DCC 030 049 831: Machine Learning
Programming Assignment 1: Perceptron and SVM

Due: Tuesday, April 19, 2016, at 5pm (Submit via UFMG Moodle)

Your submission should include a PDF file called "solutions.pdf" with your answers to the below questions (including plots), and all of the code that you write. Also, include all of the code in a folder called "code_PA1".

Instructions: You may use the programming language of your choice (we strongly recommend Python, and using matplotlib for plotting). However, you are not permitted to use or reference any machine learning code or packages not written by yourself.

1 Introduction

In this assignment, we'll be working with natural language data. In particular, we'll be doing e-mail spam classification. This problem will give you the opportunity to try your hand at feature engineering, which is one of the most important parts of many data science problems. From a technical standpoint, this homework has two pieces. First, you'll implement the Perceptron algorithm. Second, you'll be implementing Pegasos. Pegasos is essentially stochastic subgradient descent for the SVM with a particular schedule for the step-size.

2 The Data

We have provided you with two files: spam_train.txt and spam_test.txt. Each row of the data files corresponds to a single email. The first column gives the label (1=spam, 0=not spam).

The dataset included for this assignmnet is based on a subset of the SpamAssassin Public Corpus. While many emails would contain similar types of entities (e.g., numbers, other URLs, or other email addresses), the specific entities (e.g., the specific URL or specific dollar amount) will be different in almost every email. Therefore, one method often employed in processing emails is to "normalize" these values, so that all URLs are treated the same, all numbers are treated the same, etc. For example, we could replace each URL in the email with the unique string "httpaddr" to indicate that a URL was present. This has the effect of letting the spam classifier make a classification decision based on whether any URL was present, rather than whether a specific URL was present. This typically improves the performance of a spam classifier, since spammers often randomize the URLs, and thus the odds of seeing any particular URL again in a new piece of spam is very small.

We have already implemented the following email preprocessing steps: lower-casing; removal of HTML tags; normalization of URLs, e-mail addresses, and numbers. In addition, words are reduced to their stemmed form. For example, "discount", "discounts", "discounted" and "discounting" are all replaced with "discount". Finally, we removed all non-words and punctuation.

1. This programming assignment will involve your implementing several variants of the Perceptron algorithm and SVM via Pegasos. Before you can build these models and measure their performance, split your training data (i.e. spam_train.txt) into a training and validate set, putting the last 1000 emails into the validation set. Thus, you will have a new training set with 4000 emails and a validation set with 1000 emails. You will not use spam_test.txt until the next assignment.

   Explain why measuring the performance of your final classifier would be problematic had you not created this validation set.

3 Text Representation

The most basic way to represent text documents for machine learning is with a "bag-of-words" representation. Here every possible word is a feature, and the value of a word feature is the number of times that word appears in the document. Of course, most words will not appear in any particular document, and those counts will be zero. Rather than store a huge number of zeros, you may use a sparse representation, in which we only store the counts that are nonzero.

1. Transform all of the data into feature vectors. Build a vocabulary list using only the 4000 e-mail training set by finding all words that occur across the training set. Note that we assume that the data in the validation and test sets is completely unseen when we train our model, and thus we do not use any information contained in them. Ignore all words that appear in fewer than $X = 30$ e-mails of the 4000 e-mail training set - this is both a means of preventing overfitting and of improving scalability. For each email, transform it into a feature vector $\vec{x}$ where the ith entry, x_i , is 1 if the ith word in the vocabulary occurs in the email, and 0 otherwise. Hint: you may find Python's Counter class to be useful here: `https://docs.python.org/2/library/collections.html`. Note that a Counter is also a dict.

4 Perceptron

The perceptron algorithm is often the first classification algorithm taught in machine learning classes. Suppose we have a labeled training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$. In the perceptron algorithm, we are looking for a hyperplane that perfectly separates the classes. That is, we're looking for $w \in \mathbf{R}^d$ such that

$$y_i w^T x_i > 0 \; \forall i \in \{1, \ldots, n\}.$$

Visually, this would mean that all the x's for which $y = 1$ are on one side of the hyperplane $\{x \mid w^T x = 0\}$, and all the $x's$ for which $y = -1$ are on the other side. When such a hyperplane exists, we say that the data are **linearly separable**. The perceptron algorithm is given in Algorithm 1.

1. Implement the functions perceptron_train(data) and perceptron_test(w, data). The function perceptron _train(data) trains a perceptron classifier using the examples provided to the function, and should return $\vec{w}$, k, and iter, the final classification vector, the number of updates (mistakes) performed, and the number of passes through the data, respectively. You may assume that the input data provided to your function is linearly separable (so the stopping criterion should be that all points are correctly classified). For the corner case of $\vec{w} \cdot \vec{x} = 0$, predict the +1 (spam) class.

```
input: Training set (x_1, y_1), ..., (x_n, y_n) ∈ R^d × {−1, 1}
w^(0) = (0, ..., 0) ∈ R^d
k = 0  # step number
repeat
  all_correct = TRUE
  for i = 1, 2, ..., n  # loop through data
    if (y_i x_i^T w^(k) ≤ 0)
      w^(k+1) = w^(k) + y_i x_i
      all_correct = FALSE
    else
      w^(k+1) = w^(k)
    end if
    k = k + 1
  end for
until (all_correct == TRUE)
return w^(k)
```

Algorithm 1: Perceptron Algorithm

For this exercise, you do not need to add a bias feature to the feature vector (it turns out not to improve classification accuracy, possibly because a frequently occurring word already serves this purpose). Your implementation should cycle through the data points in the order as given in the data files (rather than randomizing), so that results are consistent for grading purposes.

The function perceptron_test(w, data) should take as input the weight vector $\vec{w}$ (the classification vector to be used) and a set of examples. The function should return the test error, i.e. the fraction of examples that are misclassified by $\vec{w}$.

2. Train the linear classifier using your training set. How many mistakes are made before the algorithm terminates? Test your implementation of perceptron_test by running it with the learned parameters and the training data, making sure that the training error is zero. Next, classify the emails in your validation set. What is the validation error?

3. To better understand how the spam classifier works, we can inspect the parameters to see which words the classifier thinks are the most predictive of spam. Using the vocabulary list together with the parameters learned in the previous question, output the 15 words with the most *positive weights*. What are they? Which 15 words have the most *negative weights*?

4. Also for $N = 100, 200, 400, 800, 2000, 4000$, create a plot of the number of perceptron iterations as a function of N, where by iteration we mean a complete pass through the training data. As the amount of training data increases, the margin of the training set decreases, which generally leads to an increase in the number of iterations perceptron takes to converge (although it need not be monotonic).

5. One consequence of this is that the later iterations typically perform updates on only a small subset of the data points, which can contribute to overfitting. A way to solve this is to control the maximum number of iterations of the perceptron algorithm. Add an argument of the perceptron algorithm that controls the maximum number of passes over the data.

5 Support Vector Machine via Pegasos

In this question you will build an SVM using the Pegasos algorithm. To align with the notation used in the Pegasos paper[1], we're considering the following formulation of the SVM objective function:

$$f(w) = \min_{w \in \mathbf{R}^n} \frac{\lambda}{2}\|w\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\left\{0, 1 - y_i w^T x_i\right\}.$$

Note that, for simplicity, we are leaving off the unregularized bias term b, and the expression with "max" is just another way to write $\left(1 - y_i w^T x\right)_+$. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$. The pseudocode is given below:

Input: $\lambda > 0$. Choose $w_1 = 0, t = 0$
While epoch $\leq$ max_epochs
  For $j = 1, \ldots, m$ (assumes data is randomly permuted)
    $t = t + 1$
    $\eta_t = 1/(t\lambda)$;
    If $y_j w_t^T x_j < 1$
      $w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$
    Else
      $w_{t+1} = (1 - \eta_t \lambda) w_t$

Note: To keep your algorithm simple, we will not use an offset term b when optimizing the SVM primal objective using Pegasos.

1. Implement the function pegasos_svm_train(data, lambd). The function should return w, the final classification vector. For simplicity, the stopping criterion is set so that the total number of passes over the training data is 20. After each pass through the data, evaluate the SVM objective $f(w)$. Plot $f(w)$ as a function of iteration (i.e. for $t = |data|, ..., 20|data|$), and submit the plot for $\lambda = 2^5$.

2. Implement the function pegasos_svm_test(data, w).

3. Run your learning algorithm for various values of the regularization constant, $\lambda = 2^{-9}, 2^{-8}, .., 2^1, 2^9$. Plot the average training error and average validation error as a function of $\log \lambda$. What is the minimum of your validation error? For the classifier that has the smallest validate error: What is the test error? How many training samples are support vectors? How did you find them? Compare your test error with your result from perceptron.

6 Features [Optional]

For a problem like this, the features you use are far more important than the learning model you choose. Whenever you enter a new problem domain, one of your first orders of business is to beg, borrow, or steal the best features you can find. This means looking at any relevant published work and seeing what they've used. Maybe it means asking a colleague what features they use. But eventually you'll need to engineer new features that help in your particular situation.

[1] Shalev-Shwartz et al.'s "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM" http://ttic.uchicago.edu/~nati/Publications/PegasosMPB.pdf

1. Try to get the best performance possible by generating lots of new features, so long as you are using the same core SVM model. Describe what you tried, and how much improvement each thing brought to the model. To get you thinking on features, here are some basic ideas of varying quality: 1) how many words are in the e-mail? 2) How many "negative" words are there? (You'd have to construct or find a list of negative words.) 3) Word n-gram features: Instead of single-word features, you can make every pair of consecutive words a feature. 4) Character n-gram features: Ignore word boundaries and make every sequence of n characters into a feature (this will be a lot). 5) Adding an extra feature whenever a word is preceded by "not". For example "not amazing" becomes its own feature. 6) Do we really need to eliminate those funny characters in the data loading phase? Might there be useful signal there? 7) Use tf-idf instead of raw word counts. The tf-idf is calculated as

$$\text{tfidf}(f_i) = \frac{FF_i}{\log(DF_i)} \tag{1}$$

where FF_i is the feature frequency of feature f_i and DF_i is the number of document containing f_i. In this way we increase the weight of rare words. Sometimes this scheme helps, sometimes it makes things worse.

7 Feedback (not graded)

1. Approximately how long did it take to complete this assignment?
2. If you used Python: did you find the Python programming challenging? The mathematical part?
3. Any other feedback?

Acknowledgement: This programming assignment is based partly on assignments developed by David Sontag and David Rosenberg of NYU Courant Institute of Mathematical Sciences.