CS 5350/6350: Machine Learning Fall 2022

Homework 4

Handed out: 8 Nov, 2022
Due date: 11:59pm, 22 Nov, 2022

1 Paper Problems [40 points + 10 bonus]

1. [9 points] The learning of soft SVMs is formulated as the following optimization problem,

$$\begin{aligned} \min_{\mathbf{w},b,\{\xi_i\}} \quad & \frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\sum_i \xi_i \\ \text{s.t. } \forall 1 \le i \le N, \quad & y_i(\mathbf{w}^\top\mathbf{x}_i + b) \ge 1 - \xi_i, \\ & \xi_i \ge 0 \end{aligned}$$

where N is the number of the training examples. As we discussed in the class, the slack variables $\{\xi_i\}$ are introduced to allow the training examples to break into the margin so that we can learn a linear classifier even when the data is not linearly separable.

   (a) [3 point] What values ξ_i can take when the training example $\mathbf{x}_i$ breaks into the margin?

   When the training example breaks into the margin, ξ_i can take any value greater than zero.

   (b) [3 point] What values ξ_i can take when the training example $\mathbf{x}_i$ stays on or outside the margin?

   ξ_i will always be 0 when the training example is on our outside the margin.

   (c) [3 point] Why do we incorporate the term $C \cdot \sum_i \xi_i$ in the objective function? What will happen if we throw out this term?

   We incorporate $C \cdot \sum_i \xi_i$ in the objective function because it penalizes points that break into the margin, by adding that we minimize the amount that points break into the margin. If we did not have this term, then we would not be able to optimally separate the dataset as points could break into the margin unpenalized.

2. [6 points] Write down the dual optimization problem for soft SVMs. Please clearly indicate the constraints, and explain how it is derived. (Note: do NOT directly copy slides content, write down your own understanding.)

The primal form of the optimization problem for SVM attempts to minimize the function $\frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum \xi_i$ where $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ is a regularization term that helps maximize the margin and $C\sum \xi_i$ minimizes the number of points that cross into the margin. We minimize this function subject to the constraints that the prediction $\hat{y} = \mathbf{w}^T\mathbf{x}_i + b$ times the true y value is greater than $1 - \xi_i$ and that ξ_i is greater or equal to zero. Because constrained optimizations can be very difficult to solve directly, we convert this to the Lagrange form:

$$\frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum \xi_i - \sum \beta_i\xi_i - \sum \alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + \xi_i)$$

In the primal form, we can solve this by first maximizing with respect to α and β and then minimize with respect to the weights, biases, and slack variables. We can switch this max and min to get the dual form which we know will give the same result according to Slater's condition for convex optimization. Because we are in the dual form, we first want to minimize with respect to the weights, biases, and slack variables which we can do by taking the derivative of the function with respect to each and set that to zero. When we do this we get that the minimum for the weights is $\sum \alpha_i y_i \mathbf{x}_i$, the minimum for the slack variables is C and the bias is minimized when $\sum \alpha_i y_i = 0$. After this minimization we can substitute these results back into the equation to get

$$min_{\alpha_i \geq 0, \beta_i \geq 0} \frac{1}{2}\sum_i \sum_j y_i y_j a_i a_j \mathbf{x}_i^T \mathbf{x}_j - \sum_i \alpha_i$$

With the constraints that $\sum \alpha_i y_i = 0$ and $\alpha_i + \beta_i = C$. We can remove β here by also constraining $\alpha_i \leq C$. From our minimization before we know that the optimal weights are $\mathbf{w^*} = \sum \alpha_i y_i \mathbf{x}_i$ To find the optimal bias we can use the KKT condition that $\lambda_i f_i(x) = 0$ which gives us $\beta *_i \xi_i = 0$ and $\alpha_i * (y_i(\mathbf{w^*}^T\mathbf{x}_i + b*) - 1 + \xi_i) = 0$. Since we have the constraints that $\alpha_i * + \beta_i* = C$ and $0 < \alpha_j* < C$, we can now solve for the bias term to get $b* = y_j = \mathbf{w^*}^T\mathbf{x}_j$. This is the solution to the dual optimization problem.

3. [10 points] Continue with the dual form. Suppose after the training procedure, you have obtained the optimal parameters.

   (a) [4 points] What parameter values can indicate if an example stays outside the margin?

   The slack variable associated with the example, ξ_i, determines if an example stays outside the margin. This variable will be equal to zero if the example is outside the margin and will be greater than zero if it breaks the margin.

   (b) [6 points] if we want to find out which training examples just sit on the margin (neither inside nor outside), what shall we do? Note you are not allowed to examine if the functional margin (i.e., $y_i(\mathbf{w}^\top \mathbf{x}_i + b)$) is 1.

   The training examples that are inside or on the margin are the support vectors and all of these have a corresponding $\alpha*$ value that is greater than 0. We also know that if an example lies inside the margin then the corresponding $\alpha*$ value will be equal to it's maximum value, C. Therefore to determine if a example lies exactly on the margin then we can look and see if the corresponding $\alpha*$ is greater than zero and is also less than C.

4. [6 points] How can we use the kernel trick to enable SVMs to perform nonlinear classification? What is the corresponding optimization problem?

   The prediction of an SVM model is calculated as $sgn(\mathbf{w*}^T\mathbf{x} + b*)$ and since we are using the dual form, we know that the optimal weight vector is $\mathbf{w*} = \sum \alpha_i y_i \mathbf{x}_i$. To employ the kernel trick we need to map the input vectors to a higher dimension via some function $\phi(x)$. Now we can substitute x in the weight vector function to be $\mathbf{w*}\phi(x) = \sum \alpha *_i y_i \phi(x_i)^T \phi(x)$. If we know some function $K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z})$ we can rewrite the function for the weight vector as $\mathbf{w*}\phi(x) = \sum \alpha *_i y_i K(\mathbf{x}_i, \mathbf{x})$. This gives us the equation for predictions using the kernel trick $\hat{y} = sgn(\sum_i \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b)$.

5. [9 points] Suppose we have the training dataset shown in Table 1. We want to learn a SVM classifier. We initialize all the model parameters with 0. We set the learning rates for the first three steps to $\{0.01, 0.005, 0.0025\}$. Please list the sub-gradients of the SVM objective w.r.t the model parameters for the first three steps, when using the stochastic sub-gradient descent algorithm.

x_1	x_2	x_3	y
0.5	-1	0.3	1
-1	-2	-2	-1
1.5	0.2	-2.5	1

Table 1: Dataset

Step 1:
$\gamma_1 = 0.01$, $\mathbf{w} = [0,0,0,0]$, $C = \frac{1}{3}$, $N = 3$
$y_i \mathbf{w}^T \mathbf{x}_i = 1 * [0,0,0,0]^T [0.5, -1, 0.3, 1] = 0 \leq 1$
$\nabla J^1 = \gamma_1 [\mathbf{w}_0; 0] - \gamma_1 C N y_i \mathbf{x}_i = 0.01 * [0,0,0,0] - 0.01 * \frac{1}{3} * 3 * 1 * [0.5, -1, 0.3, 1]$
$= [0,0,0,0] - [0.005, -0.01, 0.003, 0.001]$
$\mathbf{w} = \mathbf{w} - \nabla J^1 = [0,0,0,0] + [0.005, -0.01, 0.003, 0.01] = [0.005, -0.01, 0.003, 0.01]$
Step 2:
$\gamma_2 = 0.005$, $\mathbf{w} = [0.005, -0.01, 0.003, 0.01]$, $C = \frac{1}{3}$, $N = 3$
$y_i \mathbf{w}^T \mathbf{x}_i = -1 * [0.005, -0.01, 0.003, 0.01]^T [-1, -2, -2, 1] = -0.019 \leq 1$
$\nabla J^2 = \gamma_2 [\mathbf{w}_0; 0] - \gamma_2 C N y_i \mathbf{x}_i$
$= 0.005 * [0.005, -0.01, 0.003, 0] - 0.005 * \frac{1}{3} * 3 * -1 * [-1, -2, -2, 1]$
$= [0.000025, -0.00005, 0.000015, 0] - [0.005, 0.01, 0.01, -0.005]$
$\mathbf{w} = \mathbf{w} - \nabla J^2 = [0.005, -0.01, 0.003, 0.01] - [0.000025, -0.00005, 0.000015, 0]$
$+[0.005, 0.01, 0.01, -0.005] = [0.009975, -0.00005, 0.012985, 0.005]$
Step 3:
$\gamma_3 = 0.0025$, $\mathbf{w} = [0.009975, -0.00005, 0.012985, 0.005]$, $C = \frac{1}{3}$, $N = 3$
$y_i \mathbf{w}^T \mathbf{x}_i = 1 * [0.009975, -0.00005, 0.012985, 0.005]^T [1.5, 0.2, -2.5, 1] = -0.01251 \leq 1$
$\nabla J^3 = \gamma_3 [\mathbf{w}_0; 0] - \gamma_3 C N y_i \mathbf{x}_i$
$= 0.0025 * [0.009975, -0.00005, 0.012985, 0] - 0.0025 * \frac{1}{3} * 3 * 1 * [1.5, 0.2, -2.5, 1]$
$= [0.0000249375, -0.000000125, 0.0000324625, 0] - [0.00375, 0.0005, -0.00625, -0.0025]$
$\mathbf{w} = \mathbf{w} - \nabla J^3 = [0.009975, -0.00005, 0.012985, 0.005]$
$-[0.0000249375, -0.000000125, 0.0000324625, 0] + [0.00375, 0.0005, -0.00625, -0.0025]$
$= [0.0137000625, 0.000450125, 0.0067025375, 0.0025]$

6. [**Bonus**][10 points] Let us derive a dual form for Perceptron. Recall, in each step of Perceptron, we add to the current weights $\mathbf{w}$ (including the bias parameter) $y_i \mathbf{x}_i$ for some misclassified example $(\mathbf{x}_i, y_i)$. We initialize $\mathbf{w}$ with $\mathbf{0}$. So, instead of updating $\mathbf{w}$, we can maintain for each training example i a mistake count c_i — the number of times the data point $(\mathbf{x}_i, y_i)$ has been misclassified.

   - [2 points] Given the mistake counts of all the training examples, $\{c_1, \ldots, c_N\}$, how can we recover $\mathbf{w}$? How can we make predictions with these mistake counts?

- [3 points] Can you develop an algorithm that uses mistake counts to learn the Perceptron? Please list the pseudo code.
- [5 points] Can you apply the kernel trick to develop an nonlinear Perceptron? If so, how do you conduct classification? Can you give the pseudo code fo learning this kernel Perceptron?

2 Practice [60 points + 10 bonus]

1. [2 Points] Update your machine learning library. Please check in your implementation of Perceptron, voted Perceptron and average Perceptron algorithms. Remember last time you created the folders "Perceptron". You can commit your code into the corresponding folders now. Please also supplement README.md with concise descriptions about how to use your code to run these algorithms (how to call the command, set the parameters, etc). Please create a new folder "SVM" in the same level as these folders.

2. [28 points] We will first implement SVM in the primal domain with stochastic subgradient descent. We will reuse the dataset for Perceptron implementation, namely, "bank-note.zip" in Canvas. The features and labels are listed in the file "classification/data-desc.txt". The training data are stored in the file "classification/train.csv", consisting of 872 examples. The test data are stored in "classification/test.csv", and comprise of 500 examples. In both the training and test datasets, feature values and labels are separated by commas. Set the maximum epochs T to 100. Don't forget to shuffle the training examples at the start of each epoch. Use the curve of the objective function (along with the number of updates) to diagnosis the convergence. Try the hyperparameter C from $\{\frac{100}{873}, \frac{500}{873}, \frac{700}{873}\}$. Don't forget to convert the labels to be in $\{1, -1\}$.

   (a) [12 points] Use the schedule of learning rate: $\gamma_t = \frac{\gamma_0}{1+\frac{\gamma_0}{a}t}$. Please tune $\gamma_0 > 0$ and $a > 0$ to ensure convergence. For each setting of C, report your training and test error.

   To get the optimal a and γ_0 I setup a loop testing a number of configurations ten times each and then chose the parameters that gave the best average error over the 10 iterations. I got an optimal value of $a = 0.1$ and $\gamma_0 = 10$.
   For when $C = \frac{100}{873}$ I got a training error of 0.0183 test error of 0.02.
   For when $C = \frac{500}{873}$ I got a training error of 0.0172 test error of 0.018.
   For when $C = \frac{700}{873}$ I got a training error of 0.03 test error of 0.034.

   (b) [12 points] Use the schedule $\gamma_t = \frac{\gamma_0}{1+t}$. Report the training and test error for each setting of C.

   Using this schedule for the learning rate I got 0.1 as the optimal value of γ_0.
   For when $C = \frac{100}{873}$ I got a training error of 0.023 test error of 0.028.
   For when $C = \frac{500}{873}$ I got a training error of 0.0126 test error of 0.014.
   For when $C = \frac{700}{873}$ I got a training error of 0.0138 test error of 0.014.

(c) [6 points] For each C, report the differences between the model parameters learned from the two learning rate schedules, as well as the differences between the training/test errors. What can you conclude?

When using $\gamma_t = \frac{\gamma_0}{1+\frac{\gamma_0}{a}t}$ as the learning rate schedule with $a = 0.1$ and $\gamma_0 = 10$ I got the following weight vectors for each value of C:
$C = \frac{100}{873} \rightarrow \mathbf{w} = [-2.92836704, -1.28228223, -1.09884969, -0.17259185, 3.00887882]$
$C = \frac{500}{873} \rightarrow \mathbf{w} = [-10.44540441, -6.22395722, -7.78511595, -2.76523431, 12.97856014]$
$C = \frac{700}{873} \rightarrow \mathbf{w} = [-18.44515873, -13.52211754, -8.97633243, -0.4712604, 18.73860819]$
When using $\gamma_t = \frac{\gamma_0}{1+t}$ as the learning rate schedule with $\gamma_0 = 0.1$ I got the following weight vectors for each value of C:
$C = \frac{100}{873} \rightarrow \mathbf{w} = [-3.15783175, -1.12659864, -1.73718069, -0.10467725, 3.61584254]$
$C = \frac{500}{873} \rightarrow \mathbf{w} = [-10.49477461, -8.99685779, -7.34250584, -1.37162781, 13.83774027]$
$C = \frac{700}{873} \rightarrow \mathbf{w} = [-18.88804074, -9.38329448, -9.7998478, -3.82816754, 21.06125465]$
Because I shuffled the training set on every epoch, the training and testing errors varied quite a bit on each run. This made determining if there is a 'best' value of C difficult although on average, I found that I usually got the best training and test errors when $C = \frac{500}{873}$. The differences were very slight though and on average I didn't see a major difference between the different values of C.

Overall I found that the learned weights between the two different learning rate schedules were very similar. This leads me to believe that the performance of these two learning rate schedules is not very different. For both learning schedule rates, I found that as we used greater values for C, the magnitude of the learned weights increased as well. This makes sense since the higher value of C, the higher the penalty for examples that break into the margin so the separating hyperplane shifts to reduce the number of examples that break into the margin during training.

3. [30 points]

(a) [10 points] First, run your dual SVM learning algorithm with C in $\{\frac{100}{873}, \frac{500}{873}, \frac{700}{873}\}$. Recover the feature weights $\mathbf{w}$ and the bias b. Compare with the parameters learned with stochastic sub-gradient descent in the primal domain (in Problem 2) and the same settings of C, what can you observe? What do you conclude and why? Note that if your code calculates the objective function with a double loop, the optimization can be quite slow. To accelerate, consider writing down the objective in terms of the matrix and vector operations, and treat the Lagrange multipliers that we want to optimize as a vector! Recall, we have discussed about it in our class.

When $C = \{\frac{100}{873}$ I computed the following weights and bias by minimizing the dual form:
$\mathbf{w} = [-0.94303719 - 0.65148183 - 0.7336991 - 0.04098827]$
$\mathbf{b} = 2.517325$
The training error was 0.0264 and the test error was 0.03.
When $C = \{\frac{500}{873}$ I computed the following weights and bias by minimizing the dual form:

$\mathbf{w} = [-1.56402604 - 1.01358698 - 1.18028788 - 0.15630374]$
$\mathbf{b} = 3.964289$
The training error was 0.031 and the test error was 0.036.
When $C = \{\frac{700}{873}$ I computed the following weights and bias by minimizing the dual form:
$\mathbf{w} = [-2.04269191 - 1.28026842 - 1.51356736 - 0.24800877]$
$\mathbf{b} = 5.037643$
The training error was 0.0344 and the test error was 0.036.

I found that overall, the weights and biases for each of the values of C that I computed by minimizing the dual form of SVM were significantly lower than those that I computed in problem 2. This is likely because when I calculated the parameters using stochastic sub gradient descent, the weights and biases were scaled by the learning rate as the algorithm went on. I also noticed that similar to my results in problem 2, the magnitude of the weights and biases produced from solving the dual form increased as the value of C increased, which indicates that in both algorithms the hyper-plane shifts in the same direction to reduce the number of examples that break into the margin.

(b) [15 points] Now, use Gaussian kernel in the dual form to implement the non-linear SVM. Note that you need to modify both the objective function and the prediction. The Gaussian kernel is defined as follows:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\gamma}).$$

Test γ from $\{0.1, 0.5, 1, 5, 100\}$ and the hyperparameter C from $\{\frac{100}{873}, \frac{500}{873}, \frac{700}{873}\}$. List the training and test errors for the combinations of all the γ and C values. What is the best combination? Compared with linear SVM with the same settings of C, what do you observe? What do you conclude and why?

I found the following training and testing errors for the different values of C and gamma:
C = $\frac{100}{873}$
Training error when $\gamma = 0.100000$: 0.000000
Testing error when $\gamma = 0.100000$: 0.002000
Training error when $\gamma = 0.500000$: 0.000000
Testing error when $\gamma = 0.500000$: 0.002000
Training error when $\gamma = 1.000000$: 0.000000
Testing error when $\gamma = 1.000000$: 0.002000
Training error when $\gamma = 5.000000$: 0.008028
Testing error when $\gamma = 5.000000$: 0.006000
Training error when $\gamma = 100.000000$: 0.003440
Testing error when $\gamma = 100.000000$: 0.004000
C = $\frac{500}{873}$
Training error when $\gamma = 0.100000$: 0.000000
Testing error when $\gamma = 0.100000$: 0.002000

Training error when $\gamma = 0.500000$: 0.000000
Testing error when $\gamma = 0.500000$: 0.002000
Training error when $\gamma = 1.000000$: 0.000000
Testing error when $\gamma = 1.000000$: 0.002000
Training error when = 5.000000 : 0.000000
Testing error when $\gamma = 5.000000$: 0.002000
Training error when $\gamma = 100.000000$: 0.000000
Testing error when $\gamma = 100.000000$: 0.000000
C = $\frac{700}{873}$
Training error when $\gamma = 0.100000$: 0.000000
Testing error when $\gamma = 0.100000$: 0.002000
Training error when $\gamma = 0.500000$: 0.000000
Testing error when $\gamma = 0.500000$: 0.002000
Training error when $\gamma = 1.000000$: 0.000000
Testing error when $\gamma = 1.000000$: 0.002000
Training error when $\gamma = 5.000000$: 0.000000
Testing error when $\gamma = 5.000000$: 0.002000
Training error when $\gamma = 100.000000$: 0.000000
Testing error when $\gamma = 100.000000$: 0.000000
When $C = \frac{500}{873}$ or $C = \frac{700}{873}$ and when $\gamma = 100$ I found that both the training and test error was 0, meaning that it is able to perfectly separate the data. Without the kernel, when $C = \frac{500}{873}$ the training error was 0.031 and the testing error was 0.036. This increase in performance when using the kernel happens because when using the gaussian kernel, we are mapping the input variables to a higher dimension and increasing the separability of the data. By doing this we are able to find a better separating hyperplane and therefore the accuracy is improved.

(c) [5 points] Following (b), for each setting of γ and C, list the number of support vectors. When $C = \frac{500}{873}$, report the number of overlapped support vectors between consecutive values of γ, i.e., how many support vectors are the same for $\gamma = 0.01$ and $\gamma = 0.1$; how many are the same for $\gamma = 0.1$ and $\gamma = 0.5$, etc. What do you observe and conclude? Why?

I got the following number of support vectors for each combination of C and γ
C = $\frac{100}{873}$
$\gamma = 0.100000$ - Number of support vectors: 869
$\gamma = 0.500000$ - Number of support vectors: 825
$\gamma = 1.00000$ - Number of support vectors: 804
$\gamma = 5.00000$ - Number of support vectors: 442
$\gamma = 100.00000$ - Number of support vectors: 290
C = $\frac{500}{873}$
$\gamma = 0.100000$ - Number of support vectors: 869
$\gamma = 0.500000$ - Number of support vectors: 730
$\gamma = 1.00000$ - Number of support vectors: 556
$\gamma = 5.00000$ - Number of support vectors: 208

$\gamma = 100.00000$ - Number of support vectors: 116
C = $\frac{700}{873}$
$\gamma = 0.100000$ - Number of support vectors: 868
$\gamma = 0.500000$ - Number of support vectors: 693
$\gamma = 1.00000$ - Number of support vectors: 528
$\gamma = 5.00000$ - Number of support vectors: 193
$\gamma = 100.00000$ - Number of support vectors: 99
When $C = \frac{500}{873}$:
I found that 869 support vectors were the same for when $\gamma = 0.01$ and$\gamma = 0.1$
I found that 730 support vectors were the same for when $\gamma = 0.1$ and$\gamma = 0.5$
I found that 554 support vectors were the same for when $\gamma = 0.5$ and$\gamma = 1$
I found that 198 support vectors were the same for when $\gamma = 1$ and$\gamma = 5$
I found that 73 support vectors were the same for when $\gamma = 5$ and$\gamma = 100$
From these results we can see that the number of support vectors goes down when the value of γ goes up and when the value of C goes up. The number of support vectors going down when γ goes up is reasonable because when the value of gamma goes up, we are spreading out the point space more, increasing the separability so fewer points will break into or be on the margin. It is also reasonable that the number of support vectors will go down when the value of C goes up because when we increase C, we are increasing the penalty for points that break into the margin which reduces the number of points that are inside the margin.

(d) [**Bonus**] [10 points] Implement the kernel Perceptron algorithm you developed in Problem 8 (Section 1). Use Gaussian kernel and test γ from $\{0.1, 0.5, 1, 5, 100\}$. List the training and test errors accordingly. Compared with the nonlinear SVM, what do you observe? what do you conclude and why?