NLP Assignment 2

Drishya Uniyal	**MT21119**
Harshit Gupta	**MT21028**

Steps Followed.

1. **Text Preprocessing.**
    - Removal of punctuations
    - html tags
    - Urls
    - Stopwords
    - Removal of numbers.

2. **Functions have been made for the implementation of Bigram Language model.**

    a. train() : Returns the following: listOfBigrams, unigramCounts, bigramCounts
    b. get_bigram_prob() : Returns bigram probability of the sentence
    c. get_bigram_good_turing_prob() : Returns bigram probability of the sentence with good turing smoothing
    d. calculate_bigram_prob() : Assigns bigram probabilities
    e. calculate_bigram_prob_positive_negative() : Assigns bigram probability. A function that generates sentiment oriented sentences.
    f. good_turing_smooting() : Assigns good turing smoothed probabilities

    $$P(\text{words with zero-count}) = \frac{N_1}{N}$$
    $$P(\text{words with non-zero-count}) = \frac{(c+1)N_{c+1}}{N_c * N}$$

    Nc - number of words with frequency c
    Steps:
    - check the number of times word occurs, frequency.
    - Check zero occurrence probability to check these many words appear only once.
    - Maintain set for non-existing number of words.
    - Get the word unit (bigram), their count and probability.

    g. bigram_generate_sentences() : generates random sentences.
    h. ppl_bigram() : Returns the bigram perplexity of the given list of sentences with smoothed values.

i. vader_analysis() : to assign the sentiment to the generated sentences.

3. Evaluation

a. Multinomial NB for dataset A and B on test dataset.

4. Models saved

a. The models have been saved for smoothed bigram language and MNB.

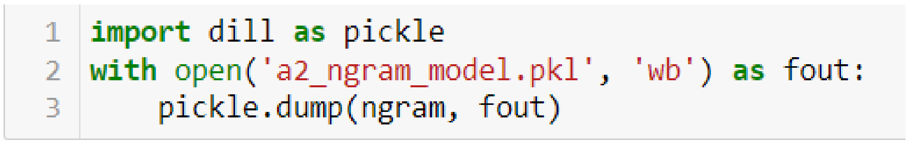
Desired outcomes:

1. Part A

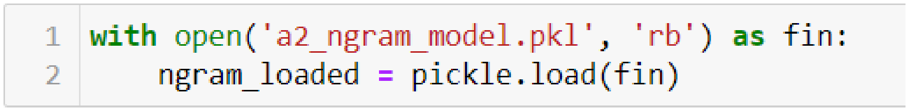
Save smoothed bigram language model

Bigram count and their probabilities before and after smoothing have been saved in “2gram.csv” and “good_turing.csv” respectively.

The bigram model that generates the sentences has been saved as `a2_ngram_model.pkl`

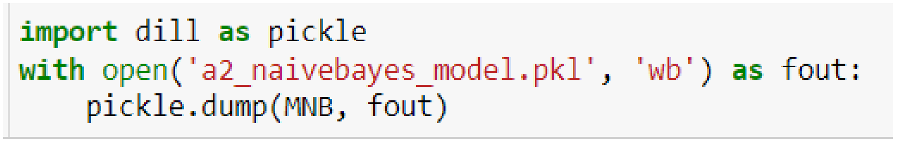


```
import dill as pickle
with open('a2_ngram_model.pkl', 'wb') as fout:
    pickle.dump(ngram, fout)
```



```
with open('a2_ngram_model.pkl', 'rb') as fin:
    ngram_loaded = pickle.load(fin)
```

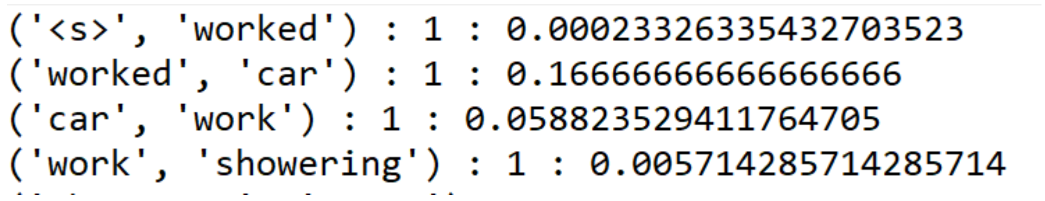
The model for naive bayes is also saved.



```
import dill as pickle
with open('a2_naivebayes_model.pkl', 'wb') as fout:
    pickle.dump(MNB, fout)
```

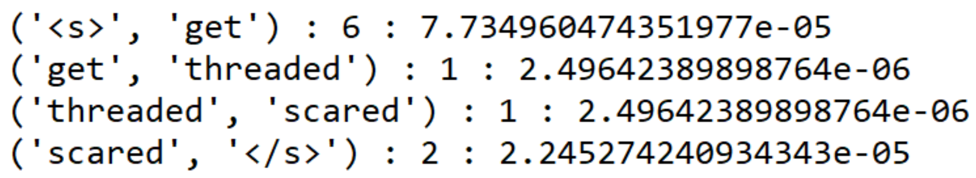
Report the Top-4 bigrams and their score after smoothing.

