MITx_6.86x_Project_1_AutomaticReviewAnalyzer

July 7, 2019

0.1 Project 1: Automatic Review Analyzer

0.1.1 1. Introduction

The goal of this project is to design a classifier to use for sentiment analysis of product reviews. Our training set consists of reviews written by Amazon customers for various food products. The reviews, originally given on a 5 point scale, have been adjusted to a +1 or -1 scale, representing a positive or negative review, respectively. Below are two example entries from our dataset. Each entry consists of the review and its label. The two reviews were written by different customers describing their experience with a sugar-free candy.

Review	label
Nasty No flavor. The candy is just red, No flavor. Just plan and chewy. I would never buy them again	1
YUMMY! You would never guess that they're sugar-free and it's so great that you can eat them pretty much guilt free! i was so impressed that i've ordered some for myself (w dark chocolate) to take to the office. These are just EXCELLENT!	1

In order to automatically analyze reviews, you will need to complete the following tasks:

Implement and compare three types of linear classifiers: the perceptron algorithm, the average perceptron algorithm, and the Pegasos algorithm.

Use your classifiers on the food review dataset, using some simple text features.

Experiment with additional features and explore their impact on classifier performance.

Setup Details: For this project and throughout the course we will be using Python 3.6 with some additional libraries. We strongly recommend that you take note of how the NumPy numerical library is used in the code provided, and read through the on-line NumPy tutorial. NumPy arrays are much more efficient than Python's native arrays when doing numerical computation. In addition, using NumPy will substantially reduce the lines of code you will need to write.

Note on software: For this project, you will need the NumPy numerical toolbox, and the matplotlib plotting toolbox.

Download sentiment_analysis.tar.gz and untar it in to a working directory. The sentiment_analysis folder contains the various data files in .tsv format, along with the following python

files:

project1.py contains various useful functions and function templates that you will use to implement your learning algorithms.

main.py is a script skeleton where these functions are called and you can run your experiments.

utils.py contains utility functions that the staff has implemented for you.

test.py is a script which runs tests on a few of the methods you will implement. Note that these tests are provided to help you debug your implementation and are not necessarily representative of the tests used for online grading. Feel free to add more test cases locally to further validate the correctness of your code before submitting to the online graders in the codeboxes.

Tip: Throughout the whole online grading system, you can assume the NumPy python library is already imported as np. In some problems you will also have access to python's random library, and other functions you've already implemented.

This project will unfold both on MITx and on your local machine. You are welcome to implement functions locally and run test.py to validate basic functionality, and then copy+paste your code into the MITx code boxes to fully check correctness and receive your grade for individual function implementations. Alternatively, you can also implement the functions online first and after finishing, copy+paste the solution to your local project1.py file. Be wary of the number of attempts you have for each problem, especially if you choose the second development flow.

How to Test Locally: In your terminal, navigate to the directory where your project files reside. Execute the command python test.py to run all the available tests.

How to Run your Project 1 Functions Locally: In your terminal, enter python main.py. You will need to uncomment/comment the relevant code as you progress through the project.

utils.py

[ ]:
```
import csv
import numpy as np
import matplotlib.pyplot as plt

import project1 as p1
import sys

if sys.version_info[0] < 3:
    PYTHON3 = False
else:
    PYTHON3 = True

def load_toy_data(path_toy_data):
    """
    Loads the 2D toy dataset as numpy arrays.
    Returns the tuple (features, labels) in which features is an Nx2 numpy␣
 ↪matrix and
    labels is a length-N vector of +1/-1 labels.
    """
    labels, xs, ys = np.loadtxt(path_toy_data, delimiter='\t', unpack=True)
    return np.vstack((xs, ys)).T, labels

def load_data(path_data, extras=False):
```

```
    """
    Returns a list of dict with keys:
    * sentiment: +1 or -1 if the review was positive or negative, respectively
    * text: the text of the review

    Additionally, if the `extras` argument is True, each dict will also include␣
 ↪the
    following information:
    * productId: a string that uniquely identifies each product
    * userId: a string that uniquely identifies each user
    * summary: the title of the review
    * helpfulY: the number of users who thought this review was helpful
    * helpfulN: the number of users who thought this review was NOT helpful
    """

    global PYTHON3

    basic_fields = {'sentiment', 'text'}
    numeric_fields = {'sentiment', 'helpfulY', 'helpfulN'}

    data = []
    if PYTHON3:
        f_data = open(path_data, encoding="latin1")
    else:
        f_data = open(path_data)

    for datum in csv.DictReader(f_data, delimiter='\t'):
        for field in list(datum.keys()):
            if not extras and field not in basic_fields:
                del datum[field]
            elif field in numeric_fields and datum[field]:
                datum[field] = int(datum[field])

        data.append(datum)

    f_data.close()

    return data

def write_predictions(path_submit_data, preds):
    if PYTHON3:
        f_data = open(path_submit_data, encoding="latin1")
    else:
        f_data = open(path_submit_data)

    reader = csv.DictReader(f_data, delimiter='\t')
    data = list(reader)
```

```
    assert len(preds) == len(data), \
           'Expected {} predictions but {} were given.'.format(len(data),␣
 ↪len(preds))

    for pred, datum in zip(preds.astype(int), data):
        assert pred == 1 or pred == -1, 'Invalid prediction: {}.'.format(pred)
        datum['sentiment'] = pred
    f_data.close()

    if PYTHON3:
        f_out = open(path_submit_data, 'w')
    else:
        f_out = open(path_submit_data, 'wb')

    writer = csv.DictWriter(f_out, delimiter='\t', fieldnames=reader.fieldnames)
    writer.writeheader()
    for datum in data:
        writer.writerow(datum)
    f_out.close()

def plot_toy_data(algo_name, features, labels, thetas):
    """
    Plots the toy data in 2D.
    Arguments:
    * features - an Nx2 ndarray of features (points)
    * labels - a length-N vector of +1/-1 labels
    * thetas - the tuple (theta, theta_0) that is the output of the learning␣
 ↪algorithm
    * algorithm - the string name of the learning algorithm used
    """
    # plot the points with labels represented as colors
    plt.subplots()
    colors = ['b' if label == 1 else 'r' for label in labels]
    plt.scatter(features[:, 0], features[:, 1], s=40, c=colors)
    xmin, xmax = plt.axis()[:2]

    # plot the decision boundary
    theta, theta_0 = thetas
    xs = np.linspace(xmin, xmax)
    ys = -(theta[0]*xs + theta_0) / (theta[1] + 1e-16)
    plt.plot(xs, ys, 'k-')

    # show the plot
    algo_name = ' '.join((word.capitalize() for word in algo_name.split(' ')))
    plt.suptitle('Classified Toy Data ({})'.format(algo_name))
    plt.show()
```

```
def plot_tune_results(algo_name, param_name, param_vals, acc_train, acc_val):
    """
    Plots classification accuracy on the training and validation data versus
    several values of a hyperparameter used during training.
    """
    # put the data on the plot
    plt.subplots()
    plt.plot(param_vals, acc_train, '-o')
    plt.plot(param_vals, acc_val, '-o')

    # make the plot presentable
    algo_name = ' '.join((word.capitalize() for word in algo_name.split(' ')))
    param_name = param_name.capitalize()
    plt.suptitle('Classification Accuracy vs {} ({})'.format(param_name,␣
 ↪algo_name))
    plt.legend(['train','val'], loc='upper right', title='Partition')
    plt.xlabel(param_name)
    plt.ylabel('Accuracy (%)')
    plt.show()

def tune(train_fn, param_vals, train_feats, train_labels, val_feats,␣
 ↪val_labels):
    train_accs = np.ndarray(len(param_vals))
    val_accs = np.ndarray(len(param_vals))

    for i, val in enumerate(param_vals):
        theta, theta_0 = train_fn(train_feats, train_labels, val)

        train_preds = p1.classify(train_feats, theta, theta_0)
        train_accs[i] = p1.accuracy(train_preds, train_labels)

        val_preds = p1.classify(val_feats, theta, theta_0)
        val_accs[i] = p1.accuracy(val_preds, val_labels)

    return train_accs, val_accs

def tune_perceptron(*args):
    return tune(p1.perceptron, *args)

def tune_avg_perceptron(*args):
    return tune(p1.average_perceptron, *args)

def tune_pegasos_T(best_L, *args):
    def train_fn(features, labels, T):
        return p1.pegasos(features, labels, T, best_L)
    return tune(train_fn, *args)
```

```
def tune_pegasos_L(best_T, *args):
    def train_fn(features, labels, L):
        return p1.pegasos(features, labels, best_T, L)
    return tune(train_fn, *args)

def most_explanatory_word(theta, wordlist):
    """Returns the word associated with the bag-of-words feature having largest␣
 ↪weight."""
    return [word for (theta_i, word) in sorted(zip(theta, wordlist))[::-1]]
```

Part I

[ ]:
```
import numpy as np
import csv
import sys
import random
import pandas as pd
import matplotlib.pyplot as plt

from string import punctuation, digits

if sys.version_info[0] < 3:
    PYTHON3 = False
else:
    PYTHON3 = True
```

0.1.2 2. Hinge Loss

In this project you will be implementing linear classifiers beginning with the Perceptron algorithm. You will begin by writing your loss function, a hinge-loss function. For this function you are given the parameters of your model and 0. Additionally, you are given a feature matrix in which the rows are feature vectors and the columns are individual features, and a vector of labels representing the actual sentiment of the corresponding feature vector.

[ ]:
```
#--------------------------------------------------------------------------
# Data loading. There is no need to edit code in this section.
#--------------------------------------------------------------------------

train_data = load_data('reviews_train.tsv')
val_data = load_data('reviews_val.tsv')
test_data = load_data('reviews_test.tsv')

train_texts, train_labels = zip(*((sample['text'], sample['sentiment']) for␣
 ↪sample in train_data))
val_texts, val_labels = zip(*((sample['text'], sample['sentiment']) for sample␣
 ↪in val_data))
```

```
test_texts, test_labels = zip(*((sample['text'], sample['sentiment']) for␣
↪sample in test_data))

dictionary = bag_of_words_original(train_texts)

train_bow_features = extract_bow_feature_vectors_original(train_texts,␣
↪dictionary)
val_bow_features = extract_bow_feature_vectors_original(val_texts, dictionary)
test_bow_features = extract_bow_feature_vectors_original(test_texts, dictionary)
```

[ ]:
```
#pragma: coderesponse template
def get_order(n_samples):
    try:
        with open(str(n_samples) + '.txt') as fp:
            line = fp.readline()
            return list(map(int, line.split(',')))
    except FileNotFoundError:
        random.seed(1)
        indices = list(range(n_samples))
        random.shuffle(indices)
        return indices
#pragma: coderesponse end
```

1) Hinge Loss on One Data Sample First, implement the basic hinge loss calculation on a single data-point. Instead of the entire feature matrix, you are given one row, representing the feature vector of a single data sample, and its label of +1 or -1 representing the ground truth sentiment of the data sample.

Reminder: You can implement this function locally first, and run python test.py in your sentiment_analysis directory to validate basic functionality before checking against the online grader here.

Available Functions: You have access to the NumPy python library as np; No need to import anything.

[ ]:
```
#pragma: coderesponse template
def hinge_loss_single(feature_vector, label, theta, theta_0):
    """
    Finds the hinge loss on a single data point given specific classification
    parameters.

    Args:
        feature_vector - A numpy array describing the given data point.
        label - A real valued number, the correct classification of the data␣
↪point.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.

    Returns: A real number representing the hinge loss associated with the
```

```
    given data point and parameters.

    feature_vector = np.array([1, 2])
    label, theta, theta_0 = 1, np.array([-1, 1]), -0.2
    exp_res = 1 - 0.8

    """
    loss = max(0, 1-label * (np.dot(feature_vector, theta) + theta_0))
    return loss

  #  for cell in feature_vector:
  #      print(theta, feature_vector, cell, np.dot(theta, cell), np.dot(theta,␣
 ↪cell)+theta_0)

#pragma: coderesponse end
```

[ ]:
```
feature_vector = np.array([1, 2])
label, theta, theta_0 = 1, np.array([-1, 1]), -0.2
hinge_loss_single(feature_vector, label, theta, theta_0)
```

2) The Complete Hinge Loss Now it's time to implement the complete hinge loss for a full set of data. Your input will be a full feature matrix this time, and you will have a vector of corresponding labels. The k^{th} row of the feature matrix corresponds to the k^{th} element of the labels vector. This function should return the appropriate loss of the classifier on the given dataset. Available Functions: You have access to the NumPy python library as np, and your previous function as hinge_loss_single

Available Functions: You have access to the NumPy python library as np; No need to import anything.

[ ]:
```
#pragma: coderesponse template
def hinge_loss_full(feature_matrix, labels, theta, theta_0):
    """
    Finds the total hinge loss on a set of data given specific classification
    parameters.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.


    Returns: A real number representing the hinge loss associated with the
    given dataset and parameters. This number should be the average hinge
    loss across all of the points in the feature matrix.
```

```
    """
    # Your code here
    n = len(feature_matrix)
    loss = 0.0
    for i in range(n):
        loss += max(0, (1-labels[i] * (np.dot(feature_matrix[i], theta) +␣
 ↪theta_0)))
    return loss/n
#pragma: coderesponse end
```

[ ]:
```
feature_matrix = np.array([[1, 2], [1, 2]])
label, theta, theta_0 = np.array([1, 1]), np.array([-1, 1]), -0.2
hinge_loss_full(feature_matrix, label, theta, theta_0)
```

0.1.3 3. Perceptron Algorithm

Now you will implement the Perceptron algorithm

1) Perceptron Single Step Update Now you will implement the single step update for the perceptron algorithm (implemented with 01 loss). You will be given the feature vector as an array of numbers, the current and 0 parameters, and the correct label of the feature vector. The function should return a tuple in which the first element is the correctly updated value of and the second element is the correctly updated value of 0.

Available Functions: You have access to the NumPy python library as np.

Tip: Because of numerical instabilities, it is preferable to identify 0 with a small range [,]. That is, when x is a float, "x=0" should be checked with |x|<.

[ ]:
```
#pragma: coderesponse template
def perceptron_single_step_update(
        feature_vector,
        label,
        current_theta,
        current_theta_0):
    """
    Properly updates the classification parameter, theta and theta_0, on a
    single step of the perceptron algorithm.

    Args:
        feature_vector - A numpy array describing a single data point.
        label - The correct classification of the feature vector.
        current_theta - The current theta being used by the perceptron
            algorithm before this update.
        current_theta_0 - The current theta_0 being used by the perceptron
            algorithm before this update.

    Returns: A tuple where the first element is a numpy array with the value of
    theta after the current update has completed and the second element is a
    real valued number with the value of theta_0 after the current updated has
```

```
    completed.
    """
    # Your code here
    theta = np.copy(current_theta)
    theta_0 = current_theta_0
    if (label * (np.dot(feature_vector, theta) + theta_0) <= 0 ):
        theta = theta + label * feature_vector
        theta_0 = theta_0 + label
    return (theta, theta_0)
#pragma: coderesponse end
```

[ ]:
```
feature_vector = np.array([1, 2])
label, theta, theta_0 = 1, np.array([-1, 1]), -1.5
perceptron_single_step_update(feature_vector, label, theta, theta_0)

feature_vector = np.array([1, 2])
label, theta, theta_0 = 1, np.array([-1, 1]), -1
perceptron_single_step_update(feature_vector, label, theta, theta_0)

#    feature_vector = np.array([1, 2])
#    label, theta, theta_0 = 1, np.array([-1, 1]), -1
#    exp_res = (np.array([0, 3]), 0)
#    if check_tuple(
#            ex_name + " (boundary case)", p1.perceptron_single_step_update,
#            exp_res, feature_vector, label, theta, theta_0):
#        return
```

2) Full Perceptron Algorithm In this step you will implement the full perceptron algorithm. You will be given the same feature matrix and labels array as you were given in The Complete Hinge Loss. You will also be given T, the maximum number of times that you should iterate through the feature matrix before terminating the algorithm. Initialize and 0 to zero. This function should return a tuple in which the first element is the final value of and the second element is the value of 0.

Tip: Call the function perceptron_single_step_update directly without coding it again.

Hint: Make sure you initialize theta to a 1D array of shape (n,), and not a 2D array of shape (1, n). Note: Please call get_order(feature_matrix.shape[0]), and use the ordering to iterate the feature matrix in each iteration. The ordering is specified due to grading purpose. In practice, people typically just randomly shuffle indices to do stochastic optimization.

Available Functions: You have access to the NumPy python library as np and perceptron_single_step_update which you have already implemented.

[ ]:
```
#pragma: coderesponse template
def perceptron(feature_matrix, labels, T):
    """
    Runs the full perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.
```

```
    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    NOTE: Iterate the data matrix by the orders returned by␣
 ↪get_order(feature_matrix.shape[0])

    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        T - An integer indicating how many times the perceptron algorithm
            should iterate through the feature matrix.

    Returns: A tuple where the first element is a numpy array with the value of
    theta, the linear classification parameter, after T iterations through the
    feature matrix and the second element is a real number with the value of
    theta_0, the offset classification parameter, after T iterations through
    the feature matrix.
    """
    current_theta = np.array([0.0 for x in range(feature_matrix.shape[1])])
    current_theta_0 = 0.0
    for t in range(int(T)):
        for i in get_order(feature_matrix.shape[0]):
            feature_vector = feature_matrix[i]
            label = labels[i]
            current_theta, current_theta_0 =␣
 ↪perceptron_single_step_update(feature_vector, label, current_theta,␣
 ↪current_theta_0)
    return (current_theta, current_theta_0)
#pragma: coderesponse end
```

[ ]:
```
feature_matrix = np.array([[1, 2], [-1, 0]])
print(feature_matrix.shape[0])
labels = np.array([1, 1])
T = 1
perceptron(feature_matrix, labels, T)
```

3) Average Perceptron Algorithm The average perceptron will add a modification to the original perceptron algorithm: since the basic algorithm continues updating as the algorithm runs, nudging parameters in possibly conflicting directions, it is better to take an average of those parameters as the final answer. Every update of the algorithm is the same as before. The returned parameters , however, are an average of the s across the nTsteps:

$$_{final} = \frac{1}{nT}\left(^{(1)} + ^{(2)} + \cdots + ^{(nT)}\right)$$

You will now implement the average perceptron algorithm. This function should be constructed similarly to the Full Perceptron Algorithm above, except that it should return the average values of and 0

Tip: Tracking a moving average through loops is difficult, but tracking a sum through loops is simple.

Note: Please call get_order(feature_matrix.shape[0]), and use the ordering to iterate the feature matrix in each iteration. The ordering is specified due to grading purpose. In practice, people typically just randomly shuffle indices to do stochastic optimization.

Available Functions: You have access to the NumPy python library as np and perceptron_single_step_update which you have already implemented.

[ ]:
```
#pragma: coderesponse template
def average_perceptron(feature_matrix, labels, T):
    """
    Runs the average perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    NOTE: Iterate the data matrix by the orders returned by␣
 ↪get_order(feature_matrix.shape[0])


    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        T - An integer indicating how many times the perceptron algorithm
            should iterate through the feature matrix.

    Returns: A tuple where the first element is a numpy array with the value of
    the average theta, the linear classification parameter, found after T
    iterations through the feature matrix and the second element is a real
    number with the value of the average theta_0, the offset classification
    parameter, found after T iterations through the feature matrix.

    Hint: It is difficult to keep a running average; however, it is simple to
    find a sum and divide.
    """
    # Your code here
    current_theta = np.array([0.0 for x in range(feature_matrix.shape[1])])
    current_theta_0 = 0.0
    final_theta = np.copy(current_theta)
    final_theta_0 = 0.0
    for t in range(int(T)):
```

```
        for i in get_order(feature_matrix.shape[0]):
            feature_vector = feature_matrix[i]
            label = labels[i]
            current_theta, current_theta_0 =␣
 ↪perceptron_single_step_update(feature_vector, label, current_theta,␣
 ↪current_theta_0)
            final_theta = final_theta + current_theta
            final_theta_0 = final_theta_0 + current_theta_0
    final_theta = final_theta/(T*feature_matrix.shape[0])
    final_theta_0 = final_theta_0/(T*feature_matrix.shape[0])
    return (final_theta, final_theta_0)
#pragma: coderesponse end
```

[ ]:
```
feature_matrix=np.array([[-0.27890596,-0.35745481,-0.15375912,0.06750791,-0.
 ↪13934302]
,[0.33195994,0.13575078,0.26774662,0.15394081,-0.47254553]
,[0.12422833,-0.32850287,-0.21272907,0.33608168,0.0151038,]
,[0.23300123,-0.35348998,-0.1609891,-0.33766974,-0.4585012,]
,[-0.12326777,-0.12161117,-0.19923582,0.34578459,-0.19464429]
,[-0.35898201,0.25530968,0.10305159,0.07165837,-0.14451136]
,[-0.36828614,0.34445647,-0.4143974,-0.18113471,-0.29878238]
,[-0.0515599,-0.23311173,0.02380431,-0.23835969,0.20633263]
,[-0.46007216,-0.33609266,0.05073226,0.32756282,-0.17777972]
,[0.14394631,0.26285361,0.14750047,0.47159866,0.28887808]])
labels=np.array([-1,1,-1,1,1,1,-1,1,1,1])
T=5
# average_perceptron output is ['-0.2969134', '-0.0117562', '1.5908964', '0.
 ↪6651561', '0.5130588']
average_perceptron(feature_matrix, labels, T)
```

0.1.4 4. Pegasos Algorithm

Now you will implement the Pegasos algorithm. For more information, refer to the original paper at original paper.

The following pseudo-code describes the Pegasos update rule.

Pegasos update rule $\left(x^{(i)}, y^{(i)}, , ,\right)$: if $y^{(i)}(x^{(i)})1$ then update $= (1) + y^{(i)}x^{(i)}$ else: update $= (1)$

The parameter is a decaying factor that will decrease over time. The parameter is a regularizing parameter.

In this problem, you will need to adapt this update rule to add a bias term (0) to the hypothesis, but take care not to penalize the magnitude of 0.

1). Pegasos Single Step Update Next you will implement the single step update for the Pegasos algorithm. This function is very similar to the function that you implemented in Perceptron Single Step Update, except that it should utilize the Pegasos parameter update rules instead of those for perceptron. The function will also be passed a and value to use for updates.

Available Functions: You have access to the NumPy python library as np.

[ ]:
```
#pragma: coderesponse template
def pegasos_single_step_update(
        feature_vector,
        label,
        L,
        eta,
        current_theta,
        current_theta_0):
    """
    Properly updates the classification parameter, theta and theta_0, on a
    single step of the Pegasos algorithm

    Args:
        feature_vector - A numpy array describing a single data point.
        label - The correct classification of the feature vector.
        L - The lamba value being used to update the parameters.
        eta - Learning rate to update parameters.
        current_theta - The current theta being used by the Pegasos
            algorithm before this update.
        current_theta_0 - The current theta_0 being used by the
            Pegasos algorithm before this update.

    Returns: A tuple where the first element is a numpy array with the value of
    theta after the current update has completed and the second element is a
    real valued number with the value of theta_0 after the current updated has
    completed.
    """
    # Your code here
    if (label * (np.dot(feature_vector, current_theta) + current_theta_0) <= 1␣
↪):
        return ((1-eta*L)*current_theta + eta*label*feature_vector,␣
↪current_theta_0 + eta*label)
    else:
        return ((1-eta*L)*current_theta, current_theta_0)
#pragma: coderesponse end
```

2). Full Pegasos Algorithm Finally you will implement the full Pegasos algorithm. You will be given the same feature matrix and labels array as you were given in Full Perceptron Algorithm. You will also be given T, the maximum number of times that you should iterate through the feature matrix before terminating the algorithm. Initialize and 0 to zero. For each update, set $= \frac{1}{\sqrt{t}}$ where t is a counter for the number of updates performed so far (between 1 and nT inclusive). This function should return a tuple in which the first element is the final value of and the second element is the value of 0.

Note: Please call get_order(feature_matrix.shape[0]), and use the ordering to iterate the feature matrix in each iteration. The ordering is specified due to grading purpose. In practice, people typically just randomly shuffle indices to do stochastic optimization.

Available Functions: You have access to the NumPy python library as np and pegasos_single_step_update which you have already implemented.

[ ]:
```
#pragma: coderesponse template
def pegasos(feature_matrix, labels, T, L):
    """
    Runs the Pegasos algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    For each update, set learning rate = 1/sqrt(t),
    where t is a counter for the number of updates performed so far (between 1
    and nT inclusive).

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        T - An integer indicating how many times the algorithm
            should iterate through the feature matrix.
        L - The lamba value being used to update the Pegasos
            algorithm parameters.

    Returns: A tuple where the first element is a numpy array with the value of
    the theta, the linear classification parameter, found after T
    iterations through the feature matrix and the second element is a real
    number with the value of the theta_0, the offset classification
    parameter, found after T iterations through the feature matrix.
    """
    # Your code here
    theta = np.array([0.0 for x in range(feature_matrix.shape[1])])
    theta_0 = 0.0
    n = 0
    for t in range(int(T)):
        for i in get_order(feature_matrix.shape[0]):
            n += 1
            feature_vector = feature_matrix[i]
            label = labels[i]
            eta = 1/n**(1/2)
            theta, theta_0 = pegasos_single_step_update(feature_vector, label,␣
 ↪L, eta, theta, theta_0)
    return (theta, theta_0)
#pragma: coderesponse end
```

0.1.5 5. Algorithm Discussion

Once you have completed the implementation of the 3 learning algorithms, you should qualitatively verify your implementations. In main.py we have included a block of code that you should uncomment. This code loads a 2D dataset from toy_data.txt, and trains your models using T=10, =0.2. main.py will compute and 0 for each of the learning algorithms that you have written. Then, it will call plot_toy_data to plot the resulting model and boundary.

1) Plots In order to verify your plots, please enter the values of and 0 for all three algorithms.

(For example, if =(1,0.5), then type 1, 0.5 without the brackets. Make sure your answers are correct up to 4 decimal places.)

For the perceptron algorithm: = [3.9174, 4.164], 0 =-8.0

For the average perceptron algorithm: = [3.4783, 3.6111], 0 =-6.373

For the Pegasos algorithm: = [0.7346, 0.63], 0 =-1.2195

2) Convergence Since you have implemented three different learning algorithm for linear classifier, it is interesting to investigate which algorithm would actually converge. Please run it with a larger number of iterations T to see whether the algorithm would visually converge. You may also check whether the parameter in your theta converge in the first decimal place. Achieving convergence in longer decimal requires longer iterations, but the conclusion should be the same.

Which of the following algorithm will converge on this dataset? (Choose all that apply.)

$\times$ perceptron algorithm $\vee$ average perceptron algorithm $\vee$ Pegasos algorithm

Part II

0.1.6 6. Automative review analyzer

Now that you have verified the correctness of your implementations, you are ready to tackle the main task of this project: building a classifier that labels reviews as positive or negative using text-based features and the linear classifiers that you implemented in the previous section!

The Data The data consists of several reviews, each of which has been labeled with 1 or +1, corresponding to a negative or positive review, respectively. The original data has been split into four files:

reviews_train.tsv (4000 examples)

reviews_validation.tsv (500 examples)

reviews_test.tsv (500 examples)

To get a feel for how the data looks, we suggest first opening the files with a text editor, spreadsheet program, or other scientific software package (like pandas).

Translating reviews to feature vectors We will convert review texts into feature vectors using a bag of words approach. We start by compiling all the words that appear in a training set of reviews into a dictionary , thereby producing a list of d unique words.

We can then transform each of the reviews into a feature vector of length d by setting the i^{th} coordinate of the feature vector to 1 if the i^{th} word in the dictionary appears in the review, or 0 otherwise. For instance, consider two simple documents “Mary loves apples” and “Red apples”.

In this case, the dictionary is the set {Mary;loves;apples;red} , and the documents are represented as (1;1;1;0) and (0;0;1;1).

A bag of words model can be easily expanded to include phrases of length m. A unigram model is the case for which m=1. In the example, the unigram dictionary would be (Mary;loves;apples;red). In the bigram case, m=2, the dictionary is (Mary loves;loves apples;Red apples), and representations for each sample are (1;1;0),(0;0;1). In this section, you will only use the unigram word features. These functions are already implemented for you in the bag of words function.

In utils.py, we have supplied you with the load data function, which can be used to read the .tsv files and returns the labels and texts. We have also supplied you with the bag_of_words function in project1.py, which takes the raw data and returns dictionary of unigram words. The resulting dictionary is an input to extract_bow_feature_vectors which computes a feature matrix of ones and zeros that can be used as the input for the classification algorithms. Using the feature matrix and your implementation of learning algorithms from before, you will be able to compute and 0.

[ ]:
```
# Part II

def accuracy(preds, targets):
    """
    Given length-N vectors containing predicted and target labels,
    returns the percentage and number of correct predictions.
    """
    return (preds == targets).mean()
```

[ ]:
```
# Question 9 1)
#pragma: coderesponse template
def bag_of_words(texts):
    """
    Inputs a list of string reviews
    Returns a dictionary of unique unigrams occurring over the input

    Feel free to change this code as guided by Problem 9
    """
    # Your code here
    stop_words = pd.read_csv("C:/root/workspace/MITx_6.86x/07_Projects/project1/
 ↪stopwords.txt", header=None, names='w')
    stop_words = stop_words["w"].tolist()

    dictionary = {} # maps word to unique index
    for text in texts:
        word_list = extract_words(text)
        for word in word_list:
            if word not in stop_words:
                if word not in dictionary:
                    dictionary[word] = len(dictionary)
    return dictionary
#pragma: coderesponse end
```

[ ]:
```
texts = [
        "He loves to walk on the beach",
        "There is nothing better"]
bag_of_words(texts)
```

[ ]:
```
dictionary2 = bag_of_words(train_texts)

train_bow_features2 = extract_bow_feature_vectors_original(train_texts,␣
 ↪dictionary2)
val_bow_features2 = extract_bow_feature_vectors_original(val_texts, dictionary2)
test_bow_features2 = extract_bow_feature_vectors_original(test_texts,␣
 ↪dictionary2)
```

7. Classification and Accuracy Now we need a way to actually use our model to classify the data points. In this section, you will implement a way to classify the data points using your model parameters, and then measure the accuracy of your model. #### 1) Classification Implement a classification function that uses and 0 to classify a set of data points. You are given the feature matrix, , and 0 as defined in previous sections. This function should return a numpy array of -1s and 1s. If a prediction is greater than zero, it should be considered a positive classification.

Available Functions: You have access to the NumPy python library as np. Tip: As in previous exercises, when x is a float, "x=0" should be checked with |x|<.

[ ]:
```
#pragma: coderesponse template
def classify(feature_matrix, theta, theta_0):
    """
    A classification function that uses theta and theta_0 to classify a set of
    data points.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
                theta - A numpy array describing the linear classifier.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.

    Returns: A numpy array of 1s and -1s where the kth element of the array is
    the predicted classification of the kth row of the feature matrix using the
    given theta and theta_0. If a prediction is GREATER THAN zero, it should
    be considered a positive classification.
    """
    # Your code here
    n = feature_matrix.shape[0]
    ys = np.empty(shape=[0, 0])
    for i in range(n):
        if (np.dot(feature_matrix[i], theta) + theta_0 > 0):
            ys = np.append(ys, 1.0)
        else:
            ys = np.append(ys, -1.0)
```

```
    return ys
#pragma: coderesponse end
```

[ ]:
```
feature_matrix = np.array([[1, 1], [1, 1], [1, 1]])
theta = np.array([1, 1])
theta_0 = 0
classify(feature_matrix, theta, theta_0)
```

2) Accuracy We have supplied you with an accuracy function. The accuracy function takes a numpy array of predicted labels and a numpy array of actual labels and returns the prediction accuracy. You should use this function along with the functions that you have implemented thus far in order to implement classifier_accuracy.

The classifier_accuracy function should take 6 arguments:
a classifier function that, itself, takes arguments (feature_matrix, labels, **kwargs)
the training feature matrix
the validation feature matrix
the training labels
the valiation labels
a **kwargs argument to be passed to the classifier function

This function should train the given classifier using the training data and then compute compute the classification accuracy on both the train and validation data. The return values should be a tuple where the first value is the training accuracy and the second value is the validation accuracy.

Implement classifier accuracy in the coding box below:

Available Functions: You have access to the NumPy python library as np, to classify which you have already implemented and to accuracy which we defined above.

[ ]:
```
#pragma: coderesponse template
def classifier_accuracy(
        classifier,
        train_feature_matrix,
        val_feature_matrix,
        train_labels,
        val_labels,
        **kwargs):
    """
    Trains a linear classifier using the perceptron algorithm with a given T
    value. The classifier is trained on the train data. The classifier's
    accuracy on the train and validation data is then returned.

    Args:
        classifier - A classifier function that takes arguments
            (feature matrix, labels, **kwargs)
        train_feature_matrix - A numpy matrix describing the training
            data. Each row represents a single data point.
        val_feature_matrix - A numpy matrix describing the training
            data. Each row represents a single data point.
```

```
        train_labels - A numpy array where the kth element of the array
            is the correct classification of the kth row of the training
            feature matrix.
        val_labels - A numpy array where the kth element of the array
            is the correct classification of the kth row of the validation
            feature matrix.
        **kwargs - Additional named arguments to pass to the classifier
            (e.g. T or L)

    Returns: A tuple in which the first element is the (scalar) accuracy of the
    trained classifier on the training data and the second element is the
    accuracy of the trained classifier on the validation data.
    """
    # Your code here
    theta_star, theta_star_0 = classifier(train_feature_matrix, train_labels,␣
↪**kwargs)
    pred_train_labels = classify(train_feature_matrix, theta_star, theta_star_0)
    pred_val_labels = classify(val_feature_matrix, theta_star, theta_star_0)
    return (accuracy(pred_train_labels, train_labels),␣
↪accuracy(pred_val_labels, val_labels))

#pragma: coderesponse end
```

[ ]:
```
t_feature_matrix = np.array([[1, 0], [1, -1], [2, 3]])
v_feature_matrix = np.array([[1, 1], [2, -1]])
t_labels = np.array([1, -1, 1])
v_labels = np.array([-1, 1])
kwargs = {"T":1}
classifier_accuracy(perceptron, t_feature_matrix, v_feature_matrix, t_labels,␣
↪v_labels, **kwargs)
```

3) Baseline Accuracy Now, uncomment the relevant lines in main.py and report the training and validation accuracies of each algorithm with T = 10 and = 0.01 (the value only applies to Pegasos).

Validation accuracy of your Perceptron algorithm: 0.7160
Validation accuracy of your Average Perceptron algorithm: 0.7980
Validation accuracy of your Pegasos algorithm: 0.7900

0.1.7 8. Parameter Tuning

You finally have your algorithms up and running, and a way to measure performance! But, it's still unclear what values the hyperparameters like T and should have. In this section, you'll tune these hyperparameters to maximize the performance of each model.

One way to tune your hyperparameters for any given Machine Learning algorithm is to perform a grid search over all the possible combinations of values. If your hyperparameters can be any real number, you will need to limit the search to some finite set of possible values for each hyperparameter. For efficiency reasons, often you might want to tune one individual parameter, keeping all others constant, and then move onto the next one; Compared to a full grid search there

are many fewer possible combinations to check, and this is what you'll be doing for the questions below.

In main.py uncomment Problem 8 to run the staff-provided tuning algorithm from utils.py. For the purposes of this assignment, please try the following values for T: [1, 5, 10, 15, 25, 50] and the following values for [0.001, 0.01, 0.1, 1, 10]. For pegasos algorithm, first fix =0.01 to tune T, and then use the best T to tune

1) Performance After Tuning After tuning, please enter the best T value for each of the perceptron and average percepton algorithms, and both the best T and for the Pegasos algorithm.

Note: Just enter the values printed in your main.py. Note that for the Pegasos algorithm, the result does not reflect the best combination of T and .

For the perceptron algorithm: T = 25.0000 With validation accuracy = 0.7940

For the average perceptron algorithm: T = 25.0000 With validation accuracy = 0.8000

For the pegasos algorithm: T = 25.0000, = 0.0100 With validation accuracy = 0.8060

2) Accuracy on the test set After you have chosen your best method (perceptron, average perceptron or Pegasos) and parameters, use this classifier to compute testing accuracy on the test set.

We have supplied the feature matrix and labels in main.py as test_bow_features and test_labels.

Note: In practice the validation set is used for tuning hyperparameters while a heldout test set is the final benchmark used to compare disparate models that have already been tuned. You may notice that your results using a validation set don't always align with those of the test set, and this is to be expected.

Accuracy on the test set : 0.802

3) The most explanatory unigrams According to the largest weights (i.e. individual i values in your vector), you can find out which unigrams were the most impactful ones in predicting positive labels. Uncomment the relevant part in main.py to call utils.most_explanatory_word.

Report the top ten most explanatory word features for positive classification below:

delicious, great, !, best, perfect, loves, wonderful, glad, love, quickly

[ ]:
```
# Question 8
data = (train_bow_features, test_bow_features, train_labels, test_labels)
kwargs = {"T":25.0000, "L":0.0100}

theta_star, theta_star_0 = pegasos(train_bow_features, train_labels, **kwargs)
pred_train_labels = classify(train_bow_features, theta_star, theta_star_0)
pred_test_labels = classify(test_bow_features, theta_star, theta_star_0)
print(accuracy(pred_test_labels, test_labels))
print(theta_star, theta_star_0)

best_theta = theta_star # Your code here
wordlist   = [word for (idx, word) in sorted(zip(dictionary.values(),␣
 ↪dictionary.keys()))]
sorted_word_features = most_explanatory_word(best_theta, wordlist)
print("Most Explanatory Word Features")
```

```
print(sorted_word_features[:10])
```

0.1.8 9. Feature Engineering

Frequently, the way the data is represented can have a significant impact on the performance of a machine learning method. Try to improve the performance of your best classifier by using different features. In this problem, we will practice two simple variants of the bag of words (BoW) representation.

1) Remove Stop Words Try to implement stop words removal in your feature engineering code. Specifically, load the file stopwords.txt, remove the words in the file from your dictionary, and use features constructed from the new dictionary to train your model and make predictions.

Compare your result in the testing data on Pegasos algorithm using T=25 and L=0.01 when you remove the words in stopwords.txt from your dictionary.

Hint: Instead of replacing the feature matrix with zero columns on stop words, you can modify the bag_of_words function to prevent adding stopwords to the dictionary

Accuracy on the test set using the original dictionary: 0.8020

Accuracy on the test set using the dictionary with stop words removed: 0.808

[ ]:
```
# Question 9 1)
data = (train_bow_features2, test_bow_features2, train_labels, test_labels)
kwargs = {"T":25.0000, "L":0.0100}

theta_star, theta_star_0 = pegasos(train_bow_features2, train_labels, **kwargs)
pred_train_labels = classify(train_bow_features2, theta_star, theta_star_0)
pred_test_labels = classify(test_bow_features2, theta_star, theta_star_0)
print(accuracy(pred_test_labels, test_labels))
print(theta_star, theta_star_0)

best_theta = theta_star # Your code here
wordlist   = [word for (idx, word) in sorted(zip(dictionary2.values(),␣
 ↪dictionary2.keys()))]
sorted_word_features = most_explanatory_word(best_theta, wordlist)
print("Most Explanatory Word Features")
print(sorted_word_features[:10])
```

[ ]:
```
texts = [
    "He loves her ",
    "He really really loves her"]
keys = ["he", "loves", "her", "really"]
dictionary = {k:i for i, k in enumerate(keys)}
extract_bow_feature_vectors_original(texts, dictionary)
```

2) Change Binary Features to Counts Features Again, use the same learning algorithm and the same feature as the last problem. However, when you compute the feature vector of a word, use its count in each document rather than a binary indicator.

Hint: You are free to modify the extract_bow_feature_vectors function to compute counts features.

Accuracy on the test set using the dictionary with stop words removed and counts features: 0.77

Try to compare your result to the last problem, and see the discussion in solution after answering the question.

[ ]:
```
# Question 9 2)
#pragma: coderesponse template
def extract_bow_feature_vectors(reviews, dictionary):
    """
    Inputs a list of string reviews
    Inputs the dictionary of words as given by bag_of_words
    Returns the bag-of-words feature matrix representation of the data.
    The returned matrix is of shape (n, m), where n is the number of reviews
    and m the total number of entries in the dictionary.

    Feel free to change this code as guided by Problem 9
    """
    # Your code here

    num_reviews = len(reviews)
    feature_matrix = np.zeros([num_reviews, len(dictionary)])

    for i, text in enumerate(reviews):
        word_list = extract_words(text)
        for word in word_list:
            if word in dictionary:
                feature_matrix[i, dictionary[word]] += 1  # counting words␣
 ↪frequency also
    return feature_matrix
#pragma: coderesponse end
```

[ ]:
```
dictionary2 = bag_of_words(train_texts)

train_bow_features3 = extract_bow_feature_vectors(train_texts, dictionary2)
val_bow_features3 = extract_bow_feature_vectors(val_texts, dictionary2)
test_bow_features3 = extract_bow_feature_vectors(test_texts, dictionary2)
```

[ ]:
```
texts = [
    "He loves her ",
    "He really really loves her"]
keys = ["he", "loves", "her", "really"]
dictionary = {k:i for i, k in enumerate(keys)}
extract_bow_feature_vectors(texts, dictionary)
```

Some additional features that you might want to explore are:
Length of the text
Occurrence of all-cap words (e.g. “AMAZING”, “DON’T BUY THIS”)
Word embeddings

Besides adding new features, you can also change the original unigram feature set. For example,

- Threshold the number of times a word should appear in the dataset before adding them to the dictionary. For example, words that occur less than three times across the train dataset could be considered irrelevant and thus can be removed. This lets you reduce the number of columns that are prone to overfitting.

There are also many other things you could change when training your model. Try anything that can help you understand the sentiment of a review. It's worth looking through the dataset and coming up with some features that may help your model. Remember that not all features will actually help so you should experiment with some simpler ones before trying anything too complicated.

0.1.9 10. Attchment

2) project1.py

[ ]:
```
from string import punctuation, digits
import numpy as np
import random
import pandas as pd

# Part I

#pragma: coderesponse template
def get_order(n_samples):
    try:
        with open(str(n_samples) + '.txt') as fp:
            line = fp.readline()
            return list(map(int, line.split(',')))
    except FileNotFoundError:
        random.seed(1)
        indices = list(range(n_samples))
        random.shuffle(indices)
        return indices
#pragma: coderesponse end


#pragma: coderesponse template
def hinge_loss_single(feature_vector, label, theta, theta_0):
    """
    Finds the hinge loss on a single data point given specific classification
    parameters.

    Args:
        feature_vector - A numpy array describing the given data point.
        label - A real valued number, the correct classification of the data
            point.
```

```
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.


    Returns: A real number representing the hinge loss associated with the
    given data point and parameters.
    """
    # Your code here
    loss = max(0, 1 - label * (np.dot(feature_vector, theta) + theta_0))
    return loss
#pragma: coderesponse end


#pragma: coderesponse template
def hinge_loss_full(feature_matrix, labels, theta, theta_0):
    """
    Finds the total hinge loss on a set of data given specific classification
    parameters.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.


    Returns: A real number representing the hinge loss associated with the
    given dataset and parameters. This number should be the average hinge
    loss across all of the points in the feature matrix.
    """
    # Your code here
    n = len(feature_matrix)
    loss = 0.0
    for i in range(n):
        loss += max(0, (1-labels[i] * (np.dot(feature_matrix[i], theta) +␣
 ↪theta_0)))
    return loss/n
#pragma: coderesponse end


#pragma: coderesponse template
def perceptron_single_step_update(
        feature_vector,
        label,
        current_theta,
```

```
        current_theta_0):
    """
    Properly updates the classification parameter, theta and theta_0, on a
    single step of the perceptron algorithm.

    Args:
        feature_vector - A numpy array describing a single data point.
        label - The correct classification of the feature vector.
        current_theta - The current theta being used by the perceptron
            algorithm before this update.
        current_theta_0 - The current theta_0 being used by the perceptron
            algorithm before this update.

    Returns: A tuple where the first element is a numpy array with the value of
    theta after the current update has completed and the second element is a
    real valued number with the value of theta_0 after the current updated has
    completed.
    """
    # Your code here
    theta = np.copy(current_theta)
    theta_0 = current_theta_0
    if (label * (np.dot(feature_vector, theta) + theta_0) <= 0 ):
        theta = theta + label * feature_vector
        theta_0 = theta_0 + label
    return (theta, theta_0)
#pragma: coderesponse end


#pragma: coderesponse template
def perceptron(feature_matrix, labels, T):
    """
    Runs the full perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    NOTE: Iterate the data matrix by the orders returned by␣
 ↪get_order(feature_matrix.shape[0])

    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        T - An integer indicating how many times the perceptron algorithm
```

```
            should iterate through the feature matrix.

    Returns: A tuple where the first element is a numpy array with the value of
    theta, the linear classification parameter, after T iterations through the
    feature matrix and the second element is a real number with the value of
    theta_0, the offset classification parameter, after T iterations through
    the feature matrix.
    """
    # Your code here
    current_theta = np.array([0.0 for x in range(feature_matrix.shape[1])])
    current_theta_0 = 0.0
    for t in range(int(T)):
        for i in get_order(feature_matrix.shape[0]):
            feature_vector = feature_matrix[i]
            label = labels[i]
            current_theta, current_theta_0 =␣
 ↪perceptron_single_step_update(feature_vector, label, current_theta,␣
 ↪current_theta_0)
    return (current_theta, current_theta_0)
#pragma: coderesponse end


#pragma: coderesponse template
def average_perceptron(feature_matrix, labels, T):
    """
    Runs the average perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    NOTE: Iterate the data matrix by the orders returned by␣
 ↪get_order(feature_matrix.shape[0])


    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        T - An integer indicating how many times the perceptron algorithm
            should iterate through the feature matrix.

    Returns: A tuple where the first element is a numpy array with the value of
    the average theta, the linear classification parameter, found after T
    iterations through the feature matrix and the second element is a real
```

```
    number with the value of the average theta_0, the offset classification
    parameter, found after T iterations through the feature matrix.

    Hint: It is difficult to keep a running average; however, it is simple to
    find a sum and divide.
    """
    current_theta = np.array([0 for x in range(feature_matrix.shape[1])])
    current_theta_0 = 0.0
    final_theta = np.copy(current_theta)
    final_theta_0 = 0.0
    for t in range(int(T)):
        for i in get_order(feature_matrix.shape[0]):
            feature_vector = feature_matrix[i]
            label = labels[i]
            current_theta, current_theta_0 =␣
 ↪perceptron_single_step_update(feature_vector, label, current_theta,␣
 ↪current_theta_0)
            final_theta = final_theta + current_theta
            final_theta_0 = final_theta_0 + current_theta_0
    final_theta = final_theta/(T*feature_matrix.shape[0])
    final_theta_0 = final_theta_0/(T*feature_matrix.shape[0])
    return (final_theta, final_theta_0)
#pragma: coderesponse end


#pragma: coderesponse template
def pegasos_single_step_update(
        feature_vector,
        label,
        L,
        eta,
        current_theta,
        current_theta_0):
    """
    Properly updates the classification parameter, theta and theta_0, on a
    single step of the Pegasos algorithm

    Args:
        feature_vector - A numpy array describing a single data point.
