Assignment_3

May 28, 2019

This particular assignment focuses on text classification using CNN. It has been picking up pace over the past few years. So, I thought this would be a good exercise to try out. The dataset is provided to you and there will be specific instrucions on how to curate the data, split into train and validation and the like. You will be using MXnet for this task. The data comprises tweets pertaining to common causes of cancer. The objective is to classify the tweets as medically relevant or not. The dataset is skewed with positive class or 'yes' being 6 times less frequent than the negative class or 'no'. (Total marks = 50). Individual marks to the sub-problems are given in bracket.

In [63]:
```
import logging
import sys
root_logger = logging.getLogger()
# stdout_handler = logging.StreamHandler(sys.stdout)
# root_logger.addHandler(stdout_handler)
root_logger.setLevel(logging.DEBUG)
```

In [27]:
```
# these are the modules you are allowed to work with.

import nltk
import re
import numpy as np
import mxnet as mx
import sys, os
import random

'''
First job is to clean and preprocess the social media text. (5)

1) Replace URLs and mentions (i.e strings which are preceeded with @)
2) Segment #hastags
3) Remove emoticons and other unicode characters
'''

def preprocess_tweet(input_text):
    '''
    Input: The input string read directly from the file

    Output: Pre-processed tweet text
```

```
    '''
    cleaned_text = re.sub(r"http\S+", "", input_text)
    cleaned_text = re.sub(r"@\S+", "", cleaned_text)
    cleaned_text = re.sub(r'[\U00010000-\U0010ffff]', "", cleaned_text)
    cleaned_text = re.sub(r'[()]', "", cleaned_text)
    cleaned_text = re.sub(r'[^\x00-\x7F]+', "", cleaned_text).strip().split()

    temp = cleaned_text
    cleaned_text = []

    for word in temp:
        if len(word) == 0:
            continue
        if word[0] == '#':
            if len(word) == 1:
                continue
            word = word[1 : ]
            sp = re.findall(r'[0-9A-Z]?[0-9a-z]+|[A-Z]+(?=[0-9A-Z]|$)', word)
            sp = [w.lower() for w in sp]
            cleaned_text += sp
        if not str.isalnum(word) and not str.isalpha(word):
            word = re.sub('[^A-Za-z0-9]+', '', word)
        else:
            cleaned_text.append(word.lower())

    #print(cleaned_text)
    return cleaned_text


# read the input file and create the set of positive examples and negative examples.

file=open('cancer_data.tsv')
pos_data=[]
neg_data=[]

for line in file:
    line=line.strip().split('\t')
    text2 = preprocess_tweet(line[0])
    if line[1]=='yes':
        pos_data.append(text2)
    if line[1]=='no':
        neg_data.append(text2)

print(len(pos_data), len(neg_data))

sentences= list(pos_data)
sentences.extend(neg_data)
pos_labels= [1 for _ in pos_data]
```

```python
neg_labels= [0 for _ in neg_data]
y=list(pos_labels)
y.extend(neg_labels)
y=np.array(y)

'''
After this you will obtain the following :

1) sentences =  List of sentences having the positive and negative examples with all th
2) y = List of labels with the positive labels first.
'''

'''
Before running the CNN there are a few things one needs to take care of: (5)

1) Pad the sentences so that all of them are of the same length
2) Build a vocabulary comprising all unique words that occur in the corpus
3) Convert each sentence into a corresponding vector by replacing each word in the sent

Example :
S1 = a b a c
S2 = d c a

Step 1:  S1= a b a c,
         S2 =d c a </s>
         (Both sentences are of equal length).

Step 2:  voc={a:1, b:2, c:3, d:4, </s>: 5}

Step 3:  S1= [1,2,1,3]
         S2= [4,3,1,5]

'''
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
porter = PorterStemmer()
stop_words = set(stopwords.words('english'))

def create_word_vectors(sentences):
    '''
    Input: List of sentences
    Output: List of word vectors corresponding to each sentence, vocabulary
    '''
    new_sentence = []
    for words in sentences:
        words = [word for word in words if word.isalpha()]
        words = [w for w in words if not w in stop_words]
