.
7. Remove the sentence markers and return the sentence generated.

The design of the bigram random sentence generator is as described below:

1. Generate a random number between 0 and 1.
2. If the sentence doesn't contain the sentence start marker (<s>), then search for all bigrams whose first word is the sentence start marker.

while len(sentence.split()) < sent\_length and '</s>' not in sentence:

* 1. Find the bigram whose probability is closest to the random number generated in first step.

closest\_key, closest\_val = min(short\_bigram\_dict.iteritems(), key=lambda (k, v): abs(v - rand\_prob))

* 1. Add the bigram (both words) to the sentence.

sentence += closest\_key[0] + " " + closest\_key[1]

OR

1. Find the last word added to sentence and search for all bigrams with that word as its first.
   1. Find the bigram whose probability is closest to the random number generated in first step.
   2. Add the second word to the sentence.
2. Repeat the steps till an end sentence marker is found or the number of words in the sentence exceed 30
3. Remove the sentence markers and return the sentence generated.

re.sub('<[^<]+?>', "", re.sub(' +([,:!?])', r'\1', sentence))

### Examples of random sentence generated

### King James Bible

#### Unigram

* I And the the his and the unto, And to of of in  it the of  to
* do to for And LORD take of and the and of and of,  the and the and,
* his and of were to the the of, and the to and  the   And ; to

#### Bigram

* hath given rest and the LORD, and the LORD, and bid them, and bid thee, threescore cities of Shittim, and the LORD, threescore
* And Abijam and they shall be Gamaliel the LORD, and the revenger of their fathers made booths for evil in beeves, and they proclaimed these are the
* And he must be Gamaliel the LORD, threescore and the revenger of the LORD saved them, and the revenger of the LORD saved alive, threescore and
* And Abijam, and bid thee, and bid them whom David, threescore and the LORD, threescore and the revenger of the LORD, and the LORD

### Hotel Review

#### Unigram

* the in  on The the stayed in! ; the was room ;  this was to  in
* the was and   and the  the not the ;  in   the the for I
* the to the in  on to in the  and open it  for on a the to

#### Bigram

* based solely on groceries to the hotel is a ride by the completely different hotel deluxe room was very nice!!.
* The room in the hotel is a great breakfast was a ride to the completely miserable.
* The room was a ride by a great breakfast ; sheets were very disappointing travel needs ; and the completely different room recently.
* based on groceries to standard to the hotel is somewhat reluctant to standard room was a great breakfast is a great breakfast servers were very nice and the completely.

## Smoothing; unknown words.

* After preprocessing and counting the words, added a symbol <UNK> to bigram dictionary and <UNK>, <UNK> to bigram dictionary of words.
* Applied good turing to smooth the counts of low frequency words. Threshold k is chosen as 5.
* The formula - updated count = ((count + 1)\* N(count+1)) / Ncount when count > 1 else count is N(count+1)) where N is the frequency of frequency of words
* Calculated the unigram and bigram probabilities as explained in the earlier section.
* Calculated the good turing probabilities for counts less than 5 which were already adjusted using formula by diving the frequency by N where N is total number of words in corpus.

freq\_c = bigrams\_freq\_dict[c]

freq\_c\_1 = bigrams\_freq\_dict[c+1]

if freq\_c\_1 != 0:

updated\_c = ((c + 1) \* freq\_c\_1) / float(freq\_c)

bigrams\_dict[key] = updated\_c

## Perplexity

* Preprocess the test file
* For each unigram word in test, sum up the logs of unigram probability of the word in base 2.
* For each bigram words in test file, sum up the logs of bigram probabilities of the word in base 2.
* Calculate perplexity as 2\*\*(-1\*(sum(probabilities)))

N = float(len(file\_words))

logP = round(((-1 \* sentence\_prob)/N), 6)

perplexity = (2 \*\* (logP))

* Unigram Perplexity for kjbible test set: 317.798388759
* Bigram Perplexity for kjbible test set: 69.3123246327
* Unigram Perplexity for test set:390.610792247
* Bigram Perplexity for test set: 66.5955558775

## Ngram general model (trigram random sentence extension)

As an extension, we developed an ngram method which will generate ngrams for input tokens. We implemented trigram using this model for generating random sentences and comparing with the bigram and unigram models. Our implementation is a general ngram extension.

We evaluated the random sentence generated using the trigram model versus the bigram model. As can be seen in the example sentences generated by both the models, the quality of sentences generated by the trigram model is definitely better than the ones generated using the bigram model. Our trigram extension, as expected, provided a better model with more context specific tokens. Hence, this model generated better random sentences than the bigram or unigram model.

#### Ngram generation algorithm design

* As in bigram generation, we use two iterators and slice the list of words depending on the value of ‘n’ in ngram.
* The ngram method is a generator which returns one ngram at a time.
* It is passed to the Counter method to calculate the frequency of each ngram
* Finally, the probability of each trigram is calculated based on its count and the corresponding bigram count

#### Random Sentence Generator

* Its design is similar to unigram and bigram random sentence generator.
* A random number is generated and the sentence start marker trigram is chosen based on the random number
* The subsequent trigrams are chosen based on the previous selection
* The sentence is considered complete either when we encounter sentence end marker or a specified number of words.

#### Trigram Random Sentences

***kbible***

* Thou shalt speak unto Joshua, mighty men and women, as his brethren and all Bashan, No, cities of Bedad, and established before the sides of
* surely to fear the foundation of me, For now I am a prophet that brought into the wilderness a perfect lot.

***Hotel review***

* room was soooo bored of the list ; helpful ; did not only did nothing to smoke and promised Service was met with many prepared for the maids ; always there wasn 't.
* Yes ; buffet style that you had to smoke and drinks even helped me!

## Programming Project Truthfulness of Hotel Reviews

The purpose of this exercise is to evaluate the truthfulness of review. The model is trained on the training data using bigram and then used to predict truthfulness of review on the validation and test set.

The approach is to create a bigram model using the training data. Then process each sentence of the validation and test data to calculate perplexity of each review. Based on the perplexity value, we predict the truthfulness of each review

The steps are as follows:

* The training set is preprocessed and separated into individual parts for processing.
* Based on the value of the is\_truthful flag, each review is flagged as truthful or false and stored into the corresponding set.
* Bigram model is generated for both the sets of reviews (true and false reviews)
* Now the test file has to be evaluated and the truthfulness of review predicted.
* For each sentence in the test file, the truth and false perplexities are calculated from the bigram model.
* The lower of the truth and false perplexity value gives an idea of whether the review is true or false. If the truth perplexity is lower than the false perplexity, the review is predicted to be true and vice versa.
* Finally, the prediction results are published to a csv file for Kaggle review

The following table shows the accuracy of the approach on validation data:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |