CS 475/675 Machine Learning: Homework 2
Supervised Classifiers 2
Programming Assignment
Due: Friday, March 6, 2020, 11:59 pm
50 Points Total Version 1.0

Before you begin, please read the Homework 2 Introduction. The introduction has information on how to use the Python codebase, the data, and how to submit your assignment.

This assignment is all about SVMs. From lecture, you know that SVMs present an optimization problem with both primal and dual formulations. You will be implementing a standard and kernelized linear SVM using sub-gradient descent of the primal problem.

We have provided Python code to serve as a testbed for your algorithms. We'll use the same coding framework that we used in homework 1. You will fill in the details. Search for comments that begin with `TODO`; these sections need to be written. Your code for this assignment will all be within the `models.py` file; do not modify any of the other files.

1 Support Vector Machines (25 points)

A Support Vector Machine constructs a hyperplane in high dimensional space, which separates training points of different classes while keeping a large margin with regards to the training points closest to the hyperplane. SVMs are typically formulated as a constrained quadratic programming problem. We can also reformulate it as an equivalent unconstrained problem of empirical loss plus a regularization term. Formally, given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i \in \mathbb{R}^M$ and $y_i \in \{+1, -1\}$, we want to find the minimizer of the problem:

$$\min_{\mathbf{w}} \lambda \frac{1}{2}||\mathbf{w}||^2 + \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{w}; (\mathbf{x}_i, y_i)) \tag{1}$$

where λ is the regularization parameter,

$$l(\mathbf{w}; (\mathbf{x}, y)) = \max\{0, 1 - y\langle \mathbf{w}, \mathbf{x}\rangle\} \tag{2}$$

and $\langle \cdot, \cdot \rangle$ is the inner product operator. As you might have noticed, we omit the parameter for the bias term throughout this homework (i.e., the hyperplane is always across the origin).

In SVM, we want $y \in \{+1, -1\}$, but the class labels in our data files are 0/1 valued. To resolve this inconsistency, convert class label 0 to -1 before training, just like you did in homework 1.

1.1 Pegasos

You will implement a version of SVM called Pegasos. Pegasos is a simple but effective stochastic gradient descent algorithm to solve a Support Vector Machine for binary classification problems. The approach is called *Primal Estimated sub-GrAdient SOlver for SVM* (Pegasos).[1] The number of iterations required by Pegasos to obtain a solution of accuracy ϵ is $O(1/\epsilon)$, while previous stochastic gradient descent methods require $O(1/\epsilon^2)$.

1.2 Training

Pegasos performs stochastic gradient descent on the primal objective Eq. 1. The learning rate decreases with each iteration to guarantee convergence. For further details, see Section 1.5.

Many descriptions of SGD call for shuffling your data before learning. This is a good idea to break any dependence in the order of your data. In order to achieve consistent results for everyone, we are requiring that you **do not shuffle your data**. When you are training, you will go through your data in the order it appeared in the data file.

On each time step Pegasos operates as follows. First, we initialize $\mathbf{w}_0 = 0$. We then make one slight modification to the Pegasos training algorithm. Typically, at each time step t of the algorithm (starting from $t = 1$), we would choose a random training example $(\mathbf{x}_i^{(t)}, y_i^{(t)})$ by picking an index $i^{(t)} \in \{1, \dots, m\}$ uniformly at random. **For your implementation, you should not select the training example randomly, and instead iterate through the examples in order.** For further details, see Section 1.6. We then replace the objective in Eq. 1 with an approximation based on the training example $(\mathbf{x}_i^{(t)}, y_i^{(t)})$, yielding:

$$f(\mathbf{w}; i^{(t)}) = \lambda \frac{1}{2} ||\mathbf{w}||^2 + l(\mathbf{w}; (\mathbf{x}_i^{(t)}, y_i^{(t)})) \tag{3}$$

We consider the sub-gradient of the above approximate objective, given by:

$$\frac{\partial f(\mathbf{w}; i^{(t)})}{\partial \mathbf{w}^{(t)}} = \lambda \mathbf{w}^{(t)} - \mathbb{1}[y_i^{(t)} \langle \mathbf{w}^{(t)}, \mathbf{x}_i^{(t)} \rangle < 1] y_i^{(t)} \mathbf{x}_i^{(t)} \tag{4}$$

where $\mathbb{1}[\cdot]$ is the indicator function which takes a value of 1 if its argument is true, and 0 otherwise. We then update $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)} \frac{\partial f(\mathbf{w}; i^{(t)})}{\partial \mathbf{w}^{(t)}}$ using a step size of $\eta^{(t)} = 1/(\lambda t)$. Note that this update can be written as:

$$\mathbf{w}^{(t+1)} \leftarrow (1 - \frac{1}{t}) \mathbf{w}^{(t)} + \frac{1}{\lambda t} \mathbb{1}[y_i^{(t)} \langle \mathbf{w}^{(t)}, \mathbf{x}_i^{(t)} \rangle < 1] y_i^{(t)} \mathbf{x}_i^{(t)} \tag{5}$$

After a predetermined total number of iterations T, which equals the total number of examples times the total number of epochs, we output the last iterate $\mathbf{w}_T$.

To be clear, a single iteration t should be a single example. So, you should be updating the weights after seeing a single example, not in batch mode.

[1] Shalev-Shwartz, S., Singer, Y., Srebro, N., Cotter, A. (2011). Pegasos: Primal estimated sub-gradient solver for svm. Mathematical programming, 127(1), 3-30.

You will note that the above update rule changes $\mathbf{w}$ even for cases where the value in $\mathbf{x}_i$ is 0. **This is a non-sparse update: every feature is updated every time.** While there are sparse versions of Pegasos, for this homework you should implement the non-sparse solution. This means that every time you update, every parameter must be updated. **This will make training slower on larger feature sets (such as NLP) but it should still be reasonable. We suggest you focus testing on the smaller and thus faster datasets.**

1.3 Gradient Projection

So far we have updated our weights using stochastic gradient descent, with the slight modification that we iterate through the examples in order rather than sample uniformly at random. We can further regularize our learned parameters by limiting the set of candidate soliutions to a ball[2] of radius $\frac{1}{\sqrt{\lambda}}$. Limiting the set of possible solutions prevents our model from overfitting. We can enforce this property by projecting $\mathbf{w}^{(t)}$ onto this sphere after each iteration t, by performing the update:

$$\mathbf{w}^{(t+1)} \leftarrow \min \left\{ 1, \frac{1/\sqrt{\lambda}}{||\mathbf{w}^{(t+1)}||} \right\} \mathbf{w}^{(t+1)} \tag{6}$$

This is an optional step, which can be set using the command line option `--pegasos-projection`.

1.4 Prediction

During prediction, a new instance $\mathbf{x}$ is predicted by the following rule:

$\hat{y}_{new} = 1$ if $\langle \mathbf{w}, \mathbf{x} \rangle \geq 0$
$\hat{y}_{new} = 0$ otherwise

1.5 Learning Rate

We adopt a simple strategy to decreasing the learning rate: we make the learning rate a function of the time steps. Specifically, your learning rate should be $\eta^{(t)} = 1/(\lambda t)$.

1.6 Sampling Instances

Pegasos relies on sampling examples for each time step. **For simplicity, we will NOT randomly sample instances. Instead, on round t you should update using the tth example.** An iteration involves a single pass in order through all the provided instances. You will make several passes through the data based on `--online-training-iterations`.

1.7 Regularization Parameter

Pegasos uses a regularization parameter λ to adjust the relative strength of regularization. Your default value for λ should be 10^{-4}. This parameter can be adjusted via the command line argument `--pegasos-lambda`.

[2] A ball is the space bounded by a sphere, which may or may not include its boundary points.

1.8 Offset Feature

None of the math above mentions an offset feature (bias feature) $\mathbf{w}_0$, that corresponds to a $x_{i,0}$ that is always 1. It turns out that we don't need this if our data is centered. By centered we mean that $E[y] = 0$. For simplicity, assume that the data used in this assignment is centered (even though this may not be true). Do not include another feature that is always 1 (x_0) or weight ($\mathbf{w}_0$) for it.

1.9 Convergence

In practice, SGD optimization requires the program to determine when it has converged. Ideally, a maximized function has a gradient value of 0, but due to issues related to your step size, random noise, and machine precision, your gradient will likely never be exactly zero. Common practice is to check that the L_p norm of the gradient is less than some δ, for some p. For the sake of simplicity and consistent results, we will not do this in this assignment. Instead, your program should take a parameter **–online-training-iterations** which is *exactly* how many iterations you should run (not an upper bound). An iteration is a single pass over every training example. **Note that the "t" in Section 1.1 indexes a time step, which is different from the "iterations" defined here.** The default of **–online-training-iterations** should be 5. Since you added this argument in the previous assignment, no further changes should be needed.

2 Kernel SVM (25 points)

One of the primary benefits to choosing SVMs as your supervised classifier is that they can be used with kernels instead of directly accessing the feature vectors X. Consider a positive-definite real-valued kernel function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Then, for all $x, x' \in \mathcal{X}$, the kernel function $K(x, x')$ can be expressed as the inner product in another space $\mathcal{V}$. This stems from the Representer Theorem[3], which implies that the optimal solution to our primal SVM objective function in (1) can be expressed as a linear combination of the training instances.

This makes it possible to train and use the SVM *without* direct access to the training instances, and instead with *only* access to their inner products specified by the kernel operator $K(x, x')$. Then, instead of considering predictors which are linear functions of the training instances X themselves, we consider predictors which are linear functions of some implicit mapping $\phi : \mathcal{X} \to \mathcal{V}$ of the training data. Formally, given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i \in \mathbb{R}^M$ and $y_i \in \{+1, -1\}$, training inolves finding the minimizer of the problem:

$$\min_{\mathbf{w}} \lambda \frac{1}{2} ||\mathbf{w}||^2 + \frac{1}{m} \sum_{(\mathbf{x}_i, y_i) \in (\mathbf{X}, \mathbf{Y})}^{N} l(\mathbf{w}; (\phi(\mathbf{x}_i), y_i)) \tag{7}$$

where λ is the regularization parameter,

$$l(\mathbf{w}; (\phi(\mathbf{x}), y)) = \max\{0, 1 - y\langle \mathbf{w}, \phi(\mathbf{x})\rangle\} \tag{8}$$

[3] Schölkopf, Bernhard; Herbrich, Ralf; Smola, Alex J. (2001). A Generalized Representer Theorem. Computational Learning Theory. Lecture Notes in Computer Science. 2111. pp. 416–426.

and $\langle \cdot, \cdot \rangle$ is the inner product operator. Note that the feature mapping $\phi(\cdot)$ is never explicitly specified, but instead defined through the kernel operator $K(x, x') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$.

To solve the optimization problem defined in (7), typically we switch to the dual formulation. To do so, we rewrite the primal problem as a function of α and then take gradients with respect to α. The Representer theorem guarantees that the optimal solution to Eq. 7 is spanned by the training instances and is of the form $\mathbf{w} = \sum_{i=1}^{m} \alpha_i \phi(\mathbf{x}_i)$. The training objective can then be written in terms of α variables and kernel evaluations:

$$\min_{\alpha} \frac{\lambda}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) + \frac{1}{m} \sum_{i=1}^{m} \max \left\{ 0, 1 - y_i \sum_{j=1}^{m} \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) \right\} \tag{9}$$

Note, this is *not* the approach we will take here.

2.1 Kernelized Pegasos

What happens when our data is not linearly separable? Remember how the finance and hard datasets from homework 1 received such lower accuracy than all the others? To address this issue, we can create a mapping $\phi(\mathbf{x})$ that takes our feature vector $\mathbf{x}$ into a higher dimensional space. Then, when projected back onto our original feature space, a linear classifer in this higher dimensional space will be like a "squiggly" line classifier, allowing us to capture some of these nonlinearities.

We can extend our Pegasos algorithm from the first part of this assignment to use kernels while still minimizing the primal problem. The number of iterations required by Kernelized Pegasos to obtain a solution of accuracy ϵ is $O(1/(\lambda\epsilon))$[4], while our standard Pegasos was $O(1/\epsilon)$. You will implement a kernelized version of Pagasos, which uses only kernel evaluations and doesn't explicitly access feature vectors $\phi(\mathbf{x})$ or weight vector $\mathbf{w}$. Fortunately, you'll be able to reuse much of your standard Pegasos SVM, without gradient-projection. The key to performing the kernel trick is substitute dual variables into the objective and to keep track of a variable α_i for each data point. Our α_i's will then indirectly update our weight matrix for us:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \tag{10}$$

So what are these α_i values? Let's derive the update rules for α_i and then see how these relate to our weight updates. Like we did before, we can replace the objective in Eq. 7 with an approximate based indirectly on $(\phi(\mathbf{x})_i^{(t)}, y_i^{(t)})$, yielding:

$$f(\mathbf{w}; i^{(t)}) = \lambda \frac{1}{2} ||\mathbf{w}||^2 + l(\mathbf{w}; (\phi(\mathbf{x}_i^{(t)}), y_i^{(t)})) \tag{11}$$

We consider the sub-gradient of the above approximate objective, given by:

$$\frac{\partial f(\mathbf{w}; i^{(t)})}{\partial \mathbf{w}^{(t)}} = \lambda \mathbf{w}^{(t)} - \mathbb{1}[y_i^{(t)} \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}_i^{(t)}) \rangle < 1] y_i^{(t)} \phi(\mathbf{x}_i^{(t)}) \tag{12}$$

[4]The number of iterations required does not depend on the number of training examples, however your runtime does. This is beause we are checking for non-zero loss at each iteration t, so we may require as much as $\min(t, m)$ kernel evaluations. This brings the overall runtime to $O(m/(\lambda\epsilon))$.

where $\mathbb{1}[\cdot]$ is the indicator function which takes a value of 1 if its argument is true, and 0 otherwise. We then update $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)} \frac{\partial f(\mathbf{w};i^{(t)})}{\partial \mathbf{w}^{(t)}}$ using a step size of $\eta^{(t)} = 1/(\lambda t)$. Note that this update can be written as:

$$\mathbf{w}^{(t+1)} \leftarrow (1 - \frac{1}{t})\mathbf{w}^{(t)} + \frac{1}{\lambda t}\mathbb{1}[y_i^{(t)}\langle \mathbf{w}^{(t)}, \phi(\mathbf{x}_i^{(t)})\rangle < 1]y_i^{(t)}\phi(\mathbf{x}_i^{(t)}) \tag{13}$$

Multiplying both sides by t and then rearranging, we get:

$$t\mathbf{w}^{(t+1)} - (t-1)\mathbf{w}^{(t)} = \frac{1}{\lambda}\mathbb{1}[y^{(t)}\langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)})\rangle < 1]y^{(t)}\phi(\mathbf{x}^{(t)}) \tag{14}$$

Observe that Eq. 14 holds for any value t, which gives us a system of t equations:

$$\begin{cases} t\mathbf{w}^{(t+1)} - (t-1)\mathbf{w}^{(t)} & = \frac{1}{\lambda}\mathbb{1}[y^{(t)}\langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)})\rangle < 1]y^{(t)}\phi(\mathbf{x}^{(t)}) \\ (t-1)\mathbf{w}^{(t)} - (t-2)\mathbf{w}^{(t-1)} & = \frac{1}{\lambda}\mathbb{1}[y^{(t-1)}\langle \mathbf{w}^{(t-1)}, \phi(\mathbf{x}^{(t-1)})\rangle < 1]y^{(t-1)}\phi(\mathbf{x}^{(t-1)}) \\ & \dots \\ \mathbf{w}_2 & = \frac{1}{\lambda}\mathbb{1}[y_1\langle \mathbf{w}_1, \phi(\mathbf{x}_1)\rangle < 1]y_1\phi(\mathbf{x}_1) \end{cases}$$

Now, by summing over these t equations and then dividing both sides by t, we get

$$\mathbf{w}^{(t+1)} = \frac{1}{\lambda t}\sum_{k=1}^{t} \mathbb{1}[y_k\langle \mathbf{w}^{(k)}, \phi(\mathbf{x}_k)\rangle < 1]y_k\phi(\mathbf{x}_k) \tag{15}$$

To match our desired form in Eq. 10, we can rewrite this as in terms of a summation over i:

$$\mathbf{w}^{(t+1)} = \sum_i \underbrace{\left(\frac{1}{\lambda t}\sum_{k=1}^{t} \mathbb{1}[y_k\langle \mathbf{w}^{(k)}, \phi(\mathbf{x}_k)\rangle < 1] \cdot \mathbb{1}[(\phi(\mathbf{x}_i), y_i) = (\phi(\mathbf{x}^{(k)}), y^{(k)})]\right)}_{\alpha_i} y_i\phi(\mathbf{x}_i) \tag{16}$$

Then, for every t, $\lambda t\alpha_i^{(t+1)}$ counts how many times example i appears before iteration t and also satisfies $y_i\langle \mathbf{w}_i, \phi(\mathbf{x}_i)\rangle < 1$. This implies a simple update rule for α_i:

$$\lambda t\alpha_i^{(t+1)} \leftarrow \lambda(t-1)\alpha_i^{(t)} + \mathbb{1}[(\phi(\mathbf{x}_i), y_i) = (\phi(\mathbf{x}^{(t)}), y^{(t)})] \cdot \mathbb{1}[y_k\langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)})\rangle < 1] \tag{17}$$

For simplicity, let's denote $\beta_i = \lambda(t-1)\alpha_i^{(t)}$. Then, we can simplify this update rule to:

$$\beta_i^{(t+1)} \leftarrow \frac{t-1}{t}\left[\beta_i^{(t)} + \mathbb{1}[(\phi(\mathbf{x}_i), y_i) = (\phi(\mathbf{x}^{(t)}), y^{(t)})] \cdot \mathbb{1}[y_k\langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)})\rangle < 1]\right] \tag{18}$$

Intuitively, this update equation tells us that if we draw a data point $(\mathbf{x}_i, y_i)$ at iteration t, then we increment α_i by 1 if and only if $y_k\langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)})\rangle < 1$. Instead of keeping in memory the weight vector $\mathbf{w}^{(t+1)}$, we will represent $\mathbf{w}^{(t+1)}$ using $\alpha^{(t+1)}$ by:

$$w^{(t+1)} = \sum_{i=1}^{m} \alpha_i^{(t+1)} y_i\phi(\mathbf{x}_i) \tag{19}$$

And we have derived the updates to our weight matrix of the form desired in Eq. 10. Note, only one element of α is changed each iteration. Notice, that we haven't defined explicitly what $\phi(\cdot)$ is. This is because your algorithm should refer only to the kernel evaluations and not an explicit feature mapping.

2.2 Kernel Function

During your updates, you will obtaining the feature maps $\phi(\cdot)$ by evaluating a polynomial kernel function. That is,

$$K(x, x') = (x^T x' + c)^d \tag{20}$$

where d is the polynomial degree and $c \geq 0$ is a free paramter. You will pre-compute the kernel matrix and then index from it when you need to evaluate the kernel function. You can specify the polynomial degree using the command line argument `--kernel-degree`.

2.3 Prediction

We can recover our α_i values from β_i by:

$$\alpha_i = \frac{1}{\lambda T} \beta_i^{T+1} \tag{21}$$

where T is the total number of iterations taken during training. Then, during prediction, we use the following formula:

$$\hat{y}_{new} = sign\left(\sum_i \alpha_i y_i \phi(\mathbf{x}_i)\right) \tag{22}$$