Bigrams as saved in the file.
Before.



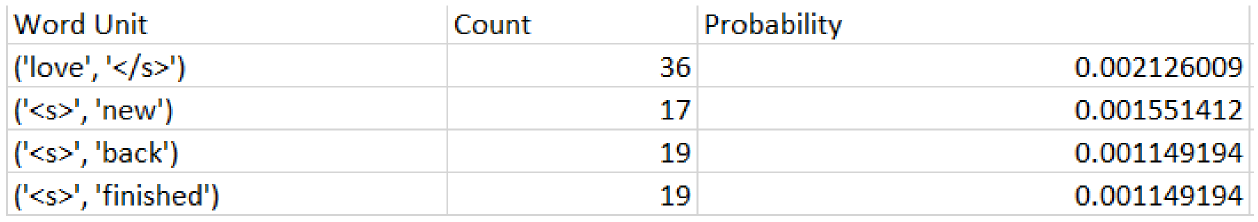
```
('<s>', 'worked') : 1 : 0.00023326335432703523
('worked', 'car') : 1 : 0.16666666666666666
('car', 'work') : 1 : 0.058823529411764705
('work', 'showering') : 1 : 0.005714285714285714
```

After good turing



```
('<s>', 'get') : 6 : 7.734960474351977e-05
('get', 'threaded') : 1 : 2.49642389898764e-06
('threaded', 'scared') : 1 : 2.49642389898764e-06
('scared', '</s>') : 2 : 2.245274240934343e-05
```

Sorted according to probability the top bigrams.



Word Unit	Count	Probability
('love', '</s>')	36	0.002126009
('<s>', 'new')	17	0.001551412
('<s>', 'back')	19	0.001149194
('<s>', 'finished')	19	0.001149194

Report the accuracy of the test set using dataset A for training.

Methodology:

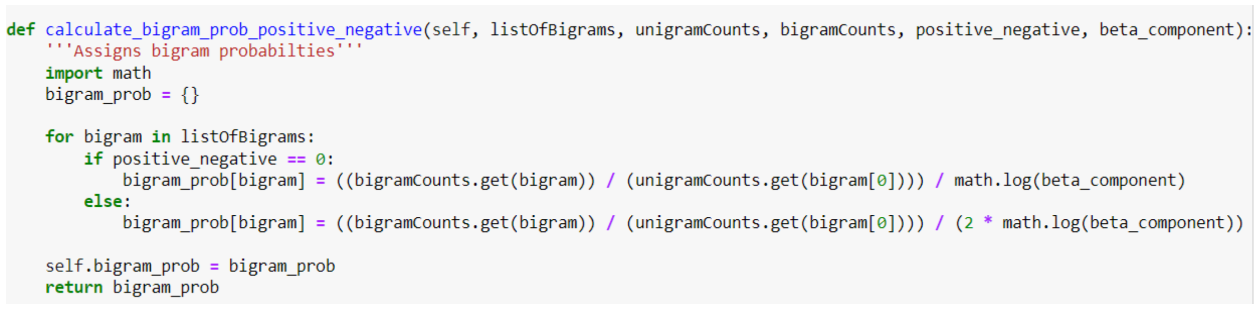
- CountVectorizer and Tfidf were used and tfidf gave better results.
- MultiNomial Naive Bayes used as the model.

90.22% is the accuracy on the test set using dataset A for training.

2. Part B

Mention in the report the method you tried and justify your solution.

Methodology:

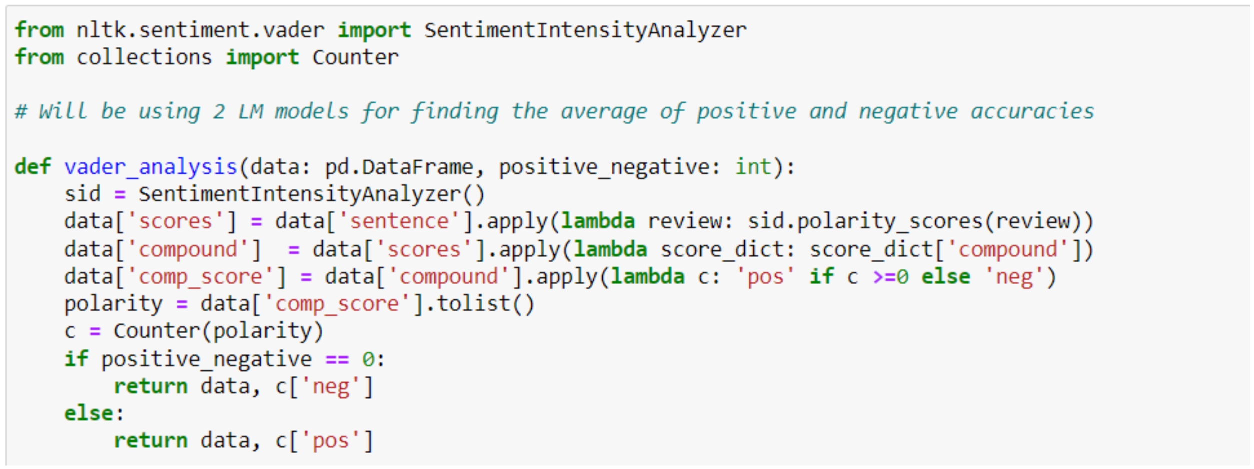


```
def calculate_bigram_prob_positive_negative(self, listOfBigrams, unigramCounts, bigramCounts, positive_negative, beta_component):
    '''Assigns bigram probabilties'''
    import math
    bigram_prob = {}

    for bigram in listOfBigrams:
        if positive_negative == 0:
            bigram_prob[bigram] = ((bigramCounts.get(bigram)) / (unigramCounts.get(bigram[0]))) / math.log(beta_component)
        else:
            bigram_prob[bigram] = ((bigramCounts.get(bigram)) / (unigramCounts.get(bigram[0]))) / (2 * math.log(beta_component))

    self.bigram_prob = bigram_prob
    return bigram_prob
```

For the beta component, a method is written in which we are using the natural log of the beta component and for generating the positive sentiment sentence we are dividing the bigram probability by the log of the beta component and for the negative sentiment sentences we are dividing the beta component by twice the log of beta component thereby penalizing the generation of negative sentences. Since we had to calculate the probability value hence we are dividing in both the cases and not multiplying or adding because doing so might push the probability value above 1 and we do not want that to happen.



