[COM6513] Assignment 1: Sentiment Analysis with Logistic Regression

Instructor: Nikos Aletras

The goal of this assignment is to develop and test a **text classification** system for **sentiment analysis**, in particular to predict the sentiment of movie reviews, i.e. positive or negative (binary classification).

For that purpose, you will implement:

- Text processing methods for extracting Bag-Of-Word features, using
  - n-grams (BOW), i.e. unigrams, bigrams and trigrams to obtain vector representations of documents where n=1,2,3 respectively. Two vector weighting schemes should be tested: (1) raw frequencies (**1 mark**); (2) tf.idf (**1 mark**).
  - character n-grams (BOCN). A character n-gram is a contiguous sequence of characters given a word, e.g. for n=2, 'coffee' is split into {'co', 'of', 'ff', 'fe', 'ee'}. Two vector weighting schemes should be tested: (1) raw frequencies (**1 mark**); (2) tf.idf (**1 mark**). **Tip: Note the large vocabulary size!**
  - a combination of the two vector spaces (n-grams and character n-grams) choosing your best performing wighting respectively (i.e. raw or tfidf). (**1 mark**) **Tip: you should merge the two representations**

- Binary Logistic Regression (LR) classifiers that will be able to accurately classify movie reviews trained with:
  - (1) BOW-count (raw frequencies)
  - (2) BOW-tfidf (tf.idf weighted)
  - (3) BOCN-count
  - (4) BOCN-tfidf
  - (5) BOW+BOCN (best performing weighting; raw or tfidf)

- The Stochastic Gradient Descent (SGD) algorithm to estimate the parameters of your Logistic Regression models. Your SGD algorithm should:
  - Minimise the Binary Cross-entropy loss function (**1 mark**)
  - Use L2 regularisation (**1 mark**)
  - Perform multiple passes (epochs) over the training data (**1 mark**)
  - Randomise the order of training data after each pass (**1 mark**)
  - Stop training if the difference between the current and previous development loss is smaller than a threshold (**1 mark**)
  - After each epoch print the training and development loss (**1 mark**)

- Discuss how did you choose hyperparameters (e.g. learning rate and regularisation strength) for each LR model? You should use a table showing model performance using different set of hyperparameter values. (**2 marks).** Tip: Instead of using all possible combinations, you could perform a random sampling of combinations.**

- After training each LR model, plot the learning process (i.e. training and validation loss in each epoch) using a line plot. Does your model underfit, overfit or is it about right? Explain why. (**1 mark**).

- Identify and show the most important features (model interpretability) for each class (i.e. top-10 most positive and top-10 negative weights). Give the top 10 for each class and comment on whether they make sense (if they don't you might have a bug!). If you were to apply the classifier into a different domain such laptop reviews or restaurant reviews, do you think these features would generalise well? Can you propose what features the classifier could pick up as important in the new domain? (**2 marks**)

- Provide well documented and commented code describing all of your choices. In general, you are free to make decisions about text processing (e.g. punctuation, numbers, vocabulary size) and hyperparameter values. We expect to see justifications and discussion for all of your choices (**2 marks**).

- Provide efficient solutions by using Numpy arrays when possible (you can find tips in Lab 1 sheet). Executing the whole notebook with your code should not take more than 5 minutes on a any standard computer (e.g. Intel Core i5 CPU, 8 or 16GB RAM) excluding hyperparameter tuning runs (**2 marks**).

Data

The data you will use are taken from here: http://www.cs.cornell.edu/people/pabo/movie-review-data/ and you can find it in the `./data_sentiment` folder in CSV format:

- `data_sentiment/train.csv` : contains 1,400 reviews, 700 positive (label: 1) and 700 negative (label: 0) to be used for training.
- `data_sentiment/dev.csv` : contains 200 reviews, 100 positive and 100 negative to be used for hyperparameter selection and monitoring the training process.
- `data_sentiment/test.csv` : contains 400 reviews, 200 positive and 200 negative to be used for testing.

Submission Instructions

You should submit a Jupyter Notebook file (assignment1.ipynb) and an exported PDF version (you can do it from Jupyter: `File->Download as->PDF via Latex` or you can print it as PDF using your browser).

You are advised to follow the code structure given in this notebook by completing all given funtions. You can also write any auxilliary/helper functions (and arguments for the functions) that you might need but note that you can provide a full solution without any such functions. Similarly, you can just use only the packages imported below but you are free to use any functionality from the [Python Standard Library](https://docs.python.org/2/library/index.html), NumPy, SciPy (excluding built-in softmax funtcions) and Pandas. You are not allowed to use any third-party library such as Scikit-learn (apart from metric functions already provided), NLTK, Spacy, Keras etc..

There is no single correct answer on what your accuracy should be, but correct implementations usually achieve F1-scores around 80\% or higher. The quality of the analysis of the results is as important as the accuracy itself.

This assignment will be marked out of 20. It is worth 20\% of your final grade in the module.

The deadline for this assignment is **23:59 on Mon, 14 Mar 2022** and it needs to be submitted via Blackboard. Standard departmental penalties for lateness will be applied. We use a range of strategies to **detect unfair means**, including Turnitin which helps detect plagiarism. Use of unfair means would result in getting a failing grade.

In [1]:

```
tune_params = input("tune hyperparameters and show model performance? [Y/N]")
```

```
tune hyperparameters and show model performance? [Y/N]Y
```

In [2]:

```
while (tune_params != 'Y') & (tune_params != 'N'):
    tune_params = input("tune hyperparameters and show model performance? [Y/N]")
```

In [3]:

```
import pandas as pd
import numpy as np
from collections import Counter
import re
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import random
from prettytable import PrettyTable

# fixing random seed for reproducibility
random.seed(123)
np.random.seed(123)
```

Load Raw texts and labels into arrays

First, you need to load the training, development and test sets from their corresponding CSV files (tip: you can use Pandas dataframes).

In [4]:

```
training_data = pd.read_csv("data_sentiment/train.csv")
development_data = pd.read_csv("data_sentiment/dev.csv")
test_data = pd.read_csv("data_sentiment/test.csv")
```

If you use Pandas you can see a sample of the data.

In [5]:

```
# training_data.sample(5)
```

The next step is to put the raw texts into Python lists and their corresponding labels into NumPy arrays:

In [6]:

```
train_content = training_data.iloc[:,0].to_list()
development_content = development_data.iloc[:,0].to_list()
test_content = test_data.iloc[:,0].to_list()
train_label = training_data.iloc[:,1].to_numpy()
development_label = development_data.iloc[:,1].to_numpy()
test_label = test_data.iloc[:,1].to_numpy()
```

Vector Representations of Text

To train and test Logisitc Regression models, you first need to obtain vector representations for all documents given a vocabulary of features (unigrams, bigrams, trigrams).

Text Pre-Processing Pipeline

To obtain a vocabulary of features, you should:

- tokenise all texts into a list of unigrams (tip: using a regular expression)
- remove stop words (using the one provided or one of your preference)
- compute bigrams, trigrams given the remaining unigrams (or character ngrams from the unigrams)
- remove ngrams appearing in less than K documents
- use the remaining to create a vocabulary of unigrams, bigrams and trigrams (or character n-grams). You can keep top N if you encounter memory issues.

In [7]:

```
stop_words = ['a','in','on','at','and','or',
              'to', 'the', 'of', 'an', 'by',
              'as', 'is', 'was', 'were', 'been', 'be',
              'are','for', 'this', 'that', 'these', 'those', 'you', 'i',
             'it', 'he', 'she', 'we', 'they', 'will', 'have', 'has',
              'do', 'did', 'can', 'could', 'who', 'which', 'what',
             'his', 'her', 'they', 'them', 'from', 'with', 'its']
stop_words = ['a', 'able', 'about', 'above', 'according', 'accordingly', 'across', '
```

N-gram extraction from a document

You first need to implement the `extract_ngrams` function. It takes as input:

- `x_raw` : a string corresponding to the raw text of a document
- `ngram_range` : a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams.
- `token_pattern` : a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation.
- `stop_words` : a list of stop words
- `vocab` : a given vocabulary. It should be used to extract specific features.
- `char_ngrams` : boolean. If true the function extracts character n-grams

and returns:

- `x': a list of all extracted features.

See the examples below to see how this function should work.

In [8]:

```
def extract_ngrams(x_raw, ngram_range=(1,3), token_pattern=r'[A-Za-z]+',
                   stop_words=[], vocab=set(), char_ngrams=False):
    x = []
    x_raw = re.findall(token_pattern, x_raw)
    if char_ngrams==True:
        x_raw = ''.join(x_raw)
```

```
        for i in range(ngram_range[1]-ngram_range[0]+1):
            for j in range(len(x_raw)-ngram_range[0]-i+1):
                x.append([x_raw[j:j+ngram_range[0]+i]])
        return x

    words = [word for word in x_raw if word not in stop_words]
    for i in range(ngram_range[1]-ngram_range[0]+1):
        for j in range(len(words)-ngram_range[0]-i+1):
            x.append(words[j:j+ngram_range[0]+i])
    return x
```

Note that it is OK to represent n-grams using lists instead of tuples: e.g. `['great', ['great', 'movie']]`

For extracting character n-grams the function should work as follows:

In [9]:

```
extract_ngrams("movie",
               ngram_range=(2,4),
               stop_words=[],
               char_ngrams=True)
```

Out[9]:

```
[['mo'], ['ov'], ['vi'], ['ie'], ['mov'], ['ovi'], ['vie'], ['movi'], ['ovie']]
```

Create a vocabulary

The `get_vocab` function will be used to (1) create a vocabulary of ngrams; (2) count the document frequencies of ngrams; (3) their raw frequency. It takes as input:

- `X_raw` : a list of strings each corresponding to the raw text of a document
- `ngram_range` : a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams.
- `token_pattern` : a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation.
- `stop_words` : a list of stop words
- `min_df` : keep ngrams with a minimum document frequency.
- `keep_topN` : keep top-N more frequent ngrams.

and returns:

- `vocab` : a set of the n-grams that will be used as features.
- `df` : a Counter (or dict) that contains ngrams as keys and their corresponding document frequency as values.
- `ngram_counts` : counts of each ngram in vocab

Hint: it should make use of the `extract_ngrams` function.

In [10]:

```
def get_vocab(X_raw, ngram_range=(1,3), token_pattern=r'\w+',
              min_df=0, keep_topN=0,
              stop_words=[],char_ngrams=False):
    all_words = extract_ngrams(x_raw=str(X_raw), ngram_range=ngram_range, stop_words

    # compute term frequency
    ngram_counts = {}
    for word in all_words:
        temp = ' '.join(word)
```

```
        if temp not in ngram_counts:
            ngram_counts[temp] = 1
        else:
            ngram_counts[temp] = ngram_counts[temp] + 1
    ngram_counts = {key:val for key, val in ngram_counts.items() if val >= keep_topN

    # compute df by using words in ngram_counts in order to be efficient.
    # The reason I can do this is that I only care about the words in ngram_counts i
    df = {}
    for i in ngram_counts.keys():
        counts = 0
        for j in range(len(X_raw)):
            if i in X_raw[j]:
                counts += 1
        df[i] = counts
    df = {key:val for key, val in df.items() if val >= min_df}

    # use the words in df to create vocab set
    # the reason to choose df is that dict df is smaller than dict ngram_counts and
    vocab = [[i] for i in df.keys()]

    return vocab, df, ngram_counts
```

Now you should use `get_vocab` to create your vocabulary and get document and raw frequencies of n-grams:

In [11]:

```
vocab, df, tf = get_vocab(X_raw=train_content,min_df=55, keep_topN=75, stop_words=st
```

Then, you need to create 2 dictionaries: (1) vocabulary id -> word; and (2) word -> vocabulary id so you can use them for reference:

In [12]:

```
vocab_to_id = {}
id_to_vocab = {}
for word in vocab:
    string = " ".join(word)
    vocab_to_id[string] = vocab.index(word)
    id_to_vocab[vocab.index(word)] = string
```

Now you should be able to extract n-grams for each text in the training, development and test sets:

In [13]:

```
# create a content that only consists of words in vocab
# because using the original content would affect the performance of bigram, trigram
def content_without_stopwords (raw_content, vocab):
    processed_content = [None]*len(raw_content)
    for i in range(len(raw_content)):
        raw_words = raw_content[i].split()
        resultwords  = [word for word in raw_words if [word] in vocab]
        processed_content[i] = ' '.join(resultwords)
    return processed_content
```

In [14]:

```
processed_train_content = content_without_stopwords(raw_content=train_content, vocab
processed_dev_content = content_without_stopwords(raw_content=development_content, v
processed_test_content = content_without_stopwords(raw_content=test_content, vocab=v
```

Vectorise documents

Next, write a function `vectoriser` to obtain Bag-of-ngram representations for a list of documents. The function should take as input:

- `X_ngram` : a list of texts (documents), where each text is represented as list of n-grams in the `vocab`
- `vocab` : a set of n-grams to be used for representing the documents

and return:

- `X_vec` : an array with dimensionality Nx|vocab| where N is the number of documents and |vocab| is the size of the vocabulary. Each element of the array should represent the frequency of a given n-gram in a document.

In [15]:

```
def vectorise(X_ngram, vocab):
    X_vec = np.empty([len(X_ngram), len(vocab)])
    for i in range(len(vocab)):
        string = " ".join(vocab[i])
        for j in range(len(X_ngram)):
            X_vec[j,i] = X_ngram[j].count(string)
    return X_vec
```

Finally, use `vectorise` to obtain document vectors for each document in the train, development and test set. You should extract both count and tf.idf vectors respectively:

Count vectors

In [16]:

```
train_vec = vectorise(X_ngram=processed_train_content, vocab=vocab)
development_vec = vectorise(X_ngram=processed_dev_content, vocab=vocab)
test_vec = vectorise(X_ngram=processed_test_content, vocab=vocab)
```

TF.IDF vectors

First compute `idfs` an array containing inverted document frequencies (Note: its elements should correspond to your `vocab`)

In [17]:

```
import math
idf = {}
for key, value in df.items():
    idf[key] = math.log(len(train_content)/value,10)
```

Then transform your count vectors to tf.idf vectors:

In [18]:

```
# create a list of idf values
idf_list = list(idf.values())
```

In [19]:

```
# define a function to compute tfidf because this function would be used many times
def compute_tfidf(idf_list, vec):
    tfidf = np.empty([(np.shape(vec)[0]), (np.shape(vec)[1])])
    for i in range(np.shape(vec)[0]):
        for j in range(np.shape(vec)[1]):
            tfidf[i,j] = vec[i,j]*idf_list[j]
    return tfidf
```

In [20]:
```
train_tfidf = compute_tfidf(idf_list=idf_list, vec=train_vec)
development_tfidf = compute_tfidf(idf_list=idf_list, vec=development_vec)
test_tfidf = compute_tfidf(idf_list=idf_list, vec=test_vec)
```

Binary Logistic Regression

After obtaining vector representations of the data, now you are ready to implement Binary Logistic Regression for classifying sentiment.

First, you need to implement the `sigmoid` function. It takes as input:

- `z` : a real number or an array of real numbers

and returns:

- `sig` : the sigmoid of `z`

In [21]:
```
def sigmoid(z):
    sig = 1/(1+np.exp(-z))
    return sig
```

Then, implement the `predict_proba` function to obtain prediction probabilities. It takes as input:

- `X` : an array of inputs, i.e. documents represented by bag-of-ngram vectors $(N \times |vocab|)$
- `weights` : a 1-D array of the model's weights $(1, |vocab|)$

and returns:

- `preds_proba` : the prediction probabilities of X given the weights

In [22]:
```
def predict_proba(X, weights):
    z = np.dot(X, weights)
    preds_proba = sigmoid(z)
    return preds_proba
```

Then, implement the `predict_class` function to obtain the most probable class for each vector in an array of input vectors. It takes as input:

- `X` : an array of documents represented by bag-of-ngram vectors $(N \times |vocab|)$
- `weights` : a 1-D array of the model's weights $(1, |vocab|)$

and returns:

- `preds_class` : the predicted class for each x in X given the weights

In [23]:
```
def predict_class(X, weights):
    preds_proba = predict_proba(X, weights)
    preds_class = np.zeros(len(preds_proba))
    for i in range(len(preds_proba)):
        if preds_proba[i] < 0.5:
            preds_class[i] = 0
        else:
```

```
            preds_class[i] = 1
    return preds_class
```

To learn the weights from data, we need to minimise the binary cross-entropy loss. Implement `binary_loss` that takes as input:

- `X` : input vectors
- `Y` : labels
- `weights` : model weights
- `alpha` : regularisation strength

and return:

- `l` : the loss score

In [24]:

```
# use array to calculate
def binary_loss(X, Y, weights, alpha=0.00001):
    '''
    Binary Cross-entropy Loss

    X:(len(X),len(vocab))
    Y: array len(Y)
    weights: array len(X)
    '''
    y = predict_proba(X, weights)
    l_first = (np.dot(np.transpose(-Y), (np.log(y))) - np.dot(np.transpose(1-Y), (np
    r = np.dot(np.transpose(weights), weights)
    l = l_first + r*alpha
    return l
```

Now, you can implement Stochastic Gradient Descent to learn the weights of your sentiment classifier. The `SGD` function takes as input:

- `X_tr` : array of training data (vectors)
- `Y_tr` : labels of `X_tr`
- `X_dev` : array of development (i.e. validation) data (vectors)
- `Y_dev` : labels of `X_dev`
- `lr` : learning rate
- `alpha` : regularisation strength
- `epochs` : number of full passes over the training data
- `tolerance` : stop training if the difference between the current and previous validation loss is smaller than a threshold
- `print_progress` : flag for printing the training progress (train/validation loss)

and returns:

- `weights` : the weights learned
- `training_loss_history` : an array with the average losses of the whole training set after each epoch
- `validation_loss_history` : an array with the average losses of the whole development set after each epoch

In [25]:

```
def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], lr=0.0001,
```

```
        alpha=0.001, epochs=180,
        tolerance=0.0000001, print_progress=False):
    w = np.zeros(np.shape(X_tr)[1])
    index = list(range(len(Y_tr)))
    training_loss_history = []
    validation_loss_history = []
    for i in range(epochs):
        random.Random(32*i).shuffle(index)
        for j in index:
            w = w - lr*(predict_proba(X_tr[j], w)-Y_tr[j])*X_tr[j]


        l = binary_loss(X_tr, Y_tr, w)
        l_val = binary_loss(X_dev, Y_dev, w)
        training_loss_history.append(l)
        validation_loss_history.append(l_val)
        if i>0:
            if abs(validation_loss_history[i-1]-validation_loss_history[i])<toleranc
                break
        weights = w

    if print_progress==True:
        print('train loss: ', training_loss_history)
        print('-------------------------------------------------')
        print('val loss: ', validation_loss_history)

    return weights, training_loss_history, validation_loss_history
```

In [26]:

```
# create a list of hyperparameters' values
lr = np.array(np.logspace(-6, -3, 4, endpoint=True))
alpha = np.array(np.logspace(-6, -3, 4, endpoint=True))
epochs = np.array([30, 75, 120, 165])
```

Train and Evaluate Logistic Regression with Count vectors

First train the model using SGD:

In [27]:

```
# to train model using different hyperparameters, including alpha, learning rate, an
if tune_params == 'Y':
    # create the table showing performance of each model. Here I use accuracy as the
    table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr3 = PrettyTable(["lr = {}".format(lr[3]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    acc_history = []
    # use for-loop to run each model.
    for i in range(len(lr)):
        for j in range(len(alpha)):
            for k in range(len(epochs)):
                w_count, t, v = SGD(X_tr=train_vec, Y_tr=train_label, X_dev=developm
                preds_te_count = predict_class(test_vec, w_count)
                if v[len(v)-1] < 1:
                    acc_history.append(accuracy_score(test_label,preds_te_count))
                else:
```

```
                acc_history.append(0)
        # add the accuracy into my tables
        if i == 0:
            table_lr0.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr0.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr0.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr0.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 1:
            table_lr1.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr1.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr1.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr1.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 2:
            table_lr2.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr2.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr2.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr2.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 3:
            table_lr3.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr3.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr3.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr3.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
    result = max(acc_history)
    print('max accuracy:', result)
    index_of_result = acc_history.index(result)
    lr_index = (index_of_result)//16
    alpha_index = (index_of_result%16)//4
    epochs_index = (index_of_result%4)
    print('lr: ', lr[lr_index])
    print('alpha: ', alpha[alpha_index])
    print('epochs: ', epochs[epochs_index])
    print(table_lr0)
    print(table_lr1)
    print(table_lr2)
    print(table_lr3)
```

```
max accuracy: 0.8245614035087719
lr:  0.001
alpha:  1e-06
epochs:  30
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-06  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.5739348370927319 | 0.5739348370927319 | 0.5739348370927319 | 0.57
39348370927319 |
| epochs = 75  | 0.6867167919799498 | 0.6867167919799498 | 0.6867167919799498 | 0.68
67167919799498 |
| epochs = 120 | 0.7192982456140351 | 0.7192982456140351 | 0.7192982456140351 | 0.71
92982456140351 |
| epochs = 165 | 0.7343358395989975 | 0.7343358395989975 | 0.7343358395989975 | 0.73
43358395989975 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-05  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.7769423558897243 | 0.7769423558897243 | 0.7769423558897243 | 0.77
69423558897243 |
| epochs = 75  | 0.7894736842105263 | 0.7894736842105263 | 0.7894736842105263 | 0.78
94736842105263 |
```

```
| epochs = 120 | 0.7869674185463659 | 0.7869674185463659 | 0.7869674185463659 | 0.78
69674185463659 |
| epochs = 165 | 0.7869674185463659 | 0.7869674185463659 | 0.7869674185463659 | 0.78
69674185463659 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| lr = 0.0001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.8070175438596491 | 0.8070175438596491 | 0.8070175438596491 | 0.80
70175438596491 |
| epochs = 75  | 0.8095238095238095 | 0.8095238095238095 | 0.8095238095238095 | 0.80
95238095238095 |
| epochs = 120 | 0.8220551378446115 | 0.8220551378446115 | 0.8220551378446115 | 0.82
20551378446115 |
| epochs = 165 | 0.8220551378446115 | 0.8220551378446115 | 0.8220551378446115 | 0.82
20551378446115 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 0.001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.8245614035087719 | 0.8245614035087719 | 0.8245614035087719 | 0.82
45614035087719 |
| epochs = 75  | 0.8245614035087719 | 0.8245614035087719 | 0.8245614035087719 | 0.82
45614035087719 |
| epochs = 120 | 0.8195488721804511 | 0.8195488721804511 | 0.8195488721804511 | 0.81
95488721804511 |
| epochs = 165 | 0.8145363408521303 | 0.8145363408521303 | 0.8145363408521303 | 0.81
45363408521303 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
```

The table above shows that if the learning rate is small, the model needs a lot of training(epochs) to have a better performance.

In [28]:

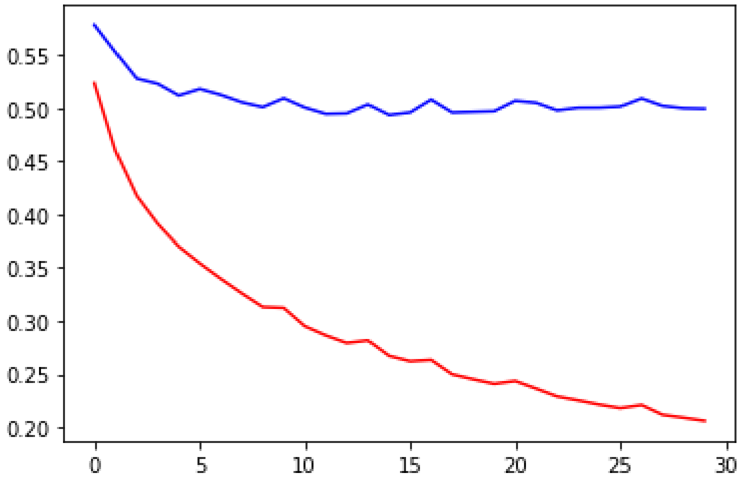
```python
# if choose to tune parameters, show the best model; otherwise, use the default valu
if tune_params == 'Y':
    w_count, t, v = SGD(train_vec, train_label, X_dev=development_vec, Y_dev=develop
                        lr=lr[lr_index], alpha=alpha[alpha_index], epochs=epochs[epochs_
else:
    w_count, t, v = SGD(train_vec, train_label, X_dev=development_vec, Y_dev=develop
```

```
train loss:  [0.5237446231674533, 0.4598958396943405, 0.417625139762551, 0.391316598
17128767, 0.3699847163197355, 0.3542820166760697, 0.33979873888380074, 0.32613765592
52188, 0.3132001307441163, 0.3123859198456283, 0.29517765831364134, 0.28649444634258
97, 0.27930319008959525, 0.28170348646284504, 0.2671498248065886, 0.262105907093944
9, 0.2633544590032282, 0.2497121788012522, 0.2451044812534158, 0.24079316510853427,
0.2434685396017481, 0.23631375180526418, 0.22893170952370084, 0.22523144680973398,
0.22126867620071952, 0.21791808774402951, 0.22086096306930622, 0.21159902472315276,
0.2088484826319793, 0.20590085801287555]
------------------------------------------------
val loss:  [0.578607392784173, 0.5528542875089766, 0.528184863434301, 0.523228340722
3768, 0.5121228999596672, 0.5182959445165406, 0.5127358809343335, 0.505759101222060
1, 0.5011817186911576, 0.5095155258594161, 0.5008169775709146, 0.49481775392802796,
0.49526182218644393, 0.5036289055091067, 0.4937563872055492, 0.49617816835467193, 0.
5082344376684081, 0.4960988775376062, 0.4967311056685078, 0.49730780530622326, 0.507
1129271308218, 0.5052426025109235, 0.4981088618185785, 0.5005587753279394, 0.5006436
754512664, 0.5017202407598437, 0.5093331363617594, 0.502325959591172, 0.500091507004
2214, 0.499669948276321]
```

Now plot the training and validation history per epoch for the best hyperparameter combination. Does your model underfit, overfit or is it about right? Explain why.

In [29]:
```
plt.plot(t, 'r')
plt.plot(v, 'b')
```

Out[29]: [<matplotlib.lines.Line2D at 0x18262fa99a0>]



The final val loss is about 0.5, which is acceptable. The curve of val loss is descending without a trend of ascending. This means the model is neither overfitting nor underfitting.

BOW-count	Precision	Recall	F1-Score
lr = 0.0001			
BOW-tfidf			
BOCN-count			
BOCN-tfidf			
BOW+BOCN			

Evaluation

Compute accuracy, precision, recall and F1-scores:

In [30]:
```
preds_te_count = predict_class(test_vec, w_count)
print('Accuracy:', accuracy_score(test_label,preds_te_count))
print('Precision:', precision_score(test_label,preds_te_count))
print('Recall:', recall_score(test_label,preds_te_count))
print('F1-Score:', f1_score(test_label,preds_te_count))
```

```
Accuracy: 0.8245614035087719
Precision: 0.8177339901477833
Recall: 0.8341708542713567
F1-Score: 0.8258706467661691
```

Finally, print the top-10 words for the negative and positive class respectively.

In [31]:
```
top_neg = w_count.argsort()[:10]
for i in top_neg:
    print(id_to_vocab[i])
```

```
bad
supposed
worst
waste
```

```
boring
attempt
ridiculous
script
write
stupid
```

In [32]:

```
top_pos = w_count.argsort()[::-1][:10]
for i in top_pos:
    print(id_to_vocab[i])
```

```
perfect
hilarious
excellent
enjoy
simple
great
true
memorable
movies
bit
```

The words 'bad', 'worst', 'waste', etc are used to express the negative feelings of a movie. The words 'perfect', 'enjoy', 'excellent', etc are used to express the positive feelings of a movie. As a result, this model is quite excellent.

If we were to apply the classifier we've learned into a different domain such laptop reviews or restaurant reviews, do you think these features would generalise well? Can you propose what features the classifier could pick up as important in the new domain?

ANS: No, the words used to express positive or negative feelings about a movie is different from those about a restaurant or laptop. For the restaurant reviews, words such as delicious, friendly, dirty, etc may be the features.

Discuss how did you choose model hyperparameters (e.g. learning rate and regularisation strength)? What is the relation between training epochs and learning rate? How the regularisation strength affects performance?

First, I create lists for learning rate, alpha, and epochs, each with four values. Then, use for loop to train the models and record the accuracy for each model if the final val loss is under 1. Last, pick up the model with the highest accuracy. The reason I use accuracy to choose the model is that both true positive and true negative are important in this model.Both of the positive reviews and negative reviews are important so the accuracy is what I need

If learning rate is small, the epochs needs to be large enough since the weight only changes a little during each epoch. If the learning rate is large, the epochs can be smaller. However, a learning rate that is too large results in unstable training and a learning rate that is too small results in failed training.

The larger the regularisation strength is, the bigger penalty term is. If the regularisation strength is near 0, the penalty term doesn't work, which means the model has a high possibility of overfitting.

Train and Evaluate Logistic Regression with TF.IDF vectors

Follow the same steps as above (i.e. evaluating count n-gram representations).

