CMPS242 HW5 Report

Team Member: Yunzhe Li(#1571061) & Yong Deng(#1571065)

Section I: word2vec

- In this section, we import the dataset 'train.csv' 'test.csv' and convert them into numerical vectors using library Spacy's wording embedding.
- According to the official documentation of Spacy, its pre-trained build-in dictionary is actually from the GloVe with 300 dimensions per word vector, which I believe will greatly improve our accuracy.

In [1]:
```
## Import all the libraries I'll use in this section.
## Here the 'en_vectors_web_lg' is the dictionary we will use.

import pandas as pd
import en_vectors_web_lg
import numpy as np
```

Read the dataset

In [2]:
```
## Define a funtion to read the .csv file and split the labels and tweets into tw
o list.
## Note that the labels of test.csv is None, thus we discard them directly.

def csv_reading(f_name):

    with open(f_name, "r", encoding='utf8') as train:
        csvfile = pd.read_csv(train)

    csvfile = csvfile.values.tolist()

    labels = [row[0] for row in csvfile]
    twitters = [row[1] for row in csvfile]

    return labels, twitters
```

In [3]:
```
## Run the .csv file reading function

train_labels, train_twitters = csv_reading('train.csv')
_, test_twitters = csv_reading('test.csv')
```

Remove the urls

In [4]:
```
## Define a function to remove all the urls at the end of each twitter.

def remove_url(twitters):

    twitters_iter = twitters.__iter__()

    for i in range(len(twitters)):
        twitters[i] = twitters_iter.__next__().split('http')[0]

    return twitters
```

In [5]:
```
train_twitters = remove_url(train_twitters)
test_twitters = remove_url(test_twitters)
```

Apply the Spacy dictionary to implement words embedding

In [6]:
```
## load the dictionary

nlp = en_vectors_web_lg.load()
```

In [7]:
```
## Define the word2vec function.
## It will return a list of numpy ndarrays which has different length.
## Each tweet will convert to a numpy array with shape [length, 300].

def word2vec(twitters):

    twitters_vectors = [None]*len(twitters)

    for i in range(len(twitters)):

        twitter_doc = nlp(twitters[i])
        twitter_vector = [None]*len(twitter_doc)

        for j in range(len(twitter_doc)):
            twitter_vector[j] = twitter_doc[j].vector

        twitters_vectors[i] = twitter_vector

    return twitters_vectors
```

In [8]:
```
## Run the word2vec function

train_twitters = word2vec(train_twitters)
test_twitters = word2vec(test_twitters)
```

Convert the binary cases labels into 1 and 0

In [9]:
```
## Convert the label 'HillaryClinton' to 0 and 'realDonaldTrump' to 1

def numeric_label(labels):
    for i in range(len(labels)):
        if labels[i] == 'HillaryClinton':
            labels[i] = 0
        elif labels[i] == 'realDonaldTrump':
            labels[i] = 1

    return labels
```

In [10]:
```
train_labels = numeric_label(train_labels)
```

Save my embedding results into a .npz file for Section II use

In [11]:
```
np.savez('embedding_matrix.npz', train_matrix=train_twitters, test_matrix=test_tw
itters, train_labels=train_labels)
```

Section II: LSTM

- In this section, we build a training model comprising a deep LSTM network concatenated with a multiple hidden layers fully-connected network.
- The output of our training model is a logit. Then we compute its corresponding probability using sigmoid function. After that we can calculate the cross entropy loss by comparing with the true labels.
- The optimizer we are using is AdamOptimizer, which is much faster than the SGD or batch GD. Each time we input a batch of size 256, do the optimization then check its loss. Keep running it untill we hit the loss we want.
- Because the computation of cross validation is too expensive for my laptop, we didn't do it. To prevent overfitting, what we do is:
  1. run several optimization steps
  2. check the loss of current batch
  3. use the model to process the test set
  4. upload the result to Kaggle to see how does it perform
- Note that to run this LSTM statically, first we find the maximal sentence length between the train set and test set, mark it as our timestep in one layer LSTM cell. Then we pad all the sentence with zero vectors to ensure they all have the same length.

In [1]:
```
import numpy as np
import tensorflow as tf
```

```
/Users/hiroyukiqaq/.virtualenvs/cv/lib/python3.6/site-packages/h5py/__i
nit__.py:36: FutureWarning: Conversion of the second argument of issubd
type from `float` to `np.floating` is deprecated. In future, it will be
treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

Read the vectorized twitter dataset we got from Section I

In [2]:
```
## Define the file reading function.
## Return the train data, test data and the labels data respectively.

def import_matrix(f_name):
    with np.load(f_name) as data:
        train_matrix = data['train_matrix']
        test_matrix = data['test_matrix']
        train_labels = data['train_labels']
    return train_matrix, test_matrix, train_labels
```

In [3]:
```
train_matrix, test_matrix, train_labels = import_matrix('embedding_matri
x.npz')
```

Find the maximal sentence length

In [4]:
```
## Define a function to find the maximal length.
## The input is the train set or test set, return their maximal length r
espectively.

def max_length(matrix):
    matrix_iter = matrix.__iter__()
    max_length = 0
    for i in range(len(matrix)):
        temp = len(matrix_iter.__next__())
        if temp > max_length:
            max_length = temp
    return max_length
```

In [5]:
```
max_size = max(max_length(train_matrix), max_length(test_matrix))
print('max sentence length in both train.csv and test.csv is ', max_size
)
```

```
max sentence length in both train.csv and test.csv is  41
```

Pad all the twitters with zero vectors to make them have same length

In [6]:
```
## Define the zero_padding function.
## Input the maximal length we just find and the dataset, it will do the
 job.

def zero_padding(matrix, max_size):
    pad = np.zeros(300)
    for i in range(len(matrix)):
        if len(matrix[i]) < max_size:
            temp = len(matrix[i])
            diff = max_size - temp
            for j in range(diff):
                matrix[i].append(pad)
        matrix[i] = np.asarray(matrix[i])
    return matrix
```

In [7]:
```
train_matrix = zero_padding(train_matrix, max_size)
test_matrix = zero_padding(test_matrix, max_size)
```

Convert out data to the standard numpy ndarray

In [8]:
```
## For the convenience of manipulation, we convert our data set to numpy
 ndarray.

def ndarray_convert(matrix, max_size):
    temp = np.zeros((len(matrix), max_size, 300))
    for i in range(len(temp)):
        temp[i] = matrix[i]
    return temp
```

In [9]:
```
## The shape of our dataset is just what we expected.

train_matrix = ndarray_convert(train_matrix, max_size)
print('the shape of train set is ',train_matrix.shape)
test_matrix = ndarray_convert(test_matrix, max_size)
print('the shape of test set is ',test_matrix.shape)
```

```
the shape of train set is  (5000, 41, 300)
the shape of test set is  (1444, 41, 300)
```

Data preprocessing

In [ ]:
```
## To use the TensorFlow build-in batching interface,
## we convert our training data along with its labels to Tensors.
## The Tensors are not iterable, so we need to slice them into
## sliced Dataset, then batching them.

## Convert to Tensors.
train_tensor = tf.convert_to_tensor(train_matrix, dtype=tf.float32)
labels_tensor = tf.convert_to_tensor(train_labels, dtype=tf.float32)


## Convert to sliced dataset.
train_slices = tf.data.Dataset.from_tensor_slices(train_tensor)
labels_slices = tf.data.Dataset.from_tensor_slices(labels_tensor)
```

In [ ]:
```
## Batching the dataset with batch size 256.
## Note that the size of the dataset may not be perfectly divided by the
 batch size.
## Thus the last batch of the dataset have the size of remainder.
## When hitting the bottom, the iterator will start from beginning agai
n.
batch_size = 256
train_batch = train_slices.batch(batch_size).repeat()
labels_batch = labels_slices.batch(batch_size).repeat()


