CSC411: Supervised and Unsupervised Learning for Sentiment Analysis

Yoshiki Shoji & **Zi Mo Su**

March 21, 2017

Corollary

Before we start with executing the code, we must obtain the data for creating the training, test and validation sets. We recall that the training set is used in contructing our initial hypothetical algorithm of interest, the validation set used for tuning the hyperparameters, and the test set used at the end (untouched until this very moment) to evaluate our performance. Now, in order to prevent such overfitting, and using the requirement that we will use 100 movie reviews for each set and making everything lower case and punctuation ridden, etc. we will first use a helper function which is given below:

```
def preprocess(line):
    """
    This function preprocesses a single line in a file by removing
    punctuation and extra spaces.
    :param line: (string) line in file
    :return: (string) processed line
    """
    # remove punctuation
    line = re.sub(r'(\W)', ' ', line)

    # remove extra spaces including at beginning and ending of sentence
    line = re.sub(r'\s+', ' ', line)
    line = re.sub(r'\s$', '', line)
    line = re.sub(r'^\s', '', line)

    # make all words lowercase
    line = line.lower()

    return line
```

With the helper function constructed, all that is left is to append each review into an empty array. During this process is when the all the sets are also made. We use the random permutation function which outputs an array of sequence of numbers from 11000 without any repetitive values. In order to ensure that there is no intersection of data between these sets, the test will use the first 100 numbers from the permutation, the validation set will use the next 100 numbers, and the remaining data used for the training set. The code is given below:

```
def partition(dir):
    """
    This function partitions all reviews in a directory into a training set (size 800),
    test set (size 100) and validation set (size 100).
    :param dir: (string) name of directory
    :return: (lists) lists of strings containing reviews for each set
    """
    # random permutation
    rand = np.random.permutation(1000)
    data = []

    # loop through all files
    for filename in os.listdir(dir):
        input_file = open(os.path.join(dir, filename))

        # process each line in current file and append as string
```

```
        processed = ''
        for line in input_file:
            processed += ' ' + preprocess(line)

        data += [processed]
        input_file.close()

    data = array(data)

    # partition into test, validation and training set using random permutation
    test = data[rand[:100]]
    val = data[rand[100:200]]
    train = data[rand[200:]]

    return test, val, train
```

Part 1

To describe the data set, we will consider one negative review example

```
    'the most interesting part of can t hardly wait just happens to be not on
ly the most human but for many of us the one part that many of us can easily rel
ate to that is the character of denise lauren ambrose the film s sole sarcastic m
ember who mocks everything that goes on in the film and at one point sits down o
n a couch and looks totally bored the film wisely holds over this moment nicely s
howing her alienation in the midst of a large high school party almost too nicely
for some members of the audience read me this is basically a mirror of what s go
ing on with them watching this film we sit there wondering why we ve even bothere
d to see a film about a long high school party we probably never felt the desire
 to go to in the first place i would actually...',
```

As we are required to evaluate positive or negative reviews based on keywords that show up in each review, we will suggest the plausibility of this by first considering how likely a word is going to show up in such sets. Consider the function below:

```python
def get_counts(set):
    counts = {}
    total_count = 0.0

    for review in train:
        counted = []
        words = review.split()
        for word in words:
            if word in counted:
                continue

            if word not in counts:
                counts[word] = 0

            counts[word] += 1.0/len(words)
            total_count += 1.0/len(words)

            counted += [word]

    return counts, total_count
```

This function takes in the set = {training,validation,text} and computes the the total word count by first initializing the dictionary called counts. Notice that every word has been split accordingly, and made into an list. All that is now left to do is to keep on updating. For every new word that is found, we append it to the dictionary. If the word already exists, add one every time. To find the likelyhood with respect to all the words that exist in the set, divide it by the length of the words array. An example is shown below:

```
{'blondie': 0.002570694087403599, 'optical': 0.0019305019305019305, 'r
ifle': 0.0015220700152207, 'convinced': 0.0013550135501355014, 'filmin': 0.00422
6972587207363, 'africaso': 0.002544529262086514}
```

To obtain the most frequent word within the developed dictionary, we consider getting all the key values and sort the computed list, using the built in sorting python function. The code and a table of words is given along with how frequent is appears.

```
#computing the three most frequenct words:
def freq_words(set):
    counts_dic = get_counts(set)[0]
    counts_val = sorted(counts_dic.values(), key=float, reverse=True)
    for i in range(3):
        print counts_val[i], counts_dic.keys()[counts_dic.values().index(counts_val[i])]
    return
```

Table 1: Postive Review

Word	Frequency Count
with	0.175905582436
to	0.182615311974
the	1.44206111808

Table 2: Negative Review

Word	Frequency Count
and	0.255394675529
is	0.251635277033
to	1.63902426407

Part 2

We first make a reasonable assumption that all words are independent. Denote, "w_i" as a word from a set of all possible words, **W**. It follows that our variable probability distribution becomes:

$$P(\mathbf{W}) = P(w_i, w_i, w_i, ..., w_n)$$

$$P(\mathbf{W}) = P(w_i)P(w_i) \cdots P(w_n)$$

With the above formula, we condition it with another variable. In our case the variable is a class, where class = {negative,positive} depending on the review. By using Bayes and conditional probability, we are left with

$$P(\mathbf{W}|class) = P(w_1|class)P(w_2|class) \cdots P(w_n|class)$$

This then becomes

$$P(class|\mathbf{W}) = P(class) \prod_i P(w_i|class)$$

For which we claim that it must compute the likelyhood of the class of interest. As we have only two classes, positive or negative, then we must have $P(class)$ to be 1/2. Keeping this in mind, and by using the given approximations to evaluate our probability, the equation boils down to

$$P(class|\mathbf{W}) = P(class) \prod_i \frac{(count(w_i), class)}{count(class)}$$

In order to prevent the probability to be of zero when the word is not found in the training set, we make the modification of setting up two conditioned statements for each case. Hence, we refine the function above to:

$$P(w_i|class) = \begin{cases} \frac{(count(w_i), class)}{count(class)}, when\ word\ exists \\ \frac{(count(w_i), class) + mk}{count(class) + k},\ otherwise \end{cases}$$

As we are already able to make the respective counts due to the count function created in Part 1, we are then able to assign the probability, $P(w_i|class)$ for the negative and positive validation sets we have also created in Part 1. We notice further that whenever we store such values, we take the $\log(P(w_i|class))$, as by the end we take the product of all the summed log probabilities, and hence, to obtain the proper expression all that is left to do is take the exponential. Hence, to give a general expression we formalize this to

$$P(class|\mathbf{W}) = \frac{1}{2} exp(\sum_i log(P(w_i|class))$$

However, we notice that the summation of the log terms become a very high negative number, and taking the exponentiation results in a value of zero. In order to prevent this from happening, we do not take the exponentiation at the end, but compare the logarithmic scaled version of the probabilities. Hence, we conclude in using:

$$P(class|\mathbf{W}) = \frac{1}{2}(\sum_i log(P(w_i|class))$$

Realizing all the above expressions into code, we obtain:

```
def classify(review, neg_probs, pos_probs, neg_total, pos_total, m, k):
    neg_prob = 0.0
    pos_prob = 0.0

    for word in review.split():
        if word in neg_probs:
            neg_prob += log(neg_probs[word])
        else:
            neg_prob += log((m * k)/(neg_total + k))

        if word in pos_probs:
            pos_prob += log(pos_probs[word])
        else:
            pos_prob += log((m * k)/(pos_total + k))

    # print 'neg:', neg_prob
    # print 'pos:', pos_prob

    # final probability [product of conditionals P(word|class)] * [P(class)]
    neg_prob = 0.5*(neg_prob)
    pos_prob = 0.5*(pos_prob)

    if neg_prob > pos_prob:
        return 'neg'
    else:
        return 'pos'
```

Observing the code above, we have in input parameter "review". This will be our validation set for tuning the hyperparameter "m" and "k" at during refinement stage. The neg_probs and the pos_prob will be from the two training sets that are created which contain the negative and positive reviews. All that is left to do in the end is to compare the two probabilities. We return the value that is higher.

As the prior function may now help us evaluate whether Naive Bayes computes the proper positive or negative test review, we consider the function below which evaluates its performance:

```
def get_performance(neg_probs, pos_probs, neg_val, pos_val, neg_total, pos_total, m, k):
    neg_correct = 0.0
    neg_wrong = 0.0
    pos_correct = 0.0
    pos_wrong = 0.0

    for review in neg_val:
        c = classify(review, neg_probs, pos_probs, neg_total, pos_total, m, k)
        if c == 'neg':
            neg_correct += 1
        else:
            neg_wrong += 1

    for review in pos_val:
        c = classify(review, neg_probs, pos_probs, neg_total, pos_total, m, k)
        if c == 'pos':
```

```
                pos_correct += 1
            else:
                pos_wrong += 1

    neg_perf = neg_correct/(len(neg_val))
    pos_perf = pos_correct/(len(pos_val))

    perf = (neg_correct + pos_correct)/(len(neg_val)+len(pos_val))
    return neg_perf, pos_perf, perf
```

Observing the code above, we separate performance evaluations into two stages. We will use Naive Bayes to obtain the performances for the positive and negative reviews separately, by using two validations set which only contain negative or positive reviews. For the negative review, if the classifier correctly returns the correct classification, all that is left to do is to keep dynamically updating an initialized variable which will store how many times the classifier was able to correctly identity its classification. This is the same for the positive review. To then evaluate the overall performance, we add the neg_perf and pos_perf variables and divide it by the length of the the two validation sets.

To tune the parameters, we initialize an array for some choice of guesses we make for "m" and "k" parameters. We then iterate through all possible values, by fixing one value and iterating through the all the choices in the other array which will evidently create a nested for loop. The code is given below:

```
def find_best_params(m_s, k_s):
    best_perf = 0

    neg_counts, neg_total = get_counts(neg_train)
    pos_counts, pos_total = get_counts(pos_train)

    for m in m_s:
        for k in k_s:
            neg_probs = {key: (val + m * k)/(neg_total + k)
                         for key, val in neg_counts.items()}
            pos_probs = {key: (val + m * k)/(pos_total + k)
                         for key, val in pos_counts.items()}

            neg_perf, pos_perf, perf =
            get_performance(neg_probs, pos_probs, neg_val,
                            pos_val, neg_total, pos_total, m, k)

            print 'Test using m =', m, 'and k =', k
            print 'Classification performance:'
            print '\tNegative:', neg_perf
            print '\tPositive:', pos_perf
            print '\tTotal:', perf
            print

            if perf > best_perf:
                best_perf = perf
                best_m = m
                best_k = k

    print 'Best parameters found to be m =', best_m,
          'and k =', best_k, 'with performance of', best_perf
```

```
    return best_m, best_k
```

Hence, the best parameters were found to be m = 0.05 and k = 0.001 with performance of 0.79
Please refer to Appendix A for this part's code.

Part 3

In order to compute the 10 words that most strongly predict whether a review is positive or negative in terms of conditional probability, we first realize that in Part 1 a dictionary was created with all the key-value pairs being word-count in our case. From Part 2 we further have the equation:

$$P(w_i|class) = \begin{cases} \frac{(count(w_i),class)}{count(class)}, when\ word\ exists \\ \frac{(count(w_i),class)+mk}{count(class)+k},\ otherwise \end{cases}$$

As we are only dealing with classes individually, we can make simplifications and consider the top 10 words which have the highest count value from observing the separately made from training set of positive and negative reviews. First however, we refine the function made in Part 1 as we recognize that some words may have the same value, and hence whenever we append this to an initialized empty array, we must pop this word from the dictionary as other words may have the same value -this will prevent resulting in an output having duplicate words. Furthermore, as we require words that strongly suggest whether a word will give positive or negative results, we will then consider the intersection of the two sets, as we get rid of words that are shared among the two dictionaries. The code is given below:

```
def freq_words(pos_train,neg_train):
    pos_words = []
    neg_words = []
    counts_pdic = get_counts(pos_train)[0]
    counts_ndic = get_counts(neg_train)[0]
    #obtain the intersection between the two dictionaries and delete them
    #as we are interested in the unique words that evaluate them as positive
    #or negative
    intersection = counts_pdic.viewkeys() & counts_ndic.viewkeys()
    for word in intersection:
        #print(word)
        if word in counts_pdic:
            counts_pdic.pop(word)
        if word in counts_ndic:
            counts_ndic.pop(word)

    #arrange the count values from highest to lowest
    counts_pval = sorted(counts_pdic.values(), key=float, reverse=True)
    counts_nval = sorted(counts_ndic.values(), key=float, reverse=True)
    for i in range(len(counts_pval)):
        pword = counts_pdic.keys()[list(counts_pdic.values()).index(counts_pval[i])]
        nword = counts_ndic.keys()[list(counts_ndic.values()).index(counts_nval[i])]
        #delete every word that will be appended as some count
        #values may be the same
        counts_pdic.pop(pword)
        counts_ndic.pop(nword)
        pos_words.append(pword+' '+str(counts_pval[i]))
        neg_words.append(nword+' '+str(counts_nval[i]))
        #stop when we have the first 10 values
        if len(pos_words) == 10 and len(neg_words) == 10:
            break
    return pos_words,neg_words
```

The function above gives the output as given below:

words strongly suggesting positive review: 'seamless 0.0199874286776', 'recalls 0.0194209938417', 'lovingly 0.0193841051974', 'weaknesses 0.0172663481827', 'criticized 0.017263678491', 'thematic 0.0169752591167', 'addresses 0.0162000412903', 'missteps 0.0156159748228', 'splitting 0.0150743245404', 'melancholy 0.0149814062452'

words strongly suggesting negative review: 'degenerates 0.0223864911759', 'preston 0.0192061433719', 'predator 0.0175067096689', 'bio 0.0169512923922', 'suvari 0.0164502479508', 'mena 0.0164502479508', 'jumbo 0.0158376345728', 'bruckheimer 0.0157167650169', 'amateur 0.0154177945277', 'leaden 0.0148876635441'

Part 4

A logistic regression model was created using Tensorflow. The network consists of:

1. Input x_i, $i = 1...k$ and bias b.
2. Input classification y_-, $y_- \in 0, 1$.
3. Output y, $0 < y < 1$.

First, the data was split into a training set containing 800 reviews from each class, and test and validation sets containing 100 reviews from each class, respectively. This was done randomly.

A list of k words was then generated from the training data, where k is the number of unique words in the entire training corpus. An input vector x for a review r is k-dimensional, and each element is either 0 or 1 depending on whether the i^{th} word in our list of unique words exists in r.

The classification input is 1 if the review is negative and 0 if the review is positive.

The network has a single output y, which uses the sigmoid activation function $\sigma(t)$ to perform logistic regression. For $y \geq 0.5$ we classify the review as negative. For $y < 0.5$ we classify the review as positive.

$$y = \sigma(t) = \frac{1}{1 + exp(-t)}$$

Here, we feed in a linear combination of our inputs weighted by w_i:

$$t = b + \sum_{i=1}^{k} w_i x_i$$

The cost function we use is:

$$C = \sum_{j=1}^{m} [y_{\text{-}j} log(y_j) + (1 - y_{\text{-}j}) log(1 - y_j)] + \lambda \sum_{i=1}^{k} w_i^2$$

where m is the number of reviews in our training corpus.

The network was trained with varying values for λ. First a wide range of λ were used from 0.00001 to 1000 (orders of magnitude). The result showed values ranging between 0.1 and 100 worked best, so a reduced set was used.

The validation set was used to tune the hyperparameter λ. The performance of the classifier was tested for different values of λ. The maximum performance was taken over 150 iterations and the λ that resulted in the best performance for the validation set was selected. This value was then used for testing on the test, validation and training sets. The value for λ that yielded the best validation performance was $\lambda = 5$. The performance is shown below in Figure 1. The performance of the test, validation and training sets reach 82.0%, 86.5% and 100.0%, respectively.

Please refer to Appendix B for this part's code.

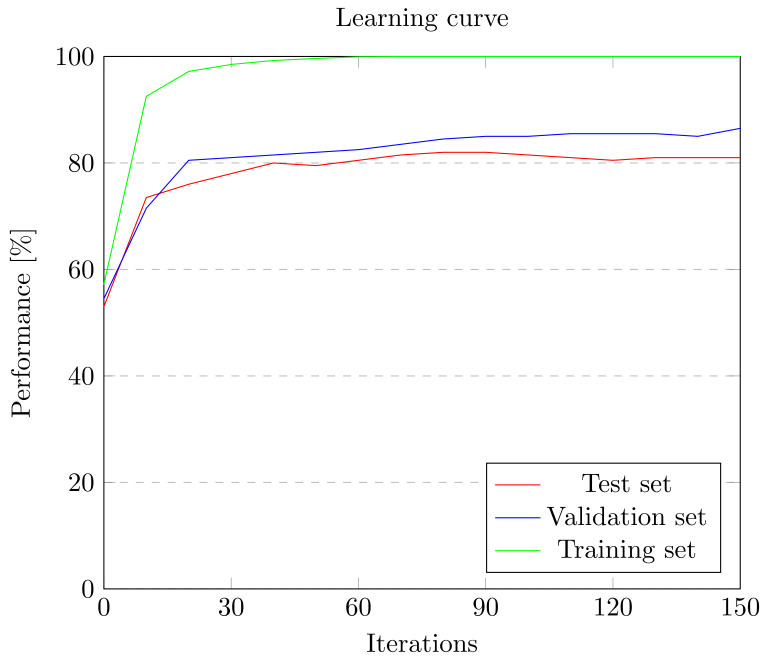
Learning curve
100
80
60
40
20
0
Performance [%]
0
30
60
90
120
150
Iterations
Test set
Validation set
Training set

Figure 1

Part 5

Both *Naive Bayes* and *Logistic Regression* can be formulated as computing:

$$\theta_0 + \theta_1 I_1 + ... + \theta_k I_k = \sum_{i=0}^{k} \theta_i I_i > thr$$

For *Naive Bayes*, we have a probability dictionary which stores the probability of a word given the class (negative or positive) as determined through training and adjustable parameters m and k (k here is the hyperparameter). The *logarithm* of this probability is taken as to operate in a computationally permissible range. Under the Naive Bayes assumption, the sum of these log probabilities yields the total probability for a review given the class. To classify, we compare the results of both classes, and the one with the greater probability is chosen. If we subtract the probability given positive from that given negative, we have a threshold value thr of 0. If the result is greater than zero then we will classify the review as negative. Otherwise, it will be classified as positive.

$$\sum_{i=1}^{k} log(Pr(x_i|negative)) + log(Pr(negative)) - \sum_{i=1}^{k} log(Pr(x_i|positive)) - log(Pr(positive)) > 0$$

$$log\left(\frac{Pr(negative)}{Pr(positive)}\right) + \sum_{i=1}^{k} log\left(\frac{Pr(x_i|negative)}{Pr(x_i|positive)}\right) > 0$$

We can generalize our expression to be applicable for all words in the training set, which is the case for *Logistic Regression*, by simply setting $I = 0$ for words that do not occur in the review we are testing. We note that in our implementation of *Naive Bayes*, we would not be looking at the set of unique words in the training set, but in all three sets, since there is a default probability defined by the hyperparameters m and k (i.e., in *Logistic Regression*, words that are not in the training set are simply ignored, whereas in *Naive Bayes*, words that are not in the training set are assigned a probability).

As such, for *Naive Bayes*, we have:

1. θ_i is a function which returns the log of the probability quotient as shown above, where x_i is the i^{th} word in the review for $i = 1...k$. For $i = 0$, the term is the log of of the probability of the negative class divided by that of the positive class.

2. I_i is a Boolean function which returns $True$ if the i^{th} unique word is present in the review and $False$ if it is not present in the review.

For *Logistic Regression*, we have a neural network that has $k + 1$ inputs x_i for $i \in 0...k$, and a single output y. Here, k is not the number of words in the review, but the count of all unique words in the training set. A linear combination of the inputs i.e., $\sum_{i=0}^{k} w_i x_i$ is fed into the sigmoid activation function at the output neuron to perform logistic regression. The output y is thus between 0 and 1. The classification input to the network is 1 if the review is negative and 0 if the review is positive. Thus, we determine the threshold thr to be 0.5. If a review has an output of 0.5 or above it will be classified as negative, and if it is below 0.5 it will be classified as positive.

$$b + \sum_{i=1}^{k} w_i x_i > 0.5$$

As such, for *Logistic Regression*, we have:

1. θ_i is the i^{th} weight w_i, where w_0 is the bias b.
2. I_i is a Boolean function which returns $True$ if the i^{th} unique word is present in the review and $False$ if it is not present in the review.

Part 6

The top 100 θs are taken from both the *Naive Bayes* and *Logistic Regression* models.

For *Naive Bayes*, the value of $log\left(\frac{Pr(x_i|negative)}{Pr(x_i|positive)}\right)$ was found for all words in the training set. Words corresponding to the top 100 θs for *Naive Bayes*:

{*'horrendous', 'seagal', 'tolerable', 'ludicrous', 'uninvolving', 'incompetent', 'sucks', 'unintentional', 'forgot', 'wasting', 'rabid', 'blah', '3000', 'unfocused', 'inspire', 'goofiness', 'thumb', 'mena', 'suvari', 'azaria', 'skimpy', 'fairness', 'predator', 'annoyingly', 'ditto', 'magically', 'climb', 'beware', 'welles', 'marginal', 'ahem', 'hewitt', 'incoherent', 'insulting', 'interminable', 'schumacher', 'unexciting', 'stupidity', 'recycles', 'readily', 'uninspired', 'pen', 'numbingly', 'inept', 'conspirator', 'warrant', 'headache', 'cancer', 'asset', 'justin', 'sober', 'bursts', 'lame', 'brained', 'illusion', 'stink', 'henchmen', 'eszterhas', 'sexist', 'unwatchable', 'scratching', 'digging', 'nonsense', 'jingle', 'turkey', 'waste', 'unimaginative', 'poorer', 'whatsoever', 'insufferable', 'um', 'idiots', 'compensate', 'omar', 'liu', 'coyote', 'sugary', 'lumet', 'idiotic', 'painfully', 'stamped', 'acrobatics', 'yell', 'robinson', 'magoo', 'lecture', 'blink', 'bruckheimer', 'squabble', 'downhill', 'wrongfully', 'wreaking', 'charmless', 'laborious', 'climbing', 'laser', 'davidson', 'avengers', 'raids', 'insipid'*}

We see that the θs are large when $\frac{Pr(x_i|negative)}{Pr(x_i|positive)}$ is large. This would result for large $Pr(x_i|negative)$ and small $Pr(x_i|positive)$. Thus if a word is likely to appear in a negative review but unlikely to appear in a positive review, it will be high in the top 100 list. Looking at the yield, words like 'horrendous', 'incompetent' and 'sucks' are all near the top. These are expected as they will likely be present in negative reviews and not present in positive ones.

For *Logistic Regression*, the weights w_i were used. Words corresponding to the top 100 θs for *Logistic Regression*:

{*'bright', 'sloppy', 'given', 'somewhere', 'under', 'center', 'ludicrous', 'cat', 'shown', 'presence', 'video', 'designed', 'depressing', 'supposed', 'protagonist', 'waste', 'biggest', 'credits', 'ridiculous', 'garbage', 'proceedings', 'boring', 'numbers', 'unless', 'grade', 'read', 'horrible', 'expected', 'awake', 'writer', 'audience', 'unfortunately', 'mess', 'wouldn', 'funeral', 'barrage', 'vehicle', '1', 'abilities', 'impression', 'falls', 'idea', 'watchable', 'across', 'performer', 'settle', 'nudity', 'maybe', 'couldn', 'fat', 'bland', 'cheap', 'generic', 'profanity', 'hopeless', 'randy', 'idiotic', 'poor', 'fails', 'showed', 'promise', 'embarrassment', 'looked', '4', 'anywhere', 'zeta', 'tries', 'laughable', 'twice', 'conflict', 'lacks', 'worse', 'chick', 'infamous', 'ms', 'save', 'misguided', 'context', 'sitcom', 'bigger', 'we', 'no', 'purpose', 'predictable', 'interesting', 'problems', 'adams', 'catch', 'stupid', 'skin', 'unbearable', 'trailer', 'rent', 'jack', 'subplot', 'nowhere', 'worst', 'suffers', 'weak', 'alas'*}

We see that the θs are large when the weights are large. A large weight refers to a neuron that is heavily activated. Heavily activated neurons correspond to words that have high likelihood of being negative. This is the case because the input is 1 if a word exists and the classification input is 1 if the review is negative. As such, inputs of 1 must have a heavy weight to classify as negative. Similarly, inputs of 1 must have a small weight to classify as positive. So words that occur commonly in negative reviews and rarely in positive reviews will have larger θ values. We see that words like 'sloppy', 'depressing' and 'garbage' occur in our yield, which supports this claim.

Please refer to Appendix C for this part's code.

Part 7

To determine the effectiveness of *word2vec* we trained a logistic regression model using:

1. Input x_i, $i = 1...256$ and bias b.
2. Input classification $y_{\_}$, $y_{\_} \in 0, 1$.
3. Output y, $0 < y < 1$.

We consider two words, w and t. Using our model we input $word2vec(w)$ and $word2vec(t)$ into the network. Each of these are of size 128, we concatenate them to form our 256 input vector, where $word2vec(w)$ is the first 128 elements and $word2vec(t)$ is the next 128 elements.

Our classification input is 1 if w occurs before t in our training set. We train our model by using a training set consisting of 80 randomly selected reviews. We note two caveats:

1. Since punctuation has been removed, the end of sentence i in a review will be indistinguishably connected to the beginning of sentence $i + 1$ in the review. As such, this entails nearness for two words that are in fact not near.
2. It is inefficient to generate all possible two word combinations for a given list of words. Ideally, we would classify w followed by t (or vice versa) as 1 and any other combination of t and w (where t and w are not next to each other) as 0. Instead, we loop through all reviews and take the i^{th} word and the $(i + 1)^{th}$ word to be a classification of 1 for all i. We also choose to generate classifications of 0 randomly by selecting two words from the embeddings. This will likely result in false classifications, but will be regarded as noise. We choose to have and equal number of classifications of 0 and 1.

Since we have used logistic regression, we will classify any output $y \geq 0.5$ as two words being near each other. An output $y < 0.5$ entails two words being not near each other. Our test and validation sets are generated in the same manor as our training set, but with 20 reviews each. We note that we have considered only unidirectional nearness i.e., w followed by t and not vice versa; this is okay since our test, validation, and training sets are consistent.

The performance of the test, validation and training sets reach 77.1%, 78.3% and 77.9%, respectively. The performance of the classifier is shown below in Figure 2.
From the performance of this classifier we see that we can predict whether a word is near another word with close to 80% accuracy. This is enough to show the effectiveness of *word2vec*. The performance of this classifier may increase if we choose to separate sentences in reviews as well as do a check when randomly generating pairs for the negative cases. That is, check if a randomly generated pair is a pair that is in the list of pairs that are near each other and discount it if so. Another way to improve performance would be to increase the size of the training, test and validation sets.
Please refer to Appendix D for this part's code.

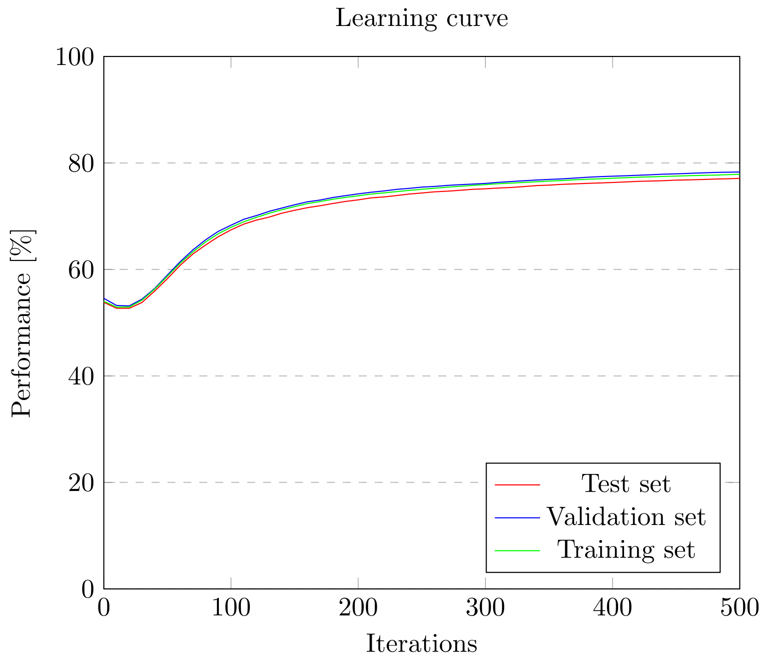
Learning curve
100
80
60
40
20
0
Performance [%]
0
100
200
300
400
500
Iterations
Test set
Validation set
Training set

Figure 2

Part 8

We show that *word2vec* works by computing the 10 nearest vectors to a given word. The distance between two vectors is computed using the negative cosine distance.

The words "story" and "good" yield the following closest vectors.

- The 10 closest words to "story" are: *plot, film, benito, simmer, sitter, lift, domineering, ricci, interviews, acclaim*
- The 10 closest words to "good" are: *bad, great, wonderful, reinforcing, decent, funny, manipulate, underused, admiral, perplexing*

We observe that words that would be used in a similar context as "story" and "good" are closest. For "story", "plot" and "film" are essentially synonyms and are expected to have similar context to that of "story" when used in a sentence. For "good", "bad" is an antonym and would also be used in similar contexts, e.g., "the good person" vs. "the bad person".

Two more interesting words are "man" and "he". We found them by thinking about what words would generally have similar context, i.e., could be replaced in a sentence with other words easily.

- The 10 closest words to "man" are: *woman, journal, guy, boy, showtime, men, cancer, arising, olympia, preacher*
- The 10 closest words to "he" are: *she, it, they, who, i, sieber, swordfish, everyone, we, joe*

Please refer to Appendix D for this part's code.

Appendix A

naivebayes.py (Python 2.7)

```python
#####################################################
# NAIVE BAYES
# Yoshiki Shoji              & Zi Mo Su
#####################################################

import os
import numpy as np
from numpy import *
import re
import time


#####################################################
# FUNCTIONS
#####################################################

def preprocess(line):
    """
    This function preprocesses a single line in a file by removing punctuation and extra spaces.
    :param line: (string) line in file
    :return: (string) processed line
    """
    # remove punctuation
    line = re.sub(r'(\W)', ' ', line)

    # remove extra spaces including at beginning and ending of sentence
    line = re.sub(r'\s+', ' ', line)
    line = re.sub(r'\s$', '', line)
    line = re.sub(r'^\s', '', line)

    # make all words lowercase
    line = line.lower()

    return line


def partition(dir):
    """
    This function partitions all reviews in a directory into a training set (size 800), test set (size 100) and
    validation set (size 100).
    :param dir: (string) name of directory
    :return: (lists) lists of strings containing reviews for each set
    """
    # random permutation
    rand = np.random.permutation(1000)
    data = []

    # loop through all files
```

```python
    for filename in os.listdir(dir):
        input_file = open(os.path.join(dir, filename))

        # process each line in current file and append as string
        processed = ''
        for line in input_file:
            processed += ' ' + preprocess(line)

        data += [processed]
        input_file.close()

    data = array(data)

    # partition into test, validation and training set using random permutation
    test = data[rand[:100]]
    val = data[rand[100:200]]
    train = data[rand[200:]]

    return test, val, train


def get_counts(train):
    """
    This function gets the number of times a word is present in a review for the class defined by the training
    set input.
    :param train: (list) training set
    :return: (dict) counts is a dictionary with keys that are strings (words) and values that are their floats (count)
             (float) total count
    """
    counts = {}
    total_count = 0.0

    for review in train:
        counted = []
        words = review.split()
        for word in words:
            if word in counted:
                continue

            if word not in counts:
                counts[word] = 0

            counts[word] += 1.0/len(words)
            total_count += 1.0/len(words)

            counted += [word]

    return counts, total_count


def classify(review, neg_probs, pos_probs, neg_total, pos_total, m, k):
```

```
    """
    This function classifies a review as either positive or negative.
    :param review: (string) a review
    :param neg_probs: (dict) probabilities generated from negative reviews
    :param pos_probs: (dict) probabilities generated from positive reviews
    :param neg_total: (float) total count for negative reviews
    :param pos_total: (float) total count for positive reviews
    :param m: (float) tunable parameter for smoothing
    :param k: (float) tunable parameter for smoothing
    :return: (string) 'neg' if classified as negative, 'pos' if classified as positive
    """
    neg_prob = 0.0
    pos_prob = 0.0

    for word in review.split():
        if word in neg_probs:
            neg_prob += log(neg_probs[word])
        else:
            neg_prob += log((m * k)/(neg_total + k))

        if word in pos_probs:
            pos_prob += log(pos_probs[word])
        else:
            pos_prob += log((m * k)/(pos_total + k))

    # print 'neg:', neg_prob
    # print 'pos:', pos_prob

    # final probability [product of conditionals P(word|class)] * [P(class)]
    neg_prob = neg_prob + log(1/2.0)
    pos_prob = pos_prob + log(1/2.0)

    if neg_prob > pos_prob:
        return 'neg'
    else:
        return 'pos'


def get_performance(neg_probs, pos_probs, neg_val, pos_val, neg_total, pos_total, m, k
    ):
    """
    This function gets the performance of the classifier on a set.
    :param neg_probs: (dict) probabilities generated from negative reviews
    :param pos_probs: (dict) probabilities generated from positive reviews
    :param neg_val: (list) list of negative reviews to be classified
    :param pos_val: (list) list of positive reviews to be classified
    :param neg_total: (float) total count for negative reviews
    :param pos_total: (float) total count for positive reviews
    :param m: (float) tunable parameter for smoothing
    :param k: (float) tunable parameter for smoothing
    :return: (floats) performance for negative, positive and total
    """
    neg_correct = 0.0
```

```
    neg_wrong = 0.0
    pos_correct = 0.0
    pos_wrong = 0.0

    for review in neg_val:
        c = classify(review, neg_probs, pos_probs, neg_total, pos_total, m, k)
        if c == 'neg':
            neg_correct += 1
        else:
            neg_wrong += 1

    for review in pos_val:
        c = classify(review, neg_probs, pos_probs, neg_total, pos_total, m, k)
        if c == 'pos':
            pos_correct += 1
        else:
            pos_wrong += 1

    neg_perf = neg_correct/(neg_correct + neg_wrong)
    pos_perf = pos_correct/(pos_correct + pos_wrong)

    perf = (neg_correct + pos_correct)/(neg_correct + neg_wrong + pos_correct +
        pos_wrong)

    return neg_perf, pos_perf, perf


def find_best_params(m_s, k_s, neg_train, pos_train, neg_val, pos_val):
    """
    This function finds the best m's and k's in terms of performance.
    :param m_s: (list) list of m values to test
    :param k_s: (list) list of k values to test
    :return: (floats) best m and k value
    """
    best_perf = 0

    neg_counts, neg_total = get_counts(neg_train)
    pos_counts, pos_total = get_counts(pos_train)

    for m in m_s:
        for k in k_s:
            neg_probs = {key: (val + m * k)/(neg_total + k) for key, val in neg_counts
                .items()}
            pos_probs = {key: (val + m * k)/(pos_total + k) for key, val in pos_counts
                .items()}

            neg_perf, pos_perf, perf = get_performance(neg_probs, pos_probs, neg_val,
                pos_val, neg_total, pos_total, m, k)

            print 'Test using m =', m, 'and k =', k
            print 'Classification performance:'
            print '\tNegative:', neg_perf
            print '\tPositive:', pos_perf
```

```
            print '\tTotal:', perf
            print

            if perf > best_perf:
                best_perf = perf
                best_m = m
                best_k = k

    print 'Best parameters found to be m =', best_m, 'and k =', best_k, 'with
        performance of', best_perf
    print

    return best_m, best_k


def get_top10(neg_counts, pos_counts):
    """
    This function gets the top 10 highest frequency words in the negative and positive
         sets that are exclusive to their
    own set.
    :param neg_counts: (dict) counts generated from negative reviews
    :param pos_counts: (dict) counts generated from positive reviews
    :return: (lists) top 10 highest frequency words for negative and positive reviews
    """
    neg_words = sorted(neg_counts, key=neg_counts.get, reverse=True)
    pos_words = sorted(pos_counts, key=pos_counts.get, reverse=True)

    neg_top10 = []
    count = 0
    for word in neg_words:
        if count > 10:
            break
        if word not in pos_words:
            neg_top10 += [word]
            count += 1

    pos_top10 = []
    count = 0
    for word in pos_words:
        if count > 10:
            break
        if word not in neg_words:
            pos_top10 += [word]
            count += 1

    return neg_top10, pos_top10


#######################################################
# MAIN CODE
#######################################################

if __name__ == '__main__':
```

```
    # generate random seed
    t = int(time.time())
    t = 1489990171
    print "t =", t
    random.seed(t)

    neg_dir = 'review_polarity/txt_sentoken/neg'
    pos_dir = 'review_polarity/txt_sentoken/pos'

    neg_test, neg_val, neg_train = partition(neg_dir)
    pos_test, pos_val, pos_train = partition(pos_dir)

    m_s = [0.000001, 0.00001, 0.0001, 0.0005, 0.001, 0.01, 0.05, 0.1]
    k_s = [0.01, 0.05, 0.1, 0.3, 0.5, 0.8, 1, 1.5, 2]

    best_m, best_k = find_best_params(m_s, k_s, neg_train, pos_train, neg_val, pos_val
        )

    neg_counts, neg_total = get_counts(neg_train)
    pos_counts, pos_total = get_counts(pos_train)

    neg_probs = {key: (val + best_m * best_k)/(neg_total + best_k) for key, val in
        neg_counts.items()}
    pos_probs = {key: (val + best_m * best_k)/(pos_total + best_k) for key, val in
        pos_counts.items()}

    neg_perf, pos_perf, perf = get_performance(neg_probs, pos_probs, neg_test,
        pos_test, neg_total, pos_total, best_m, best_k)
    print 'Using the best parameters, the performance on the test set is:'
    print '\tNegative:', neg_perf
    print '\tPositive:', pos_perf
    print '\tTotal:', perf
    print

    neg_perf, pos_perf, perf = get_performance(neg_probs, pos_probs, neg_train,
        pos_train, neg_total, pos_total, best_m, best_k)
    print 'Using the best parameters, the performance on the training set is:'
    print '\tNegative:', neg_perf
    print '\tPositive:', pos_perf
    print '\tTotal:', perf
    print

    neg_top10, pos_top10 = get_top10(neg_counts, pos_counts)

    print 'The top 10 words for determining a negative review are:', neg_top10
    print 'The top 10 words for determining a positive review are:', pos_top10
```