BOW-tfidf

In [33]:

```
if tune_params == 'Y':
    acc_history = []
    table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr3 = PrettyTable(["lr = {}".format(lr[3]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    for i in range(len(lr)):
        for j in range(len(alpha)):
            for k in range(len(epochs)):
                w_tfidf, t_tfidf, v_tfidf = SGD(train_tfidf, train_label, X_dev=deve
                preds_te_count = predict_class(test_tfidf, w_tfidf)
                if v_tfidf[len(v_tfidf)-1] < 1:
                    acc_history.append(accuracy_score(test_label,preds_te_count))
                else:
                    acc_history.append(0)
        if i == 0:
            table_lr0.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr0.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr0.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr0.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 1:
            table_lr1.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr1.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr1.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr1.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 2:
            table_lr2.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr2.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr2.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr2.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 3:
            table_lr3.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr3.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr3.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr3.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
    result = max(acc_history)
    print('max accuracy:', result)
    index_of_result = acc_history.index(result)
    lr_index = (index_of_result)//16
    alpha_index = (index_of_result%16)//4
    epochs_index = (index_of_result%4)
    print('lr: ', lr[lr_index])
    print('alpha: ', alpha[alpha_index])
    print('epochs: ', epochs[epochs_index])
    print(table_lr0)
    print(table_lr1)
    print(table_lr2)
    print(table_lr3)
```

```
max accuracy: 0.8421052631578947
lr:  0.001
alpha:  1e-06
epochs:  30
+--------------+-------------------+-------------------+-------------------+-----
---------------+
```

```
|  lr = 1e-06  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001   |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.6090225563909775 | 0.6090225563909775 | 0.6090225563909775 | 0.60
90225563909775 |
| epochs = 75  | 0.6265664160401002 | 0.6265664160401002 | 0.6265664160401002 | 0.62
65664160401002 |
| epochs = 120 | 0.6491228070175439 | 0.6491228070175439 | 0.6491228070175439 | 0.64
91228070175439 |
| epochs = 165 | 0.6691729323308271 | 0.6691729323308271 | 0.6691729323308271 | 0.66
91729323308271 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-05  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001   |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.7243107769423559 | 0.7243107769423559 | 0.7243107769423559 | 0.72
43107769423559 |
| epochs = 75  | 0.7694235588972431 | 0.7694235588972431 | 0.7694235588972431 | 0.76
94235588972431 |
| epochs = 120 | 0.7794486215538847 | 0.7794486215538847 | 0.7794486215538847 | 0.77
94486215538847 |
| epochs = 165 | 0.7894736842105263 | 0.7894736842105263 | 0.7894736842105263 | 0.78
94736842105263 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| lr = 0.0001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001   |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.8020050125313283 | 0.8020050125313283 | 0.8020050125313283 | 0.80
20050125313283 |
| epochs = 75  | 0.8245614035087719 | 0.8245614035087719 | 0.8245614035087719 | 0.82
45614035087719 |
| epochs = 120 | 0.8370927318295739 | 0.8370927318295739 | 0.8370927318295739 | 0.83
70927318295739 |
| epochs = 165 | 0.8370927318295739 | 0.8370927318295739 | 0.8370927318295739 | 0.83
70927318295739 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 0.001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001   |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.8421052631578947 | 0.8421052631578947 | 0.8421052631578947 | 0.84
21052631578947 |
| epochs = 75  | 0.8421052631578947 | 0.8421052631578947 | 0.8421052631578947 | 0.84
21052631578947 |
| epochs = 120 | 0.8370927318295739 | 0.8370927318295739 | 0.8370927318295739 | 0.83
70927318295739 |
| epochs = 165 | 0.8421052631578947 | 0.8421052631578947 | 0.8421052631578947 | 0.84
21052631578947 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
```

In [34]:

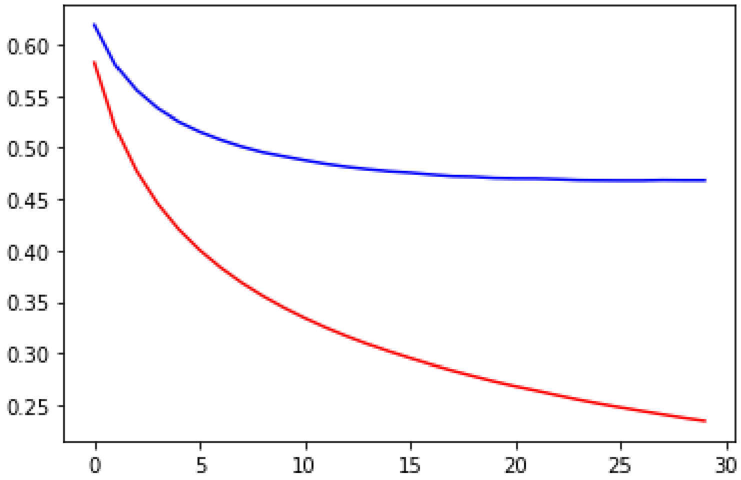
```
if tune_params == 'Y':
    w_tfidf, t_tfidf, v_tfidf = SGD(train_tfidf, train_label, X_dev=development_tfid
                                    lr=lr[lr_index], alpha=alpha[alpha_index], epochs=ep
else:
    w_tfidf, t_tfidf, v_tfidf = SGD(train_tfidf, train_label, X_dev=development_tfid
```

```
train loss:  [0.5821423604447067, 0.5191824525351423, 0.4771485270391641, 0.44551052
09681702, 0.4206253516438873, 0.4004119687788791, 0.3834264214319399, 0.368736498302
26524, 0.3559369071707424, 0.34458945221997633, 0.33442878925625347, 0.3251911145364
116, 0.31680218695582163, 0.30913201678853824, 0.30205401863364145, 0.29561387714164
516, 0.289374353545951, 0.28366865282050013, 0.2783334226875282, 0.2731890999786553,
0.26847640662830324, 0.2642349730304086, 0.2598222304485384, 0.25554997408348923, 0.
25167316304893955, 0.24800368139970355, 0.24449192831281358, 0.24118190275631513, 0.
2378454268603819, 0.23487269208847114]
------------------------------------------------
val loss:  [0.6188721607708753, 0.5797247672149108, 0.5554238039052309, 0.5379952194
868945, 0.5250780603283466, 0.5154174859931234, 0.5076522389648609, 0.50101538302323
92, 0.4955418407020135, 0.4915979604982202, 0.4878159080322553, 0.4843017757290691,
0.4814125952698l806, 0.47906874017566115, 0.47705445595394064, 0.47567849357640823,
0.473862015044236З, 0.4723020879546861, 0.4717916307729878, 0.47053285033939973, 0.4
7005192421152897, 0.4699817529210617, 0.4693524439525947, 0.46849659303745517, 0.468
2221147447083, 0.46808057061172914, 0.46807520409825015, 0.4685228947778664, 0.46833
299473138323, 0.46827970845677463]
```

In [35]:

```python
plt.plot(t_tfidf, 'r')
plt.plot(v_tfidf, 'b')
```

Out[35]: [<matplotlib.lines.Line2D at 0x18263043340>]



The final val loss is about 0.5, which is acceptable. The curve of val loss is descending without a trend of ascending. This means the model is neither overfitting nor underfitting.

In [36]:

```python
preds_te_count = predict_class(test_tfidf, w_tfidf)
print('Accuracy:', accuracy_score(test_label,preds_te_count))
print('Precision:', precision_score(test_label,preds_te_count))
print('Recall:', recall_score(test_label,preds_te_count))
print('F1-Score:', f1_score(test_label,preds_te_count))
```

```
Accuracy: 0.8421052631578947
Precision: 0.8333333333333334
Recall: 0.8542713567839196
F1-Score: 0.8436724565756824
```

In [37]:

```python
top_neg = w_tfidf.argsort()[:10]
for i in top_neg:
    print(id_to_vocab[i])
```

```
bad
worst
supposed
boring
waste
ridiculous
```

```
awful
fails
script
filmmakers
```

In [38]:

```
top_pos = w_tfidf.argsort()[::-1][:10]
for i in top_pos:
    print(id_to_vocab[i])
```

```
perfect
hilarious
great
excellent
terrific
memorable
simple
perfectly
world
enjoyed
```

The words 'bad', 'worst', 'boring', etc are used to express the negative feelings of a movie. The words 'perfect', 'terrific', 'memorable', etc are used to express the positive feelings of a movie. As a result, this model is quite excellent.

Now repeat the training and evaluation process for BOW-tfidf, BOCN-count, BOCN-tfidf, BOW+BOCN including hyperparameter tuning for each model...

BOCN-count

In [39]:

```
vocab_bocn, df_bocn, tf_bocn = get_vocab(X_raw=train_content,min_df=55, keep_topN=75
```

In [40]:

```
vocab_bocn_to_id = {}
id_to_vocab_bocn = {}
for word in vocab_bocn:
    string = " ".join(word)
    vocab_bocn_to_id[string] = vocab_bocn.index(word)
    id_to_vocab_bocn[vocab_bocn.index(word)] = string
```

In [41]:

```
train_vec_bocn = vectorise(X_ngram=processed_train_content, vocab=vocab_bocn)
development_vec_bocn = vectorise(X_ngram=processed_dev_content, vocab=vocab_bocn)
test_vec_bocn = vectorise(X_ngram=processed_test_content, vocab=vocab_bocn)
```

In [42]:

```
import math
idf_bocn = {}
for key, value in df_bocn.items():
    idf_bocn[key] = math.log(len(train_content)/value,10)
```

In [43]:

```
idf_bocn_list = list(idf_bocn.values())
```

In [44]:

```
train_tfidf_bocn = compute_tfidf(idf_list=idf_bocn_list, vec=train_vec_bocn)
development_tfidf_bocn = compute_tfidf(idf_list=idf_bocn_list, vec=development_vec_b
test_tfidf_bocn = compute_tfidf(idf_list=idf_bocn_list, vec=test_vec_bocn)
```

In [45]:
```
if tune_params == 'Y':
    acc_history = []
    table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr3 = PrettyTable(["lr = {}".format(lr[3]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    for i in range(len(lr)):
        for j in range(len(alpha)):
            for k in range(len(epochs)):
                w_count_bocn, t_bocn, v_bocn = SGD(train_vec_bocn, train_label, X_de
                preds_te_count = predict_class(test_vec_bocn, w_count_bocn)
                if v_bocn[len(v_bocn)-1] < 1:
                    acc_history.append(accuracy_score(test_label,preds_te_count))
                else:
                    acc_history.append(0)
        if i == 0:
            table_lr0.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr0.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr0.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr0.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 1:
            table_lr1.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr1.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr1.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr1.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 2:
            table_lr2.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr2.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr2.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr2.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 3:
            table_lr3.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr3.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr3.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr3.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
    result = max(acc_history)
    print('max accuracy:', result)
    index_of_result = acc_history.index(result)
    lr_index = (index_of_result)//16
    alpha_index = (index_of_result%16)//4
    epochs_index = (index_of_result%4)
    print('lr: ', lr[lr_index])
    print('alpha: ', alpha[alpha_index])
    print('epochs: ', epochs[epochs_index])
    print(table_lr0)
    print(table_lr1)
    print(table_lr2)
    print(table_lr3)
```

```
<ipython-input-24-1c73d9e9e108>:11: RuntimeWarning: divide by zero encountered in lo
g
  l_first = (np.dot(np.transpose(-Y), (np.log(y))) - np.dot(np.transpose(1-Y), (np.l
og(1-y))))/len(Y)
max accuracy: 0.7869674185463659
lr:  1e-05
alpha:  1e-06
epochs:  120
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-06  |    alpha = 1e-06    |    alpha = 1e-05    |    alpha = 0.0001    |    al
```

```
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.7042606516290727 | 0.7042606516290727 | 0.7042606516290727 | 0.70
42606516290727 |
| epochs = 75  | 0.7268170426065163 | 0.7268170426065163 | 0.7268170426065163 | 0.72
68170426065163 |
| epochs = 120 | 0.7343358395989975 | 0.7343358395989975 | 0.7343358395989975 | 0.73
43358395989975 |
| epochs = 165 | 0.7418546365914787 | 0.7418546365914787 | 0.7418546365914787 | 0.74
18546365914787 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-05  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.7619047619047619 | 0.7619047619047619 | 0.7619047619047619 | 0.76
19047619047619 |
| epochs = 75  | 0.7644110275689223 | 0.7644110275689223 | 0.7644110275689223 | 0.76
44110275689223 |
| epochs = 120 | 0.7869674185463659 | 0.7869674185463659 | 0.7869674185463659 | 0.78
69674185463659 |
| epochs = 165 | 0.7719298245614035 | 0.7719298245614035 | 0.7719298245614035 | 0.77
19298245614035 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| lr = 0.0001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.7619047619047619 | 0.7619047619047619 | 0.7619047619047619 | 0.76
19047619047619 |
| epochs = 75  | 0.7819548872180451 | 0.7819548872180451 | 0.7819548872180451 | 0.78
19548872180451 |
| epochs = 120 | 0.7744360902255639 | 0.7744360902255639 | 0.7744360902255639 | 0.77
44360902255639 |
| epochs = 165 | 0.7794486215538847 | 0.7794486215538847 | 0.7794486215538847 | 0.77
94486215538847 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+---------------+---------------+----------------+---------------+
|  lr = 0.001  | alpha = 1e-06 | alpha = 1e-05 | alpha = 0.0001 | alpha = 0.001 |
+--------------+---------------+---------------+----------------+---------------+
| epochs = 30  |       0       |       0       |       0        |       0       |
| epochs = 75  |       0       |       0       |       0        |       0       |
| epochs = 120 |       0       |       0       |       0        |       0       |
| epochs = 165 |       0       |       0       |       0        |       0       |
+--------------+---------------+---------------+----------------+---------------+
```

In [46]:

```python
if tune_params == 'Y':
    w_count_bocn, t_bocn, v_bocn = SGD(train_vec_bocn, train_label, X_dev=developmen
                                       lr=lr[lr_index], alpha=alpha[alpha_index], epo
else:
    w_count_bocn, t_bocn, v_bocn = SGD(train_vec_bocn, train_label, X_dev=developmen
```

```
train loss:  [0.6586678865543996, 0.6409576538108084, 0.614961752041814, 0.609961262
2626234, 0.6050659590053403, 0.5941109401285373, 0.5721538260755585, 0.6165177248681
937, 0.566789992670101, 0.551701504992079, 0.5748317519206638, 0.5565582981160806,
0.5505612337997011, 0.5310324752930027, 0.53272522450547, 0.520464313048930 7, 0.5195
529458636964, 0.51419421594282, 0.5115296745251963, 0.5053134249491735, 0.5402928891
318021, 0.49891279946082934, 0.49711488254886027, 0.502233472414653, 0.4900059660727
521, 0.5120546353977561, 0.4915674117822513, 0.48343438281031853, 0.4858985781232244
6, 0.4792423396467732, 0.47528670720434685, 0.47420522320685415, 0.4820408484794968,
```

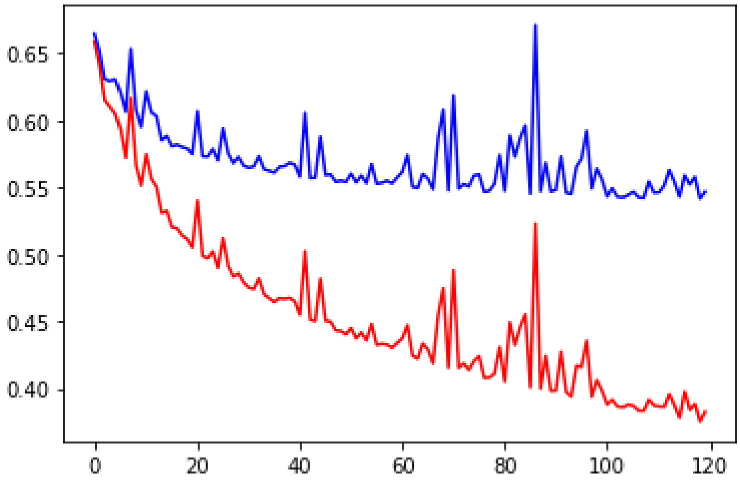
```
0.470250683620504, 0.4672661452776422, 0.46446897579310625, 0.4673847070834981, 0.46
67535314683005, 0.467561051568702, 0.4649346457598132, 0.454931236776774, 0.50244403
97583462, 0.4513051951032547, 0.4502441373159806, 0.48183475374283263, 0.45043273958
14808, 0.44976175360269477, 0.44327001845302216, 0.442759881454071, 0.44032156936500
24, 0.44504379969436436, 0.4373524344368781, 0.441704384847751, 0.4358514294652173,
0.44804104836844005, 0.4325045373689771, 0.4332877341623857, 0.43271257777239186, 0.
43039211909913067, 0.43396754086869743, 0.43729111473110205, 0.4471386963229371, 0.4
245352911182315, 0.42215034445914884, 0.4335283508617555, 0.4289956851403873, 0.4187
1614183971523, 0.45475305935271004, 0.47479137679037814, 0.41566503750319544, 0.4882
471680918078, 0.415616579934175, 0.4186650199065944, 0.4140433205609044, 0.420382340
10861414, 0.4240694965913989, 0.4082672370312182, 0.40842830011226644, 0.41125149209
71855, 0.4308078249068538, 0.4054237346492821, 0.4492079719122969, 0.432344592247088
93, 0.44569085198769653, 0.4552294425822614, 0.4008497611844382, 0.523012167005965,
0.400008128092228, 0.42425204705396574, 0.39842318441221153, 0.39858988199588863, 0.
427336685833453, 0.3975197023070988, 0.39413233343636006, 0.41691435222691625, 0.416
2229173617541, 0.4355669985262978, 0.3939794161562867, 0.4064426592225231, 0.3985738
7675386063, 0.3881398230619971, 0.39175427397123713, 0.38673640214372396, 0.38618665
214238396, 0.38816938541191814, 0.38707579138856324, 0.3836651704398538, 0.383563020
1946036, 0.3917725652755658, 0.3875150550478987, 0.386728040952225, 0.38647414906558
12, 0.39582918463982825, 0.38748144402856965, 0.37843766444870525, 0.397775478316136
2, 0.3842851247062608, 0.38855089407022847, 0.37564006095844893, 0.3827835770353082]
------------------------------------------------
val loss:  [0.6646190952775195, 0.6515164237119583, 0.6307041686231704, 0.6290891399
532217, 0.6304929937438217, 0.6208362201901493, 0.6062444170816965, 0.65323808809514
32, 0.6076443315501273, 0.5952794587036999, 0.6214650604181211, 0.6058508126194757,
0.603216357346822, 0.585261201323867, 0.5887177528803663, 0.5808810542019617, 0.5821
067381452095, 0.5803643727668861, 0.5792202932015043, 0.5750700778728969, 0.60672412
70954076, 0.5734240064814479, 0.5731812014577775, 0.5788889593215684, 0.570279589547
3923, 0.5942650144295636, 0.575586184915815, 0.5682417877849514, 0.5729535307322632,
0.5665330339293609, 0.5648816352525249, 0.565728819352655, 0.5734811611450534, 0.563
7307546993011, 0.5625354452585436, 0.5612171070758918, 0.5654758990706145, 0.5660467
088043534, 0.568534316494792, 0.567129510322393, 0.5581626460412947, 0.6054834471321
324, 0.5571520210293404, 0.5574841105009477, 0.5881551786526648, 0.5593991499554652,
0.55982085780667, 0.5543880305121868, 0.555215543161526, 0.5542724914693433, 0.56047
64955699588, 0.5538891676154909, 0.5590934597558103, 0.5533107832119936, 0.567554894
2315566, 0.5529064705236083, 0.5535728455377483, 0.5551658485372193, 0.5531444665462
881, 0.557626520310379, 0.5616811767825285, 0.5743757399797584, 0.5506456390639086,
0.5497432568158498, 0.56021882603543, 0.5567017969301012, 0.5488547664754305, 0.5869
790558386544, 0.6078032369244792, 0.547958688839032, 0.6185083557120051, 0.549285454
9661711, 0.5527013637788553, 0.5508535978179011, 0.5590693119843488, 0.5597762255537
369, 0.5467737896377398, 0.5476068878707552, 0.5529385174965005, 0.5743653265481432,
0.5475260114039677, 0.5891202929617728, 0.5730648234825999, 0.5869077387766829, 0.59
63772232131564, 0.5452768190984316, 0.6711398438829281, 0.547205796980679, 0.5684815
026735099, 0.5470732844879539, 0.5483382021826207, 0.5732655719518658, 0.54613338894
55267, 0.545293420113468, 0.5655172381851292, 0.5715590750697115, 0.592462876258665,
0.5493970986111696, 0.5641828078244441, 0.5561130960362227, 0.5433518738453489, 0.54
96189274925894, 0.5430905017600167, 0.5426923052617778, 0.5445602279955098, 0.546978
9338852911, 0.5427898498534715, 0.5422870021821604, 0.5545544372310017, 0.5462603520
675879, 0.5464709817664191, 0.5515259389040469, 0.563102630422833, 0.554286075453994
7, 0.5432375430079339, 0.5591488820772587, 0.552380688319582, 0.5579625570453565, 0.
5418069212066783, 0.5469521512025083]
```

