CSC411: Project 3

Due on Tuesday, March 21, 2017

Zheng Yu Chen, Michael Pham-Hung

October 14, 2017

Part 1

Note: please extract the dataset into the working directory before running any code for this project.

The dataset consists of full-length reviews that are classified as either "positive" or "negative". The reviews are split on sentences (i.e. an end-of-sentence detecting algorithm has already been used on the reviews). To further process this, a couple of steps were taken:

- All punctuation and math symbols were replaced with a space
- All multiple spaces were reduced to one space
- All sentences in one review were condensed into one long string of words (separated by spaces)

The specific regular expression code that accomplishes the first two points is shown Listing 1 below.

Listing 1: Code for removing punctuation from a sentence

```
def preprocess(sent):
    '''
    Preprocesses the sentences
    '''
    sent = sent.lower()
    sent = re.sub('[-"(),.?!*/:;+=<>`\[\]\_]', ' ', sent)
    sent = re.sub("'", ' ', sent)
    sent = re.sub("\s{2,}", ' ', sent)
    sent = sent.strip()
    return sent
```

The reviews (now represented as one long string of words) were stored into a dictionary M that mimics the MNIST digits dataset. All positive reviews are stored under `M["train0"]` and `M["test0"]`, and all negative in `M["train1"]` and `M["test0"]`. This is so the code from the previous project can be used. This also means that one-hot encoding will be used even though there are only two classes.

It is easy to predict that certain adjectives will be indicative of whether a review is positive or negative. Ignoring sarcasm, the sentences "the movie is great" and "the plot was bad" are completely dependent on the adjectives "great" and "bad". To see how frequently these adjectives appear in the dataset, a quick script was created to count the frequency of all words in all reviews.

The word "bad" represents 0.164% of all words used in negative reviews, whereas it is only 0.0512% of all words in positive reviews. This means that it is much more frequent in negative reviews and it will be a useful word when predicting sentiment. The word "great" represents 0.107% of words in positive reviews and 0.063% in negative reviews, again confirming our hypothesis to some degree. A last example we will analyze is the word "funny", which one might predict is more frequent in positive reviews. However, the dataset shows the opposite, with it having 0.055% frequency in positive reviews and 0.071% in negative reviews. Of course, any of these three words can be used in any review, so this small difference is not unexpected. For example, "it's funny how horrendous the jokes are" could appear in a negative review.

The ten most frequent words for positive and negative reviews are shown below along with their frequency percentages. Immediately, they are not very useful, since they are simply common words that appear in the English language. Thus, it is still likely that certain adjectives will be the most telling features when it comes to sentiment. The code used to compute all values in Part 1 is included in the submitted `part1.py`.

```
Top 10 words that are most frequent in positive reviews, with percentages
                    (most frequent at bottom):
                       that:  1.15143823427
                        it:  1.1841946668
                        s:  1.36882183195
                         in:  1.662920711
                       is:  1.99388830198
                       to:  2.34229762977
                        of:  2.6426358293
                       and:  2.82144908218
                        a:  2.86342269269
                       the:  5.88453003027

Top 10 words that are most frequent in negative reviews, with percentages
                    (most frequent at bottom):
                       it:  1.22741819586
                       that:  1.23517364288
                        s:  1.39851285347
                       in:  1.59841345713
                       is:  1.76349368643
                       to:  2.44074996755
                       of:  2.45167090315
                       and:  2.48237614235
                        a:  2.83374537443
                       the:  5.55163386693
```

Part 2

We wish to implement the Naive Bayes Algorithm for classifying the reviews. We can define:

$$class = argmax_{class} P(class)\Pi_{i=0}^{i} P(w_i|class) \tag{1}$$

Where we want to learn $P(w_i = 1|class) \approx \frac{count(w_i|class)}{count(class)}$ given a data set. Note that we can also get $P(w_i = 0|class) = 1 - P(w_i = 1|class)$.

Training the Model

Since the data set is so large, the computation time to get each individual probability would take the model an hour to compute each time. To mitigate the amount of time for each run, the probabilities were stored in a pickle file. The probabilities themselves were stored in two different dictionaries. The keys of each dictionaries represented the **conditional probabilities** of the word describing the key, given the class, and were appropriately named *pos* and *neg*.

In the event the model sees a word not previously seen before, we want to avoid giving the word a probability of zero (Since it was defined by count). Therefore, we re-define the conditional probability as:

$$P(w_i = 1|class) \approx \frac{count(w_i|class) + mk}{count(class) + k}$$

where m and k are hyper parameters that can be tuned with the validation set. For simplicity in this experiment, the model used $k = 1$ and only m was tuned.

To train the model, only the value $\frac{count(w_i|class)}{count(class)+k}$ was stored in the dictionaries, in order for us to tune parameter m more quickly later on.

Evaluating and Tuning the Model

When actually evaluating Equation 1, the program would experience underflow since the probabilities are so small (most on a scale of 10^{-6}). However, we can use the fact that $a_1a_2a_3... = exp(loga_1 + loga_2 + loga_3 + ...)$ to avoid this problem. Note that if $a > b$ then $log(a) > log(b)$ and that if $a_1a_2 > b_1b_2$, then $log(a_1a_2) = log(a_1) + log(a_2) > log(b_1) + log(b_2)$. Then, for probability $p << 1$, we can use the log probabilities for each class and take the maximum as the output class.

Additionally, since the training set size for the positive and negative cases were exactly even, the term $P(class)$ was left out during evaluation.

Leaving the term $\frac{mk}{count(class)+k}$ from each probability allows us to tune m more quickly and easily. When evaluating the model, for each word that was seen in the training set, the conditional probabilities given positive and negative classes were computed using the look-up dictionaries. Each probability was then increased by $\frac{mk}{count(class)+k}$.

For each word that was not seen previously from the training set, a probability of $\frac{mk}{count(class)+k}$ was assigned. Then, summing the log probabilities, including $P(w_i = 0|class)$ and taking the argmax returns the predicted output.

The prediction function in Listing 2 runs relatively quickly and allows quick tuning with a for loop and different values for m.

Listing 2: Code for Validation

```python
def validation(tst, pos, neg, clas, m = 0.0004):
    #validation of reviews
    k = 1.
    count = 0
    tie_count = 0

    for review in tst:
        pos_words = set(review.split(' ')).intersection(set(pos.keys()))
        neg_words = set(review.split(' ')).intersection(set(neg.keys()))

        pos_prob = [pos[word] for word in pos_words] + [(1-pos[word]) for word
                    in pos.keys() if word not in pos_words]
        delp = set(review.split(' ')).difference(set(pos.keys()))

        neg_prob = [neg[word] for word in neg_words] + [(1-neg[word]) for word
                    in neg.keys() if word not in neg_words]
        deln = set(review.split(' ')).difference(set(neg.keys()))

        pos_prob = np.array(pos_prob) + m*k/(800.+k)
        neg_prob = np.array(neg_prob) + m*k/(800.+k)
        pos_prob = np.sum(np.log(pos_prob)) + len(delp)*np.log(m*k/(len(tst) + k))
        neg_prob = np.sum(np.log(neg_prob)) + len(deln)*np.log(m*k/(len(tst) + k))

        if pos_prob == neg_prob:
            tie_count += 1
        if clas:
            if pos_prob>neg_prob:
                count += 1

        else:
            if pos_prob<neg_prob:
                count += 1


    return (count)/float(len(tst))
```

Naive Bayes Results

Using a value $m = 0.0004$, the performance of the Naive Bayes Classifier was as follows:

```
Performance on positive training set 0.99625
Performance on negative training set 0.99875
 Performance on positive validation set 0.89
 Performance on negative validation set 0.78
    Performance on positive test set 0.76
    Performance on negative test set 0.81
```

Part 3

With the built dictionaries of conditional probabilities, we can sort by probabilities to obtain a list of words that contribute most to whether a review is positive or negative. However, we could not naive pick the word with the highest conditional probabilities given each class. Otherwise, we would result in getting the words outlined in Part 1.

To get a more accurate representation is to take the ratio of the conditional probabilities of each word given positive and negative classes (i.e. $\frac{P(w|+)}{P(w|-)}$). The following are the top ten words for each class.

```
List of words that indicate negative reviews:
                 jawbreaker
                   rabid
                 numbingly
                 zellweger
                  chester
                   sammy
                 eszterhas
                  darnell
                  headache
                 musketeers

List of words that indicate positive reviews:
                   mulan
                   flynt
                 groundhog
                   kermit
                   rudys
                  redford
                   cinque
                    taxi
                  benigni
                    coen
```

The results are very different from the adjectives anticipated in Part 1. A quick look at some of the words seems to indicate that the algorithm has associated certain movies with positive/negative reviews. Alternatively, certain movies may be mentioned a lot in positive reviews (e.g. “the director depicted a better version of Mulan”).

Part 4

Pre-processing

We wish to train a Logistic Regression model using the same data set using TensorFlow. Instead of inputting the review as a list of words, we first pre-process the data such that each review is represented as a k-dimensional vector v, where $v[k] = 1$ if the k-th keyword appears in the review.

L2 - Regularization

Regularization stops weights from growing too large however, given that the inputs range from 0 to 1, we expected little increase in performance when using regularization.

In order to experiment, the model was trained using different regularization parameters λ from 0 to 10. Every single model had performance similar to 1 which had $\lambda = 0$.

Results

Using Tensorflow, we found that the performance of the model peaked around 84%. Compared to the Naive Models trained in Part 2, both models performed quite similarly.

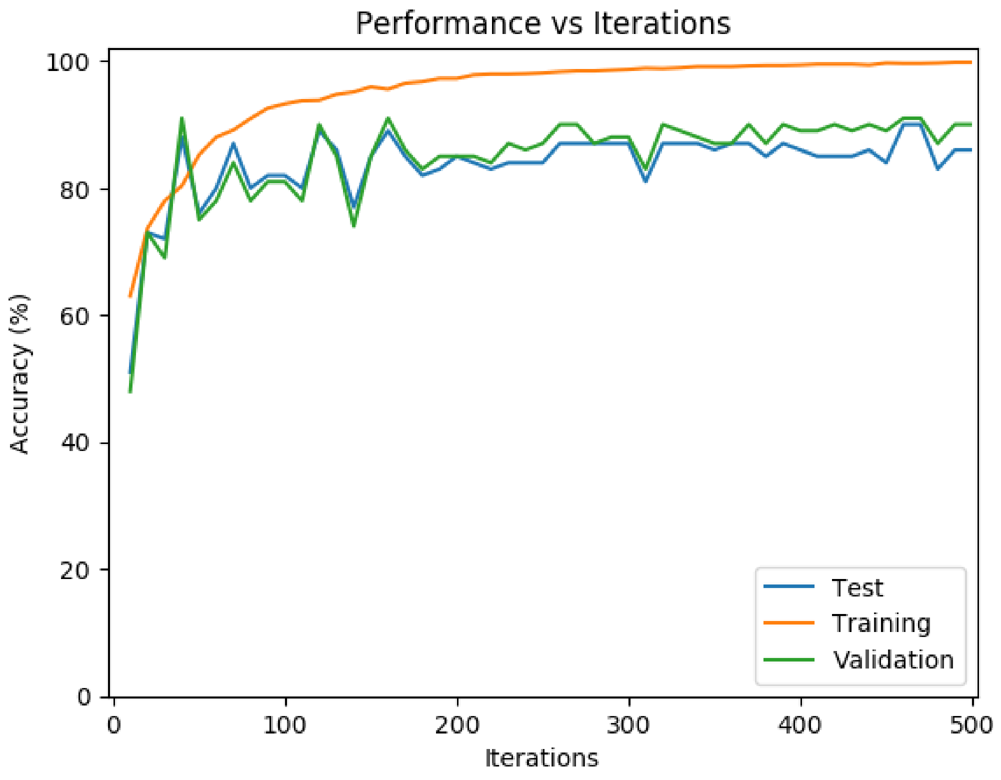

Figure 1: Learning Curve of the model up to 1000 iterations of gradient descent.

Part 5

At test time, we can compute the prediction for both the Naive Bayes and Logistic Regression Models with:

$$\theta_0 + \theta_1 I_1(x_1) + ... + \theta_k I_k(x_k) > thr \tag{2}$$

Logistic Regression

For a Logistic Regression Model, this is familiar as it can be represented as the dot products between the inputs and weights. However, using one-hot encoding, we have two sets of θ's corresponding to the likelihood of being positive or negative and we take the argmax of the two. Then we have for a positive prediction:

$$\theta_{pos0} + \theta_{pos1} x_1 + ... > \theta_{neg0} + \theta_{neg1} x_1 + ... \tag{3}$$

Rearranging,

$$\theta_{pos0} - \theta_{neg0} + (\theta_{pos1} - \theta_{neg1}) x_1 + ... > 0 \tag{4}$$

For the case of positive, the θ's represent the difference in weights for some key word and $I(x_n)$ is an indication variable of 1 or 0 indicating the presence of the keyword with a threshold of 0.

Naive Bayes

When computing the prediction using the Naive Bayes model, we compute the argmax between two classes. This can be redefined as:

$$P(\mathbf{w}|positive)P(positive) - P(\mathbf{w}|negative)P(negative) > 0 \tag{5}$$

where a positive value corresponds to a positive classification. Using the fact from Part 2, we actually use the log probabilities. So:

$$log(P(positive) - log(P(negative)) + log(P(w_1|positive)I_1) + ... - log(P(w_1|negative)I_1) - ... > 0 \tag{6}$$

$$log(\frac{P(positive)}{P(negative)}) + log(\frac{P(w_1|positive)}{P(w_1|negative)} \times I_1) + ... > 0 \tag{7}$$

Exponentiating both sides:

$$\frac{P(positive)}{P(negative)} + \frac{P(w_1|positive)}{P(w_1|negative)} \times I_1 + ... > 1 \tag{8}$$

From here we can see that θ_0 is the bias of the review. Since we used and equal amount of reviews per class, this is simply 1, **making the prediction unbiased**. θ_n is then the ratio of the conditional probabilities of the words. I_n is represented as the count of the nth key word which appears in the review. Additionally, note that the threshold is $e^0 = 1$.

Part 6

We wish to compare the top 100 θ's for each of the implemented models. Using the derivations from Part 5, we were able to achieve and sort the weights. Note the difference in magnitude of the weights. Since the logistic model takes in as in put a vector representing the review where each element in the vector is 0 or 1. This makes it intuitive that the weights won't be able to grow. For the Naive Bayesian model, the weights are trained by the frequency of the words in proportion to the number of reviews the model is trained on. The count can obviously be greater than one and therefore allows for the weights to grow.

Table 1: Top 100 words from the Bayesian and Logistic Models

Bayesian Weights	Value	Logistic Weights	Value
mulan	100.72346421	chasms	0.0736963
flynt	60.2437428084	opportunies	0.0738435
groundhog	30.9622069917	nightly	0.0738703
kermit	29.6641791045	reckless	0.0738735
rudys	28.4566498316	litany	0.0739404
redford	28.292151104	divorced	0.0740926
cinque	28.2884575672	remark	0.0741476
taxi	27.9504826394	pete	0.0741681
benigni	24.6317366616	imaginatively	0.0741909
coen	24.1860629243	defense	0.0742016
amidala	23.9104868451	burt	0.0742472
margaret	22.31048744	incidental	0.0743235
winslet	21.2590957053	walnuts	0.074371
behaviour	20.7655364233	cheque	0.0743958
library	20.3202831324	impacting	0.0744465
bergman	20.0156358528	reformed	0.0744535
exotica	19.6171981271	moralistically	0.0745498
nosferatu	19.4018210374	mazursky	0.074592
bandits	19.2914244159	rubell	0.074692
nello	18.9631688673	shot	0.0746945
jerome	18.7294469096	vannesa	0.0747323
astronomer	18.019024736	fulfillment	0.0748276
pleasantville	17.7818710659	verify	0.0749391
grodin	17.7186700767	unpretentious	0.0749955
hounsou	16.9278645005	painfully	0.0750849
spacey	16.5520537563	grainy	0.0751431
submarine	16.4779819771	deviousness	0.0752164
jordans	16.4750777146	shroud	0.0753491
jedi	16.3755102246	gears	0.0753711
homer	15.8242761791	macht	0.0753904
sandy	15.7581498503	annihilate	0.0756744
bishop	15.4930006962	escort	0.0757714
chess	15.3476757357	motivator	0.0759199
conveys	15.304084104	sullivan	0.0759871
journeys	15.2341051035	steinberg	0.0760358
symbols	15.2100161287	whimpers	0.0760686

hustler	15.1293883838	carriage	0.0762946
cerebral	15.1074855568	resists	0.0763488
beforehand	14.9156234562	dosage	0.0764432
obiwan	14.8679360924	meatiest	0.0764618
vader	14.5809282584	reneges	0.0765984
expertly	14.5184966263	entanglements	0.076602
secondly	14.2814017637	speechless	0.0766061
bateman	14.2223198594	sexually	0.0773126
michele	13.9481263535	protein	0.0778239
cousins	13.9455244494	knew	0.0780605
altman	13.7474566616	tripplehorn	0.0780792
egoyan	13.3597708776	overextended	0.0781987
carlitos	13.3308321737	carjacking	0.0784842
viewings	13.2546258785	consolidate	0.0787711
crowe	12.9489236275	heartbreak	0.0788413
addresses	12.8886910962	jeremiah	0.0788583
kenobi	12.591688518	keenan	0.078905
whisperer	12.5894987287	borrowing	0.0792814
modus	12.5425207789	encounters	0.0795515
swinton	12.2351181717	kozmo	0.0800802
versatile	12.2127632287	fabio	0.080308
osment	12.1243978832	travelling	0.0803272
hatred	12.113883023	fecal	0.0805055
accidents	12.106826532	rekindle	0.0806552
mathilda	12.0970737109	sloppily	0.0808078
bulworth	12.0868363313	orgasmic	0.0808321
rekindle	12.0305851049	tendancies	0.080901
passage	12.0181102654	put	0.0812131
naval	11.9537880691	fending	0.0813442
skis	11.9177674973	practical	0.0815711
daphne	11.9140122406	hermit	0.0821136
historian	11.9097568944	frontman	0.0821894
woodys	11.8795466163	nudie	0.0824448
rebellious	11.8461300214	patially	0.0830417
gunplay	11.8007406145	stapled	0.0838548
subtitled	11.7757428352	stupidity	0.0846706
denial	11.7432744461	hype	0.0852924
carmen	11.5720595441	atomic	0.0867402
fernando	11.5498708422	generously	0.0870617
unspoken	11.4054537082	numerous	0.0873174
finchers	11.2224333176	virtual	0.0875268
compilation	11.1752062986	active	0.0875584
amistad	11.1215703339	lawnmowers	0.0878295
gardener	11.1211499407	sakaguchi	0.0888877
observes	10.9559462549	delicacy	0.0892452
benign	10.9500553218	few	0.0892548
admits	10.9421952749	weighs	0.0900163

