Machine Learning Lecture 6 Note

Compiled by Abhi Ashutosh, Daniel Chen, and Yijun Xiao

February 16, 2016

1 Pegasos Algorithm

The Pegasos Algorithm looks very similar to the Perceptron Algorithm. In fact, just by changing a few lines of code in our Perceptron Algorithms, we can get the Pegasos Algorithm.

Algorithm 1: Perceptron to Pegasos

```
initialize $\mathbf{w}_1 = 0$, t = 0;
for $iter = 1, 2, ..., 20$ do
    for $j = 1, 2, ..., |data|$ do
        t = t + 1; $\eta_t = \frac{1}{t\lambda}$;
        if $y_j(\mathbf{w}_t^T \mathbf{x}_j) < 1$ then
            $\mathbf{w}_{t+1} = (1 - \eta_t \lambda)\mathbf{w}_t + \eta_t y_j \mathbf{x}_j$;
        else
            $\mathbf{w}_{t+1} = (1 - \eta_t \lambda)\mathbf{w}_t$;
        end
    end
end
```

Side note: We can optimize both the Pegasos and Perceptron Algorithm by using sparse vectors in the case of document classification because most entries in the feature vector $\mathbf{x}$ will be zeros.

As we discussed in the lecture, the original Pegasos algorithm randomly chooses one data point at each iteration instead of going through each data point in order as shown in Algorithm 1. Pegasos algorithm is an application of the stochastic sub-gradient descent method.

2 Using Pegasos to Solve Other SVM Objectives

2.1 Imbalanced data set

Sometimes it may be hard to classify an imbalanced data set where the classification categories are not equally represented. In this case, we want to weigh each data point differently by placing more weights on the data points in the underrepresented categories. We can do this very easily by changing our optimization problem to

$$\min_{\mathbf{w}} \|\mathbf{w}\|_2^2 + \frac{CN}{2N_+} \sum_{j:y_j=+1} \xi_j + \frac{CN}{2N_-} \sum_{j:y_j=-1} \xi_j$$

where N_+, N_- are the number of positive data points and negative data points respectively. ξ_j's are the slack variables.

An intuitive way to think about this is if we want to build a classifier that classifies whether a point is blue or red. If in our data set we only have 1 data point that's labelled as red and 10 data points labelled as blue, then using the modified objective function is equivalent to duplicating the 1 red point 10 times without explicitly creating more training data.

2.2 Transfer learning

Suppose we want to build a personalized spam classifier for Professor David Sontag. However, Professor David only has few of his emails labelled. Professor Rob, on the other hand, has labelled all of the emails he has ever received as spam or not spam and trained an accurate spam filter on them. Since Professor David and Professor Rob are both Computer Science Professors and run a lab together, we hope that they probably share similar standards for spams/non-spams. In this case, a spam classifier built for Professor Rob should work well to a certain extent for Professor David as well. What should the SVM objective be? (Class ideas: average the weight vectors of both Professors; combine David and Rob's data and put more weights on David's data.)

One solution is to solve the following modified optimization problem,

$$\min_{\mathbf{w}_d, b_d} \frac{C}{|D_d|} \sum_{\mathbf{x}, y \in D_d} \max(0, 1 - y(\mathbf{w}_d^T \mathbf{x} + b_d)) + \frac{1}{2}\|\mathbf{w}_d - \mathbf{w}_r\|^2$$

The idea here is we assume that the weight for Rob is going to be very close to that for David. We then try to penalize the distance between the two. C here can be interpreted as how confident we are that the weights for Rob will be similar to the weights for David. If we are very confident, a low C, then we will really try to minimize the distance between the two weight vectors. If we are not confident, large C, then we are more focused on David's labelled data.

2.3 Multiclass classification

If we want to extend these ideas further to multi-class classification, we have a number of options. The simplest is called a One-vs-all Classifier in which we learn n classifiers, one for each of the n classes. We could run into issues if we want to classify a point that fell in between our classifiers as we would need to decide in which class it belongs. We can predict the most probable class using the formula

$$\hat{y} = \underset{k}{\operatorname{argmax}}\, \mathbf{w}_k^T \mathbf{x} + b_k$$

Another solution is called Multiclass SVM. Here, we put soft restrictions on predicting correct labels for the training data using:

$$\mathbf{w}^{(y_j)T}\mathbf{x}_j + b^{(y_j)} \geq \mathbf{w}^{(y')T}\mathbf{x}_j + b^{(y')} + 1 - \xi_j, \forall y' \neq y_j, \xi_j \geq 0, \forall j$$

Notice that we have one slack variable ξ_j per data point and one set of weights $\mathbf{w}^{(k)}, b^{(k)}$ for each class k. We could derive a similar Pegasos Algorithm for a multiclass classifier.

3 Kernel Trick

What if the data is not linearly separable? We can create a mapping $\phi(\mathbf{x})$ that takes our feature vector $\mathbf{x}$ and converts it into a higher dimensional space. Creating a linear classification in this higher dimension and projecting that onto our original feature space will give us a squiggly line classifier.

Kernel tricks allow us to perform the aforementioned classification with little extra cost. For Pegasos algorithm, we can do this by keeping track of just a single variable per data point, α_i, and calculating vector $\mathbf{w}$ when required.