In [47]:

```
plt.plot(t_bocn, 'r')
plt.plot(v_bocn, 'b')
```

Out[47]:

```
[<matplotlib.lines.Line2D at 0x1821b585520>]
```



This is a unstable training, which means the learning rate may be too large for this model. However, the final val loss and accuracy are still acceptable.

In [48]:

```
preds_te_count_bocn = predict_class(test_vec_bocn, w_count_bocn)
print('Accuracy:', accuracy_score(test_label,preds_te_count_bocn))
print('Precision:', precision_score(test_label,preds_te_count_bocn))
print('Recall:', recall_score(test_label,preds_te_count_bocn))
print('F1-Score:', f1_score(test_label,preds_te_count_bocn))
```

```
Accuracy: 0.7869674185463659
Precision: 0.7821782178217822
Recall: 0.7939698492462312
F1-Score: 0.7880299251870324
```

In [49]:

```
top_neg = w_count_bocn.argsort()[:10]
for i in top_neg:
    print(id_to_vocab_bocn[i])
```

```
bad
rs
ad
te
ors
ba
p
st
gu
gi
```

In [50]:

```
top_pos = w_count_bocn.argsort()[::-1][:10]
for i in top_pos:
    print(id_to_vocab_bocn[i])
```

```
li
per
rf
erf
od
fu
gr
gre
is
la
```

Here we get 'bad' for the negative reviews. Also, 'per', 'gre', 'erf', seems like 'perfect' and 'great' in positive reviews. Thus, this model is also reasonable.

BOCN-tfidf

In [51]:

```
if tune_params == 'Y':
    acc_history = []
    table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr3 = PrettyTable(["lr = {}".format(lr[3]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    for i in range(len(lr)):
        for j in range(len(alpha)):
            for k in range(len(epochs)):
                w_tfidf_bocn, t_tfidf_bocn, v_tfidf_bocn = SGD(train_tfidf_bocn, tra
                preds_te_count = predict_class(test_tfidf_bocn, w_tfidf_bocn)
                if v_tfidf_bocn[len(v_tfidf_bocn)-1] < 1:
                    acc_history.append(accuracy_score(test_label,preds_te_count))
                else:
                    acc_history.append(0)
        if i == 0:
            table_lr0.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr0.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr0.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr0.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 1:
            table_lr1.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr1.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr1.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr1.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 2:
            table_lr2.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr2.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr2.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr2.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 3:
            table_lr3.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr3.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr3.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr3.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
    result = max(acc_history)
    print('max accuracy:', result)
    index_of_result = acc_history.index(result)
    lr_index = (index_of_result)//16
    alpha_index = (index_of_result%16)//4
    epochs_index = (index_of_result%4)
    print('lr: ', lr[lr_index])
    print('alpha: ', alpha[alpha_index])
    print('epochs: ', epochs[epochs_index])
    print(table_lr0)
    print(table_lr1)
    print(table_lr2)
    print(table_lr3)
```

```
max accuracy: 0.8345864661654135
lr:  0.0001
alpha:  1e-06
epochs:  165
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-06  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
```

```
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.5639097744360902 | 0.5639097744360902 | 0.5639097744360902 | 0.56
39097744360902 |
| epochs = 75  | 0.5839598997493735 | 0.5839598997493735 | 0.5839598997493735 | 0.58
39598997493735 |
| epochs = 120 | 0.6290726817042607 | 0.6290726817042607 | 0.6290726817042607 | 0.62
90726817042607 |
| epochs = 165 | 0.6466165413533834 | 0.6466165413533834 | 0.6466165413533834 | 0.64
66165413533834 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-05  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.6967418546365914 | 0.6967418546365914 | 0.6967418546365914 | 0.69
67418546365914 |
| epochs = 75  | 0.7619047619047619 | 0.7619047619047619 | 0.7619047619047619 | 0.76
19047619047619 |
| epochs = 120 | 0.7794486215538847 | 0.7794486215538847 | 0.7794486215538847 | 0.77
94486215538847 |
| epochs = 165 | 0.7769423558897243 | 0.7769423558897243 | 0.7769423558897243 | 0.77
69423558897243 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| lr = 0.0001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.7969924812030075 | 0.7969924812030075 | 0.7969924812030075 | 0.79
69924812030075 |
| epochs = 75  | 0.8145363408521303 | 0.8145363408521303 | 0.8145363408521303 | 0.81
45363408521303 |
| epochs = 120 | 0.8295739348370927 | 0.8295739348370927 | 0.8295739348370927 | 0.82
95739348370927 |
| epochs = 165 | 0.8345864661654135 | 0.8345864661654135 | 0.8345864661654135 | 0.83
45864661654135 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
+--------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 0.001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.8320802005012531 | 0.8320802005012531 | 0.8320802005012531 | 0.83
20802005012531 |
| epochs = 75  | 0.8195488721804511 | 0.8195488721804511 | 0.8195488721804511 | 0.81
95488721804511 |
| epochs = 120 | 0.8170426065162907 | 0.8170426065162907 | 0.8170426065162907 | 0.81
70426065162907 |
| epochs = 165 | 0.8270676691729323 | 0.8270676691729323 | 0.8270676691729323 | 0.82
70676691729323 |
+--------------+--------------------+--------------------+--------------------+-----
---------------+
```

In [52]:

```
if tune_params == 'Y':
    w_tfidf_bocn, t_tfidf_bocn, v_tfidf_bocn = SGD(train_tfidf_bocn, train_label, X_
                                               Y_dev=development_label,lr=lr[lr_
                                               epochs=epochs[epochs_index], prin
else:
    w_tfidf_bocn, t_tfidf_bocn, v_tfidf_bocn = SGD(train_tfidf_bocn, train_label,
                                               X_dev=development_tfidf_bocn, Y_d
```

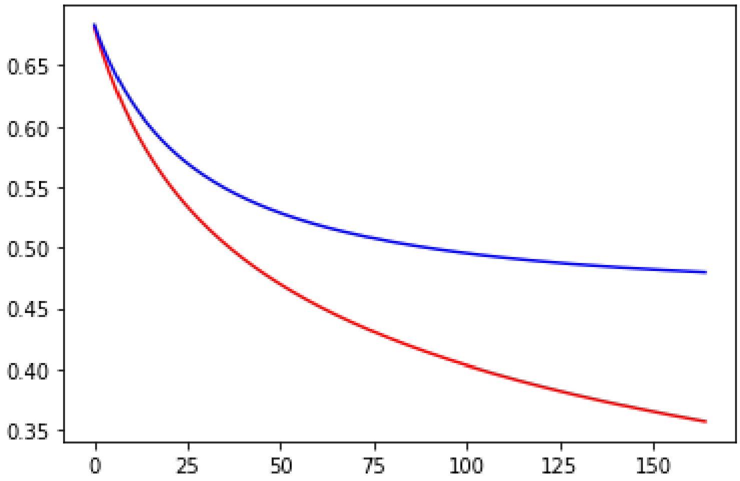
```
train loss:  [0.681458442528966, 0.6711049079825542, 0.6615120432983105, 0.652562730
9449156, 0.644122907984897, 0.6361412955962436, 0.628572718914092, 0.621384323687209
3, 0.6145407854238141, 0.6080267079756276, 0.6018109413717951, 0.5958745143890353,
0.5901966065070277, 0.5847585483407477, 0.5795432190294567, 0.5745344793852059, 0.56
97189752673516, 0.565080162874092, 0.5606187557619792, 0.5563127654729154, 0.5521559
545579795, 0.5481459773742027, 0.5442693642569337, 0.5405060459593192, 0.53686688272
5984, 0.5333422719183369, 0.529922478049714, 0.5266035268968914, 0.5233796612711548,
0.520242842405782, 0.5171968429622555, 0.5142292337685429, 0.511340007973965, 0.5085
263546740816, 0.5057828769010678, 0.5031063165920482, 0.5004938093366652, 0.49794630
533921563, 0.49545441537364154, 0.4930266495941144, 0.4906438404704505, 0.488318839
0709106, 0.48604404691858927, 0.4838176166001264, 0.4816382148925806, 0.479503630325
5345, 0.4774131552726171, 0.47536227280423987, 0.47335248372816224, 0.47138231789146
15, 0.46944905402040366, 0.46755205507576697, 0.4656904636909933, 0.4638609312480682
6, 0.46206556787016134, 0.4603020821597286, 0.4585683813426348, 0.45686612889525485,
0.45519037012553737, 0.4535436447033751, 0.4519233514766584, 0.45032983360154566, 0.
4487620742249391, 0.4472185157354583, 0.44569938695979, 0.4442039595563238, 0.442731
3739348888, 0.4412817323570726, 0.4398547969818796, 0.4384443659759084, 0.4370578932
4511063, 0.43568969054389395, 0.4343406451558378, 0.4330113544204128, 0.431701227178
0796, 0.43040746331129986, 0.42913215384112785, 0.42787347806891274, 0.4266318470667
8897, 0.42540665301000197, 0.4241974885355095, 0.4230034818452375, 0.421824999538063
1, 0.4206611680598982, 0.41951192346831434, 0.4183774283275817, 0.41725548066532736,
0.41614705631896093, 0.41505260515194037, 0.41397107167391967, 0.41290220911176456,
0.41184532914874816, 0.410800563860188, 0.40976822991741657, 0.4087474451731889, 0.4
07737787990035, 0.40673954357197667, 0.40575214917273994, 0.4047762866235605, 0.403
8098039326749, 0.40285409552400153, 0.40190851308442477, 0.40097290654278595, 0.4000
470420812517, 0.3991308510580404, 0.39822424970235565, 0.39732645917357484, 0.396438
13771853165, 0.39555824062776607, 0.39468711835766285, 0.39382493714226907, 0.392970
9331585915, 0.39212509736093937, 0.3912877235890065, 0.3904577642661508, 0.389635989
28256304, 0.3888218881635632, 0.3880153516317313, 0.38721634663881627, 0.38642456052
11805, 0.3856400489538874, 0.3848627782031098, 0.3840921972391298, 0.38332839915448,
0.38257170556022435, 0.3818213429678281, 0.3810776531376982, 0.3803402298463152, 0.3
796092159310637, 0.37888414007629867, 0.37816550047926056, 0.37745303985274864, 0.37
674628154433365, 0.37604543659643147, 0.3753503001473401, 0.3746610223936662, 0.3739
7718044086264, 0.3732988710326554, 0.37262634468975486, 0.37195849276196535, 0.37129
645505368986, 0.37063909331930034, 0.3699871900393843, 0.36934039307421657, 0.368698
5165117873, 0.36806174375174505, 0.36742968004665083, 0.36680237768847374, 0.3661798
6723157886, 0.3655619026241726, 0.36494855642306234, 0.3643396608462779, 0.363735070
88954443, 0.3631349933908765, 0.36253943638312935, 0.3619482079772649, 0.36136089986
97336, 0.36077790821619365, 0.3601990116875522, 0.35962466052003095, 0.3590534300265
95, 0.358486647475162, 0.35792376429268796, 0.35736450114704477, 0.3568093340000903]
------------------------------------------------
val loss:  [0.6834083006124824, 0.6749253367257749, 0.6671082381026187, 0.6598886073
269211, 0.6531256342898885, 0.6467613922060683, 0.6407537677160425, 0.63507965131950
82, 0.6297252824367883, 0.6246415755543845, 0.6198212468768168, 0.6152519484926106,
0.6109068374732151, 0.606769163506183, 0.6028226455488358, 0.5990621161233635, 0.595
4725839086251, 0.5920423302567313, 0.5887641594929917, 0.5856235145600309, 0.5826144
367095336, 0.5797337670445599, 0.5769702187859095, 0.5743058831012909, 0.57174684754
93378, 0.5692886489693283, 0.5669232948019062, 0.564648146696283, 0.562451907092570
8, 0.5603219185436017, 0.5582736936873525, 0.5562944039229171, 0.5543912888424071,
0.5525356710585876, 0.5507455977608238, 0.5490151090362343, 0.547338130092621, 0.545
7111633585945, 0.544142263380822, 0.542614726742417, 0.5411345578357055, 0.539700074
0791217, 0.538301690284257, 0.5369402535976249, 0.5356184903231817, 0.53433824796161
04, 0.5331034181981584, 0.5318835355794476, 0.5307049658824505, 0.5295460575735462,
0.5284364591313824, 0.527350841440798, 0.526295110911051, 0.5252347666088922, 0.5242
354301408176, 0.5232557342265556, 0.522277476503812, 0.5213571655878169, 0.520429494
2183693, 0.5195351303395905, 0.5186487580699454, 0.5177851928071755, 0.5169441390231
758, 0.5161348002812944, 0.5153366917407438, 0.5145590679931507, 0.5138069517423969,
0.513070454989694, 0.5123554910549195, 0.5116172100288376, 0.5109313915382705, 0.510
2051164043659, 0.5095520615864637, 0.5089021281744496, 0.5082683440929373, 0.5076285
5533666, 0.5069944024424939, 0.506391761467474, 0.5058064127878441, 0.50521243759285
54, 0.5046400426998101, 0.5040870476421235, 0.5035399356042609, 0.5030072889009319,
0.502483878030043, 0.5019802342616816, 0.501464434047776, 0.5009501489959505, 0.5004
51635728508, 0.49996624629507763, 0.49952470017075196, 0.4990386457004219, 0.4985890
5457417246, 0.49814335382703256, 0.49769025013979756, 0.49727344042757937, 0.4968594
0340899454, 0.49643435307668515, 0.49604597318790855, 0.4956272518484385, 0.49523105
52343427, 0.4948348898347804, 0.49446689544380934, 0.4940835542476692, 0.49371231465
619675, 0.493341863478258, 0.4929927532846747, 0.49262676856791626, 0.49230393362987
135, 0.4919664380125923, 0.4916362400720357, 0.49130433530895623, 0.490983084491248
1, 0.4906828736985571, 0.4903588146016524, 0.49004258735734163, 0.4897473433229131,
```

```
0.48944600891885376, 0.4891461701309081, 0.4888606870027838, 0.4885927868264016, 0.4
883143722679472, 0.48804309471357504, 0.48776159722220563, 0.48748086004791796, 0.48
722937914607634, 0.48696240503368515, 0.48672476405919074, 0.4864562399222419, 0.486
22200265885607, 0.4859799665972549, 0.48575166410492293, 0.4855107743577757, 0.48526
95736559774, 0.4850457457621577, 0.4848134166152787, 0.48460901571695836, 0.48438587
463237454, 0.484188533265841, 0.48396169811445766, 0.4837728067992811, 0.48354205032
4524, 0.48333955152478425, 0.4831417835478862, 0.48294110225371767, 0.48274009898553
005, 0.4825548166746792, 0.48237473535466246, 0.4821893645886932, 0.4820165744734120
6, 0.48184143454580197, 0.48166230302597834, 0.4814773883390656, 0.4813199553035722,
0.4811599299338769, 0.4809929637288825, 0.48082184126599675, 0.48065125476693527, 0.
48048696214816866, 0.4803101774199123, 0.4801739858066424, 0.4800121933321794, 0.479
8638975642629, 0.4797356991470545, 0.47957865383553094]
```

In [53]:

```python
plt.plot(t_tfidf_bocn, 'r')
plt.plot(v_tfidf_bocn, 'b')
```

Out[53]: [<matplotlib.lines.Line2D at 0x1821b5d4040>]



The final val loss is about 0.5, which is acceptable. The curve of val loss is descending without a trend of ascending. This means the model is neither overfitting nor underfitting.

In [54]:

```python
preds_te_tfidf_bocn = predict_class(test_tfidf_bocn, w_tfidf_bocn)
print('Accuracy:', accuracy_score(test_label,preds_te_tfidf_bocn))
print('Precision:', precision_score(test_label,preds_te_tfidf_bocn))
print('Recall:', recall_score(test_label,preds_te_tfidf_bocn))
print('F1-Score:', f1_score(test_label,preds_te_tfidf_bocn))
```

```
Accuracy: 0.8345864661654135
Precision: 0.8181818181818182
Recall: 0.8592964824120602
F1-Score: 0.838235294117647
```

In [55]:

```python
top_neg = w_tfidf_bocn.argsort()[:10]
for i in top_neg:
    print(id_to_vocab_bocn[i])
```

```
bad
bor
ipt
ulo
wfu
ors
awf
mpt
wf
upi
```

In [56]:
```
top_pos = w_tfidf_bocn.argsort()[::-1][:10]
for i in top_pos:
    print(id_to_vocab_bocn[i])
```

```
rfe
ila
erf
gre
rf
rue
rld
osc
rfu
fic
```

Here we get 'bad', 'awf', 'wuf', 'wf' for 'bad' and 'awful' in negative reviews. Similarly, we get 'rfe', 'erf', and 'rf' for 'perfect' in positive reviews. Thus, this model is also reasonable.

