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:

1. Implement and compare three types of linear classifiers: the **perceptron** algorithm, the **average perceptron** algorithm, and the **Pegasos** algorithm.
2. Use your classifiers on the food review dataset, using some simple text features.
3. 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.**

1. *Note on software: For this project, you will need the **NumPy** numerical toolbox, and the **matplotlib** plotting toolbox.*
2. Download the Project1 folder and upload it to your azure notebooks account. The 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.
- **MainML_project1.ipynb** 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.
- **TestML_project1.ipynb** 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.

How to Test: Simply run the TestML_project1.ipynb script. A checklist should appear indicating the methods that have already been implemented successfully.

How to Run your Project 1 Functions: Simply run the MainML_project1.ipynb.

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 θ and $\theta 0$. 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.

Hinge Loss on One Data Sample

First, implement the basic hinge loss calculation on a single data-point. Instead of the

```
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.
```

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.

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 kth element of the labels vector. This function should return the appropriate loss of the classifier on the given dataset.

```
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
```

Perceptron Algorithm

Perceptron Single Step Update

Now you will implement the single step update for the perceptron algorithm (implemented with $0-1$ loss). You will be given the feature vector as an array of numbers, the current θ and θ_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 θ and the second element is the correctly updated value of θ_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 $[-\varepsilon,\varepsilon]$. That is, when x is a float, "$x=0$" should be checked with $|x|<\varepsilon$.

```
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
```

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 θ and θ_0 to zero. This function should return a tuple in which the first element is the final value of θ and the second element is the value of θ_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.

```
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
```

.

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 θ, however, are an average of the θs across the nT steps:

$$\theta_{final} = \frac{1}{nT}(\theta^{(1)} + \theta^{(2)} + \ldots + \theta^{(nT)})$$

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 θ and θ_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.

```
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.
```

Pegasos Algorithm

Now you will implement the Pegasos algorithm. For more information, refer to the original pape:

Pegasos: Primal Estimated sub-GrAdientSOlver for SVM

The following pseudo-code describes the Pegasos update rule.

Pegasos update rule $\left(x^{(i)}, y^{(i)}, \lambda, \eta, \theta\right)$:

if $y^{(i)}\left(\theta \cdot x^{(i)}\right) \leq 1$ then

update $\theta = (1 - \eta\lambda)\,\theta + \eta y^{(i)} x^{(i)}$

else:

update $\theta = (1 - \eta\lambda)\,\theta$

The η parameter is a decaying factor that will decrease over time. The Λ parameter is a regularizing parameter. In this problem, you will need to adapt this update rule to add a bias term (θ_0) to the hypothesis, but take care not to penalize the magnitude of θ_0.

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 λ and η value to use for updates.

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

```
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.
```

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 θ and θ_0 to zero. For each update, set $\eta = 1/\sqrt{t}$ 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 θ and the second element is the value of θ_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.

.

```
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
```

.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,λ=0.2. **MainML:project1.ipynb** will compute θ and θ_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.

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.

Classification

Implement a classification function that uses θ and θ_0 to classify a set of data points. You are given the feature matrix, θ, and θ_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 $|x|<\epsilon$

```
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
```

.

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 λ 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 **MainML_project1.ipynb** Problem 8 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 λ [0.001, 0.01, 0.1, 1, 10]. For pegasos algorithm, first fix $\lambda=0.01$ to tune T, and then use the best T to tune λ.

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 λ for the Pegasos algorithm.

Note: Just enter the values printed in your MainML:project1.py. Note that for the Pegasos algorithm, the result does not reflect the best combination of T and λ

.

For the **perceptron** algorithm:

T = [], with validation accuracy = []

For the **average perceptron** algorithm:

T = [], with validation accuracy = []

For the **pegasos** algorithm:

T = [], λ=[] with validation accuracy = []

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 : []

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:

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.

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: