assignment1

February 27, 2021

1 [COM6513] 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 reviews, i.e. positive or negative (binary classification).
- **Task 2:** topic classification, to predict whether a news article is about International issues, Sports or Business (multi-class 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 (**3 marks**); (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 (**3 marks**); (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). (**3 marks**) **Tip: you should merge the two representations**

- Binary Logistic Regression (LR) classifiers for Task 1 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)

- Multiclass Logistic Regression classifiers for Task 2 that will be able to accurately classify news articles 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 for Task 1 (**3 marks**)
  - Minimise the Categorical Cross-entropy loss function for Task 2 (**3 marks**)
  - Use L2 regularisation (**2 marks**)
  - 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. (**5 marks; 2.5 for each task**). **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. (**2 marks**).
- 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? (**3 marks; 1.5 for each task**)
- 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 (**5 marks**).
- Provide efficient solutions by using Numpy arrays when possible. Executing the whole notebook with your code should not take more than 10 minutes on a any standard computer (e.g. Intel Core i5 CPU, 8 or 16GB RAM) excluding hyperparameter tuning runs. You can find tips in Intro to Python for NLP (**2 marks**).

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, 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` 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, 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.. You should mention if you've used Windows to write and test your code because we mostly use Unix based machines for marking (e.g. Ubuntu, MacOS).

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 40. It is worth 40% of your final grade in the module.

The deadline for this assignment is **23:59 on Fri, 25 Feb 2021** and it needs to be submitted via Blackboard. Standard departmental penalties for lateness will be applied. We use a range of strategies to **detect unfair means**, including Turnitin which helps detect plagiarism. Use of unfair means would result in getting a failing grade.

[88]:
```python
import pandas as pd
import numpy as np
from collections import Counter
import re
import matplotlib.pyplot as plt
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score
import random

from tqdm import tqdm # To show the progress
plt.style.use("ggplot")
# fixing random seed for reproducibility
random.seed(123)
np.random.seed(123)
```

1.1 Task 1: Binary classification

1.2 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).

[2]:
```
sentiment_train_df = pd.read_csv('./data_sentiment/train.csv',
                                 names = ['Sentence', 'Sentiment'])
sentiment_dev_df = pd.read_csv('./data_sentiment/dev.csv',
                               names = ['Sentence', 'Sentiment'])
sentiment_test_df = pd.read_csv('./data_sentiment/test.csv',
                                names = ['Sentence', 'Sentiment'])
```

[3]:
```
sentiment_train_df.shape
```

[3]:
```
(1400, 2)
```

[4]:
```
frames = [sentiment_train_df, sentiment_dev_df, sentiment_test_df]
sentiment_whole_doc_pd = pd.concat(frames)
```

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

[5]:
```
sentiment_train_df.head()
```

[5]:
```
                                            Sentence  Sentiment
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:

[6]:
```
sentiment_train_np = sentiment_train_df.values
sentiment_test_np = sentiment_test_df.values
sentiment_dev_np = sentiment_dev_df.values
```

2 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).

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 (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.

[7]:
```
stop_words = ['a','in','on','at','and','or',
              'to', 'the', 'of', 'an', 'by',
              'as', 'is', 'was', 'were', 'been', 'be',
              'are','for', 'this', 'that', 'these', 'those', 'you', 'i',
             'it', 'he', 'she', 'we', 'they' 'will', 'have', 'has',
              'do', 'did', 'can', 'could', 'who', 'which', 'what',
             'his', 'her', 'they', 'them', 'from', 'with', 'its']
```

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. - `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.

[8]:
```
def extract_ngrams(x_raw, ngram_range=(1,3), token_pattern=r'',
                   stop_words=[], vocab=set(), char_ngrams=False):
    # token_pattern="[a-z|\']+"  If char_ngrams=False
    # token_pattern = "[^0-9A-Za-z\u4e00-\u9fa5]"  If char_ngrams=True
    if char_ngrams == True:
        c = ""
        x = x_raw.split()
        x = list(filter(lambda w:w not in stop_words, x)) # remove stopwords
        x = c.join(x)
        x = re.sub(token_pattern,'', x) # Keep letters
    else:
        x = re.compile(token_pattern).findall(x_raw) # only keep word in data
        x = list(filter(lambda w:w not in stop_words, x)) # remove stopwords
    result = slide_word(x, ngram_range) # use slide_word to get ngrams

    if vocab:
        result = list(filter(lambda d: len(set(d).intersection(vocab)) > 0,␣
 ↪tuple(result)))
        # get ngram which contains word in vocab

    return result

def slide_word(text, ngram_range):
    result = list()
    for n in range(ngram_range[0], ngram_range[1]+1):
        for i in range(len(text)):
            word = text[i:i + n]
            if len(word) < n:
                break
            result.append(tuple(word))
    return result
```

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:

[9]:
```
extract_ngrams("movie",
               ngram_range=(2,4),
               token_pattern = "[^0-9A-Za-z\u4e00-\u9fa5]" ,
               stop_words=[],
               char_ngrams=True)
```

[9]:
```
[('m', 'o'),
 ('o', 'v'),
 ('v', 'i'),
 ('i', 'e'),
 ('m', 'o', 'v'),
 ('o', 'v', 'i'),
 ('v', 'i', 'e'),
 ('m', 'o', 'v', 'i'),
 ('o', 'v', 'i', 'e')]
```

2.1.2 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.

[10]:
```
def get_vocab(X_raw, ngram_range=(1,3), token_pattern=r'',
              min_df=0, keep_topN=0,
              stop_words=[],char_ngrams=False):

    x_array = []
    df_all = {}
    n = len(X_raw)
    for comment in tqdm(X_raw):
        x = extract_ngrams(comment, ngram_range=ngram_range,␣
↪token_pattern=token_pattern,
                            stop_words=stop_words, char_ngrams=char_ngrams)
        # get ngrams for each comment
        x_array.extend(x)

        x_set = set(x)
        for word in x_set:
            df_all[word] = df_all.get(word, 0) + 1
            # Calculate how many documents contains ngrams

    df_min = {k:v for k,v in df_all.items() if v > min_df}
    # Get the df_min dictionary that contains those ngram which appear more␣
↪than min_df times

    ngram_counts_all = Counter(x_array)
    ngram_counts_sorted = sorted(ngram_counts_all.items(),key=lambda item:
↪item[1],reverse=True)
    # Sort ngrams to help calculate topN

    if keep_topN == 0:
        ngram_counts_top = list(dict(ngram_counts_sorted).keys())#return all
    elif keep_topN < 1:
        ngram_counts_top = list(dict(ngram_counts_sorted).keys())[:
↪int(len(ngram_counts_sorted)*keep_topN)]
        #return topN by percentages
    else:
        ngram_counts_top =  list(dict(ngram_counts_sorted).keys())[:keep_topN]
        #return first keep_topN ngram

    vocab = set(df_min.keys()).intersection(set(ngram_counts_top))
    #select the intersection of min_df and keep_topN
    df ={}
    ngram_counts ={}
    for word in vocab:
        df[word] = df_all[word]
        ngram_counts[word] = ngram_counts_all[word]


    return vocab, df, ngram_counts
```

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

[11]:
```
sentiment_train_vocab, sentiment_train_df, sentiment_train_ngram_counts =␣
 ↪get_vocab(sentiment_train_np[:,0],
                                                                               ␣
 ↪    token_pattern="[a-z|\']+",
                                                                               ␣
 ↪    stop_words=stop_words, min_df=4)
```

```
100%|| 1400/1400 [00:02<00:00, 681.76it/s]
```

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

[12]:
```
id2word = dict(enumerate(sentiment_train_vocab))
word2id = dict(zip(sentiment_train_vocab, range(len(sentiment_train_vocab))))
```

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

[13]:
```
def whole_extracy_ngrams(documents, char_ngrams = False, token_pattern = r''):
    result = {}
    i = 0
    for comment in tqdm(documents):
        result[i] = extract_ngrams(comment, token_pattern=token_pattern,␣
 ↪stop_words=stop_words, char_ngrams=char_ngrams)
        i += 1
    return result
```

[14]:
```
sentiment_train_extract_ngrams = whole_extracy_ngrams(sentiment_train_np[:,0],␣
 ↪token_pattern="[a-z|\']+")
sentiment_test_extrace_ngrams = whole_extracy_ngrams(sentiment_test_np[:,0],␣
 ↪token_pattern="[a-z|\']+")
sentiment_dev_extrace_ngrams = whole_extracy_ngrams(sentiment_dev_np[:,0],␣
 ↪token_pattern="[a-z|\']+")
```

```
100%|| 1400/1400 [00:01<00:00, 1071.56it/s]
100%|| 400/400 [00:00<00:00, 1055.44it/s]
100%|| 200/200 [00:00<00:00, 1083.97it/s]
```

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.

[15]:
```
def vectorise(X_ngram, vocab):
    N = len(X_ngram)
    size_vocab = len(vocab)
    X_vec = np.zeros(shape=(N, size_vocab))
    for index, n_grams in tqdm(X_ngram.items()):
        frequency = Counter(n_grams)
        j = 0
        for word in vocab:
            X_vec[index,j] = frequency.get(word,0)
            j += 1
    return X_vec
```

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

Count vectors

[16]:
```
sentiment_train_tfmatrix = vectorise(sentiment_train_extract_ngrams,␣
 ↪sentiment_train_vocab)
sentiment_test_tfmatrix = vectorise(sentiment_test_extrace_ngrams,␣
 ↪sentiment_train_vocab)
sentiment_dev_tfmatrix = vectorise(sentiment_dev_extrace_ngrams,␣
 ↪sentiment_train_vocab)
```

```
100%|| 1400/1400 [00:10<00:00, 131.09it/s]
100%|| 400/400 [00:03<00:00, 128.18it/s]
100%|| 200/200 [00:01<00:00, 141.82it/s]
```

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

[17]:
```
sentiment_idf = []
number_documents = len(sentiment_train_np)
for word in sentiment_train_vocab:
    df = sentiment_train_df.get(word)
    sentiment_idf.append(np.log(number_documents/df))
```

Then transform your count vectors to tf.idf vectors:

[18]:
```
tfidf_train_matrix = sentiment_train_tfmatrix * np.array(sentiment_idf)
tfidf_dev_matrix = sentiment_dev_tfmatrix * np.array(sentiment_idf)
```

```
tfidf_test_matrix = sentiment_test_tfmatrix * np.array(sentiment_idf)
```

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

[19]:
```
def sigmoid(z):
    sig = 1 / (1 + np.exp(-z))
    return sig
```

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

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

and returns:

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

[20]:
```
def predict_proba(X, weights):

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

    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

[21]:
```
def predict_class(X, weights):

    z = predict_proba(X, weights)
    preds_class = np.where(z>0.5,1,0)
    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

[22]:
```
def binary_loss(X, Y, weights, alpha=0.00001):
    '''
    Binary Cross-entropy Loss

    X:(len(X),len(vocab))
    Y: array len(Y)
    weights: array len(X)
    '''
    preds_proba = predict_proba(X, weights)
    preds_proba = np.clip(preds_proba,1e-9,1-1e-9)

    l = -np.mean(Y*np.log(preds_proba)+(1-Y)*np.log(1-preds_proba)) + alpha *␣
 ↪(weights.T @ weights)

    #L2 regularisation

    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

[170]:
```
def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], lr=0.1,
        alpha=1e-5, epochs=5,
        tolerance=0.0001, print_progress=True):

        m, n = X_tr.shape
        weights = np.random.normal(0,0.1,n)
        training_loss_history = []
        validation_loss_history = []
        epochs_list = []
        index = list(range(m))

        for epochs_count in range(epochs): # Multiple passes
            np.random.shuffle(index) # Randomise the order of training data␣
↪after each pass
            for i in index:
                hypothesis = predict_proba(X_tr[i], weights)
                error = Y_tr[i] - hypothesis
                grad = -np.array(np.dot(error,X_tr[i]).tolist()) + weights *␣
↪alpha/len(weights)
                #Gradient with L2 regularisation
                #Minimise the Binary Cross-entropy loss function
                weights = weights - lr * grad

            training_loss = binary_loss(X_tr, Y_tr, weights=weights,␣
↪alpha=alpha)
            # Training loss
            training_loss_history.append(training_loss)

            validation_loss = binary_loss(X_dev, Y_dev, weights=weights,␣
↪alpha=alpha)
            #Validation loss
            validation_loss_history.append(validation_loss)
            if print_progress==True: # Print the training and developemnt loss
                print('Epoch:{}   training_loss:{}   validation_loss:{}'.
↪format(epochs_count,training_loss,validation_loss))

            if len(validation_loss_history) > 1:
                if abs(validation_loss_history[-1] -␣
↪validation_loss_history[-2]) < tolerance:
                    break

            epochs_list.append(epochs_count)
        print("Stop Epoch: {}".format(epochs_count))
        return weights, training_loss_history, validation_loss_history
```

3.1 Train and Evaluate Logistic Regression with Count vectors

First train the model using SGD:

[171]:
```
Bow_count, Bow_training_loss_history, Bow_validation_loss_history =␣
 ↪SGD(sentiment_train_tfmatrix, sentiment_train_np[:,1],
                                                                          ␣
 ↪sentiment_dev_tfmatrix, sentiment_dev_np[:,1],
                                                                           lr=0.
 ↪06,epochs=30,tolerance=0.001, print_progress=False)
```

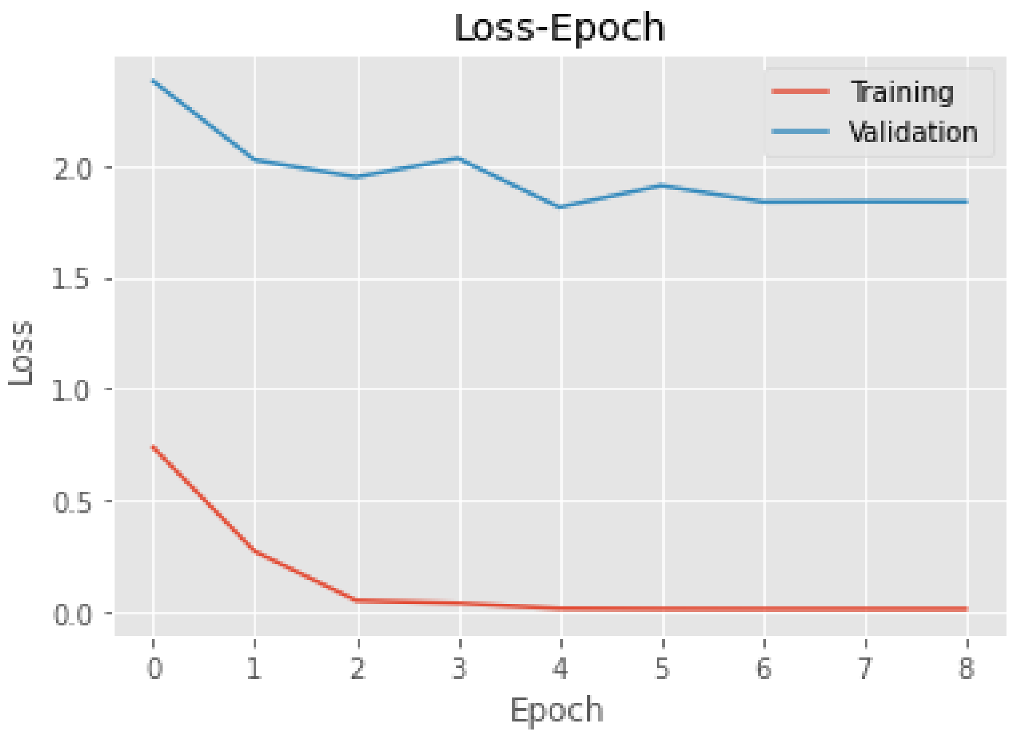
```
Stop Epoch: 8
```

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.

The model is about right. The figure showed that the training loss and validation loss has decreased in the first few epochs and becomes stable after several epochs which means the model did not underfit. At the end of training, the loss did not vibrate or increase which means the model isn't overfit.

[172]:
```
plt.plot(Bow_training_loss_history, label = "Training")
plt.plot(Bow_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[172]:
```
<matplotlib.legend.Legend at 0x19e48723ef0>
```



Evaluation Compute accuracy, precision, recall and F1-scores:

[173]:
```
X_te_count = np.array(sentiment_test_tfmatrix)
Y_te = np.array(sentiment_test_np[:,1].tolist())
preds_te_count = predict_class(X_te_count, Bow_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: 0.83
Precision: 0.8203883495145631
Recall: 0.845
F1-Score: 0.8325123152709358
```

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

Most words in the top-10 words can express emotion and other words may relate to the sentiment train dataset's domain.

[27]:
```
top_neg = Bow_count.argsort()[:10]
for i in top_neg:
    print(id2word[i])
```

```
('bad',)
('any',)
('unfortunately',)
('why',)
('boring',)
('better',)
('nothing',)
('plot',)
('worst',)
('supposed',)
```

[28]:
```
top_pos = Bow_count.argsort()[::-1][:10]
for i in top_pos:
    print(id2word[i])
```

```
('great',)
('seen',)
('black',)
('cauldron',)
('quite',)
('fun',)
('change',)
('also',)
('many',)
('well',)
```

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?

Actually, the classifier we've learned may not have good performance when it applied to a new domain. The classifier with the parameters was learned on the sentiment analysis dataset. The weights of the classifier should corresponding to the features which means when we apply the classfier to a different domain, the features of the new domain will same as the sentiment dataset. In fact, some words which have high frequcy in the sentiment dataset may not appear in the different domain's dataset and it will affect the results of the classifier.

Those features with obvious emotion in the different domain could be picked up as important. If the classifier be used to do sentiment analysis in a different domain, the words with obvious emotion will not have too much difference with those words in sentiment train dataset which means the weights learned from the sentiment train dataset can also be applied to those fearures.

3.1.1 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?

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.06	1e-5	4	0.833
Example1	0.1	1e-9	7	0.822
Example2	0.1	1e-5	8	0.831
Example3	0.04	1e-5	6	0.827
Example4	0.04	1e-9	9	0.824

In order to select the appropriate parameters, we need to use different sets of hyperparameters. To show the different results of different examples, parameters with huge differences will be used. Also, the difference in parameters within ten times of the appropriate has little effect on the results. We can notice that some example may achieve a higher F1-socre that we used, but they also need more epochs to finish training but only increase the F1-Score less than 0.01 which will waste our time.

When the learning rate is too large or too small, the classifier will need more epochs to finish training. Also, when the learning rate is too large, SGD may difficult to get reach local optimum and will vibrate around the local optimum so that it will need more epochs. When the learning weight is too small, the gradiend will converge with a small convergence rate and it will also need more epochs to reach the local optimum.

L2 regularisation aims to give a "penaty" to the loss. A small regularisation strength may cause underfit and a big regularisation stregnth may cause overfit.

3.2 Train and Evaluate Logistic Regression with TF.IDF vectors

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

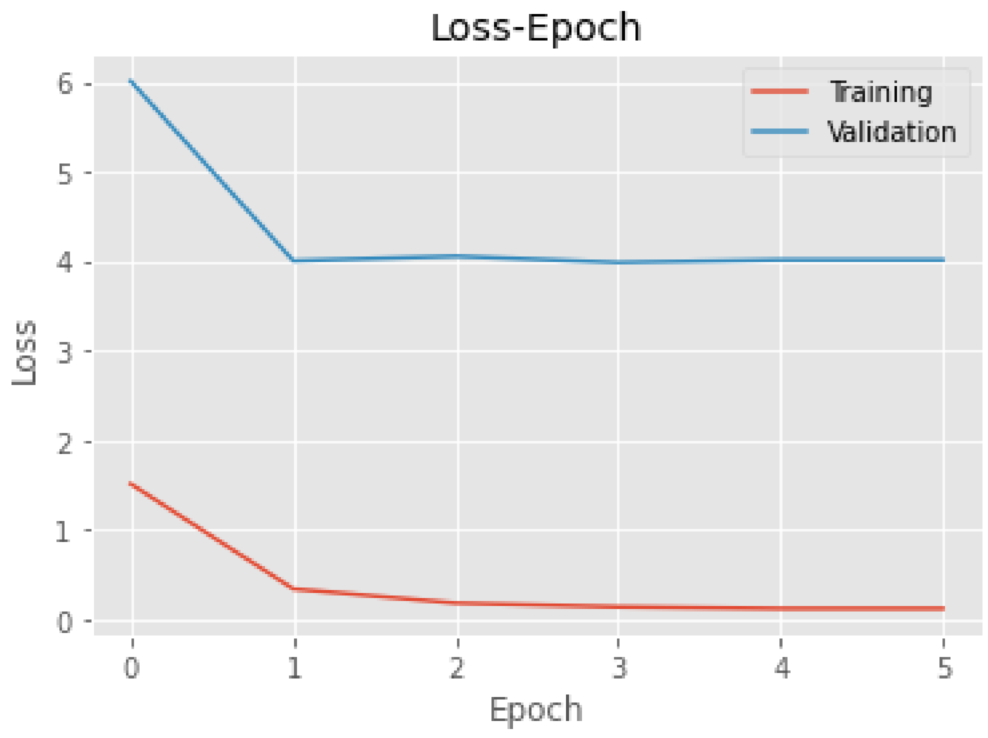
[174]:
```
w_count, training_loss_history, validation_loss_history =␣
 ↪SGD(tfidf_train_matrix,sentiment_train_np[:,1], tfidf_dev_matrix,
                                                       sentiment_dev_np[:
 ↪,1],lr=0.06,epochs=30,tolerance=0.001,
                                                                    ␣
 ↪print_progress=False)
```

```
D:\PROGRAM\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp


Stop Epoch: 5
```

[175]:
```
plt.plot(training_loss_history, label = "Training")
plt.plot(validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[175]:
```
<matplotlib.legend.Legend at 0x19e4814a400>
```



The model is about right because the figure for training loss and validation loss is decreasing smoothly and did nor increase.

[176]:
```
X_te_count = np.array(tfidf_test_matrix)
Y_te = np.array(sentiment_test_np[:,1].tolist())
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: 0.8225
Precision: 0.8056872037914692
Recall: 0.85
F1-Score: 0.8272506082725061
```

The words below showed that most words can express emotions except words such as ‘data’, ‘cameron’ which may related to the special domian.

[177]:
```
top_neg = w_count.argsort()[:10]
for i in top_neg:
    print(id2word[i])
```

```
('worst',)
('dune',)
```

```
('bad',)
('harry',)
('unfortunately',)
('data',)
('ridiculous',)
('series',)
('why',)
('boring',)
```

[178]:
```
top_pos = w_count.argsort()[::-1][:10]
for i in top_pos:
    print(id2word[i])
```

```
('ghost',)
('exotica',)
('adventure',)
('poker',)
('cameron',)
('spoon',)
('robocop',)
('shaw',)
('damon',)
('world',)
```

To choose the best parameters, we used different sets of parameters to test and got the results below.

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.06	1e-5	5	0.827
Example1	0.04	1e-9	7	0.825
Example2	0.1	1e-9	5	0.806
Example3	0.06	1e-9	6	0.821
Example4	0.04	1e-5	5	0.818

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

BOW-tfidf has showed above

BOCN-count

[236]:
```
sentiment_trainbocn_vocab, sentiment_trainbocn_df, sentiment_trainbocn_counts =␣
↪get_vocab(sentiment_train_np[:,0],
                                                                              ␣
↪           token_pattern = "[^0-9A-Za-z\u4e00-\u9fa5]",
                                                                              ␣
↪           stop_words=stop_words, min_df=5,
                                                                        ␣
↪          char_ngrams=True)
```

```
100%|| 1400/1400 [00:05<00:00, 233.65it/s]
```

[237]:
```
id2voc = dict(enumerate(sentiment_trainbocn_vocab))
voc2id = dict(zip(sentiment_trainbocn_vocab,␣
 ↪range(len(sentiment_trainbocn_vocab))))
```

[197]:
```
sentiment_trainbocn_extract_ngrams = whole_extracy_ngrams(sentiment_train_np[:
 ↪,0], char_ngrams=True,
                                                          token_pattern =␣
 ↪"[^0-9A-Za-z\u4e00-\u9fa5]")
sentiment_testbocn_extrace_ngrams = whole_extracy_ngrams(sentiment_test_np[:
 ↪,0], char_ngrams=True,
                                                         token_pattern =␣
 ↪"[^0-9A-Za-z\u4e00-\u9fa5]")
sentiment_devbocn_extrace_ngrams = whole_extracy_ngrams(sentiment_dev_np[:,0],␣
 ↪char_ngrams=True,
                                                        token_pattern =␣
 ↪"[^0-9A-Za-z\u4e00-\u9fa5]")
```

```
100%|| 1400/1400 [00:04<00:00, 318.45it/s]
100%|| 400/400 [00:01<00:00, 316.80it/s]
100%|| 200/200 [00:00<00:00, 320.86it/s]
```

[198]:
```
sentiment_trainbocn_tfmatrix = vectorise(sentiment_trainbocn_extract_ngrams,␣
 ↪sentiment_trainbocn_vocab)
sentiment_testbocn_tfmatrix = vectorise(sentiment_testbocn_extrace_ngrams,␣
 ↪sentiment_trainbocn_vocab)
sentiment_devbocn_tfmatrix = vectorise(sentiment_devbocn_extrace_ngrams,␣
 ↪sentiment_trainbocn_vocab)
```

```
100%|| 1400/1400 [00:04<00:00, 330.76it/s]
100%|| 400/400 [00:01<00:00, 360.67it/s]
100%|| 200/200 [00:00<00:00, 335.90it/s]
```

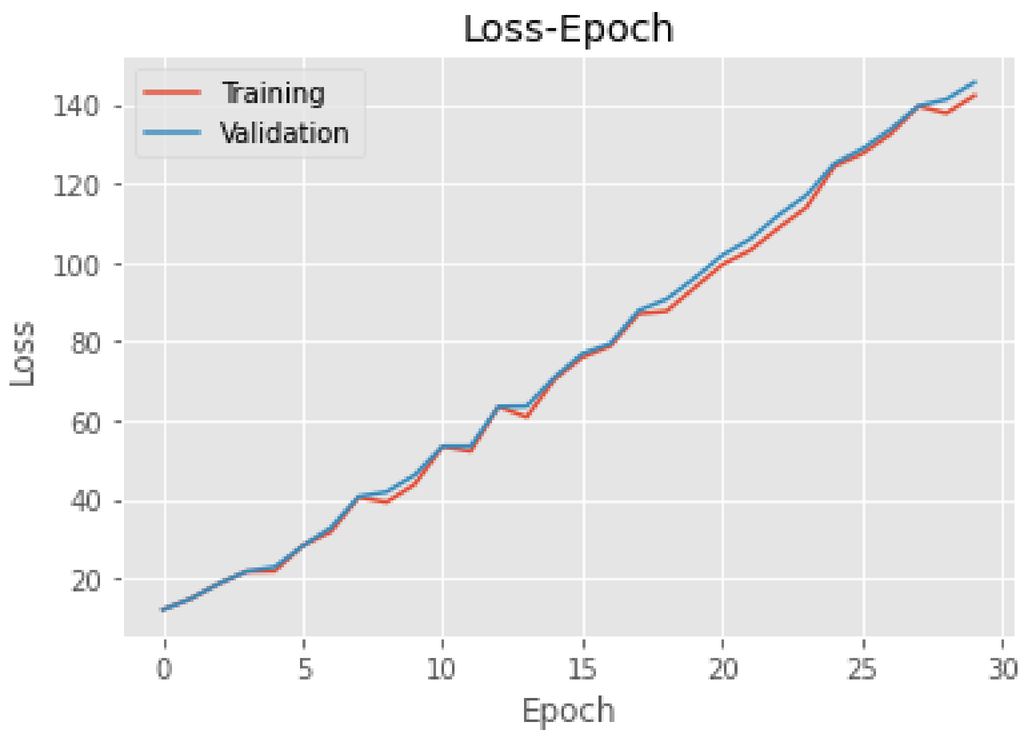
[277]:
```
Bocn_count, Bocn_training_loss_history, Bocn_validation_loss_history =␣
 ↪SGD(sentiment_trainbocn_tfmatrix, sentiment_train_np[:,1],
                                                                   ␣
 ↪sentiment_devbocn_tfmatrix, sentiment_dev_np[:,1],
                                                                    lr=0.
 ↪1,epochs=30,tolerance=0.001, print_progress=False)
```

```
D:\PROGRAM\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp
```

```
Stop Epoch: 29
```

[278]:
```
plt.plot(Bocn_training_loss_history, label = "Training")
plt.plot(Bocn_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[278]:
```
<matplotlib.legend.Legend at 0x19e433c9828>
```



We used 5 sets of parameters and got the best parameters for the model. But from the loss-epoch figure, it is clearly to see that the model with lr=0.1, alpha=1e-5 may have some problems. We tried to change the parameters but sadly find that the results did not changed also every time the model can achieve a high F1-score.

1) The model might have reached the optima point during the first epoch but when the first epoch finished, the model has left the optimal point. Then the model can not find an optimal point then the loss becomes bigger and bigger.

2) The learning rate maight too big for the later epochs which means the model has skipped the optimal points. Adam or some other optimization methods may solve the problem.

[279]:
```
top_neg = Bocn_count.argsort()[:10]
for i in top_neg:
    print(id2voc[i])
```

```
('a', 'd')
('b', 'a', 'd')
('b', 'a')
('h', 'e')
('s', 'a')
('l', 'o')
('e', 'r', 'e')
('u', 'n')
('g', 'i')
('o', 'r', 's')
```

[280]:
```
top_pos = Bocn_count.argsort()[::-1][:10]
for i in top_pos:
    print(id2voc[i])
```

```
('a', 'l')
('g', 'r')
('e', 'l')
('u', 'r')
('e', 'a', 't')
('t', 'r')
('l', 's', 'o')
('g', 'r', 'e')
('e', 'r', 'f')
('r', 'f')
```

It is difficult to know the meaning of those vocabulary above when we use BOCN.

[281]:
```
X_te_count = np.array(sentiment_testbocn_tfmatrix)
Y_te = np.array(sentiment_test_np[:,1].tolist())
preds_te_count = predict_class(X_te_count, Bocn_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: 0.775
Precision: 0.7894736842105263
Recall: 0.75
F1-Score: 0.7692307692307692
```

```
D:\PROGRAM\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp
```

Five different sets of parameters have been used to find the best one.

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.1	1e-5	29	0.769
Example1	0.1	1e-9	>30	0.688
Example2	0.04	1e-9	>30	0.10
Example3	0.06	1e-9	19	0.75
Example4	0.06	1e-5	21	0.713

Bocn-tfidf

[238]:
```
sentiment_bocnidf = []
for word in sentiment_trainbocn_vocab:
    df = sentiment_trainbocn_df.get(word)
    sentiment_bocnidf.append(np.log(number_documents/df))
```

[239]:
```
bocntfidf_train_matrix = sentiment_trainbocn_tfmatrix * sentiment_bocnidf
bocntfidf_dev_matrix = sentiment_devbocn_tfmatrix * sentiment_bocnidf
bocntfidf_test_matrix = sentiment_testbocn_tfmatrix * sentiment_bocnidf
```

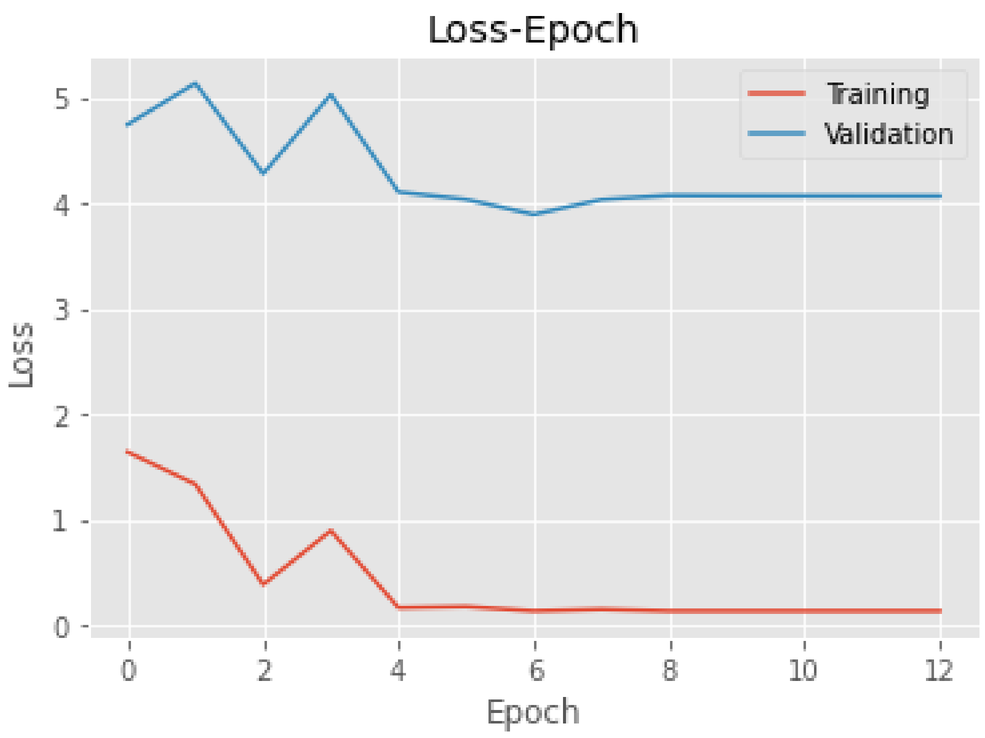
[240]:
```
Bocntfidf_count,Bocntfidf_training_loss_history,Bocntfidf_validation_loss_history␣
 ↪= SGD(bocntfidf_train_matrix, sentiment_train_np[:,1],
                                                                            ␣
 ↪bocntfidf_dev_matrix, sentiment_dev_np[:,1],
                                                                             lr=0.
 ↪06,epochs=30,tolerance=0.001, print_progress=False)
```

```
D:\PROGRAM\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp


Stop Epoch: 12
```

[241]:
```
plt.plot(Bocntfidf_training_loss_history, label = "Training")
plt.plot(Bocntfidf_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[241]:
```
<matplotlib.legend.Legend at 0x19ee2144dd8>
```



The model is about right because the loss of training is decreasing and close to 0. And the last epochs, the loss curve become smoothness.

[242]:
```
top_neg = Bocntfidf_count.argsort()[:10]
for i in top_neg:
    print(id2voc[i])
```

```
('b', 's', 'p')
('b', 'a', 'd')
('w', 'e', 'b')
('e', 'b', 'b')
('a', 'm', 'm')
('w', 'h', 'y')
('w', 'a', 's')
('n', 'a', 'k')
('j', 'i')
('w', 'f', 'u')
```

[244]:
```
top_pos = Bocntfidf_count.argsort()[::-1][:10]
for i in top_pos:
    print(id2voc[i])
```

```
('r', 'f', 'u')
('s', 'c', 'u')
```

```
('j', 'a', 'y')
('w', 'f', 'i')
('r', 'f', 'e')
('j', 'o', 'b')
('n', 't', 'y')
('i', 'l', 'a')
('a', 'g', 'r')
('b', 'o', 'c')
```

It is difficult to find the meaning of those vocabularies above when we use BOCN.

[245]:
```
X_te_count = np.array(bocntfidf_test_matrix)
Y_te = np.array(sentiment_test_np[:,1].tolist())
preds_te_count = predict_class(X_te_count, Bocntfidf_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: 0.785
Precision: 0.7478260869565218
Recall: 0.86
F1-Score: 0.7999999999999999

D:\PROGRAM\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp
```

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.06	1e-5	4	0.800
Example1	0.06	1e-9	3	0.787
Example2	0.04	1e-9	5	0.790
Example3	0.1	1e-9	2	0.742
Example4	0.1	1e-5	>30	0.74

BOW(tfidf)+BOCN(tfidf)

[246]:
```
bow_bocn_train = np.hstack((tfidf_train_matrix,bocntfidf_train_matrix))
bow_bocn_dev = np.hstack((tfidf_dev_matrix, bocntfidf_dev_matrix))
bow_bocn_test = np.hstack((tfidf_test_matrix, bocntfidf_test_matrix))
```

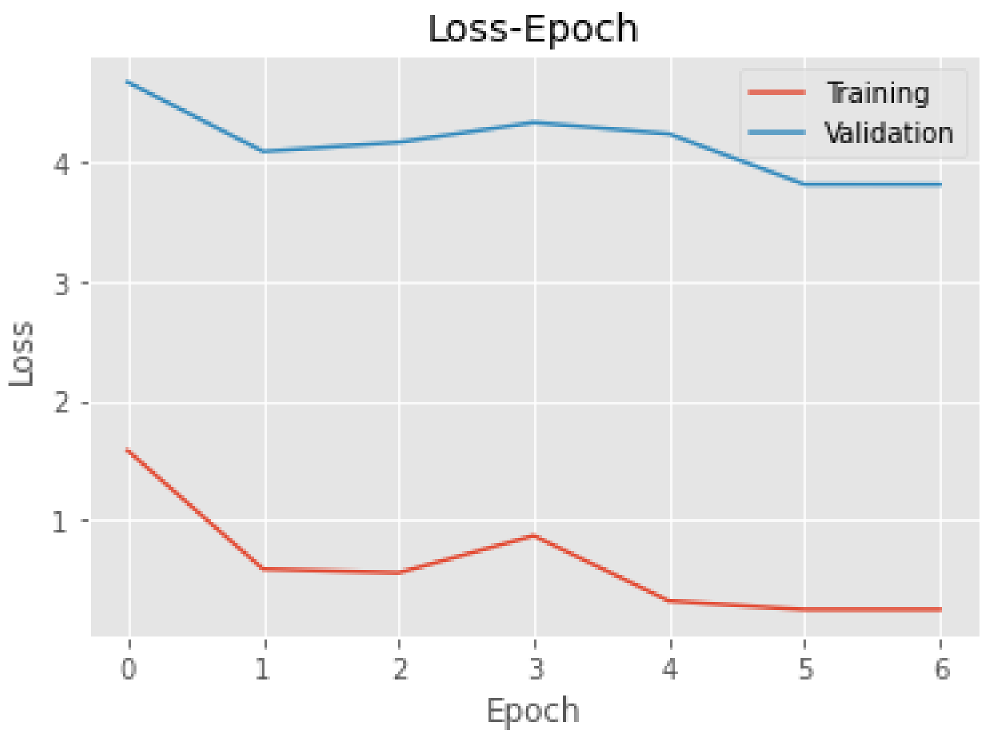
[264]:
```
bowbocn_count, bowbocn_training_loss_history, bowbocn_validation_loss_history =␣
 ↪SGD(bow_bocn_train, sentiment_train_np[:,1],
                                                                          ␣
 ↪bow_bocn_dev, sentiment_dev_np[:,1],
                                                                        lr=0.
↪06,epochs=30,tolerance=0.001, print_progress=False)
```

```
D:\PROGRAM\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp


Stop Epoch: 6
```

[265]:
```
plt.plot(bowbocn_training_loss_history, label = "Training")
plt.plot(bowbocn_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[265]:
```
<matplotlib.legend.Legend at 0x19e73312400>
```



The model is about right althogh during the 4th epoch the loss has increased. At the end of training we can find that the loss is decreased and close to 0 which means the model is about right.

[266]:
```
X_te_count = np.array(bow_bocn_test)
Y_te = np.array(sentiment_test_np[:,1].tolist())
preds_te_count = predict_class(X_te_count, bowbocn_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: 0.835
Precision: 0.8221153846153846
Recall: 0.855
F1-Score: 0.838235294117647

D:\PROGRAM\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp
```

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-BOCN	0.1	1e-9	2	0.839
Example1	0.1	1e-5	4	0.837
Example2	0.06	1e-5	1	0.793
Example3	0.06	1e-9	2	0.796
Example4	0.04	1e-9	2	0.819

3.3 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count	0.820	0.845	0.832
BOW-tfidf	0.805	0.85	0.827
BOCN-count	0.790	0.75	0.770
BOCN-tfidf	0.748	0.86	0.800
BOW(tfidf)+BOCN(tfidf)	0.822	0.855	0.838

The best model is BOW(tfidf)+BOCN(tfidf), it combines BOW-tfidf and BOCN-tfidf which can have more features and representative the dataset better. Also, based on the equation: tfidf=TF*IDF , IDF can reduce the importance of words that appear more frequency in the document in the model which can help the classifier give higher weights to those words which can representitive the sentiment of sentence better.

Also, we can notice that BOCN-count has the worst performance in the five models. It might because of the ngram range is (1,3) which means features will contain some single letters which will appear in all sentence with a high weights.

4 Task 2: 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.

[282]:
```
topic_train_df = pd.read_csv('./data_topic/train.csv',names = ['Sentence',␣
 ↪'Sentiment'])
topic_test_df = pd.read_csv('./data_topic/test.csv',names = ['Sentence',␣
 ↪'Sentiment'])
topic_dev_df = pd.read_csv('./data_topic/dev.csv',names = ['Sentence',␣
 ↪'Sentiment'])
```

[283]:
```
topic_dev_df.head()
```

[283]:
```
   Sentence                                          Sentiment
0         1  BAGHDAD, Iraq - An Islamic militant group that...
1         1  Parts of Los Angeles international airport are...
2         1  AFP - Facing a issue that once tripped up his ...
3         1  The leader of militant Lebanese group Hezbolla...
4         1  JAKARTA : ASEAN finance ministers ended a meet...
```

[284]:
```
topic_train_np = topic_train_df.values
topic_test_np = topic_test_df.values
topic_dev_np = topic_dev_df.values
```

[285]:
```
topic_train_np.shape
```

[285]:
```
(2400, 2)
```

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

[286]:
```
def softmax(z):
    smax = np.longfloat(np.exp(z - np.max(z)))
    smax = smax/np.sum(smax).reshape(-1,1)
    return smax
```

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

[287]:
```
def predict_proba(X, weights):

    preds_proba = softmax(np.dot(X, weights.T))

    return preds_proba
```

[364]:
```
def predict_class(X, weights):
    z = predict_proba(X, weights)
    z_list = z.tolist()
    preds_class = np.argmax(z_list, axis=1) + 1
    return preds_class
```

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

[529]:
```
def categorical_loss(X, Y, weights, num_classes=5, alpha=0.00001):
    '''
    X:(len(X),len(vocab))
    Y: array len(Y)
    weights: array len(X)
    '''
    preds_prob = np.clip(predict_proba(X, weights),1e-11, 1-1e-11)
    l = np.sum(-np.log(np.max(preds_prob, axis=0)), axis=0) + alpha * np.
 ↪sum(weights * weights)/(2 * weights.shape[0] * weights.shape[1])

    # Categorical loss with L2 regularisation,
    return l
```

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

[592]:
```
def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], num_classes=5, lr=0.01, alpha=0.00001,
        epochs=5, tolerance=0.001, print_progress=True):

        m, n = X_tr.shape
        weights = np.random.randn(num_classes, n) # Initialize weights
        training_loss_history = []
        validation_loss_history = []
        epochs_list = []
        index = list(range(m))

        for epochs_count in range(epochs): #Perform multiple passes (epochs)␣
 ↪over the training data
            np.random.shuffle(index) # Randomise the order of training data␣
 ↪after each pass
            for i in index:
                hypothesis = predict_proba(X_tr[i], weights)
                hypothesis[:,Y_tr[i]-1] = hypothesis[:,Y_tr[i]-1]-1
                grad = np.array(hypothesis.T * X_tr[i].reshape(-1,1).T) +␣
 ↪weights * alpha/ (num_classes*n)
                #Use L2 regularisation
                weights = weights - lr * grad
                #Minimise the Categorical Cross-entropy loss function

            training_loss = np.sum(categorical_loss(X_tr, Y_tr, weights,␣
↪alpha=alpha, num_classes=num_classes))/ m # training loss
            training_loss_history.append(training_loss)

            validation_loss = np.sum(categorical_loss(X_dev,Y_dev, weights,␣
↪alpha=alpha, num_classes=num_classes)) / X_dev.shape[0]
            #Loss after
            validation_loss_history.append(validation_loss)
            if print_progress==True:
                print('Epoch:{}   training_loss:{}   validation_loss:{}'.
↪format(epochs_count,training_loss_new,validation_loss_new))

            if len(validation_loss_history) > 1:
                if abs(validation_loss_history[-1] -␣
↪validation_loss_history[-2]) < tolerance:
                    break
            epochs_list.append(epochs_count)
        print("Stop Epoch: {}".format(epochs_count))
        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 the different vector representations

[582]:
```
topic_train_vocab, topic_train_df, topic_train_counts =␣
↪get_vocab(topic_train_np[:,1],token_pattern="[a-z|\']+",
                                                                          ␣
↪stop_words=stop_words, min_df=5)
```

```
100%|| 2400/2400 [00:00<00:00, 11681.63it/s]
```

[583]:
```
topic_id2word = dict(enumerate(topic_train_vocab))
topic_word2id = dict(zip(topic_train_vocab, range(len(topic_train_vocab))))
```

[584]:
```
topic_train_extract_ngrams = whole_extracy_ngrams(topic_train_np[:,1],␣
↪token_pattern="[a-z|\']+")
topic_test_extract_ngrams = whole_extracy_ngrams(topic_test_np[:,1],␣
↪token_pattern="[a-z|\']+")
topic_dev_extract_ngrams = whole_extracy_ngrams(topic_dev_np[:,1],␣
↪token_pattern="[a-z|\']+")
```

```
100%|| 2400/2400 [00:00<00:00, 19098.44it/s]
100%|| 900/900 [00:00<00:00, 18800.01it/s]
100%|| 150/150 [00:00<00:00, 18800.11it/s]
```

[585]:
```
topic_train_tfmatrix = vectorise(topic_train_extract_ngrams, topic_train_vocab)
topic_test_tfmatrix = vectorise(topic_test_extract_ngrams, topic_train_vocab)
topic_dev_tfmatrix = vectorise(topic_dev_extract_ngrams, topic_train_vocab)
```

```
100%|| 2400/2400 [00:01<00:00, 1546.51it/s]
100%|| 900/900 [00:00<00:00, 1532.90it/s]
100%|| 150/150 [00:00<00:00, 1367.28it/s]
```

[599]:
```
Bow_count, Bow_training_loss_history, Bow_validation_loss_history =␣
↪SGD(topic_train_tfmatrix, topic_train_np[:,0], topic_dev_tfmatrix,␣
↪topic_dev_np[:,0],
                                                                              ␣
↪num_classes=3,lr=0.1,epochs=30,tolerance=0.001, print_progress=False)
```

```
Stop Epoch: 16
```

[600]:
```
top_first = Bow_count[0].argsort()[:10]
for i in top_first:
    print(topic_id2word[i])
```

```
('consecutive',)
('teams',)
('investigation',)
('apital',)
("country's", 'oil', 'exports')
('euters', 'ecurities')
('havez', 'office')
('medals', 'thens')
('dollar',)
('playing',)
```

[601]:
```
top_second = Bow_count[1].argsort()[:10]
for i in top_second:
    print(topic_id2word[i])
```

```
('uote', 'rofile')
('holy', 'shrine')
('stock', 'market')
('dollar',)
('replaced',)
('immediately',)
('extra',)
('hoping',)
('nc',)
('ortel',)
```

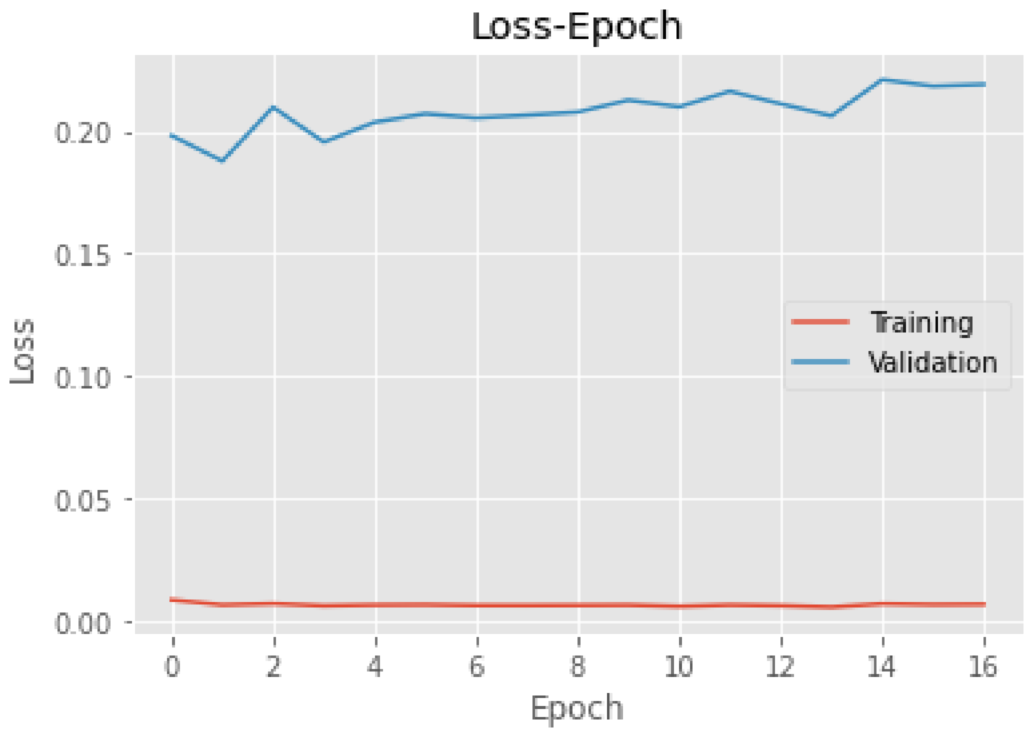
[602]:
```
top_third = Bow_count[2].argsort()[:10]
for i in top_third:
    print(topic_id2word[i])
```

```
('tewart',)
('reece', 'ichael', 'helps')
('round',)
('hite',)
('obe',)
('futures',)
('closely',)
('opposition',)
('put',)
('country',)
```

The words above are focus on speacial domain which means it is different to apply to other domain.

[603]:
```
plt.plot(Bow_training_loss_history, label = "Training")
plt.plot(Bow_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[603]:
```
<matplotlib.legend.Legend at 0x19e43ca1908>
```



The model is about right. Although the curve of training loss looks like a line, it maight because in the first epoch, the model has reach the optimal point which can achieve a very small loss. Then the it looks like a line.

Compute accuracy, precision, recall and F1-scores:

[604]:
```
X_te_count = np.array(topic_test_tfmatrix)
label = np.array(topic_test_np[:,0])
predicted = predict_class(X_te_count, Bow_count)
preds_te = list()
Y_te = list()

for i in range(len(label)):
    Y_te.append(int(label[i]))
    preds_te.append(int(predicted[i]))

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: 0.8022222222222222
Precision: 0.8062752264569267
Recall: 0.8022222222222223
F1-Score: 0.802703018208769
```

Also, we notice that the way to select hyperparameters is same with task1 and the difference in parameters within ten times of the appropriate has little effect on the results. 4 examples has showed above and it is easily to find that those hyperparameters in examples can not increase the perofrmance of the model, it may need more time to finish training but can not improve the F1-score.

The effect of regularisation strength and the relationship between learning rate and the number of epochs is same as task1.

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.1	1e-5	16	0.803
Example1	0.1	1e-7	11	0.767
Example2	0.04	1e-5	5	0.706
Example3	0.06	1e-5	10	0.758
Example4	0.06	1e-7	6	0.736

4.0.1 Now repeat the training and evaluation process for BOW-tfidf, BOCN-count, BOCN-tfidf, BOW+BOCN including hyperparameter tuning…

BOW-tfidf

[385]:
```
topic_idf = []
number_documents = len(topic_train_np)
for word in topic_train_vocab:
    df = topic_train_df.get(word)
    topic_idf.append(np.log(number_documents/df))
```

[386]:
```
tfidf_train_matrix = topic_train_tfmatrix * np.array(topic_idf)
tfidf_dev_matrix = topic_dev_tfmatrix * np.array(topic_idf)
tfidf_test_matrix = topic_test_tfmatrix * np.array(topic_idf)
```

[459]:
```
w_count, training_loss_history, validation_loss_history =␣
 ↪SGD(tfidf_train_matrix,topic_train_np[:,0], tfidf_dev_matrix,
                                                            topic_dev_np[:
 ↪,0],num_classes=3,lr=0.1,alpha=1e-6,
                                                                        ␣
 ↪epochs=30,tolerance=0.001, print_progress=False)
```

```
Stop Epoch: 1
```

[538]:
```
top_bowtfidf_first = w_count[0].argsort()[:10]
for i in top_bowtfidf_first:
    print(topic_id2word[i])
```

```
('fund',)
('season',)
('exports',)
('more',)
('t',)
('irways',)
('reece', 's')
('stopped',)
('challenge',)
('executive',)
```

[539]:
```
top_bowtfidf_second = w_count[1].argsort()[:10]
for i in top_bowtfidf_second:
    print(topic_id2word[i])
```

```
('said',)
('resident',)
('proposal',)
('after', 'flash')
('double',)
('such',)
('gt',)
('arfur', 'region')
('government',)
('quickinfo', 'fullquote')
```

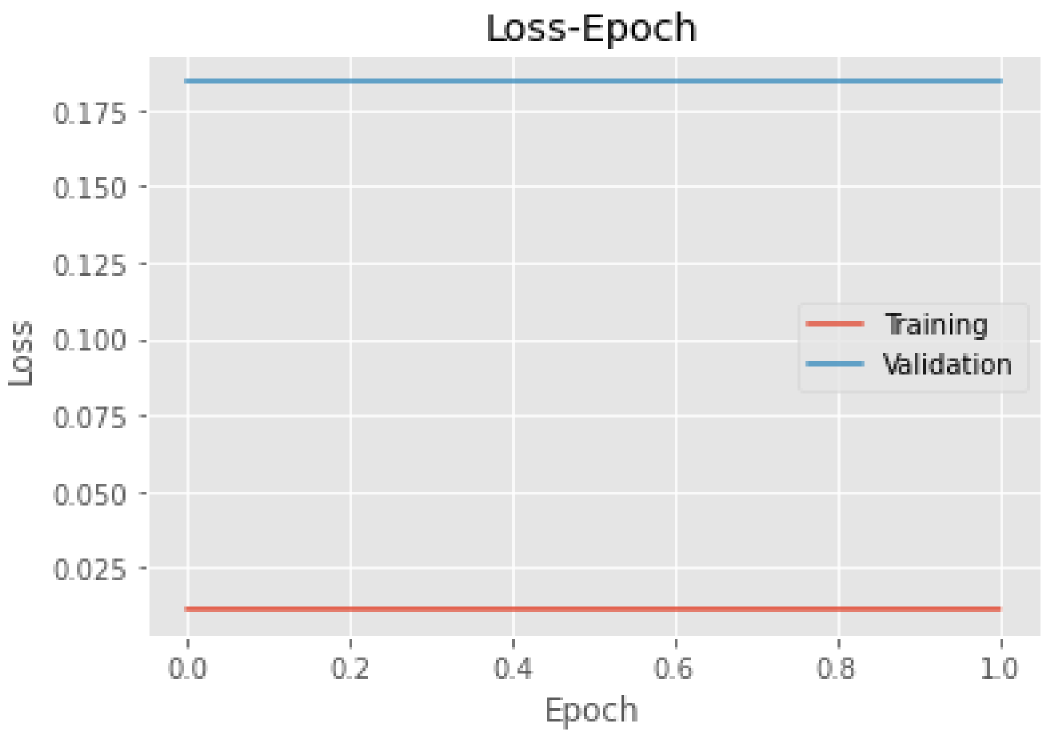
[540]:
```
top_bowtfidf_third = w_count[2].argsort()[:10]
for i in top_bowtfidf_third:
    print(topic_id2word[i])
```

```
('gold',)
('ope', 'ohn', 'aul')
('race',)
('seen',)
('beat',)
('ovember',)
('movement',)
('basketball',)
('law',)
('thens',)
```

The words above are focus on speacial domain which means it is different to apply to other domain.

[520]:
```
plt.plot(training_loss_history, label = "Training")
plt.plot(validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[520]:
```
<matplotlib.legend.Legend at 0x19e44e204e0>
```



Because the model only trained 2 epochs, the loss of training and validation will becomes lines.But we can find the model can achieve a high F1-score and the loss did not increase which men

[461]:
```
X_te_count = np.array(tfidf_test_matrix)
label = np.array(topic_test_np[:,0].tolist())
predicted = predict_class(X_te_count, w_count)
preds_te = list()
Y_te = list()
for i in range(len(label)):
    Y_te.append(int(label[i]))
    preds_te.append(int(predicted[i]))

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: 0.7933333333333333
Precision: 0.7970458466029724
Recall: 0.7933333333333333
F1-Score: 0.7927693396701904
```

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.1	1e-5	6	0.793
Example1	0.1	1e-7	7	0.775
Example2	0.04	1e-5	1	0.736
Example3	0.06	1e-5	1	0.765
Example4	0.06	1e-7	2	0.770

BOCN-count

[390]:
```
topic_trainbocn_vocab, topic_trainbocn_df, topic_trainbocn_counts =␣
 ↪get_vocab(topic_train_np[:,1],
                                                                      ␣
 ↪token_pattern = "[^0-9A-Za-z\u4e00-\u9fa5]",
                                                                    ␣
 ↪stop_words=stop_words, min_df=5,char_ngrams=True)
```

```
100%|| 2400/2400 [00:00<00:00, 3182.48it/s]
```

[542]:
```
topicBocn_id2word = dict(enumerate(topic_trainbocn_vocab))
topicBocn_word2id = dict(zip(topic_train_vocab,␣
 ↪range(len(topic_trainbocn_vocab))))
```

[391]:
```
topic_trainbocn_extract_ngrams = whole_extracy_ngrams(topic_train_np[:,1],␣
↪char_ngrams=True,
                                                    token_pattern =␣
↪"[^0-9A-Za-z\u4e00-\u9fa5]")
topic_testbocn_extrace_ngrams = whole_extracy_ngrams(topic_test_np[:,1],␣
↪char_ngrams=True,
                                                    token_pattern =␣
↪"[^0-9A-Za-z\u4e00-\u9fa5]")
topic_devbocn_extrace_ngrams = whole_extracy_ngrams(topic_dev_np[:,1],␣
↪char_ngrams=True,
                                                    token_pattern =␣
↪"[^0-9A-Za-z\u4e00-\u9fa5]")
```

```
100%|| 2400/2400 [00:00<00:00, 5474.90it/s]
100%|| 900/900 [00:00<00:00, 4811.00it/s]
100%|| 150/150 [00:00<00:00, 4297.03it/s]
```

[392]:
```
topic_trainbocn_tfmatrix = vectorise(topic_trainbocn_extract_ngrams,␣
↪topic_trainbocn_vocab)
topic_testbocn_tfmatrix = vectorise(topic_testbocn_extrace_ngrams,␣
↪topic_trainbocn_vocab)
topic_devbocn_tfmatrix = vectorise(topic_devbocn_extrace_ngrams,␣
↪topic_trainbocn_vocab)
```

```
100%|| 2400/2400 [00:06<00:00, 399.58it/s]
100%|| 900/900 [00:02<00:00, 402.00it/s]
100%|| 150/150 [00:00<00:00, 293.18it/s]
```

[517]:
```
Bocn_count, Bocn_training_loss_history, Bocn_validation_loss_history =␣
↪SGD(topic_trainbocn_tfmatrix, topic_train_np[:,0],
                                                                     ␣
↪topic_devbocn_tfmatrix, topic_dev_np[:,0],num_classes=3,
                                                                      lr=0.
↪06, alpha=1e-7,epochs=30,tolerance=0.001, print_progress=False)
```

```
Stop Epoch: 1
```

[544]:
```
top_bocn_first = Bocn_count[0].argsort()[:10]
for i in top_bocn_first:
    print(topicBocn_id2word[i])
```

```
('a', 'm')
('A',)
('s', 'p')
('a', 'm', 'e')
```

```
('i', 's')
('p', 'i')
('l', 'd')
('O', 'l', 'y')
('x', 'r')
('P',)
```

[545]:
```
top_bocn_second = Bocn_count[1].argsort()[:10]
for i in top_bocn_second:
    print(topicBocn_id2word[i])
```

```
('l', 'i', 't')
('s', 'a', 'i')
('I',)
('a', 'd', 'i')
('r', 'a', 'q')
('I', 'N', 'G')
('d', 'a')
('N', 'D', 'O')
('n', 'e', 'a')
('e', 'n', 'e')
```

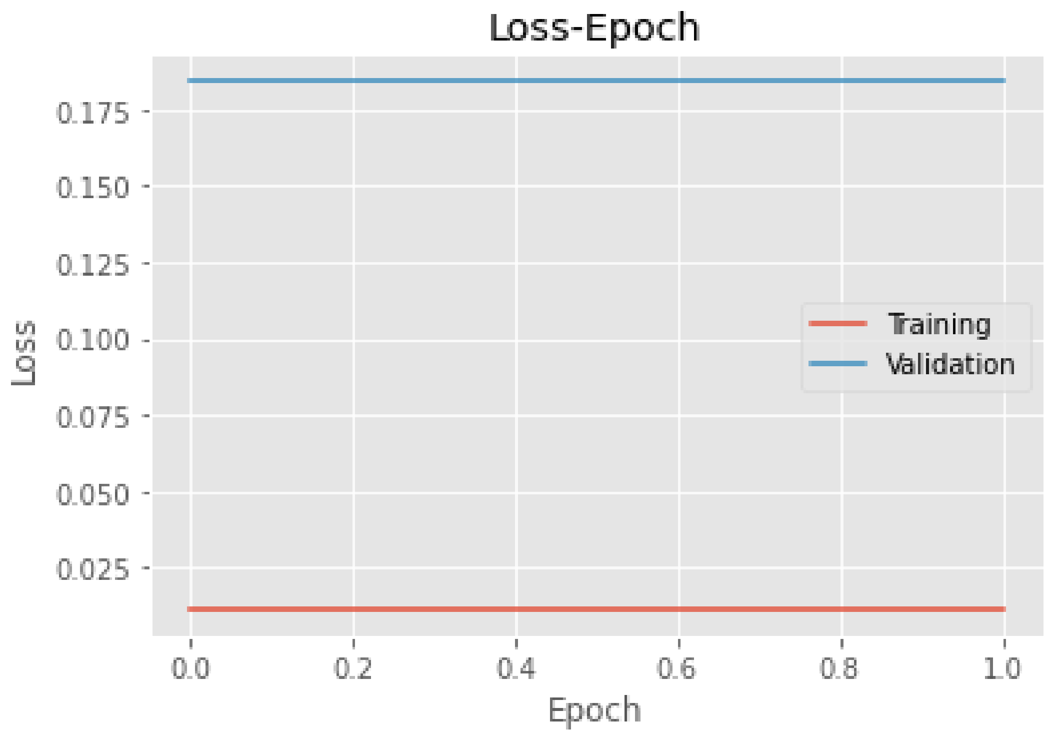
[546]:
```
top_bocn_third = Bocn_count[2].argsort()[:10]
for i in top_bocn_third:
    print(topicBocn_id2word[i])
```

```
('a', 'm')
('A',)
('s', 'p')
('a', 'm', 'e')
('i', 's')
('p', 'i')
('l', 'd')
('O', 'l', 'y')
('x', 'r')
('P',)
```

When we use BOCN, it is difficult to know what those features means.

[518]:
```
plt.plot(Bocn_training_loss_history, label = "Training")
plt.plot(Bocn_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[518]:
```
<matplotlib.legend.Legend at 0x19e446fd470>
```



The figure is almost same as last model which means it has reach the optimal point in 2 epochs and we will get 2 lines. Base on the F1-score is 0.792 and we can get the conclusion that the model is about right.

[519]:
```
X_te_count = np.array(topic_testbocn_tfmatrix)
label = np.array(topic_test_np[:,0].tolist())
predicted = predict_class(X_te_count, Bocn_count)
preds_te = list()
Y_te = list()
for i in range(len(label)):
    Y_te.append(int(label[i]))
    preds_te.append(int(predicted[i]))

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: 0.7922222222222223
Precision: 0.793938194960797
Recall: 0.7922222222222222
F1-Score: 0.7920339913053782
```

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.06	1e-7	4	0.792
Example1	0.1	1e-7	1	0.542
Example2	0.06	1e-5	1	0.746
Example3	0.04	1e-7	1	0.664
Example4	0.06	1e-7	2	0.677

BOCN-tfidf

[396]:
```
topic_bocnidf = []
number_documents = len(topic_train_np)
for word in topic_trainbocn_vocab:
    df = topic_trainbocn_df.get(word)
    topic_bocnidf.append(np.log(number_documents/df))
```

[397]:
```
bocntfidf_train_matrix = topic_trainbocn_tfmatrix * topic_bocnidf
bocntfidf_dev_matrix = topic_devbocn_tfmatrix * topic_bocnidf
bocntfidf_test_matrix = topic_testbocn_tfmatrix * topic_bocnidf
```

[618]:
```
Bocntfidf_count, Bocntfidf_training_loss_history,␣
 ↪Bocntfidf_validation_loss_history = SGD(bocntfidf_train_matrix,␣
 ↪topic_train_np[:,0],
                                                                   ␣
 ↪bocntfidf_dev_matrix, topic_dev_np[:,0],
                                                                   ␣
 ↪num_classes=3, lr=0.04,epochs=30,
                                                                   ␣
 ↪tolerance=0.001, print_progress=False)
```

```
Stop Epoch: 3
```

[619]:
```
top_bocntfidf_first = Bocntfidf_count[0].argsort()[:10]
for i in top_bocntfidf_first:
    print(topicBocn_id2word[i])
```

```
('R', 'B')
('f', 'u', 'n')
('V', 'i', 'c')
('s', 'o', 'n')
('r', 'b', 'i')
('f', 'h')
('e', 'a', 'm')
('n', 'a', 'b')
('y', 'W', 'e')
('g', 'o', 'a')
```

[620]:
```
top_bocntfidf_second = Bocntfidf_count[1].argsort()[:10]
for i in top_bocntfidf_second:
    print(topicBocn_id2word[i])
```

```
('o', 'c', 'k')
('w', 'e', 'd')
('d', 'd', 'e')
('g', 'c', 'r')
('e', 's', 'i')
('i', 'L', 'a')
('r', 'a', 'q')
('g', 'l', 'o')
('l', 'l', 'q')
('e', 'f', 'u')
```

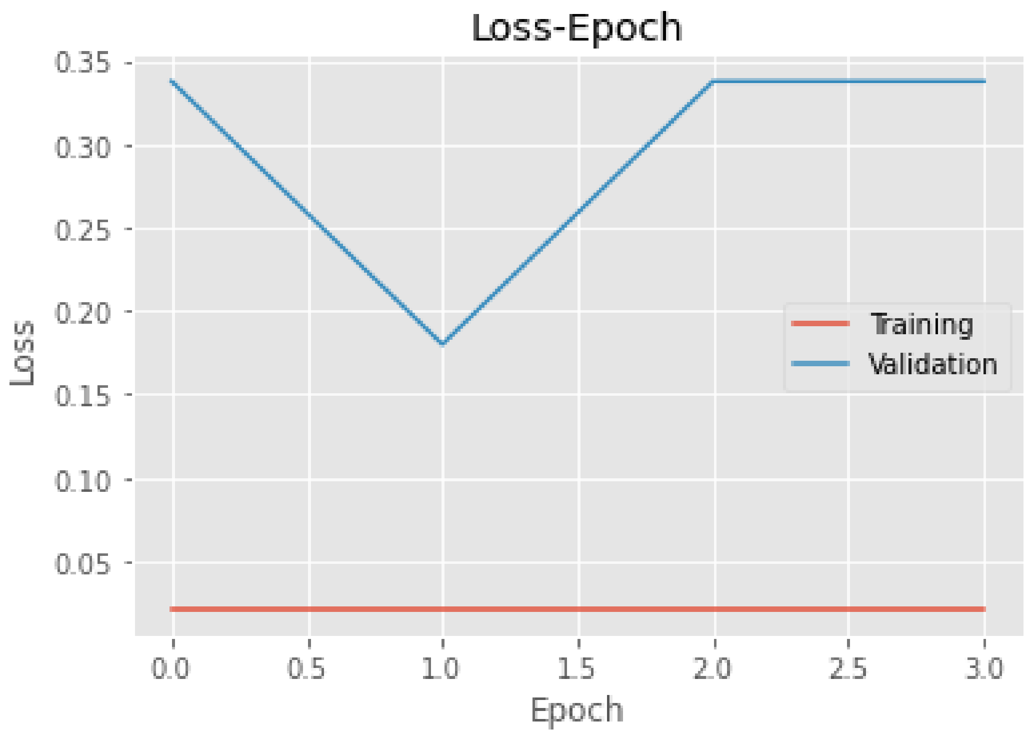
[621]:
```
top_bocntfidf_third = Bocntfidf_count[2].argsort()[:10]
for i in top_bocntfidf_third:
    print(topicBocn_id2word[i])
```

```
('n', 'R')
('o', 'u', 's')
('o', 'p', 'r')
('A', 'P', 'J')
('e', 'd', 'B')
('r', 'i', 's')
('r', 'W', 'o')
('C', 'h', 'e')
('z', 'e', 'm')
('h', 'u', 'm')
```

When we use BOCN it is difficult to know the meaning of those vocabulary.

[622]:
```
plt.plot(Bocntfidf_training_loss_history, label = "Training")
plt.plot(Bocntfidf_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[622]:
```
<matplotlib.legend.Legend at 0x19e43de62b0>
```



The figure means the maight close to the optimal point in the first epoch and in the other epochs the difference of loss will be very small, that's why the curve of training loss looks like a line. We can obtain the conclusion that the model is about right.

[623]:
```
X_te_count = np.array(bocntfidf_test_matrix)
label = np.array(topic_test_np[:,0].tolist())
predicted = predict_class(X_te_count, Bocntfidf_count)
preds_te = list()
Y_te = list()
for i in range(len(label)):
    Y_te.append(int(label[i]))
    preds_te.append(int(predicted[i]))

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: 0.7955555555555556
Precision: 0.7979475578385791
Recall: 0.7955555555555556
F1-Score: 0.7963336100745285
```

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-count	0.1	1e-5	1	0.796
Example1	0.1	1e-7	1	0.790
Example2	0.06	1e-5	1	0.792
Example3	0.04	1e-7	24	0.763
Example4	0.06	1e-7	10	0.792

BOW(tfidf)+BOCN(tfidf)

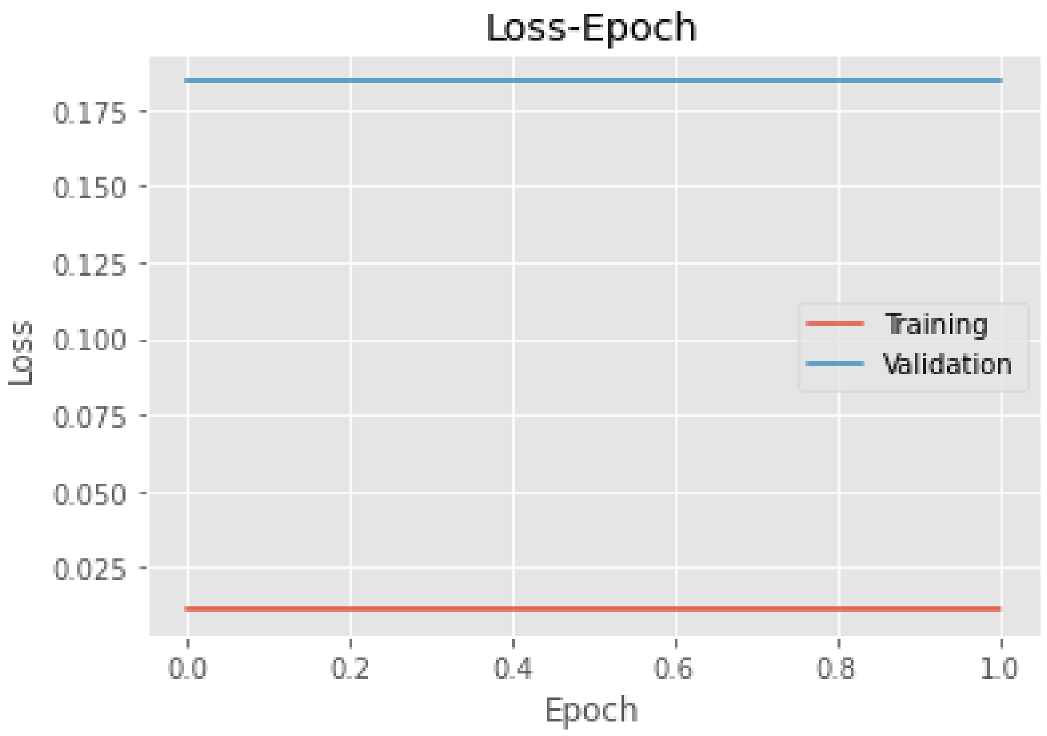
[485]:
```
bow_bocn_train = np.hstack((tfidf_train_matrix,bocntfidf_train_matrix))
bow_bocn_dev = np.hstack((tfidf_dev_matrix, bocntfidf_dev_matrix))
bow_bocn_test = np.hstack((tfidf_test_matrix, bocntfidf_test_matrix))
```

[504]:
```
bowbocn_count, bowbocn_training_loss_history, bowbocn_validation_loss_history =␣
 ↪SGD(bow_bocn_train, topic_train_np[:,0],
                                                                              ␣
 ↪bow_bocn_dev, topic_dev_np[:,0],
                                                                              ␣
 ↪num_classes=3,lr=0.06,epochs=30,
                                                                              ␣
 ↪tolerance=0.001, print_progress=False)
```

```
Stop Epoch: 1
```

[506]:
```
plt.plot(bowbocn_training_loss_history, label = "Training")
plt.plot(bowbocn_validation_loss_history, label = "Validation")
plt.title("Loss-Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[506]:
```
<matplotlib.legend.Legend at 0x19e440be080>
```



Beacuse the model only trained 2 epochs so the line of the loss will become lines but the F1-socre can achieve 0.804 then we can obtain the conclusion that the model is about right.

[505]:
```
X_te_count = np.array(bow_bocn_test)
label = np.array(topic_test_np[:,0].tolist())
predicted = predict_class(X_te_count, bowbocn_count)
preds_te = list()
Y_te = list()
for i in range(len(label)):
    Y_te.append(int(label[i]))
    preds_te.append(int(predicted[i]))

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: 0.8044444444444444
Precision: 0.8065129979800639
Recall: 0.8044444444444445
F1-Score: 0.8042379898540304
```

Example	Learing Rate	Alpha	Epochs	F1-Score
BOW-BOCN	0.06	1e-5	1	0.804
Example1	0.06	1e-7	4	0.803
Example2	0.04	1e-5	1	0.792
Example3	0.01	1e-5	1	0.219
Example4	0.01	1e-7	2	0.797

4.1 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count	0.764	0.761	0.761
BOW-tfidf	0.797	0.793	0.792
BOCN-count	0.796	0.785	0.784
BOCN-tfidf	0.783	0.779	0.796
BOW+BOCN	0.806	0.804	0.804

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

The best performing model is BOW+BOCN. It maight have sereval reasons: 1) BOW(tfidf)+BOCN(tfidf) has more features than other 4 models which can representive the sentence better.

2) Both of BOW and BOCN have use tfidf to give weights to each features which can achieve better performance than only use count. TFIDF = TF*IDF and IDF can reduce the importance of words that appear more frequency in the document in the model.