assignment1

April 19, 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.

[301]:
```python
# Total time to run the notebook is around 15/16 mins on a 2015 Macbook Pro
from datetime import datetime
now = datetime.now()

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).

[302]:
```python
# Load training data with named columns
data_sentiment_train = pd.read_csv("data_sentiment/train.csv", names=["Text",␣
 ↪"Label"])
data_sentiment_test = pd.read_csv("data_sentiment/test.csv", names=["Text",␣
 ↪"Label"])
data_sentiment_dev = pd.read_csv("data_sentiment/dev.csv", names=["Text",␣
 ↪"Label"])

data_topic_train = pd.read_csv("data_topic/train.csv", names=["Label", "Text"])
data_topic_test = pd.read_csv("data_topic/test.csv", names=["Label", "Text"])
data_topic_dev = pd.read_csv("data_topic/dev.csv", names=["Label", "Text"])
```

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

[303]:
```python
data_sentiment_train.head()
```

[303]:

```
                                                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:

[304]:

```
# Split data into lists and np arrays

# TRAINING DATA
data_sentiment_train_data = list(data_sentiment_train.loc[:,"Text"])
Y_tr = np.array(data_sentiment_train.loc[:,"Label"])


# TEST SET
data_sentiment_test_data = list(data_sentiment_test.loc[:,"Text"])
Y_test = np.array(data_sentiment_test.loc[:,"Label"])


# VALIDATION SET
data_sentiment_dev_data = list(data_sentiment_dev.loc[:,"Text"])
Y_dev = np.array(data_sentiment_dev.loc[:,"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).

[305]:

```
# 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']


# This list was found on GitHub here ------->    https://gist.github.com/
 ↪sebleier/554280
# Also added days of the week which improved results for the muticlass model
# Cant see the list in the pdf because the line is too long
# Also removed days of the week as this increased the F1 measure for each␣
 ↪classifier
```

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.

[306]:
```
# token_pattern was changed to allow ' marks and one letter tokens


def extract_ngrams(x_raw, ngram_range=(1,3),␣
↪token_pattern=r"\b[A-Za-z\'][A-Za-z\']*\b", stop_words=[]):

    # To remove caps so to avoid e.g. This/this being seperate tokens
    x_raw = x_raw.lower()

    # Tokenise document
    x_raw = re.findall(token_pattern,  x_raw)

    # filter out stopwords
    tokens = [token for token in x_raw if token not in stop_words]

    # for each n-gram in ngram_range ---> construct a list of all n grams
    sequences = []
    for n in range(ngram_range[0], ngram_range[1] + 1) :
        sequences.extend([tuple(tokens[i:i + n]) for i in range(len(tokens) - n␣
↪+ 1)])



    return sequences
```

[307]:
```
extract_ngrams("this is a great movie to watch",
               ngram_range=(1,3),
               stop_words=stop_words)
```

[307]:
```
[('great',),
 ('movie',),
 ('watch',),
 ('great', 'movie'),
 ('movie', 'watch'),
 ('great', 'movie', 'watch')]
```

[308]:
```
extract_ngrams("this is a great movie to watch",
               ngram_range=(1,3),
               stop_words=stop_words)
```

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

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

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

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

2.1.2 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 - `vocab`: a given vocabulary. It should be used to extract specific features. - `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.

```
[310]: 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=[]):

           #  init empty values
           df = dict()
           vocab = set()
           ngram_counts = []

           for x in X_raw :
               # Get N-Grams for each document
               v = extract_ngrams(x, ngram_range = ngram_range, token_pattern =␣
       ↪token_pattern, stop_words = stop_words)

               # Add these ngrams to a list of words
               ngram_counts.extend(v)

               # Remove duplications and add to the df dict accordingly
               v = set(v)
               for item in list(v) :
                   df[item] = df.get(item, 0) + 1

               # add the ngrams to the vocab
               vocab |= v

           # Filter Document frequencies
           df = {k: v for (k, v) in df.items() if v >= min_df}
           # Keep only the top N
           df = dict(list({k: v for (k, v) in sorted(df.items(), reverse = True,␣
       ↪key=lambda x: x[1])}.items())[:keep_topN])

           # remove the filtered ngrams from the vocab
       #     vocab = vocab.intersection(set(df.keys()))
           vocab = set(df.keys())
           # count the number of ngrams in the vocab
           ngram_counts = [(x, y) for (x, y) in Counter(ngram_counts).items() if x in␣
       ↪vocab]
           return vocab, df, ngram_counts
```

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

[311]:
```
vocab, df, ngram_counts = get_vocab(data_sentiment_train_data,␣
↪ngram_range=(1,3), keep_topN=8000, stop_words=stop_words)
```

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

[312]:
```
# vocabulary id -> word and word -> id
v_id = {i: v for i, v in enumerate(vocab)}
w_id = {v: i for i, v in v_id.items()}
```

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

[313]:
```
tr_ngram_count = [extract_ngrams(x, stop_words=stop_words) for x in␣
↪data_sentiment_train_data]
dev_ngram_count = [extract_ngrams(x, stop_words=stop_words) for x in␣
↪data_sentiment_dev_data]
```

```
test_ngram_count = [extract_ngrams(x, stop_words=stop_words) for x in␣
 ↪data_sentiment_test_data]
```

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.

[314]:
```
def vectorise(X_ngram, vocab):
    # Get size of vocab and X_ngram
    V = len(vocab)
    X = len(X_ngram)

    # Create an empty matrix
    X_vec = np.zeros((X, V))

    # For each document construct a vector of size 1 x |V|
    for doc_id, doc in enumerate(X_ngram) :
        doc = [d for d in doc if d in vocab]
        for word in doc :
            X_vec[doc_id, w_id[word]] += 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

[315]:
```
# TRAINING COUNT MATRIX
X_tr_count = vectorise(tr_ngram_count, vocab)
# DEV COUNT MATRIX
X_dev_count = vectorise(dev_ngram_count, vocab)
# TEST COUNT MATRIX
X_test_count = vectorise(test_ngram_count, vocab)
```

[316]:
```
X_tr_count.shape
```

[316]:
```
(1400, 8000)
```

[317]:
```
X_tr_count[:1,:50]
```

[317]:
```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 2.,
        0., 0.]])
```

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

[318]:
```python
def tfidf(no_docs, df, v_id):
    # Get size of vocab and X_ngram
    V = len(v_id)
    # Create an empty matrix
    X_vec = np.zeros((1, V))
    # For each document construct a vector of size 1 x |V|
    for i in range(V) :
        X_vec[0, i] = np.log10(no_docs/df[v_id[i]])
    return X_vec
```

Then transform your count vectors to tf.idf vectors:

[319]:
```python
"""

Using the Count matrixes, enumerated vocabs, and the vocabs document frequencies
Calculate the TF.IDF for each value in the count matrixes

"""

no_docs = len(tr_ngram_count)

idf = tfidf(no_docs, df, v_id)

X_tr_tfidf = np.log10(1 + X_tr_count) * idf
X_dev_tfidf = np.log10(1 + X_dev_count) * idf
X_test_tfidf = np.log10(1 + X_test_count) * idf
```

[320]:
```python
X_tr_tfidf.shape
```

[320]:
```
(1400, 8000)
```

[321]:
```python
X_tr_tfidf[:1,:50]
```

[321]:
```
array([[0.59304016, 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.66698689, 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`

[322]:
```python
def sigmoid(z):
    z = 1 / (1 + np.exp(-z))
#     z = np.minimum(z, 0.9999)  # Set upper bound
#     z = np.maximum(z, 0.0001)  # Set lower bound
    return z
```

[323]:
```python
print(sigmoid(0))
print(sigmoid(np.array([-5., 1.2])))
```

```
0.5
[0.00669285 0.76852478]
```

[324]:
```python
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

[325]:
```python
def predict_proba(X, weights):
    # Use sigmoid to prodict the probability
    preds_proba = np.dot(weights, X.T)
    return sigmoid(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

[326]:
```
def predict_class(X, weights):
    # Return True if value is greater than 0.5 else 0
    preds_proba = predict_proba(X, weights)
    return preds_proba > 0.5
```

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

[327]:
```
def binary_loss(X, Y, weights, alpha=0.00001):
    # Sum the binary loss
    loss = 0
    for i, y in enumerate(Y) :
        prob = predict_proba(X[i], weights)
        loss += - np.log(prob) if y == 1 else - np.log(1 - prob)

    # Return the avg. loss and L2 regularisated
    return loss/len(Y) + alpha * np.dot(weights,weights.T)
```

[328]:
```
def binary_loss_dev(X, Y, weights, alpha=0.00001):

    # Calculate the gradient of the loss function
    prob = predict_proba(X, weights)
    loss = X * (prob - Y)

    #  return the gradient of the loss fuction
    return loss + 2 * alpha * weights
```

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

[329]:
```
def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], loss="binary", lr=0.1, alpha=0.00001,␣
↪epochs=50, tolerance=0.0001, print_progress=True):

    training_loss_history = []
    validation_loss_history = []

    #  Init weights with zeros
    weights = np.zeros(shape=(1, X_tr.shape[1]))

    # for each epoch
    for e in range(epochs) :

        # Shuffle documents
        s = np.arange(0, len(Y_tr))
        np.random.shuffle(s)

        # Shuffle the order of the docs
        X = np.array(X_tr)[s]
        Y = np.array(Y_tr)[s]

        # For each document in the training data descend in the gradient space ␣
↪
        for i, x in enumerate(X) :
            weights -= lr * binary_loss_dev(x, Y[i], weights,alpha=alpha)

        # Record the training and validation losses
        training_loss_history.extend(binary_loss(X_tr, Y_tr, weights,␣
↪alpha=alpha))
        validation_loss_history.extend(binary_loss(X_dev, Y_dev, weights,␣
↪alpha=alpha))

        if print_progress :
            print("Epoch:", e,"| Training loss:", training_loss_history[e],"|␣
↪Validation loss:", validation_loss_history[e])

        # Stop when the difference between the current and previous validation␣
↪loss is smaller than a threshold
        if e > 0 and np.abs(validation_loss_history[e - 1] -␣
↪validation_loss_history[e]) < 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:

[330]:
```
"""

A Naïve form of a grid-search to find good hyperparameters

"""
learning_rate = np.logspace(-1, -4, 4)
alpha = np.logspace(-1, -4, 4)

# Save best params in a hash
best = {"learning_rate" : 0, "Alpha" : 0, "f1" : 0}

# for each hyperparam evaluate the model
for rate in learning_rate :
    for al in alpha :
        w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr,
                                                    X_dev=X_dev_count,
                                                    Y_dev=Y_dev,
                                                    epochs=100,
                                                    lr=rate,
                                                    alpha=al,
                                                    print_progress=False)

        # Evaluate
        p = predict_class(X_test_count, w_count)
        f1 = f1_score(Y_test, p.squeeze() // 1)

        # if the F1 score is better than prev then replace
        if f1 > best["f1"]:
            best = {"learning_rate" : rate, "Alpha" : al, "f1" : f1}

print("Final", best)
```

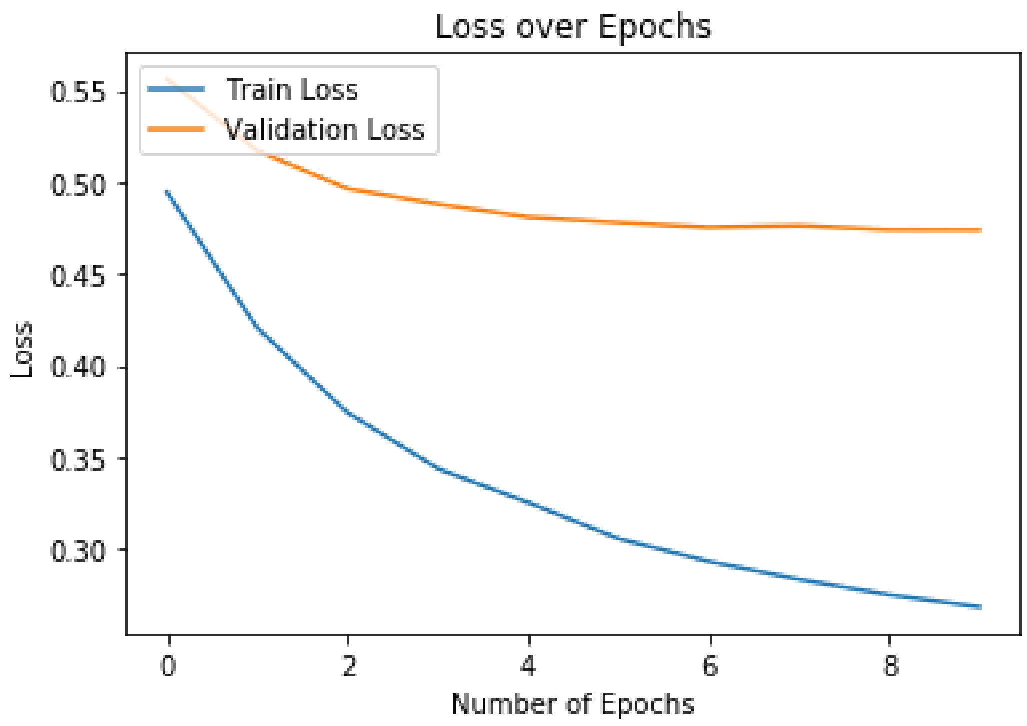
```
/Users/Jake/Documents/University/ComputerScience/YearFour/Text/env/lib/python3.7
/site-packages/ipykernel_launcher.py:6: RuntimeWarning: divide by zero
encountered in log

/Users/Jake/Documents/University/ComputerScience/YearFour/Text/env/lib/python3.7
/site-packages/ipykernel_launcher.py:32: RuntimeWarning: invalid value
encountered in subtract
```

```
Final {'learning_rate': 0.001, 'Alpha': 0.01, 'f1': 0.8564476885644767}
```

[331]:
```python
# Using optimal hyperparams train one last time
w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr,
                                             X_dev=X_dev_count,
                                             Y_dev=Y_dev,
                                             epochs=100,
                                                lr=best["learning_rate"],
                                                alpha=best["Alpha"])
```

```
Epoch: 0 | Training loss: [0.49440928] | Validation loss: [0.55627793]
Epoch: 1 | Training loss: [0.42064031] | Validation loss: [0.51709535]
Epoch: 2 | Training loss: [0.3744549] | Validation loss: [0.4964978]
Epoch: 3 | Training loss: [0.34402809] | Validation loss: [0.48802015]
Epoch: 4 | Training loss: [0.32568629] | Validation loss: [0.48100167]
Epoch: 5 | Training loss: [0.30595253] | Validation loss: [0.477951]
Epoch: 6 | Training loss: [0.29345698] | Validation loss: [0.47528786]
Epoch: 7 | Training loss: [0.28366447] | Validation loss: [0.47624076]
Epoch: 8 | Training loss: [0.27525469] | Validation loss: [0.47397944]
Epoch: 9 | Training loss: [0.26867319] | Validation loss: [0.47391554]
```

[332]:
```python
plt.plot(loss_tr_count,label='Train Loss')
plt.plot(dev_loss_count,label='Validation Loss')
plt.legend(loc="upper left")
plt.ylabel("Loss")
plt.xlabel("Number of Epochs")
plt.title("Loss over Epochs")
plt.show()
```



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

The model can be shown to be overfitting the data. This can be see by the differetn rate of loss between the validation and training losses. The rate of training loss is greater than that of the validation loss and the gap between plots increases overtime.

Compute accuracy, precision, recall and F1-scores:

[333]:
```
def conf(pred, labels) :
    N = len(np.unique(labels)) # Number of classes
    result = np.zeros((N, N))
    for i in range(len(Y_test)):
        result[labels[i]][pred[i]] += 1
    return result
```

[334]:
```
# Make predictions
preds_te_count = predict_class(X_test_count, w_count)
# Convert to 1d vector of 1/0
preds_te_count = preds_te_count.squeeze()//1
print("Confusion Matrix")
print(conf(preds_te_count, Y_test))
print('Accuracy:', accuracy_score(Y_test, preds_te_count))
print('Precision:', precision_score(Y_test, preds_te_count))
print('Recall:', recall_score(Y_test, preds_te_count))
print('F1-Score:', f1_score(Y_test, preds_te_count))
```

```
Confusion Matrix
[[161.  39.]
 [ 23. 177.]]
Accuracy: 0.845
Precision: 0.8194444444444444
Recall: 0.885
F1-Score: 0.8509615384615384
```

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

[335]:
```
pos = np.argsort(w_count)[0][::-1]
neg = np.argsort(w_count)[0]
print("Most Positive: ")
print([v_id[x] for x in pos[:10]])
print("Most Negative: ")
print([v_id[x] for x in neg[:10]])
```

```
Most Positive:
[('great',), ('fun',), ('hilarious',), ('life',), ('movies',), ('perfect',),
('american',), ('true',), ('perfectly',), ('simple',)]
Most Negative:
[('bad',), ('worst',), ('boring',), ('supposed',), ('plot',), ('script',),
('reason',), ('ridiculous',), ('stupid',), ('attempt',)]
```

comment on whether they make sense:

These Ngrams do make sense. The most positive words express positive sentiment such as “great” and “hilarious”. As well as the most negative words being “bad” and “worst”.

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?

In terms of generalisabilty, this model could be used in an other domains like laptop reviews. Features such as “great”, “simple”, and “perfect” for positive sentiment as well as “bad”, “worst”, and “boring” for negative sentiment would be picked up as important.

However, context specific features such as “movies” and “plot” suggests that the model has over-fitted to movies and could struggle in other domains.

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?

I chose my hyperparameters by using a very basic grid-search to go through a range of values (The ranges for both models were arbitrary for both models due to time taken to train the models). I went through each pair of values and evaluated its F1 score. I chose the pair that produced the highest F1 Score. If I had more time I would have tested more values as well as calculated an average for each pair to remove anomalous values.

Smaller learning rates require more training epochs as each update only changes the weights a little, whereas larger learning rates result in larger jumps so require fewer epochs.

A Regularisation value is a measure of the model's complexity. A Regularisation value encourages a model's weights, as well as its mean, to tend to zero. However, there is a trade-off with a large or small regularisation value. Too large and the model runs the risk of underfitting the data, and too small runs the risk of overfitting the data.

3.2 Train and Evaluate Logistic Regression with TF.IDF vectors

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

[336]:
```python
"""

A Naïve form of a grid-search to find good hyperparameters

"""

learning_rate = np.logspace(-2, -5, 4)
alpha = np.logspace(-2, -5, 4)

# Save best params in a hash
best = {"learning_rate" : 0, "Alpha" : 0, "f1" : 0}

# for each hyperparam evaluate the model
for rate in learning_rate :
    for al in alpha :

        w_tfidf, trl, devl = SGD(X_tr_tfidf, Y_tr,
                                        X_dev=X_dev_tfidf,
                                        Y_dev=Y_dev,
                                        epochs=100,
                                           lr=rate,
                                               alpha=al,
                                               print_progress=False)
        # Evaluate
        p = predict_class(X_test_tfidf, w_tfidf)
        f1 = f1_score(Y_test, p.squeeze() // 1)

        # if the F1 score is better than prev then replace
        if f1 > best["f1"]:
             best = {"learning_rate" : rate, "Alpha" : al, "f1" : f1}

print("Final", best)
```

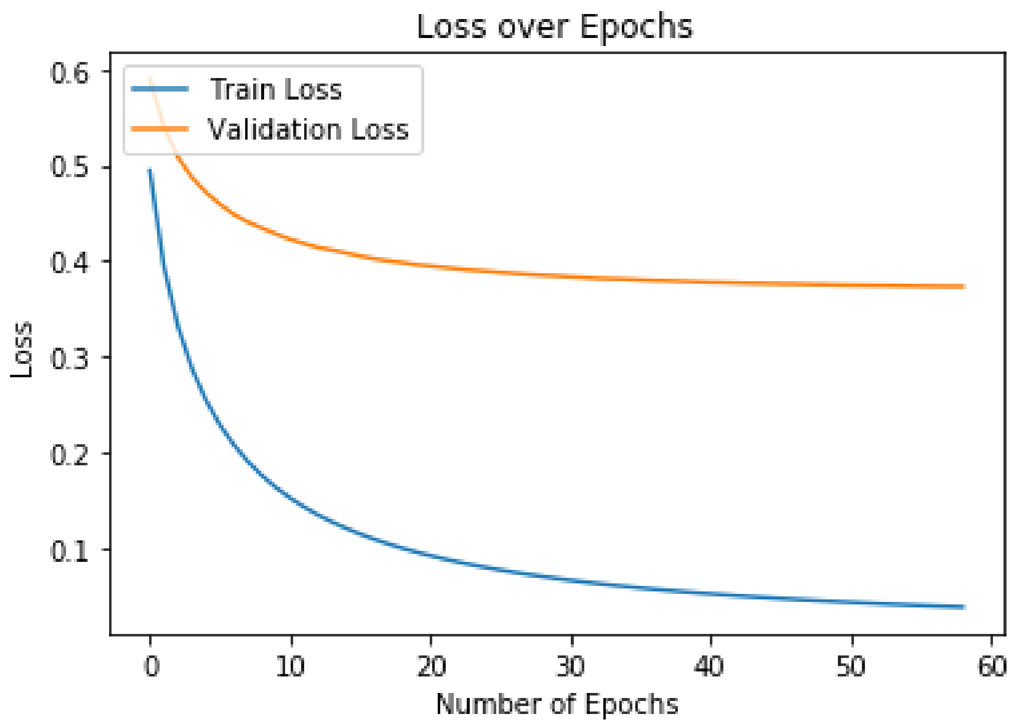
```
Final {'learning_rate': 0.01, 'Alpha': 1e-05, 'f1': 0.874074074074074}
```

[337]:
```
# Using optimal hyperparams train one last time on tf.idf
w_tfidf, trl, devl = SGD(X_tr_tfidf, Y_tr,
                         X_dev=X_dev_tfidf,
                         Y_dev=Y_dev,
                         epochs=200,
                         lr=best["learning_rate"],
                        alpha=best["Alpha"])
```

```
Epoch: 0 | Training loss: [0.49443372] | Validation loss: [0.59017414]
Epoch: 1 | Training loss: [0.39457887] | Validation loss: [0.53981015]
Epoch: 2 | Training loss: [0.33190666] | Validation loss: [0.50844918]
Epoch: 3 | Training loss: [0.28771364] | Validation loss: [0.48713464]
Epoch: 4 | Training loss: [0.25461255] | Validation loss: [0.47159972]
Epoch: 5 | Training loss: [0.22862585] | Validation loss: [0.45916746]
Epoch: 6 | Training loss: [0.20772113] | Validation loss: [0.44837382]
Epoch: 7 | Training loss: [0.19023266] | Validation loss: [0.44046194]
Epoch: 8 | Training loss: [0.1756368] | Validation loss: [0.43401405]
Epoch: 9 | Training loss: [0.16313011] | Validation loss: [0.42790424]
Epoch: 10 | Training loss: [0.15230997] | Validation loss: [0.42234792]
Epoch: 11 | Training loss: [0.14286022] | Validation loss: [0.4178983]
Epoch: 12 | Training loss: [0.13454466] | Validation loss: [0.41382591]
Epoch: 13 | Training loss: [0.12715717] | Validation loss: [0.41109577]
Epoch: 14 | Training loss: [0.12053985] | Validation loss: [0.40805409]
Epoch: 15 | Training loss: [0.11455517] | Validation loss: [0.40483978]
Epoch: 16 | Training loss: [0.10917467] | Validation loss: [0.4021367]
Epoch: 17 | Training loss: [0.10427701] | Validation loss: [0.39999584]
Epoch: 18 | Training loss: [0.09980991] | Validation loss: [0.39815892]
Epoch: 19 | Training loss: [0.0957222] | Validation loss: [0.39594923]
Epoch: 20 | Training loss: [0.09195364] | Validation loss: [0.39462861]
Epoch: 21 | Training loss: [0.08848183] | Validation loss: [0.39304649]
Epoch: 22 | Training loss: [0.08526809] | Validation loss: [0.39155136]
Epoch: 23 | Training loss: [0.08228693] | Validation loss: [0.3901756]
Epoch: 24 | Training loss: [0.07951716] | Validation loss: [0.38913503]
Epoch: 25 | Training loss: [0.07692843] | Validation loss: [0.38787668]
Epoch: 26 | Training loss: [0.07451233] | Validation loss: [0.38688637]
Epoch: 27 | Training loss: [0.07224774] | Validation loss: [0.38577783]
Epoch: 28 | Training loss: [0.07012372] | Validation loss: [0.38492448]
Epoch: 29 | Training loss: [0.06813031] | Validation loss: [0.38426708]
Epoch: 30 | Training loss: [0.06624712] | Validation loss: [0.38334607]
Epoch: 31 | Training loss: [0.06447263] | Validation loss: [0.38252192]
Epoch: 32 | Training loss: [0.06280071] | Validation loss: [0.3816052]
Epoch: 33 | Training loss: [0.0612116] | Validation loss: [0.38112042]
Epoch: 34 | Training loss: [0.05971016] | Validation loss: [0.38069648]
Epoch: 35 | Training loss: [0.058284] | Validation loss: [0.37999041]
Epoch: 36 | Training loss: [0.05693013] | Validation loss: [0.37948405]
Epoch: 37 | Training loss: [0.05564377] | Validation loss: [0.37910646]
```

```
Epoch: 38 | Training loss: [0.05441755] | Validation loss: [0.37860798]
Epoch: 39 | Training loss: [0.05324854] | Validation loss: [0.37807904]
Epoch: 40 | Training loss: [0.05213394] | Validation loss: [0.37757445]
Epoch: 41 | Training loss: [0.05106937] | Validation loss: [0.37729343]
Epoch: 42 | Training loss: [0.05005145] | Validation loss: [0.37687908]
Epoch: 43 | Training loss: [0.04907758] | Validation loss: [0.37648944]
Epoch: 44 | Training loss: [0.04814522] | Validation loss: [0.3762398]
Epoch: 45 | Training loss: [0.04725183] | Validation loss: [0.37597644]
Epoch: 46 | Training loss: [0.04639528] | Validation loss: [0.37574195]
Epoch: 47 | Training loss: [0.04557158] | Validation loss: [0.3754263]
Epoch: 48 | Training loss: [0.04478036] | Validation loss: [0.37511589]
Epoch: 49 | Training loss: [0.04401971] | Validation loss: [0.37478708]
Epoch: 50 | Training loss: [0.04328833] | Validation loss: [0.37454399]
Epoch: 51 | Training loss: [0.04258453] | Validation loss: [0.37438566]
Epoch: 52 | Training loss: [0.04190649] | Validation loss: [0.37419275]
Epoch: 53 | Training loss: [0.0412528] | Validation loss: [0.37400105]
Epoch: 54 | Training loss: [0.04062209] | Validation loss: [0.37378848]
Epoch: 55 | Training loss: [0.0400133] | Validation loss: [0.37349325]
Epoch: 56 | Training loss: [0.03942577] | Validation loss: [0.37336109]
Epoch: 57 | Training loss: [0.03885815] | Validation loss: [0.37318327]
Epoch: 58 | Training loss: [0.03830991] | Validation loss: [0.3731348]
```

[338]:
```python
# fill in your code...
plt.plot(trl,label='Train Loss')
plt.plot(devl,label='Validation Loss')
plt.legend(loc="upper left")
plt.ylabel("Loss")
plt.xlabel("Number of Epochs")
plt.title("Loss over Epochs")
plt.show()
```



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

The model can be shown to be overfitting the data. This can be see by the differetn rate of loss between the validation and training losses. The rate of training loss is greater than that of the validation loss and the gap between plots increases overtime.

Compute accuracy, precision, recall and F1-scores:

```
[339]: # Make Predictions
preds_te_tfidf = predict_class(X_test_tfidf, w_tfidf)
preds_te_tfidf = preds_te_tfidf.squeeze()//1

print("Confusion Matrix")
print(conf(preds_te_tfidf, Y_test))
print('Accuracy:', accuracy_score(Y_test, preds_te_tfidf))
print('Precision:', precision_score(Y_test, preds_te_tfidf))
print('Recall:', recall_score(Y_test, preds_te_tfidf))
print('F1-Score:', f1_score(Y_test, preds_te_tfidf))
```

```
Confusion Matrix
[[171.  29.]
 [ 23. 177.]]
Accuracy: 0.87
Precision: 0.8592233009708737
```

```
Recall: 0.885
F1-Score: 0.8719211822660098
```

Print top-10 most positive and negative words:

```
[340]: pos = np.argsort(w_tfidf)[0][::-1]
neg = np.argsort(w_tfidf)[0]
print("Most Positive: ")
print([v_id[x] for x in pos[:10]])
print("Most Negative: ")
print([v_id[x] for x in neg[:10]])
```

```
Most Positive:
[('hilarious',), ('terrific',), ('memorable',), ('excellent',), ('perfectly',),
('fun',), ('great',), ('enjoyed',), ('perfect',), ('superb',)]
Most Negative:
[('bad',), ('worst',), ('boring',), ('supposed',), ('waste',), ('ridiculous',),
('poor',), ('minute',), ('stupid',), ('fails',)]
```

comment on whether they make sense:

These words make sense. Furthermore, all these features express sentiment unlike the raw count features which epressed several features that were context specific to movie reviews

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?

These features suggest that this model could perform/generalise well in other domains such as laptop reviews as all these features express sentiment. For exaple, features like "excellent" and "worst".

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?

I chose my hyperparameters by using a very basic grid-search to go through a range of values (The ranges for both models were arbitrary for both models due to time taken to train the models). I went through each pair of values and evaluated its F1 score. I chose the pair that produced the highest F1 Score. If I had more time I would have tested more values as well as calculated an average for each pair to remove anomalous values.

Smaller learning rates require more training epochs as each update only changes the weights a little, whereas larger learning rates result in larger jumps so require fewer epochs.

A Regularisation value is a measure of the model's complexity. A Regularisation value encourages a model's weights, as well as its mean, to tend to zero. However, there is a trade-off with a large or small regularisation value. Too large and the model runs the risk of underfitting the data, and too small runs the risk of overfitting the data.

3.3 Full Results

Add here your results: to 3.s.f

LR	Precision	Recall	F1-Score
BOW-count	0.819	0.885	0.851
BOW-tfidf	0.859	0.885	0.872

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.

[341]:
```
# LOAD TRAINING DATA
data_topic_train_data = list(data_topic_train.loc[:,"Text"])
Y_tr = np.array(data_topic_train.loc[:,"Label"])

# LOAD TEST DATA
data_topic_test_data = list(data_topic_test.loc[:,"Text"])
Y_test = np.array(data_topic_test.loc[:,"Label"])

# LOAD VALIDATION DATA
data_topic_dev_data = list(data_topic_dev.loc[:,"Text"])
Y_dev = np.array(data_topic_dev.loc[:,"Label"])
```

[342]:
```
# Extract the new vocab, document frequencies, and counts
vocab, df, ngram_counts = get_vocab(data_topic_train_data, ngram_range=(1,3),␣
 ↪keep_topN=5000, stop_words=stop_words)

v_id = {i: v for i, v in enumerate(vocab)}
w_id = {v: i for i, v in v_id.items()}

# GET dicts for the data
tr_ngram_count = [extract_ngrams(x, stop_words=stop_words) for x in␣
 ↪data_topic_train_data]
dev_ngram_count = [extract_ngrams(x, stop_words=stop_words) for x in␣
 ↪data_topic_dev_data]
```

```
test_ngram_count = [extract_ngrams(x, stop_words=stop_words) for x in␣
 ↪data_topic_test_data]

# TRAINING COUNT MATRIX
X_tr_count = vectorise(tr_ngram_count, vocab)
# DEV COUNT MATRIX
X_dev_count = vectorise(dev_ngram_count, vocab)
# TEST COUNT MATRIX
X_test_count = vectorise(test_ngram_count, vocab)


no_docs = len(tr_ngram_count)
idf = tfidf(no_docs, df, v_id)
# Get tf.idf bag-of-words
X_tr_tfidf = np.log10(1 + X_tr_count) * idf
X_dev_tfidf = np.log10(1 + X_dev_count) * idf
X_test_tfidf = np.log10(1 + X_test_count) * idf
```

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`

[343]:
```
def softmax(z):
    # Subtract the max value in z to all elements
    # Too make the function numerically stable with large numbers
    z -= np.max(z)
    smax = np.exp(z) / np.sum(np.exp(z))

    return smax
```

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

[344]:
```
def predict_proba(X, weights):
    # Obtain the probabilties using softmax
    preds_proba = np.dot(X, weights.T)
    preds_proba = softmax(preds_proba)
    return preds_proba
```

[345]:
```
def predict_class(X, weights):
    # Predict the class using argmax
    preds_class = predict_proba(X, weights)
    pre = np.argmax(preds_class, axis=1)
    return pre
```

Toy example and expected functionality of the functions above:

[346]:
```
X = np.array([[0.1,0.2],[0.2,0.1],[0.1,-0.2]])
w = np.array([[2,-5],[-5,2]])

print(X.shape, w.shape)
```

```
(3, 2) (2, 2)
```

[347]:
```
predict_proba(X, w)
```

[347]:
```
array([[0.06982558, 0.14061145],
       [0.14061145, 0.06982558],
       [0.51594513, 0.0631808 ]])
```

[348]:
```
predict_class(X, w)
```

[348]:
```
array([1, 0, 0])
```

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

[349]:
```
def categorical_loss_dev(X, Y, weights, num_classes=5, alpha=0.00001):

    # init loss
    loss = np.zeros(shape=(weights.shape))
    # Calculate probabilities
    prob = predict_proba(X, weights)

    # Calculate the derivative of the loss
    loss[Y] = X * (prob[Y] - 1)

    # Apply the derived regulisation term to the loss
    return loss + 2 * alpha * weights
```

[350]:
```
def categorical_loss(X, Y, weights, num_classes=5, alpha=0.00001):

    # init loss
    loss = 0
    # Calculate the loss for each document in X
    for i, y in enumerate(Y) :
        prob = predict_proba(X[i], weights)
        loss += -np.log(prob[y])

    # Return the average loss + regulisation term
    return loss/len(Y) + alpha * np.sum(weights**2)
```

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

```
[351]: 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):


           training_loss_history = []
           validation_loss_history = []

           #  Init weights with zeros
           weights = np.zeros(shape=(num_classes, X_tr.shape[1]))

           # For each epoch
           for e in range(epochs) :

               # Shuffle documents
               s = np.arange(0, len(Y_tr))
               np.random.shuffle(s)

               # Shuffle the order of the docs
               X = np.array(X_tr)[s]
               Y = np.array(Y_tr)[s]

               # For each document calculate the gradient and descend
               for i, x in enumerate(X) :
                   weights -= lr * categorical_loss_dev(x, Y[i], weights, alpha=alpha,␣
        ↪num_classes=num_classes)

               # record losses
               training_loss_history.append(categorical_loss(X_tr, Y_tr, weights,␣
        ↪alpha=alpha, num_classes=num_classes))
               validation_loss_history.append(categorical_loss(X_dev, Y_dev, weights,␣
        ↪alpha=alpha, num_classes=num_classes))

               if print_progress :
                   print("Epoch:", e,"| Training loss:", training_loss_history[e],"|␣
        ↪Validation loss:", validation_loss_history[e])

               # Stop when the difference between the current and previous validation␣
        ↪loss is smaller than a threshold
               if e > 1 and np.abs(validation_loss_history[e - 1] -␣
        ↪validation_loss_history[e]) < 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:

[352]:
```
"""

A Naïve form of a grid-search to find good hyperparameters

"""
learning_rate = np.logspace(-1, -3, 3)
alpha = np.logspace(-4, -6, 3)

# Save best params in a hash
best = {"learning_rate" : 0, "Alpha" : 0, "f1" : 0}

# for each hyperparameter
for rate in learning_rate :
    for al in alpha :
        print(rate,al)
        w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr - 1,
                                                     X_dev=X_dev_count,
                                                     Y_dev=Y_dev - 1,
                                                     num_classes=3,
                                                     epochs=100,
                                                     lr=rate,
                                                     alpha=al,
                                                     print_progress=False)
        # Evaluate
        p = predict_class(X_test_count, w_count)
        f1 = f1_score(Y_test - 1, p.squeeze(), average='macro')

        if f1 > best["f1"]:
            best = {"learning_rate" : rate, "Alpha" : al, "f1" : f1}

print("Final", best)
```

```
0.1 0.0001
0.1 1e-05
0.1 1e-06
0.01 0.0001
0.01 1e-05
0.01 1e-06
0.001 0.0001
0.001 1e-05
0.001 1e-06
Final {'learning_rate': 0.001, 'Alpha': 0.0001, 'f1': 0.8688188195620731}
```

[353]:
```
# Use searched hyperparameters
w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr-1,
                                          X_dev=X_dev_count,
                                              Y_dev=Y_dev-1 ,
                                              num_classes=3,
                                              lr=best["learning_rate"],
                                              alpha=best["Alpha"],
                                              epochs=200)
```

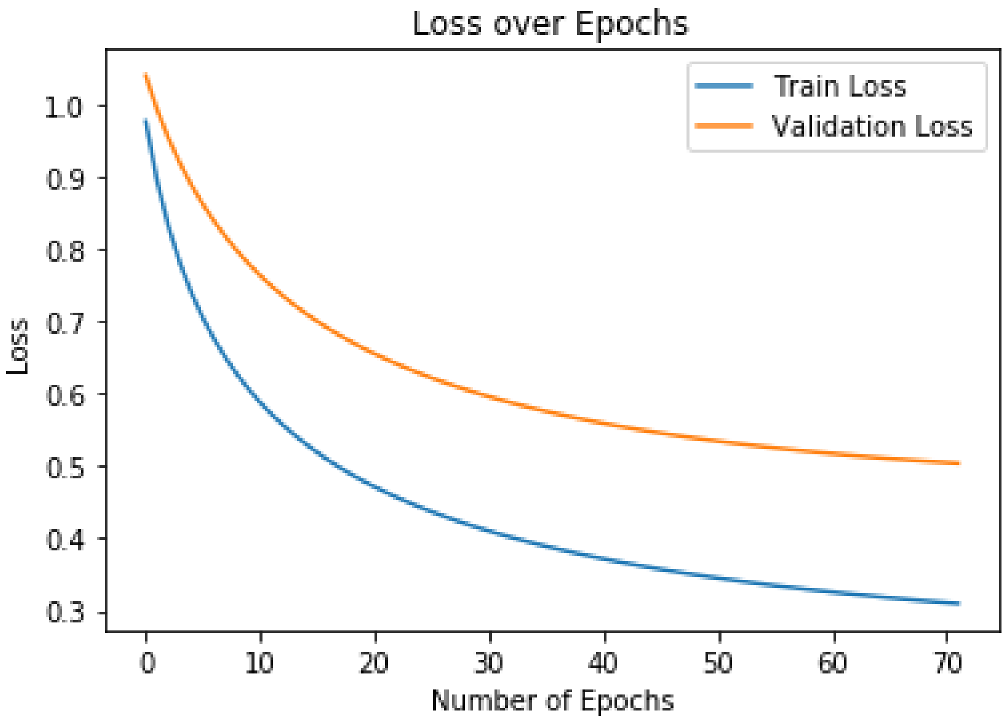
```
Epoch: 0 | Training loss: 0.9757768363812795 | Validation loss:
1.0386695889431747
Epoch: 1 | Training loss: 0.8926981578077007 | Validation loss:
0.9915021094077285
Epoch: 2 | Training loss: 0.8297093908700244 | Validation loss:
0.9517713198533554
Epoch: 3 | Training loss: 0.7796362357383748 | Validation loss:
0.9175156363086793
Epoch: 4 | Training loss: 0.7385052031713584 | Validation loss:
0.8874697731130149
Epoch: 5 | Training loss: 0.7039765172202689 | Validation loss: 0.86092390272327
Epoch: 6 | Training loss: 0.674362156427306 | Validation loss:
0.8372038441068257
Epoch: 7 | Training loss: 0.6485951027460155 | Validation loss:
0.8158823529934582
Epoch: 8 | Training loss: 0.6259005925214388 | Validation loss:
0.7965818506770842
Epoch: 9 | Training loss: 0.6056955285733022 | Validation loss:
0.7790102780790952
Epoch: 10 | Training loss: 0.5875520023977826 | Validation loss:
0.7629329082673548
Epoch: 11 | Training loss: 0.571147567731156 | Validation loss:
0.7481632686067579
Epoch: 12 | Training loss: 0.5562230613233325 | Validation loss:
0.7345681961078294
Epoch: 13 | Training loss: 0.5425634481864647 | Validation loss:
0.7219771768727237
Epoch: 14 | Training loss: 0.5299983795436652 | Validation loss:
0.710297982724522
Epoch: 15 | Training loss: 0.5183992158439101 | Validation loss:
0.6994266467590373
Epoch: 16 | Training loss: 0.5076461262797639 | Validation loss:
0.6892846569821308
Epoch: 17 | Training loss: 0.49763749546464553 | Validation loss:
0.6797859302332458
Epoch: 18 | Training loss: 0.488299800181027 | Validation loss:
0.670882930581769
Epoch: 19 | Training loss: 0.47956216412604924 | Validation loss:
0.662507277698234
Epoch: 20 | Training loss: 0.47136616600328235 | Validation loss:
0.6546287063841546
```

```
Epoch: 21 | Training loss: 0.4636565575799837 | Validation loss:
0.6471996932318509
Epoch: 22 | Training loss: 0.4563872216887594 | Validation loss:
0.6401791665434156
Epoch: 23 | Training loss: 0.44952113305255276 | Validation loss:
0.6335404537655659
Epoch: 24 | Training loss: 0.44302626168726916 | Validation loss:
0.627250057168018
Epoch: 25 | Training loss: 0.43686753628120295 | Validation loss:
0.6212780802892484
Epoch: 26 | Training loss: 0.4310214800895976 | Validation loss:
0.6155984027738897
Epoch: 27 | Training loss: 0.42546145162986665 | Validation loss:
0.6102094318261749
Epoch: 28 | Training loss: 0.4201642920785155 | Validation loss:
0.605077083564933
Epoch: 29 | Training loss: 0.4151157185502461 | Validation loss:
0.6001783220025906
Epoch: 30 | Training loss: 0.41029588278967677 | Validation loss:
0.5955073530250296
Epoch: 31 | Training loss: 0.40568724271148693 | Validation loss:
0.5910499608324598
Epoch: 32 | Training loss: 0.4012787491394392 | Validation loss:
0.5868000944303243
Epoch: 33 | Training loss: 0.39705622011091496 | Validation loss:
0.5827292819688454
Epoch: 34 | Training loss: 0.3930071350327689 | Validation loss:
0.5788342133591815
Epoch: 35 | Training loss: 0.389121197440541 | Validation loss:
0.5751076039469322
Epoch: 36 | Training loss: 0.3853882646431746 | Validation loss:
0.5715296711809646
Epoch: 37 | Training loss: 0.3817991876258306 | Validation loss:
0.5680938210855068
Epoch: 38 | Training loss: 0.37834612093075054 | Validation loss:
0.564801196235339
Epoch: 39 | Training loss: 0.37502281001999915 | Validation loss:
0.5616427484405929
Epoch: 40 | Training loss: 0.3718200173710681 | Validation loss:
0.5586096683872085
Epoch: 41 | Training loss: 0.3687327938239101 | Validation loss:
0.5556927284252078
Epoch: 42 | Training loss: 0.3657527316790326 | Validation loss:
0.5528885978196162
Epoch: 43 | Training loss: 0.3628773250196926 | Validation loss:
0.5501933805219933
Epoch: 44 | Training loss: 0.3600983214583877 | Validation loss:
0.5475986963984021
```

```
Epoch: 45 | Training loss: 0.3574135316175244 | Validation loss:
0.5451040354905772
Epoch: 46 | Training loss: 0.35481733588983594 | Validation loss:
0.5426972606058489
Epoch: 47 | Training loss: 0.35230528382959725 | Validation loss:
0.5403790770912509
Epoch: 48 | Training loss: 0.34987414823039803 | Validation loss:
0.538149271452474
Epoch: 49 | Training loss: 0.3475208077796515 | Validation loss:
0.5359952285679587
Epoch: 50 | Training loss: 0.34524115000901034 | Validation loss:
0.5339220879566348
Epoch: 51 | Training loss: 0.3430306997734499 | Validation loss:
0.531922851060731
Epoch: 52 | Training loss: 0.34088842270737246 | Validation loss:
0.5299950278358285
Epoch: 53 | Training loss: 0.3388098483929065 | Validation loss:
0.5281298719270948
Epoch: 54 | Training loss: 0.3367932238510872 | Validation loss:
0.5263376866816204
Epoch: 55 | Training loss: 0.33483620082631904 | Validation loss:
0.5246057443478324
Epoch: 56 | Training loss: 0.33293471499723915 | Validation loss:
0.5229245568274633
Epoch: 57 | Training loss: 0.3310887521975759 | Validation loss:
0.5213086035020591
Epoch: 58 | Training loss: 0.32929496367678096 | Validation loss:
0.5197517178759111
Epoch: 59 | Training loss: 0.32755075394374106 | Validation loss:
0.5182459209049256
Epoch: 60 | Training loss: 0.32585655183904483 | Validation loss:
0.516795803934853
Epoch: 61 | Training loss: 0.32420732912067146 | Validation loss:
0.5153886778742626
Epoch: 62 | Training loss: 0.32260326276390083 | Validation loss:
0.5140322706916564
Epoch: 63 | Training loss: 0.32104217284734626 | Validation loss:
0.5127201705288565
Epoch: 64 | Training loss: 0.3195241251362389 | Validation loss:
0.511461206014645
Epoch: 65 | Training loss: 0.3180456034071637 | Validation loss:
0.5102488893294702
Epoch: 66 | Training loss: 0.3166057795548078 | Validation loss:
0.509073254141149
Epoch: 67 | Training loss: 0.31520338264928577 | Validation loss:
0.507933405412588
Epoch: 68 | Training loss: 0.313837109834145 | Validation loss:
0.5068314881731403
```

```
Epoch: 69 | Training loss: 0.31250649775470213 | Validation loss:
0.5057717861702363
Epoch: 70 | Training loss: 0.3112100452323182 | Validation loss:
0.5047530150965296
Epoch: 71 | Training loss: 0.30994545681713714 | Validation loss:
0.5037596134647986
```

[354]:
```
plt.plot(loss_tr_count,label='Train Loss')
plt.plot(dev_loss_count,label='Validation Loss')
plt.legend(loc="upper right")
plt.ylabel("Loss")
plt.xlabel("Number of Epochs")
plt.title("Loss over Epochs")

plt.show()
```



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

This graph suggests that the model is overfitting a tiny bit. This can be see from the fact that both plots rates of loss look to be similar with only a little bit of seperation. It also suggests that that the model capacity isnt high enough so to improve this model would be to increase the number of parameters

Compute accuracy, precision, recall and F1-scores:

[355]:
```
# predict
preds_te = predict_class(X_test_count, w_count).squeeze()

print("Confusion Matix")
print(conf(preds_te, Y_test - 1))
print('Accuracy:', accuracy_score(Y_test - 1, preds_te))
print('Precision:', precision_score(Y_test - 1, preds_te, average='macro'))
print('Recall:', recall_score(Y_test - 1, preds_te, average='macro'))
print('F1-Score:', f1_score(Y_test - 1, preds_te, average='macro'))
```

```
Confusion Matix
[[260.  18.  22.]
 [ 14. 281.   5.]
 [ 51.   8. 241.]]
Accuracy: 0.8688888888888889
Precision: 0.8715210591991184
Recall: 0.8688888888888888
F1-Score: 0.8688188195620731
```

Print the top-10 words for each class respectively.

[356]:
```
one = np.argsort(w_count)[0][::-1]
two = np.argsort(w_count)[1][::-1]
three = np.argsort(w_count)[2][::-1]
print("Class One: ")
print([v_id[x] for x in one][:10])
print()
print("Class Two: ")
print([v_id[x] for x in two][:10])
print()
print("Class Three: ")
print([v_id[x] for x in three][:10])
```

```
Class One:
[('reuters',), ('athens',), ('oil',), ('afp',), ('president',), ('prices',),
('greece',), ('olympic',), ('athens', 'greece'), ('united',)]

Class Two:
[('athens',), ('reuters',), ('olympic',), ('team',), ('night',), ('win',),
('gold',), ('games',), ('olympics',), ('year',)]

Class Three:
[('reuters',), ('oil',), ('prices',), ('company',), ('york',), ('year',),
('percent',), ('quot',), ('oil', 'prices'), ('billion',)]
```

comment on whether they make sense: These features make sense. Class One includes features related to "World"; Class Two includes features related to "Sports"; and Class Three includes features related to "Business". However, there are a lot of overlap accross classes. For example the feature "reuters" is in the top 2 for all classes.

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?

In terms of generalisability, this model would act very poorly as the model has been trained with only three specific topics. If a new topic is added, such as technology, then a new model would have to be trained to allow for this increase of classifier output otherwise misclassification will occur. However, in terms of classifing other news items from different sources could perform well.

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?

I chose my hyperparameters by using a very basic grid-search to go through a range of values (The ranges for both models were arbitrary for both models due to time taken to train the models). I went through each pair of values and evaluated its F1 score. I chose the pair that produced the highest F1 Score. If I had more time I would have tested more values as well as calculated an average for each pair to remove anomalous values.

Smaller learning rates require more training epochs as each update only changes the weights a little, whereas larger learning rates result in larger jumps so require fewer epochs.

A Regularisation value is a measure of the model's complexity. A Regularisation value encourages a model's weights, as well as its mean, to tend to zero. However, there is a trade-off with a large or small regularisation value. Too large and the model runs the risk of underfitting the data, and too small runs the risk of overfitting the data.

[ ]:

```
```

[357]:

```
# Init hyperparam space
learning_rate = np.logspace(-1, -3, 3)
alpha = np.logspace(-4, -6, 3)

# Save best hyperparams in hash
best = {"learning_rate" : 0, "Alpha" : 0, "f1" : 0}

# for each hyperparam
for rate in learning_rate :
    for al in alpha :
        print(rate,al)
        w_tfidf, trl, devl = SGD(X_tr_tfidf, Y_tr - 1,
                                 X_dev=X_dev_tfidf,
                                 Y_dev=Y_dev - 1,
                                 epochs=100,
                                 lr = rate,
                                 alpha=al,
                                print_progress=False)
        # Evaluate
        p = predict_class(X_test_tfidf, w_tfidf)
        f1 = f1_score(Y_test - 1, p.squeeze(), average='macro')
        if f1 > best["f1"]:
            best = {"learning_rate" : rate, "Alpha" : al, "f1" : f1}

print("Final", best)
```

```
0.1 0.0001
0.1 1e-05
0.1 1e-06
0.01 0.0001
0.01 1e-05
0.01 1e-06
0.001 0.0001
0.001 1e-05
0.001 1e-06
Final {'learning_rate': 0.001, 'Alpha': 0.0001, 'f1': 0.8815144683496156}
```

[358]:
```
w_tfidf, trl, devl = SGD(X_tr_tfidf, Y_tr - 1,
                         X_dev=X_dev_tfidf,
                         Y_dev=Y_dev - 1,
                         lr = best["learning_rate"],
                         alpha=best["Alpha"],
                         epochs=100)
```

```
Epoch: 0 | Training loss: 1.5492263439697442 | Validation loss:
1.5768520621001325
Epoch: 1 | Training loss: 1.4934446766651297 | Validation loss:
1.5457703268156944
Epoch: 2 | Training loss: 1.4419061909969957 | Validation loss:
1.5161481322155956
Epoch: 3 | Training loss: 1.3942758378850744 | Validation loss:
1.4879227746088652
Epoch: 4 | Training loss: 1.3501164632679352 | Validation loss:
1.461004407356531
Epoch: 5 | Training loss: 1.3090097042466897 | Validation loss:
1.4352855721102482
Epoch: 6 | Training loss: 1.2706538662217488 | Validation loss:
1.410697284381645
Epoch: 7 | Training loss: 1.2347668122281623 | Validation loss:
1.387161985251461
Epoch: 8 | Training loss: 1.2011057480990865 | Validation loss:
1.3646076730633991
Epoch: 9 | Training loss: 1.1694880963994736 | Validation loss:
1.3429824160905175
Epoch: 10 | Training loss: 1.139736639195379 | Validation loss:
1.3222297140437207
Epoch: 11 | Training loss: 1.111701414481293 | Validation loss:
```

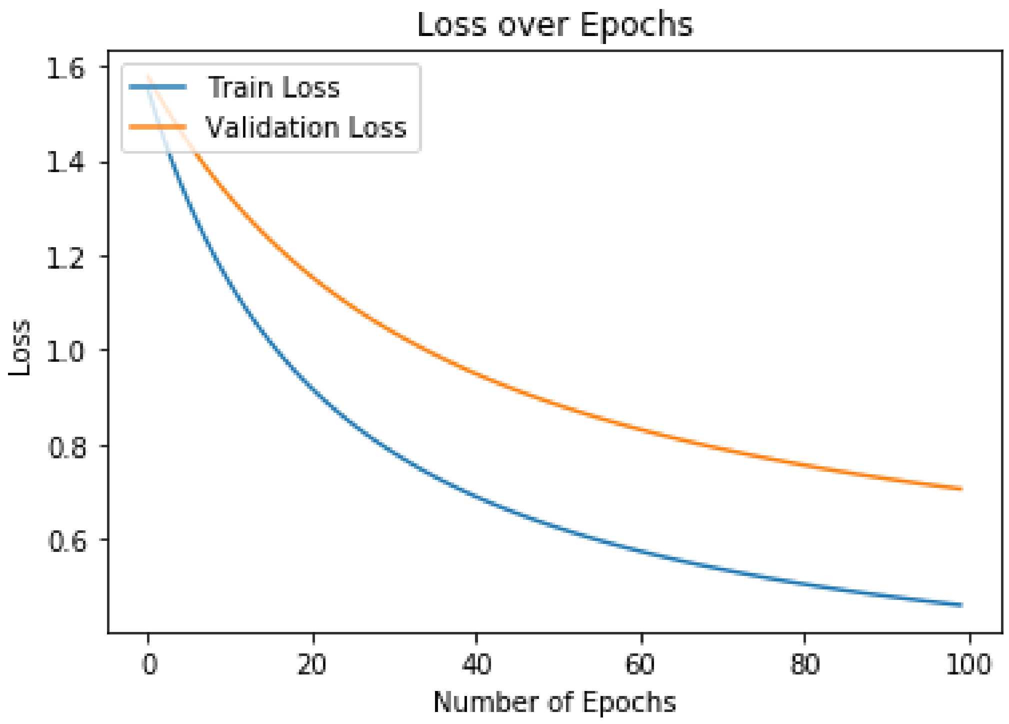
```
1.302309301193003
Epoch: 12 | Training loss: 1.0852503197814993 | Validation loss:
1.2831750097261077
Epoch: 13 | Training loss: 1.0602621915342387 | Validation loss:
1.2647817710127531
Epoch: 14 | Training loss: 1.0366264805401315 | Validation loss:
1.247094126702406
Epoch: 15 | Training loss: 1.0142382934006884 | Validation loss:
1.230073693598378
Epoch: 16 | Training loss: 0.9930095671646916 | Validation loss:
1.2136855969767881
Epoch: 17 | Training loss: 0.9728579881877658 | Validation loss:
1.1978947082923954
Epoch: 18 | Training loss: 0.9537102344077594 | Validation loss:
1.1826795523471414
Epoch: 19 | Training loss: 0.935493635496598 | Validation loss:
1.1680068333704379
Epoch: 20 | Training loss: 0.9181448416868335 | Validation loss:
1.1538502358239184
Epoch: 21 | Training loss: 0.9016029851418722 | Validation loss:
1.1401830858690587
Epoch: 22 | Training loss: 0.8858210013812645 | Validation loss:
1.126987179567643
Epoch: 23 | Training loss: 0.8707482111779292 | Validation loss:
1.1142369681187172
Epoch: 24 | Training loss: 0.856336193740278 | Validation loss:
1.1019095768819687
Epoch: 25 | Training loss: 0.8425450453106949 | Validation loss:
1.0899857674457014
Epoch: 26 | Training loss: 0.8293383559087659 | Validation loss:
1.0784496327921889
Epoch: 27 | Training loss: 0.8166785898136837 | Validation loss:
1.0672843603554998
Epoch: 28 | Training loss: 0.8045333907802409 | Validation loss:
1.056469048909277
Epoch: 29 | Training loss: 0.7928718418678665 | Validation loss:
1.0459896391871142
Epoch: 30 | Training loss: 0.7816686331374361 | Validation loss:
1.0358352091543506
Epoch: 31 | Training loss: 0.7708952851887872 | Validation loss:
1.0259874123309318
Epoch: 32 | Training loss: 0.7605309292923061 | Validation loss:
1.016436314262821
Epoch: 33 | Training loss: 0.7505510822415856 | Validation loss:
1.0071654031017043
Epoch: 34 | Training loss: 0.740934483933626 | Validation loss:
0.9981639693106625
Epoch: 35 | Training loss: 0.7316626175278667 | Validation loss:
```

```
0.9894228759912945
Epoch: 36 | Training loss: 0.7227175234487433 | Validation loss:
0.9809315755666823
Epoch: 37 | Training loss: 0.7140814780130711 | Validation loss:
0.9726786460556301
Epoch: 38 | Training loss: 0.7057409001854591 | Validation loss:
0.9646558149235983
Epoch: 39 | Training loss: 0.697678973624688 | Validation loss:
0.9568506257620255
Epoch: 40 | Training loss: 0.6898825462044585 | Validation loss:
0.9492582045181289
Epoch: 41 | Training loss: 0.6823394971235963 | Validation loss:
0.9418684338333367
Epoch: 42 | Training loss: 0.6750373902023137 | Validation loss:
0.9346742340789356
Epoch: 43 | Training loss: 0.6679646707690712 | Validation loss:
0.9276682440042532
Epoch: 44 | Training loss: 0.6611114263227637 | Validation loss:
0.9208436422011366
Epoch: 45 | Training loss: 0.6544668846354269 | Validation loss:
0.9141928604661033
Epoch: 46 | Training loss: 0.6480223548808712 | Validation loss:
0.9077110792105757
Epoch: 47 | Training loss: 0.641767958692066 | Validation loss:
0.9013899806254045
Epoch: 48 | Training loss: 0.6356959610804402 | Validation loss:
0.8952266969306745
Epoch: 49 | Training loss: 0.6297990297798532 | Validation loss:
0.8892121424479161
Epoch: 50 | Training loss: 0.6240693871253401 | Validation loss:
0.8833446137716755
Epoch: 51 | Training loss: 0.6184995967503993 | Validation loss:
0.877616484842943
Epoch: 52 | Training loss: 0.6130832565975577 | Validation loss:
0.8720239125721848
Epoch: 53 | Training loss: 0.6078146986810018 | Validation loss:
0.8665626541809875
Epoch: 54 | Training loss: 0.6026875296060286 | Validation loss:
0.8612281582088435
Epoch: 55 | Training loss: 0.597696475392183 | Validation loss:
0.8560165790581605
Epoch: 56 | Training loss: 0.592835799855437 | Validation loss:
0.850923022998189
Epoch: 57 | Training loss: 0.588100418929732 | Validation loss:
0.8459437569387233
Epoch: 58 | Training loss: 0.5834862748641484 | Validation loss:
0.8410757312576824
Epoch: 59 | Training loss: 0.5789885059352826 | Validation loss:
```

```
0.8363148206026253
Epoch: 60 | Training loss: 0.5746023719271213 | Validation loss:
0.8316585956703529
Epoch: 61 | Training loss: 0.5703240471670863 | Validation loss:
0.8271033813525177
Epoch: 62 | Training loss: 0.5661501860579996 | Validation loss:
0.82264552969499
Epoch: 63 | Training loss: 0.5620763413992208 | Validation loss:
0.8182822779639646
Epoch: 64 | Training loss: 0.5580989581535233 | Validation loss:
0.8140109034952479
Epoch: 65 | Training loss: 0.5542149682786474 | Validation loss:
0.8098291884792723
Epoch: 66 | Training loss: 0.5504206333918603 | Validation loss:
0.8057335454905515
Epoch: 67 | Training loss: 0.546713371761394 | Validation loss:
0.8017225475258523
Epoch: 68 | Training loss: 0.5430902706615734 | Validation loss:
0.7977930622908432
Epoch: 69 | Training loss: 0.539548386876289 | Validation loss:
0.7939425799844174
Epoch: 70 | Training loss: 0.5360853297085387 | Validation loss:
0.7901693242041409
Epoch: 71 | Training loss: 0.5326984911072034 | Validation loss:
0.7864709248698198
Epoch: 72 | Training loss: 0.5293854590780266 | Validation loss:
0.7828447675363988
Epoch: 73 | Training loss: 0.5261435964332478 | Validation loss:
0.7792901570138613
Epoch: 74 | Training loss: 0.5229710217538038 | Validation loss:
0.7758043080414423
Epoch: 75 | Training loss: 0.5198652557111186 | Validation loss:
0.7723850279563681
Epoch: 76 | Training loss: 0.5168240679282324 | Validation loss:
0.7690312873725416
Epoch: 77 | Training loss: 0.5138456352816719 | Validation loss:
0.7657407736609707
Epoch: 78 | Training loss: 0.5109286172466018 | Validation loss:
0.7625116165334487
Epoch: 79 | Training loss: 0.508070692122886 | Validation loss:
0.7593428609213101
Epoch: 80 | Training loss: 0.5052703650783112 | Validation loss:
0.756232815204053
Epoch: 81 | Training loss: 0.5025256640755481 | Validation loss:
0.7531799015507121
Epoch: 82 | Training loss: 0.49983478930045183 | Validation loss:
0.7501829805561502
Epoch: 83 | Training loss: 0.4971970922644877 | Validation loss:
```

```
0.7472395249210357
Epoch: 84 | Training loss: 0.4946105828319703 | Validation loss:
0.7443491513954731
Epoch: 85 | Training loss: 0.49207361897141 | Validation loss:
0.7415103568794926
Epoch: 86 | Training loss: 0.4895848126693119 | Validation loss:
0.7387215238706494
Epoch: 87 | Training loss: 0.487143211315364 | Validation loss:
0.7359816908503234
Epoch: 88 | Training loss: 0.48474726505348364 | Validation loss:
0.7332900844519596
Epoch: 89 | Training loss: 0.48239618628343955 | Validation loss:
0.7306449080385496
Epoch: 90 | Training loss: 0.48008832631453535 | Validation loss:
0.7280454316593122
Epoch: 91 | Training loss: 0.4778223071298232 | Validation loss:
0.7254909306066991
Epoch: 92 | Training loss: 0.47559767266682207 | Validation loss:
0.7229800180433963
Epoch: 93 | Training loss: 0.47341320800863934 | Validation loss:
0.7205112727502971
Epoch: 94 | Training loss: 0.4712676656070854 | Validation loss:
0.7180837403302598
Epoch: 95 | Training loss: 0.46915985839087293 | Validation loss:
0.7156969397220809
Epoch: 96 | Training loss: 0.4670893936183737 | Validation loss:
0.7133500613998939
Epoch: 97 | Training loss: 0.465055037413056 | Validation loss:
0.7110417758078632
Epoch: 98 | Training loss: 0.46305594385827775 | Validation loss:
0.7087713753477446
Epoch: 99 | Training loss: 0.4610914881749136 | Validation loss:
0.7065373937302709
```

[359]:
```
plt.plot(trl,label='Train Loss')
plt.plot(devl,label='Validation Loss')
plt.legend(loc="upper left")
plt.ylabel("Loss")
plt.xlabel("Number of Epochs")
plt.title("Loss over Epochs")

plt.show()
```



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

This graph suggests that the model is overfitting a tiny bit. This can be see from the fact that both plots rates of loss look to be similar with only a little bit of seperation. It also suggests that that the model capacity isnt high enough so to improve this model would be to increase the number of parameters

[360]:
```
preds_te = predict_class(X_test_tfidf, w_tfidf).squeeze()

print("Confusion Matix")
print(conf(preds_te, Y_test - 1))

print('Accuracy:', accuracy_score(Y_test - 1, preds_te))
print('Precision:', precision_score(Y_test - 1, preds_te, average='macro'))
print('Recall:', recall_score(Y_test - 1, preds_te, average='macro'))
print('F1-Score:', f1_score(Y_test - 1, preds_te, average='macro'))
```

```
Confusion Matix
[[258.  21.  21.]
 [  7. 290.   3.]
 [ 45.   9. 246.]]
Accuracy: 0.8822222222222222
Precision: 0.8832063918757468
Recall: 0.8822222222222221
F1-Score: 0.8815144683496156
```

[361]:
```python
one = np.argsort(w_tfidf)[0][::-1]
two = np.argsort(w_tfidf)[1][::-1]
three = np.argsort(w_tfidf)[2][::-1]
print("Class One: ")
print([v_id[x] for x in one][:10])
print()
print("Class Two: ")
print([v_id[x] for x in two][:10])
print()
print("Class Three: ")

print([v_id[x] for x in three][:10])
```

```
Class One:
[('afp',), ('president',), ('reuters',), ('people',), ('government',),
('city',), ('iraq',), ('officials',), ('minister',), ('united',)]

Class Two:
[('athens',), ('olympic',), ('team',), ('night',), ('games',), ('olympics',),
('win',), ('gold',), ('athens', 'reuters'), ('season',)]

Class Three:
[('company',), ('oil',), ('reuters',), ('prices',), ('york',), ('percent',),
('billion',), ('corp',), ('business',), ('sales',)]
```

comment on whether they make sense: These features make sense. Class One includes features related to “World”; Class Two includes features related to “Sports”; and Class Three includes features related to “Business”.

Compared to the raw count features these features have a lot less overlap and each class seems to be a lot more context specific. This can be seen in the increased F1-score.

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?

In terms of generalisability, this model would act very poorly as the model has been trained with only three specific topics. If a new topic is added, such as technology, then a new model would have to be trained to allow for this increase of classifier output otherwise misclassification will occur. However, in terms of classifing other news items from different sources could perform well.

4.0.2 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?

I chose my hyperparameters by using a very basic grid-search to go through a range of values (The ranges for both models were arbitrary for both models due to time taken to train the models). I went through each pair of values and evaluated its F1 score. I chose the pair that produced the highest F1 Score. If I had more time I would have tested more values as well as calculated an average for each pair to remove anomalous values.

Smaller learning rates require more training epochs as each update only changes the weights a little, whereas larger learning rates result in larger jumps so require fewer epochs.

A Regularisation value is a measure of the model's complexity. A Regularisation value encourages a model's weights, as well as its mean, to tend to zero. However, there is a trade-off with a large or small regularisation value. Too large and the model runs the risk of underfitting the data, and too small runs the risk of overfitting the data.

4.1 Full Results

Add here your results: to 3s.f.

LR	Precision	Recall	F1-Score
BOW-count	0.872	0.869	0.869
BOW-tfidf	0.883	0.882	0.882

[362]:
```
current_time = datetime.now() - now
print("Total Time =", current_time)
```

```
Total Time = 0:15:42.337432
```

[ ]:
```
```

[ ]:
```
```

[ ]:
```
```