Appendix B

logistic.py (Python 2.7)

```
######################################################
# LOGISTIC
# Yoshiki Shoji              & Zi Mo Su
######################################################

import os
import numpy as np
from numpy import *
import re
import tensorflow as tf
import time


######################################################
# FUNCTIONS
######################################################

def preprocess(line):
    """
    This function preprocesses a single line in a file by removing punctuation and
        extra spaces.
    :param line: (string) line in file
    :return: (string) processed line
    """
    # remove punctuation
    line = re.sub(r'(\W)', ' ', line)

    # remove extra spaces including at beginning and end of sentence
    line = re.sub(r'\s+', ' ', line)
    line = re.sub(r'\s$', '', line)
    line = re.sub(r'^\s', '', line)

    # make all words lowercase
    line = line.lower()

    return line


def partition(dir):
    """
    This function partitions all reviews in a directory into a training set (size 800)
        , test set (size 100) and
    validation set (size 100).
    :param dir: (string) name of directory
    :return: (lists) lists of strings containing reviews for each set
    """
    # random permutation
    rand = np.random.permutation(1000)
    data = []
```

```
    # loop through all files
    for filename in os.listdir(dir):
        input_file = open(os.path.join(dir, filename))

        # process each line in current file and append as string
        processed = ''
        for line in input_file:
            processed += ' ' + preprocess(line)

        data += [processed]
        input_file.close()

    data = array(data)

    # partition into test, validation and training set using random permutation
    test = data[rand[:100]]
    val = data[rand[100:200]]
    train = data[rand[200:]]

    return test, val, train


def get_unique_words(train):
    """
    This function gets the list of unique words given a training set.
    :param train: (list) training set
    :return: (list) list of unique words
    """
    unique_words = []

    for review in train:
        words = review.split()

        for word in words:
            if word in unique_words:
                continue

            unique_words += [word]

    return unique_words


def get_input(set, unique_words):
    """
    This function gets the inputs to the neural network.
    :param set: (list) training, test or validation set
    :param unique_words: (list) list of unique words in the training set
    :return: (np arrays) x and y_ inputs generated from set
    """
    total_count = len(unique_words)
    set_x = zeros((0, total_count))
    set_length = len(set)
```

```
    for review in set:
        vector = zeros((1, total_count))
        words = review.split()

        for word in words:
            if word in unique_words:
                vector[:, unique_words.index(word)] = 1

        set_x = vstack((set_x, vector))

    set_y_ = vstack((ones((set_length/2, 1)), zeros((set_length/2, 1))))

    return set_x, set_y_


def create_nn(total_count, lam):
    """
    This function creates a neural network for a logistic regression model.
    :param total_count: (int) size of input layer
    :param lam: (float) regularization parameter
    :return: neural network
    """
    # create placeholder for input
    x = tf.placeholder(tf.float32, [None, total_count])

    # output
    W0 = tf.Variable(tf.random_normal([total_count, 1], stddev=0.01))
    b0 = tf.Variable(tf.random_normal([1], stddev=0.01))

    y = tf.nn.sigmoid(tf.matmul(x, W0)+b0)

    # create placeholder for classification input
    y_ = tf.placeholder(tf.float32, [None, 1])

    # define cost and training step
    decay_penalty = lam*tf.reduce_sum(tf.square(W0))
    reg_NLL = -tf.reduce_sum(y_*tf.log(y)+(1-y_)*tf.log(1-y))+decay_penalty

    train_step = tf.train.AdamOptimizer(0.0005).minimize(reg_NLL)

    # init = tf.initialize_all_variables()
    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)

    correct_prediction = tf.equal(tf.round(y), y_)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return sess, x, y_, train_step, decay_penalty, accuracy, W0, y


def get_best_nn(train_x, train_y_, val_x, val_y_, total_count, lams):
    """
```

```
    This function returns the best regularization parameter lambda.
    :param train_x: (np array) training set input x
    :param train_y_: (np array) training set input y_
    :param val_x: (np array) validation set input x
    :param val_y_: (np array) validation set input y_
    :param total_count: (int) size of input layer
    :param lams: (list) list of lambdas to test
    :return: (float) lambda value that yeilds best performance on the validation set
    """
    final_max = 0
    for lam in lams:
        max_val = 0
        sess, x, y_, train_step, decay_penalty, accuracy, W0, y = create_nn(
            total_count, lam)
        print 'Testing with:'
        print '\tLambda', lam

        for i in range(151):
            sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            if val_acc > max_val:
                max_val = val_acc

        if max_val > final_max:
            final_max = max_val
            best_lam = lam
        print 'Best validation performance was:', final_max
        print

    print 'The chosen lambda is:'
    print '\tLambda', best_lam
    print

    return best_lam


#####################################################
# MAIN CODE
#####################################################

if __name__ == '__main__':

    # generate random seed
    t = int(time.time())
    t = 1490118450
    print "t =", t
    random.seed(t)

    neg_dir = 'review_polarity/txt_sentoken/neg'
    pos_dir = 'review_polarity/txt_sentoken/pos'

    if not os.path.exists('logistic/test_x.npy'):
```

```
        neg_test, neg_val, neg_train = partition(neg_dir)
        pos_test, pos_val, pos_train = partition(pos_dir)

        test = hstack((neg_test, pos_test))
        val = hstack((neg_val, pos_val))
        train = hstack((neg_train, pos_train))

        unique_words = get_unique_words(train)
        total_count = len(unique_words)

        test_x, test_y_ = get_input(test, unique_words)
        val_x, val_y_ = get_input(val, unique_words)
        train_x, train_y_ = get_input(train, unique_words)

        np.save('logistic/test_x.npy', test_x)
        np.save('logistic/test_y_.npy', test_y_)
        np.save('logistic/val_x.npy', val_x)
        np.save('logistic/val_y_.npy', val_y_)
        np.save('logistic/train_x.npy', train_x)
        np.save('logistic/train_y_.npy', train_y_)
        np.save('logistic/unique_words.npy', unique_words)
        np.save('logistic/total_count.npy', array([total_count]))

    else:
        test_x = np.load('logistic/test_x.npy')
        test_y_ = np.load('logistic/test_y_.npy')
        val_x = np.load('logistic/val_x.npy')
        val_y_ = np.load('logistic/val_y_.npy')
        train_x = np.load('logistic/train_x.npy')
        train_y_ = np.load('logistic/train_y_.npy')
        unique_words = np.load('logistic/unique_words.npy')
        total_count = np.load('logistic/total_count.npy')[0]

    lams = [0, 0.1, 0.5, 1, 2, 5, 10, 100]

    lam = get_best_nn(train_x, train_y_, val_x, val_y_, total_count, lams)

    sess, x, y_, train_step, decay_penalty, accuracy, W0, y = create_nn(total_count,
        lam)

    test_plot = ''
    val_plot = ''
    train_plot = ''

    for i in range(151):
        sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

        if i % 10 == 0:
            print 'i=', i

            test_acc = sess.run(accuracy, feed_dict={x: test_x, y_: test_y_})
            print 'Test:', test_acc
```

```
            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            print 'Validation:', val_acc

            train_acc = sess.run(accuracy, feed_dict={x: train_x, y_: train_y_})
            print 'Train:', train_acc

            print 'Penalty:', sess.run(decay_penalty)

            test_plot += str((i, test_acc*100))
            val_plot += str((i, val_acc*100))
            train_plot += str((i, train_acc*100))

    print
    print 'Output for LaTeX plotting:'
    print 'Test', test_plot
    print 'Validation', val_plot
    print 'Train', train_plot
```

Appendix C

compare_nb_log.py (Python 2.7)

```
#####################################################
# COMPARE NAIVE BAYES TO LOGISTIC
# Yoshiki Shoji              & Zi Mo Su
#####################################################

import os
import numpy as np
from numpy import *
import re
import tensorflow as tf
import time
import naivebayes as nb
import logistic as lg


#####################################################
# FUNCTIONS
#####################################################

def get_largest_thetas(thetas, unique_words):
    """
    This function gets the 100 largest values in thetas.
    :param thetas: (iterable) list of theta values
    :param unique_words: (list) list of unique words in the training set
    :return: (list) 100 words corresponding to the largest 100 thetas
    """
    largest_thetas = []

    for _ in range(100):
        max_index = argmax(thetas)
        thetas[max_index] = -float('inf')
        largest_thetas += [unique_words[max_index]]

    return largest_thetas


#####################################################
# MAIN CODE
#####################################################

if __name__ == '__main__':

    # generate random seed
    t = int(time.time())
    t = 1490118450  # used to generate logistic data sets
    print "t =", t
    random.seed(t)

    neg_dir = 'review_polarity/txt_sentoken/neg'
    pos_dir = 'review_polarity/txt_sentoken/pos'
```

```
    neg_test, neg_val, neg_train = nb.partition(neg_dir)
    pos_test, pos_val, pos_train = nb.partition(pos_dir)

    if not os.path.exists('logistic/test_x.npy'):
        test = hstack((neg_test, pos_test))
        val = hstack((neg_val, pos_val))
        train = hstack((neg_train, pos_train))

        unique_words = lg.get_unique_words(train)
        total_count = len(unique_words)

        test_x, test_y_ = lg.get_input(test, unique_words)
        val_x, val_y_ = lg.get_input(val, unique_words)
        train_x, train_y_ = lg.get_input(train, unique_words)

        np.save('logistic/test_x.npy', test_x)
        np.save('logistic/test_y_.npy', test_y_)
        np.save('logistic/val_x.npy', val_x)
        np.save('logistic/val_y_.npy', val_y_)
        np.save('logistic/train_x.npy', train_x)
        np.save('logistic/train_y_.npy', train_y_)
        np.save('logistic/unique_words.npy', unique_words)
        np.save('logistic/total_count.npy', array([total_count]))

    else:
        test_x = np.load('logistic/test_x.npy')
        test_y_ = np.load('logistic/test_y_.npy')
        val_x = np.load('logistic/val_x.npy')
        val_y_ = np.load('logistic/val_y_.npy')
        train_x = np.load('logistic/train_x.npy')
        train_y_ = np.load('logistic/train_y_.npy')
        unique_words = np.load('logistic/unique_words.npy')
        total_count = np.load('logistic/total_count.npy')[0]


    # Naive Bayes

    m_s = [0.0005]
    k_s = [1.5]

    best_m, best_k = nb.find_best_params(m_s, k_s, neg_train, pos_train, neg_val,
        pos_val)

    neg_counts, neg_total = nb.get_counts(neg_train)
    pos_counts, pos_total = nb.get_counts(pos_train)

    neg_probs = {key: (val + best_m * best_k)/(neg_total + best_k) for key, val in
        neg_counts.items()}
    pos_probs = {key: (val + best_m * best_k)/(pos_total + best_k) for key, val in
        pos_counts.items()}

    thetas = []
```

```
    for word in unique_words:
        if word in neg_probs and word in pos_probs:
            thetas += [log(neg_probs[word]/pos_probs[word])]

        else:
            thetas += [-float('inf')]

    nb_largest_thetas = get_largest_thetas(thetas, unique_words)


    # Logistic Regression

    lams = [10]

    lam = lg.get_best_nn(train_x, train_y_, val_x, val_y_, total_count, lams)

    sess, x, y_, train_step, decay_penalty, accuracy, W0, y = lg.create_nn(total_count
        , lam)

    for i in range(151):
        sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

        if i % 10 == 0:
            print "i=", i

            test_acc = sess.run(accuracy, feed_dict={x: test_x, y_: test_y_})
            print "Test:", test_acc

            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            print "Validation:", val_acc

            train_acc = sess.run(accuracy, feed_dict={x: train_x, y_: train_y_})
            print "Train:", train_acc

            print "Penalty:", sess.run(decay_penalty)

    thetas = sess.run(W0)

    lg_largest_thetas = get_largest_thetas(thetas, unique_words)


    # Compare Naive Bayes to Logistic Regression

    print nb_largest_thetas
    print lg_largest_thetas
```

