CSE556: NATURAL LANGUAGE PROCESSING

ASSIGNMENT 2

SIDDHANT AGARWAL (2020247) and SHANTANU DIXIT (2020118)

ALL FILES SUBMITTED:
report.pdf - Project report detailing all methodology and answers
A2_2020247_2020118.ipynb - ipynb file for all the language modelling
Preprocessing.ipynb - ipynb file for the preprocessing of data
generated_sentences.csv - csv file containing 500 generated sentences and the Vader labels corresponding to them
senti_sentences_with_label.json - json file containing more information with tokens and intended sentiment labels for 500 generations
spell_checked.json - json file containing pre-processed data
Pkn.csv - csv file containing saved bigram smoothed model for non-sentiment generation
prob_pos.csv - csv file containing saved bigram smoothed model for negative-sentiment generation
prob_neg.csv - csv file containing saved bigram smoothed model for negative-sentiment generation
A1_dataset.csv - given training dataset A csv
A2_test_dataset.csv - given testing dataset csv

Preprocessing:
Sentences were tokenized,spaces and punctuations were removed,we found that there were a lot of unnecessary words in the corpus so spell check was necessary.For spell correction part pyspell" library is used.After preprocessing the vocab size reduced to half since a lot of unnecessary words were removed.
Preprocessing notebook as Preprocessing.ipynb

(A) Incorporate one of the smoothing algorithms (Default: Laplace)
Three different types of smoothing were developed namely Laplace,Add-k and Knesser Ney smoothing. Knesser ney was carried forward since it gave the best results.

```python
def laplace_smoothing(bigram_count,unigram_count):
    n = bigram_count.shape[0]
    laplace_smoothing = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            laplace_smoothing[i,j] = (bigram_count[i,j]+1)/(unigram_count[j] + n)
    return laplace_smoothing
```

```python
def add_k_smoothing(bigram_count,unigram_count,k=2):
    n = bigram_count.shape[0]
    addk_smoothing = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            addk_smoothing[i,j] = (bigram_count[i,j]+k)/(unigram_count[j]+k*n)
    return addk_smoothing
```

```python
def KnesserNeySmoothing(bigram_count,unigram_count,d,n):
    n = bigram_count.shape[0]
    # d = 0.5
    total_bigrams = np.count_nonzero(bigram_count)
    pcont = np.zeros(n)

    ## calculating P_continuation(w)
    for i in range(n):
        pcont[i] = (np.count_nonzero(bigram_count[:, i]))/total_bigrams

    ## calculating alpha
    alpha = np.zeros(n)
    for i in range(n):
        if i == word2idx['<END>']:
            continue
        alpha[i] = (np.count_nonzero(bigram_count[i, :]) * d) / (np.sum(bigram_count[i, :]))
    alpha[word2idx['<END>']] = d #to be thought

    ## calculating P_AD(wi|wi-1) = [max(count(w_i-1w_i)-d,0) / count(w_i-1)] + alpha(w_i-1)*P_cont(w_i)
    Pkn = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            Pkn[i,j] = (max(bigram_count[i,j]-d,0)/unigram_count[i]) + (alpha[i]*pcont[j])
    return Pkn
```

(B) Beta or Sentiment component for our best model:

We use an HMM type formulation with transition and emission probabilities. We formulate that we may consider our smoothed (using Knesser-Kney Smoothing) bigram probabilities as the transition probabilities and the use a novel formulation using the Vader Scores for each unigram as the emission probability. We define the emission to be

an N*2 matrix where N is the number of unique unigrams in our preprocessed data. There are two classes of Positive or Negative and for each unigram we define this emission probability as:

1. When a unigram appears in negative sentiment sentence in our data, we do the operation: emission(unigram, negative) += max(VADER_negative_score(unigram), 0.01)
2. When a unigram appears in positive sentiment sentence in our data, we do the operation: emission(unigram, positive) += max(VADER_positive_score(unigram), 0.01)

We then normalise each row of this emission matrix to sum to one.

We then use two different bigram probability matrices to produce positive and negative sentences separately. This is done with the formulation transition_probability * emission_probability for each bigram. We get two matrices, prob_pos[i, j] = Pkn[i, j] * emission[j, 1] and prob_neg[i, j] = Pkn[i, j] * emission[j, 0] which we then use for our generations.