chad	10.8627912171	nosebleeds	0.090035
frequency	10.8168437466	engineering	0.0903419
skarsgard	10.8102643496	doos	0.0903867
lofty	10.7479971696	sliver	0.0906627
jules	10.7203320546	awhile	0.0907368
annual	10.6497731532	checks	0.0925657
bowden	10.6307104671	keital	0.0926671
symbolize	10.6116518733	sketchily	0.093953
pfeiffer	10.5628054066	argumentative	0.0939613
ringwald	10.551755608	wizard	0.0954431
homosexuality	10.5423028267	figurines	0.0954488
stylishly	10.5224429335	dawns	0.0973295
socks	10.4067281957	desired	0.0986627
weakness	10.3715673621	throughout	0.101213
weaknesses	10.3457000996	conceptualization	0.101616
bowfinger	10.3151009286	problem	0.102309
emperor	10.2842227512	coding	0.112358

Part 7

Parts 7 and 8 aim to investigate the effectiveness of `word2vec`, an unsupervised machine learning algorithm that turns strings into k dimensional vectors. As with earlier parts, the vocabulary of the movie dataset will be used. Note that we will not be training `word2vec` as it has already been done for us. Instead, given the embeddings (i.e. the k dimensional vector) corresponding with each word in the vocabulary, we wish to investigate different properties and the usefulness of `word2vec`.

In this part, we will conduct an experiment to see if `word2vec` is useful when we wish to find context within words. Specifically, given two words, can `word2vec` predict whether they would every appear adjacent together in a sentence? As an example, the words "good" and "professor" can certainly be used together, since one is an adjective and one is a noun. However, "surprising" and "of", one being an adjective and one a preposition, is much less likely to appear adjacently (a preposition cannot be the object of an adjective). Without classifying words with grammar tags or applying syntax rules, we will see if using `word2vec` embeddings as features is useful for a machine learning algorithm that wishes to classify the aforementioned property given two words.

Constructing the Dataset

Constructing the dataset will be a fairly important step in this experiment. **All** positive and negative reviews will be used to construct a dataset of possible word pairings. As an example, if we see the sentence `he's rather fast-for a horse that is.`, we will first preprocess it according to the same rules as the other parts of this project. However, we will not split a word on apostrophes for this part and just remove the apostrophes instead. The resulting sentence will thus be `hes rather fast for a horse that is`. This means the following word pairs will be constructed:

```
[(hes, rather), (rather, hes), (rather, fast), (fast, rather), (fast, for),
                  (for, fast), ..., (that, is), (is, that)]
```

This list of pairs will then be added to a set of all pairs found in all reviews. The set will be unique–each pair will only appear once. This process will be repeated for every sentence in every review. Then, after the set of all pairs has been built, a set of non-pairs will be built as well. This set will consist of words that never appear as pairs, *at least not in the review dataset.* To build this, two random words are selected from the vocabulary and put in a tuple. If this tuple does not appear in the set of all pairs, then we can append it to the set of non-pairs.

Finally, notice that this is slightly different from the instructions in the project. Two more pairings are constructed this way, namely `(rather, hes)` near the start and `(that, is)` near the end. We saw no reason to exclude these pairings, and it makes the logic in the code slightly simpler.

Issues with Dataset

There are certain problems that this dataset will inherently contain. Firstly, because we are only considering the pairs seen in the reviews, many generated non-pairs will be perfectly acceptable pairs. This particular method of generating the dataset improves as the dataset gets larger and larger. On another note, we conversely could see many pairs that do not actually work in English, since we are removing all punctuation. The sentence given above contains a perfect example. Because we remove dashes, `(fast, for)` will be marked as a pair. However, just earlier we made the argument that a preposition and an adjective will seldom be used together. In this case, removing punctuation means that they will be marked as a pair.

The Model

The process after constructing these two sets is simple. Given that we want a training set of size $2n$, we randomly select n pairs and n non-pairs. We take these pairs/non-pairs and turn them into embeddings and concatenate them together. Since $k = 128$ in this particular implementation of `word2vec`, each embedding is a (128,) vector. Concatenating them will create (256,) vectors, giving us 256 features per data point. Finally, the dataset is put into an M dictionary that mirrors the MNIST dataset, so that we can utilize the code from Project 2. The code that accomplishes everything discussed so far is shown after the results in Listing XXX.

TensorFlow with a logistic regression model will be used to train on the dataset. The initial weights and biases were generated with a normal distribution of standard deviation 0.01 and mean 0. The learning rate used was 0.0005. 10000 iterations will be run, since the performance seems to continue increasing slightly over time, without much overfitting. These parameters were determined through trial and error. In other words, the numbers were edited, and the performance on the validation set was tracked. The final values used optimized the performance on the validation set. However, since we are using mini batches for the training set, the performance will vary every time the training takes place. The code for TensorFlow can of course be found in the main file, `word2vec.py`.

Note that the project guidelines specified that the network should return 1 when the two words are a pair, and vice versa. One-hot encoding was used in this particular model. However, this is not an issue since you can just return the first or the second index of the one-hot encoding result if you wish to have a binary decision as an output.

Experiment Results

The final result on the test set, at the last iteration, was:

```
       Iter 10000
Valid:  82.4000000954
 Test:  82.9999983311
Train:  81.4400017262
```

The learning curve for the training, test, and validation sets are shown below in Figure 2. Notice how the performance jitters due to the use of mini batches to train. In conclusion, using the embeddings as features to classify context **was successful with a performance on the test set of 83%**. As we will further investigate in Part 8, this likely implies that `word2vec` manages to map words into vectors that retain certain syntactic and grammar rules. For example, parts of speech could be mapped similarly, so whenever the model sees a noun and an adjective together, it will mark it as a pair.

Finally, notice that the performance on the training set is not always the highest among the three sets as the number of iterations get higher and higher. This is due to the nature of the issues that were discussed earlier. Specifically, since the training set is so much larger, a lot of faulty pairs and non-pairs could have been selected. For example, the performance on the test set being the highest could simply mean that the validation set was generated with the least erroneous pairs/non-pairs.

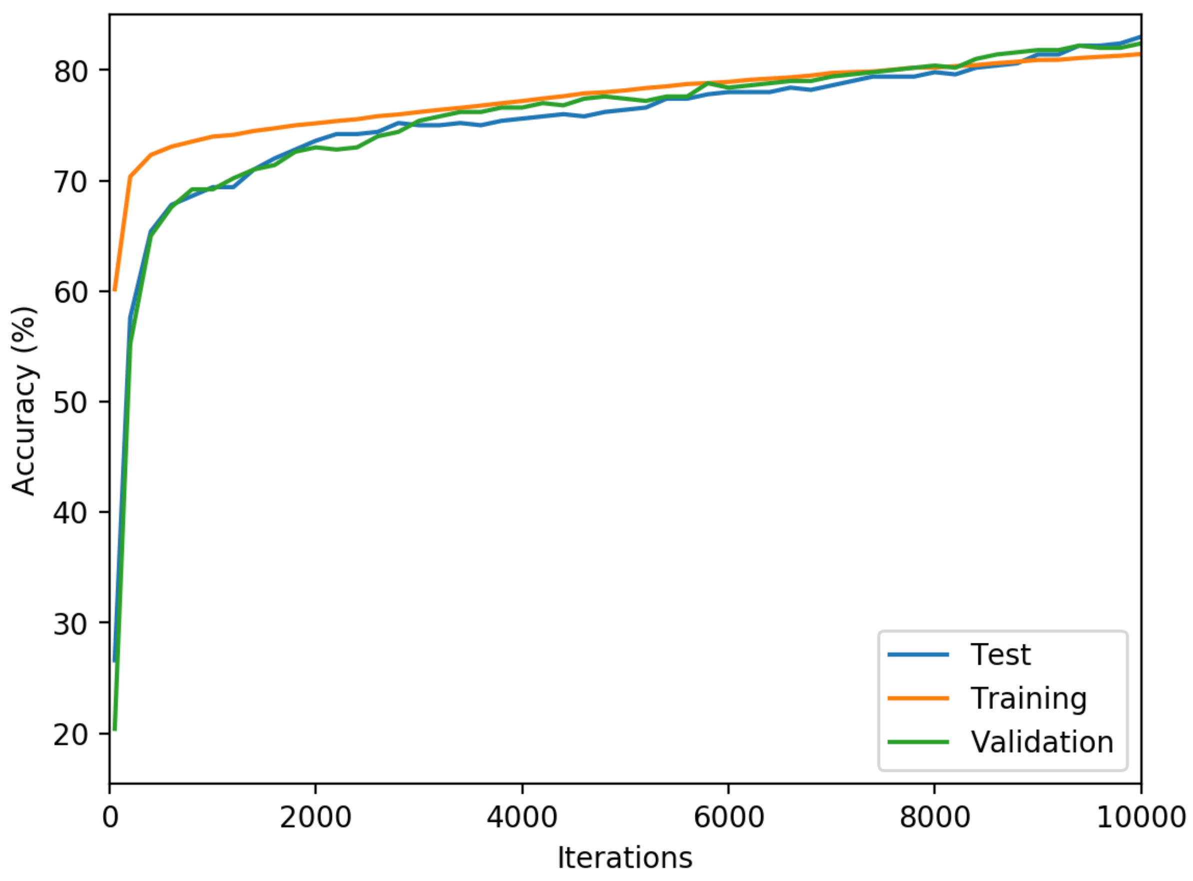

Figure 2: Performance of the model on the training, validation, and test sets vs the number of iterations

Listing 3: Code for constructing the dataset used in the Part 7 experiment

```
#---------------------------------EM FUNCTIONS-----------------------------------

def check_pair(pair):
    return (pair[0] in EM_I.values()) and (pair[1] in EM_I.values())

def get_i(word): #reverse lookup for words
    return EM_I.keys()[EM_I.values().index(word)]

def get_em(word): #returns the embedding for a word
    return EM[get_i(word)]

def pairs_to_em(array):
    result = []
    for pair in array:
        pair_em = list(get_em(pair[0]))
        pair_em.extend(list(get_em(pair[1])))
        result.append(pair_em)

    return result

#--------------------------------------------------------------------------------



#---------------------------DATASET BUILDING FUNCTIONS--------------------------

def preprocess(sent):
    '''
    Preprocesses the sentences
    '''
    sent = sent.lower()
    sent = re.sub('[-"(),.?!*/:;+=<>`\[\]\_]', ' ', sent)
    sent = re.sub("'", '', sent)
    sent = re.sub("\s{2,}", ' ', sent)
    sent = sent.strip()
    return sent

def get_pairs():
    '''
    Helper function that gets the images with file names from an input list
    'filenames'
    '''
    result = set()

    # Positive reviews first
    directory = reviews_dir + "pos/"

    for filename in os.listdir(directory):
        sentence = ''
        with open(directory + filename) as f:
            sentence = preprocess(f.read())
        f.close()
        splt = sentence.split()
        for i in range(len(splt) - 1):
            cur_pair = (splt[i], splt[i+1])
            if cur_pair not in result:
                result.add(cur_pair)

    # Negative reviews
    directory = reviews_dir + "pos/"

    for filename in os.listdir(directory):
        sentence = ''
        with open(directory + filename) as f:
            sentence = preprocess(f.read())
        f.close()

        splt = sentence.split()
        for i in range(len(splt) - 1):
            cur_pair = (splt[i], splt[i+1])
            cur_pair_flipped = (splt[i+1], splt[i])
            if cur_pair not in result:
                result.add(cur_pair)
            if cur_pair_flipped not in result:
                result.add(cur_pair_flipped)

    return result

def build_array(train_size, test_size):
    '''
    Builds a dictionary M that mimics the MNIST digits database. Assumes we want
    positive and negative reviews
    '''
    train_size, test_size = int(round(train_size/2)), int(round(test_size/2))
    print "Building M matrix with %d training pairs and %d testing pairs (per"
        + "class)" % (train_size, test_size)
    name = "cv" # all reviews should start with the characters 'cv'
    M = {}
    all_pairs = get_pairs()

    # randomly permutes the list of actor's faces
    pairs = np.random.permutation(list(all_pairs))

    # splits the permuted list into 2 to use for training and testing
    pair_i = 0
    training_set_pairs = []
    testing_set_pairs = []
    while len(training_set_pairs) < train_size:
        if check_pair(pairs[pair_i]):
            training_set_pairs.append(pairs[pair_i])
        pair_i += 1

    while len(testing_set_pairs) < test_size:
        if check_pair(pairs[pair_i]):
            testing_set_pairs.append(pairs[pair_i])
        pair_i += 1

    # chooses some words that dont appear as pairs in the reviews, at random
    not_pairs = set()

    while len(not_pairs) < train_size + test_size:
        #randomly gets two words from all reviews
        random_pairs = random.sample(EM_I.values(), 2)
        not_pair = (random_pairs[0], random_pairs[1])
        if not_pair not in all_pairs and check_pair(not_pair):
            if not_pair not in not_pairs:
                not_pairs.add(not_pair)

    not_pairs = list(not_pairs)
    training_set_npairs = not_pairs[:train_size]
    testing_set_npairs = not_pairs[train_size:]

    # mimic the MNIST dataset
    M["train0"] = np.array(pairs_to_em(training_set_pairs))
    M["test0"] = np.array(pairs_to_em(testing_set_pairs))
    M["train1"] = np.array(pairs_to_em(training_set_npairs))
    M["test1"] = np.array(pairs_to_em(testing_set_npairs))

    return M
```

Part 8

In this part, we wish to study whether a small cosine distance between two embeddings imply that the words have similar meanings and contexts. To do this, we will take the words "good" and "story" and retrieve their embeddings. Then, we will iterate through all other words in the dataset and calculate the distance between the two embeddings. The smallest 10 distances, along with their respective words, will be returned. The function used to do so is shown below in Listing XXX. Note that we developed our own implementations of vector 2-norm and inner product since the NumPy and SciPy implementations did not accurately compute these values.

Listing 4: Code for calculating the 10 closest words by cosine distance

```
'''
For some reason, the scipy and numpy distance calculating functions did not
work with the embeddings, so the cosine distance is quickly computed manually
'''
def inner(a, b):
    return sum(a[:]*b[:])

def norm(a):
    return np.sqrt(np.sum(a**2))

def cosine_dist(a, b):
    return 1 - inner(a,b)/(norm(a)*norm(b))

def find_closest(word, n):
    skip = get_i(word)
    word_em = np.array(get_em(word))

    closest = {}
    cur_max = inf

    for i in EM_I.keys():
        if i != skip:
            if len(closest.keys()) < 10:
                cmp_em = np.array(EM[i])
                result = cosine_dist(word_em, cmp_em)
                closest[result] = EM_I[i]
                cur_max = max(closest.keys())
            else:
                cmp_em = np.array(EM[i])
                result = cosine_dist(word_em, cmp_em)

                if result < cur_max:
                    del closest[cur_max]
                    closest[result] = EM_I[i]
                    cur_max = max(closest.keys())


    return closest
```

Analysis

The `find_closest()` function was then called for the words "story" and "good", which returns the list of words shown below. The words are sorted by closest to farthest, with their cosine distances shown beside them.

```
Words that are similar to story:
plot:  0.382376194
film:  0.480496644974
benito:  0.529078215361
simmer:  0.545108318329
sitter:  0.546389520168
lift:  0.561821967363
domineering:  0.56201082468
ricci:  0.56806397438
interviews:  0.569240659475
acclaim:  0.569423019886

Words that are similar to good:
bad:  0.477731823921
great:  0.505374789238
wonderful:  0.522868603468
reinforcing:  0.532412469387
decent:  0.534330278635
funny:  0.539387345314
manipulate:  0.54369816184
underused:  0.547805517912
admiral:  0.560093551874
perplexing:  0.570656359196
```

The closest words to "story" and "good" are typically used in the same contexts and are of the same types of speech. They do not necessarily have the same meaning: "bad" is closest to "good". The words "film" and "plot", however, are appropriate nouns that approximate "story". Both lists contain some words that are quite peculiar, but it is acceptable since there are not that many synonyms that are frequent in the reviews.

We can further analyze some words and see if the embeddings are determined through words of speech and context. Only a few of the closest words will be displayed for the test words below to keep things concise. Verbs seems to work well to display this result. The results for the verbs below show that the closest words are similar verbs, or sometimes just different conjugations.

```
Words that are similar to have:
has:  0.368212878704
had:  0.38553494215
be:  0.479516267776
are:  0.487275779247

Words that are similar to tell:
helps:  0.521380513906
babysitter:  0.525037676096
get:  0.535829842091
```

Even parts of speech like adverbs or conjunctions demonstrate that `word2vec` successfully determines the parts of speech:

```
Words that are similar to certainly:
     actually:  0.485852062702

Words that are similar to creatively:
        eat:  0.553073465824
   stylistically:  0.559642702341

   Words that are similar to and:
        but:  0.392793238163
         or:  0.458242416382

   Words that are similar to but:
        and:  0.392793238163
     however:  0.444784164429
   unfortunately:  0.466971814632
      although:  0.498668670654
```

However, the performance on weirder, harder-to-classify words like the prepositions "for" and "of" is not as good. This is likely because these words are very common and used in many different situations. Thus, it is hard for `word2vec` to determine their exact usage patterns and context.

```
Words that are similar to for:
  attacking:  0.555248111486
   preaching:  0.56739410758
     sixth:  0.568130284548
      ogre:  0.570289075375
  undeterred:  0.572483181953

Words that are similar to of:
 humiliating:  0.551697909832
  whistlers:  0.553060442209
     rizzo:  0.557013511658
     youths:  0.560865551233
     cubas:  0.565582275391
```