[COM6513] Assignment 1: Sentiment Analysis with Logistic Regression

Instructor: Nikos Aletras

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

For that purpose, you will implement:

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

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

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

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

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

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

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

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

Data

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

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

Submission Instructions

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

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

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

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

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

In [1]:

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

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

Load Raw texts and labels into arrays

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

In [2]:

```
dev_data = pd.read_csv('data_sentiment/dev.csv')        # Reading CSV file
dev_data.columns=['Text','Label']                       # Column names are given here
test_data = pd.read_csv('data_sentiment/test.csv')
test_data.columns = ['Text','Label']
train_data = pd.read_csv('data_sentiment/train.csv')
train_data.columns = ['Text','Label']
```

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

In [3]:

```
dev_data.head()
```

Out[3]:

	Text	Label
0	wong kar-wei's " fallen angels " is , on a pur...	1
1	there is nothing like american history x in th...	1
2	an unhappy italian housewife , a lonely waiter...	1
3	when people are talking about good old times ,...	1
4	the rocky horror picture show 'special edition...	1

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

In [4]:

```
dev_text = list(dev_data['Text'])         # Seprating text and label data
dev_label = list(dev_data['Label'])
train_text = list(train_data['Text'])
train_label = list(train_data['Label'])
test_text = list(test_data['Text'])
test_label = list(test_data['Label'])
```

Vector Representations of Text

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

Text Pre-Processing Pipeline

To obtain a vocabulary of features, you should:

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

In [5]:

```
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']
```

N-gram extraction from a document

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

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

and returns:

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

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

In [6]:

```
def extract_ngrams(x_raw, ngram_range=(1,3), token_pattern=r'\b[A-Za-z]{2,}\b',
                   stop_words= stop_words, vocab=None, char_ngrams = True):
    if char_ngrams == False:
        tokens = []
        for word in re.findall(token_pattern,x_raw):
            if word.lower() not in stop_words:
                tokens.append(word.lower())

        ngrams_list = []
                                                       # Extracting tokens by words
        for num in range(0, len(tokens)):
            ngram = ' '.join(tokens[num:num + 1])
            ngrams_list.append(ngram)

        for num in range(0, len(tokens)):
            ngram = ' '.join(tokens[num:num + 2])
            ngrams_list.append(ngram)

        for num in range(0, len(tokens)):
            ngram = ' '.join(tokens[num:num + 3])
            ngrams_list.append(ngram)

    #return ngrams_list

        x = set(ngrams_list)
                                                       # Extracting tokens by characters
        return list(x)
    elif char_ngrams == True:
        ngrams_list = []
        for x in range(len(x_raw)):
            n=x_raw[x:x+2]
            ngrams_list.append(n)
        for x in range(len(x_raw)):
            n=x_raw[x:x+3]
            ngrams_list.append(n)
        for x in range(len(x_raw)):
            n=x_raw[x:x+4]
            ngrams_list.append(n)
        x = set(ngrams_list)

        return list(x)
```

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

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

In [7]:

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

Out[7]:

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

Create a vocabulary

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

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

and returns:

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

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

In [8]:

```
def get_vocab(X_raw,
              ngram_range=(1, 3),
              token_pattern=r'\b[A-Za-z]{2,}\b',
              min_df=1,
              keep_topN=0,
              stop_words=stop_words):

    doc_freq = Counter()
    ngram_count = Counter()
    z=[]
    for text in X_raw:
        # A list of ngrams for the given document `text`
        ngram_list = extract_ngrams(text, ngram_range, token_pattern, stop_words,char_ngram
        doc_freq.update(set(ngram_list))                              # Here we are counting docu
        for ngram in ngram_list:                                      # Here we are counting ngra
            list=[]
            if doc_freq[ngram]>=min_df:
                list.append(ngram)
            ngram_count.update(list)
    vocab = {ngram for ngram, _ in ngram_count.most_common(keep_topN)}   # Here we are extr
    return vocab, doc_freq, ngram_count
```

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

In [9]:

```
vocab, doc_freq = get_vocab(train_text, keep_topN=5000)[0:2]
```

In [10]:

```
len(vocab)
```

Out[10]:

```
5000
```

In [11]:

```
vocab
```

Out[11]:

```
{'troopers',
 'police officer',
 'while',
 'san',
 'well but',
 'talented',
 'dead',
 'george clooney',
 'bell',
 'all film',
 'benefit',
 'love story',
 'upon',
 'beating',
 'never',
 'intricate',
 'inevitable',
 'wonders',
```

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

In [12]:

```
id2word = {index: value for index, value in enumerate(vocab)}     # vocabulary id -> word

word_to_vocab_id = {k : v for k, v in id2word.items()}            # word -> vocabulary id
print(word_to_vocab_id)
```

```
{0: 'troopers', 1: 'police officer', 2: 'while', 3: 'san', 4: 'well but',
5: 'talented', 6: 'dead', 7: 'george clooney', 8: 'bell', 9: 'all film', 1
0: 'benefit', 11: 'love story', 12: 'upon', 13: 'beating', 14: 'never', 1
5: 'intricate', 16: 'inevitable', 17: 'wonders', 18: 'scenario', 19: 'proj
ect', 20: 'taken', 21: 'exact', 22: 'death', 23: 'unpleasant', 24: 'anna',
25: 'there also', 26: 'finds', 27: 'clean', 28: 'my favorite', 29: 'jennif
er', 30: 'lone', 31: 'painting', 32: 'murders', 33: 'provoking', 34: 'sign
ificance', 35: 'walter', 36: 'cop', 37: 'rich', 38: 'moments', 39: 'suspen
se', 40: 'first rate', 41: 'best known', 42: 'makes up', 43: 'more', 44:
'big', 45: 'must', 46: 'marry', 47: 'profound', 48: 'timing', 49: 'positiv
e', 50: 'so many', 51: 'digital', 52: 'neve campbell', 53: 'explored', 54:
'happily', 55: 'exception', 56: 'generally', 57: 'phenomenon', 58: 'look l
ike', 59: 'jazz', 60: 'capture', 61: 'point', 62: 'coming', 63: 'regular',
64: 'disaster', 65: 'unique', 66: 'turkey', 67: 'wonder if', 68: 'qualit
y', 69: 'forms', 70: 'covers', 71: 'seen before', 72: 'bad but', 73: 'firs
t all', 74: 'camp', 75: 'purpose', 76: 'keanu', 77: 'audiences', 78: 'kean
u reeves', 79: 'something', 80: 'among', 81: 'needed', 82: 'luck', 83: 'he
lps', 84: 'training', 85: 'missed', 86: 'daniel', 87: 'supposedly', 88: 'i
nternational', 89: 'when first', 90: 'five years', 91: 'plot', 92: 'indepe
```

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

In [13]:

```
def generator_train ():
    return (extract_ngrams(text, vocab=vocab,char_ngrams=False)for text in train_text)
```

In [14]:

```
def generator_dev ():
    return (extract_ngrams(text, vocab=vocab,char_ngrams=False)for text in dev_text)
```

In [15]:

```
def generator_test ():
    return (extract_ngrams(text, vocab=vocab,char_ngrams=False)for text in test_text)
```

In [16]:

```
train_texts_ngrams = generator_train()
dev_texts_ngrams = generator_dev ()
test_texts_ngrams = generator_test()
```

Vectorise documents

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

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

and return:

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

In [17]:

```
def vectorise(X_ngram, vocab):
    X_vec = []
    for ngram_list in X_ngram:          # Here we are creating a function to vectorise th
        counter = Counter(ngram_list)     # Here we are using temperory list variable to ap
        list = []
        for v in vocab:
            list.append(counter[v])
        X_vec.append(list)
    return np.array(X_vec)
```

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

Count vectors

In [18]:

```
train_count = vectorise(train_texts_ngrams, vocab)

dev_count = vectorise(dev_texts_ngrams, vocab)

test_count = vectorise(test_texts_ngrams, vocab)
```

In [19]:

```
train_count.shape
```

Out[19]:

```
(1399, 5000)
```

In [20]:

```
train_count[:2,:100]
```

Out[20]:

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

TF.IDF vectors

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

In [21]:

```
total_train_docs = len(train_text)
total_dev_docs = len(dev_text)
total_test_docs = len(test_text)

dev_df = get_vocab(dev_text, keep_topN=5000)[1]
test_df = get_vocab(test_text, keep_topN=5000)[1]
for v in vocab:                                                                          # Re
    train_idf = np.array([np.log10(total_train_docs/doc_freq[v])])
for v in vocab:
    if dev_df[v]:                                                              # Log normalis
        dev_idf = np.array([np.log10(total_dev_docs / dev_df[v])])
for v in vocab:
    if test_df[v]:
        test_idf = np.array([np.log10(total_test_docs/test_df[v])])
```

Then transform your count vectors to tf.idf vectors:

In [22]:

```
Train_norm = np.log10(1 + train_count)  # Reference ---> Lecture Notes
Dev_norm = np.log10(1 + dev_count)      # squash the raw frequency, by using the Log10.
Test_norm = np.log10(1 + test_count)
```

In [23]:

```
train_tfidf = Train_norm * train_idf   # Calculating Tfidf
dev_tfidf = Dev_norm * dev_idf         # tfidf = tf * idf
test_tfidf = Test_norm * test_idf
```

Binary Logistic Regression

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

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

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

and returns:

- `sig` : the sigmoid of `z`

In [24]:

```
def sigmoid(z):                    # Reference
    sig = 1 / (1 + np.exp(-z))     # # https://towardsdatascience.com/building-a-logistic-regr
    return sig
```

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

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

and returns:

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

In [25]:

```
def predict_proba(X, weights):   # Reference Logic
    z = X.dot(weights)           # https://pyimagesearch.com/2016/10/17/stochastic-gradient-
    preds_proba = sigmoid(z)

    return preds_proba
```

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

- `X` : an array of documents represented by bag-of-ngram vectors ($N \times |vocab|$)

- `weights` : a 1-D array of the model's weights $(1, |vocab|)$

and returns:

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

In [26]:

```
def predict_class(X, weights):
    list = []
    for prob in predict_proba(X,weights):      # Assignining range if <= 0.5 then assign to
        if prob <= 0.5:
            list.append(0)
        else:
            list.append(1)
    preds_class = list
    return preds_class
```

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

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

and return:

- `l` : the loss score

In [27]:

```
def binary_loss(X, Y, weights, alpha=0.00001):
    l = -Y * np.log(predict_proba(X, weights)) - (1 - Y) * np.log(1 - predict_proba(X, weig

    # L2 Regularisation                       # Reference
    l = l + alpha * weights.dot(weights)      # https://github.com/akashmantry/LogisticRegres
                                              # Lreg = L + αR(w)    --> Lecture notes
    # Return the average loss
    return np.mean(l)
```

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

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

- `print_progress` : flag for printing the training progress (train/validation loss)

and returns:

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

In [28]:

```
def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], lr=0.1, alpha=0.00001, epochs=5, tolerance=0.0001,

    np.random.seed(123)                     # Random seed is fixed here so that we can get sa
    training_loss_history = []
    validation_loss_history = []

    weights_int_list = []
    for i in range(train_count.shape[1]):
        weights_int_list.append(0)          # Creating weights with all zeros so that we can
    weights_int = np.array(weights_int_list)
    weights = weights_int.astype(np.float)

    def zipper(X_tr, Y_tr):
        size = len(X_tr) if len(X_tr) < len(Y_tr) else len(Y_tr)
        retList = []                        # Create training tuples
        for i in range(size):               # Adding values from two list simultaneously
            retList.append((X_tr[i], Y_tr[i]))
        return retList

    train_docs = zipper(X_tr, Y_tr)

    for epoch in range(epochs):
        np.random.shuffle(train_docs)       # Shuffling to randomise all values
                                            # Reference
        for first, second in train_docs:    # w = w − η∇wL(w;xi;yi)  --> Lecture Notes
            weights = weights - lr * (first * (predict_proba(first, weights) - second) + 2

        # Monitor training and validation loss
        loss_in_training = binary_loss(X_tr, Y_tr, weights, alpha)
        loss_in_dev = binary_loss(X_dev, Y_dev, weights, alpha)

        # Early stopping            # Reference
                                    # previous validation loss − current validation loss; s
        if epoch > 0 and validation_loss_history[-1] - loss_in_dev < tolerance:
            break
        else:
            training_loss_history.append(loss_in_training)
            validation_loss_history.append(loss_in_dev)

        if print_progress:
            print("Epoch:- ",epoch,"  ","Training loss:- ",loss_in_training,"  ","Validatio

    return weights, training_loss_history, validation_loss_history
```

Train and Evaluate Logistic Regression with Count vectors

First train the model using SGD:

In [29]:

```
w_count, training_loss_count, dev_loss_count = SGD(X_tr=train_count,Y_tr=np.array(train_lab
```

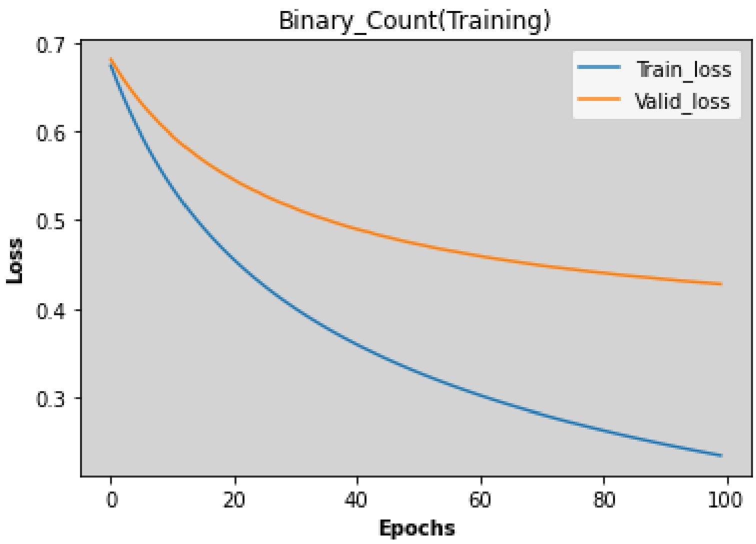
```
<ipython-input-28-1bbe47e1eeaf>:11: DeprecationWarning: `np.float` is a de
precated alias for the builtin `float`. To silence this warning, use `floa
t` by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.or
g/devdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdoc
s/release/1.20.0-notes.html#deprecations)
  weights = weights_int.astype(np.float)
```

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

In [30]:

```
plt.plot(training_loss_count, label='Train_loss')
plt.plot(dev_loss_count, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Binary_Count(Training)')
ax = plt.axes()
ax.set_facecolor("lightgray")


plt.legend()

plt.show()
```

```
<ipython-input-30-41ab2a87d266>:8: MatplotlibDeprecationWarning: Adding an a
xes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and re
turned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



Explain here...

From the plot above, it is clear that.. (i) Train loss is decreasing as count of epoch increases until it reaches a certian standard of stability..(ii) Valid loss is also decreasing as count of epoch increases until it reaches a certain standard of stability. From this I conclude that, the model is about right.

Evaluation

Compute accuracy, precision, recall and F1-scores:

In [31]:

```
array_test_label = np.array(test_label)                # Changing list to array as accu
```

In [32]:

```
preds_te_count = predict_class(test_count, w_count)

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

```
Accuracy: 0.8596491228070176
Precision: 0.8454106280193237
Recall: 0.8793969849246231
F1-Score: 0.8620689655172413
```

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

In [33]:

```
top_neg = w_count.argsort()[:10]       # Printing top ten negative elements
for i in top_neg:
    print(id2word[i])
```

```
bad
script
worst
unfortunately
nothing
plot
boring
only
looks
supposed
```

In [34]:

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

```
hilarious
also
both
great
well
many
seen
true
perfect
perfectly
```

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?

The classifier here predicts a good set of positive and negative words and it is expected that it will correctly predict the some of the reviews about laptops and restaurant. However,specially for laptop and restaurant there may have some words which defines best positive and best negative words for review and using this classifier will give accuracy but not to a great extend and it will somewhat make model underfit. Hence I dont think this features would generalise well in laptop and restaurant reviews.

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 have carried out trial and error strategy to choose model hyperparameter. As mentioned in the instruction, *Instead of using all possible combinations, you could perform a random sampling of combinations.* I have done the same. In hyperparamter optimisation, the lower bound and upper bound values are need to be defined where lower bound value is the baseline for optimal performance of model and upper bound is the one where the performance starts to mitigate.For this purpose I have set the initial lower and upper bound for learning rate as 0.0001 and 0.1 where for regularisation i have set it to 0.00001 and 0.01 precisely. After that I started chaging the lower bound and upper bound to see the results in terms of precision, recall and F1-score.

Count Vectors

Table showing model performance for learning rate :-

Trial	Learning rate	Epochs	Tr. loss	Val. loss	Precision	Recall	F1-Score
0	0.0001	99	0.2033	0.4097	0.8523	0.876	0.8634
1	0.00011	99	0.1945	0.4086	0.8522	0.868	0.8567
2	0.000105	99	0.1975	0.4083	0.8457	0.871	0.8592
3	0.00010124	99	0.2033	0.4097	0.8523	0.876	0.8635

Table showing model performance for Regularisation strength :-

Trial	Alpha	Epochs	Tr. loss	Val. loss	Precision	Recall	F1-Score
0	0.00001	99	0.2031	0.40954	0.8523	0.874	0.8640
1	0.00002	99	0.2017	0.40953	0.8522	0.873	0.8640
2	0.000015	99	0.2022	0.40948	0.8522	0.874	0.8639
3	0.0000124	99	0.2029	0.40944	0.8523	0.874	0.8640

TF.IDF Vectors

Table showing model performance for learning rate :-

Trial	Learning rate	Epochs	Tr. loss	Val. loss	Precision	Recall	F1-Score
0	0.0001	23	0.5025	0.5845	0.8412	0.867	0.8654
1	0.0002	23	0.4053	0.5321	0.8632	0.875	0.8733
2	0.0003	23	0.3439	0.4998	0.8654	0.848	0.8798
3	0.0025	23	0.0856	0.3750	0.8869	0.875	0.8759

Table showing model performance for Regularisation strength :-

Trial	Alpha	Epochs	Tr. loss	Val. loss	Precision	Recall	F1-Score
0	0.00001	23	0.0891	0.3756	0.8878	0.873	0.8810

Trial	Alpha	Epochs	Tr. loss	Val. loss	Precision	Recall	F1-Score
1	0.00002	23	0.0903	0.3785	0.8880	0.873	0.8810
2	0.0004	23	0.1645	0.4265	0.88	0.88	0.8866
3	0.0008	23	0.2206	0.4583	0.875	0.875	0.8860

Relationship Between Epochs and Learning Rate

The relationship between Epochs and learning arte follows a proportionality rule, where greater the learning rate, larger the weight update after each epochs.

How regularisation strength affects performance.

Regularisation rate affects the convergence of epoch which in turn affects the model performance, so we can say that regularisation strength indirectly correlates with the model performance.

Train and Evaluate Logistic Regression with TF.IDF vectors

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

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

In [35]:

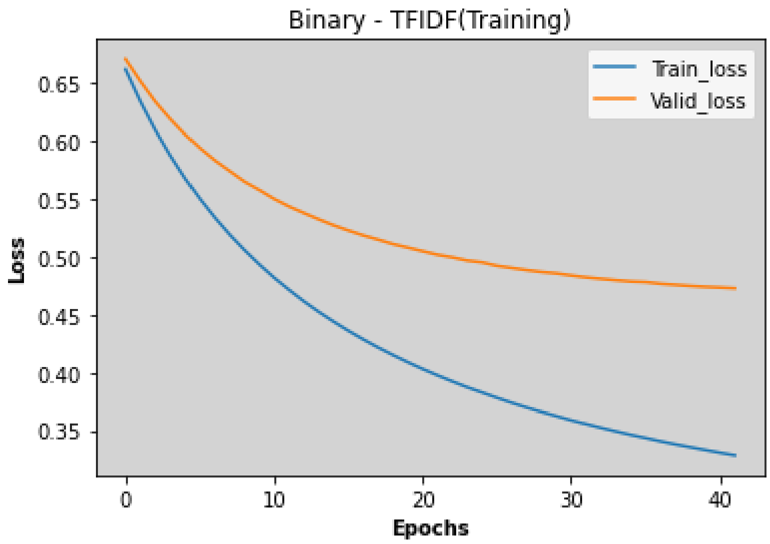
```
w_tfidf, training_loss_tfidf, dev_loss_tfidf = SGD(X_tr=train_tfidf,Y_tr=np.array(train_lab
```

```
Epoch:- 0    Training loss:-  0.6619948876678392    Validation loss:-  0.
6708951556266846
Epoch:- 1    Training loss:-  0.6344381228181969    Validation loss:-  0.
651831140695691
Epoch:- 2    Training loss:-  0.6097707917410218    Validation loss:-  0.
6342512289210034
Epoch:- 3    Training loss:-  0.5877746679357756    Validation loss:-  0.
619360059437997

<ipython-input-28-1bbe47e1eeaf>:11: DeprecationWarning: `np.float` is a de
precated alias for the builtin `float`. To silence this warning, use `floa
t` by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.or
g/devdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdoc
s/release/1.20.0-notes.html#deprecations)
  weights = weights_int.astype(np.float)
```

