assignment1

April 17, 2020

1 [COM4513-6513] Assignment 1: Text Classification with Logistic Regression

1.0.1 Instructor: Nikos Aletras

The goal of this assignment is to develop and test two text classification systems:

- **Task 1:** sentiment analysis, in particular to predict the sentiment of movie review, i.e. positive or negative (binary classification).
- **Task 2:** topic classification, to predict whether a news article is about International issues, Sports or Business (multiclass classification).

For that purpose, you will implement:

- Text processing methods for extracting Bag-Of-Word features, using (1) unigrams, bigrams and trigrams to obtain vector representations of documents. Two vector weighting schemes should be tested: (1) raw frequencies (**3 marks; 1 for each ngram type**); (2) tf.idf (**1 marks**).
- Binary Logistic Regression classifiers that will be able to accurately classify movie reviews trained with (1) BOW-count (raw frequencies); and (2) BOW-tfidf (tf.idf weighted) for Task 1.
- Multiclass Logistic Regression classifiers that will be able to accurately classify news articles trained with (1) BOW-count (raw frequencies); and (2) BOW-tfidf (tf.idf weighted) for Task 2.
- 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 for Task 1 (**3 marks**)
  - Minimise the Categorical Cross-entropy loss function for Task 2 (**3 marks**)
  - Use L2 regularisation (both tasks) (**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 validation 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)? (**2 marks; 0.5 for each model in each task**).

- After training the LR models, plot the learning process (i.e. training and validation loss in each epoch) using a line plot (**1 mark; 0.5 for both BOW-count and BOW-tfidf LR models in each task**) and discuss if your model overfits/underfits/is about right.
- Model interpretability by showing the most important features for each class (i.e. most positive/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 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? (**2 marks; 0.5 for BOW-count and BOW-tfidf LR models respectively in each task**)

1.0.2 Data - Task 1

The data you will use for Task 1 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.

1.0.3 Data - Task 2

The data you will use for Task 2 is a subset of the AG News Corpus and you can find it in the `./data_topic` folder in CSV format:

- `data_topic/train.csv`: contains 2,400 news articles, 800 for each class to be used for training.
- `data_topic/dev.csv`: contains 150 news articles, 50 for each class to be used for hyperparameter selection and monitoring the training process.
- `data_topic/test.csv`: contains 900 news articles, 300 for each class to be used for testing.

1.0.4 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`).

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, NumPy, SciPy 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..

Please make sure to comment your code. You should also mention if you've used Windows (not recommended) to write and test your code. 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 Fri, 20 Mar 2020** and it needs to be submitted via MOLE. Standard departmental penalties for lateness will be applied. We use a range of strategies to detect unfair means, including Turnitin which helps detect plagiarism, so make sure you do not plagiarise.

[1919]:
```
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

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

1.1 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).

[1920]:
```
# fill in your code...
data_tr = pd.read_csv('./data_sentiment/train.
 ↪csv',header=None,names=['text','label'])
data_te = pd.read_csv('./data_sentiment/test.
 ↪csv',header=None,names=['text','label'])
data_de = pd.read_csv('./data_sentiment/dev.
 ↪csv',header=None,names=['text','label'])
```

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

[1921]:
```
data_tr.head()
#data_te.head()
#data_de.head()
```

[1921]:
```
                                                text  label
0  note : some may consider portions of the follo...      1
1  note : some may consider portions of the follo...      1
2  every once in a while you see a film that is s...      1
3  when i was growing up in 1970s , boys in my sc...      1
4  the muppet movie is the first , and the best m...      1
```

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

[1922]:
```
# fill in your code...
#train data transformation
X_tr_raw = data_tr['text'].tolist()
```

```
Y_tr = np.array(data_tr['label'])
#test data transformatino
X_te_raw = data_te['text'].tolist()
Y_te = np.array(data_te['label'])
#development data transformatino
X_de_raw = data_de['text'].tolist()
Y_de = np.array(data_de['label'])
```

2 Bag-of-Words Representation

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).

2.1 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 - remove ngrams appearing in less than K documents - use the remaining to create a vocabulary of unigrams, bigrams and trigrams (you can keep top N if you encounter memory issues).

[1923]:
```
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', 'us','they', 'will', 'have', 'has',
              'do', 'did', 'can', 'could', 'who', 'which', 'what',
             'his', 'her', 'they', 'them', 'their','from', 'with', 'its']
```

2.1.1 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.

and returns:

- a list of all extracted features.

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

[1924]:
```
def extract_ngrams(x_raw, ngram_range=(1,3),␣
 ↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b', stop_words=[], vocab=set()):
    # fill in your code...
    unigram = []
    bigram = []
    trigram = []
    # get unigram by using regex
    pattern = re.compile(token_pattern)
    x_re = pattern.findall(x_raw.lower())
    # remove stop words
    for i in range(len(x_re)):
        if x_re[i] not in stop_words:
            unigram.append(x_re[i])
    # return bigram or trigram
    if ngram_range==(1,3):
        for i in range(len(unigram)-1):
            bigram.append((unigram[i],unigram[i+1]))
        for i in range(len(unigram)-2):
            trigram.append((unigram[i],unigram[i+1],unigram[i+2]))
        x = unigram + bigram + trigram

    if ngram_range==(1,2):
        for i in range(len(unigram)-1):
            bigram.append((unigram[i],unigram[i+1]))
        x = unigram + bigram
    if vocab:
        x = list(vocab)
    return x
```

[1925]:
```
extract_ngrams("this is a great movie to watch ",
               ngram_range=(1,3),
               stop_words=stop_words)
```

[1925]:
```
['great',
 'movie',
 'watch',
 ('great', 'movie'),
 ('movie', 'watch'),
 ('great', 'movie', 'watch')]
```

[1926]:
```
extract_ngrams("this is a great movie to watch",
               ngram_range=(1,2),
               stop_words=stop_words,
               vocab=set(['great',  ('great','movie')]))
```

[1926]:
```
['great', ('great', 'movie')]
```

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

Create a vocabulary of n-grams Then 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.

[1927]:
```
def get_vocab(X_raw, ngram_range=(1,3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b',␣
↪min_df=0, keep_topN=0, stop_words=[]):
    # fill in your code..
    ngrams = []
    ngrams_df = []
    for i in X_raw:
        n_grams =␣
↪extract_ngrams(i,ngram_range=ngram_range,token_pattern=token_pattern,stop_words=stop_words)
        ngrams += n_grams
        ngrams_df += list(set(n_grams))
    more_frequent_ngrams = Counter(ngrams).most_common(keep_topN)
    #get more frequent ngrams key
    features = []
    ngram_counts = 0
    for k in more_frequent_ngrams:
        features.append(k[0])
        ngram_counts += k[1]
    vocab = set(features)
    #get document frequency for ngrams
    df = Counter(ngrams_df)
    # set min df for ngrams whose df is 0
    for i in features:
        if i not in ngrams_df:
            df[i]=min_df
    return vocab, df, ngram_counts
```

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

[1928]:
```
vocab, df, ngram_counts = get_vocab(X_tr_raw,␣
↪ngram_range=(1,3),keep_topN=5000,stop_words=stop_words)
print(len(vocab))
print()
print(list(vocab)[:100])
print()
print(df.most_common()[:10])
```

```
5000
```

```
['pre', ('some', 'scenes'), 'ann', 'struggles', 'empty', ('high', 'school'),
'number', 'barry', 'ace', 'threatening', 'henstridge', 'ignore', 'exploring',
'ensemble', 'motion', 'record', 'suffice', 'thrilling', 'walking',
'predictable', 'bunch', 'machines', ('but', 'only'), 'bloody', ('not', 'so'),
'natalie', 'surely', 'non', 'ability', 'impact', 'life', 'line', 'emotions',
('first', 'film'), 'duchovny', 'featuring', ('but', 'never'), 'matters',
'stolen', 'below', 'hard', 'round', 'wanting', 'albert', ('not', 'enough'),
'tense', 'darth', 'bruce', 'chicken', 'actress', 'oh', 'edward', ('but', 'so'),
'police', 'jewish', 'degree', 'floor', 'happen', ('so', 'bad'), 'peak', 'raimi',
('all', 'other'), 'warrior', ('dante', 'peak'), 'ideal', 'trio', 'guard',
'followed', 'predecessor', 'mainly', 'points', ('some', 'other'), 'headed',
'fishburne', 'looking', 'along', ('private', 'ryan'), 'upcoming', 'created',
'richards', ('mel', 'gibson'), 'together', 'lazy', 'murdered', ('while', 'not'),
'families', 'plus', 'wong', 'west', ('film', 'one'), ('go', 'through'),
'duvall', 'mini', 'mitchell', 'explain', 'vampires', 'fugitive', 'knight',
'top', ('fifteen', 'minutes')]

[('but', 1334), ('one', 1247), ('film', 1231), ('not', 1170), ('all', 1117),
('movie', 1095), ('out', 1080), ('so', 1047), ('there', 1046), ('like', 1043)]
```

Then, you need to create vocabulary id -> word and id -> word dictionaries for reference:

[1929]:
```
# fill in your code...
id_word_dic = dict(enumerate(vocab))
word_id_dic = {v:k for k,v in id_word_dic.items()}
```

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

[1930]:
```
# get X_ngram
def get_X_ngram(X_raw,vocab,ngram_range=(1,3),␣
↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',stop_words=stop_words):
    # create a list of docs,each doc is a list of ngrams
    N_grams = []
    for i in X_raw:
        n_grams =␣
↪extract_ngrams(i,ngram_range=ngram_range,token_pattern=token_pattern,stop_words=stop_words)
        N_grams.append(n_grams)
    # remove those ngrams which are not in vocab
    X_ngram = []
    for texts in N_grams:
        for ngram in texts:
            if ngram not in vocab:
                texts.remove(ngram)
        X_ngram.append(texts)
    return X_ngram
X_tr_ngram = get_X_ngram(X_tr_raw,vocab,ngram_range=(1,3),␣
↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',stop_words=stop_words)
```

```
X_te_ngram = get_X_ngram(X_te_raw,vocab,ngram_range=(1,3),␣
 ↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',stop_words=stop_words)
X_de_ngram = get_X_ngram(X_de_raw,vocab,ngram_range=(1,3),␣
 ↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',stop_words=stop_words)
```

2.2 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.

[1931]:
```
def vectorise(X_ngram, vocab):
    # fill in your code...
    N = len(X_ngram)
    X_vec = np.zeros((N,len(vocab)))
    vocab_list = list(vocab)
    for i in range(N):
        for j in range(len(vocab)):
            #count word frequency in each doc
            X_vec[i][j]=X_ngram[i].count(vocab_list[j])
    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:

[1932]:
```
X_tr_count = vectorise(X_tr_ngram, vocab)
X_te_count = vectorise(X_te_ngram, vocab)
X_de_count = vectorise(X_de_ngram, vocab)
```

Count vectors

[1933]:
```
X_tr_count.shape
```

[1933]:
```
(1400, 5000)
```

[1934]:
```
X_tr_count[:2,:50]
```

[1934]:
```
array([[0., 0., 0., 0., 0., 0., 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.,
        1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 0.,
        0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.]])
```

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

[1935]:
```
# fill in your code...
def get_idf(X_raw,vocab,df):
    N = len(X_raw)
    idf = np.zeros((len(vocab)))
    vocab_list = list(vocab)
    for i in range(len(vocab)):
        ngram = vocab_list[i]
        idf[i] = np.log10(N/df[ngram])
    return idf
idfs = get_idf(X_tr_raw,vocab,df)
```

Then transform your count vectors to tf.idf vectors:

[1936]:
```
X_tr_tfidf = X_tr_count*idfs
X_te_tfidf = X_te_count*idfs
X_de_tfidf = X_de_count*idfs
```

[1937]:
```
X_tr_tfidf[1,:50]
```

[1937]:
```
array([0.        , 1.66900678, 0.        , 0.        , 0.        ,
       0.        , 1.03553833, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 1.28280518,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.38120505, 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 1.43012469, 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

3 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`

[1938]:
```
def sigmoid(z):

    # fill in your code...
    sig = 1/(1+np.exp(-z))
    return sig
```

[1939]:
```
print(sigmoid(0))
print(sigmoid(np.array([-5., 1.2])))
```

```
0.5
[0.00669285 0.76852478]
```

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

[1940]:
```
def predict_proba(X, weights):
    # fill in your code...
    preds_proba = sigmoid(np.dot(weights,X.T))
    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

[1941]:
```
def predict_class(X, weights):
    # fill in your code...
    preds_proba = predict_proba(X, weights)
    N = preds_proba.shape[0]
    preds_class = np.zeros((N,)).astype(int)
    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

[1942]:
```
def binary_loss(X, Y, weights, alpha=0.00001):
    # fill in your code...
    preds_proba = predict_proba(X, weights)
    L = np.zeros((len(Y),))
    for i in range(len(Y)):
        # loss of each doc
        L[i] = -1*Y[i]*np.log(preds_proba[i])-(1-Y[i])*np.log(1-preds_proba[i])
        # regularisation part
        regular = alpha*np.square(np.sum(np.power(weights,2)))
        L[i] = L[i] + regular
    l = np.mean(L)
    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

[1946]:
```
def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], loss="binary", lr=0.1, alpha=0.00001,␣
 ↪epochs=5, tolerance=0.0001, print_progress=True):

    cur_loss_tr = 1.
    cur_loss_dev = 1.
    training_loss_history = []
    validation_loss_history = []
    # initialize w with zeros
    m,n = X_tr.shape
    weights = np.zeros((n,))
    for i in range(epochs):
        # randomise order in training Data
        permutation = np.random.permutation(m)
        shuffled_x = X_tr[permutation,:]
        shuffled_y = Y_tr[permutation]
        for j in range(len(shuffled_y)):
            # prediction value
            h = predict_proba(shuffled_x[j],weights)
            # compute the error
            error = h - shuffled_y[j]
            # update weights and use L2 regularisation
            weights = weights - lr*((shuffled_x[j]*error)+2*alpha*weights)
        pre_loss_tr = cur_loss_tr
        pre_loss_dev = cur_loss_dev
        cur_loss_tr = binary_loss(shuffled_x,shuffled_y,weights,alpha)
        cur_loss_dev = binary_loss(X_dev,Y_dev,weights,alpha)
        training_loss_history.append(cur_loss_tr)
        validation_loss_history.append(cur_loss_dev)
        if print_progress==True:
            print("Epoch: ",i,"| Training loss: ",cur_loss_tr,"| Validation loss:
↪ ",cur_loss_dev)
        if (pre_loss_dev - cur_loss_dev) < tolerance:
            break


    return weights, training_loss_history, validation_loss_history
```

3.1 Train and Evaluate Logistic Regression with Count vectors

First train the model using SGD:

[1947]:
```
w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr,
                                             X_dev=X_de_count,
                                             Y_dev=Y_de,
                                             lr=0.0001,
                                             alpha=0.001,
                                             epochs=100
                                             )
```

```
Epoch:  0 | Training loss:  0.6287448383180287 | Validation loss:
0.6460804805745204
Epoch:  1 | Training loss:  0.5847009748726169 | Validation loss:
0.6155549977825893
Epoch:  2 | Training loss:  0.5514903395604694 | Validation loss:
```

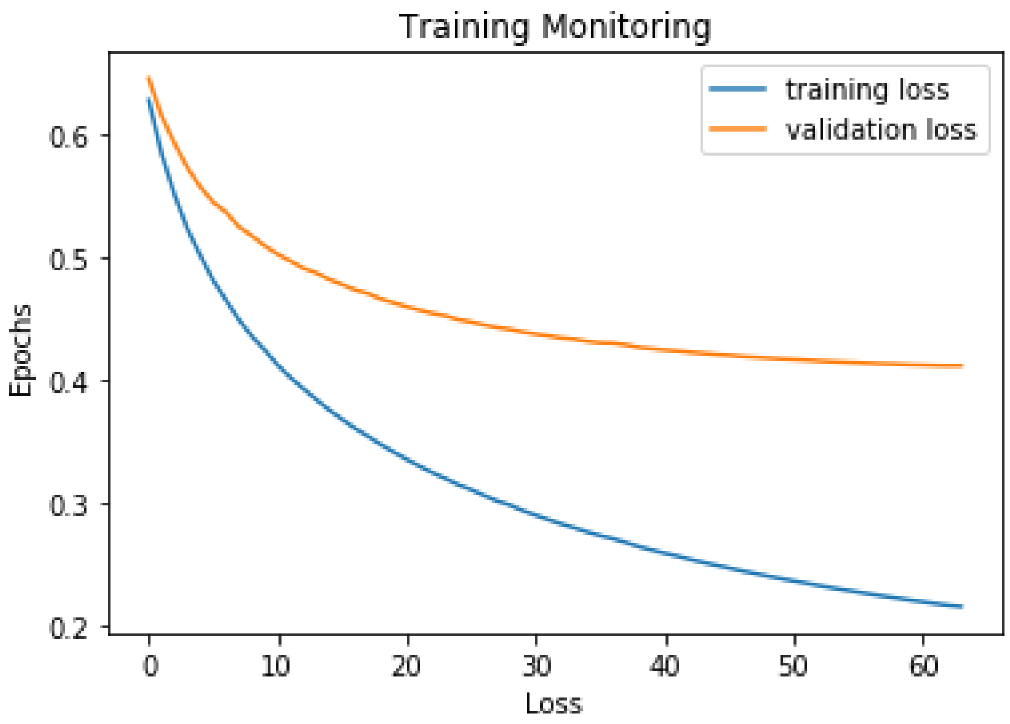
```
0.593369204588327
Epoch:  3 | Training loss:  0.5235667821132639 | Validation loss:
0.5734761277101876
Epoch:  4 | Training loss:  0.5019700388425717 | Validation loss:
0.5578722999651631
Epoch:  5 | Training loss:  0.48168011997147864 | Validation loss:
0.5453146365860989
Epoch:  6 | Training loss:  0.4654191896964703 | Validation loss:
0.537264611135136
Epoch:  7 | Training loss:  0.4494563060260835 | Validation loss:
0.5252515891478492
Epoch:  8 | Training loss:  0.4360583053465348 | Validation loss:
0.5179887824260425
Epoch:  9 | Training loss:  0.4248582315006029 | Validation loss:
0.5098375242346687
Epoch:  10 | Training loss:  0.41260975974958713 | Validation loss:
0.5030286197437708
Epoch:  11 | Training loss:  0.4023022299223317 | Validation loss:
0.4973363395173602
Epoch:  12 | Training loss:  0.3933294388314568 | Validation loss:
0.49155344641076854
Epoch:  13 | Training loss:  0.38446207185867315 | Validation loss:
0.4877625414993604
Epoch:  14 | Training loss:  0.3759684381221936 | Validation loss:
0.48250520240676287
Epoch:  15 | Training loss:  0.3683616711775805 | Validation loss:
0.47838580848454737
Epoch:  16 | Training loss:  0.36097454016012137 | Validation loss:
0.473925593832637
Epoch:  17 | Training loss:  0.35457688066305504 | Validation loss:
0.471030177746228
Epoch:  18 | Training loss:  0.3477275709803097 | Validation loss:
0.46647542258094227
Epoch:  19 | Training loss:  0.3416181905108455 | Validation loss:
0.46332955663067077
Epoch:  20 | Training loss:  0.3358452045039249 | Validation loss:
0.46012523039659287
Epoch:  21 | Training loss:  0.3303087956312837 | Validation loss:
0.457402137519982
Epoch:  22 | Training loss:  0.32506304931143437 | Validation loss:
0.4547532011851533
Epoch:  23 | Training loss:  0.3203170765826494 | Validation loss:
0.4527465879465594
Epoch:  24 | Training loss:  0.31526243583809754 | Validation loss:
0.4496792300744611
Epoch:  25 | Training loss:  0.3112310238186765 | Validation loss:
0.44760627967762806
Epoch:  26 | Training loss:  0.3064793223794771 | Validation loss:
```

```
0.4452801628139267
Epoch:  27 | Training loss:  0.3020586844760524 | Validation loss:
0.44321457049171753
Epoch:  28 | Training loss:  0.29870808457738135 | Validation loss:
0.4417146836816957
Epoch:  29 | Training loss:  0.29411819919081755 | Validation loss:
0.4395435294620444
Epoch:  30 | Training loss:  0.2903874002326494 | Validation loss:
0.437849812291881
Epoch:  31 | Training loss:  0.28677313885918326 | Validation loss:
0.43640055447168263
Epoch:  32 | Training loss:  0.28325397264342456 | Validation loss:
0.43468736194461255
Epoch:  33 | Training loss:  0.28011833152314164 | Validation loss:
0.43371681739412943
Epoch:  34 | Training loss:  0.2767087288509685 | Validation loss:
0.43189773530226766
Epoch:  35 | Training loss:  0.27377621371044225 | Validation loss:
0.4307006152956043
Epoch:  36 | Training loss:  0.27120892286202425 | Validation loss:
0.43042629654015196
Epoch:  37 | Training loss:  0.2681407868400335 | Validation loss:
0.42903206665635324
Epoch:  38 | Training loss:  0.2647358124239204 | Validation loss:
0.42695648528724006
Epoch:  39 | Training loss:  0.2619649246033636 | Validation loss:
0.42591664142306795
Epoch:  40 | Training loss:  0.25930309069090235 | Validation loss:
0.4248105365321222
Epoch:  41 | Training loss:  0.25695575221312994 | Validation loss:
0.42400823135435134
Epoch:  42 | Training loss:  0.2542516795065611 | Validation loss:
0.42299894113416187
Epoch:  43 | Training loss:  0.2518130838741669 | Validation loss:
0.4220654286362428
Epoch:  44 | Training loss:  0.249521098018587 | Validation loss:
0.4212011676309671
Epoch:  45 | Training loss:  0.24725911545014762 | Validation loss:
0.4205430524928969
Epoch:  46 | Training loss:  0.24493572065040728 | Validation loss:
0.41959926699797934
Epoch:  47 | Training loss:  0.24285772862194302 | Validation loss:
0.418947847075571
Epoch:  48 | Training loss:  0.24068863342107769 | Validation loss:
0.4182376588546588
Epoch:  49 | Training loss:  0.2386620537004955 | Validation loss:
0.4176097928296234
Epoch:  50 | Training loss:  0.2367640063935614 | Validation loss:
```

```
0.4170973271377263
Epoch:  51 | Training loss:  0.23488906502142093 | Validation loss:
0.41660959452098967
Epoch:  52 | Training loss:  0.23288575769087955 | Validation loss:
0.4158458972855778
Epoch:  53 | Training loss:  0.23106697225996958 | Validation loss:
0.4153153310888517
Epoch:  54 | Training loss:  0.2293054852169156 | Validation loss:
0.41483509000422936
Epoch:  55 | Training loss:  0.22763069390418436 | Validation loss:
0.4144302132121893
Epoch:  56 | Training loss:  0.22590112899464182 | Validation loss:
0.4139405911797138
Epoch:  57 | Training loss:  0.22429562758716035 | Validation loss:
0.4135588568202402
Epoch:  58 | Training loss:  0.222752191121785 | Validation loss:
0.41323838767359183
Epoch:  59 | Training loss:  0.22115462703978775 | Validation loss:
0.4128082001964299
Epoch:  60 | Training loss:  0.21971674274921255 | Validation loss:
0.41257454249791414
Epoch:  61 | Training loss:  0.21823014225196688 | Validation loss:
0.412232016095769
Epoch:  62 | Training loss:  0.21688017164939163 | Validation loss:
0.41204413571053
Epoch:  63 | Training loss:  0.21578241541954396 | Validation loss:
0.4121543851592924
```

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

[1948]:
```python
epochs = []
for i in range(len(loss_tr_count)):
    epochs.append(i)
plt.plot(epochs, loss_tr_count,label='training loss')
plt.plot(epochs, dev_loss_count,label='validation loss')
plt.legend()
plt.title('Training Monitoring')
plt.xlabel('Loss')
plt.ylabel('Epochs')
plt.show()
```



Explain here…

The model is about right. Because both of the training loss and validation loss are declining and when train processes stop they are close to be convergent.

Compute accuracy, precision, recall and F1-scores:

[1949]:
```
# fill in your code...
preds_te_count = predict_class(X_te_count, w_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))
print('F1-Score:', f1_score(Y_te,preds_te_count))
```

```
Accuracy: 0.84
Precision: 0.84
Recall: 0.84
F1-Score: 0.8399999999999999
```

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

[1950]:
```
# fill in your code..
# top10 negative words
vocab_list = list(vocab)
neg_w = sorted(w_count)
print("top-10 words for negative class:")
for i in range(10):
    index = np.where(w_count == neg_w[i])[0][0]
    print(vocab_list[index])
```

```
top-10 words for negative class:
bad
only
unfortunately
worst
plot
script
why
boring
any
nothing
```

[1951]:
```
# fill in your code...
# top10 positive words
pos_w = sorted(w_count,reverse = True)
print("top-10 words for positive class:")
for i in range(10):
    index = np.where(w_count == pos_w[i])[0][0]
    print(vocab_list[index])
```

```
top-10 words for positive class:
great
well
also
seen
life
fun
many
world
both
movies
```

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...

These features would not generalise commodity reviews perfectly because they include some noun words(movies,world,script,life). These features cannot show wheather reviews are positive or negative. Only those adjective features like emotional words(bad, worst boring great fun...) that are used to evaluate commodity could be picked up.

3.2 Train and Evaluate Logistic Regression with TF.IDF vectors

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

[1952]:
```
w_tfidf, trl, devl = SGD(X_tr_tfidf, Y_tr,
                         X_dev=X_de_tfidf,
                         Y_dev=Y_de,
                         lr=0.0001,
                         alpha=0.00001,
                         epochs=50)
```

```
Epoch:  0 | Training loss:  0.6387394645509009 | Validation loss:
0.6646193340071946
Epoch:  1 | Training loss:  0.5968689754636599 | Validation loss:
0.6429415956760386
Epoch:  2 | Training loss:  0.5623396165166045 | Validation loss:
0.6251052956422272
Epoch:  3 | Training loss:  0.5331379897978898 | Validation loss:
0.6097801594524369
Epoch:  4 | Training loss:  0.5080009553911554 | Validation loss:
0.5967061958522705
Epoch:  5 | Training loss:  0.48598397188207254 | Validation loss:
0.5852016523207958
Epoch:  6 | Training loss:  0.4664557163211444 | Validation loss:
0.5750434455815823
Epoch:  7 | Training loss:  0.4490006784861553 | Validation loss:
0.5659601624233466
Epoch:  8 | Training loss:  0.4332453248112313 | Validation loss:
0.5578201901696366
Epoch:  9 | Training loss:  0.41891181526807875 | Validation loss:
0.5504702286063256
Epoch:  10 | Training loss:  0.40578933346393653 | Validation loss:
0.5437661232597261
Epoch:  11 | Training loss:  0.39363428069185424 | Validation loss:
0.5374856132202338
Epoch:  12 | Training loss:  0.3823978913677033 | Validation loss:
0.531657727382089
Epoch:  13 | Training loss:  0.37196456568343533 | Validation loss:
0.5263272146767536
Epoch:  14 | Training loss:  0.36222390713009184 | Validation loss:
0.5213038380766178
Epoch:  15 | Training loss:  0.3531019409600564 | Validation loss:
0.5166899487179863
Epoch:  16 | Training loss:  0.34453622061064515 | Validation loss:
0.5123455370500083
Epoch:  17 | Training loss:  0.33645757152619127 | Validation loss:
0.5082311892349027
Epoch:  18 | Training loss:  0.32881037987183626 | Validation loss:
0.5042000621273476
Epoch:  19 | Training loss:  0.321583681151182 | Validation loss:
0.5005679663496797
```

```
Epoch:  20 | Training loss:  0.3147228728621788 | Validation loss:
0.497021386122795
Epoch:  21 | Training loss:  0.3082165823797972 | Validation loss:
0.4936039850236934
Epoch:  22 | Training loss:  0.3019943571276996 | Validation loss:
0.4905235736334467
Epoch:  23 | Training loss:  0.29606955986937333 | Validation loss:
0.4875410480189032
Epoch:  24 | Training loss:  0.2904111385121604 | Validation loss:
0.4846823924442425
Epoch:  25 | Training loss:  0.2849964509024652 | Validation loss:
0.4818991116231041
Epoch:  26 | Training loss:  0.27980857460449987 | Validation loss:
0.47928190469190524
Epoch:  27 | Training loss:  0.274833192568072 | Validation loss:
0.4767866366671801
Epoch:  28 | Training loss:  0.2700546002750944 | Validation loss:
0.47432922975645425
Epoch:  29 | Training loss:  0.2654607529512865 | Validation loss:
0.4720240422717186
Epoch:  30 | Training loss:  0.2610410923376829 | Validation loss:
0.4698019883011004
Epoch:  31 | Training loss:  0.25678880939325444 | Validation loss:
0.46774915811630696
Epoch:  32 | Training loss:  0.2526803782822503 | Validation loss:
0.46562795374280724
Epoch:  33 | Training loss:  0.24871673314045295 | Validation loss:
0.46358318104778645
Epoch:  34 | Training loss:  0.2448903391797806 | Validation loss:
0.4616270444727639
Epoch:  35 | Training loss:  0.24119248651650593 | Validation loss:
0.4597532651780613
Epoch:  36 | Training loss:  0.23761694324886434 | Validation loss:
0.4579394563578903
Epoch:  37 | Training loss:  0.23415764812877732 | Validation loss:
0.45611896147078296
Epoch:  38 | Training loss:  0.2308088169231436 | Validation loss:
0.4543985486058348
Epoch:  39 | Training loss:  0.22755623912614162 | Validation loss:
0.45284892270040283
Epoch:  40 | Training loss:  0.2244061859705956 | Validation loss:
0.4513106468764714
Epoch:  41 | Training loss:  0.22135026591827195 | Validation loss:
0.44975104926108656
Epoch:  42 | Training loss:  0.21838274428460258 | Validation loss:
0.448311126287099
Epoch:  43 | Training loss:  0.21550113555180828 | Validation loss:
0.4469140041284044
```

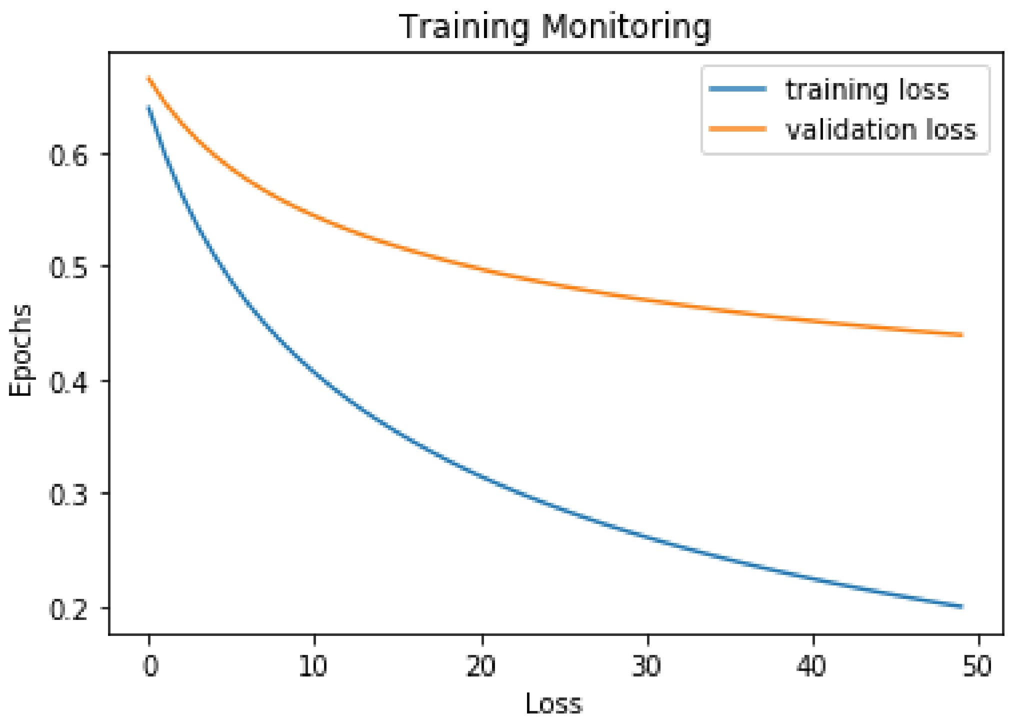
```
Epoch:  44 | Training loss:  0.21269989481822482 | Validation loss:
0.44551189721149426
Epoch:  45 | Training loss:  0.20997585741655642 | Validation loss:
0.44410302704336346
Epoch:  46 | Training loss:  0.2073284359445056 | Validation loss:
0.442719501558827
Epoch:  47 | Training loss:  0.20474833720654634 | Validation loss:
0.44150217075647746
Epoch:  48 | Training loss:  0.20223810637003645 | Validation loss:
0.44026093860706395
Epoch:  49 | Training loss:  0.19979269371503494 | Validation loss:
0.4390808525967976
```

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

[1953]:
```
# fill in your code...

epochs = []
for i in range(len(trl)):
    epochs.append(i)

plt.plot(epochs, trl, label='training loss')
plt.plot(epochs, devl, label='validation loss')
plt.legend()
plt.title('Training Monitoring')
plt.xlabel('Loss')
plt.ylabel('Epochs')
plt.show()
```



The model is about right. The validation loss is declining and convergent at the end of the training process, but it seems that training loss is not quite convergent and it still can be declining.

Compute accuracy, precision, recall and F1-scores:

[1954]:
```
# fill in your code...
preds_te = predict_class(X_te_tfidf, w_tfidf)
print('Accuracy:', accuracy_score(Y_te,preds_te))
print('Precision:', precision_score(Y_te,preds_te))
print('Recall:', recall_score(Y_te,preds_te))
print('F1-Score:', f1_score(Y_te,preds_te))
```

```
Accuracy: 0.855
Precision: 0.8480392156862745
Recall: 0.865
F1-Score: 0.8564356435643564
```

Print top-10 most positive and negative words:

[1955]:
```
# fill in your code...
neg_w_tfidf = sorted(w_tfidf)
print("top-10 words for negative class:")
for i in range(10):
    index = np.where(w_tfidf == neg_w_tfidf[i])[0][0]
    print(vocab_list[index])
```

```
top-10 words for negative class:
bad
worst
boring
supposed
why
unfortunately
stupid
harry
ridiculous
nothing
```

[1956]:
```
# fill in your code...
pos_w_tfidf = sorted(w_tfidf,reverse = True)
print("top-10 words for positive class:")
for i in range(10):
    index = np.where(w_tfidf == pos_w_tfidf[i])[0][0]
    print(vocab_list[index])
```

```
top-10 words for positive class:
great
truman
life
perfectly
hilarious
terrific
perfect
excellent
memorable
world
```

The model perform better after using tf-idf vectors. More adjective words are generated in the results. The words like unfortunately, ridiculous, stupid, ridiculous, perfectly, hilarious… could be picked up in different domains.

3.2.1 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…

I attempt to choose learning rate by multiple 0.1. If the loss is increasing, I will adjust the learning rate by choosing a smaller number. If the loss is gradually decreasing and can be convergence, that would be the best choice for learing rate. After choosing the learning rate, then according to the validation loss increasing or decreasing the regularistion strength 10 times of 0.1. The optimal hyperparameters would give a high precision and recall rate model. The smaller the learning rate, the more training epochs. The regularisation strength can prevent model overfitting, to make sure the model has both good performance on both training set and test set.

3.3 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count	0.84	0.84	0.83
BOW-tfidf	0.84	0.86	0.85

4 Multi-class Logistic Regression

Now you need to train a Multiclass Logistic Regression (MLR) Classifier by extending the Binary model you developed above. You will use the MLR model to perform topic classification on the AG news dataset consisting of three classes:

- Class 1: World
- Class 2: Sports
- Class 3: Business

You need to follow the same process as in Task 1 for data processing and feature extraction by reusing the functions you wrote.

[1957]:
```
# fill in your code...
data_tr = pd.read_csv('./data_topic/train.
 ↪csv',header=None,names=['label','text'])
data_te = pd.read_csv('./data_topic/test.csv',header=None,names=['label','text'])
data_de = pd.read_csv('./data_topic/dev.csv',header=None,names=['label','text'])
```

[1958]:
```
data_tr.head()
```

[1958]:
```
   label                                               text
0      1  Reuters - Venezuelans turned out early\and in ...
1      1  Reuters - South Korean police used water canno...
2      1  Reuters - Thousands of Palestinian\prisoners i...
3      1  AFP - Sporadic gunfire and shelling took place...
4      1  AP - Dozens of Rwandan soldiers flew into Suda...
```

[1959]:
```
# fill in your code...
#train data transformation
X_tr_raw = data_tr['text'].tolist()
Y_tr = np.array(data_tr['label'])
#test data transformatino
X_te_raw = data_te['text'].tolist()
Y_te = np.array(data_te['label'])
#development data transformation
X_de_raw = data_de['text'].tolist()
Y_de = np.array(data_de['label'])
```

[1960]:
```
vocab, df, ngram_counts = get_vocab(X_tr_raw, ngram_range=(1,3), keep_topN=5000,␣
 ↪stop_words=stop_words)
print(len(vocab))
print()
print(list(vocab)[:100])
print()
print(df.most_common()[:10])
```

```
5000

['berlusconi', 'pre', 'empty', 'number', 'barry', 'basketball', 'threatening',
('internet', 'ipo'), 'dug', 'thumb', 'record', ('agency', 'said'), 'eastern',
('nfl', 'players', 'association'), ('beijing', 'reuters', 'china'), ('higher',
'wednesday'), ('two', 'most', 'spectacular'), 'keller', 'bloody', ('lowe',
'cos', 'lt'), 'non', 'impact', ('general', 'motors'), 'life', 'tumbled', 'line',
'ohalete', 'below', 'hard', 'round', 'calif', ('street', 'estimates'),
'sprinters', ('months', 'industry', 'group'), ('iraq', 'national',
'conference'), ('strength', 'agricultural'), ('ninth', 'inning'), ('fullquote',
'gt'), ('public', 'offering', 'registration'), ('baltimore', 'orioles'),
('home', 'improvement'), ('man', 'among', 'sick'), ('election', 'year'),
'police', 'jewish', 'knee', 'happen', 'venus', ('public', 'offering'),
('offering', 'nearly'), 'bush', ('go', 'ahead'), ('agreed', 'disarm'), ('final',
'phelps'), 'burundi', 'guard', 'hostile', 'points', 'programs', 'headed',
('farm', 'equipment', 'makers'), 'looking', 'along', 'stores', ('inc',
'slashed', 'size'), 'created', ('starts', 'rebounded'), 'kenteris', ('about',
'inflation', 'least'), 'puerto', ('cruciate', 'ligament', 'right'), 'together',
('reuters', 'stocks'), ('premier', 'silvio'), 'dorman', ('anticipated',
'initial', 'public'), ('more', 'than', 'congolese'), 'families', ('medals',
'athens', 'olympics'), 'plus', 'west', ('search', 'engine', 'wednesday'),
('cleveland', 'indians'), 'cos', ('john', 'howard'), ('rival', 'province',
'healthcare'), ('shi', 'ite', 'uprising'), ('tournament', 'tuesday'), ('ipo',
'years'), ('monday', 'night'), 'top', ('conspiracy', 'murder'), 'torri',
'spend', 'boxing', ('than', 'teenager', 'free'), ('pressure', 'held'), ('said',
'quarterly'), 'sick', ('meter', 'freestyle', 'semifinals')]

[('reuters', 631), ('said', 432), ('tuesday', 413), ('wednesday', 344), ('new',
325), ('after', 295), ('ap', 275), ('athens', 245), ('monday', 221), ('first',
210)]
```

[1961]:
```
# fill in your code...
X_tr_ngram = get_X_ngram(X_tr_raw,vocab,ngram_range=(1,3),␣
↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',stop_words=stop_words)
X_te_ngram = get_X_ngram(X_te_raw,vocab,ngram_range=(1,3),␣
↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',stop_words=stop_words)
X_de_ngram = get_X_ngram(X_de_raw,vocab,ngram_range=(1,3),␣
↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',stop_words=stop_words)
```

[1962]:
```
X_tr_count = vectorise(X_tr_ngram, vocab)
X_te_count = vectorise(X_te_ngram, vocab)
X_de_count = vectorise(X_de_ngram, vocab)
```

[1963]:
```
idfs = get_idf(X_tr_raw,vocab,df)
X_tr_tfidf = X_tr_count*idfs
X_te_tfidf = X_te_count*idfs
X_de_tfidf = X_de_count*idfs
```

Now you need to change `SGD` to support multiclass datasets. First you need to develop a `softmax` function. It takes as input:

- z: array of real numbers

and returns:

- smax: the softmax of z

[1964]:
```
def softmax(z):
    t = np.exp(z)
    smax = np.exp(z) / np.sum(t, axis=1).reshape(-1,1)
    return smax

```

Then modify `predict_proba` and `predict_class` functions for the multiclass case:

[1965]:
```
def predict_proba(X, weights):

    # fill in your code...
    preds_proba = softmax(np.dot(X,weights.T))
    return preds_proba
```

[1966]:
```
def predict_class(X, weights):

    # fill in your code...
    n = X.shape[0]
    preds_class = np.zeros((n,)).astype(int)
    preds_proba = predict_proba(X, weights)
    for i in range(len(preds_proba)):
        # find the correct class
        pred_class = np.where(preds_proba[i]==np.max(preds_proba[i]))[0][0] + 1
        preds_class[i] = pred_class

    return preds_class
```

Toy example and expected functionality of the functions above:

[1967]:
```
X = np.array([[0.1,0.2],[0.2,0.1],[0.1,-0.2]])
w = np.array([[2,-5],[-5,2]])
```

[1968]:
```
predict_proba(X, w)
```

[1968]:
```
array([[0.33181223, 0.66818777],
       [0.66818777, 0.33181223],
       [0.89090318, 0.10909682]])
```

[1969]:
```
predict_class(X, w)
```

[1969]:
```
array([2, 1, 1])
```

Now you need to compute the categorical cross entropy loss (extending the binary loss to support multiple classes).

[1970]:
```
def categorical_loss(X, Y, weights, num_classes=5, alpha=0.00001):
    preds_proba = predict_proba(X, weights)
    L = np.zeros((num_classes,len(Y)))
    for i in range(len(Y)):
        index = Y[i] - 1
        #compute loss of correct class
        L[index][i] = (-Y[i])*np.log(preds_proba[i][index])
        #regularisation part
        regular = alpha*np.square(np.sum(np.power(weights[index],2)))
        L[index][i] = L[index][i] + regular
    l = np.mean(L)

    return l
```

Finally you need to modify SGD to support the categorical cross entropy loss:

[1971]:
```
def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], num_classes=5, lr=0.01, alpha=0.00001,␣
↪epochs=5, tolerance=0.0001, print_progress=True):

    # fill in your code...
    cur_loss_tr = 2.
    cur_loss_dev = 2.
    training_loss_history = []
    validation_loss_history = []
    m,n = X_tr.shape
    # initialize w with zeros
    weights = np.zeros((num_classes,n))
    for i in range(epochs):
        # randomise order in training Data
        permutation = np.random.permutation(m)
        shuffled_x = X_tr[permutation,:]
        shuffled_y = Y_tr[permutation]
        for j in range(len(Y_tr)):
            #find correct index
            index = shuffled_y[j] - 1
            #compute predcition value
            pred = predict_proba(shuffled_x[j].reshape(1,-1),weights)
            #update weights of correct class
            weights[index] = weights[index] -␣
↪lr*(shuffled_x[j]*(pred[0][index]-1)+2*alpha*weights[index])
        pre_loss_tr = cur_loss_tr
        pre_loss_dev = cur_loss_dev
        cur_loss_tr =␣
↪categorical_loss(shuffled_x,shuffled_y,weights,num_classes,alpha)
        cur_loss_dev = categorical_loss(X_dev,Y_dev,weights,num_classes,alpha)
        training_loss_history.append(cur_loss_tr)
        validation_loss_history.append(cur_loss_dev)
        if print_progress==True:
            print("Epoch: ",i,"| Training loss: ",cur_loss_tr,"| Validation loss:
↪ ",cur_loss_dev)
        if (pre_loss_dev - cur_loss_dev) < tolerance:
            break

    return weights, training_loss_history, validation_loss_history
```

Now you are ready to train and evaluate you MLR following the same steps as in Task 1 for both Count and tfidf features:

[1972]:
```
w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr,
                                             X_dev=X_de_count,
                                             Y_dev=Y_de,
                                             num_classes=3,
                                             lr=0.0001,
                                             alpha=0.001,
                                             epochs=200)
```

```
Epoch:  0 | Training loss:  0.7154662929931689 | Validation loss:
0.7245454264341528
Epoch:  1 | Training loss:  0.7000640189159985 | Validation loss:
0.7171704652129751
Epoch:  2 | Training loss:  0.6860314104375709 | Validation loss:
0.7102427173971237
Epoch:  3 | Training loss:  0.673161153042827 | Validation loss:
0.703704383326883
Epoch:  4 | Training loss:  0.6612669979319432 | Validation loss:
0.6975054887449498
Epoch:  5 | Training loss:  0.6502128248971629 | Validation loss:
0.691608504671137
Epoch:  6 | Training loss:  0.6398596396871865 | Validation loss:
0.6859690335873189
Epoch:  7 | Training loss:  0.6301377824698938 | Validation loss:
0.6805697882290901
Epoch:  8 | Training loss:  0.6209535214729662 | Validation loss:
0.6753794702694033
Epoch:  9 | Training loss:  0.6122527733751929 | Validation loss:
0.6703808879434647
Epoch:  10 | Training loss:  0.6039784813635876 | Validation loss:
0.6655557729039074
Epoch:  11 | Training loss:  0.5960959089628614 | Validation loss:
0.6608914785788164
Epoch:  12 | Training loss:  0.5885675496948574 | Validation loss:
```

```
0.6563760352673605
Epoch:  13 | Training loss:  0.5813641487476977 | Validation loss:
0.6519987527954603
Epoch:  14 | Training loss:  0.5744573830701827 | Validation loss:
0.6477503995566956
Epoch:  15 | Training loss:  0.5678290822308081 | Validation loss:
0.643624723704633
Epoch:  16 | Training loss:  0.5614599822066387 | Validation loss:
0.639614987524299
Epoch:  17 | Training loss:  0.5553327543038247 | Validation loss:
0.6357151778427089
Epoch:  18 | Training loss:  0.5494323990243316 | Validation loss:
0.6319196940640678
Epoch:  19 | Training loss:  0.5437470206478049 | Validation loss:
0.6282250136430888
Epoch:  20 | Training loss:  0.5382574381950597 | Validation loss:
0.6246225269864959
Epoch:  21 | Training loss:  0.5329614172167866 | Validation loss:
0.6211135862732122
Epoch:  22 | Training loss:  0.5278419637429437 | Validation loss:
0.6176909049287188
Epoch:  23 | Training loss:  0.5228965389544675 | Validation loss:
0.6143547038578274
Epoch:  24 | Training loss:  0.5181109573647261 | Validation loss:
0.6110984247806671
Epoch:  25 | Training loss:  0.5134776998787274 | Validation loss:
0.6079197397034691
Epoch:  26 | Training loss:  0.508993868856459 | Validation loss:
0.6048182059899738
Epoch:  27 | Training loss:  0.5046503224312388 | Validation loss:
0.6017902888997582
Epoch:  28 | Training loss:  0.5004382211323621 | Validation loss:
0.5988320205979315
Epoch:  29 | Training loss:  0.4963547276542038 | Validation loss:
0.5959427662907094
Epoch:  30 | Training loss:  0.4923931636791587 | Validation loss:
0.5931202144411349
Epoch:  31 | Training loss:  0.4885484525096889 | Validation loss:
0.5903618774427409
Epoch:  32 | Training loss:  0.4848160215555879 | Validation loss:
0.5876663645191045
Epoch:  33 | Training loss:  0.48119218287102045 | Validation loss:
0.585032757183362
Epoch:  34 | Training loss:  0.47767248298306747 | Validation loss:
0.582458646077208
Epoch:  35 | Training loss:  0.47425415565236045 | Validation loss:
0.5799435878952672
Epoch:  36 | Training loss:  0.47093124752276594 | Validation loss:
```

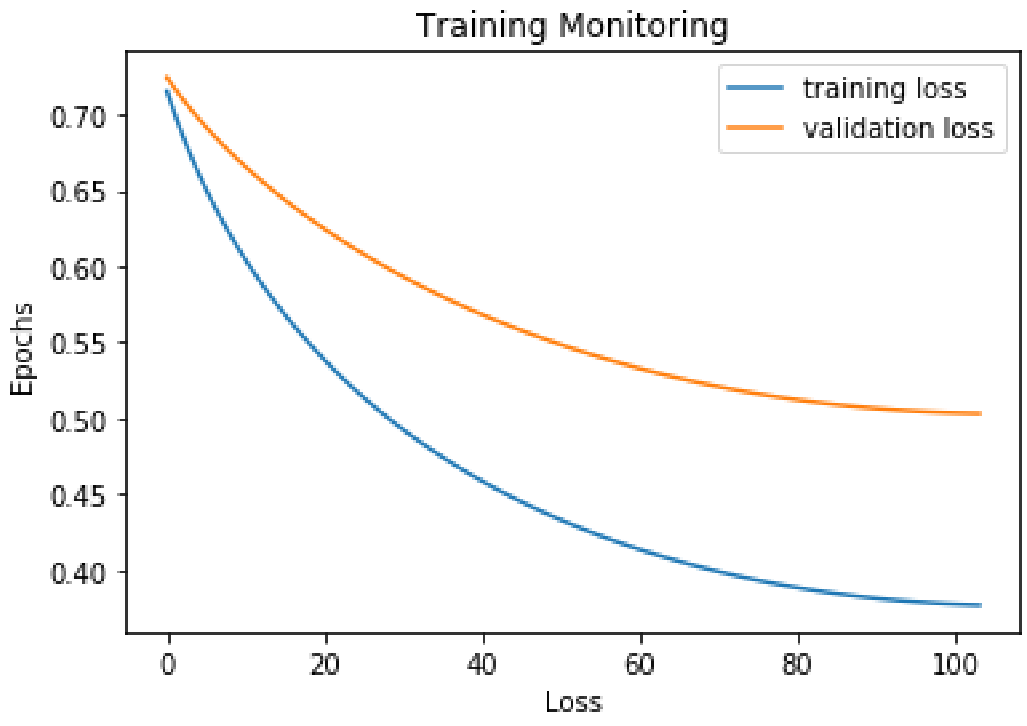
```
0.5774843444149014
Epoch:  37 | Training loss:  0.4676998561013921 | Validation loss:
0.5750795289988899
Epoch:  38 | Training loss:  0.46455959356675824 | Validation loss:
0.5727297530779902
Epoch:  39 | Training loss:  0.46150734994051357 | Validation loss:
0.5704339567130327
Epoch:  40 | Training loss:  0.4585381889108778 | Validation loss:
0.5681891738492562
Epoch:  41 | Training loss:  0.45565162154324523 | Validation loss:
0.5659963125497824
Epoch:  42 | Training loss:  0.4528434328185934 | Validation loss:
0.5638528603037432
Epoch:  43 | Training loss:  0.4501117457909412 | Validation loss:
0.5617583682096455
Epoch:  44 | Training loss:  0.44745191673542917 | Validation loss:
0.5597100023457503
Epoch:  45 | Training loss:  0.4448628628790316 | Validation loss:
0.5577077287506088
Epoch:  46 | Training loss:  0.44234500386705644 | Validation loss:
0.555752749435282
Epoch:  47 | Training loss:  0.43989526368481946 | Validation loss:
0.5538430441919991
Epoch:  48 | Training loss:  0.43750986517542567 | Validation loss:
0.551976567451553
Epoch:  49 | Training loss:  0.4351899806381919 | Validation loss:
0.5501547740028865
Epoch:  50 | Training loss:  0.4329312337840261 | Validation loss:
0.548374777989108
Epoch:  51 | Training loss:  0.43073402847136766 | Validation loss:
0.5466375496891391
Epoch:  52 | Training loss:  0.42859347123023606 | Validation loss:
0.5449396987636698
Epoch:  53 | Training loss:  0.42651290555607 | Validation loss:
0.5432843041720585
Epoch:  54 | Training loss:  0.42448857659387285 | Validation loss:
0.5416689795323466
Epoch:  55 | Training loss:  0.4225188394582299 | Validation loss:
0.5400927989612486
Epoch:  56 | Training loss:  0.42060239270310656 | Validation loss:
0.53855542010432
Epoch:  57 | Training loss:  0.41873756525961364 | Validation loss:
0.537055540229253
Epoch:  58 | Training loss:  0.4169255872938257 | Validation loss:
0.5355945966588037
Epoch:  59 | Training loss:  0.4151639851787944 | Validation loss:
0.5341709848498538
Epoch:  60 | Training loss:  0.4134506362148771 | Validation loss:
```

```
0.532783595356858
Epoch:  61 | Training loss:  0.4117862154622929 | Validation loss:
0.5314331087325589
Epoch:  62 | Training loss:  0.4101691065915576 | Validation loss:
0.5301185300427538
Epoch:  63 | Training loss:  0.4085981202977738 | Validation loss:
0.5288391803066764
Epoch:  64 | Training loss:  0.4070719038975894 | Validation loss:
0.5275943660664983
Epoch:  65 | Training loss:  0.40559178597430207 | Validation loss:
0.5263853466235155
Epoch:  66 | Training loss:  0.404155848845459 | Validation loss:
0.525210889108414
Epoch:  67 | Training loss:  0.4027617448903528 | Validation loss:
0.5240691835444147
Epoch:  68 | Training loss:  0.4014100349486307 | Validation loss:
0.5229610021084522
Epoch:  69 | Training loss:  0.40010051410836506 | Validation loss:
0.5218865341476024
Epoch:  70 | Training loss:  0.3988321690131842 | Validation loss:
0.5208450410077261
Epoch:  71 | Training loss:  0.3976045282106389 | Validation loss:
0.5198363464686405
Epoch:  72 | Training loss:  0.3964163940547592 | Validation loss:
0.5188596072011868
Epoch:  73 | Training loss:  0.3952670107805208 | Validation loss:
0.5179145508611708
Epoch:  74 | Training loss:  0.3941558969221967 | Validation loss:
0.5170007680236043
Epoch:  75 | Training loss:  0.39308355490160135 | Validation loss:
0.5161190144832563
Epoch:  76 | Training loss:  0.3920481132483873 | Validation loss:
0.5152678736960146
Epoch:  77 | Training loss:  0.3910502950107839 | Validation loss:
0.514448129218928
Epoch:  78 | Training loss:  0.39008854292299006 | Validation loss:
0.5136585033139611
Epoch:  79 | Training loss:  0.38916221373607174 | Validation loss:
0.5128987085813045
Epoch:  80 | Training loss:  0.3882723474202656 | Validation loss:
0.5121697481718958
Epoch:  81 | Training loss:  0.38741749655774815 | Validation loss:
0.511470522133858
Epoch:  82 | Training loss:  0.38659673556379853 | Validation loss:
0.5108002401871892
Epoch:  83 | Training loss:  0.3858112089591565 | Validation loss:
0.5101602434498345
Epoch:  84 | Training loss:  0.38505931326516224 | Validation loss:
```

```
0.5095492939909971
Epoch:  85 | Training loss:  0.38434096014089936 | Validation loss:
0.5089671710171573
Epoch:  86 | Training loss:  0.38365550419015076 | Validation loss:
0.508413624529714
Epoch:  87 | Training loss:  0.3830035274837779 | Validation loss:
0.5078892273632567
Epoch:  88 | Training loss:  0.38238373341475607 | Validation loss:
0.5073929407913511
Epoch:  89 | Training loss:  0.3817962162237479 | Validation loss:
0.5069250661972449
Epoch:  90 | Training loss:  0.38123992868232803 | Validation loss:
0.5064846477312117
Epoch:  91 | Training loss:  0.38071561607973725 | Validation loss:
0.5060724779637025
Epoch:  92 | Training loss:  0.38022255906222785 | Validation loss:
0.5056880542597534
Epoch:  93 | Training loss:  0.3797603562866898 | Validation loss:
0.5053310039003108
Epoch:  94 | Training loss:  0.37932939835214396 | Validation loss:
0.5050018931738822
Epoch:  95 | Training loss:  0.3789287353276426 | Validation loss:
0.5046999662093616
Epoch:  96 | Training loss:  0.3785591260855154 | Validation loss:
0.5044258589551871
Epoch:  97 | Training loss:  0.37821894943716694 | Validation loss:
0.504178276883653
Epoch:  98 | Training loss:  0.37790796732416704 | Validation loss:
0.5039570672079764
Epoch:  99 | Training loss:  0.37762687442593296 | Validation loss:
0.5037629330356164
Epoch:  100 | Training loss:  0.3773750529996957 | Validation loss:
0.5035953426988301
Epoch:  101 | Training loss:  0.3771531213188183 | Validation loss:
0.5034550466256815
Epoch:  102 | Training loss:  0.37695955748188986 | Validation loss:
0.5033407007298886
Epoch:  103 | Training loss:  0.3767951522417175 | Validation loss:
0.5032530275795278
```

Plot training and validation process and explain if your model overfit, underfit or is about right:

[1973]:
```
# fill in your code...
epochs = []
for i in range(len(loss_tr_count)):
    epochs.append(i)
plt.plot(epochs, loss_tr_count, label='training loss')
plt.plot(epochs, dev_loss_count, label='validation loss')
plt.legend()
plt.title('Training Monitoring')
plt.xlabel('Loss')
plt.ylabel('Epochs')
plt.show()
```



The model is about right. Because both training loss and validation loss are decling and can be convergence when the training stops.

Compute accuracy, precision, recall and F1-scores:

[1974]:
```
preds_te = predict_class(X_te_count, w_count)
print('Accuracy:', accuracy_score(Y_te,preds_te))
print('Precision:', precision_score(Y_te,preds_te,average='macro'))
print('Recall:', recall_score(Y_te,preds_te,average='macro'))
print('F1-Score:', f1_score(Y_te,preds_te,average='macro'))
```

```
Accuracy: 0.8277777777777777
Precision: 0.8302519609617837
Recall: 0.8277777777777778
F1-Score: 0.8266459551840306
```

Print the top-10 words for each class respectively.

[1975]:
```
# fill in your code...
list_vocab = list(vocab)
w_class_1 = sorted(w_count[0],reverse = True)
print("top-10 words for class 1:")
for i in range(10):
    index = np.where(w_count[0] == w_class_1[i])[0][0]
    print(list_vocab[index])
print("\n")
w_class_2 = sorted(w_count[1],reverse = True)
print("top-10 words for class 2:")
for i in range(10):
    index = np.where(w_count[1] == w_class_2[i])[0][0]
    print(list_vocab[index])
print("\n")
w_class_3 = sorted(w_count[2],reverse = True)
print("top-10 words for class 3:")
for i in range(10):
    index = np.where(w_count[2] == w_class_3[i])[0][0]
    print(list_vocab[index])
```

```
top-10 words for class 1:
said
reuters
tuesday
ap
monday
after
wednesday
new
president
afp


top-10 words for class 2:
athens
ap
olympic
reuters
first
tuesday
after
team
two
wednesday


top-10 words for class 3:
```

```
reuters
said
new
tuesday
company
oil
wednesday
prices
inc
after
```

4.0.1 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?

Explain here…

I attempt to choose learning rate by multiple 0.1. If the loss is increasing, I will adjust the learning rate by choosing a smaller number. If the loss is gradually decreasing and can be convergence, that would be the best choice for learing rate. This is quite similar as hyperparameter choice in binary classfication. After choosing the learning rate, then according to the validation loss increasing or decreasing the regularistion strength 10 times of 0.1. The optimal hyperparameters would give a high precision and recall rate model. The smaller the learning rate, the more training epochs. The regularisation strength can prevent model overfitting, to make sure the model has both good performance on both training set and test set.

4.0.2 Now evaluate BOW-tfidf…

[1848]:
```
w_tfidf, trl, devl = SGD(X_tr_tfidf, Y_tr,
                     X_dev=X_de_tfidf,
                     Y_dev=Y_de,
                     num_classes=3,
                     lr=0.0001,
                     alpha=0.001,
                     epochs=200)
```

```
Epoch:  0 | Training loss:  0.6931591223099036 | Validation loss:
0.7136912272046976
Epoch:  1 | Training loss:  0.6606735990177162 | Validation loss:
0.696990472457131
Epoch:  2 | Training loss:  0.6329067269567162 | Validation loss:
0.6818435154216601
Epoch:  3 | Training loss:  0.6085114958256774 | Validation loss:
0.6679012552799062
Epoch:  4 | Training loss:  0.5866870150652593 | Validation loss:
0.6549525724020693
Epoch:  5 | Training loss:  0.5669235689344793 | Validation loss:
0.6428443096078473
Epoch:  6 | Training loss:  0.5488897471682153 | Validation loss:
```

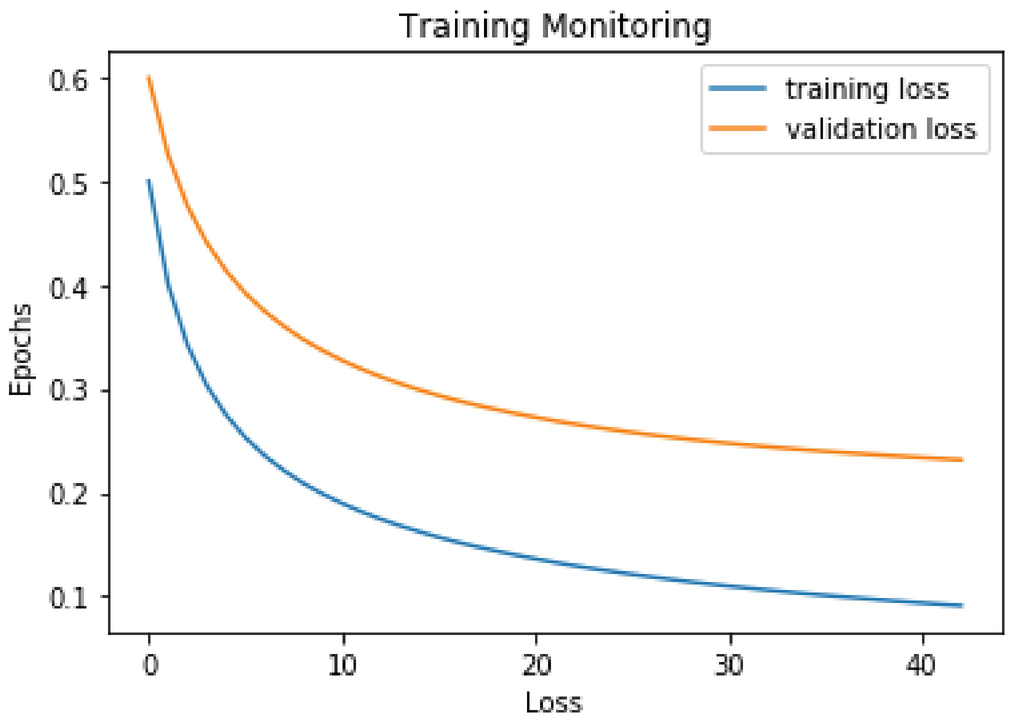
```
0.6314787767470467
Epoch:  7 | Training loss:  0.5323369312443285 | Validation loss:
0.6207695585408777
Epoch:  8 | Training loss:  0.5170598118180788 | Validation loss:
0.6106464212945907
Epoch:  9 | Training loss:  0.5029127725755828 | Validation loss:
0.6010626146011896
Epoch:  10 | Training loss:  0.48977105274379357 | Validation loss:
0.591975432075103
Epoch:  11 | Training loss:  0.4775191482121849 | Validation loss:
0.5833380300369468
Epoch:  12 | Training loss:  0.4660695317754694 | Validation loss:
0.5751189612609197
Epoch:  13 | Training loss:  0.4553408455325511 | Validation loss:
0.5672828694798321
Epoch:  14 | Training loss:  0.4452638194633515 | Validation loss:
0.5598031632942863
Epoch:  15 | Training loss:  0.43578019454284156 | Validation loss:
0.5526540483242214
Epoch:  16 | Training loss:  0.4268388340422906 | Validation loss:
0.5458152688099328
Epoch:  17 | Training loss:  0.41839093190787097 | Validation loss:
0.5392635915454445
Epoch:  18 | Training loss:  0.410396807922975 | Validation loss:
0.5329826016206989
Epoch:  19 | Training loss:  0.40281823268078154 | Validation loss:
0.526954217564874
Epoch:  20 | Training loss:  0.3956190019050496 | Validation loss:
0.5211600835231749
Epoch:  21 | Training loss:  0.38877792369682224 | Validation loss:
0.5155901883058642
Epoch:  22 | Training loss:  0.3822593772863226 | Validation loss:
0.5102278100736356
Epoch:  23 | Training loss:  0.3760437656504773 | Validation loss:
0.5050621904487649
Epoch:  24 | Training loss:  0.37010606126086953 | Validation loss:
0.5000791585127143
Epoch:  25 | Training loss:  0.36442802667607177 | Validation loss:
0.4952705467178687
Epoch:  26 | Training loss:  0.3589924711542398 | Validation loss:
0.4906266182851119
Epoch:  27 | Training loss:  0.3537853938675508 | Validation loss:
0.4861413621483185
Epoch:  28 | Training loss:  0.3487897303924128 | Validation loss:
0.4818027703609131
Epoch:  29 | Training loss:  0.34399090126820464 | Validation loss:
0.4776031034270167
Epoch:  30 | Training loss:  0.3393807217164435 | Validation loss:
```

```
0.4735397867337919
Epoch:  31 | Training loss:  0.33494259698532897 | Validation loss:
0.4695999034945045
Epoch:  32 | Training loss:  0.3306708892579585 | Validation loss:
0.4657825853763357
Epoch:  33 | Training loss:  0.32655323493306915 | Validation loss:
0.46207960355016603
Epoch:  34 | Training loss:  0.3225801532749901 | Validation loss:
0.45848483971908205
Epoch:  35 | Training loss:  0.31874234081487296 | Validation loss:
0.45499179843370124
Epoch:  36 | Training loss:  0.31503757901489804 | Validation loss:
0.451601548267084
Epoch:  37 | Training loss:  0.31145365069495634 | Validation loss:
0.4483036101643574
Epoch:  38 | Training loss:  0.30798350459342344 | Validation loss:
0.4450938218999411
Epoch:  39 | Training loss:  0.3046266612727863 | Validation loss:
0.44197386645615944
Epoch:  40 | Training loss:  0.3013719013176531 | Validation loss:
0.43893401950380867
Epoch:  41 | Training loss:  0.29821778292984263 | Validation loss:
0.43597492071247856
Epoch:  42 | Training loss:  0.29515843016573007 | Validation loss:
0.4330917480355047
Epoch:  43 | Training loss:  0.2921906497948561 | Validation loss:
0.43028329370887225
Epoch:  44 | Training loss:  0.28930564726729163 | Validation loss:
0.427542164879224
Epoch:  45 | Training loss:  0.28650326076649985 | Validation loss:
0.4248695875119241
Epoch:  46 | Training loss:  0.28377782161653753 | Validation loss:
0.42226019732186953
Epoch:  47 | Training loss:  0.28112739195523784 | Validation loss:
0.41971394758457375
Epoch:  48 | Training loss:  0.27854551044897186 | Validation loss:
0.41722462930911214
Epoch:  49 | Training loss:  0.2760360208090077 | Validation loss:
0.41479833457272713
Epoch:  50 | Training loss:  0.2735884658580383 | Validation loss:
0.4124235112987171
Epoch:  51 | Training loss:  0.2712028311370672 | Validation loss:
0.41010113048642466
Epoch:  52 | Training loss:  0.2688765727587074 | Validation loss:
0.4078304208490512
Epoch:  53 | Training loss:  0.26660782040124625 | Validation loss:
0.40560946228865724
Epoch:  54 | Training loss:  0.2643928629796724 | Validation loss:
```

```
0.4034349864522952
Epoch:  55 | Training loss:  0.2622326783953159 | Validation loss:
0.401309045087897
Epoch:  56 | Training loss:  0.2601222084466941 | Validation loss:
0.3992269342345321
Epoch:  57 | Training loss:  0.25806161657408566 | Validation loss:
0.39718939871777054
Epoch:  58 | Training loss:  0.2560471352162297 | Validation loss:
0.3951923753746151
Epoch:  59 | Training loss:  0.2540769724309686 | Validation loss:
0.39323425497464926
Epoch:  60 | Training loss:  0.25215020567927704 | Validation loss:
0.3913160183434733
Epoch:  61 | Training loss:  0.2502664644911616 | Validation loss:
0.38943607170640504
Epoch:  62 | Training loss:  0.24842283066104687 | Validation loss:
0.38759306444985553
Epoch:  63 | Training loss:  0.24661730191991912 | Validation loss:
0.3857840658898915
Epoch:  64 | Training loss:  0.2448483370552017 | Validation loss:
0.38400848135497995
Epoch:  65 | Training loss:  0.24311696404619582 | Validation loss:
0.382268012713818
Epoch:  66 | Training loss:  0.24141936775306988 | Validation loss:
0.38055829619903186
Epoch:  67 | Training loss:  0.23975525053522925 | Validation loss:
0.37887910107941614
Epoch:  68 | Training loss:  0.23812406654350007 | Validation loss:
0.3772305702129952
Epoch:  69 | Training loss:  0.23652467369952515 | Validation loss:
0.3756117043845273
Epoch:  70 | Training loss:  0.2349569329269498 | Validation loss:
0.37402272977669454
Epoch:  71 | Training loss:  0.23341762812186628 | Validation loss:
0.37246038229341594
Epoch:  72 | Training loss:  0.2319080117498566 | Validation loss:
0.3709260795736667
Epoch:  73 | Training loss:  0.2304264377728021 | Validation loss:
0.36941833963627907
Epoch:  74 | Training loss:  0.22897046368957233 | Validation loss:
0.36793415850742184
Epoch:  75 | Training loss:  0.2275414724828749 | Validation loss:
0.3664762883827339
Epoch:  76 | Training loss:  0.2261368610474703 | Validation loss:
0.3650408333702568
Epoch:  77 | Training loss:  0.2247570338856005 | Validation loss:
0.3636295705563022
Epoch:  78 | Training loss:  0.22340116952538439 | Validation loss:
```

```
0.36224139694807245
Epoch:  79 | Training loss:  0.22206881234137996 | Validation loss:
0.36087568295590977
Epoch:  80 | Training loss:  0.22075835036751307 | Validation loss:
0.3595312937587639
Epoch:  81 | Training loss:  0.21946959954074902 | Validation loss:
0.3582073214522699
Epoch:  82 | Training loss:  0.21820330954762118 | Validation loss:
0.3569062577532071
Epoch:  83 | Training loss:  0.21695716973720983 | Validation loss:
0.35562406475184055
Epoch:  84 | Training loss:  0.21573083853984562 | Validation loss:
0.3543612187057058
Epoch:  85 | Training loss:  0.21452496098817922 | Validation loss:
0.35311868711753575
Epoch:  86 | Training loss:  0.21333763552278565 | Validation loss:
0.35189441206374467
Epoch:  87 | Training loss:  0.21216924563601378 | Validation loss:
0.35068901000127867
Epoch:  88 | Training loss:  0.2110179673560825 | Validation loss:
0.3495000024492001
Epoch:  89 | Training loss:  0.20988525743091146 | Validation loss:
0.3483296366944608
Epoch:  90 | Training loss:  0.208768367975384 | Validation loss:
0.3471743073695379
Epoch:  91 | Training loss:  0.20766940751592675 | Validation loss:
0.34603733773577783
Epoch:  92 | Training loss:  0.20658562189752022 | Validation loss:
0.344915091917045
Epoch:  93 | Training loss:  0.2055191106994472 | Validation loss:
0.34381077257787196
Epoch:  94 | Training loss:  0.20446682392608104 | Validation loss:
0.34271978427916816
Epoch:  95 | Training loss:  0.2034311339493938 | Validation loss:
0.3416464516756788
Epoch:  96 | Training loss:  0.202409373480102 | Validation loss:
0.34058663709020515
Epoch:  97 | Training loss:  0.20140200847697612 | Validation loss:
0.3395414282605117
Epoch:  98 | Training loss:  0.20040841196198264 | Validation loss:
0.3385099826537525
Epoch:  99 | Training loss:  0.19942873306723982 | Validation loss:
0.3374925825896531
Epoch:  100 | Training loss:  0.19846260991223297 | Validation loss:
0.33648885912011534
Epoch:  101 | Training loss:  0.1975095254179566 | Validation loss:
0.3354982714465582
```

[1845]:
```
epochs = []
for i in range(len(trl)):
    epochs.append(i)
plt.plot(epochs, trl, label='training loss')
plt.plot(epochs, devl, label='validation loss')
plt.legend()
plt.title('Training Monitoring')
plt.xlabel('Loss')
plt.ylabel('Epochs')
plt.show()
```



[1846]:
```
# fill in your code...
preds_te = predict_class(X_te_tfidf, w_tfidf)
print('Accuracy:', accuracy_score(Y_te,preds_te))
print('Precision:', precision_score(Y_te,preds_te,average='macro'))
print('Recall:', recall_score(Y_te,preds_te,average='macro'))
print('F1-Score:', f1_score(Y_te,preds_te,average='macro'))
```

```
Accuracy: 0.8711111111111111
Precision: 0.8754389667572947
Recall: 0.8711111111111111
F1-Score: 0.8709990762440044
```

Print the top-10 words for each class respectively.

[1847]:
```python
# fill in your code...
w_class_1 = sorted(w_tfidf[0],reverse = True)
print("top-10 words for class 1:")
for i in range(10):
    index = np.where(w_tfidf[0] == w_class_1[i])[0][0]
    print(list_vocab[index])
print("\n")
w_class_2 = sorted(w_tfidf[1],reverse = True)
print("top-10 words for class 2:")
for i in range(10):
    index = np.where(w_tfidf[1] == w_class_2[i])[0][0]
    print(list_vocab[index])
print("\n")
w_class_3 = sorted(w_tfidf[2],reverse = True)
print("top-10 words for class 3:")
for i in range(10):
    index = np.where(w_tfidf[2] == w_class_3[i])[0][0]
    print(list_vocab[index])
```

```
top-10 words for class 1:
said
tuesday
afp
greece
monday
ap
('athens', 'greece')
oil
athens
new


top-10 words for class 2:
athens
ap
olympic
team
first
greece
after
olympics
quot
games


top-10 words for class 3:
oil
```

```
company
new
prices
said
reuters
more
after
google
inc
```

4.1 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count	0.84	0.83	0.83
BOW-tfidf	0.88	0.87	0.87