## Define the iterator of our batched dataset.
train_iter = train_batch.make_initializable_iterator()
labels_iter = labels_batch.make_initializable_iterator()

next_train = train_iter.get_next()
next_label = labels_iter.get_next()
```

Model building

- This is where we build our model. The model has two parts:
  - LSTM: it's a three layers LSTM network which the number of cell units is the same of the maximal length of the sentence, with initial state zeros and tanh activation function. The input of next layer is the output of last layer. The input of the first layer is a sentence vector with shape (41, 300), which means it have 41 words and each word is 300 dimension vector. The output of it is the last time step output of the third layer.
  - Fully-connected NN: it also has 3 hidden layers, with number of neurons 20, 30 and 10 respectively. The input of first layer is the output of our LSTM module. The activation function of the 3 hidden layers are leaky RELU, and the output layer has no activation function. It's just the simple scalar of the dot product result.

In [11]:
```
## Construct two placeholders for the input data and labels.

inputs = tf.placeholder(tf.float32, shape=(None, 41, 300))
labels = tf.placeholder(tf.float32, shape=(None, 1))
```

In [12]:
```
## Build the 3 LSTM cells.

with tf.variable_scope("lstm1"):
    lstm_cell1 = tf.contrib.rnn.BasicLSTMCell(41, forget_bias=1.0, activ
ation=tf.tanh)
with tf.variable_scope("lstm2"):
    lstm_cell2 = tf.contrib.rnn.BasicLSTMCell(41, forget_bias=1.0, activ
ation=tf.tanh)
with tf.variable_scope("lstm3"):
    lstm_cell3 = tf.contrib.rnn.BasicLSTMCell(41, forget_bias=1.0, activ
ation=tf.tanh)
```

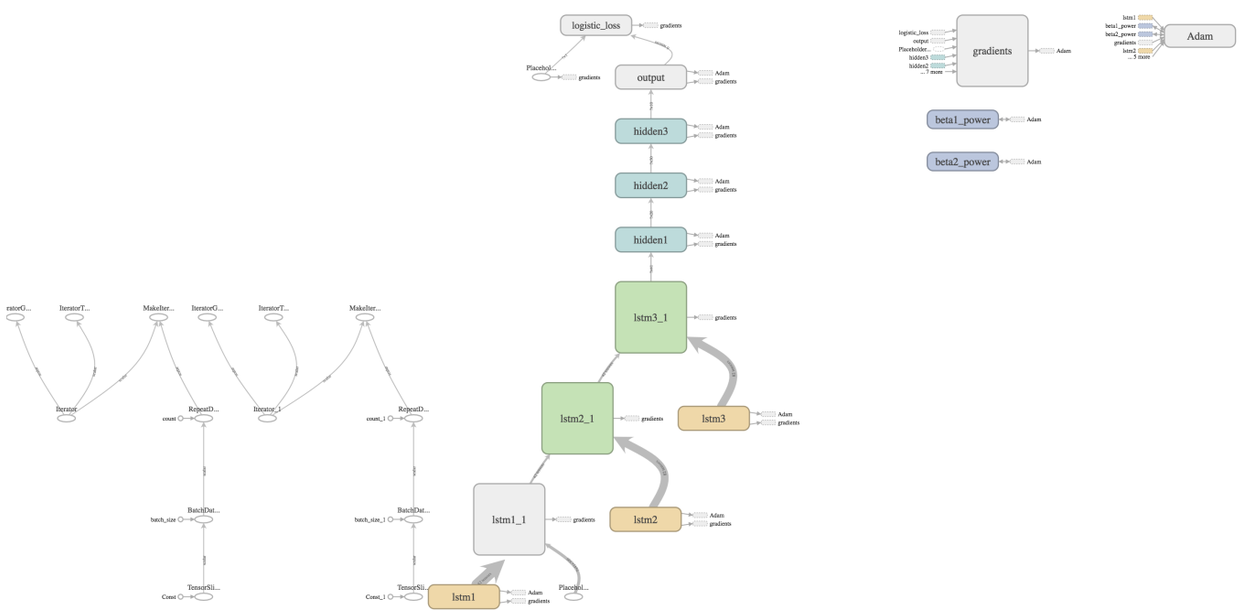
In [13]:
```
## Bind the three LSTM cell together to become a 3 layers deep RNN mode
l.

with tf.variable_scope("lstm1"):
    outputs1, state1 = tf.contrib.rnn.static_rnn(lstm_cell1, tf.unstack(
inputs, axis=1), dtype=tf.float32)
with tf.variable_scope("lstm2"):
    outputs2, state2 = tf.contrib.rnn.static_rnn(lstm_cell2, outputs1, d
type=tf.float32)
with tf.variable_scope("lstm3"):
    outputs3, state3 = tf.contrib.rnn.static_rnn(lstm_cell3, outputs2, d
type=tf.float32)
```

In [14]:
```
## Build the 3 hidden layers fully-connected neuron network.

with tf.variable_scope("hidden1"):
    hidden_output1 = tf.contrib.layers.fully_connected(outputs3[-1], 20,
 activation_fn=tf.nn.leaky_relu)
with tf.variable_scope("hidden2"):
    hidden_output2 = tf.contrib.layers.fully_connected(hidden_output1, 3
0, activation_fn=tf.nn.leaky_relu)
with tf.variable_scope("hidden3"):
    hidden_output3 = tf.contrib.layers.fully_connected(hidden_output2, 1
0, activation_fn=tf.nn.leaky_relu)
with tf.variable_scope("output"):
    logits = tf.contrib.layers.fully_connected(hidden_output3, 1, activa
tion_fn=None)
```

Optimization

In [15]:
```
# Define the cross entropy loss function.
loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=labels, logits=log
its)

# Initialize the AdamOptimizer
adamoptimizer = tf.train.AdamOptimizer()
train_op = adamoptimizer.minimize(loss)
```

The graph below is visualization of my whole workflow.



Train the model

In [16]:
```
## Define the session and initialize all the variables and iterators.

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
sess.run([train_iter.initializer, labels_iter.initializer])
```

Out[16]:
```
[None, None]
```

In [26]:
```
## Here is where we do the 'step by step' run and check trick to prevent
 overfitting.
## After each loop we will use the current model to predict the test set
 then upload
## to Kaggle to check the result.

for step in range(10):
    temp_input = sess.run(next_train)
    temp_label = sess.run(next_label)
    _, temp_loss = sess.run([train_op, loss], feed_dict={inputs: temp_in
put, labels: temp_label.reshape([-1, 1])})
    current_loss = temp_loss
    print('the log loss of current batch is ', np.mean(current_loss))
```

```
the log loss of current batch is  0.11498016
the log loss of current batch is  0.19568348
the log loss of current batch is  0.1094781
the log loss of current batch is  0.14224565
the log loss of current batch is  0.15021595
the log loss of current batch is  0.11847182
the log loss of current batch is  0.087889284
the log loss of current batch is  0.15502524
the log loss of current batch is  0.12887405
the log loss of current batch is  0.071093015
```

Predict the test set

In [28]:
```
## Run the model using the test dataset as input to get the correspondin
g logits.

result = sess.run(logits, feed_dict={inputs: test_matrix})
result = result.reshape(1444)
```

In [29]:
```
## Define a sigmoid function to convert the logits to probabilities.
def sigmoid(x):
    return 1/(1+np.exp(-x, dtype=np.float64))


## Output the result to a csv file according to the required format.
## Note that the result is probabilities of being '1', or 'realDonaldTru
mp'.
## To get the probabilities of being '0', or 'HillaryClinton', we only n
eed to
## compute 1-result.
def out_csv(test_result, f_name):

    temp = np.zeros([3, 1444])
    tweet_id = np.arange(1444, dtype=np.int)
    prob_trump = test_result
    prob_hillary = 1 - test_result
    temp[0] = tweet_id
    # temp[0] = temp[0].astype(int)
    temp[1] = prob_trump
    temp[2] = prob_hillary
    np.savetxt(f_name, temp.T, header="id,realDonaldTrump,HillaryClinto
n", comments="", fmt="%i,%.18e,%.18e")
    return 0
```

In [30]:
```
out_csv(sigmoid(result), 'result.csv')
```