Homework 4: SVM and Sentiment Analysis

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

1 Introduction

In this assignment, we'll be working with natural language data. In particular, we'll be doing sentiment analysis on movie reviews. 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 new pieces. First, you'll be implementing Pegasos. Pegasos is essentially stochastic subgradient descent for the SVM with a particular schedule for the step-size. Second, because in natural langauge domains we typically have huge feature spaces, we work with sparse representations of feature vectors, where only the non-zero entries are explicitly recorded. This will require coding your gradient and SGD code using hash tables (dictionaries in Python), rather than numpy arrays. We begin with some practice with subgradients and an easy problem that introduces the Perceptron algorithm.

2 [Optional] Calculating Subgradients

Recall that a vector $g \in \mathbf{R}^d$ is a **subgradient** of $f : \mathbf{R}^d \to \mathbf{R}$ at x if for all z,

$$f(z) \geq f(x) + g^T(z - x).$$

As we noted in lecture, there may be 0, 1, or infinitely many subgradients at any point. The **subdifferential** of f at a point x, denoted $\partial f(x)$, is the set of all subgradients of f at x.

Just as there is a calculus for gradients, there is a calculus for subgradients[1]. For our purposes, we can usually get by using the definition of subgradient directly. However, in the first problem below we derive a property that will make our life easier for finding a subgradient of the hinge loss and perceptron loss.

1. [Subgradients for pointwise maximum of functions] Suppose $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ are convex functions, and
$$f(x) = \max_{i=1,\ldots,m} f_i(x).$$

[1] A good reference for subgradients are the course notes on Subgradients by Boyd et al.

Let k be any index for which $f_k(x) = f(x)$, and choose $g \in \partial f_k(x)$. [We are using the fact that a convex function on $\mathbf{R}^d$ has a non-empty subdifferential at all points.] Show that $g \in \partial f(x)$.

2. [Subgradient of hinge loss for linear prediction] Give a subgradient of

$$J(w) = \max\left\{0, 1 - yw^T x\right\}.$$

3 [Optional] 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 with label $y = 1$ are on one side of the hyperplane $\left\{x \mid w^T x = 0\right\}$, and all the $x's$ with label $y = -1$ 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.

Algorithm 1: Perceptron Algorithm

```
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)
```

There is also something called the **perceptron loss,** given by

$$\ell(\hat{y}, y) = \max\left\{0, -\hat{y}y\right\}.$$

In this problem we will see why this loss function has this name.

1. Show that if $\{x \mid w^T x = 0\}$ is a separating hyperplane for a training set $\mathcal{D} = ((x_1, y_1), \ldots, (x_n, y_n))$, then the average perceptron loss on $\mathcal{D}$ is 0.

2. Let $\mathcal{H}$ be the linear hypothesis space consisting of functions $x \mapsto w^T x$. Consider running stochastic subgradient descent (SSGD) to minimize the empirical risk with the perceptron loss. We'll use the version of SSGD in which we cycle through the data points in each epoch. Show that if we use a fixed step size 1, we terminate when our training data are separated, and we make the right choice of subgradient, then we are exactly doing the Perceptron algorithm.

3. Suppose the perceptron algorithm returns w. Show that w is a linear combination of the input points. That is, we can write $w = \sum_{i=1}^{n} \alpha_i x_i$ for some $\alpha_1, \ldots, \alpha_n \in \mathbf{R}$. The x_i for which $\alpha_i \neq 0$ are called support vectors. Give a characterization of points that are support vectors and not support vectors.

4 The Data

We will be using the Polarity Dataset v2.0, constructed by Pang and Lee. It has the full text from 2000 movies reviews: 1000 reviews are classified as "positive" and 1000 as "negative." Our goal is to predict whether a review has positive or negative sentiment from the text of the review. Each review is stored in a separate file: the positive reviews are in a folder called "pos", and the negative reviews are in "neg". We have provided some code in `load.py` to assist with reading these files. You can use the code, or write your own version. The code removes some special symbols from the reviews. Later you can check if this helps or hurts your results.

1. Load all the data and randomly split it into 1500 training examples and 500 validation examples.

5 Sparse Representations

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, we use a sparse representation, in which we only store the counts that are nonzero. The counts are stored in a key/value store (such as a dictionary in Python). For example, "Harry Potter and Harry Potter II" would be represented as the following Python dict: `x={'Harry':2, 'Potter':2, 'and':1, 'II':1}`. We will be using linear classifiers of the form $f(x) = w^T x$, and we can store the w vector in a sparse format as well, such as `w={'minimal':1.3, 'Harry':-1.1, 'viable':-4.2, 'and':2.2, 'product':9.1}`. The inner product between w and x would only involve the features that appear in both x and w, since whatever doesn't appear is assumed to be zero. For this example, the inner product would be x[Harry] * w[Harry] + x[and] * w[and] = 2*(-1.1) + 1*(2.2). To help you along, we've included two functions for working with sparse vectors: 1) a dot product between two vectors represented as dict's and 2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. These

functions are located in `util.py`. It is worth reading the code, even if you intend to implement it yourself. You may get some ideas on how to make things faster.

1. [Optional - Solution provided – you may want to write your own] Write a function that converts an example (e.g. a list of words) into a sparse bag-of-words representation. 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. [We've provided two solutions for this: `load-solution.py`, which uses plain Python, and `libnlp-code/load-libnlp.py` a version based on libnlp provided by Amanda Stent. Directions for installing libnlp can be found here. A note from Amanda:

   > I attach a simple tfidf (plus some more types of feature) libnlp program `load-libnlp.py` , with an input test file and some simple sentiment dictionaries - they could get "real" ones from the web. The features exemplified are: 1) compute tfidf on a corpus; apply it to a new document 2) given a document, extract which tokens match a set of gazetteers 3) given a new document, find word, wordshape, stem and lemma features. Students can ask on the libnlp pchat if questions.]

6 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[2], we're considering the following formulation of the SVM objective function:

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

Note that, for simplicity, we are leaving off the unregularized bias term b. 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 termination condition not met
  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
      $w_{t+1} = (1 - \eta_t \lambda) w_t$

---

1. [Written] Consider the "stochastic" SVM objective function, which is the SVM objective function with a single training point[3]: $J_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y_i w^T x_i\right\}$. The function $J_i(\theta)$

[2] Shalev-Shwartz et al.'s "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM".

[3] Recall that if i is selected uniformly from the set $\{1, \ldots, m\}$, then this stochastic objective function has the same expected value as the full SVM objective function. If this is unfamiliar, review the previous homework solutions.

is not differentiable everywhere. Give an expression for the gradient of $J_i(w)$ where it's defined, and specify where it is not defined.

2. [Written, Optional] Show that a subgradient of $J_i(w)$ is given by

$$g = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i \geq 1. \end{cases}$$

You may use the following facts without proof: 1) If $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ are convex functions and $f = f_1 + \cdots + f_m$, then $\partial f(x) = \partial f_1(x) + \cdots + \partial f_m(x)$. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$. [Hint: Use the rules provided and the calculation in the first problem.]

3. [Written] Show that if your step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from the previous problem is the same as given in the pseudocode.

4. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector w. Note that our Pegasos algorithm starts at $w = 0$. In a sparse representation, this corresponds to an empty dictionary. **Note:** With this problem, you will need to take some care to code things efficiently. In particular, be aware that making copies of the weight dictionary can slow down your code significantly. If you want to make a copy of your weights (e.g. for checking for convergence), make sure you don't do this more than once per epoch.

5. Note that in every step of the Pegasos algorithm, we rescale every entry of w_t by the factor $(1 - \eta_t \lambda)$. Implementing this directly with dictionaries is very slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathbf{R}$ and $W \in \mathbf{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates (i.e. whether or not we have a margin error) start with rescaling w_t, which we can do simply by setting $s_{t+1} = (1 - \eta_t \lambda) s_t$. If the update is $w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$, then **verify that the Pegasos update step is equivalent to**:

$$\begin{aligned} s_{t+1} &= (1 - \eta_t \lambda) s_t \\ W_{t+1} &= W_t + \frac{1}{s_{t+1}} \eta_t y_j x_j. \end{aligned}$$

There is one subtle issue with the approach described above: if we ever have $1 - \eta_t \lambda = 0$, then $s_{t+1} = 0$, and we'll have a divide by 0 in the calculation for W_{t+1}. This only happens when $\eta_t = 1/\lambda$. With our step-size rule of $\eta_t = 1/(\lambda t)$, it happens exactly when $t = 1$. So one approach is to just start at $t = 2$. More generically, note that if $s_{t+1} = 0$, then $w_{t+1} = 0$. Thus an equivalent representation is $s_{t+1} = 1$ and $W = 0$. Thus if we ever get $s_{t+1} = 0$, simply set it back to 1 and reset W_{t+1} to zero, which is an empty dictionary in a sparse representation. **Implement the Pegasos algorithm with the (s, W) representation described above**. [See section 5.1 of Leon Bottou's Stochastic Gradient Tricks for a more generic version of this technique, and many other useful tricks.]

6. Run both implementations of Pegasos to train an SVM on the training data (using the bag-of-words feature representation described above). Make sure your implementations are correct by verifying that the two approaches give essentially the same result. Report on the time taken to run each approach.

7. Write a function that takes a sparse weight vector w and a collection of (x, y) pairs, and returns the percent error when predicting y using $\text{sign}(w^T x)$. In other words, the function reports the 0-1 loss of the linear predictor $x \mapsto w^T x$.

8. Using the bag-of-words feature representation described above, search for the regularization parameter that gives the minimal percent error on your test set. A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. Then, continue to zoom in until you're convinced that additional search will not significantly improve your test performance. Once you have a sense of the general range of regularization parameters that give good results, you do not have to search over orders of magnitude every time you change something (such as adding a new feature).

9. [Optional] Recall that the "score" is the value of the prediction $f(x) = w^T x$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute. Break the predictions into groups based on the score (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage error. You can make a table or graph. Summarize the results. Is there a correlation between higher magnitude scores and accuracy?

10. [Optional] Our objective is not differentiable when $y_i w^T x_i = 1$. Investigate how often and when we have $y_i w^T x_i = 1$ (or perhaps within a small distance of 1 – this is for you to explore) . Describe your findings. If we didn't know about subgradients, one might suggest just skipping the update when $y w^T x_i = 1$. Does this seem reasonable?

7 Error Analysis

The natural language processing domain is particularly nice in that often one can often interpret why a model has performed well or poorly on a specific example, and sometimes it is not very difficult to come up with ideas for new features that might help fix a problem. The first step in this process is to look closely at the errors that our model makes.

1. Choose some examples that the model got wrong. List the features that contributed most heavily to the descision (e.g. rank them by $|w_i x_i|$), along with $x_i, w_i, x w_i$. Do you understand why the model was incorrect? Can you think of a new feature that might be able to fix the issue? Include a short analysis for at least 2 incorrect examples.

8 Features

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. To get ideas for this dataset, you might check the discussion board on this Kaggle competition, which is using a very similar dataset. There are also a very large number of academic research papers on sentiment analysis that you can look at for ideas.

1. [Optional] Based on your error analysis, or on some idea you have, construct a new feature (or group of features) that you hope will improve your test performance. Describe the features and what kind of improvement they give. At this point, it's important to consider the standard errors $\sqrt{p(1-p)/n}$ (where p is the proportion of the test examples you got correct, and n is the size of the test set) on your performance estimates, to know whether the improvement is statistically significant.

2. [Optional] Try to get the best performance possible by generating lots of new features, changing the pre-processing, or any other method you want, 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 review? 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. You could try using both! [Extra credit points will be awarded in proportion to how much improvement you achieve.]