        #words = [porter.stem(word) for word in words]
```

```
            new_sentence.append(words)

        max_length = max([len(a) for a in new_sentence])
        word_vectors = []
        vocabulary = {}
        count = 1
        for sentence in new_sentence:
            word_vectors.append(sentence + ["</s>"] * (max_length - len(sentence)))
            for word in sentence:
                if word not in vocabulary:
                    vocabulary[word] = count
                    count = count + 1
        vocabulary["</s>"] = count
        wv = []
        for word_vector in word_vectors:
            wv.append([vocabulary[word] for word in word_vector])
        return np.array(wv), vocabulary


    def create_shuffle(x,y):
        '''
        Create an equal distribution of the positive and negative examples.
        Please do not change this particular shuffling method.
        '''
        pos_len= len(pos_data)
        neg_len= len(neg_data)
        pos_len_train= int(0.8*pos_len)
        neg_len_train= int(0.8*neg_len)
        train_data= [(x[i],y[i]) for i in range(0, pos_len_train)]
        train_data.extend([(x[i],y[i]) for i in range(pos_len, pos_len+ neg_len_train )])
        test_data=[(x[i],y[i]) for i in range(pos_len_train, pos_len)]
        test_data.extend([(x[i],y[i]) for i in range(pos_len+ neg_len_train, len(x) )])

        random.shuffle(train_data)
        x_train=[i[0] for i in train_data]
        y_train=[i[1] for i in train_data]
        random.shuffle(test_data)
        x_test=[i[0] for i in test_data]
        y_test=[i[1] for i in test_data]

        x_train=np.array(x_train)
        y_train=np.array(y_train)
        x_test= np.array(x_test)
        y_test= np.array(y_test)
        return x_train, y_train, x_test, y_test

    x_train, y_train, x_test, y_test= create_shuffle(x,y)
```

```
208 1298
```

```
In [47]: x, vocabulary = create_word_vectors(sentences)
         print(x.shape)
         vocab_size = len(vocabulary)
         sent_size = x.shape[1]
         print(x_train.shape)
```

```
(1506, 68)
(1204, 68)
```

```
In [52]: def sym_gen(batch_size=20, sentences_size=sent_size, num_embed=200,
                     num_label=2, filter_list=[2, 3, 4, 5], num_filter=100,
                     dropout=0.1):

             input_x = mx.sym.Variable('data')
             input_y = mx.sym.Variable('softmax_label')

             # embedding layer
             embed_layer = mx.sym.Embedding(data=input_x,
                                            input_dim=vocab_size,
                                            output_dim=num_embed,
                                            name='vocab_embed')
             conv_input = mx.sym.Reshape(data=embed_layer, target_shape=(batch_size, 1, sentence

             # create convolution + (max) pooling layer for each filter operation
             pooled_outputs = []
             for i, filter_size in enumerate(filter_list):
                 convi = mx.sym.Convolution(data=conv_input, kernel=(filter_size, num_embed), nu
                 relui = mx.sym.Activation(data=convi, act_type='relu')
                 pooli = mx.sym.Pooling(data=relui, pool_type='max', kernel=(sentences_size - fi
                 pooled_outputs.append(pooli)

             # combine all pooled outputs
             total_filters = num_filter * len(filter_list)
             concat = mx.sym.Concat(*pooled_outputs, dim=1)
             h_pool = mx.sym.Reshape(data=concat, target_shape=(batch_size, total_filters))

             # dropout layer
             if dropout > 0.0:
                 h_drop = mx.sym.Dropout(data=h_pool, p=dropout)
             else:
                 h_drop = h_pool

             # fully connected
             cls_weight = mx.sym.Variable('cls_weight')
```

```
        cls_bias = mx.sym.Variable('cls_bias')

        fc = mx.sym.FullyConnected(data=h_drop, weight=cls_weight, bias=cls_bias, num_hidde

        # softmax output
        sm = mx.sym.SoftmaxOutput(data=fc, label=input_y, name='softmax')

        return sm, ('data',), ('softmax_label',)
```

In [59]:
```
def train(symbol_data, train_iterator, valid_iterator, data_column_names, target_names)
    devs = mx.cpu()  # default setting
    module = mx.mod.Module(symbol_data, data_names=data_column_names, label_names=targe
    module.fit(train_data=train_iterator,
               eval_data=valid_iterator,
               eval_metric="acc",
               optimizer='RMSProp',
               optimizer_params={'learning_rate': 0.005},
               initializer=mx.initializer.Xavier(),
               num_epoch=10,
               batch_end_callback=mx.callback.Speedometer(20, 60))

train_set = mx.io.NDArrayIter(x_train, y_train, batch_size=20)
test_set = mx.io.NDArrayIter(x_test, y_test, batch_size=20)
sym_data, names_data, names_label = sym_gen()
```

In [64]:
```
model = train(sym_data, train_set, test_set, names_data, names_label)
```

```
INFO:root:Epoch[0] Batch [60]        Speed: 258.03 samples/sec        accuracy=0.886885


Epoch[0] Batch [60]        Speed: 258.03 samples/sec        accuracy=0.886885


INFO:root:Epoch[0] Train-accuracy=0.886885


Epoch[0] Train-accuracy=0.886885


INFO:root:Epoch[0] Time cost=4.716


Epoch[0] Time cost=4.716


INFO:root:Epoch[0] Validation-accuracy=0.900000


Epoch[0] Validation-accuracy=0.900000
```

```
INFO:root:Epoch[1] Batch [60]	Speed: 247.41 samples/sec	accuracy=0.969672

Epoch[1] Batch [60]	Speed: 247.41 samples/sec	accuracy=0.969672

INFO:root:Epoch[1] Train-accuracy=0.969672

Epoch[1] Train-accuracy=0.969672

INFO:root:Epoch[1] Time cost=4.904

Epoch[1] Time cost=4.904

INFO:root:Epoch[1] Validation-accuracy=0.893750

Epoch[1] Validation-accuracy=0.893750

INFO:root:Epoch[2] Batch [60]	Speed: 241.46 samples/sec	accuracy=0.992623

Epoch[2] Batch [60]	Speed: 241.46 samples/sec	accuracy=0.992623

INFO:root:Epoch[2] Train-accuracy=0.992623

Epoch[2] Train-accuracy=0.992623

INFO:root:Epoch[2] Time cost=5.010

Epoch[2] Time cost=5.010

INFO:root:Epoch[2] Validation-accuracy=0.903125

Epoch[2] Validation-accuracy=0.903125

INFO:root:Epoch[3] Batch [60]	Speed: 240.12 samples/sec	accuracy=0.996721
```

```
Epoch[3] Batch [60]        Speed: 240.12 samples/sec        accuracy=0.996721

INFO:root:Epoch[3] Train-accuracy=0.996721

Epoch[3] Train-accuracy=0.996721

INFO:root:Epoch[3] Time cost=5.040

Epoch[3] Time cost=5.040

INFO:root:Epoch[3] Validation-accuracy=0.896875

Epoch[3] Validation-accuracy=0.896875

INFO:root:Epoch[4] Batch [60]        Speed: 263.11 samples/sec        accuracy=0.996721

Epoch[4] Batch [60]        Speed: 263.11 samples/sec        accuracy=0.996721

INFO:root:Epoch[4] Train-accuracy=0.996721

Epoch[4] Train-accuracy=0.996721

INFO:root:Epoch[4] Time cost=4.601

Epoch[4] Time cost=4.601

INFO:root:Epoch[4] Validation-accuracy=0.900000

Epoch[4] Validation-accuracy=0.900000

INFO:root:Epoch[5] Batch [60]        Speed: 247.88 samples/sec        accuracy=0.997541

Epoch[5] Batch [60]        Speed: 247.88 samples/sec        accuracy=0.997541
```

```
INFO:root:Epoch[5] Train-accuracy=0.997541

Epoch[5] Train-accuracy=0.997541

INFO:root:Epoch[5] Time cost=4.884

Epoch[5] Time cost=4.884

INFO:root:Epoch[5] Validation-accuracy=0.896875

Epoch[5] Validation-accuracy=0.896875

INFO:root:Epoch[6] Batch [60]        Speed: 252.02 samples/sec        accuracy=0.998361

Epoch[6] Batch [60]        Speed: 252.02 samples/sec        accuracy=0.998361

INFO:root:Epoch[6] Train-accuracy=0.998361

Epoch[6] Train-accuracy=0.998361

INFO:root:Epoch[6] Time cost=4.813

Epoch[6] Time cost=4.813

INFO:root:Epoch[6] Validation-accuracy=0.903125

Epoch[6] Validation-accuracy=0.903125

INFO:root:Epoch[7] Batch [60]        Speed: 274.14 samples/sec        accuracy=0.998361

Epoch[7] Batch [60]        Speed: 274.14 samples/sec        accuracy=0.998361

INFO:root:Epoch[7] Train-accuracy=0.998361
```

```
Epoch[7] Train-accuracy=0.998361

INFO:root:Epoch[7] Time cost=4.419

Epoch[7] Time cost=4.419

INFO:root:Epoch[7] Validation-accuracy=0.900000

Epoch[7] Validation-accuracy=0.900000

INFO:root:Epoch[8] Batch [60]        Speed: 255.46 samples/sec        accuracy=0.998361

Epoch[8] Batch [60]        Speed: 255.46 samples/sec        accuracy=0.998361

INFO:root:Epoch[8] Train-accuracy=0.998361

Epoch[8] Train-accuracy=0.998361

INFO:root:Epoch[8] Time cost=4.727

Epoch[8] Time cost=4.727

INFO:root:Epoch[8] Validation-accuracy=0.903125

Epoch[8] Validation-accuracy=0.903125

INFO:root:Epoch[9] Batch [60]        Speed: 295.75 samples/sec        accuracy=0.998361

Epoch[9] Batch [60]        Speed: 295.75 samples/sec        accuracy=0.998361

INFO:root:Epoch[9] Train-accuracy=0.998361

Epoch[9] Train-accuracy=0.998361
```

```
INFO:root:Epoch[9] Time cost=4.097

Epoch[9] Time cost=4.097

INFO:root:Epoch[9] Validation-accuracy=0.900000

Epoch[9] Validation-accuracy=0.900000
```

In [ ]:

In [1]:
```
'''
1) Would the results improve if instead of using the word embeddings that is based
solely on frequency, if you have been able to incorporate sub-word information
   (In short run fasttext on the corpus and use the word embeddings generated by fasttxt

2) Accuracy might not be the best way to measure the performance of a skewed dataset.
What other metrics would you use ? Why?
   Experiment with different hyper-paramters to show the performance in terms of metric?
   You can assume that we want to identify all the medically relevant tweets (i.e. tweet
   with 'yes' class more). (7)


Delivearbles:

The ipython notebook with the results to each part of the question.

'''
```

Out[1]:
```
"\n1) Would the results improve if instead of using the word embeddings that is based\ns
```

In [ ]: