[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/ (http://www.cs.cornell.edu/people/pabo/movie-review-data/)](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)](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 (https://www.sheffield.ac.uk/ssid/unfair-means/index)](https://www.sheffield.ac.uk/ssid/unfair-means/index), including Turnitin which helps detect plagiarism. Use of unfair means would

In [1]:

```
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
import random
import string

# 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 [2]:

```
# read csv file with the test sets
test = pd.read_csv("data_sentiment/test.csv")


# displaying the list of column names
#Column 1 = TEXT
#Column 2 = LABELS

# creating a list of column names by
# calling the columns
```

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

In [3]:

```
# read csv file with the train sets
train = pd.read_csv("data_sentiment/train.csv")


# displaying the list of column names
#Column 0 = TEXT
#Column 1 = LABELS


# creating a list of column names by
# calling the .columns
train_column_names = list(train.columns)


##########################################################


# read csv file with the development sets
dev  = pd.read_csv("data_sentiment/dev.csv")

```

```python

# displaying the list of column names
#Column 0 = TEXT
#Column 1 = LABELS


# creating a list of column names by
# calling the .columns
```

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

In [4]:

```python
#put the trainning raw texts into Python lists
train_text = list(train[train_column_names[0]])

#print the text for verification
#print(train_text,"\n")

#put the trainning labels into a NumPy arrays
train_label = train[train_column_names[1]].values

#print the train label for verification
print(train_label,"\n")


#############################################


#put the testing raw texts into Python lists
test_text = list(test[test_column_names[0]])

#print the text for verification
#print(test_text,"\n")

#put the testing labels into a NumPy arrays
test_label = test[test_column_names[1]].values

#print the test label for verification
print(test_label,"\n")


##############################################


#put the development raw texts into Python lists
dev_text = list(dev[dev_column_names[0]])

#print the text for verification
#print(dev_text,"\n")

#put the development labels into a NumPy arrays
dev_label = dev[dev_column_names[1]].values

#print the dev label for verification
```

```
[1 1 1 ... 0 0 0]

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
```

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 [5]:

```
#make a list for unigrams from each
def generate_ngrams(s, n):

    # Convert to lowercases
    #s = s.lower()

```

```python
    # Replace all none alphanumeric characters with spaces
    s = re.sub(r'[^a-zA-Z0-9\s]', ' ', str(s))

    # Break sentence in the token, remove empty tokens
    tokens = [token for token in s.split(" ") if token != ""]

    # Use the zip function to help generate n-grams
    # Concatentate the tokens into ngrams and return
    ngrams = zip(*[tokens[i:] for i in range(n)])
    return [" ".join(ngram) for ngram in ngrams]

#Generate unigrams, using the trainning set
train_unigram = generate_ngrams(train_text, n=1)

#Generate unigrams, using the testing set
test_unigram = generate_ngrams(test_text, n=1)

#Generate unigrams, using the development set
dev_unigrams = generate_ngrams(dev_text, n=1)




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'
             'it', 'he', 'she', 'we', 'they', 'will', 'have', 'has
              'do', 'did', 'can', 'could', 'who', 'which', 'what',
             'his', 'her', 'they', 'them', 'from', 'with', 'its']


#Remove these stop words from the list of ngrams
def remove_stop_word(ngram):

    #use list comprehension,
    #to only return words not inlcuded in stop_words
    return [word for word in ngram if word not in stop_words]

#Remove stopwords on the train/test/dev unigrams
uni_train_no_sw = remove_stop_word(train_unigram)
uni_test_no_sw = remove_stop_word(test_unigram)
uni_dev_no_sw = remove_stop_word(dev_unigrams)

#Make Bigrams from the created unigrams
bi_train = generate_ngrams(uni_train_no_sw, n=2)
bi_test = generate_ngrams(uni_test_no_sw, n=2)
bi_dev = generate_ngrams(uni_dev_no_sw, n=2)

#Make Trigrams from the created unigrams
tri_train = generate_ngrams(uni_train_no_sw, n=3)
tri_test = generate_ngrams(uni_test_no_sw, n=3)
tri_dev = generate_ngrams(uni_dev_no_sw, n=3)


#####################################
#Remove ngrams appearing in less than K documents
def doc_counter(set_train, set_test, dev_test):

    #use Counter type in order to count all unique words,
```

```
    #from each train/test/dev set
    c = Counter()
    c.update(set_train)
    c.update(set_test)
    c.update(dev_test)

    return c

#Initialise the set versions of the train/test/dev unigrams
set_uni_train = set(uni_train_no_sw)
set_uni_test = set(uni_test_no_sw)
set_uni_dev = set(uni_dev_no_sw)

#Call doc_counter for all the unigram sets
uni_doc_appearances = doc_counter(set_uni_train, set_uni_test, set_

#Initialise the set versions of the train/test/dev bigrams
set_bi_train = set(bi_train)
set_bi_test = set(bi_test)
set_bi_dev = set(bi_dev)

#Call doc_counter for all the bigram sets
bi_doc_appearances = doc_counter(set_bi_train, set_bi_test, set_bi_


set_tri_train = set(tri_train)
set_tri_test = set(tri_test)
set_tri_dev = set(tri_dev)

#Initialise the set versions of the train/test/dev trigrams
tri_doc_appearances = doc_counter(set_tri_train, set_tri_test, set_


#Number of documents(set between 1 and 3)
def find_words(c, k):

    #Output list variable
    found_words =[]

    #c is a Counter,
    # go through every word contained by c
    for words in c.keys():

    #if documents appearance value is smaller than k
    #in that case continue
        if c[words] < k :
            continue
        else:
            #Add this ngram to the list
            found_words.append(words)

    #No need to keep it as a list,
    # arrays will help with efficiency
    return np.array(found_words)


#Function that will return lists,
#containning only ngrams  appearing at least in K documents
def remove_k(set_train, set_test, set_dev ,doc_ap):

```

```
    clean_train = []
    clean_test = []
    clean_dev = []

    print("Finding words....", "\n")

    #Get all ngrams that appear in at least k documents
    found_words = find_words(doc_ap, k=3)

    print(found_words)

    print("Starting trainning....", "\n")
    clean_train = [word for word in set_train if word in found_wor

    print("Starting testing....", "\n")
    clean_test = [word for word in set_test if word in found_words

    print("Starting dev....", "\n")
    clead_dev = [word for word in set_dev if word in found_words]

    return np.array(clean_train), np.array(clean_test), np.array(c

```

This code cell is separated from the rest due to the heavy computation needed.

Only run this cell once!

Estimated processing time with the full lists of ngrams: 12 minutes.

5 Minutes with the sets version

In [6]:

```



print("Starting UNIGRAMS....", "\n")


clean_uni_train, clean_uni_test, clean_uni_dev  = remove_k(set_uni_
                                                            set_u
                                                            set_u
                                                          uni_doc



print("Starting BIGRAMS....", "\n")


clean_bi_train, clean_bi_test, clean_bi_dev = remove_k(set_bi_trair
                                                       set_bi_test
                                                       set_bi_dev,
                                                         bi_doc_ap


```

```python
print("Starting TRIGRAMS....", "\n")


clean_tri_train, clean_tri_test, clean_tri_dev = remove_k(set_tri_t
                                                          set_tri_te
                                                          set_tri_de
                                                            tri_doc_
```

```
Starting UNIGRAMS....

Finding words....

['emotive' 'macy' 'sketched' ... 'ms' 'virtues' 'aiming']
Starting trainning....

Starting BIGRAMS....

Finding words....

['no spark' 'take back' 'one films' ... 'every movie' 'friend amazin
g'
 'known actors']
Starting trainning....

Starting TRIGRAMS....

Finding words....

['still doesn t' 'movie going experience' 'few far between' ...
 'pretty good but' 'scale 0 4' 'best thing about']
Starting trainning....
```

In [7]:

```python
#Create a vocabulary of unigrams, bigrams and trigrams
vocab = set(clean_uni_train)
vocab.update(clean_uni_test)
vocab.update(clean_uni_dev)

vocab.update(clean_bi_train)
vocab.update(clean_bi_test)
vocab.update(clean_bi_dev)

vocab.update(clean_tri_train)
vocab.update(clean_tri_test)
vocab.update(clean_tri_dev)



print(vocab, "\n")

```

```
{'emotive', 'macy', 'sketched', 'shrewd', 'director lasse', 'deep ri
sing', 'very believable', 'afraid', 'there more', 'goof', 'parallel
', 'civilization', 'physical appearance', 'ricci', 's possible', 'de
picting', 'covert', 'so ll', 'when', 'no sense all', 'no spark', 'po
lice station', 'store', 'years s', 'time just', 'gentleman', 's fell
ow', 'if s', 'taste', 'rd', 'gumption', 'bitchy', 'but comes', 'even
some', 'but overall', 'right hand', 'dr evil s', 'anytime', 'sexual
relationship', 'knocks', 'focuses', 'd probably', 'universal studios
', 'take back', 'doesn t get', 'founded', 'their friends', 'cloud',
```

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]:
```python
def extract_ngrams(x_raw, ngram_range, token_pattern,
                   stop_words, vocab, char_ngrams):

    #Set the smallest value of ngram types
    min_ = ngram_range[0]

    #Set the biggest value of ngram types
    max_ = ngram_range[-1]

    #Initialise output values
    output_ngram = []
    output_char_gram =[]

    #Produce Character ngrams or regular ngrams
    if char_ngrams == False:

        #Go through every type of ngram(i.e. unigram, bigram)
        for rn in range(min_,max_+1):
            print(rn)

            # Replace all none alphanumeric characters with spaces
            x_sub = re.sub(r'[^a-zA-Z0-9\s]', ' ', str(x_raw))

            x_sub.replace("'", " ")

            # Break sentence in the token, remove empty tokens
            tokens = [token for token in x_sub.split(token_pattern)
```

```python

            # Use the zip function to help generate n-grams
            # Concatentate the tokens into ngrams and return
            ngrams = zip(*[tokens[i:] for i in range(rn)])
            final_ngrams = [" ".join(ngram) for ngram in ngrams]


            #Remove stop words from ngrams
            no_stop_ngram = [word for word in final_ngrams if word
            #if rn == 3:
                #print('This is the stop_words',*no_stop_ngram, sep

            #filter ngrams in vocabulary
            for word_o in no_stop_ngram:
                if word_o in vocab:
                    output_ngram.append(word_o)

        print(output_ngram)
        return output_ngram

    else:

    #Generate character ngrams

    #Go through every type of ngram(i.e. unigram, bigram)
        for rn in range(min,max+1):

            final_char =[]
            #b[i:i+n] for i in range(len(b)-n+1)

            # Replace all none alphanumeric characters with spaces
            x_sub = re.sub(r"[^a-zA-Z0-9\s]", "", str(x_raw))

            x_sub.replace("'", "")
            x_sub.replace(" ","")
            # tokens = [token for token in x_sub.split(" ") if toke


            # Use the zip function to help generate character n-gra
            # Concatentate the tokens into ngrams and return
            char_grams = zip(*[x_sub[i:] for i in range(rn)])

             #Split words by character, not by whitespace
            final_char = ["".join(char_gram) for char_gram in char_


            #Remove stopwords
            output_char_gram = [word for word in final_char if word

        print(output_char_gram)
        return output_char_gram
```

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 [163]:

```python
##### Will Keep this cell commneted, for preview purposes ######
```

```
##### To check the real running code go to cell above #####

# def extract_ngrams(x_raw="movie",
#                    ngram_range=(2,4),
#                    stop_words=[],
#                    char_ngrams=True):

#      min = ngram_range[0]

#      max = ngram_range[-1]

#      output_char_gram, no_stop_char =[]


#      for rn in range(min,max+1):

#              #b[i:i+n] for i in range(len(b)-n+1)

#              # Replace all none alphanumeric characters with space
#              x_sub = re.sub(r'[^a-zA-Z0-9\s]', ' ', str(x_raw))

#              # Break sentence in the token, remove empty tokens
#             # tokens = [token for token in x_sub.split(token_patte

#              # Use the zip function to help generate character n-g
#              # Concatentate the tokens into ngrams and return
#              char_grams = zip(*[x_sub[i:] for i in range(rn)])
#              final_char = ["".join(char_gram) for char_gram in cha


#              #Remove stopwords
#              no_stop_char = [word for word in final_char if word r


#              #search in vocab
#              output_char_gram = [word_o for word_o in no_stop_char




#      return output_char_gram
```

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 [162]:

```
def get_vocab(X_raw, ngram_range, token_pattern,
              min_df, keep_topN, stop_words, char_ngrams):

    #Set the smallest value of ngram types
    min_ = ngram_range[0]

    #Set the biggest value of ngram types
    max_ = ngram_range[-1]

    ngrams = np.array([0])

    #Go through every type of ngram(i.e. unigram, bigram)
    for rn in range(min_, max_+1):
    #       n = ngram-range
        print("Filter rn.... ", rn ,"\n")


        special_char=[",",":"," ",";",".","?","'"]

        # Replace all none alphanumeric characters with spaces
        s = re.sub(r'[^a-zA-Z0-9\s]', ' ', str(X_raw))

        # Break sentence in the token, remove empty tokens
        tokens = [token for token in s.split(" ") if token != ""]

        # Use the zip function to help generate n-grams
        # Concatentate the tokens into ngrams and return
        n_grams = zip(*[tokens[i:] for i in range(rn)])
        ngrams = np.append(ngrams, [" ".join(ngram) for ngram in n


    print("Filter vocab....","\n")

    #Remove stop words and special charcters from the list of ngra
    filtered_vocab = [w for w in ngrams if w not in stop_words and



    print("Start extract_ngram....","\n")

    #Initialise and pass the filtered vocab as a set
    original_vocab = set(filtered_vocab)

    #Extract ngrams from the vocabulary
    ngram = extract_ngrams(X_raw, ngram_range, token_pattern, stop

    #print(ngram)

    #Count all the ngrams
    df_count = Counter()
    df_count.update(ngram)
```

```python



def Compute_DF(ngrams):

    DF = {}

    print("Started DFs small.. \n")
    for i in range(len(ngrams)):

        for w in ngrams[i]:
            try:
                DF[w].add(i)
            except:
                DF[w] = {i}


    for i in DF:
        DF[i] = len(DF[i])
    print(DF)
    return DF


def find_doc_freq(word,DF):

    #Method to get a specific ngram's Document frequency

    c = 0

    try:
        c = DF[word]
    except:
        pass
    return c



DF = Compute_DF(ngram)

vocab_init = set(ngram)

#     found_gram = np.array([0])
#Filter ngrams through vocabulary PROBLEM
#     found_gram = [w for w in ngram if w in vocab]


N = len(ngram)
#     print(N)

#Initialise a new Counter to pass into ngrams with a count hig
df_final = Counter()


df ={}


for i in range(N):

    tokens = found_gram[i]
    counter = Counter(tokens)#Replace with count vector
```

```
            words_count = len(tokens)

            #df_final.update(np.unique(tokens))

            for token in np.unique(tokens):
                tf = counter[token]/words_count
                df_word = find_doc_freq(token,DF)
                if df_word >= min_df:
                    df.update({token: df_word}) #was df
                    df_final.update(token)

        vocab = set()
        ngram_counts = []

        #Go through the the top n most common ngrams,
        # and extract their raw frequency and word

        for word, count in df_final.most_common(keep_topN):
            vocab.add(word)
            ngram_counts.append(count)#Count is raw frequency


        print(vocab)
        print(df)
        print(ngram_counts)
#         print(type(top_ngrams))

#         count = Counter()
#         count.update(top_ngrams)


#         ngram_counts = count.values()
#         print(types(ngram_counts))

```

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

In [164]:

```
test_vocab, test_df, test_count = get_vocab(test_text, ngram_range=
                            min_df=2, keep_topN=500,
                            stop_words = stop_words,char_ngrams = False)



# print('TEST VOCAB: ', test_vocab, '\n')
```

```
Filter rn....  1

Filter rn....  2

Filter rn....  3

Filter vocab....

Start extract_ngram....

1
2
3
```

```
IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.
```

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

In [24]:

```python
def create_2dict(df_dict):
    id2word = {}
    word2id = {}
    dic_id = 0

    for word in test_df.keys():

            #(1) vocabulary id -> word
            id2word.update({dic_id : word})

            # (2) word -> vocabulary id
            word2id.update({word: dic_id})

            dic_id += 1

    print('Dictionary [ID : WORD] : ',id2word, "\n")
    print('Dictionary [WORD : ID] : ',word2id, "\n")

    return id2word , word2id
```

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

In [165]:

```python
#TEST
# test_vocab, test_df, test_count = get_vocab(test_text, ngram_rang
#                           min_df=2, keep_topN=500,
#                           stop_words = stop_words)

#train
train_vocab, train_df, train_count = get_vocab(train_text, ngram_ra
                       min_df=10, keep_topN=100,
                       stop_words=stop_words,char_ngrams=False)
#Dev
dev_vocab, dev_df, dev_count = get_vocab(dev_text, ngram_range=(1,
                       min_df=10, keep_topN=100,
                       stop_words=stop_words,char_ngrams=False)

```

```
Filter rn....  1

IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.
```

In [30]:

```python
print("Test Dictionary >>> ", "\n")
id_w_test, w_id_test = create_2dict(test_df)

print("Train Dictionary >>> ", "\n")
id_w_train, w_id_train =create_2dict(train_df)

print("DEV Dictionary >>> ", "\n")
```

```
Test Dictionary >>>

Dictionary [ID : WORD] :  {0: 'know', 1: 'but', 2: 'got', 3: 'around
', 4: 'last', 5: 'one', 6: 'about', 7: 'final', 8: 'scene', 9: 'out
', 10: 'enough', 11: 'watch', 12: 'such', 13: 'good', 14: 'behind',
15: 'show', 16: 'most', 17: 'well', 18: 'gets', 19: 'school', 20: 's
', 21: 'plays', 22: 'him', 23: 'plot', 24: 'help', 25: 'very', 26: '
finds', 27: 'himself', 28: 'love', 29: 'fun', 30: 'begins', 31: 'go
', 32: 'while', 33: 'goes', 34: 'like', 35: 'too', 36: 'young', 37:
't', 38: 'way', 39: 'two', 40: 'people', 41: 'really', 42: 'up', 43:
'year', 44: 'completely', 45: 'if', 46: 'me', 47: 'maybe', 48: 'best
', 49: 'picture', 50: 'instead', 51: 'film', 52: 'reason', 53: 'so',
54: 'point', 55: 'next', 56: 'turn', 57: 'gives', 58: 'movie', 59: '
how', 60: 'having', 61: 'bit', 62: 'bad', 63: 'not', 64: 'being', 6
5: 'big', 66: 'woman', 67: 'performance', 68: 'had', 69: 'just', 70:
'because', 71: 'both', 72: 'into', 73: 'great', 74: 'character', 75:
'long', 76: 'when', 77: 'course', 78: 'actually', 79: 'even', 80: 'm
an', 81: 'every', 82: 'something', 83: 'life', 84: 'first', 85: 'fri
end', 86: 'where', 87: 'couple', 88: 'getting', 89: 'see', 90: 'anyt
```

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 [152]:

```python
def vectorise(X_ngram, vocab):


    def count_vectorize(tokens):
```

```
        ''' This function takes list of words in a sentence as inp
        and returns a vector of size of filtered_vocab.It puts 0 i
        word is not present in tokens and count of token if presen

        vector = np.array([0])
        for w in np.array(filtered_vocab):
            vector = np.append(vector, tokens.count(w))


        return vector

    def Compute_DF(ngrams):

        DF = {}

        print("Started DFs small.. \n")
        for i in range(len(ngrams)):

            for w in ngrams[i]:
                try:
                    DF[w].add(i)
                except:
                    DF[w] = {i}


        for i in DF:
            DF[i] = len(DF[i])
        print(DF)
        return DF


    def find_doc_freq(word,DF):

        #Method to get a specific ngram's Document frequency

        c = 0

        try:
            c = DF[word]
        except:
            pass
        return c



    def compute_tf_IDF(ngram):

        #Calculate the Document Frequency

        DF = Compute_DF(ngram)



        doc = 0
        token_counter = 0

        print("Started TF.IDF...\n")

        #Calculate TF.IDF

```

```

         # N=Total number of documents in the dataset


        found_gram = np.array([0])

        #Filter ngrams through vocabulary PROBLEM
        found_gram = [w for w in ngram if w in filtered_vocab]



        print(found_gram)

        N = len(found_gram)
        print(N)

        vocab_size = len(filtered_vocab)


        dim_row = N
        dim_columns = vocab_size


        tf_idf = [[0 for j in range(dim_columns)] for i in range(d
        print(tf_idf)

        for i in range(N):

            tokens = found_gram[i]
            counter = Counter(tokens)#Replace with count vector
            words_count = len(tokens)

            token_counter =0
            for token in np.unique(tokens):


                tf = counter[token]/words_count
                df = find_doc_freq(token,DF)
                idf = np.log(N/(df+1)) #numerator is added 1 to av

                # df=total number of documents in which nth word o

                tf_idf[i][token_counter] = tf*idf
                token_counter +=1
#                 tf_idf[doc, token] = tf*idf

#             doc += 1
#             token_counter += 1

        print(np.array(tf_idf))
        return np.array(tf_idf)







    #list of special characters.You can use regular expressions to
    special_char=[",",":"," ",";",".","?","'"]
```

```
    #split the sentences into tokens
    x_sub = re.sub(r"[^a-zA-Z0-9\s]", " ", str(X_ngram))

    tokens1 = [token for token in x_sub.split(" ") if token != ""]


    #filter the vocabulary list
    filtered_vocab = [w for w in vocab if w not in stop_words and

    #print(filtered_vocab)

    print("Count Vector...\n")
    vector1=count_vectorize(tokens1)

    print("Start compute_tf_IDF...\n")
    TF_IDF_vector=compute_tf_IDF(tokens1)

```

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 [153]:

```
#COPY COUNT VECTORIZER HERE





#UNIGRAMS, SET_UNIGRAMS
print("Vectorise test text....","\n")
test_count, test_vect = vectorise(test_text, test_vocab)

print("Vectorise train text....","\n")
train_count, train_vect = vectorise(train_text, train_vocab)

print("Vectorise dev text....","\n")
```

Vectorise test text....

In [150]:
```
print('Array Shape = ',np.shape(test_vect) ) # test_vect.shape[0]
print('Array Shape = ',np.shape(train_vect) )

# print(np.shape(train_vect[0:19724, 0:]))

train_vect_sliced = train_vect[0:19724, 0:]#reshape
print('Array Shape = ',np.shape(train_vect_sliced) )

```

```
Array Shape =  (72016, 500)
Array Shape =  (141944, 100)
Array Shape =  (19724, 100)
Array Shape =  (19724, 100)
```

TF.IDF vectors

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

In [ ]:
```
######COMPUTE TF.IFD using Term Frequency, Document Frequency and I

#Copy the TF.IDF method

#      tf_idf = {}
#      for i in range(N):
    #      tokens = processed_text[i]
    #      counter = Counter(tokens)
    #      words_count = len(tokens)

    #      for token in np.unique(tokens):
#              tf = count vector
    #          df = doc_freq(token)

    # ------> idf = np.log(N/(df+1)) <------




#   Formula can be one of these two:
#
#   IDF = 1+log(N/dN)
#
#   idf = log(N/(dN+1))

# Where

# N=Total number of documents in the dataset
```

Then transform your count vectors to tf.idf vectors:

In [ ]:
```
 #          tf = counter[token]/words_count
 # Replace with the count vector
```

In [ ]:
```
#          tf_idf[doc, token] = tf*idf
```

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 [34]:
```
def sigmoid(z):

    sig = 1 / (1 + np.exp(-z))
    return sig

# z = np.dot(X, theta)
# h = sigmoid(z)
```

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 [35]:
```
def predict_proba(X, weights):

    preds_proba = sigmoid(np.dot(X, weights))


```

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 [36]:
```
def predict_class(X, weights):


```

```

    """
#         Predict the class between 0 and 1 using learned logistic
#         Using threshold value 0.5 to convert probability value to

#         I/P
#         ----------
#         X : 2D array where each row represents a docuemnt  and ea
#
#         weights : 1D array of weights. Dimension (1 x |vocab|)

#         O/P
#         -------
#         Class type based on threshold
#         """

    p = preds_proba(X,weights) >= 0.5

    preds_class = p.astype(int)

#   if y_pred_tr>=0.5: #LABELS

#       predictions.append(1)
#   else:
#       predictions.append(0)

    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 [37]:

```
def binary_loss(X, Y, weights, alpha=0.00001):

    """
#         Compute cost for logistic regression.

#         I/P
#         ----------
#         X : 2D array where each row represents a document  and ea
#
#         y : 1D array of labels/target value for each traing examp

#         weights : 1D array of fitting parameters or weights. Dime

#         alpha: regularisation strengths to be added when calculat

#         O/P
#         -------
```

```
#         J : The cost of using theta as the parameter for linear r
#         """


    m = len(X)
    yhat = sigmoid(np.dot(X, weights) + alpha)

    predict = Y * np.log(yhat) + (1 - Y) * np.log(1 - yhat)

    l = -sum(predict) / m




    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 [174]:

```
def SGD(X_tr, Y_tr, X_dev, Y_dev, lr,
        alpha, epochs,
        tolerance, print_progress):

#         X = # data points with some features which we want to tr
#         y = # labels of all datapoints
#         # Initialize the weights and bias i.e. 'm' and 'c'
#         m = np.zeros_like(X[0]) # array with shape equal to no.
#         c = 0#regularisation
#         LR = 0.0001   # The learning Rate
#         epochs = 50 # no. of iterations for optimization


#     w=np.zeros(shape=(1,train_data.shape[1]-1))
```

```

#     C = f_integ(np.array([1]))
#     print "C", C
    m_tr = np.zeros_like(X_tr)

    m_dev = np.zeros_like(X_dev)

    alpha_tr = alpha
    alpha_dev = alpha


    training_loss_history = np.array([0])
    validation_loss_history = np.array([0])

    training_loss_prev = np.array([0])
    validation_loss_prev = np.array([0])

    training_loss_current = np.array([0])
    validation_loss_current = np.array([0])


    # for every epoch
    for epoch in range(1,epochs+1):

        ####TRAINNING####
        # for every data point(X_train,y_train)
        for i in range(len(X_tr)):

            #compute gradient w.r.t 'm'
            form_train = np.dot(X_tr[i], m_tr.T) + alpha_tr

            gr_wrt_m_tr = X_tr[i]*(Y_tr[i] - sigmoid(form_train))

            #compute gradient w.r.t 'c'
            gr_wrt_c_tr = Y_tr[i] - sigmoid(form_train)        #up

            m_tr = m_tr - lr * gr_wrt_m_tr

            alpha_tr = alpha_tr - lr * gr_wrt_c_tr# At the end of




        if training_loss_prev == np.array([0]):

            training_loss_prev = binary_loss(X_tr,Y_tr,m_tr,alpha_
            training_loss_history = np.append(training_loss_histor

        else:

            training_loss_current = binary_loss(X_tr,Y_tr,m_tr,alp

            if (training_loss_current - training_loss_prev) >= tol

                training_loss_history = np.append(training_loss_hi
                training_loss_prev = training_loss_current


        #         if i % 10000 == 0:
        if print_progress == True:
```

```
            print("Loss after %d steps is: %.10f " % (epoch,traini

        ####Development####
        # for every data point(X_train,y_train)
        for j in range(len(X_dev)):

            #compute gradient w.r.t 'm'
            form_train = np.dot(X_dev[j], m_dev.T) + alpha_dev

#             In [1]: import numpy

#             In [2]: numpy.dot(numpy.ones([97, 2]), numpy.ones([2
#             Out[2]: (97, 1)

            gr_wrt_m_dev = X_dev[j]*(Y_dev[j] - sigmoid(form_train

            #compute gradient w.r.t 'c'
            gr_wrt_c_dev = Y_tr[j] - sigmoid(form_train)        #u

            m_dev = m_dev - lr * gr_wrt_m_dev

            alpha_dev = alpha_dev - lr * gr_wrt_c_dev# At the end




        if validation_loss_prev == np.array([0]):

            validation_loss_prev = binary_loss(X_dev,Y_dev,m_dev,a
            validation_loss_history = np.append(validation_loss_hi

        else:

            validation_loss_current = binary_loss(X_dev,Y_dev,m_de

            if (validation_loss_current - validation_loss_prev) >=

                validation_loss_history = np.append(validation_los
                validation_loss_prev = validation_loss_current



#         validation_loss_history = np.append(validation_loss_hist

        if print_progress == True:
            print("Loss after %d steps is: %.10f " % (epoch,valida


#     weights

#     binary_loss(X_tr,Y_tr,m_tr,alpha_tr)

#     binary_loss(X_dev,Y_dev,m_dev,alpha_dev)

    if print_progress == True:
        print("Final loss after %d steps is: %.10f " % (epoch,trai
        print("Loss after %d steps is: %.10f " % (epoch,validation
        print("Final weights for trainning: ", m_tr,"\n")
        print("Final weights for development: ", m_dev,"\n")

```

```
    weigths = np.array([0])
    weigths = np.append(weigths, m_tr)
    weigths = np.append(weigths, m_dev)

    # So by using those optimum values of 'm' and 'c' we can perfo
    #############MAYBE CALL predict class ###########
#     for i in range(len(X_tr)):
#         z_tr = np.dot(X_tr[i], m) + alpha
#         y_pred_tr = sigmoid(z_tr)

#         if y_pred_tr>=0.5: #LABELS
#             predictions.append(1)
#         else:
#             predictions.append(0)

#     for i in range(len(X_dev)):

#         z_dev = np.dot(X_dev[i], m) + alpha
#         y_pred_dev = sigmoid(z_dev)

#         if y_pred_dev>=0.5:#LABELS
#             predictions.append(1)
#         else:
#             predictions.append(0)





    # Make a prediction with coefficients
#     def predict(row, coefficients):
#         yhat = coefficients[0]
#         for i in range(len(row)-1):
#             yhat += coefficients[i + 1] * row[i]
#         return 1.0 / (1.0 + exp(-yhat))


#     # Estimate logistic regression coefficients using stochastic
#     def coefficients_sgd(train, l_rate, n_epoch):
#         coef = [0.0 for i in range(len(train[0]))]
#         for epoch in range(n_epoch):
#             for row in train:
#                 yhat = predict(row, coef)
#                 error = row[-1] - yhat
#                 coef[0] = coef[0] + l_rate * error * yhat * (1.0
#                 for i in range(len(row)-1):
#                     coef[i + 1] = coef[i + 1] + l_rate * error *
#         return coef

#     # Linear Regression Algorithm With Stochastic Gradient Desce
#     def logistic_regression(train, test, l_rate, n_epoch):
#         predictions = list()
#         coef = coefficients_sgd(train, l_rate, n_epoch)
#         for row in test:
#             yhat = predict(row, coef)
#             yhat = round(yhat)
#             predictions.append(yhat)
#         return(predictions)

#     def MyCustomSGD(train_data,learning_rate,n_iter,k,divideby):
```

```
#         # Initially we will keep our W and B as 0 as per the Tra
#         w=np.zeros(shape=(1,train_data.shape[1]-1))
#         b=0

#         cur_iter=1
#         while(cur_iter<=n_iter):

#             # We will create a small training data set of size K
#             temp=train_data.sample(k)

#             # We create our X and Y from the above temp dataset
#             y=np.array(temp['price'])
#             x=np.array(temp.drop('price',axis=1))

#             # We keep our initial gradients as 0
#             w_gradient=np.zeros(shape=(1,train_data.shape[1]-1))
#             b_gradient=0

#             for i in range(k): # Calculating gradients for point
#                 prediction=np.dot(w,x[i])+b
#                 w_gradient=w_gradient+(-2)*x[i]*(y[i]-(predictio
#                 b_gradient=b_gradient+(-2)*(y[i]-(prediction))

#             #Updating the weights(W) and Bias(b) with the above
#             w=w-learning_rate*(w_gradient/k)
#             b=b-learning_rate*(b_gradient/k)

#             # Incrementing the iteration value
#             cur_iter=cur_iter+1

#             #Dividing the learning rate by the specified value
#             learning_rate=learning_rate/divideby

#         return w,b #Returning the weights and Bias
    ################################################################
#     class LogisticRegressionCustom():

#     def __init__(self, l_rate=1e-5, n_iterations=50000):
#         self.l_rate = l_rate
#         self.n_iterations = n_iterations

#     def initial_weights(self, X):
#         self.weights = np.zeros(X.shape[1])

#     def sigmoid(self, s):
#         return 1/(1+np.exp(-s))

    #      m = len(X)
#     yhat = sigmoid(np.dot(X, weights) + alpha)

#     predict = Y * np.log(yhat) + (1 - Y) * np.log(1 - yhat)

#     l = -sum(predict) / m
#     return l

#     def binary_cross_entropy(self, X, y):
#         return -(1/len(y))*(y*np.log(self.sigmoid(np.dot(X,self.

#     def gradient(self, X, y):
```

```
#          return np.dot(X.T, (y-self.sigmoid(np.dot(X,self.weights

#      def fit(self, X, y):
#          self.initial_weights(X)
#          for i in range(self.n_iterations):
#              self.weights = self.weights+self.l_rate*self.gradien
#              if i % 10000 == 0:
#                  print("Loss after %d steps is: %.10f " % (i,self
#          print("Final loss after %d steps is: %.10f " % (i,self.b
#          print("Final weights: ", self.weights)
#          return self

#      def predict(self, X):
#          y_predict = []
#          for t in X:
#              y_predict.append(1) if self.sigmoid(np.dot(self.weig
#          return y_predict

#      def predict_proba(self, X):
#          y_predict = []
#          for t in X:
#              y_predict.append(self.sigmoid(np.dot(self.weights,t)
#          return y_predict

###################################################################



#          def sigmoid(z):
#           sig = 1/(1+np.exp(-z))
#           return sig# Performing Gradient Descent Optimization







    return weights, training_loss_history, validation_loss_history
```

Train and Evaluate Logistic Regression with Count vectors

First train the model using SGD:

In [175]:

```
print(type(train_vect))
print(np.shape(train_label))
print(type(train_count))

#BOW-count

weights, training_loss_history, validation_loss_history = SGD(trair
                                                                alpha
                                                                toler

# (X_tr, Y_tr, X_dev, Y_dev, lr=0.1,
#          alpha=0.00001, epochs=5,
#          tolerance=0.0001, print_progress=True):

# print("Vectorise test text....","\n")
# test_vect = vectorise(test_text, test_vocab)
```

```
# print("Vectorise train text....","\n")
# train_vect = vectorise(train_text, train_vocab)
# print("Vectorise dev text....","\n")
# dev_vect = vectorise(dev_text, dev_vocab)


# print(train_label,"\n")


# ##############################################


# #put the testing raw texts into Python lists
# test_text = list(test[test_column_names[0]])

# #print the text for verification
# #print(test_text,"\n")

# #put the testing labels into a NumPy arrays
# test_label = test[test_column_names[1]].values

# #print the label for verification
# print(test_label,"\n")


# ##############################################


# #put the development raw texts into Python lists
# dev_text = list(dev[dev_column_names[0]])

# #print the text for verification
# #print(dev_text,"\n")

# #put the development labels into a NumPy arrays
# dev_label = dev[dev_column_names[1]].values

# #print the label for verification
# print(dev_label,"\n")
```

```
<class 'numpy.ndarray'>
(1399,)
<class 'list'>

<ipython-input-34-6527dd331435>:3: RuntimeWarning: overflow encounte
red in exp
  sig = 1 / (1 + np.exp(-z))
<ipython-input-37-8d7bf092e40c>:8: RuntimeWarning: divide by zero en
countered in log
  predict = Y * np.log(yhat) + (1 - Y) * np.log(1 - yhat)

---------------------------------------------------------------------
-------
ValueError                               Traceback (most recent cal
l last)
<ipython-input-175-e176f2c34a9e> in <module>
      5 #BOW-count
      6
----> 7 weights, training_loss_history, validation_loss_history = SG
D(train_count, train_label,dev_count, dev_label, lr=0.1,
      8
```

```
alpha=0.00001, epochs=5,
      9
tolerance=0.0001, print_progress=True)

<ipython-input-174-58f0580d6eb2> in SGD(X_tr, Y_tr, X_dev, Y_dev, l
r, alpha, epochs, tolerance, print_progress)
     58         if training_loss_prev == np.array([0]):
     59
---> 60             training_loss_prev = binary_loss(X_tr,Y_tr,m_tr,
alpha_tr)
     61             training_loss_history = np.append(training_loss_
history, training_loss_prev)
     62

<ipython-input-37-8d7bf092e40c> in binary_loss(X, Y, weights, alpha)
      6     yhat = sigmoid(np.dot(X, weights) + alpha)
      7
----> 8     predict = Y * np.log(yhat) + (1 - Y) * np.log(1 - yhat)
      9
     10     l = -sum(predict) / m

ValueError: operands could not be broadcast together with shapes (13
99,) (100,)
```

In [ ]:

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 [ ]:

```
# #plot

# from sklearn.metrics import roc_curve, roc_auc_score
# fpr, tpr, _ = roc_curve(y_test,  y_prob)
# auc = roc_auc_score(y_test, y_prob)

# plt.figure(figsize=(10,8))
# plt.plot(fpr,tpr,label="data, auc="+str(round(auc,4)))

# plt.xlabel("False Positive Rate")
# plt.ylabel("True Positive Rate")

# plt.title("ROC Curve for Model from Sratch")
# plt.legend(loc=4)
# plt.show()

####################### MAIN #####################
# training_loss_history
# validation_loss_history

plt.figure(figsize=(25,6))

plt.title('Cost Function Slope')
plt.plot(training_loss_history, label='Training Loss History')
plt.plot(validation_loss_history, label='Validation Loss History')
plt.legend(prop={'size': 16})
plt.xlabel('Number of Iterations')
plt.ylabel('Error Values')
plt.show()
```

```
                                 
########################## MAIN ########################

# plt.figure(figsize=(10,8))
# plt.title('Cost Function Slope')
# plt.plot(cost)
# plt.xlabel('Number of Iterations')
# plt.ylabel('Error Values')
```

Explain here...

In [ ]:

```
#Underfit??

#Overfit??

```

Evaluation

Compute accuracy, precision, recall and F1-scores:

In [170]:

```
X_te_count = train_count

w_count = weights

preds_te_count = predict_class(X_te_count, w_count)

# train_count, weights


Y_te = dev_count

print('Accuracy:', accuracy_score(Y_te,preds_te_count))
print('Precision:', precision_score(Y_te,preds_te_count))
print('Recall:', recall_score(Y_te,preds_te_count))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-170-f6a9ad36b6d1> in <module>
      1 X_te_count = train_count
      2 
----> 3 w_count = weights
      4 
      5 preds_te_count = predict_class(X_te_count, w_count)

NameError: name 'weights' is not defined
```

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

In [171]:

```
# id_w_test, w_id_test = create_2dict(test_df)
# print("Train Dictionary >>> ", "\n")

# id_w_train, w_id_train =create_2dict(train_df)
# print("DEV Dictionary >>> ", "\n")

```

```
# id_w_dev, w_id_dev =create_2dict(dev_df)

top_neg = w_count.argsort()[:10]
for i in top_neg:
#     print(id2word[i])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-171-e06a79cbdd75> in <module>
      7 # id_w_dev, w_id_dev =create_2dict(dev_df)
      8 
----> 9 top_neg = w_count.argsort()[:10]
     10 for i in top_neg:
     11 #     print(id2word[i])

NameError: name 'w_count' is not defined
```

In [172]:

```
top_pos = w_count.argsort()[::-1][:10]
for i in top_pos:
#     print(id2word[i])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-172-b8de8e434f3d> in <module>
----> 1 top_pos = w_count.argsort()[::-1][:10]
      2 for i in top_pos:
      3 #     print(id2word[i])
      4     print(id_w_train)

NameError: name 'w_count' is not defined
```

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?

Provide your answer here...

Sentiment Analysis

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?

Enter your answer here...

(e.g. learning rate and regularisation strength)

Train and Evaluate Logistic Regression with TF.IDF

vectors

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

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

BOW-tfidf:

In [ ]:

```
###########BOW-tfidf###########

# #TEST
# test_vocab, test_df, test_count = get_vocab(test_text, ngram_rang
#                          min_df=2, keep_topN=500,
#                          stop_words = stop_words,char_ngrams=False)

# #train
# train_vocab, train_df, train_count = get_vocab(train_text, ngram_
#                          min_df=10, keep_topN=100,
#                          stop_words=stop_words,char_ngrams=False)
# #Dev
# dev_vocab, dev_df, dev_count = get_vocab(dev_text, ngram_range=(1
#                          min_df=10, keep_topN=100,
#                          stop_words=stop_words,char_ngrams=False)

# #Test Vectorisation
# print("Vectorise test text....","\n")
# test_count, test_vect = vectorise(test_text, test_vocab)

# #Train Vectorisation
# print("Vectorise train text....","\n")
# train_count, train_vect = vectorise(train_text, train_vocab)

# #Dev Vectorisation
# print("Vectorise dev text....","\n")
# dev_count, dev_vect = vectorise(dev_text, dev_vocab)


weights_tfidf, training_loss_history_tfidf, validation_loss_history
                                                              alpha
                                                              toler

X_te_count = train_vect

w_count = weights_tfidf

preds_te_count = predict_class(X_te_count, w_count)

# train_count, weights


Y_te = dev_vect

print('Accuracy:', accuracy_score(Y_te,preds_te_count))
print('Precision:', precision_score(Y_te,preds_te_count))
```

```
print('Recall:', recall_score(Y_te,preds_te_count))
print('F1-Score:', f1_score(Y_te,preds_te_count))

# training_loss_history
# validation_loss_history

plt.figure(figsize=(25,6))

plt.title('Cost Function Slope')
plt.plot(training_loss_history_tfidf, label='Training Loss History'
plt.plot(validation_loss_history_tfidf, label='Validation Loss Hist
plt.legend(prop={'size': 16})
plt.xlabel('Number of Iterations')
plt.ylabel('Error Values')
plt.show()