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

Let's now derive the updating rule for such α_i's. Notice in Algorithm 1, the update rule at each iteration is

$$\mathbf{w}_{t+1} = (1 - \eta_t \lambda)\mathbf{w}_t + \mathbb{1}[y_j \mathbf{w}_t^T \mathbf{x}_j < 1] \cdot \eta_t y_j \mathbf{x}_j$$

where $\mathbb{1}[\textit{condition}]$ is the indicator function. Now instead of $\mathbf{x}_j, y_j$, let us use $\mathbf{x}^{(t)}, y^{(t)}$ to denote the data point we randomly selected at iteration t. Substitute η_t, we have

$$\mathbf{w}_{t+1} = \left(1 - \frac{1}{t}\right)\mathbf{w}_t + \frac{1}{\lambda t}\mathbb{1}[y^{(t)}\mathbf{w}_t^T \mathbf{x}^{(t)} < 1] \cdot y^{(t)}\mathbf{x}^{(t)}$$

Multiplying both sides with t, rearranging,

$$t\mathbf{w}_{t+1} - (t-1)\mathbf{w}_t = \frac{1}{\lambda}\mathbb{1}[y^{(t)}\mathbf{w}_t^T \mathbf{x}^{(t)} < 1] \cdot y^{(t)}\mathbf{x}^{(t)}$$

As the above equation holds for any t, we have the following t equations

$$\begin{cases} t\mathbf{w}_{t+1} - (t-1)\mathbf{w}_t & = \frac{1}{\lambda}\mathbb{1}[y^{(t)}\mathbf{w}_t^T \mathbf{x}^{(t)} < 1] \cdot y^{(t)}\mathbf{x}^{(t)} \\ (t-1)\mathbf{w}_t - (t-2)\mathbf{w}_{t-1} & = \frac{1}{\lambda}\mathbb{1}[y^{(t-1)}\mathbf{w}_{t-1}^T \mathbf{x}^{(t-1)} < 1] \cdot y^{(t-1)}\mathbf{x}^{(t-1)} \\ & \cdots \\ \mathbf{w}_2 & = \frac{1}{\lambda}\mathbb{1}[y^{(1)}\mathbf{w}_1^T \mathbf{x}^{(1)} < 1] \cdot y^{(1)}\mathbf{x}^{(1)} \end{cases}$$

Summing over the above t equations and dividing both sides by t, we can have

$$\mathbf{w}_{t+1} = \frac{1}{\lambda t} \sum_{k=1}^{t} \mathbb{1}[y^{(k)} \mathbf{w}_k^T \mathbf{x}^{(k)} < 1] \cdot y^{(k)} \mathbf{x}^{(k)}$$

written in the form of summation over i:

$$\mathbf{w}_{t+1} = \sum_i \left(\frac{1}{\lambda t} \sum_{k=1}^{t} \mathbb{1}[y^{(k)} \mathbf{w}_k^T \mathbf{x}^{(k)} < 1] \cdot \mathbb{1}[(\mathbf{x}_i, y_i) = (\mathbf{x}^{(k)}, y^{(k)})] \right) y_i \mathbf{x}_i$$

All the stuff in the huge parenthesis corresponds to α_i we defined earlier. $\lambda t \alpha_i^{(t+1)}$ counts the number of times data point i appears before iteration t and satisfies $y_i \mathbf{w}_k^T \mathbf{x}_i < 1$. This implies a simple updating rule for $\lambda t \alpha_i^{(t+1)}$:

$$\lambda t \alpha_i^{(t+1)} = \lambda(t-1)\alpha_i^{(t)} + \mathbb{1}[(\mathbf{x}_i, y_i) = (\mathbf{x}^{(t)}, y^{(t)})] \cdot \mathbb{1}[y_i \mathbf{w}_t^T \mathbf{x}_i < 1]$$

i.e. suppose we draw data point $(\mathbf{x}_i, y_i)$ at iteration t, we increment $\lambda t \alpha_i$ by 1 iff $y_i \mathbf{w}_t^T \mathbf{x}_i < 1$. The algorithm is shown in Algorithm 2. To simplify the notations, we denote $\beta_i^{(t)} = \lambda(t-1)\alpha_i^{(t)}$.

Algorithm 2: Kernelized Pegasos

```
1 initialize $\beta^{(1)} = 0$;
2 for $t = 1, 2, ..., T$ do
3     randomly choose $(\mathbf{x}^{(t)}, y^{(t)}) = (\mathbf{x}_j, y_j)$ from training data
4     if $y_j \frac{1}{\lambda(t-1)} \sum_i \beta_i^{(t)} y_i \mathbf{x}_i^T \mathbf{x}_j < 1$ then
5         $\beta_j^{(t+1)} = \beta_j^{(t)} + 1$;
6     else
7         $\beta_j^{(t+1)} = \beta_j^{(t)}$;
8     end
9 end
```

After convergence, we can get back α_i's using $\alpha_i = \frac{1}{\lambda T}\beta_i^{(T+1)}$. In testing time, predictions can be made with

$$\hat{y} = \text{sign}\left(\sum_i \alpha_i y_i \mathbf{x}_i^T \mathbf{x} \right)$$

Now suppose we want to use more complex features $\phi(\mathbf{x})$ which can be obtained by transforming the original features $\mathbf{x}$ to a higher dimensional space, all we need to do is to substitute $\mathbf{x}_i^T \mathbf{x}_j$ in both training and testing with $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$.

Further notice that $\phi(\mathbf{x})$ always appears in the form of dot products. Which indicates we do not necessarily need to explicitly compute it as long as we have a formula to calculate the dot products. This is where kernels come into use. Instead of defining the function ϕ to do the projection, we directly define a kernel function K to calculate the dot product of the projected features.

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

We can create different kernel functions $K(\mathbf{x}_i, \mathbf{x}_j)$ as long as those functions are based on dot products. We can also create new valid kernel functions using other valid kernel functions following certain rules. Examples of popular kernel functions include Polynomial Kernels, Gaussian Kernels, and many more.

References

Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, Andrew Cotter. Extended version: Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. *Mathematical Programming, Series B, 127(1):3-30*, 2011