Appendix D

word2vec.py (Python 2.7)

```
#####################################################
# WORD2VEC
# Yoshiki Shoji & Zi Mo Su
#####################################################

import os
import numpy as np
from numpy import *
import re
import tensorflow as tf
import time


#####################################################
# FUNCTIONS
#####################################################

def preprocess(line):
    """
    This function preprocesses a single line in a file by removing punctuation and extra spaces.
    :param line: (string) line in file
    :return: (string) processed line
    """
    # remove punctuation
    line = re.sub(r'(\W)', ' ', line)

    # remove extra spaces including at beginning and end of sentence
    line = re.sub(r'\s+', ' ', line)
    line = re.sub(r'\s$', '', line)
    line = re.sub(r'^\s', '', line)

    # make all words lowercase
    line = line.lower()

    return line


def partition(dir):
    """
    This function partitions all reviews in a directory into a training set (size 800), test set (size 100) and
    validation set (size 100).
    :param dir: (string) name of directory
    :return: (lists) lists of strings containing reviews for each set
    """
    # random permutation
    rand = np.random.permutation(1000)
    data = []
```

```
    # loop through all files
    for filename in os.listdir(dir):
        input_file = open(os.path.join(dir, filename))

        # process each line in current file and append as string
        processed = ''
        for line in input_file:
            processed += ' ' + preprocess(line)

        data += [processed]
        input_file.close()

    data = array(data)

    # partition into test, validation and training set using random permutation
    test = data[rand[:100]]
    val = data[rand[100:200]]
    train = data[rand[200:]]

    return test, val, train


def get_vector(word0, word1):
    """
    This function gets the word2vec vectors from word0 and word1 and concatenates them
        into one 256 dimension vector.
    :param word0: (string) a word
    :param word1: (string) a word
    :return: (None or np array) None if one of the words is not in the embeddings,
        otherwise return the vector
    """
    global EMBEDDINGS, WORD2INDEX

    if word0 in WORD2INDEX and word1 in WORD2INDEX:
        vec0 = EMBEDDINGS[WORD2INDEX[word0], :]
        vec1 = EMBEDDINGS[WORD2INDEX[word1], :]
        vec = hstack((vec0, vec1))

        return vec

    else:
        return None


def get_input(set, set_name):
    """
    This function gets the inputs to the neural network.
    :param set: (list) training, test or validation set
    :param set_name: (string) name of the set
    :return: (np arrays) x and y_ inputs generated from set
    """
    global EMBEDDINGS, WORD2INDEX
```

```
    set_x = zeros((0, 256))

    for review in set:
        words = review.split()
        for i in range(len(words)-1):
            if words[i] not in WORD2INDEX or words[i+1] not in WORD2INDEX:
                continue

            x = get_vector(words[i], words[i+1])

            set_x = vstack((set_x, x))

    print 'Size of', set_name + ':', 2*set_x.shape[0]
    set_y_ = vstack((ones((set_x.shape[0], 1)), zeros((set_x.shape[0], 1))))

    rand = random.randint(0, 41524, size=(set_x.shape[0], 2))
    x = zeros(((set_x.shape[0]), 256))
    for i in range(rand.shape[0]):
        for j in range(2):
            x[i, j*128:(j+1)*128] = EMBEDDINGS[rand[i, j], :]

    set_x = vstack((set_x, x))

    return set_x, set_y_


def create_nn(lam):
    """
    This function creates a neural network for a logistic regression model.
    :param lam: (float) regularization parameter
    :return: neural network
    """
    # create placeholder for input
    x = tf.placeholder(tf.float32, [None, 256])

    # output
    W0 = tf.Variable(tf.random_normal([256, 1], stddev=0.01))
    b0 = tf.Variable(tf.random_normal([1], stddev=0.01))

    y = tf.nn.sigmoid(tf.matmul(x, W0)+b0)

    # create placeholder for classification input
    y_ = tf.placeholder(tf.float32, [None, 1])

    # define cost and training step
    decay_penalty = lam*tf.reduce_sum(tf.square(W0))
    reg_NLL = -tf.reduce_sum(y_*tf.log(y)+(1-y_)*tf.log(1-y))+decay_penalty

    train_step = tf.train.AdamOptimizer(0.0005).minimize(reg_NLL)

    # init = tf.initialize_all_variables()
    init = tf.global_variables_initializer()
    sess = tf.Session()
```

```
    sess.run(init)

    correct_prediction = tf.equal(tf.round(y), y_)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return sess, x, y_, train_step, decay_penalty, accuracy, W0, y


def get_best_nn(train_x, train_y_, val_x, val_y_, lams):
    """
    This function returns the best regularization parameter lambda.
    :param train_x: (np array) training set input x
    :param train_y_: (np array) training set input y_
    :param val_x: (np array) validation set input x
    :param val_y_: (np array) validation set input y_
    :param lams: (list) list of lambdas to test
    :return: (float) lambda value that yeilds best performance on the validation set
    """
    final_max = 0
    for lam in lams:
        max_val = 0
        sess, x, y_, train_step, decay_penalty, accuracy, W0, y = create_nn(lam)
        print 'Testing with:'
        print '\tLambda =', lam

        for i in range(301):
            sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

            # print sess.run(y, feed_dict={x: val_x, y_: train_y_})

            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            if val_acc > max_val:
                max_val = val_acc

            # if i % 10 == 0:
            #      print i
            #      print val_acc

        if max_val > final_max:
            final_max = max_val
            best_lam = lam
        print 'Best validation performance was:', final_max
        print

    print 'The chosen lambda is:'
    print '\tLambda =', best_lam
    print

    return best_lam


def get_closest_words(word):
    """
```

```
    This function gets the 10 closest words to the input word.
    :param word: (string) word
    :return: (list) list of 10 closest words to word
    """
    index = WORD2INDEX[word]
    vec = EMBEDDINGS[index, :]

    cos_distance = -dot(EMBEDDINGS, reshape(vec, (128, 1)))/reshape((linalg.norm(vec)*
        linalg.norm(EMBEDDINGS, axis=1)), (41524, 1))

    cos_distance[index] = float('inf')

    closest_words = []
    for _ in range(10):
        min_index = argmin(cos_distance)
        cos_distance[min_index, :] = float('inf')
        closest_words += [INDEX2WORD[min_index]]

    return closest_words


#######################################################
# MAIN CODE
#######################################################

if __name__ == '__main__':

    # generate random seed
    t = int(time.time())
    t = 1490071244  # for generating data
    print "t =", t
    random.seed(t)

    EMBEDDINGS = load('embeddings.npz')['emb']  # shape (41524, 128)
    INDEX2WORD = load('embeddings.npz')['word2ind'].flatten()[0]  # dictionary of
        length 41524
    WORD2INDEX = {v: k for k, v in INDEX2WORD.iteritems()}

    neg_dir = 'review_polarity/txt_sentoken/neg'
    pos_dir = 'review_polarity/txt_sentoken/pos'

    if not os.path.exists('word2vec/test_x.npy'):
        neg_test, neg_val, neg_train = partition(neg_dir)
        pos_test, pos_val, pos_train = partition(pos_dir)

        test = hstack((neg_test[:10], pos_test[:10]))
        val = hstack((neg_val[:10], pos_val[:10]))
        train = hstack((neg_train[:40], pos_train[:40]))

        test_x, test_y_ = get_input(test, 'test set')  # size: 22308
        val_x, val_y_ = get_input(val, 'validation set')  # size: 24468
        train_x, train_y_ = get_input(train, 'training set')  # size: 109144
```

```
        np.save('word2vec/test_x.npy', test_x)
        np.save('word2vec/test_y_.npy', test_y_)
        np.save('word2vec/val_x.npy', val_x)
        np.save('word2vec/val_y_.npy', val_y_)
        np.save('word2vec/train_x.npy', train_x)
        np.save('word2vec/train_y_.npy', train_y_)

    else:
        test_x = np.load('word2vec/test_x.npy')
        test_y_ = np.load('word2vec/test_y_.npy')
        val_x = np.load('word2vec/val_x.npy')
        val_y_ = np.load('word2vec/val_y_.npy')
        train_x = np.load('word2vec/train_x.npy')
        train_y_ = np.load('word2vec/train_y_.npy')

    lams = [0, 0.1, 0.5, 1, 2, 5, 10, 100]

    lam = get_best_nn(train_x, train_y_, val_x, val_y_, lams)

    sess, x, y_, train_step, decay_penalty, accuracy, W0, y = create_nn(lam)

    test_plot = ''
    val_plot = ''
    train_plot = ''

    for i in range(501):
        sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

        if i % 10 == 0:
            print 'i=', i

            test_acc = sess.run(accuracy, feed_dict={x: test_x, y_: test_y_})
            print 'Test:', test_acc

            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            print 'Validation:', val_acc

            train_acc = sess.run(accuracy, feed_dict={x: train_x, y_: train_y_})
            print 'Train:', train_acc

            print 'Penalty:', sess.run(decay_penalty)

            test_plot += str((i, test_acc*100))
            val_plot += str((i, val_acc*100))
            train_plot += str((i, train_acc*100))

    print
    print 'Output for LaTeX plotting:'
    print 'Test', test_plot
    print 'Validation', val_plot
    print 'Train', train_plot
    print
```

```
    # PART 8
    closest_words = get_closest_words('story')
    print 'Closest words to "story" are:', closest_words

    closest_words = get_closest_words('good')
    print 'Closest words to "good" are:', closest_words

    closest_words = get_closest_words('man')
    print 'Closest words to "man" are:', closest_words

    closest_words = get_closest_words('he')
    print 'Closest words to "he" are:', closest_words
```