Code Snippet -

```
# HMM BASED MODEL
# Emission probabilities with Vader
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
def HMMProbabilites(vocab, tokenized_sentences, data_list, Pkn)
    obj = SentimentIntensityAnalyzer()
    n = len(vocab)
    emission = np.zeros((n, 2))

    for i in range(len(data_list)):
        sentence = tokenized_sentences[i]
        for j in range(len(sentence)):
            if data_list[i][0] == 0:
                emission[word2idx[sentence[j]], 0] += max(obj.polarity_scores(sentence[j])['neg'], 0.01)
            else:
                emission[word2idx[sentence[j]], 1] += max(obj.polarity_scores(sentence[j])['compound'], 0.01)
    for i in range(len(emission)):
        emission[i] = emission[i]/emission[i].sum()
    emission[word2idx['<END>']] = [1, 1]
    prob_pos = np.zeros((n, n))
    prob_neg = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            prob_pos[i, j] = Pkn[i, j] * emission[j, 1]
            prob_neg[i, j] = Pkn[i, j] * emission[j, 0]
    for i in range(n):
        if (prob_pos[i, :].sum() == 0):
            print(i)
        prob_pos[i, :] = prob_pos[i, :]/prob_pos[i, :].sum()
        prob_neg[i, :] = prob_neg[i, :]/prob_neg[i, :].sum()
    return prob_pos, prob_neg
```

(C) Yes,we can extend our solution to generate positive/negative sentences only. Using our emission probabilities we can control that we make positive only or negative only sentences and use sample from the distribution which have the positive and negative factors included using the emission probabilities.

(D) Vader sentiment score to obtain labels:
Following function runs through all the sentences generated and creates a dataframe which stores the labels given by VADER

```
## getLabels return 500 sentences in a dataframe with a corresponding label from VADER
def getLabels(sentences):
    gen = []
    for i in range(len(sentences)):
        sentence = sentences[i][0][1:-1]
        sentence = ltos(sentence)
        tried_label = sentences[i][1]
        lbl = sentimentAnalyze(sentence)
        gen.append((sentences[i][0],sentence,tried_label,lbl))
    Generated = pd.DataFrame(gen,columns=['Tokens','Sentences','LABEL','VADER_LABEL'])
    return Generated
```

Evaluation:

1.Intrinsic evaluation:
Function from scratch to calculate perplexity of the sentence (Pkn stores the smoothed probabilities obtained after applying KnesserNey smoothing,word2idx converts a particular word to its

index as in the vocabulary developed)

[28]:
```
def avgPerp_senti(txtlist,pos, neg,word2idx):
    sentence_count = len(txtlist)
    perplexity_sum = 0
    c = 0
    for i in range(sentence_count):
        if txtlist[i][1] == 0:
            perplexity_sum = perplexity_sum + perplexity(txtlist[i][0],neg,word2idx)
        else:
            perplexity_sum = perplexity_sum + perplexity(txtlist[i][0],pos,word2idx)
    avg_perplexity = perplexity_sum/sentence_count
    return avg_perplexity

def perplexity(txt,Pkn,word2idx):
    length = len(txt)
    prob = 1
    for i in range(0,length-1):
        prob = prob * Pkn[word2idx[txt[i]],word2idx[txt[i+1]]]
    perp = math.pow(1/prob,1/length)
    return perp
```

2.Extrinsic evaluation:

We use the given code for a Naive Bayes classification for the extrinsic evaluation. We were able to beat Dataset A using our Dataset B using thisNaive Bayes ML Model.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.metrics import accuracy_score
def train_and_evaluate(train_sentences, train_labels,
    test_sentences, test_labels):
    '''
    parameters:
    train_sentences : list of training sentences
    train_labels : list of training labels
    test_sentences : list of test sentences
    test_labels : list of test labels
    output:
    accuracy : accuracy of the test set
    '''
    # Model building
    model = make_pipeline(TfidfVectorizer(), MultinomialNB())
    # Training the model with the training data
    model.fit(train_sentences, train_labels)
    # Predicting the test data categories
    predicted_test_labels = model.predict(test_sentences)
    return accuracy_score(test_labels, predicted_test_labels)
```

Part A

a. Save smoothed bigram language model:

Please find the smoothed probabilities for our best language model as:

Pkn.csv for the Knesser-Ney smoothed probabilities before adding sentiment component

prob_pos.csv for the probabilities used for probabilities for positive sentence generation after considering sentiment component

prob_neg.csv for the probabilities used for probabilities for negative sentence generation after considering sentiment component

Please also find word2idx.json and idx2word.json which are dictionary files corresponding to the indexing of the probability matrix with respect to our vocabulary.

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

We have separate probabilities for non-sentiment, positive sentiment and negative sentiment generations so we are providing top-4 bigram and their score for all:

For non-sentiment:

Score: 0.991804939480732 Bigram: gon na
Score: 0.9875025327081163 Bigram: wan na
Score: 0.9540858024349025 Bigram: ca it
Score: 0.928879488579037 Bigram: waking up

For positive sentiment:

Score: 0.9898043489761318 Bigram: gon na
Score: 0.984468080312772 Bigram: wan na
Score: 0.9480645234701405 Bigram: ex <END>
Score: 0.9317716927306368 Bigram: ca it

For negative sentiment:

Score: 0.9926733442190903 Bigram: gon na
Score: 0.9888217115030428 Bigram: wan na
Score: 0.9699021422237434 Bigram: ca it
Score: 0.9251765285324683 Bigram: waking up

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

= 0.9254658385093167