        label - The correct classification of the feature vector.
        L - The lamba value being used to update the parameters.
        eta - Learning rate to update parameters.
        current_theta - The current theta being used by the Pegasos
            algorithm before this update.
        current_theta_0 - The current theta_0 being used by the
            Pegasos algorithm before this update.
```

```
    Returns: A tuple where the first element is a numpy array with the value of
    theta after the current update has completed and the second element is a
    real valued number with the value of theta_0 after the current updated has
    completed.
    """
    if (label * (np.dot(feature_vector, current_theta) + current_theta_0) <= 1␣
↪):
        return ((1-eta*L)*current_theta + eta*label*feature_vector,␣
↪current_theta_0 + eta*label)
    else:
        return ((1-eta*L)*current_theta, current_theta_0)
#pragma: coderesponse end


#pragma: coderesponse template
def pegasos(feature_matrix, labels, T, L):
    """
    Runs the Pegasos algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    For each update, set learning rate = 1/sqrt(t),
    where t is a counter for the number of updates performed so far (between 1
    and nT inclusive).

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        T - An integer indicating how many times the algorithm
            should iterate through the feature matrix.
        L - The lamba value being used to update the Pegasos
            algorithm parameters.

    Returns: A tuple where the first element is a numpy array with the value of
    the theta, the linear classification parameter, found after T
    iterations through the feature matrix and the second element is a real
    number with the value of the theta_0, the offset classification
    parameter, found after T iterations through the feature matrix.
    """
    # Your code here
    theta = np.array([0.0 for x in range(feature_matrix.shape[1])])
    theta_0 = 0.0
```

```
    n = 0
    for t in range(int(T)):
        for i in get_order(feature_matrix.shape[0]):
            n += 1
            feature_vector = feature_matrix[i]
            label = labels[i]
            eta = 1/n**(1/2)
            theta, theta_0 = pegasos_single_step_update(feature_vector, label,␣
 ↪L, eta, theta, theta_0)
    return (theta, theta_0)
#pragma: coderesponse end


# Part II

#pragma: coderesponse template
def classify(feature_matrix, theta, theta_0):
    """
    A classification function that uses theta and theta_0 to classify a set of
    data points.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
                theta - A numpy array describing the linear classifier.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.

    Returns: A numpy array of 1s and -1s where the kth element of the array is
    the predicted classification of the kth row of the feature matrix using the
    given theta and theta_0. If a prediction is GREATER THAN zero, it should
    be considered a positive classification.
    """
    # Your code here
    # import pdb; pdb.set_trace()
    n = feature_matrix.shape[0]
    ys = np.empty(shape=[0, 0], dtype=int)
    for i in range(n):
        if (np.dot(feature_matrix[i], theta) + theta_0 > 0):
            ys = np.append(ys, 1)
        else:
            ys = np.append(ys, -1)
    return ys
#pragma: coderesponse end


#pragma: coderesponse template
```

```
def classifier_accuracy(
        classifier,
        train_feature_matrix,
        val_feature_matrix,
        train_labels,
        val_labels,
        **kwargs):
    """
    Trains a linear classifier using the perceptron algorithm with a given T
    value. The classifier is trained on the train data. The classifier's
    accuracy on the train and validation data is then returned.

    Args:
        classifier - A classifier function that takes arguments
            (feature matrix, labels, **kwargs)
        train_feature_matrix - A numpy matrix describing the training
            data. Each row represents a single data point.
        val_feature_matrix - A numpy matrix describing the training
            data. Each row represents a single data point.
        train_labels - A numpy array where the kth element of the array
            is the correct classification of the kth row of the training
            feature matrix.
        val_labels - A numpy array where the kth element of the array
            is the correct classification of the kth row of the validation
            feature matrix.
        **kwargs - Additional named arguments to pass to the classifier
            (e.g. T or L)

    Returns: A tuple in which the first element is the (scalar) accuracy of the
    trained classifier on the training data and the second element is the
    accuracy of the trained classifier on the validation data.
    """
    # Your code here
    theta_star, theta_star_0 = classifier(train_feature_matrix, train_labels,␣
 ↪**kwargs)
    pred_train_labels = classify(train_feature_matrix, theta_star, theta_star_0)
    pred_val_labels = classify(val_feature_matrix, theta_star, theta_star_0)
    return (accuracy(pred_train_labels, train_labels),␣
 ↪accuracy(pred_val_labels, val_labels))
#pragma: coderesponse end


#pragma: coderesponse template
def extract_words(input_string):
    """
    Helper function for bag_of_words()
    Inputs a text string
```

```
    Returns a list of lowercase words in the string.
    Punctuation and digits are separated out into their own words.
    """
    for c in punctuation + digits:
        input_string = input_string.replace(c, ' ' + c + ' ')

    return input_string.lower().split()
#pragma: coderesponse end


#pragma: coderesponse template
def bag_of_words_original(texts):
    """
    Inputs a list of string reviews
    Returns a dictionary of unique unigrams occurring over the input

    Feel free to change this code as guided by Problem 9
    """
    # Your code here
    # see bag_of_words

    dictionary = {} # maps word to unique index
    for text in texts:
        word_list = extract_words(text)
        for word in word_list:
                if word not in dictionary:
                        dictionary[word] = len(dictionary)
    return dictionary
#pragma: coderesponse end


#pragma: coderesponse template
def bag_of_words(texts):
    """
    Inputs a list of string reviews
    Returns a dictionary of unique unigrams occurring over the input

    Feel free to change this code as guided by Problem 9
    """
    # Your code here
    stop_words = pd.read_csv("C:/root/workspace/MITx_6.86x/07_Projects/project1/
 ↪stopwords.txt", header=None, names='w')
    stop_words = stop_words["w"].tolist()

    dictionary = {} # maps word to unique index
    for text in texts:
        word_list = extract_words(text)
```

```
        for word in word_list:
            if word not in stop_words:
                if word not in dictionary:
                    dictionary[word] = len(dictionary)
    return dictionary
#pragma: coderesponse end


#pragma: coderesponse template
def extract_bow_feature_vectors_original(reviews, dictionary):
    """
    Inputs a list of string reviews
    Inputs the dictionary of words as given by bag_of_words
    Returns the bag-of-words feature matrix representation of the data.
    The returned matrix is of shape (n, m), where n is the number of reviews
    and m the total number of entries in the dictionary.

    Feel free to change this code as guided by Problem 9
    """
    # Your code here
    # see extract_bow_feature_vectors

    num_reviews = len(reviews)
    feature_matrix = np.zeros([num_reviews, len(dictionary)])

    for i, text in enumerate(reviews):
        word_list = extract_words(text)
        for word in word_list:
            if word in dictionary:
                feature_matrix[i, dictionary[word]] = 1
    return feature_matrix
#pragma: coderesponse end


#pragma: coderesponse template
def accuracy(preds, targets):
    """
    Given length-N vectors containing predicted and target labels,
    returns the percentage and number of correct predictions.
    """
    return (preds == targets).mean()
#pragma: coderesponse end


#pragma: coderesponse template
def extract_bow_feature_vectors(reviews, dictionary):
    """
```

```
    Inputs a list of string reviews
    Inputs the dictionary of words as given by bag_of_words
    Returns the bag-of-words feature matrix representation of the data.
    The returned matrix is of shape (n, m), where n is the number of reviews
    and m the total number of entries in the dictionary.

    Feel free to change this code as guided by Problem 9
    """
    # Your code here

    num_reviews = len(reviews)
    feature_matrix = np.zeros([num_reviews, len(dictionary)])

    for i, text in enumerate(reviews):
        word_list = extract_words(text)
        for word in word_list:
            if word in dictionary:
                feature_matrix[i, dictionary[word]] += 1
    return feature_matrix
#pragma: coderesponse end
```

3). main.py

[ ]:
```
import project1 as p1
import utils
import numpy as np

#------------------------------------------------------------------------
# Data loading. There is no need to edit code in this section.
#------------------------------------------------------------------------

train_data = utils.load_data('reviews_train.tsv')
val_data = utils.load_data('reviews_val.tsv')
test_data = utils.load_data('reviews_test.tsv')

train_texts, train_labels = zip(*((sample['text'], sample['sentiment']) for␣
 ↪sample in train_data))
val_texts, val_labels = zip(*((sample['text'], sample['sentiment']) for sample␣
 ↪in val_data))
test_texts, test_labels = zip(*((sample['text'], sample['sentiment']) for␣
 ↪sample in test_data))

dictionary = p1.bag_of_words_original(train_texts)

train_bow_features = p1.extract_bow_feature_vectors_original(train_texts,␣
 ↪dictionary)
```

```
val_bow_features = p1.extract_bow_feature_vectors_original(val_texts,␣
 ↪dictionary)
test_bow_features = p1.extract_bow_feature_vectors_original(test_texts,␣
 ↪dictionary)

dictionary2 = p1.bag_of_words(train_texts)

train_bow_features2 = p1.extract_bow_feature_vectors_original(train_texts,␣
 ↪dictionary2)
val_bow_features2 = p1.extract_bow_feature_vectors_original(val_texts,␣
 ↪dictionary2)
test_bow_features2 = p1.extract_bow_feature_vectors_original(test_texts,␣
 ↪dictionary2)

train_bow_features3 = p1.extract_bow_feature_vectors(train_texts, dictionary2)
val_bow_features3 = p1.extract_bow_feature_vectors(val_texts, dictionary2)
test_bow_features3 = p1.extract_bow_feature_vectors(test_texts, dictionary2)
#---------------------------------------------------------------------------
# Problem 5
#---------------------------------------------------------------------------

# toy_features, toy_labels = toy_data = utils.load_toy_data('toy_data.tsv')

# T = 10
# L = 0.2

# thetas_perceptron = p1.perceptron(toy_features, toy_labels, T)
# thetas_avg_perceptron = p1.average_perceptron(toy_features, toy_labels, T)
# thetas_pegasos = p1.pegasos(toy_features, toy_labels, T, L)

# def plot_toy_results(algo_name, thetas):
#     print('theta for', algo_name, 'is', ', '.join(map(str,list(thetas[0]))))
#     print('theta_0 for', algo_name, 'is', str(thetas[1]))
#     utils.plot_toy_data(algo_name, toy_features, toy_labels, thetas)

# plot_toy_results('Perceptron', thetas_perceptron)
# plot_toy_results('Average Perceptron', thetas_avg_perceptron)
# plot_toy_results('Pegasos', thetas_pegasos)

#---------------------------------------------------------------------------
# Problem 7
#---------------------------------------------------------------------------

# T = 10
# L = 0.01

# pct_train_accuracy, pct_val_accuracy = \
```

```
#    p1.classifier_accuracy(p1.perceptron,␣
 ↪train_bow_features,val_bow_features,train_labels,val_labels,T=T)
# print("{:35} {:.4f}".format("Training accuracy for perceptron:",␣
 ↪pct_train_accuracy))
# print("{:35} {:.4f}".format("Validation accuracy for perceptron:",␣
 ↪pct_val_accuracy))

# avg_pct_train_accuracy, avg_pct_val_accuracy = \
#    p1.classifier_accuracy(p1.average_perceptron,␣
 ↪train_bow_features,val_bow_features,train_labels,val_labels,T=T)
# print("{:43} {:.4f}".format("Training accuracy for average perceptron:",␣
 ↪avg_pct_train_accuracy))
# print("{:43} {:.4f}".format("Validation accuracy for average perceptron:",␣
 ↪avg_pct_val_accuracy))

# avg_peg_train_accuracy, avg_peg_val_accuracy = \
#    p1.classifier_accuracy(p1.pegasos,␣
 ↪train_bow_features,val_bow_features,train_labels,val_labels,T=T,L=L)
# print("{:50} {:.4f}".format("Training accuracy for Pegasos:",␣
 ↪avg_peg_train_accuracy))
# print("{:50} {:.4f}".format("Validation accuracy for Pegasos:",␣
 ↪avg_peg_val_accuracy))

#-----------------------------------------------------------------------------
# Problem 8
#-----------------------------------------------------------------------------

# data = (train_bow_features, train_labels, val_bow_features, val_labels)

# values of T and lambda to try
# Ts = [1, 5, 10, 15, 25, 50]
# Ls = [0.001, 0.01, 0.1, 1, 10]

# pct_tune_results = utils.tune_perceptron(Ts, *data)
# print('perceptron valid:', list(zip(Ts, pct_tune_results[1])))
# print('best = {:.4f}, T={:.4f}'.format(np.max(pct_tune_results[1]), Ts[np.
 ↪argmax(pct_tune_results[1])]))

# avg_pct_tune_results = utils.tune_avg_perceptron(Ts, *data)
# print('avg perceptron valid:', list(zip(Ts, avg_pct_tune_results[1])))
# print('best = {:.4f}, T={:.4f}'.format(np.max(avg_pct_tune_results[1]), Ts[np.
 ↪argmax(avg_pct_tune_results[1])]))

# # fix values for L and T while tuning Pegasos T and L, respective
# fix_L = 0.01
# peg_tune_results_T = utils.tune_pegasos_T(fix_L, Ts, *data)
```

```
# print('Pegasos valid: tune T', list(zip(Ts, peg_tune_results_T[1])))
# print('best = {:.4f}, T={:.4f}'.format(np.max(peg_tune_results_T[1]), Ts[np.
 ↪argmax(peg_tune_results_T[1])]))

# fix_T = Ts[np.argmax(peg_tune_results_T[1])]
# peg_tune_results_L = utils.tune_pegasos_L(fix_T, Ls, *data)
# print('Pegasos valid: tune L', list(zip(Ls, peg_tune_results_L[1])))
# print('best = {:.4f}, L={:.4f}'.format(np.max(peg_tune_results_L[1]), Ls[np.
 ↪argmax(peg_tune_results_L[1])]))

# utils.plot_tune_results('Perceptron', 'T', Ts, *pct_tune_results)
# utils.plot_tune_results('Avg Perceptron', 'T', Ts, *avg_pct_tune_results)
# utils.plot_tune_results('Pegasos', 'T', Ts, *peg_tune_results_T)
# utils.plot_tune_results('Pegasos', 'L', Ls, *peg_tune_results_L)

#-----------------------------------------------------------------------------
# Use the best method (perceptron, average perceptron or Pegasos) along with
# the optimal hyperparameters according to validation accuracies to test
# against the test dataset. The test data has been provided as
# test_bow_features and test_labels.
#-----------------------------------------------------------------------------

# Your code here
# data = (train_bow_features, test_bow_features, train_labels, test_labels)
# kwargs = {"T":25.0000, "L":0.0100}

# print(p1.classifier_accuracy(p1.perceptron, *data, T=25.0000))
# print(p1.classifier_accuracy(p1.average_perceptron, *data, T=25.0000))
# print(p1.classifier_accuracy(p1.pegasos, *data, **kwargs))

# theta_star, theta_star_0 = p1.perceptron(train_bow_features, train_labels,␣
 ↪T=25.0000)
# ['originally', 'bottle', 'perfectly', 'pleased', 'wanting', 'satisfied',␣
 ↪'reasonable', 'newman', 'hands', 'glad']

# theta_star, theta_star_0 = p1.average_perceptron(train_bow_features,␣
 ↪train_labels, T=25.0000)
# ['pleased', 'originally', 'perfectly', 'bottle', 'delicious', 'hands',␣
 ↪'glad', 'perfect', 'satisfied', 'reasonable']

# theta_star, theta_star_0 = p1.pegasos(train_bow_features, train_labels,␣
 ↪**kwargs)
# pred_train_labels = p1.classify(train_bow_features, theta_star, theta_star_0)␣
 ↪
# pred_test_labels = p1.classify(test_bow_features, theta_star, theta_star_0)
# print(p1.accuracy(pred_test_labels, test_labels))
```

```
# print(theta_star, theta_star_0)

#-------------------------------------------------------------------------------
# Assign to best_theta, the weights (and not the bias!) learned by your most
# accurate algorithm with the optimal choice of hyperparameters.
#-------------------------------------------------------------------------------

# best_theta = theta_star # Your code here
# wordlist   = [word for (idx, word) in sorted(zip(dictionary.values(),␣
 ↪dictionary.keys()))]
# sorted_word_features = utils.most_explanatory_word(best_theta, wordlist)
# print("Most Explanatory Word Features")
# print(sorted_word_features[:10])
# ['delicious', 'great', '!', 'best', 'perfect', 'loves', 'wonderful', 'glad',␣
 ↪'love', 'quickly']

#␣
 ↪-------------------------------------------------------------------------------
# Problem 9
#␣
 ↪-------------------------------------------------------------------------------
# 1) Remove Stop Words

# Try to implement stop words removal in your feature engineering code.␣
 ↪Specifically, load the file stopwords.txt,
# remove the words in the file from your dictionary, and use features␣
 ↪constructed from the new dictionary to train
# your model and make predictions.

# Compare your result in the testing data on Pegasos algorithm using
# T=25 and L=0.01
# when you remove the words in stopwords.txt from your dictionary.

# Your code here

# data = (train_bow_features2, test_bow_features2, train_labels, test_labels)
# kwargs = {"T":25.0000, "L":0.0100}

# theta_star, theta_star_0 = p1.pegasos(train_bow_features2, train_labels,␣
 ↪**kwargs)
# pred_train_labels = p1.classify(train_bow_features2, theta_star,␣
 ↪theta_star_0)
# pred_test_labels = p1.classify(test_bow_features2, theta_star, theta_star_0)
# print(p1.accuracy(pred_test_labels, test_labels))
# print(theta_star, theta_star_0)
```

```
# best_theta = theta_star # Your code here
# wordlist   = [word for (idx, word) in sorted(zip(dictionary2.values(),␣
 ↪dictionary2.keys()))]
# sorted_word_features = utils.most_explanatory_word(best_theta, wordlist)
# print("Most Explanatory Word Features")
# print(sorted_word_features[:10])

# 2) Change Binary Features to Counts Features
# Again, use the same learning algorithm and the same feature as the last␣
 ↪problem. However,
# when you compute the feature vector of a word, use its count in each document␣
 ↪rather than a binary indicator.
# Hint: You are free to modify the extract_bow_feature_vectors function to␣
 ↪compute counts features.

data = (train_bow_features3, test_bow_features3, train_labels, test_labels)
kwargs = {"T":25.0000, "L":0.0100}

theta_star, theta_star_0 = p1.pegasos(train_bow_features3, train_labels,␣
 ↪**kwargs)
pred_train_labels = p1.classify(train_bow_features3, theta_star, theta_star_0)
pred_test_labels = p1.classify(test_bow_features3, theta_star, theta_star_0)
print(p1.accuracy(pred_test_labels, test_labels))
print(theta_star, theta_star_0)

best_theta = theta_star # Your code here
wordlist   = [word for (idx, word) in sorted(zip(dictionary2.values(),␣
 ↪dictionary2.keys()))]
sorted_word_features = utils.most_explanatory_word(best_theta, wordlist)
print("Most Explanatory Word Features")
print(sorted_word_features[:10])
```

4) test.py

[ ]:
```
import os
import sys
import time
import traceback
import project1 as p1
import numpy as np

verbose = False

def green(s):
    return '\033[1;32m%s\033[m' % s

def yellow(s):
```

```
    return '\033[1;33m%s\033[m' % s

def red(s):
    return '\033[1;31m%s\033[m' % s

def log(*m):
    print(" ".join(map(str, m)))

def log_exit(*m):
    log(red("ERROR:"), *m)
    exit(1)


def check_real(ex_name, f, exp_res, *args):
    try:
        res = f(*args)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not np.isreal(res):
        log(red("FAIL"), ex_name, ": does not return a real number, type: ",␣
↪type(res))
        return True
    if res != exp_res:
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
↪got: ", res)
        return True


def equals(x, y):
    if type(y) == np.ndarray:
        return (x == y).all()
    return x == y

def check_tuple(ex_name, f, exp_res, *args, **kwargs):
    try:
        res = f(*args, **kwargs)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not type(res) == tuple:
        log(red("FAIL"), ex_name, ": does not return a tuple, type: ",␣
↪type(res))
        return True
    if not len(res) == len(exp_res):
        log(red("FAIL"), ex_name, ": expected a tuple of size ", len(exp_res),␣
↪" but got tuple of size", len(res))
```

```
        return True
    if not all(equals(x, y) for x, y in zip(res, exp_res)):
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
↪got: ", res)
        return True

def check_array(ex_name, f, exp_res, *args):
    try:
        res = f(*args)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not type(res) == np.ndarray:
        log(red("FAIL"), ex_name, ": does not return a numpy array, type: ",␣
↪type(res))
        return True
    if not len(res) == len(exp_res):
        log(red("FAIL"), ex_name, ": expected an array of shape ", exp_res.
↪shape, " but got array of shape", res.shape)
        return True
    if not all(equals(x, y) for x, y in zip(res, exp_res)):
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
↪got: ", res)
        return True

def check_list(ex_name, f, exp_res, *args):
    try:
        res = f(*args)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not type(res) == list:
        log(red("FAIL"), ex_name, ": does not return a list, type: ", type(res))
        return True
    if not len(res) == len(exp_res):
        log(red("FAIL"), ex_name, ": expected a list of size ", len(exp_res), "␣
↪but got list of size", len(res))
        return True
    if not all(equals(x, y) for x, y in zip(res, exp_res)):
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
↪got: ", res)
        return True


def check_get_order():
    ex_name = "Get order"
```

```
    if check_list(
            ex_name, p1.get_order,
            [0], 1):
        log("You should revert `get_order` to its original implementation for␣
↪this test to pass")
        return
    if check_list(
            ex_name, p1.get_order,
            [1, 0], 2):
        log("You should revert `get_order` to its original implementation for␣
↪this test to pass")
        return
    log(green("PASS"), ex_name, "")


def check_hinge_loss_single():
    ex_name = "Hinge loss single"

    feature_vector = np.array([1, 2])
    label, theta, theta_0 = 1, np.array([-1, 1]), -0.2
    exp_res = 1 - 0.8
    if check_real(
            ex_name, p1.hinge_loss_single,
            exp_res, feature_vector, label, theta, theta_0):
        return
    log(green("PASS"), ex_name, "")


def check_hinge_loss_full():
    ex_name = "Hinge loss full"

    feature_vector = np.array([[1, 2], [1, 2]])
    label, theta, theta_0 = np.array([1, 1]), np.array([-1, 1]), -0.2
    exp_res = 1 - 0.8
    if check_real(
            ex_name, p1.hinge_loss_full,
            exp_res, feature_vector, label, theta, theta_0):
        return

    log(green("PASS"), ex_name, "")


def check_perceptron_single_update():
    ex_name = "Perceptron single update"

    feature_vector = np.array([1, 2])
    label, theta, theta_0 = 1, np.array([-1, 1]), -1.5
```

```
    exp_res = (np.array([0, 3]), -0.5)
    if check_tuple(
            ex_name, p1.perceptron_single_step_update,
            exp_res, feature_vector, label, theta, theta_0):
        return

    feature_vector = np.array([1, 2])
    label, theta, theta_0 = 1, np.array([-1, 1]), -1
    exp_res = (np.array([0, 3]), 0)
    if check_tuple(
            ex_name + " (boundary case)", p1.perceptron_single_step_update,
            exp_res, feature_vector, label, theta, theta_0):
        return

    log(green("PASS"), ex_name, "")


def check_perceptron():
    ex_name = "Perceptron"

    feature_matrix = np.array([[1, 2]])
    labels = np.array([1])
    T = 1
    exp_res = (np.array([1, 2]), 1)
    if check_tuple(
            ex_name, p1.perceptron,
            exp_res, feature_matrix, labels, T):
        return

    feature_matrix = np.array([[1, 2], [-1, 0]])
    labels = np.array([1, 1])
    T = 1
    exp_res = (np.array([0, 2]), 2)
    if check_tuple(
            ex_name, p1.perceptron,
            exp_res, feature_matrix, labels, T):
        return

    feature_matrix = np.array([[1, 2]])
    labels = np.array([1])
    T = 2
    exp_res = (np.array([1, 2]), 1)
    if check_tuple(
            ex_name, p1.perceptron,
            exp_res, feature_matrix, labels, T):
        return
```

```
    feature_matrix = np.array([[1, 2], [-1, 0]])
    labels = np.array([1, 1])
    T = 2
    exp_res = (np.array([0, 2]), 2)
    if check_tuple(
            ex_name, p1.perceptron,
            exp_res, feature_matrix, labels, T):
        return

    log(green("PASS"), ex_name, "")


def check_average_perceptron():
    ex_name = "Average perceptron"

    feature_matrix = np.array([[1, 2]])
    labels = np.array([1])
    T = 1
    exp_res = (np.array([1, 2]), 1)
    if check_tuple(
            ex_name, p1.average_perceptron,
            exp_res, feature_matrix, labels, T):
        return

    feature_matrix = np.array([[1, 2], [-1, 0]])
    labels = np.array([1, 1])
    T = 1
    exp_res = (np.array([-0.5, 1]), 1.5)
    if check_tuple(
            ex_name, p1.average_perceptron,
            exp_res, feature_matrix, labels, T):
        return

    feature_matrix = np.array([[1, 2]])
    labels = np.array([1])
    T = 2
    exp_res = (np.array([1, 2]), 1)
    if check_tuple(
            ex_name, p1.average_perceptron,
            exp_res, feature_matrix, labels, T):
        return

    feature_matrix = np.array([[1, 2], [-1, 0]])
    labels = np.array([1, 1])
    T = 2
    exp_res = (np.array([-0.25, 1.5]), 1.75)
    if check_tuple(
```

```
            ex_name, p1.average_perceptron,
            exp_res, feature_matrix, labels, T):
        return

    log(green("PASS"), ex_name, "")


def check_pegasos_single_update():
    ex_name = "Pegasos single update"

    feature_vector = np.array([1, 2])
    label, theta, theta_0 = 1, np.array([-1, 1]), -1.5
    L = 0.2
    eta = 0.1
    exp_res = (np.array([-0.88, 1.18]), -1.4)
    if check_tuple(
            ex_name, p1.pegasos_single_step_update,
            exp_res,
            feature_vector, label, L, eta, theta, theta_0):
        return

    feature_vector = np.array([1, 1])
    label, theta, theta_0 = 1, np.array([-1, 1]), 1
    L = 0.2
    eta = 0.1
    exp_res = (np.array([-0.88, 1.08]), 1.1)
    if check_tuple(
            ex_name +  " (boundary case)", p1.pegasos_single_step_update,
            exp_res,
            feature_vector, label, L, eta, theta, theta_0):
        return

    feature_vector = np.array([1, 2])
    label, theta, theta_0 = 1, np.array([-1, 1]), -2
    L = 0.2
    eta = 0.1
    exp_res = (np.array([-0.88, 1.18]), -1.9)
    if check_tuple(
            ex_name, p1.pegasos_single_step_update,
            exp_res,
            feature_vector, label, L, eta, theta, theta_0):
        return

    log(green("PASS"), ex_name, "")


def check_pegasos():
```

```
    ex_name = "Pegasos"

    feature_matrix = np.array([[1, 2]])
    labels = np.array([1])
    T = 1
    L = 0.2
    exp_res = (np.array([1, 2]), 1)
    if check_tuple(
            ex_name, p1.pegasos,
            exp_res, feature_matrix, labels, T, L):
        return

    feature_matrix = np.array([[1, 1], [1, 1]])
    labels = np.array([1, 1])
    T = 1
    L = 1
    exp_res = (np.array([1-1/np.sqrt(2), 1-1/np.sqrt(2)]), 1)
    if check_tuple(
            ex_name, p1.pegasos,
            exp_res, feature_matrix, labels, T, L):
        return

    log(green("PASS"), ex_name, "")


def check_classify():
    ex_name = "Classify"

    feature_matrix = np.array([[1, 1], [1, 1], [1, 1]])
    theta = np.array([1, 1])
    theta_0 = 0
    exp_res = np.array([1, 1, 1])
    if check_array(
            ex_name, p1.classify,
            exp_res, feature_matrix, theta, theta_0):
        return

    feature_matrix = np.array([[-1, 1]])
    theta = np.array([1, 1])
    theta_0 = 0
    exp_res = np.array([-1])
    if check_array(
            ex_name + " (boundary case)", p1.classify,
            exp_res, feature_matrix, theta, theta_0):
        return

    log(green("PASS"), ex_name, "")
```

```
def check_classifier_accuracy():
    ex_name = "Classifier accuracy"

    train_feature_matrix = np.array([[1, 0], [1, -1], [2, 3]])
    val_feature_matrix = np.array([[1, 1], [2, -1]])
    train_labels = np.array([1, -1, 1])
    val_labels = np.array([-1, 1])
    exp_res = 1, 0
    T=1
    if check_tuple(
            ex_name, p1.classifier_accuracy,
            exp_res,
            p1.perceptron,
            train_feature_matrix, val_feature_matrix,
            train_labels, val_labels,
            T=T):
        return

    train_feature_matrix = np.array([[1, 0], [1, -1], [2, 3]])
    val_feature_matrix = np.array([[1, 1], [2, -1]])
    train_labels = np.array([1, -1, 1])
    val_labels = np.array([-1, 1])
    exp_res = 1, 0
    T=1
    L=0.2
    if check_tuple(
            ex_name, p1.classifier_accuracy,
            exp_res,
            p1.pegasos,
            train_feature_matrix, val_feature_matrix,
            train_labels, val_labels,
            T=T, L=L):
        return

    log(green("PASS"), ex_name, "")

def check_bag_of_words():
    ex_name = "Bag of words"

    texts = [
        "He loves to walk on the beach",
        "There is nothing better"]

    try:
        res = p1.bag_of_words(texts)
    except NotImplementedError:
```

```
        log(red("FAIL"), ex_name, ": not implemented")
        return
    if not type(res) == dict:
        log(red("FAIL"), ex_name, ": does not return a tuple, type: ",␣
↪type(res))
        return

    vals = sorted(res.values())
    exp_vals = list(range(len(res.keys())))
    if not vals == exp_vals:
        log(red("FAIL"), ex_name, ": wrong set of indices. Expected: ",␣
↪exp_vals, " got ", vals)
        return

    log(green("PASS"), ex_name, "")

    keys = sorted(res.keys())
    exp_keys = ['beach', 'better', 'he', 'is', 'loves', 'nothing', 'on', 'the',␣
↪'there', 'to', 'walk']
    stop_keys = ['beach', 'better', 'loves', 'nothing', 'walk']

    if keys == exp_keys:
        log(yellow("WARN"), ex_name, ": does not remove stopwords:", [k for k␣
↪in keys if k not in stop_keys])
    elif keys == stop_keys:
        log(green("PASS"), ex_name, " stopwords removed")
    else:
        log(red("FAIL"), ex_name, ": keys are missing:", [k for k in stop_keys␣
↪if k not in keys], " or are not unexpected:", [k for k in keys if k not in␣
↪stop_keys])


def check_extract_bow_feature_vectors():
    ex_name = "Extract bow feature vectors"
    texts = [
        "He loves her ",
        "He really really loves her"]
    keys = ["he", "loves", "her", "really"]
    dictionary = {k:i for i, k in enumerate(keys)}
    exp_res = np.array(
        [[1, 1, 1, 0],
        [1, 1, 1, 1]])
    non_bin_res = np.array(
        [[1, 1, 1, 0],
        [1, 1, 1, 2]])
```

```
    try:
        res = p1.extract_bow_feature_vectors(texts, dictionary)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return

    if not type(res) == np.ndarray:
        log(red("FAIL"), ex_name, ": does not return a numpy array, type: ",␣
↪type(res))
        return
    if not len(res) == len(exp_res):
        log(red("FAIL"), ex_name, ": expected an array of shape ", exp_res.
↪shape, " but got array of shape", res.shape)
        return

    log(green("PASS"), ex_name)

    if (res == exp_res).all():
        log(yellow("WARN"), ex_name, ": uses binary indicators as features")
    elif (res == non_bin_res).all():
        log(green("PASS"), ex_name, ": correct non binary features")
    else:
        log(red("FAIL"), ex_name, ": unexpected feature matrix")
        return

def main():
    log(green("PASS"), "Import project1")
    try:
        check_get_order()
        check_hinge_loss_single()
        check_hinge_loss_full()
        check_perceptron_single_update()
        check_perceptron()
        check_average_perceptron()
        check_pegasos_single_update()
        check_pegasos()
        check_classify()
        check_classifier_accuracy()
        check_bag_of_words()
        check_extract_bow_feature_vectors()
    except Exception:
        log_exit(traceback.format_exc())

if __name__ == "__main__":
    main()
```