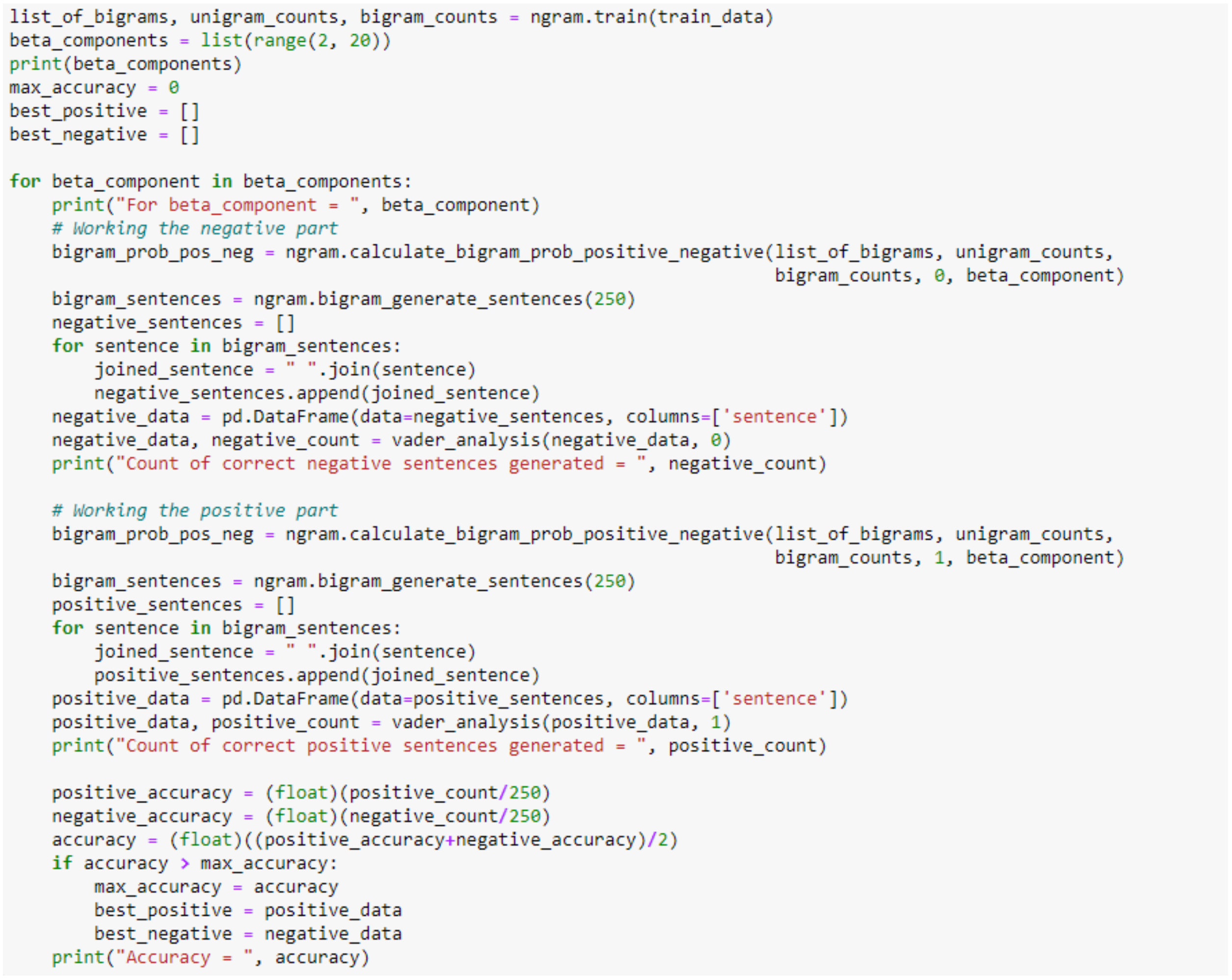
```python
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from collections import Counter

# Will be using 2 LM models for finding the average of positive and negative accuracies

def vader_analysis(data: pd.DataFrame, positive_negative: int):
    sid = SentimentIntensityAnalyzer()
    data['scores'] = data['sentence'].apply(lambda review: sid.polarity_scores(review))
    data['compound']  = data['scores'].apply(lambda score_dict: score_dict['compound'])
    data['comp_score'] = data['compound'].apply(lambda c: 'pos' if c >=0 else 'neg')
    polarity = data['comp_score'].tolist()
    c = Counter(polarity)
    if positive_negative == 0:
        return data, c['neg']
    else:
        return data, c['pos']
```

After calculating the modified bigram probabilities, we are generating 250 positive and 250 negative sentences and then using Vader Sentiment analysis for calculating two separate accuracies, one for the positive 250 sentences (checking that out of the required 250 positive sentences how many actual positive sentences were generated) and one for the negative 250 sentences, after which we are taking the average of the two accuracies as the final accuracy for a certain “beta_component”. We have applied the same thing for a range of beta_component values and then selected the one which provides us with the best accuracy.

This is the code we have used for finding the best beta component: -



```python
list_of_bigrams, unigram_counts, bigram_counts = ngram.train(train_data)
beta_components = list(range(2, 20))
print(beta_components)
max_accuracy = 0
best_positive = []
best_negative = []

for beta_component in beta_components:
    print("For beta_component = ", beta_component)
    # Working the negative part
    bigram_prob_pos_neg = ngram.calculate_bigram_prob_positive_negative(list_of_bigrams, unigram_counts,
                                                                        bigram_counts, 0, beta_component)
    bigram_sentences = ngram.bigram_generate_sentences(250)
    negative_sentences = []
    for sentence in bigram_sentences:
        joined_sentence = " ".join(sentence)
        negative_sentences.append(joined_sentence)
    negative_data = pd.DataFrame(data=negative_sentences, columns=['sentence'])
    negative_data, negative_count = vader_analysis(negative_data, 0)
    print("Count of correct negative sentences generated = ", negative_count)

    # Working the positive part
    bigram_prob_pos_neg = ngram.calculate_bigram_prob_positive_negative(list_of_bigrams, unigram_counts,
                                                                        bigram_counts, 1, beta_component)
    bigram_sentences = ngram.bigram_generate_sentences(250)
    positive_sentences = []
    for sentence in bigram_sentences:
        joined_sentence = " ".join(sentence)
        positive_sentences.append(joined_sentence)
    positive_data = pd.DataFrame(data=positive_sentences, columns=['sentence'])
    positive_data, positive_count = vader_analysis(positive_data, 1)
    print("Count of correct positive sentences generated = ", positive_count)

    positive_accuracy = (float)(positive_count/250)
    negative_accuracy = (float)(negative_count/250)
    accuracy = (float)((positive_accuracy+negative_accuracy)/2)
    if accuracy > max_accuracy:
        max_accuracy = accuracy
        best_positive = positive_data
        best_negative = negative_data
    print("Accuracy = ", accuracy)
```

Save the generated 500 sentences and their sentiment labels in a CSV. The final zip file should contain the code and output CSV for all the generation methods you mention in the report.

The 500 sentences generated have been saved in the “sentiment_sentences.csv” file along with their sentiment that was computed using vader.

Report the average perplexity of the generated 500 sentences.

Methodology:

- Done by calculating Maximum Likelihood Probability. A function is made for this.
- $p(w_i|w_{i-1}) = \frac{count(w_{i-1},w_i)}{count(w_i-1)}$
- Perplexity
- For bigram $l = \frac{1}{n}(logp(w_1) + logp(w_2|w_1) + \cdots + logp(w_n|w_{n-1}))$

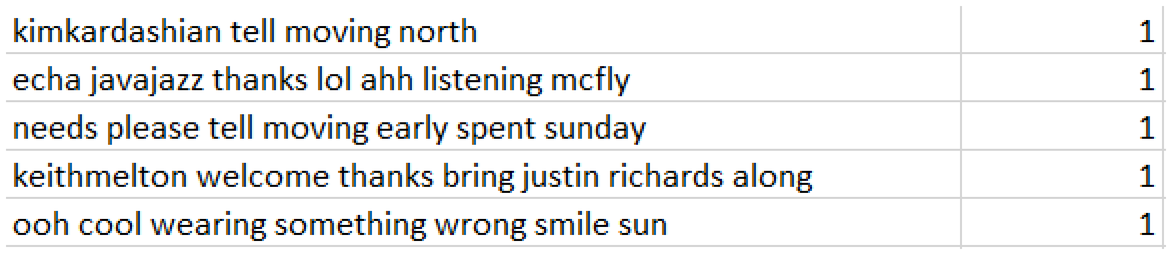
The average perplexity of the 500 generated sentences came out to be 1.0
For positive sentences = 1.0
For negative sentences = 1.0
The perplexity changes if we consider different values used in the code.

The average perplexity of Dataset A = 47.94826475553644

Report 10 generated samples: 5 positives + 5 negatives

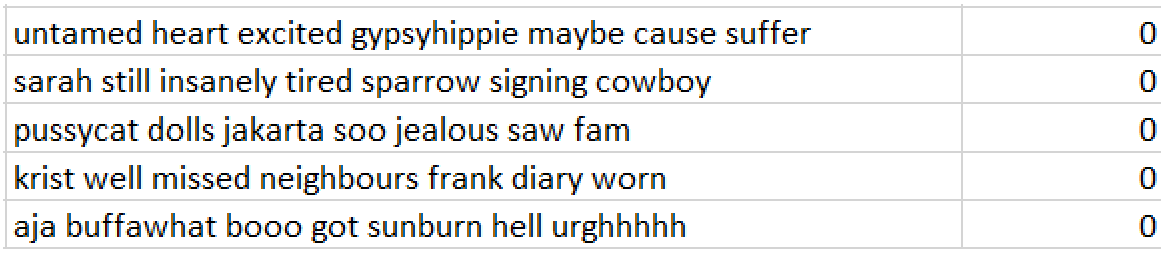
- The sentences have a minimum length of 7 initially after being pre processed (was required as it still contained some unwanted text which decreased the model accuracy) the length of the sentences changed!

Positive Sentences



kimkardashian tell moving north	1
echa javajazz thanks lol ahh listening mcfly	1
needs please tell moving early spent sunday	1
keithmelton welcome thanks bring justin richards along	1
ooh cool wearing something wrong smile sun	1

Negative Sentences



untamed heart excited gypsyhippie maybe cause suffer	0
sarah still insanely tired sparrow signing cowboy	0
pussycat dolls jakarta soo jealous saw fam	0
krist well missed neighbours frank diary worn	0
aja buffawhat booo got sunburn hell urghhhhh	0

Report the accuracy of the test set using dataset B for training.

92.24% is the accuracy on the test set using dataset B for training.
-Tfidf used with Multinomial NB.

Note: The accuracy for dataset B is higher than that of dataset A by just a percent.
After trying the improvement of accuracy it was seen that the processing part plays a role in the generation of sentences and that is why it has been done to a minimum.

The training accuracies of Dataset A and B are both not so good resulting in overfitting a bit.
That has not been tuned but can be improved by trying different models for instance.

Contributions:

1. Drishya Uniyal - Bigram Language Model that calculates everything.
2. Harshit Gupta - Generation of sentiment oriented sentences (beta component) and model MNB.

(Report was equally made by both)