# print("Test Dictionary >>> ", "\n")
# id_w_test, w_id_test = create_2dict(test_df)

# print("Train Dictionary >>> ", "\n")
# id_w_train, w_id_train =create_2dict(train_df)

# print("DEV Dictionary >>> ", "\n")
# id_w_dev, w_id_dev =create_2dict(dev_df)



top_neg = w_count.argsort()[:10]
for i in top_neg:
#     print(id2word[i])
    print(id_w_train[i])

top_pos = w_count.argsort()[::-1][:10]
for i in top_pos:
#     print(id2word[i])
    print(id_w_train[i])
```

BOCN-count:

In [ ]:

```
############ BOCN-count ############

# #TEST
# test_vocab, test_df, test_count = get_vocab(test_text, ngram_rang
#                          min_df=2, keep_topN=500,
#                          stop_words = stop_words,char_ngrams=True)

#train
train_vocab_BOCN, train_df_BOCN, train_count_BOCN = get_vocab(trair
                       min_df=10, keep_topN=100,
                       stop_words=stop_words,char_ngrams=True)
#Dev
dev_vocab_BOCN, dev_df_BOCN, dev_count_BOCN = get_vocab(dev_text, r
                       min_df=10, keep_topN=100,
                       stop_words=stop_words,char_ngrams=True)

# #Test Vectorisation
# print("Vectorise test text....","\n")
# test_count, test_vect = vectorise(test_text, test_vocab)

```

```
#Train Vectorisation
print("Vectorise train text....","\n")
train_count_BOCN, train_vect_BOCN = vectorise(train_text, train_voc

#Dev Vectorisation
print("Vectorise dev text....","\n")
dev_count_BOCN, dev_vect_BOCN = vectorise(dev_text, dev_vocab_BOCN)


weights_BOCN, training_loss_history_BOCN, validation_loss_history_E



X_te_count = train_count_BOCN

w_count = weights_BOCN

preds_te_count = predict_class(X_te_count, w_count)

# train_count, weights


Y_te = dev_count_BOCN

print('Accuracy:', accuracy_score(Y_te,preds_te_count))
print('Precision:', precision_score(Y_te,preds_te_count))
print('Recall:', recall_score(Y_te,preds_te_count))
print('F1-Score:', f1_score(Y_te,preds_te_count))

# training_loss_history
# validation_loss_history

# plt.figure(figsize=(25,6))

plt.title('Cost Function Slope')
plt.plot(training_loss_history_BOCN, label='Training Loss History')
plt.plot(validation_loss_history_BOCN, label='Validation Loss Histo
plt.legend(prop={'size': 16})
plt.xlabel('Number of Iterations')
plt.ylabel('Error Values')
plt.show()

# print("Test Dictionary >>> ", "\n")
# id_w_test, w_id_test = create_2dict(test_df)

print("Train Dictionary >>> ", "\n")
id_w_train_BOCN, w_id_train_BOCN =create_2dict(train_df_BOCN)

print("DEV Dictionary >>> ", "\n")
id_w_dev_BOCN, w_id_dev_BOCN =create_2dict(dev_df_BOCN)



top_neg = w_count.argsort()[:10]
for i in top_neg:
#     print(id2word[i])
    print(id_w_train_BOCN[i])

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

```
#     print(id2word[i])
    print(id_w_train_BOCN[i])
```

BOCN-tfidf:

In [ ]:

```
############ BOCN-tfidf ############

# #TEST
# test_vocab, test_df, test_count = get_vocab(test_text, ngram_rang
#                          min_df=2, keep_topN=500,
#                          stop_words = stop_words,char_ngrams=True)

#train
# train_vocab, train_df, train_count = get_vocab(train_text, ngram_
#                          min_df=10, keep_topN=100,
#                          stop_words=stop_words,char_ngrams=True)
# #Dev
# dev_vocab, dev_df, dev_count = get_vocab(dev_text, ngram_range=(1
#                          min_df=10, keep_topN=100,
#                          stop_words=stop_words,char_ngrams=True)

# #Test Vectorisation
# print("Vectorise test text....","\n")
# test_count, test_vect = vectorise(test_text, test_vocab)

#Train Vectorisation
# print("Vectorise train text....","\n")
# train_count, train_vect = vectorise(train_text, train_vocab)

# #Dev Vectorisation
# print("Vectorise dev text....","\n")
# dev_count, dev_vect = vectorise(dev_text, dev_vocab)


weights_BOCN_tfidf, training_loss_history_BOCN_tfidf, validation_lo



X_te_count = train_vect_BOCN_tfidf

w_count = weights_BOCN_tfidf

preds_te_count = predict_class(X_te_count, w_count)

# train_count, weights


Y_te = dev_vect_BOCN

print('Accuracy:', accuracy_score(Y_te,preds_te_count))
print('Precision:', precision_score(Y_te,preds_te_count))
print('Recall:', recall_score(Y_te,preds_te_count))
print('F1-Score:', f1_score(Y_te,preds_te_count))

# training_loss_history
# validation_loss_history

plt.figure(figsize=(25,6))

```

```
plt.title('Cost Function Slope')
plt.plot(training_loss_history_BOCN_tfidf, label='Training Loss His
plt.plot(validation_loss_history_BOCN_tfidf, label='Validation Loss
plt.legend(prop={'size': 16})
plt.xlabel('Number of Iterations')
plt.ylabel('Error Values')
plt.show()

# print("Test Dictionary >>> ", "\n")
# id_w_test, w_id_test = create_2dict(test_df)

# print("Train Dictionary >>> ", "\n")
# id_w_train, w_id_train =create_2dict(train_df)

# print("DEV Dictionary >>> ", "\n")
# id_w_dev, w_id_dev =create_2dict(dev_df)



top_neg = w_count.argsort()[:10]
for i in top_neg:
#     print(id2word[i])
    print(id_w_train_BOCN_tfidf[i])

top_pos = w_count.argsort()[::-1][:10]
for i in top_pos:
#     print(id2word[i])
    print(id_w_train_BOCN_tfidf[i])
```

BOW+BOCN:

In [ ]:

Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count			
BOW-tfidf			
BOCN-count			
BOCN-tfidf			
BOW+BOCN			

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

In [ ]:

In [ ]: