Assignment 2: Naïve Bayes and Logistic Regression

Part 1: Implement Naïve Bayes from scratch to classify women's clothing e-commerce reviews. Generate a bag of words model to produce a numeric sequence of numbers from text in the reviews, implement a derived multinomial Naïve Bayes model, and tune the hyperparameter, beta, of the model to achieve high ROC_AUC.

Python file on github: naïve_bayes.py

```
import re
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from collections import Counter
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import f1_score, precision_score, recall_score, roc_auc_score, accuracy_score

class BagOfWords(object):
    """
    Class for implementing Bag of Words
     for Q1.1
    """
    def __init__(self, vocabulary_size):
        """
        Initialize the BagOfWords model
        """
        self.vocabulary_size = vocabulary_size

    def preprocess(self, text):
        """
        text: a string of words from Review Text feature

        Preprocessing of one Review Text
            - convert to lowercase
            - remove punctuation
            - empty spaces
            - remove 1-letter words
            - split the sentence into words

        Return the split words
        """
        #covnert to lowercase
        lc_text = text.lower()
        #remove punctuation
        no_punc_text = re.sub(r'[^\w\s]', '', lc_text)
        #empty spaces
        no_sp_text = re.sub(' +', ' ', no_punc_text)
        #remove 1 letter words
        no_one_letter = re.sub('(\\b[A-Za-z] \\b|\\b [A-Za-z]\\b)', '', no_sp_text)
        split_text = no_one_letter.split()
        return split_text

    def fit(self, X_train):
    #for testing
    #def fit(X_train):
        """
        Building the vocabulary using X_train
        """
        rows = len(X_train)
        #create an empty vocabulary list
        vocab_list = {}
        #loop over all rows
        for row in range(1, rows):
            #assign review text string
            text = X_train[row]
            #split the text
            split_text = self.preprocess(text)
            #loop over all words in the current review text
            for word in split_text:
                #check if the word is in the vocabulary list. If it is not, add to list
                if word not in vocab_list:
                    vocab_list[word] = 1
```

```
                #and if it is, add increase the count of that word by 1
                else:
                    vocab_list[word] += 1
        #Now we have to extract the top 10 most frequent words in the vocabulary list
        sorted_vocab_list = sorted(vocab_list, key=vocab_list.get, reverse = True)[:10]
        #put the top 10 most frequent words in alphabetical order
        alph_vocab_list = sorted(sorted_vocab_list)
        #convert from a list to a dictionary, which will make it easy to get indices of strings
        self.alphabetized_vocab_list = dict(zip(alph_vocab_list,range(len(alph_vocab_list))))
        pass

    def transform(self, X):
        """
        Transform the texts into word count vectors (representation matrix)
            using the fitted vocabulary
        """
        #create an empty matrix to store vocab words in strings of X
        representation = np.zeros((100,10))
        #loop over rows of X
        for row in range(len(X)):
            #peprocess row of x
            split_text = self.preprocess(X[row])
            #loop over words in current row,
            for word in split_text:
                if word in self.alphabetized_vocab_list:
                    word_index = self.alphabetized_vocab_list[word]
                    #add a count to the representation matrix
                    representation[row, word_index] +=1
        #add up the counts of vocabulary in the X array
        summed_representation = sum(representation).reshape((1,-1))
        return self.alphabetized_vocab_list.keys(), summed_representation

class NaiveBayes(object):
    def __init__(self, beta=1, n_classes=2):
        """
        Initialize the Naive Bayes model
            w/ beta and n_classes
        """
        self.beta = beta
        self.n_classes = n_classes

    def fit(self, X_train, y_train):
        """
        Fit the model to X_train, y_train
            - build the conditional probabilities
            - and the prior probabilities
        """
        #build a bag of words from X_train, which will contain all vocabulary present in the training set
        #create vectorizer
        X_train_array = X_train.toarray()
        #calcualate the priors
        self.prior_negative, self.prior_positive = np.bincount(y_train)/len(y_train)

        #caculate the number of words that correspond to each class
        negative_word_counts = sum(X_train_array[y_train==0])
        positive_word_counts = sum(X_train_array[y_train==1])
        #sum up all of the words in the positive and negative classes
        total_negative_word_count = np.sum(negative_word_counts)
        total_positive_word_count = np.sum(positive_word_counts)
        #calculate the liklihoods
        negative_probability = (negative_word_counts + self.beta)/(total_negative_word_count +
(X_train_array.shape[1] * self.beta))
        positive_probability = (positive_word_counts + self.beta)/(total_positive_word_count +
(X_train_array.shape[1] * self.beta))

        #concatenate liklihood estimates into one array, where column 0 represents the negative class and column
1 represents the positive class
        self.cond_probabilities = np.stack((negative_probability, positive_probability))
        #cond_probabilities = np.stack((negative_probability, positive_probability))
        return

    def predict(self, X_test):
        """
        Predict the X_test with the fitted model
        """
        #create array of the bag of words for X_test
        X_test_array = X_test.toarray()
        #initialize a list of prediction values
        y_pred = []
        #loop over the the array representation of X_test
        for row in X_test_array:
```

```
            #get the nonzero indices of the row of X_test array
            non_zero_idx = np.array(np.nonzero(row))
            #and the values of those indices
            non_zero_vals = row[non_zero_idx]
            #calculate the probability that the row belongs to each class, 0 or 1
            neg_post_prob = self.prior_negative *
np.prod(self.cond_probabilities[0][non_zero_idx]**non_zero_vals)
            pos_post_prob = self.prior_positive *
np.prod(self.cond_probabilities[1][non_zero_idx]**non_zero_vals)
            #get the argument of the higher probability
            y_pred_instance = np.argmax([neg_post_prob, pos_post_prob])
            #and append the prediction vector
            y_pred.append(y_pred_instance)

        y_pred = np.array(y_pred)

        return y_pred

def confusion_matrix(y_true, y_pred):
    """
    Calculate the confusion matrix of the
        predictions with true labels
    """
    #creating a confusion matrix
    y_true = pd.Series(y_true, name = 'Actual')
    y_pred = pd.Series(y_pred, name = 'Predicted')
    confusion_matrix = pd.crosstab(y_true, y_pred, rownames=['Actual'], colnames=['Predicted'], margins=True)
    return confusion_matrix

def load_data(return_numpy=False):
    """
    Load data

    Params
    ------
    return_numpy:   when true return the representation of Review Text
                    using the CountVectorizer or BagOfWords
                    when false return the Review Text
    Return
    ------
    X_train
    y_train
    X_valid
    y_valid
    X_test
    """
    X_train = pd.read_csv("Data/X_train.csv")['Review Text'].values
    X_valid = pd.read_csv("Data/X_val.csv")['Review Text'].values
    X_test  = pd.read_csv("Data/X_test.csv")['Review Text'].values
    y_train = (pd.read_csv("Data/Y_train.csv")['Sentiment'] == 'Positive').astype(int).values
    y_valid = (pd.read_csv("Data/Y_val.csv")['Sentiment'] == 'Positive').astype(int).values

    if return_numpy:
        # To do (not for Q1.1, used in Q1.3)
        # transform the Review Text into bag of word representation using vectorizer
        # process X_train, X_valid, X_test
        vectorizer = CountVectorizer()
        #fit the vectorizer so that we can get a list of all strings in the vocab list
        global_vectorizer = vectorizer.fit(X_train)
        X_train = global_vectorizer.transform(X_train)
        X_valid = global_vectorizer.transform(X_valid)
        X_test = global_vectorizer.transform(X_test)
        pass

    return X_train, y_train, X_valid, y_valid, X_test


def main():
    #create a dictionary to store the metrics
    metrics = ['Beta','ROC_AUC','F1 score', 'Accuracy', 'Precision', 'Recall']
    results = dict([(key, []) for key in metrics])

    #for tuning beta hyperparameter
    #beta_list = [0.1, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    beta_list = [1]

    # Load in data
    X_train, y_train, X_valid, y_valid, X_test = load_data(return_numpy=False)

    # Fit the Bag of Words model for Q1.1
    bow = BagOfWords(vocabulary_size=10)
```

```
    bow.fit(X_train[:100])
    vocab_list, summed_representation = bow.transform(X_train[100:200])

    print('Words in Vocabulary List for Question 1.1:\n', vocab_list)
    print('Word Count of Vocabulary List for Question 1.1:\n', summed_representation)

    # Load in data
    X_train, y_train, X_valid, y_valid, X_test = load_data(return_numpy=True)

    # Fit the Naive Bayes model for Q1.3
    for n in beta_list:

        nb = NaiveBayes(beta=n)
        nb.fit(X_train, y_train)
        y_pred = nb.predict(X_valid)
        #for predicting on the test set
        #y_pred = nb.predict(X_test)
        print("Beta: ", n)
        print(confusion_matrix(y_valid, y_pred))

        calculate and store performance metrics
        results['Beta'].append(n)
        results['F1 score'].append(f1_score(y_valid, y_pred))
        results['Precision'].append(precision_score(y_valid, y_pred))
        results['Recall'].append(recall_score(y_valid, y_pred))
        results['ROC_AUC'].append(roc_auc_score(y_valid, y_pred))
        results['Accuracy'].append(accuracy_score(y_valid,y_pred))

    print(results)

    #Plot beta and ROC_AUC score
    plt.plot(results['Beta'], results['ROC_AUC'])
    plt.scatter(results['Beta'], results['ROC_AUC'])
    plt.xlabel("Beta Values")
    plt.ylabel("ROC AUC Score")
    plt.title("ROC AUC Score for Various\nValues of Beta")
    plt.savefig("Submission/Figures/roc_auc_scores_for_param_combos.png")
    plt.show()

    return y_pred

if __name__ == '__main__':
     y_pred = main()
```

Part 2: Implement L2 regularized logistic regression from scratch. Use a training and validation set to optimize hyperparameters, and classify the ratings dataset from part 1 with by using CountVectorizer.

Python script on github: logistic_regression.py

```
from __future__ import print_function

import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import roc_auc_score

def getauc(y_true, probs):
    """
    Use sklearn roc_auc_score to get the auc given predicted probabilities and the true labels

    Args:
        - y_true: The true labels for the data
        - probs: predicted probabilities
    """
    return roc_auc_score(y_true, probs)


def conf_mat(y_true, y_pred):
    """
```

```
    The method for calculating confusion matrix, you have to implement this by yourself.

    Args:
        - y_true: the true labels for the data
        - y_pred: the predicted labels
    """
    y_true = pd.Series(y_true, name = 'Actual')
    y_pred = pd.Series(y_pred, name = 'Predicted')
    confusion_matrix = pd.crosstab(y_true, y_pred, rownames=['Actual'], colnames=['Predicted'], margins=True)
    return confusion_matrix

class LogisticRegression(object):
  def __init__(self, input_size, reg=0.0, std=1e-4):
    """
    Initializing the weight vector

    Input:
    - input_size: the number of features in dataset, for bag of words it would be number of unique words in your
training data
    - reg: the l2 regularization weight
    - std: the variance of initialized weights
    """
    self.W = std * np.random.randn(input_size)
    self.reg = reg

  def sigmoid(self, x):
    """
    Compute sigmoid of x

    Input:
    - x: Input data of shape (N,)

    Returns:
    - sigmoid of each value in x with the same shape as x (N,)
    """
    sigmoid = 1/(1 + np.exp(-x))
    return sigmoid

  def loss(self, X, y):
    """
    Compute the loss and gradients for your logistic regression.

    Inputs:
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: A numpy array f shape (N,) giving training labels.

    Returns:
    - loss: Loss (data loss and regularization loss) for the entire training samples
    - dLdW: gradient of loss with respect to W
    """
    N, D = X.shape
    reg = self.reg

    #TODO: Compute scores

    #TODO: Compute the loss
    #First term  (-y*log(y_hat))
    #inner = np.reshape(np.dot(X,self.W), -1)
    inner = np.dot(X,self.W)
    y_hat = self.sigmoid(inner)
    log_y_hat = np.log(y_hat)
    first_term = y*log_y_hat

    #second term (1-y)*log(1-y_hat)
    second_inner = 1-y_hat
    second_inner = np.clip(second_inner, 0.0001, 0.9999)
    second_term = (1-y)*np.log(second_inner)

    #third term
    third_term = reg*((np.linalg.norm(self.W))**2)

    #compute the loss
    loss = (1/N)*(-(np.sum(first_term + second_term))) + third_term

    #loss = (1/N)*np.sum(-(y*np.log([self.sigmoid(scores @ np.transpose(X))]))+((1-y)*np.log(1-
self.sigmoid(scores*np.transpose(X))))+(reg*(np.linalg.norm(scores))**2))

    #TODO: Compute gradients
    # Calculate dLdW meaning the gradient of loss function according to W
    # you can use chain rule here with calculating each step to make your job easier
    dLdW = (1/N) * np.dot(np.transpose(X),(y_hat - y)) + ((reg/N)*self.W)
```

```
    return loss, dLdW

  def gradDescent(self,X, y, learning_rate, num_epochs):
    """
    We will use Gradient Descent for updating our weights, here we used gradient descent instead of stochastic
gradient descent for easier implementation
    so you will use the whole training set for each update of weights instead of using batches of data.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - num_epochs: integer giving the number of epochs to train

    Returns:
    - loss_hist: numpy array of size (num_epochs,) giving the loss value over epochs
    """
    N, D = X.shape
    loss_hist = np.zeros(num_epochs)
    X = X.toarray()
    for i in range(num_epochs):
      #TODO: implement steps of gradient descent
      #compute the loss and gradient
      loss, dLdW = self.loss(X, y)
      #adjust the theta parameters based on learning rate and gradient
      self.W = self.W - learning_rate*dLdW
      #add the loss to the loss_hist array
      loss_hist[i] = loss
      # printing loss, you can also print accuracy here after few iterations to see how your model is doing
      print("Epoch : ", i, " loss : ", loss)

    return loss_hist

  def predict(self, X):
    """
    Use the trained weights to predict labels for data given as X

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.

    Returns:
        - probs: A numpy array of shape (N,) giving predicted probabilities for each of the elements of X.
        - y_pred: A numpy array of shape (N,) giving predicted labels for each of the elements of X. You can get
this by putting a threshold of 0.5 on probs
    """
    #TODO: get the scores (probabilities) for input data X and calculate the labels (0 and 1 for each input
data) and return them

    X = X.toarray()
    #X is (N,D)
    #self.W is (D,)
    weighted_X = X @ self.W
    #use the sigmoid function to calculate the probability
    probs = self.sigmoid(weighted_X)
    #create an empty array to store predictions
    y_pred = np.zeros(X.shape[0])

    #assign to class 0 or 1 based on the probability value
    for s in range(len(probs)):
        if probs[s] >=0.5:
            y_pred[s] = 1

    return probs, y_pred

def load_data(return_numpy=True):
    """
    Load data

    Params
    ------

    Return
    ------
    X_train
    y_train
    X_valid
    y_valid
    X_test
```

```
    """
    # If you implemented the load_data function for Q1.3
    # you can use the same function here.

    # TODO: Preprocess the data, here we will only select Review Text column in both train and validation and
    #       use CountVectorizer from sklearn to get bag of word representation of the review texts
    #       Careful that you should fit vectorizer only on train data and use the same vectorizer for
    #       transforming X_train, X_val and X_test

    X_train = pd.read_csv("Data/X_train.csv")['Review Text'].values
    X_valid = pd.read_csv("Data/X_val.csv")['Review Text'].values
    X_test  = pd.read_csv("Data/X_test.csv")['Review Text'].values
    y_train = (pd.read_csv("Data/Y_train.csv")['Sentiment'] == 'Positive').astype(int).values
    y_valid = (pd.read_csv("Data/Y_val.csv")['Sentiment'] == 'Positive').astype(int).values

    if return_numpy:
        vectorizer = CountVectorizer()
        #fit the vectorizer so that we can get a list of all strings in the vocab list
        global_vectorizer = vectorizer.fit(X_train)
        X_train = global_vectorizer.transform(X_train)
        X_valid = global_vectorizer.transform(X_valid)
        X_test = global_vectorizer.transform(X_test)
        pass

    return X_train, y_train, X_valid, y_valid, X_test

def main():
    # Load in data
    X_train, y_train, X_valid, y_valid, X_test = load_data()

    #hyperparameters
    reg = 0.2
    num_epochs = 50
    learning_rate = 0.01


    #create a list of lists to store hyperparameters to test out, where c_n is a combination of
    #the regularization parameter, num_epochs, and learning_rate in that order. this was implemented
    #before testing some optimal hyperparameter combinations
    #c1 = [0, 50, 0.01]
    #c2 = [0.1, 50, 0.01]
    #c3 = [0.1, 150, 0.01]
    #c4 = [0.1, 150, 0.001]
    #c5 = [0.2, 150, 0.01]
    #c6 = [0.1, 300, 0.001]
    #hyperparameter_list = [c1, c2, c3, c4, c5, c6]


    #Hyperparameter configurations for Question 1
    #c1 = [0.01, 1500, 0.1]
    #c2 = [0.01, 1500, 1]
    #c3 = [0.01, 1500, 10]
    #c4 = [0.15, 1500, 0.1]
    #c5 = [0.15, 1500, 1]
    #c6 = [0.15, 1500, 10]
    #hyperparameter_list = [c1, c2, c3, c4, c5, c6]

    #Note: for turning this assignment in, I kept my for loop, but I adjusted the hyperparameters
    #for my best ROC_AUC score
    c1 = [0.01, 1500, 1.0]
    hyperparameter_list = [c1]

    #y_predictions = np.zeros((len(y_valid), 6))
    roc_auc_scores = []
    #loop over list of combinations
    for i in range(len(hyperparameter_list)):
        #assign hyperparameters
        reg = hyperparameter_list[i][0]
        num_epochs = hyperparameter_list[i][1]
        learning_rate = hyperparameter_list[i][2]

        #generate model
        LR = LogisticRegression(input_size=X_train.shape[1], reg=reg)
        #run gradient descent function
        loss_hist = LR.gradDescent(X_train, y_train, learning_rate, num_epochs)
        #generate ROC_AUC score
        probs, y_pred = LR.predict(X_train)
        # Write a for loop for each hyper parameter here each time initialize logistic regression train it on
the train data and get auc on validation data and get confusion matrix using the best hyper params
        roc_auc = getauc(y_train, probs)
        print('ROC_AUC:', roc_auc)
```

```
        roc_auc_scores.append(roc_auc)
        #epochs = np.array(range(0, num_epochs))
        #y_predictions[:,i] = y_pred

        #Before the actual tuning of hyperparameters for Question 3.1, I tested
        #how some different hyperparameter values affected the loss function
        #plt.plot(epochs, loss_hist)
        #plt.scatter(epochs, loss_hist)
        #plt.xlabel("Number of Epochs")
        #plt.ylabel("Loss")
        #plt.title("Loss at Each Epoch\nreg: "+str(reg)+", num_epochs: "+str(num_epochs)+", learning_rate: 
"+str(learning_rate))
        #plt.show()
        #plt.savefig("plots/hyperparameters_{i}.png".format(i=i))
        #plt.clf()

    hyperparameters_count = list(range(len(hyperparameter_list)))
    plt.plot(hyperparameters_count, roc_auc_scores)
    plt.title('ROC_AUC Scores for Combinations of Hyperparameters')
    plt.xlabel('Hyperparameter Combination')
    plt.ylabel('ROC_AUC Score')
    plt.savefig("plots/roc_auc_scores_for_param_combos.png")

    print(roc_auc_scores)
    return roc_auc_scores, y_pred



if __name__ == '__main__':
    roc_auc_scores, y_pred = main()
```