BOW+BOCN

In [57]:
```
train_bow_bocn_tfidf = np.concatenate((train_tfidf,train_tfidf_bocn), axis=1)
development_bow_bocn_tfidf = np.concatenate((development_tfidf,development_tfidf_boc
test_bow_bocn_tfidf = np.concatenate((test_tfidf,test_tfidf_bocn), axis=1)
```

In [58]:
```
if tune_params == 'Y':
    acc_history = []
    table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    table_lr3 = PrettyTable(["lr = {}".format(lr[3]), "alpha = {}".format(alpha[0]),
                , "alpha = {}".format(alpha[2]), "alpha = {}".format(alpha[3])])
    for i in range(len(lr)):
        for j in range(len(alpha)):
            for k in range(len(epochs)):
                w_bow_bocn_tfidf, t_bow_bocn_tfidf, v_bow_bocn_tfidf = SGD(X_tr=trai
                preds_te_bow_bocn_tfidf = predict_class(test_bow_bocn_tfidf, w_bow_b
                if v_bow_bocn_tfidf[len(v_bow_bocn_tfidf)-1] < 1:
                    acc_history.append(accuracy_score(test_label,preds_te_bow_bocn_t
                else:
                    acc_history.append(0)
        if i == 0:
            table_lr0.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr0.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr0.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr0.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 1:
            table_lr1.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr1.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr1.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr1.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 2:
            table_lr2.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr2.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr2.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
            table_lr2.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
        if i == 3:
            table_lr3.add_row(["epochs = {}".format(epochs[0]), acc_history[i*16+0],
            table_lr3.add_row(["epochs = {}".format(epochs[1]), acc_history[i*16+1],
            table_lr3.add_row(["epochs = {}".format(epochs[2]), acc_history[i*16+2],
```

```
            table_lr3.add_row(["epochs = {}".format(epochs[3]), acc_history[i*16+3],
    result = max(acc_history)
    print('max accuracy:', result)
    index_of_result = acc_history.index(result)
    lr_index = (index_of_result)//16
    alpha_index = (index_of_result%16)//4
    epochs_index = (index_of_result%4)
    print('lr: ', lr[lr_index])
    print('alpha: ', alpha[alpha_index])
    print('epochs: ', epochs[epochs_index])
    print(table_lr0)
    print(table_lr1)
    print(table_lr2)
    print(table_lr3)
```

```
max accuracy: 0.8571428571428571
lr:  0.001
alpha:  1e-06
epochs:  30
+---------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-06   |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+---------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30   | 0.6165413533834586 | 0.6165413533834586 | 0.6165413533834586 | 0.61
65413533834586 |
| epochs = 75   | 0.6466165413533834 | 0.6466165413533834 | 0.6466165413533834 | 0.64
66165413533834 |
| epochs = 120  | 0.6917293233082706 | 0.6917293233082706 | 0.6917293233082706 | 0.69
17293233082706 |
| epochs = 165  | 0.7192982456140351 | 0.7192982456140351 | 0.7192982456140351 | 0.71
92982456140351 |
+---------------+--------------------+--------------------+--------------------+-----
---------------+
+---------------+--------------------+--------------------+--------------------+-----
---------------+
|  lr = 1e-05   |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+---------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30   | 0.7543859649122807 | 0.7543859649122807 | 0.7543859649122807 | 0.75
43859649122807 |
| epochs = 75   | 0.7944862155388471 | 0.7944862155388471 | 0.7944862155388471 | 0.79
44862155388471 |
| epochs = 120  | 0.8020050125313283 | 0.8020050125313283 | 0.8020050125313283 | 0.80
20050125313283 |
| epochs = 165  | 0.8120300751879699 | 0.8120300751879699 | 0.8120300751879699 | 0.81
20300751879699 |
+---------------+--------------------+--------------------+--------------------+-----
---------------+
+---------------+--------------------+--------------------+--------------------+-----
---------------+
| lr = 0.0001   |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001    |
+---------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30   | 0.8370927318295739 | 0.8370927318295739 | 0.8370927318295739 | 0.83
70927318295739 |
| epochs = 75   | 0.8421052631578947 | 0.8421052631578947 | 0.8421052631578947 | 0.84
21052631578947 |
| epochs = 120  | 0.849624060150376  | 0.849624060150376  | 0.849624060150376  | 0.84
9624060150376  |
| epochs = 165  | 0.849624060150376  | 0.849624060150376  | 0.849624060150376  | 0.84
9624060150376  |
+---------------+--------------------+--------------------+--------------------+-----
---------------+
+---------------+--------------------+--------------------+--------------------+-----
```

```
---------------+
| lr = 0.001  |   alpha = 1e-06    |   alpha = 1e-05    |   alpha = 0.0001   |   al
pha = 0.001  |
+-------------+--------------------+--------------------+--------------------+-----
---------------+
| epochs = 30  | 0.8571428571428571 | 0.8571428571428571 | 0.8571428571428571 | 0.85
71428571428571 |
| epochs = 75  | 0.8471177944862155 | 0.8471177944862155 | 0.8471177944862155 | 0.84
71177944862155 |
| epochs = 120 | 0.8421052631578947 | 0.8421052631578947 | 0.8421052631578947 | 0.84
21052631578947 |
| epochs = 165 | 0.8345864661654135 | 0.8345864661654135 | 0.8345864661654135 | 0.83
45864661654135 |
+-------------+--------------------+--------------------+--------------------+-----
---------------+
```

In [59]:

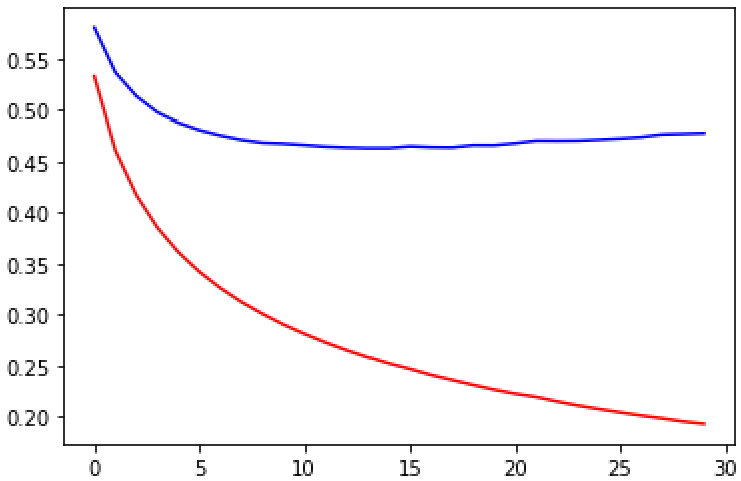
```
if tune_params == 'Y':
    w_bow_bocn_tfidf, t_bow_bocn_tfidf, v_bow_bocn_tfidf = SGD(train_bow_bocn_tfidf,
                                                               Y_dev=development_lab
                                                               epochs=epochs[epochs_
else:
    w_bow_bocn_tfidf, t_bow_bocn_tfidf, v_bow_bocn_tfidf = SGD(train_bow_bocn_tfidf,
                                                               Y_dev=development_lab
```

```
train loss:  [0.5326226599730167, 0.4608078445091674, 0.4171088783073348, 0.38533661
616981485, 0.3612415661834907, 0.34203780290868385, 0.326136251475283, 0.3125426275
771912, 0.30092211849012984, 0.29043550946735486, 0.281296956930362, 0.2728883377198
892, 0.2653410593908214, 0.258502002935498, 0.25214064668513586, 0.2468048557242297
8, 0.2408931465385257, 0.23602232448544883, 0.23117783271573916, 0.2265185215652098
5, 0.22259348905669524, 0.2193859255885909, 0.2149572798015082, 0.21098378943928875,
0.20755885359677753, 0.20434022663679863, 0.20126684908663325, 0.198481190758188, 0.
19540744074028224, 0.1931055893962769]
------------------------------------------------
val loss:  [0.5802836441951238, 0.5364640899039577, 0.5133538564140353, 0.4975822807
445059, 0.48759905759390165, 0.4804719340733244, 0.47529928541722805, 0.471050661648
9357, 0.46816921363955033, 0.4673238564444099, 0.4660015386253393, 0.464436722687755
5, 0.46354491325260716, 0.46301853090826534, 0.463049482937284, 0.4648087464532036,
0.4639172046919725, 0.4636600988866982, 0.4658726365463701, 0.46579918730765457, 0.4
676984482008383, 0.4702113085165863, 0.47007049044599875, 0.47022673722423824, 0.471
1932045635471, 0.47232553415385947, 0.4737444072439565, 0.4763071403940996, 0.476860
2909254012, 0.4774864400802157]
```

In [60]:

```
plt.plot(t_bow_bocn_tfidf, 'r')
plt.plot(v_bow_bocn_tfidf, 'b')
```

Out[60]: [<matplotlib.lines.Line2D at 0x1827a7a7460>]



The final val loss is about 0.5, which is acceptable. The curve of val loss is descending without a trend of ascending. This means the model is neither overfitting nor underfitting.

In [61]:
```
preds_te_bow_bocn_tfidf = predict_class(test_bow_bocn_tfidf, w_bow_bocn_tfidf)
print('Accuracy:', accuracy_score(test_label,preds_te_bow_bocn_tfidf))
print('Precision:', precision_score(test_label,preds_te_bow_bocn_tfidf))
print('Recall:', recall_score(test_label,preds_te_bow_bocn_tfidf))
print('F1-Score:', f1_score(test_label,preds_te_bow_bocn_tfidf))
```

```
Accuracy: 0.8571428571428571
Precision: 0.855
Recall: 0.8592964824120602
F1-Score: 0.8571428571428571
```

In [62]:
```
id_to_vocab_bow_bocn = {**id_to_vocab_bocn, **id_to_vocab}
```

In [63]:
```
top_neg = w_bow_bocn_tfidf.argsort()[:10]
for i in top_neg:
    print(id_to_vocab_bow_bocn[i])
```

```
bad
cam
waste
worst
boring
supposed
fails
flat
worse
ridiculous
```

In [64]:
```
top_pos = w_bow_bocn_tfidf.argsort()[::-1][:10]
for i in top_pos:
    print(id_to_vocab_bow_bocn[i])
```

```
hilarious
great
excellent
simple
terrific
perfect
fiction
nfe
memorable
perfectly
```

Here we get 'fails', 'worse', etc. for negative reviews and 'great', 'terrific', etc. for positive reviews. Thus, this model performs well.

Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count	0.8177339901477833	0.8341708542713567	0.8258706467661691
BOW-tfidf	0.8333333333333334	0.8542713567839196	0.8436724565756824
BOCN-count	0.7821782178217822	0.7939698492462312	0.7880299251870324
BOCN-tfidf	0.8181818181818182	0.8592964824120602	0.838235294117647
BOW+BOCN	0.855	0.8592964824120602	0.8571428571428571

Please discuss why your best performing model is better than the rest.

ANS: My best performing model is BOW+BOCN. This is because this model has more features than the other four models. Also, this model does not include many noises as features. Thus, this model performs best.