In [36]:

```
plt.plot(training_loss_tfidf, label='Train_loss')
plt.plot(dev_loss_tfidf, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Binary - TFIDF(Training)')
ax = plt.axes()
ax.set_facecolor("lightgray")

plt.legend()

plt.show()
```

```
<ipython-input-36-77ed45a3342a>:8: MatplotlibDeprecationWarning: Adding an a
xes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and re
turned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



In [37]:

```
preds_te_tfidf = predict_class(test_tfidf, w_tfidf)

print('Accuracy:', accuracy_score(np.array(test_label),preds_te_tfidf))
print('Precision:', precision_score(np.array(test_label),preds_te_tfidf))
print('Recall:', recall_score(np.array(test_label),preds_te_tfidf))
print('F1-Score:', f1_score(np.array(test_label),preds_te_tfidf))
```

```
Accuracy: 0.8621553884711779
Precision: 0.86
Recall: 0.864321608040201
F1-Score: 0.8621553884711779
```

In [38]:

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

```
bad
script
worst
nothing
plot
unfortunately
boring
looks
only
least
```

In [39]:

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

```
hilarious
also
both
great
many
well
perfect
true
best
seen
```

BOCN

In [40]:

```
def extract_ngrams_c(x_raw, ngram_range=(1,3), token_pattern=r'\b[A-Za-z]{2,}\b',
                    stop_words= stop_words, vocab=None, char_ngrams = True):
    if char_ngrams == False:
        tokens = []
        for word in re.findall(token_pattern,x_raw):
            if word.lower() not in stop_words:
                tokens.append(word.lower())

        ngrams_list = []

        for num in range(0, len(tokens)):
            ngram = ' '.join(tokens[num:num + 1])              # Extracting tokens by words
            ngrams_list.append(ngram)

        for num in range(0, len(tokens)):
            ngram = ' '.join(tokens[num:num + 2])
            ngrams_list.append(ngram)

        for num in range(0, len(tokens)):
            ngram = ' '.join(tokens[num:num + 3])
            ngrams_list.append(ngram)

    #return ngrams_list

        x = set(ngrams_list)

        return list(x)
    elif char_ngrams == True:
        ngrams_list = []
        for x in range(len(x_raw)):                          # Extracting tokens by charact
            n=x_raw[x:x+2]
            ngrams_list.append(n)
        for x in range(len(x_raw)):
            n=x_raw[x:x+3]
            ngrams_list.append(n)
        for x in range(len(x_raw)):
            n=x_raw[x:x+4]
            ngrams_list.append(n)
        x = set(ngrams_list)

        return list(x)
```

In [41]:

```
def get_vocab_c(X_raw,
              ngram_range=(1, 3),
              token_pattern=r'\b[A-Za-z]{2,}\b',
              min_df=1,
              keep_topN=0,
              stop_words=stop_words):

    doc_freq_c = Counter()
    ngram_count_c = Counter()
    for text in X_raw:
        # A list of ngrams for the given document `text`
        ngram_list = extract_ngrams_c(text, ngram_range, token_pattern, stop_words,char_ngr
        doc_freq_c.update(set(ngram_list))    # Here we are counting document frequency
        for ngram in ngram_list:              # Here we are counting ngram frequency
            list=[]
            if doc_freq_c[ngram]>=min_df:
                list.append(ngram)
            ngram_count_c.update(list)
    vocab_c = {ngram for ngram, _ in ngram_count_c.most_common(keep_topN)}   # Here we are
    return vocab_c, doc_freq_c, ngram_count_c
```

In [42]:

```
vocab_c, doc_freq_c = get_vocab_c(train_text, keep_topN=5000)[0:2]
```

In [43]:

```
len(vocab_c)
```

Out[43]:

```
5000
```

In [44]:

```
vocab_c
```

Out[44]:

```
{'t f',
 'nit',
 'lp',
 'san',
 ' cr',
 'ood ',
 'e an',
 'his',
 'y ) ',
 'ng s',
 "'s c",
 'at b',
 'ille',
 'm i',
 'o ,',
 ' lon',
 'his ',
 ' who',
```

In [45]:

```
def generator_train_c ():
    return (extract_ngrams_c(text, vocab=vocab_c,char_ngrams=True)for text in train_text)
```

In [46]:

```
def generator_dev_c ():
    return (extract_ngrams_c(text, vocab=vocab_c,char_ngrams=True)for text in dev_text)
```

In [47]:

```
def generator_test_c ():
    return (extract_ngrams_c(text, vocab=vocab_c,char_ngrams=True)for text in test_text)
```

In [48]:

```
train_texts_ngrams_c = generator_train()
dev_texts_ngrams_c = generator_dev ()
test_texts_ngrams_c = generator_test()
```

In [49]:

```
id2word_c = {index: value for index, value in enumerate(vocab_c)}    # vocabulary id -> wor

word_to_vocab_id_c = {k : v for k, v in id2word_c.items()}           # word -> vocabulary i
print(word_to_vocab_id_c)
```


```
{0: 't f', 1: 'nit', 2: 'lp', 3: 'san', 4: ' cr', 5: 'ood ', 6: 'e an', 7:
'his', 8: 'y ) ', 9: 'ng s', 10: "'s c", 11: 'at b', 12: 'ille', 13: 'm
i', 14: 'o ,', 15: ' lon', 16: 'his ', 17: ' who', 18: ' mov', 19: ' ne',
20: 's ev', 21: 'il', 22: 'succ', 23: 'e li', 24: 'unti', 25: 'ay .', 26:
'ince', 27: ', w', 28: 'more', 29: 'big', 30: 'e si', 31: 'must', 32: ' al
w', 33: 'ma', 34: 'ho ', 35: 'eal ', 36: 's m', 37: 'cro', 38: 'or', 39:
'nd l', 40: 'ie ,', 41: 'ish ', 42: 's ne', 43: " he'", 44: 'lity', 45: '
i ', 46: 'dien', 47: 'opl', 48: 'cre', 49: 'o be', 50: 'ead ', 51: 'fli',
52: 'as m', 53: ' la', 54: 'phe', 55: ' eac', 56: 'uin', 57: 'l i', 58: 'p
ro', 59: 'cie', 60: 'e ri', 61: 'itin', 62: 'wat', 63: ' mar', 64: 're f',
65: 'in ,', 66: 'ieve', 67: 'sic', 68: 'ema', 69: 'cove', 70: ' nat', 71:
'n hi', 72: 'be t', 73: ' sma', 74: 'al ', 75: 'tory', 76: 'en a', 77: ' t
er', 78: 'irs', 79: 'stea', 80: 'tro', 81: 'ses ', 82: 'ped', 83: 'of f',
84: 'to e', 85: 'plot', 86: 'r .', 87: ' an', 88: 'h .', 89: 'nvol', 90:
'ortu', 91: 'sma', 92: 'st s', 93: '( th', 94: 'len', 95: 'ne a', 96: 'o l
o', 97: 'd ,', 98: 'es i', 99: 'enta', 100: ' off', 101: 'wr', 102: 'rec
o', 103: 'out', 104: 'ces ', 105: 'leav', 106: ' mus', 107: 'y we', 108:
'ula', 109: ' . "', 110: 'ly r', 111: 'is f', 112: 'ne o', 113: ' cas', 11
4: 'n t', 115: 'd ac', 116: 'er m', 117: 'ou', 118: 'hon', 119: 'eme', 12
```

In [50]:

```
def vectorise_c(X_ngram, vocab):
    X_vec_c = []
    for ngram_list in X_ngram:             # Here we are creating a function to vectorise t
        counter = Counter(ngram_list)      # Here we are using temperory list variable to a
        list_c = []
        for v in vocab:
            list_c.append(counter[v])
        X_vec_c.append(list_c)
    return np.array(X_vec_c)
```

In [51]:

```
train_count_c = vectorise_c(train_texts_ngrams_c, vocab_c)

dev_count_c = vectorise_c(dev_texts_ngrams_c, vocab_c)

test_count_c = vectorise_c(test_texts_ngrams_c, vocab_c)
```

In [52]:

```
train_count_c.shape
```

Out[52]:

```
(1399, 5000)
```

In [53]:

```
total_train_docs_c = len(train_text)
total_dev_docs_c = len(dev_text)
total_test_docs_c = len(test_text)

dev_df_c = get_vocab_c(dev_text, keep_topN=5000)[1]
test_df_c = get_vocab_c(test_text, keep_topN=5000)[1]
for v in vocab_c:              # Reference  --> Lecture Notes
    train_idf_c = np.array([np.log10(total_train_docs_c/doc_freq_c[v])])
for v in vocab_c:
    if dev_df_c[v]:
        dev_idf_c = np.array([np.log10(total_dev_docs_c / dev_df_c[v])])
for v in vocab_c:
    if test_df_c[v]:
        test_idf_c = np.array([np.log10(total_test_docs_c/test_df_c[v])])
```

In [54]:

```
Train_norm_c = np.log10(1 + train_count_c)  # Reference ---> Lecture Notes
Dev_norm_c = np.log10(1 + dev_count_c)      # squash the raw frequency, by using the log10.
Test_norm_c = np.log10(1 + test_count_c)
```

In [55]:

```
train_tfidf_c = Train_norm_c * train_idf_c              # Calculating Tfidf
dev_tfidf_c = Dev_norm_c * dev_idf_c                    # tfidf = tf * idf
test_tfidf_c = Test_norm_c * test_idf_c
```

In [56]:

```
def sigmoid_c(z):                    # Reference
    return 1 / (1 + np.exp(-z))      # https://towardsdatascience.com/building-a-logistic-re
```

In [57]:

```
def predict_proba_c(X, weights):     # Reference Logic
    z = X.dot(weights)               # https://pyimagesearch.com/2016/10/17/stochastic-gradi
    return sigmoid_c(z)
```

In [58]:

```
def predict_class_c(X,weights):
    list = []
    for prob in predict_proba_c(X,weights):
        if prob <= 0.5:
            list.append(0)             # Assignining range if <= 0.5 then assign to 0 else
        else:
            list.append(1)
    return list
```

In [59]:

```
def binary_loss_c(X, Y, weights, alpha=0.00001):
    l = -Y * np.log(predict_proba_c(X, weights)) - (1 - Y) * np.log(1 - predict_proba_c(X,

    # L2 Regularisation                    # Reference
    l += alpha * weights.dot(weights)      # https://github.com/akashmantry/LogisticRegressio
                                           # Lreg = L + aR(w)    --> Lecture notes
    # Return the average loss
    return np.mean(l)
```

In [60]:

```
def SGD_c(X_tr, Y_tr, X_dev, Y_dev, lr=0.1, alpha=0.00001, epochs=5, tolerance=0.0001, prin

    np.random.seed(123)                    # Random seed is fixed here so that we can get sa
    training_loss_history_c = []
    validation_loss_history_c = []

    weights_int_list_c = []                # Creating weights with all zeros so that we can
    for i in range(train_count_c.shape[1]):
        weights_int_list_c.append(0)
    weights_int_c = np.array(weights_int_list_c)
    weights_c = weights_int_c.astype(np.float)

    def zipper_c(X_tr, Y_tr):                                  # Create training tuples
        size = len(X_tr) if len(X_tr) < len(Y_tr) else len(Y_tr)  # Adding values from two
        retList = []
        for i in range(size):
            retList.append((X_tr[i], Y_tr[i]))
        return retList

    train_docs_c = zipper_c(X_tr, Y_tr)

    for epoch in range(epochs):
        np.random.shuffle(train_docs_c)      # Shuffling to randomise all values
                                             # Reference
        for first, second in train_docs_c:     # w = w − η∇wL(w;xi;yi)  --> Lecture Notes
            weights_c = weights_c - lr * (first * (predict_proba_c(first, weights_c) - seco

        # Monitor training and validation loss
        loss_in_training_c = binary_loss_c(X_tr, Y_tr, weights_c, alpha)
        loss_in_dev_c = binary_loss_c(X_dev, Y_dev, weights_c, alpha)

        # Early stopping            # Reference
                                    # previous validation loss − current validation loss; s
        if epoch > 0 and validation_loss_history_c[-1] - loss_in_dev_c < tolerance:
            break
        else:
            training_loss_history_c.append(loss_in_training_c)
            validation_loss_history_c.append(loss_in_dev_c)

        if print_progress:
            #print(f'Epoch: {epoch} | Training Loss: {cur_loss_tr} | Validation Loss: {cur_
            print("Epoch:- ",epoch,"  ","Training loss:- ",loss_in_training_c,"  ","Validat
    return weights_c, training_loss_history_c, validation_loss_history_c
```

In [61]:

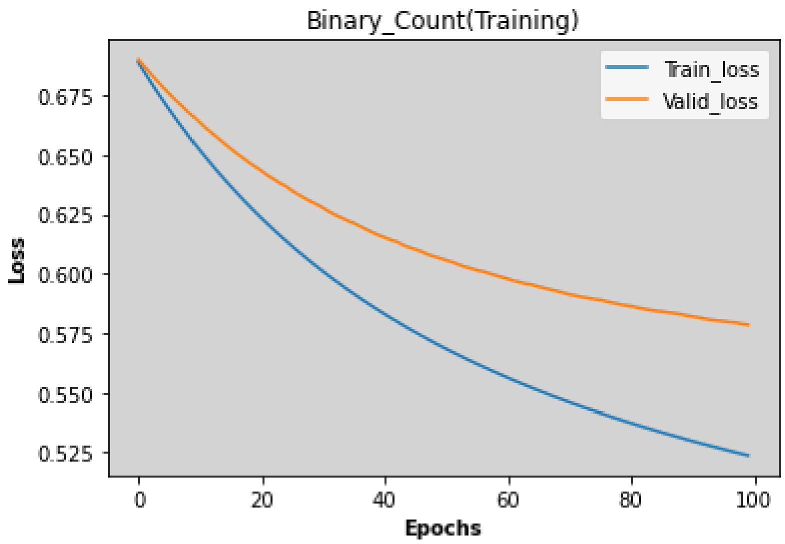
```
w_count_c, training_loss_count_c, dev_loss_count_c = SGD_c(X_tr=train_count_c,Y_tr=np.array
```

```
<ipython-input-60-962918fee0ef>:11: DeprecationWarning: `np.float` is a de
precated alias for the builtin `float`. To silence this warning, use `floa
t` by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.or
g/devdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdoc
s/release/1.20.0-notes.html#deprecations)
  weights_c = weights_int_c.astype(np.float)
```

In [62]:

```
plt.plot(training_loss_count_c, label='Train_loss')
plt.plot(dev_loss_count_c, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Binary_Count(Training)')
ax = plt.axes()
ax.set_facecolor("lightgray")


plt.legend()

plt.show()
```

```
<ipython-input-62-4417a4b65adf>:8: MatplotlibDeprecationWarning: Adding an a
xes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and re
turned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



In [63]:

```
array_test_label_c = np.array(test_label)     # Changing list to array as accuracy score ta
```

In [64]:

```
preds_te_count_c = predict_class_c(test_count_c, w_count_c)

print('Accuracy:', accuracy_score(array_test_label_c,preds_te_count_c))
print('Precision:', precision_score(array_test_label_c,preds_te_count_c))
print('Recall:', recall_score(array_test_label_c,preds_te_count_c))
print('F1-Score:', f1_score(array_test_label_c,preds_te_count_c))
```

```
Accuracy: 0.6992481203007519
Precision: 0.7046632124352331
Recall: 0.6834170854271356
F1-Score: 0.6938775510204082
```

In [65]:

```
w_tfidf_c, training_loss_tfidf_c, dev_loss_tfidf_c = SGD_c(X_tr=train_tfidf_c,Y_tr=np.array
```

```
Epoch:-  0    Training loss:-  0.6904966795485448    Validation loss:-  0.69
12148003649882
Epoch:-  1    Training loss:-  0.6879143906142949    Validation loss:-  0.68
93460373561778
Epoch:-  2    Training loss:-  0.685398383432307    Validation loss:-  0.687
5191950035022
Epoch:-  3    Training loss:-  0.6829470269153042    Validation loss:-  0.68
57156108230362

<ipython-input-60-962918fee0ef>:11: DeprecationWarning: `np.float` is a depr
ecated alias for the builtin `float`. To silence this warning, use `float` b
y itself. Doing this will not modify any behavior and is safe. If you specif
ically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/d
evdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdocs/rel
ease/1.20.0-notes.html#deprecations)
  weights_c = weights_int_c.astype(np.float)

Epoch:-  4    Training loss:-  0.680558160644537    Validation loss:-  0.683
9878531458286
Epoch:-  5    Training loss:-  0.6782310802076481    Validation loss:-  0.68
22830053666049
Epoch:-  6    Training loss:-  0.6759636483164315    Validation loss:-  0.68
06552389948863
Epoch:-  7    Training loss:-  0.6737528936179766    Validation loss:-  0.67
91565087576937
Epoch:-  8    Training loss:-  0.6715987837294469    Validation loss:-  0.67
75782747496061
Epoch:-  9    Training loss:-  0.6695001744519493    Validation loss:-  0.67
61485316169866
Epoch:-  10    Training loss:-  0.6674506086436001    Validation loss:-  0.6
746425538482389
Epoch:-  11    Training loss:-  0.6654592925586996    Validation loss:-  0.6
731673889756857
Epoch:-  12    Training loss:-  0.6635130903135239    Validation loss:-  0.6
718266619510327
Epoch:-  13    Training loss:-  0.6616168004478585    Validation loss:-  0.6
705692210081527
Epoch:-  14    Training loss:-  0.6597692076603948    Validation loss:-  0.6
692164750684666
Epoch:-  15    Training loss:-  0.6579683188466647    Validation loss:-  0.6
679330967753333
Epoch:-  16    Training loss:-  0.6562125801024464    Validation loss:-  0.6
666896282719543
Epoch:-  17    Training loss:-  0.6544973561428005    Validation loss:-  0.6
655142793448491
Epoch:-  18    Training loss:-  0.6528302780182359    Validation loss:-  0.6
643263326142681
Epoch:-  19    Training loss:-  0.6511946014688571    Validation loss:-  0.6
632791263933235
Epoch:-  20    Training loss:-  0.64960538135053    Validation loss:-  0.662
1994711040016
Epoch:-  21    Training loss:-  0.6480596468159974    Validation loss:-  0.6
610611943364745
Epoch:-  22    Training loss:-  0.6465434471536388    Validation loss:-  0.6
600869591122408
```

```
Epoch:-  23    Training loss:-  0.645068576042016    Validation loss:-  0.65
91066316909521
Epoch:-  24    Training loss:-  0.6436397865873372    Validation loss:-  0.6
582540602283568
Epoch:-  25    Training loss:-  0.6422264631067357    Validation loss:-  0.6
571663337373143
Epoch:-  26    Training loss:-  0.6408572043309972    Validation loss:-  0.6
562113329625406
Epoch:-  27    Training loss:-  0.6395199364942084    Validation loss:-  0.6
553189405528105
Epoch:-  28    Training loss:-  0.6382144288631433    Validation loss:-  0.6
544948083909319
Epoch:-  29    Training loss:-  0.6369467834926484    Validation loss:-  0.6
537361040835109
Epoch:-  30    Training loss:-  0.6357022008609299    Validation loss:-  0.6
528996591755573
Epoch:-  31    Training loss:-  0.634487537703162    Validation loss:-  0.65
19781291926887
Epoch:-  32    Training loss:-  0.6333072223571137    Validation loss:-  0.6
511708657201877
Epoch:-  33    Training loss:-  0.6321496257458664    Validation loss:-  0.6
504381205381176
Epoch:-  34    Training loss:-  0.631020405955202    Validation loss:-  0.64
97141497734192
Epoch:-  35    Training loss:-  0.6299213123352827    Validation loss:-  0.6
491141699736936
Epoch:-  36    Training loss:-  0.6288437483137984    Validation loss:-  0.6
483826365280088
Epoch:-  37    Training loss:-  0.627792778156486    Validation loss:-  0.64
76876601753361
Epoch:-  38    Training loss:-  0.6267697340866303    Validation loss:-  0.6
469560402377199
Epoch:-  39    Training loss:-  0.62577109504772    Validation loss:-  0.646
2968754880194
Epoch:-  40    Training loss:-  0.6247893995781216    Validation loss:-  0.6
457195940601848
Epoch:-  41    Training loss:-  0.6238330308730168    Validation loss:-  0.6
451456599410135
Epoch:-  42    Training loss:-  0.622907769239038    Validation loss:-  0.64
46865211713771
Epoch:-  43    Training loss:-  0.621990637125021    Validation loss:-  0.64
39500472776369
Epoch:-  44    Training loss:-  0.6211004221479085    Validation loss:-  0.6
433982353769517
Epoch:-  45    Training loss:-  0.6202294821263465    Validation loss:-  0.6
429330341276622
Epoch:-  46    Training loss:-  0.6193795408687143    Validation loss:-  0.6
424055070556662
Epoch:-  47    Training loss:-  0.6185523986549033    Validation loss:-  0.6
418127534026768
Epoch:-  48    Training loss:-  0.6177405997587774    Validation loss:-  0.6
413214054657524
Epoch:-  49    Training loss:-  0.6169444931016976    Validation loss:-  0.6
409139392368264
Epoch:-  50    Training loss:-  0.6161696317570117    Validation loss:-  0.6
404589995523223
Epoch:-  51    Training loss:-  0.6154144437424934    Validation loss:-  0.6
400543657975111
Epoch:-  52    Training loss:-  0.6146721579861727    Validation loss:-  0.6
395194951842923
Epoch:-  53    Training loss:-  0.613950512277618    Validation loss:-  0.63
```

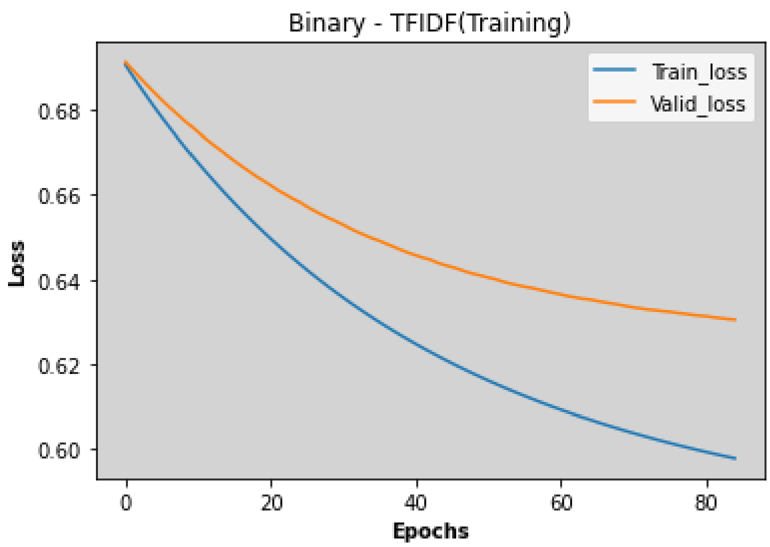
```
90642984891007
Epoch:-  54    Training loss:-  0.6132409817658497    Validation loss:-  0.6
386889410985886
Epoch:-  55    Training loss:-  0.6125495430507765    Validation loss:-  0.6
382842781491925
Epoch:-  56    Training loss:-  0.6118787950755831    Validation loss:-  0.6
380094188129629
Epoch:-  57    Training loss:-  0.6112159077092548    Validation loss:-  0.6
376084825497832
Epoch:-  58    Training loss:-  0.6105684468967958    Validation loss:-  0.6
372249852269722
Epoch:-  59    Training loss:-  0.6099342206403221    Validation loss:-  0.6
368385280955706
Epoch:-  60    Training loss:-  0.6093158962334234    Validation loss:-  0.6
36479997351823
Epoch:-  61    Training loss:-  0.608712076620917    Validation loss:-  0.63
60716884566442
Epoch:-  62    Training loss:-  0.6081199126310607    Validation loss:-  0.6
357828762129853
Epoch:-  63    Training loss:-  0.6075416844678508    Validation loss:-  0.6
354582779947391
Epoch:-  64    Training loss:-  0.6069815125984883    Validation loss:-  0.6
352588358636613
Epoch:-  65    Training loss:-  0.6064251797514182    Validation loss:-  0.6
34920600417333
Epoch:-  66    Training loss:-  0.6058821679186666    Validation loss:-  0.6
346028260796892
Epoch:-  67    Training loss:-  0.6053533133986634    Validation loss:-  0.6
343262217219329
Epoch:-  68    Training loss:-  0.6048355743651367    Validation loss:-  0.6
340569617137566
Epoch:-  69    Training loss:-  0.6043268551978904    Validation loss:-  0.6
337115608781594
Epoch:-  70    Training loss:-  0.6038331569017004    Validation loss:-  0.6
33406484987694
Epoch:-  71    Training loss:-  0.6033474111540452    Validation loss:-  0.6
331631692582707
Epoch:-  72    Training loss:-  0.6028733997493327    Validation loss:-  0.6
329042721094521
Epoch:-  73    Training loss:-  0.6024050587065095    Validation loss:-  0.6
327409716237511
Epoch:-  74    Training loss:-  0.6019497888711972    Validation loss:-  0.6
32525273056419
Epoch:-  75    Training loss:-  0.6015085511101819    Validation loss:-  0.6
323764032399776
Epoch:-  76    Training loss:-  0.6010696505272307    Validation loss:-  0.6
321239352822157
Epoch:-  77    Training loss:-  0.6006434009345868    Validation loss:-  0.6
319180256713917
Epoch:-  78    Training loss:-  0.600223881844529    Validation loss:-  0.63
16816557953381
Epoch:-  79    Training loss:-  0.5998148450714139    Validation loss:-  0.6
314661202546591
Epoch:-  80    Training loss:-  0.5994173299296831    Validation loss:-  0.6
313418608829497
Epoch:-  81    Training loss:-  0.599022568286133    Validation loss:-  0.63
10726372519033
Epoch:-  82    Training loss:-  0.5986391409054845    Validation loss:-  0.6
308675121232522
Epoch:-  83    Training loss:-  0.5982646586755134    Validation loss:-  0.6
306671301765195
```

```
Epoch:-  84    Training loss:-  0.5978955735054345    Validation loss:-  0.6
305092030870798
```

In [66]:

```
plt.plot(training_loss_tfidf_c, label='Train_loss')
plt.plot(dev_loss_tfidf_c, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Binary - TFIDF(Training)')
ax = plt.axes()
ax.set_facecolor("lightgray")

plt.legend()

plt.show()
```

```
<ipython-input-66-735fd7e32112>:8: MatplotlibDeprecationWarning: Adding an a
xes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and re
turned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



In [67]:

```
preds_te_tfidf_c = predict_class_c(test_tfidf_c, w_tfidf_c)

print('Accuracy:', accuracy_score(np.array(test_label),preds_te_tfidf_c))
print('Precision:', precision_score(np.array(test_label),preds_te_tfidf_c))
print('Recall:', recall_score(np.array(test_label),preds_te_tfidf_c))
print('F1-Score:', f1_score(np.array(test_label),preds_te_tfidf_c))
```

```
Accuracy: 0.7042606516290727
Precision: 0.7055837563451777
Recall: 0.6984924623115578
F1-Score: 0.7020202020202021
```

BOW+BOCN

In [68]:

```
def extract_ngrams_c_w(x_raw, ngram_range=(1,3), token_pattern=r'\b[A-Za-z]{2,}\b', stop_wo
    tokens = []
    for word in re.findall(token_pattern,x_raw):
        if word.lower() not in stop_words:
            tokens.append(word.lower())

    ngrams_list_c_w = []

    for num in range(0, len(tokens)):                          # Creating tokens by words
        ngram = ' '.join(tokens[num:num + 1])
        ngrams_list_c_w.append(ngram)

    for num in range(0, len(tokens)):
        ngram = ' '.join(tokens[num:num + 2])
        ngrams_list_c_w.append(ngram)

    for num in range(0, len(tokens)):
        ngram = ' '.join(tokens[num:num + 3])
        ngrams_list_c_w.append(ngram)

    for x in range(len(x_raw)):                                # Creating tokens by characters
        n=x_raw[x:x+2]
        ngrams_list_c_w.append(n)
    for x in range(len(x_raw)):
        n=x_raw[x:x+3]
        ngrams_list_c_w.append(n)
    for x in range(len(x_raw)):
        n=x_raw[x:x+4]
        ngrams_list_c_w.append(n)
    x = set(ngrams_list_c_w)                                    # Appending both tokens by words an

    return list(x)
```

In [69]:

```
def get_vocab_c_w(X_raw,
              ngram_range=(1, 3),
              token_pattern=r'\b[A-Za-z]{2,}\b',
              min_df=1,
              keep_topN=0,
              stop_words=stop_words):

    doc_freq_c_w = Counter()              # Using counter so we can count no of times elemen
    ngram_count_c_w = Counter()           # Also counter function provides most common param
                                          # Also counter function provides update paramter t
    for text in X_raw:
        # A list of ngrams for the given document `text`
        ngram_list_c_w = extract_ngrams_c_w(text, ngram_range, token_pattern, stop_words) #
        doc_freq_c_w.update(set(ngram_list_c_w))  # Here we are using set so as to get uniq
        for ngram in ngram_list_c_w:      # Here we are counting ngram frequency
            list=[]
            if doc_freq_c_w[ngram]>=min_df:
                list.append(ngram)
            ngram_count_c_w.update(list)
    vocab_c_w = {ngram for ngram, _ in ngram_count_c_w.most_common(keep_topN)}   # Here we
    return vocab_c_w, doc_freq_c_w, ngram_count_c_w
```

In [70]:

```
vocab_c_w, doc_freq_c_w = get_vocab_c_w(train_text, keep_topN=5000)[0:2]
```

In [71]:

```
len(vocab_c_w)
```

Out[71]:

```
5000
```

In [72]:

```
vocab_c_w
```

Out[72]:

```
{'t f',
 'nit',
 'lp',
 'while',
 'san',
 ' cr',
 'ood ',
 'e an',
 'his',
 'y ) ',
 'ng s',
 "'s c",
 'at b',
 'ille',
 'm i',
 'never',
 'o ,',
 ' lon',
```

In [73]:

```
def generator_train_c_w ():
    return (extract_ngrams_c_w(text, vocab=vocab_c_w)for text in train_text)
```

In [74]:

```
def generator_dev_c_w ():
    return (extract_ngrams_c_w(text, vocab=vocab_c_w)for text in dev_text)
```

In [75]:

```
def generator_test_c_w ():
    return (extract_ngrams_c_w(text, vocab=vocab_c_w)for text in test_text)
```

In [76]:

```
train_texts_ngrams_c_w = generator_train()
dev_texts_ngrams_c_w = generator_dev ()
test_texts_ngrams_c_w = generator_test()
```

In [77]:

```
id2word_c_w = {index: value for index, value in enumerate(vocab_c_w)}      # vocabulary id

word_to_vocab_id_c_w = {k : v for k, v in id2word_c_w.items()}            # word -> vocabu
print(word_to_vocab_id_c_w)
```

```
{0: 't f', 1: 'nit', 2: 'lp', 3: 'while', 4: 'san', 5: ' cr', 6: 'ood ',
7: 'e an', 8: 'his', 9: 'y ) ', 10: 'ng s', 11: "'s c", 12: 'at b', 13: 'i
lle', 14: 'm i', 15: 'never', 16: 'o ,', 17: ' lon', 18: 'his ', 19: ' wh
o', 20: ' mov', 21: ' ne', 22: 's ev', 23: 'il', 24: 'succ', 25: 'e li', 2
6: 'unti', 27: 'ay .', 28: 'ince', 29: ', w', 30: 'more', 31: 'big', 32:
'e si', 33: 'must', 34: 'ma', 35: 'ho ', 36: 'eal ', 37: 's m', 38: 'cro',
39: 'or', 40: 'nd l', 41: 'ie ,', 42: 'ish ', 43: 's ne', 44: " he'", 45:
'lity', 46: ' i ', 47: 'dien', 48: 'opl', 49: 'point', 50: 'cre', 51: 'o b
e', 52: 'ead ', 53: 'fli', 54: 'as m', 55: ' la', 56: 'phe', 57: 'l i', 5
8: 'pro', 59: 'cie', 60: 'e ri', 61: 'itin', 62: 'wat', 63: ' mar', 64: 'r
e f', 65: 'in ,', 66: 'ieve', 67: 'sic', 68: 'ema', 69: 'cove', 70: ' na
t', 71: 'n hi', 72: 'something', 73: 'be t', 74: ' sma', 75: 'al ', 76: 't
ory', 77: 'en a', 78: ' ter', 79: 'irs', 80: 'stea', 81: 'tro', 82: 'ses
', 83: 'ped', 84: 'of f', 85: 'to e', 86: 'plot', 87: 'r .', 88: ' an', 8
9: 'h .', 90: 'nvol', 91: 'ortu', 92: 'sma', 93: 'st s', 94: '( th', 95:
'len', 96: 'ne a', 97: 'o lo', 98: 'd ,', 99: 'es i', 100: 'enta', 101: '
off', 102: 'wr', 103: 'out', 104: 'ces ', 105: 'leav', 106: ' mus', 107:
'y we', 108: 'ula', 109: ' . "', 110: 'ly r', 111: 'is f', 112: 'ne o', 11
3: ' cas', 114: 'n t', 115: 'd ac', 116: 'er m', 117: 'ou', 118: 'hon', 11
```

In [78]:

```
def vectorise_c_w(X_ngram, vocab):
    X_vec_c_w = []
    for ngram_list in X_ngram:                # Here we are creating a function to vectorise
        counter = Counter(ngram_list)          # Here we are using temperory list variable to
        list_c_w = []
        for v in vocab:
            list_c_w.append(counter[v])
        X_vec_c_w.append(list_c_w)
    return np.array(X_vec_c_w)
```

In [79]:

```
train_count_c_w = vectorise_c_w(train_texts_ngrams_c_w, vocab_c_w)

dev_count_c_w = vectorise_c_w(dev_texts_ngrams_c_w, vocab_c_w)

test_count_c_w = vectorise_c_w(test_texts_ngrams_c_w, vocab_c_w)
```

In [80]:

```
train_count_c_w.shape
```

Out[80]:

```
(1399, 5000)
```

In [81]:

```
total_train_docs_c_w = len(train_text)
total_dev_docs_c_w = len(dev_text)
total_test_docs_c_w = len(test_text)

dev_df_c_w = get_vocab_c_w(dev_text, keep_topN=5000)[1]
test_df_c_w = get_vocab_c_w(test_text, keep_topN=5000)[1]
for v in vocab_c_w:            # Reference  --> Lecture Notes
    train_idf_c_w = np.array([np.log10(total_train_docs_c_w/doc_freq_c_w[v])])
for v in vocab_c_w:
    if dev_df_c_w[v]:
        dev_idf_c_w = np.array([np.log10(total_dev_docs_c_w / dev_df_c_w[v])])
for v in vocab_c_w:
    if test_df_c_w[v]:
        test_idf_c_w = np.array([np.log10(total_test_docs_c_w/test_df_c_w[v])])
```

In [82]:

```
Train_norm_c_w = np.log10(1 + train_count_c_w)  # Reference ---> Lecture Notes
Dev_norm_c_w = np.log10(1 + dev_count_c_w)      # squash the raw frequency, by using the lo
Test_norm_c_w = np.log10(1 + test_count_c_w)
```

In [83]:

```
train_tfidf_c_w = Train_norm_c_w * train_idf_c_w        # Calculating Tfidf
dev_tfidf_c_w = Dev_norm_c_w * dev_idf_c_w              # tfidf = tf * idf
test_tfidf_c_w = Test_norm_c_w * test_idf_c_w
```

In [84]:

```
def sigmoid_c_w(z):                     # Reference
    return 1 / (1 + np.exp(-z))      # https://towardsdatascience.com/building-a-logistic-re
```

In [85]:

```
def predict_proba_c_w(X, weights):      # Reference Logic
    z = X.dot(weights)                 # https://pyimagesearch.com/2016/10/17/stochastic-gradi
    return sigmoid_c_w(z)
```

In [86]:

```
def predict_class_c_w(X,weights):
    list = []
    for prob in predict_proba_c_w(X,weights):   # Assignining range if <= 0.5 then assign t
        if prob <= 0.5:
            list.append(0)
        else:
            list.append(1)
    return list
```

In [87]:

```
def binary_loss_c_w(X, Y, weights, alpha=0.00001):
    l = -Y * np.log(predict_proba_c_w(X, weights)) - (1 - Y) * np.log(1 - predict_proba_c_w

    # L2 Regularisation                    # Reference
    l += alpha * weights.dot(weights)      # https://github.com/akashmantry/LogisticRegressio
                                           # Lreg = L + αR(w)    --> Lecture notes
    # Return the average loss
    return np.mean(l)
```

In [88]:

```
def SGD_c_w(X_tr, Y_tr, X_dev, Y_dev, lr=0.1, alpha=0.00001, epochs=5, tolerance=0.0001, pr

    np.random.seed(123)                        # Random seed is fixed here so that we can get
    training_loss_history_c_w = []
    validation_loss_history_c_w = []

    weights_int_list_c_w = []                  # Creating weights with all zeros so that we ca
    for i in range(train_count_c_w.shape[1]):
        weights_int_list_c_w.append(0)
    weights_int_c_w = np.array(weights_int_list_c_w)
    weights_c_w = weights_int_c_w.astype(np.float)

    def zipper(X_tr, Y_tr):                                     # Create training tuples
        size = len(X_tr) if len(X_tr) < len(Y_tr) else len(Y_tr)   # Adding values from two
        retList = []
        for i in range(size):
            retList.append((X_tr[i], Y_tr[i]))
        return retList

    train_docs_c_w = zipper(X_tr, Y_tr)

    for epoch in range(epochs):
        np.random.shuffle(train_docs_c_w)    # Shuffling to randomise all values
                                             # Reference
        for first, second in train_docs_c_w:    # w = w − η∇wL(w;xi;yi)  --> Lecture Notes
            weights_c_w = weights_c_w - lr * (first * (predict_proba_c_w(first, weights_c_w

        # Monitor training and validation loss
        loss_in_training_c_w = binary_loss_c_w(X_tr, Y_tr, weights_c_w, alpha)
        loss_in_dev_c_w = binary_loss_c_w(X_dev, Y_dev, weights_c_w, alpha)

                                          # Reference
                                          # previous validation loss − current validation loss; s
        if epoch > 0 and validation_loss_history_c_w[-1] - loss_in_dev_c_w < tolerance:
            break
        else:
            training_loss_history_c_w.append(loss_in_training_c_w)
            validation_loss_history_c_w.append(loss_in_dev_c_w)

        if print_progress:
            #print(f'Epoch: {epoch} | Training loss: {cur_loss_tr} | Validation loss: {cur_
            print("Epoch:- ",epoch,"  ","Training loss:- ",loss_in_training_c_w,"  ","Valid
    return weights_c_w, training_loss_history_c_w, validation_loss_history_c_w
```

In [89]:

```
w_count_c_w, training_loss_count_c_w, dev_loss_count_c_w = SGD_c_w(X_tr=train_count_c_w,Y_t
```

```
<ipython-input-88-a7f79af50d5e>:11: DeprecationWarning: `np.float` is a depr
ecated alias for the builtin `float`. To silence this warning, use `float` b
y itself. Doing this will not modify any behavior and is safe. If you specif
ically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/d
evdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdocs/rel
ease/1.20.0-notes.html#deprecations)
  weights_c_w = weights_int_c_w.astype(np.float)


Epoch:-  0    Training loss:-  0.6867616762953799    Validation loss:-  0.68
8662994837779
Epoch:-  1    Training loss:-  0.6805971773460067    Validation loss:-  0.68
43968844521979
Epoch:-  2    Training loss:-  0.6746274686149971    Validation loss:-  0.68
0144417580446
Epoch:-  3    Training loss:-  0.6688696422950303    Validation loss:-  0.67
60317758642047
Epoch:-  4    Training loss:-  0.6633003554688951    Validation loss:-  0.67
21490079854291
Epoch:-  5    Training loss:-  0.6579190436566391    Validation loss:-  0.66
83357562193539
Epoch:-  6    Training loss:-  0.652708069921851    Validation loss:-  0.664
7520226582013
Epoch:-  7    Training loss:-  0.6476875003339986    Validation loss:-  0.66
1629564605675
Epoch:-  8    Training loss:-  0.6427935428904774    Validation loss:-  0.65
79495679975379
Epoch:-  9    Training loss:-  0.6380817207680346    Validation loss:-  0.65
50194349984351
Epoch:-  10    Training loss:-  0.633490209212943    Validation loss:-  0.65
16038096941142
Epoch:-  11    Training loss:-  0.6291039194471432    Validation loss:-  0.6
484350345509996
Epoch:-  12    Training loss:-  0.6247632285196985    Validation loss:-  0.6
457145349567553
Epoch:-  13    Training loss:-  0.6206123686762235    Validation loss:-  0.6
431694524223667
Epoch:-  14    Training loss:-  0.616570545733435    Validation loss:-  0.64
01596229146272
Epoch:-  15    Training loss:-  0.6126590560566499    Validation loss:-  0.6
375004468692987
Epoch:-  16    Training loss:-  0.6088642492086557    Validation loss:-  0.6
349291652259604
Epoch:-  17    Training loss:-  0.6051523149204795    Validation loss:-  0.6
325877278185086
Epoch:-  18    Training loss:-  0.6015951747574431    Validation loss:-  0.6
300874937335704
Epoch:-  19    Training loss:-  0.5980882283160485    Validation loss:-  0.6
281110730962193
Epoch:-  20    Training loss:-  0.5946988075261816    Validation loss:-  0.6
258516184296478
Epoch:-  21    Training loss:-  0.5914627284995453    Validation loss:-  0.6
233718479773764
Epoch:-  22    Training loss:-  0.5882085807335026    Validation loss:-  0.6
```

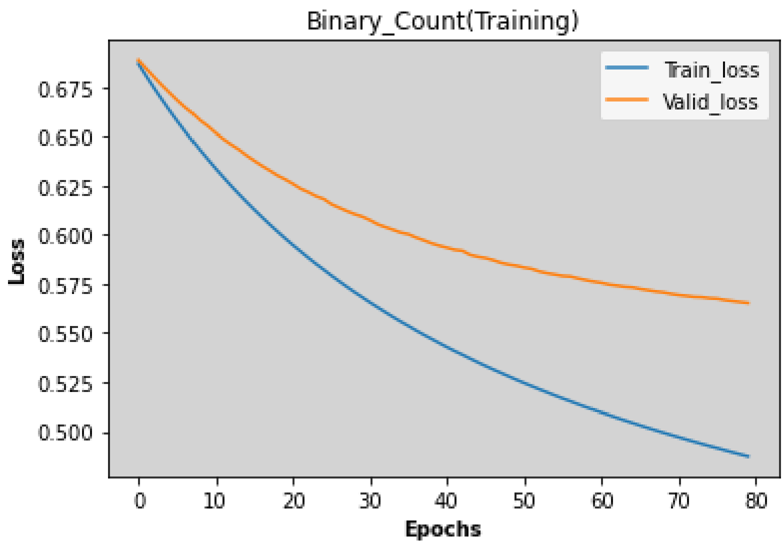
```
216552088358144
Epoch:-  23    Training loss:-  0.5850930112142171    Validation loss:-  0.6
195816096248331
Epoch:-  24    Training loss:-  0.5821583022108827    Validation loss:-  0.6
181383090807161
Epoch:-  25    Training loss:-  0.5791482098963159    Validation loss:-  0.6
155017473182208
Epoch:-  26    Training loss:-  0.5762623364447237    Validation loss:-  0.6
137055591790329
Epoch:-  27    Training loss:-  0.5734464949804937    Validation loss:-  0.6
120066411912459
Epoch:-  28    Training loss:-  0.5707129753882814    Validation loss:-  0.6
104406885078096
Epoch:-  29    Training loss:-  0.5681175705629719    Validation loss:-  0.6
091725443667702
Epoch:-  30    Training loss:-  0.5654810310599871    Validation loss:-  0.6
073639214180108
Epoch:-  31    Training loss:-  0.5629879815334035    Validation loss:-  0.6
052839427583832
Epoch:-  32    Training loss:-  0.5604983064944548    Validation loss:-  0.6
038306002210284
Epoch:-  33    Training loss:-  0.5580649621142231    Validation loss:-  0.6
02481923366284
Epoch:-  34    Training loss:-  0.5557124352510104    Validation loss:-  0.6
010701971209971
Epoch:-  35    Training loss:-  0.5534830319273188    Validation loss:-  0.6
002196303535654
Epoch:-  36    Training loss:-  0.5511746259711399    Validation loss:-  0.5
984922635513193
Epoch:-  37    Training loss:-  0.5489820218948812    Validation loss:-  0.5
971125866760734
Epoch:-  38    Training loss:-  0.5468850902229903    Validation loss:-  0.5
955819495972406
Epoch:-  39    Training loss:-  0.5447826686304875    Validation loss:-  0.5
943774214123583
Epoch:-  40    Training loss:-  0.5427061144831757    Validation loss:-  0.5
933492636292556
Epoch:-  41    Training loss:-  0.5407021400928198    Validation loss:-  0.5
922712453227861
Epoch:-  42    Training loss:-  0.5388685241466262    Validation loss:-  0.5
917774681444494
Epoch:-  43    Training loss:-  0.5368896478296042    Validation loss:-  0.5
897209666419778
Epoch:-  44    Training loss:-  0.5349776713666059    Validation loss:-  0.5
887529008488324
Epoch:-  45    Training loss:-  0.5331258667342416    Validation loss:-  0.5
880671738219826
Epoch:-  46    Training loss:-  0.531322620414796    Validation loss:-  0.58
70000258669776
Epoch:-  47    Training loss:-  0.5296009264937869    Validation loss:-  0.5
856664354604934
Epoch:-  48    Training loss:-  0.5278545684509905    Validation loss:-  0.5
847807360393954
Epoch:-  49    Training loss:-  0.5261447363643502    Validation loss:-  0.5
841580790826334
Epoch:-  50    Training loss:-  0.5244857371950931    Validation loss:-  0.5
832468943823458
Epoch:-  51    Training loss:-  0.5228839737238052    Validation loss:-  0.5
825608237935246
Epoch:-  52    Training loss:-  0.521277896509942    Validation loss:-  0.58
12450315946218
```

```
Epoch:- 53    Training loss:-  0.5197247895941459    Validation loss:-  0.5
80367357089618
Epoch:- 54    Training loss:-  0.5181674734503027    Validation loss:-  0.5
796972510425905
Epoch:- 55    Training loss:-  0.5166592372909627    Validation loss:-  0.5
789655192792105
Epoch:- 56    Training loss:-  0.5152487292584949    Validation loss:-  0.5
787014311760628
Epoch:- 57    Training loss:-  0.5137501020617173    Validation loss:-  0.5
777143269648036
Epoch:- 58    Training loss:-  0.5123152713418281    Validation loss:-  0.5
768903779211736
Epoch:- 59    Training loss:-  0.5109103009001279    Validation loss:-  0.5
76068126330121
Epoch:- 60    Training loss:-  0.5095393746147446    Validation loss:-  0.5
754485778383727
Epoch:- 61    Training loss:-  0.5081978798439779    Validation loss:-  0.5
745569078734861
Epoch:- 62    Training loss:-  0.5068619296677492    Validation loss:-  0.5
740509213627891
Epoch:- 63    Training loss:-  0.50555948026714    Validation loss:-  0.573
4506742333619
Epoch:- 64    Training loss:-  0.5043157554149298    Validation loss:-  0.5
73154872632372
Epoch:- 65    Training loss:-  0.5030267160153955    Validation loss:-  0.5
723492329141104
Epoch:- 66    Training loss:-  0.5017850154809316    Validation loss:-  0.5
717171555470087
Epoch:- 67    Training loss:-  0.5005655275200466    Validation loss:-  0.5
710728492805731
Epoch:- 68    Training loss:-  0.49937266785066725    Validation loss:-  0.
5706135129869576
Epoch:- 69    Training loss:-  0.4981951926942913    Validation loss:-  0.5
698506578387003
Epoch:- 70    Training loss:-  0.49704698076252124    Validation loss:-  0.
5692451698035732
Epoch:- 71    Training loss:-  0.49589635595008    Validation loss:-  0.568
8142794487164
Epoch:- 72    Training loss:-  0.49477432247860914    Validation loss:-  0.
5683067830952089
Epoch:- 73    Training loss:-  0.4936762189884289    Validation loss:-  0.5
681533642083794
Epoch:- 74    Training loss:-  0.4925826898827807    Validation loss:-  0.5
676388011424371
Epoch:- 75    Training loss:-  0.4915478038332081    Validation loss:-  0.5
674086923425322
Epoch:- 76    Training loss:-  0.4904504826608226    Validation loss:-  0.5
666792008689355
Epoch:- 77    Training loss:-  0.4894083185731051    Validation loss:-  0.5
661784494592169
Epoch:- 78    Training loss:-  0.48838420457363463    Validation loss:-  0.
5656643851179889
Epoch:- 79    Training loss:-  0.48737936493338685    Validation loss:-  0.
5651928738968522
```

In [90]:

```
plt.plot(training_loss_count_c_w, label='Train_loss')
plt.plot(dev_loss_count_c_w, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Binary_Count(Training)')
ax = plt.axes()
ax.set_facecolor("lightgray")


plt.legend()

plt.show()
```

```
<ipython-input-90-8bcc12968281>:8: MatplotlibDeprecationWarning: Adding an a
xes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and re
turned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



In [91]:

```
array_test_label_c_w = np.array(test_label) # Changing list to array as accuracy score only
```

In [92]:

```
preds_te_count_c_w = predict_class_c_w(test_count_c_w, w_count_c_w)

print('Accuracy:', accuracy_score(array_test_label_c_w,preds_te_count_c_w))
print('Precision:', precision_score(array_test_label_c_w,preds_te_count_c_w))
print('Recall:', recall_score(array_test_label_c_w,preds_te_count_c_w))
print('F1-Score:', f1_score(array_test_label_c_w,preds_te_count_c_w))
```

```
Accuracy: 0.7192982456140351
Precision: 0.7164179104477612
Recall: 0.7236180904522613
F1-Score: 0.7200000000000001
```

In [93]:

```
w_tfidf_c_w, training_loss_tfidf_c_w, dev_loss_tfidf_c_w = SGD_c_w(X_tr=train_tfidf_c_w,
                                                          Y_tr=np.array(train_label),
                                                          X_dev=dev_tfidf_c_w,
                                                          Y_dev=np.array(dev_label),
                                                          lr=0.00322,
                                                          alpha=0.0005,
                                                          epochs=100)
```

```
Epoch:-  0    Training loss:-  0.6891974926424698    Validation loss:-  0.69
03373432508986
Epoch:-  1    Training loss:-  0.6853670496553458    Validation loss:-  0.68
76460321183888
Epoch:-  2    Training loss:-  0.6816495692504321    Validation loss:-  0.68
50171312064209
Epoch:-  3    Training loss:-  0.6780439691953213    Validation loss:-  0.68
24209757479942

<ipython-input-88-a7f79af50d5e>:11: DeprecationWarning: `np.float` is a depr
ecated alias for the builtin `float`. To silence this warning, use `float` b
y itself. Doing this will not modify any behavior and is safe. If you specif
ically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/d
evdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdocs/rel
ease/1.20.0-notes.html#deprecations)
  weights_c_w = weights_int_c_w.astype(np.float)

Epoch:-  4    Training loss:-  0.6745456536671272    Validation loss:-  0.67
99691079814295
Epoch:-  5    Training loss:-  0.6711524108682186    Validation loss:-  0.67
75533859872013
Epoch:-  6    Training loss:-  0.667857922337171    Validation loss:-  0.675
2793165097853
Epoch:-  7    Training loss:-  0.6646632891414384    Validation loss:-  0.67
32508632111102
Epoch:-  8    Training loss:-  0.6615558623549975    Validation loss:-  0.67
09941230273697
Epoch:-  9    Training loss:-  0.6585467275733858    Validation loss:-  0.66
90670033026984
Epoch:-  10    Training loss:-  0.6556144743540093    Validation loss:-  0.6
669480704477625
Epoch:-  11    Training loss:-  0.6527868784207067    Validation loss:-  0.6
649095360123098
Epoch:-  12    Training loss:-  0.6500180807131299    Validation loss:-  0.6
631277366351428
Epoch:-  13    Training loss:-  0.6473404658174516    Validation loss:-  0.6
614914427873714
Epoch:-  14    Training loss:-  0.6447368374068324    Validation loss:-  0.6
59614712053002
Epoch:-  15    Training loss:-  0.6422114529019294    Validation loss:-  0.6
578943194553456
Epoch:-  16    Training loss:-  0.6397580318038434    Validation loss:-  0.6
562416007971652
Epoch:-  17    Training loss:-  0.6373656161643616    Validation loss:-  0.6
547108558669336
Epoch:-  18    Training loss:-  0.6350551546753361    Validation loss:-  0.6
5313107600437
Epoch:-  19    Training loss:-  0.6327881982145934    Validation loss:-  0.6
518195147935503
Epoch:-  20    Training loss:-  0.6305961951313848    Validation loss:-  0.6
```

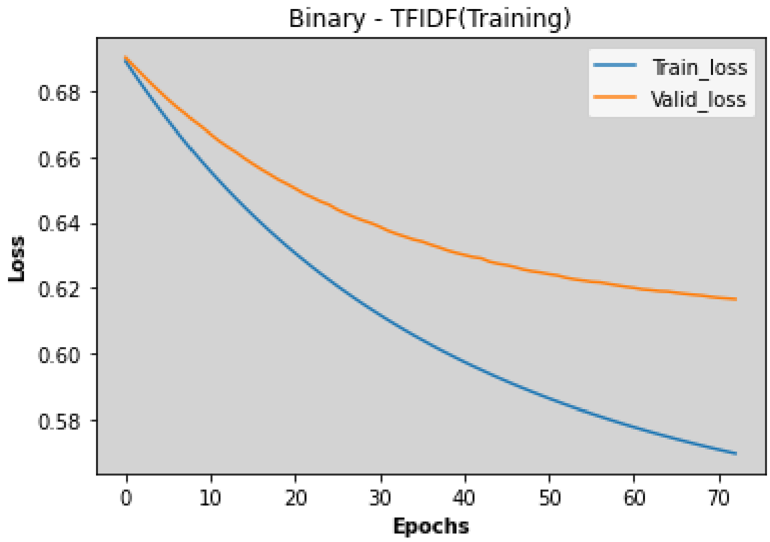
```
504082519035101
Epoch:-  21    Training loss:-  0.6284788911851917    Validation loss:-  0.6
488664460731325
Epoch:-  22    Training loss:-  0.6263929774499469    Validation loss:-  0.6
476843055550392
Epoch:-  23    Training loss:-  0.6243789157811888    Validation loss:-  0.6
464236409064694
Epoch:-  24    Training loss:-  0.6224503005419547    Validation loss:-  0.6
454341081124544
Epoch:-  25    Training loss:-  0.6205156034235068    Validation loss:-  0.6
439030858602147
Epoch:-  26    Training loss:-  0.6186648264049013    Validation loss:-  0.6
427011563243432
Epoch:-  27    Training loss:-  0.6168563272364022    Validation loss:-  0.6
416093591020052
Epoch:-  28    Training loss:-  0.6150978103193234    Validation loss:-  0.6
40617987903677
Epoch:-  29    Training loss:-  0.6134072282900002    Validation loss:-  0.6
397628802456576
Epoch:-  30    Training loss:-  0.61173351849451    Validation loss:-  0.638
6843327495056
Epoch:-  31    Training loss:-  0.6101161411857254    Validation loss:-  0.6
374410005904216
Epoch:-  32    Training loss:-  0.6085435874709294    Validation loss:-  0.6
364841901741802
Epoch:-  33    Training loss:-  0.6070001918947184    Validation loss:-  0.6
356343026820224
Epoch:-  34    Training loss:-  0.6055048167329065    Validation loss:-  0.6
347605338144376
Epoch:-  35    Training loss:-  0.6040616191587256    Validation loss:-  0.6
341563531303621
Epoch:-  36    Training loss:-  0.6026316388706937    Validation loss:-  0.6
332097299765354
Epoch:-  37    Training loss:-  0.6012493849794215    Validation loss:-  0.6
323750884907592
Epoch:-  38    Training loss:-  0.5999145657063153    Validation loss:-  0.6
314496183118206
Epoch:-  39    Training loss:-  0.5986067765610381    Validation loss:-  0.6
306907762318517
Epoch:-  40    Training loss:-  0.597318994272567    Validation loss:-  0.63
00572928950062
Epoch:-  41    Training loss:-  0.5960723145216986    Validation loss:-  0.6
294253455222294
Epoch:-  42    Training loss:-  0.5948915617894784    Validation loss:-  0.6
290337520179067
Epoch:-  43    Training loss:-  0.5936873521815146    Validation loss:-  0.6
27973239541002
Epoch:-  44    Training loss:-  0.5925313512370677    Validation loss:-  0.6
273692610200647
Epoch:-  45    Training loss:-  0.5914029641017373    Validation loss:-  0.6
269226767494822
Epoch:-  46    Training loss:-  0.5903065141183284    Validation loss:-  0.6
263243255049634
Epoch:-  47    Training loss:-  0.5892495104195308    Validation loss:-  0.6
255779417807289
Epoch:-  48    Training loss:-  0.5882036370170088    Validation loss:-  0.6
25054354596652
Epoch:-  49    Training loss:-  0.5871784602377996    Validation loss:-  0.6
24682602242702
Epoch:-  50    Training loss:-  0.5861865865885991    Validation loss:-  0.6
241775953024264
```

```
Epoch:-  51    Training loss:-  0.5852271342319111    Validation loss:-  0.6
23795311802667
Epoch:-  52    Training loss:-  0.5842773418515842    Validation loss:-  0.6
231023077972939
Epoch:-  53    Training loss:-  0.583360780710365    Validation loss:-  0.62
26047462931033
Epoch:-  54    Training loss:-  0.5824549524376215    Validation loss:-  0.6
222337196654332
Epoch:-  55    Training loss:-  0.5815767382920334    Validation loss:-  0.6
218292053620608
Epoch:-  56    Training loss:-  0.5807387313868261    Validation loss:-  0.6
216619565996286
Epoch:-  57    Training loss:-  0.5798926487958017    Validation loss:-  0.6
211919585072774
Epoch:-  58    Training loss:-  0.5790719449309535    Validation loss:-  0.6
207619047925461
Epoch:-  59    Training loss:-  0.5782711918190223    Validation loss:-  0.6
203267340179767
Epoch:-  60    Training loss:-  0.5774936585901443    Validation loss:-  0.6
199907509757269
Epoch:-  61    Training loss:-  0.5767366889201465    Validation loss:-  0.6
195256211698039
Epoch:-  62    Training loss:-  0.575990672558028    Validation loss:-  0.61
92869784736128
Epoch:-  63    Training loss:-  0.5752659068957235    Validation loss:-  0.6
18977663544562
Epoch:-  64    Training loss:-  0.5745730196538625    Validation loss:-  0.6
188618123923215
Epoch:-  65    Training loss:-  0.5738707373205054    Validation loss:-  0.6
184784401732966
Epoch:-  66    Training loss:-  0.5731919719340509    Validation loss:-  0.6
181552204786027
Epoch:-  67    Training loss:-  0.5725317577554059    Validation loss:-  0.6
178756421612964
Epoch:-  68    Training loss:-  0.5718878117764333    Validation loss:-  0.6
176387959442443
Epoch:-  69    Training loss:-  0.5712561688729434    Validation loss:-  0.6
172564103770853
Epoch:-  70    Training loss:-  0.570646006965033    Validation loss:-  0.61
6951171985971
Epoch:-  71    Training loss:-  0.5700411505974223    Validation loss:-  0.6
167477685043423
Epoch:-  72    Training loss:-  0.5694526498515728    Validation loss:-  0.6
165222922624678
```

In [94]:

```
plt.plot(training_loss_tfidf_c_w, label='Train_loss')
plt.plot(dev_loss_tfidf_c_w, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Binary - TFIDF(Training)')
ax = plt.axes()
ax.set_facecolor("lightgray")

plt.legend()

plt.show()
```

```
<ipython-input-94-2895b5c1aa72>:8: MatplotlibDeprecationWarning: Adding an a
xes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and re
turned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



In [95]:

```
preds_te_tfidf_c_w = predict_class_c_w(test_tfidf_c_w, w_tfidf_c_w)

print('Accuracy:', accuracy_score(np.array(test_label),preds_te_tfidf_c_w))
print('Precision:', precision_score(np.array(test_label),preds_te_tfidf_c_w))
print('Recall:', recall_score(np.array(test_label),preds_te_tfidf_c_w))
print('F1-Score:', f1_score(np.array(test_label),preds_te_tfidf_c_w))
```

```
Accuracy: 0.7218045112781954
Precision: 0.72
Recall: 0.7236180904522613
F1-Score: 0.7218045112781956
```

Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count	0.8454106280193237	0.8793969849246231	0.8620689655172413
BOW-tfidf	0.8514851485148515	0.864321608040201	0.85785536159601
BOCN-count	0.7046632124352331	0.6834170854271356	0.6938775510204082
BOCN-tfidf	0.7114427860696517	0.7185929648241206	0.7150000000000001
BOW+BOCN-count	0.7164179104477612	0.7236180904522613	0.7200000000000001
BOW+BOCN-tfidf	0.72	0.7236180904522613	0.7218045112781956

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

My model is having a great accuracy score, precision score, Recall score and F1-score in all three sections which implies that it is predicting better results and giving output at its best. Thats why I think my model is performing better than the rest.