Lab 3 - Support Vector Machines

Kyle Swanson

January 10, 2018

0 Introduction

This lab will be a short extension of Lab 2, in which you implemented the perceptron algorithm. In this lab, we will be working with support vector machines (SVMs). Unlike perceptrons, which only care about maximizing training accuracy, SVMs try to simultaneously maximize training accuracy and maximize the margin of the decision boundary, which allows SVMs to generalize better than simple perceptrons.

This lab will consist of three parts:

1. Implement the pegasos algorithm for SVM and experiment with different hyperparameters.
2. Use scikit-learn to build and train an SVM and examine the SVM's decision boundary, margin, and support vectors.
3. Optimize scikit-learn's SVM using a technique called grid search.

1 Pegasos Algorithm

First we will implement the pegasos algorithm, which is an *online* algorithm for learning a support vector machine. The algorithm is called an online algorithm because it optimizes the decision boundary and margins for each data point one at a time rather than optimizing for all the data points all at once. (Optimizing for all the data points at once would be called an *offline* algorithm and it is much harder to do).

Below is the pegasos algorithm:

Algorithm 1 Pegasos Algorithm

```
procedure PEGASOS
    θ = 0⃗ (vector)
    for t = 1, ..., T do
        for i = 1, ..., n do
            if y^(i)(θ · x^(i)) ≤ 1 then
                θ = θ − η ∗ (−y^(i)x^(i) + λθ)
            else
                θ = θ − η ∗ (λθ)
    return θ
```

1.1 Implementation

In this part, you need to go into `lab3.py` and implement the function `pegasos`. This function takes as input a feature matrix containing all of the data points, a labels array containing a label for each of the data points, and three hyperparameters T, η (eta), and λ (lam). This function should run the pegasos algorithm on the data and return θ and θ_0.

1.1.1 Note on θ and θ_0

Your function implementation needs to include a couple extra steps in addition to the pegasos algorithm as written above. If you notice in the algorithm above, you only see θ, not θ_0. So in order to include a θ_0 parameter, we have to do a trick that we talked about in lecture.

In lecture we said that the equation of a linear decision boundary is

$$\theta \cdot x + \theta_0 = 0$$

This can be rewritten as

$$\theta \cdot x + \theta_0 * 1 = 0$$

$$\begin{pmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \theta_0 * 1 = 0$$

$$\begin{pmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \\ \theta_0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \\ 1 \end{pmatrix} = 0$$

$$\theta' \cdot x' = 0$$

where $\theta' = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \\ \theta_0 \end{pmatrix}$ and $x' = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \\ 1 \end{pmatrix}$

So in order to include a θ_0 parameter in the pegasos algorithm, we have to extend θ and x to θ' and x', run the pegasos algorithm using θ' and x', and then extract θ and θ_0 from θ'. More specifically, this requires appending a 1 to every row of the feature matrix (since each row is a feature vector x) and then using a θ which is $d + 1$ dimensions instead of d dimensions.

This part is pretty subtle and confusing, so if you don't understand what to do, please ask me for help.

1.1.2 Running the code

Once you have completed the implementation, uncomment Part 1.1 in `main.py` and run `python main.py`. First you will see a plot of just the data points, then you will see a plot of the data points along with the decision boundary and margins generated by the pegasos algorithm. Check that the decision boundary and margins look reasonable given the data.

1.2 Modifying hyperparameters

While the perceptron algorithm only had one hyperparameter (T), the pegasos algorithm has three (T, eta, and lambda). Each one has a different affect on the resulting decision boundary and margins depending on whether that hyperparameter is large or small. In this section, we're going to try modifying each hyperparameter one at a time to see what sort of affect it has.

Go into `main.py` and uncomment Part 1.2. Most of the code is already written for you. Your job is to modify the three lists `Ts`, `etas`, and `lams` by appending various values of T, eta, and lambda. You should try some which are larger than the value currently in the list and some which are smaller. After adding a few values of each, run `main.py` and examine the graphs produced for each of the three hyperparameters. What happens when each hyperparameter gets larger or smaller? Why do you think the hyperparameters have that effect? Check with me to see if your intuition is correct.

2 Support Vector Machines

In the first part of the lab we developed an online SVM algorithm that optimized the objective function data point by data point. Programming an offline SVM algorithm which produces the optimal solution across all points is much more difficult. Luckily it's already been done for us! `scikit-learn` provides an implementation of SVM which is extremely easy to use.

Note: For the rest of the lab, no coding is necessary on your part, but try to make sense of the code I've written. `scikit-learn` is an very powerful Python library for machine learning and it's worth knowing how to use some of its functions.

2.1 SVM for generated data

To see `scikit-learn`'s SVM in action, go into `main.py`, uncomment Part 2.1 and run the code. You should see a decision boundary and margins along with two support vectors which lie on the margins. Unlike the solution produced by the pegasos algorithm, which was good but not perfect, the solution produced by the `scikit-learn` SVM is optimal.

2.2 SVM for sentiment analysis

Next, we're going to use `scikit-learn` to classify the Amazon food reviews from Lab 2. `scikit-learn` provides a variety of tools for turning raw data into vectors, so we're going to use one called `CountVectorizer`, which builds a feature vector based on how many times certain words appear in the text of the review.

Uncomment and take a look at the code in Part 2.2. Here we construct a "pipeline", which consists of the CountVectorizer followed by an SVM classifier. This pipeline will take the raw text of the reviews, convert them into feature vectors, and then train an SVM classifier using those features. Run the code and examine the output. Compare the accuracies produced by the SVM to the accuracies produced by the perceptron algorithm from Lab 2.

It turns out that even though SVMs are usually more powerful than perceptrons, the perceptron actually performs better for this data set. However, the SVM is still a very good classifier for this data set.

3 Grid Search

Just like the pegasos algorithm, `scikit-learn`'s SVM has hyperparameters which need to be tuned in order to optimize the classifier. Tuning hyperparameters is done through a process called *grid search* in which different combinations of hyperparameters are tried to determine which hyperparameters produce the highest validation accuracy.

Fortunately, `scikit-learn` provides a tool called `GridSearchCV` which makes performing a grid search over hyperparameters extremely easy. Uncomment and examine the code in Part 3. `param_grid` contains all the values of the hyperparameters that we want to tune (in this case there are two SVM hyperparameters called "C" and "kernel" - take a look at the `scikit-learn` SVM specification for details: `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC`).

Run `main.py` and examine the results. It might take a minute or two to run, but once it's complete, you'll see the results printed to the screen. Note how it's now easy to see which combinations of hyperparameters produce the greatest accuracy. We can then select these hyperparameter values to build the best model.