```
#Extrinsic Evaluation using Dataset A
train_and_evaluate(train_sentences_A, train_labels_A,test_sentences, test_labels)
```

```
0.9254658385093167
```

PART B

(A)

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

The method we tried is using an HMM approach. It is described in detail earlier in the report about how to include sentiment.

Our method involves smoothing using the Knesser-Ney smoothing technique. We specifically use this technique because it provided the best results and the sentences carried some meaning and were not merely a random collection of words which was the case when we were applying smoothing using Laplace Smoothing or Add-K smoothing.
We then use an HMM based approach for our formulation which is described earlier. In short, it allows us to include the sentiment component at a unigram level and that changes the bigram probabilities based on whether we want to generate a positive or negative sentence.
Then, in our generation, for each sentence we randomly choose if we want a positive or negative sentiment and then sample each word from the positive or negative bigram probability distributions until we reach the <END> token. These generations help us add to the dataset in a meaningful way performing well in both the intrinsic and extrinsic evaluation.

(B)
Please find the best attempt generated sentences saved as generated_sentences.csv. We also have another json file corresponding to this generation with the name senti_sentences_with_label.json to preserve more information about this attempt for our usage purposes.

(C)
Report the average perplexity of the generated 500 sentences.
= 135.76769921750366

```
# Intrinsic evaluation using perplexity
avgPerp_senti(sentisent,prob_pos, prob_neg,word2idx)
```

```
135.76769921750366
```

Perplexity Code:

```
def avgPerp_senti(txtlist,pos, neg,word2idx):
    sentence_count = len(txtlist)
    perplexity_sum = 0
    c = 0
    for i in range(sentence_count):
        if txtlist[i][1] == 0:
            perplexity_sum = perplexity_sum + perplexity(txtlist[i][0],neg,word2idx)
        else:
            perplexity_sum = perplexity_sum + perplexity(txtlist[i][0],pos,word2idx)
        ## print(i," ",perplexity_sum)
        c+=1
        ## print(perplexity_sum/c)
    avg_perplexity = perplexity_sum/sentence_count
    return avg_perplexity
def perplexity(txt,Pkn,word2idx):
    length = len(txt)
    prob = 1
    for i in range(0,length-1):
        prob = prob * Pkn[word2idx[txt[i]],word2idx[txt[i+1]]]
    perp = math.pow(1/prob,1/length)
    return perp
```

(D)

Report 10 generated samples: 5 positives + 5 negatives:

Negative Samples:

1. if i thinking be an hour if i was medic sigh our overtime his going with a ball soooo to head off to see turn right now sobs
2. is scrambling my in the quit i work tonight comes in the house believe the time for me your now is goons be sad i hate when will be a migraine you tweet i have
3. me about the shower now everyone unfortunate combination fans old u ass i not in the times him 2 work wont it online no dead crash
4. talks tired but when i had to stay in the dentist ca it when you needs to the day in nodding off you re but alas i sad
5. sorry sick again maybe in the impossible the mail yet at work i everyone myself that sucks sniff clouds to work my mouse in one

Positive Samples:

1. ok my breakfast for topics i new good drink in there how goes 2 go away and funny on twitter haha i do i excited about
2. loving the cuz archie i want sure update itself everyday being so bow adorable help mar fan love it wait you know why do now
3. luke why it 3 thanks honey actually i see i have a great victorious fully enjoy 09 happened to next we is going back home more
4. why do it like everybody excited for you re really dont know when i was actually i a poet me i shall loved set nice it win
5. good fairy today it at a few quit lens theme help tweets i ever spreading the latest greatest girl is mario part of me a day

(E)
Report the accuracy of the test set using dataset B for training:
= 0.9285714285714286

```
# Extrinsic Evaluation using Dataset B
train_and_evaluate(train_sentences_B, train_labels_B,test_sentences, test_labels)
```

```
0.9285714285714286
```

We observe an increase of 0.3% in the accuracy using our Dataset B with 500 more generated sentences and corresponding labels using Vader.
Another Observation -
In our analysis we have observed that our performance is lagging due to the inaccuracy of Vader in detecting the sentiment. If we use our own labels that we are using to generate the sentences instead of the Vader generated sentences, then we get a 93.63% accuracy which is a whole 1.2% increase in accuracy over our Dataset A which is very meaningful for a model which is already performing at a very high accuracy of 92.54%

```
# To highlight inaccuracy of Vader
train_sentences_B = np.concatenate((train_sentences_A, sd[:, 1]))
train_labels_B = np.concatenate((train_labels_A, sd[:, 2].astype(np.int8)))
train_and_evaluate(train_sentences_B, train_labels_B,test_sentences, test_labels)
```

```
0.9363354037267081
```

CONTRIBUTIONS:

Siddhant Agarwal 2020247 - EQUAL CONTRIBUTION

Shantanu Dixit 2020118 - EQUAL CONTRIBUTION

- constant discussion and inputs for all parts by both team members.