

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.036—Introduction to Machine Learning
 Spring 2017

Project 1: Automatic Review Analyzer. Issued: Mon., 2/20 Due: Fri., 3/3 at 9am

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. Table 1 shows an example of two reviews from our dataset and their corresponding labels.

Review	Label
<i>Nasty No flavor. The candy is just red , No flavor . Just plan and chewy . I would never buy them again</i>	-1
<i>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!</i>	1

Table 1: Example entries from the reviews dataset. The two reviews were written by different customers describing their experience with a sugar-free candy.

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 (Section 1).
- Use your classifiers on the food review dataset, using some simple text features (Section 2).
- Experiment with additional features and explore their impact on classifier performance (Section 3).

In addition, as part of the project we will run a competition where you will attempt to classify a set of examples for which no labels are provided. You will receive extra credit if your method performs particularly well on this dataset (see below for details).

0. Setup (0 Points)

For this project and throughout the course we will be using Python 3.0 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.**

0. Download the necessary libraries and tools.

*Note on software: For the all the 6.036 projects, we will use python 3.0 augmented with the **NumPy** numerical toolbox, the **matplotlib** plotting toolbox, and the **scikit.learn** machine learning toolbox (though you are not permitted to use **scikit.learn** until project two).*

1. Download `project1.zip` from Stellar and unzip it in to a working directory. The zip file 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_suite.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 autograder.

1. Implementing classifiers (55 Points)

First we will implement three linear classifiers: (1) perceptron, (2) average perceptron, and (3) Pegasos.

All the algorithms will run for a fixed number of iterations, T , through the training set. As a result, the parameters will be updated at most nT times, where n is the number of training examples that you are given.

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 θ s across the nT steps:

$$\theta_{\text{final}} = \frac{1}{nT} \left(\theta^{(1)} + \theta^{(2)} + \dots + \theta^{(nT)} \right)$$

All of the following functions can be found in the `project1.py` script.

2. You will begin by implementing the hinge loss function. For this function you will be given the parameters, θ and θ_0 , a feature matrix, in which the rows are feature vectors and the columns are individual features, and a vector of +1/-1 labels representing the actual sentiment of the corresponding feature vector. 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. [5 Points]
 3. 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 . [10 Points]
 - 4-a. 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 step 2. 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 . [5 Points]
 - 4-b. You will now implement the average perceptron algorithm. This function should be constructed similarly to the perceptron algorithm from part 4-a, except that it should return the average values of θ and θ_0 . [10 Points]
- Hint: Tracking a moving average through loops is difficult, but tracking a sum through loops is simple.

5. Next you will implement the single step update for the Pegasos algorithm. This function is very similar to the function that you implemented in step 3, 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. [15 Points]
6. Finally you will implement the full Pegasos algorithm. You will be given the same feature matrix and labels array as you were given in step 2. 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 $\eta = \frac{1}{\sqrt{t}}$ 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 θ and the second element is the value of θ_0 . [5 Points]
7. 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 $T = 10$, $\lambda = 0.2$. `main.py` 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. **Why do these algorithms provide different decision boundaries? Hand in the plots obtained for each algorithm along with a short paragraph answering the question in your pdf writeup.** [5 Points]

2. The automatic review analyzer (35 Points)

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) and `reviews_submit.tsv` (500 examples). Only the first three sets are labeled—you will use them to train, tune, and evaluate your classifier. Once you have optimized your method, you will use it to predict labels for the submit data, which you will submit along with your code and answers.

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.

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 zero, 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, section 2, 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 Section 1, you will be able to compute θ and θ_0 .

For this section, your tasks are the following:

8. Implement the `classify` function in the file `project1.py`. This function takes 3 parameters, the feature matrix, the θ parameter vector found by your learning algorithms, and the θ_0 offset. The function should return an array of the predicted classifications of the data points in the feature matrix. [10 Points]
- 9-a. We have supplied you with an `accuracy` function that can be found in the `project1.py` file. This 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 the `perceptron_accuracy`, `average_perceptron_accuracy`, and `pegasos_accuracy` functions. All of the methods take a T value, and the Pegasos method takes a λ value, as well. The functions should first train themselves using the supplied train data and then 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.
- 9-b. When these functions are complete, uncomment the relevant lines in `main.py` and **report the training and validation accuracies of each algorithm with $T = 10$ and $\lambda = 0.01$ in your pdf writeup (the λ value only applies to Pegasos)**.
10. To improve the performance of your methods, you will need to *tune* (i.e. optimize) the hyper-parameters T and λ . To this end, you will train your classifiers with several values of λ and T , each time using the validation data to assess the performance. Reasonable values to try are $\lambda = [0.01, 0.1, 0.2, 0.5, 1]$ and $T = [1, 5, 10, 15, 25, 50, 100]$. For the Pegasos algorithm, you can first fix $\lambda = 0.01$, and then vary T to find the best performing T . After tuning T , fix T to your tuned value and vary λ . (Alternatively, you could perform a grid search over the two parameters to jointly tune λ and T .) You may also test your own range of λ and T values if you so wish. When tuning λ and tuning T for your classifiers, compute the training and validation accuracies and plot your results for each. The code to plot these functions can be found in the `main.py` file. Simply uncomment the appropriate lines and fill out the `Ts` and `Ls` lists. Once you have tuned the T values for the Pegasos algorithm, you will also need to fill out the `best_T` variable. This variable will be used while tuning λ .
Hand in your plot and your solutions to the following questions: [10 Points]
 - (a) Do the training and validation accuracies behave similarly as a function of λ and T ? Why or why not?
 - (b) Which algorithm performed the best of the three?
 - (c) What are the optimal values of T for your perceptron? For average perceptron? What are the optimal values of T and λ for your Pegasos algorithm?
- 11-a. 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`. Call the appropriate accuracy function to find the accuracy on the test set. **Report this value in your writeup. [3 Points]**

- 11-b. According to the largest weights (i.e. individual θ_i values in your θ vector), you can find out which unigrams were the most impactful ones in predicting the labels. Uncomment the relevant part in `main.py` and **report top ten most explanatory word features in your writeup. [2 Points]**

3. New Features and the challenge (20 Points)

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 augmenting columns to the feature matrix. For example, you are encouraged to try the union of unigram and bigram features. Using the same example from the previous section, the final dictionary would be (*Mary, loves, apples, red, Mary loves, loves apples, Red apples*) and the feature representations would be (1, 1, 1, 0, 1, 1, 0) and (0, 0, 1, 1, 0, 0, 1). 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,

- Remove common, or *stop*, words from the unigram-bigram dictionary (e.g., words such as “the”, “to”, “for”)
- 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.

We have provided you the list of stop words in the file `stopwords.txt`. There are also many other things you could change when training your model:

- **Normalize Feature Vectors** - Because the feature vectors indicate the occurrence of each word in the reviews, they will be of varying magnitude (i.e. a longer review will have a lot more 1s). This can cause a the Hinge Loss to be extremely high for longer reviews because the value of $\theta \cdot x^{(i)}$ will be much larger for a larger $x^{(i)}$. Normalizing the feature vectors (i.e. scaling them so their magnitudes are 1) can potentially solve this problem
- **Change Pegasos’s learning rate** from $1/\sqrt{t}$ to some other function - Why should we choose $1/\sqrt{t}$ as the learning rate each iteration? Try playing around with this function of t and consider choosing something different.
- **Sampling** - The Pegasos algorithm discussed in class functions by randomly selecting a feature vector each step. This does not ensure that the algorithm uses your entire set of training data, and it may end up using certain vectors many times (because they are randomly selected often). Instead of doing this, you may wish to randomly permute your feature vectors to make sure you see all examples at some

point in the algorithm (Note: be careful that you keep the labels and feature vectors aligned when doing this!).

Another interesting idea is to consider batching your training examples into subgroups (called mini-batches), each with positive and negative examples, so you ensure you never accidentally look at too many positive or negative data points in a row (as this will throw off your θ by a lot).

- Lastly, be creative: 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 its worth experimenting with some simpler ones before trying anything too complicated

Note: If you wish to read more on Pegasos, feel free to read [their paper](#).

12. You should now experiment with at least one additional improvement (from the suggested improvements above or you can come up with your own) and compare the performance of your method to your results with the original features. You can modify functions `bag_of_words` and `extract_bow_feature_vectors` should you improve the original bag-of-words features. **NOTE: Section 2 depends on these functions, so if you wish to reproduce section 2 results, use the original function implementations. For this reason you may wish to create our own versions of these functions instead of directly modifying the original.** To augment any additional feature columns besides bag-of-words, you can do so by implementing `extract_additional_features`. Again, you should use the validation set to choose values for the parameters and **report the error on the test set. Your report should contain:**

- A clear explanation of how you changed the feature set, and why you think the features you chose might be useful. Include the code you use for this purpose in `extract_additional_features`.
- A description of the experiment you conducted to compare it to the original feature-computation method, and the results of that experiment. Include plots if applicable.

It's okay if it turns out that your new method performs worse than the original one—just explain and document your results. It is very common that a feature you add will worsen your model, so do not be discouraged by this. If you wish to improve your model as much as possible, it's worth testing a few different ideas and seeing if they help or not! [20 Points]

13. **(Extra Credit)** Now, after hours of twiddling features, classifiers, and hyperparameters, it is time to put your machine learning skills to the test by classifying the reviews in `reviews_submit.tsv` for which the labels are not provided.

Modify this section's code as directed in `main.py` to populate `reviews_submit.tsv` with your predictions. Including this file in your project submission will be your entry into the (mandatory) class competition!

You will receive 3 points of extra credit if the accuracy of your submitted labels is above the staff baseline (the staff is busy and probably won't spend as much time as you). You will receive 10 points extra credit and looks of jealousy from all of your friends if your predictions are the best in the class. The top performing scores will be posted on Piazza.

4. Submit Instructions

Please submit a .zip file to stellar named `project1.zip`. This file should contain `project1.py`, `main.py`, and `utils.py`. It should also contain `reviews_submit.tsv` and your report in PDF format named `writeup.pdf`.

In your pdf writeup, please include the written answers and/or plots from the following sections:

- 7
- 9ab
- 10abc
- 11ab
- 12

Please also include all the files that are required to run the code in the zip file. We have provided a python script `checker.py` which checks whether the required files are in the directory as well as some of the basic behavior in the functions you implemented in `project1.py`. Please place the script and the zip file that you are going to submit in the same directory and run the script before submitting.