CSE 547: Homework 3

Due on May 19, 2017

Brian de Silva

Collaborators

I collaborated with Emily Dinan on problems 1.5, 2.2, and understanding how to do the batch updates for SDCA in problem 3.
I collaborated with Weston Barger on problem 1.5.

1 (Dual) Coordinate Ascent

In this problem, we will derive the dual coordinate ascent algorithm for the special case of ridge regression. In particular, we seek to solve the problem:

$$\min_w L(w) \text{ where } L(w) = \sum_{i=1}^{n} (w \cdot x_i - y_i)^2 + \lambda \|w\|^2$$

Recall that the solution is:

$$w^* = (X^\top X + \lambda I)^{-1} X^\top Y$$

where X be the $n \times d$ matrix whose rows are x_i, and Y is an n-dim vector.
Recall the argument:

$$\begin{aligned} w^* &= (X^\top X + \lambda I)^{-1} X^\top Y \\ &= X^\top (XX^\top + \lambda I)^{-1} Y \\ &:= \frac{1}{\lambda} X^\top \alpha^* \end{aligned}$$

where $\alpha^* = (I + XX^\top/\lambda)^{-1} Y$.

1. (Linear algebra brush up) Prove that the above manipulations are true (without any assumptions on X or n or d). This involves proving the second equality in the expression.

As in class, define:

$$G(\alpha_1, \alpha_2, \dots \alpha_n) = \frac{1}{2} \alpha^\top (I + XX^\top/\lambda)\alpha - Y^\top \alpha$$

Now let us consider the consider the coordinate ascent algorithm:

- start with $\alpha = 0$.
- choose coordinate i randomly, and update:

$$\alpha_i = \arg\min_z G(\alpha_1, \dots \alpha_{i-1}, z, \dots, \alpha_n)$$

- repeat the previous step until some termination criterion is reached.
- return $w = \frac{1}{\lambda} X^\top \alpha$.

2. Show that the solution to the inner optimization problem for α_i is:

$$\alpha_i = \frac{(y_i - \frac{1}{\lambda}(\sum_{j \neq i} \alpha_j x_j) \cdot x_i)}{1 + \|x_i\|^2/\lambda}$$

3. What is the computational complexity of this update, as it is stated? (Assume that scalar multiplication and scalar addition is order one.)

4. What is the computational complexity of one stochastic gradient descent update?

Now let us consider the update rule from class:

- start with $\alpha = 0$, $w = \frac{1}{\lambda}X^\top \alpha = 0$.
- Choose coordinate i randomly and perform the following update:
  - Compute the differences:
$$\Delta\alpha_i = \frac{(y_i - w \cdot x_i) - \alpha_i}{1 + \|x_i\|^2/\lambda} \tag{1}$$
  - Update the parameters as follows:
$$\alpha_i \leftarrow \alpha_i + \Delta\alpha_i, \quad w \leftarrow w + \frac{1}{\lambda}x_i \cdot \Delta\alpha_i \tag{2}$$
- Repeat the previous step until some termination criterion is reached.
- return $w = \frac{1}{\lambda}X^\top \alpha$.

Now let us examine why this algorithm is more efficient.

5. Prove that the above update is valid.
6. What is the computational complexity of the update, Equation (2), in the above implementation?

Solution:

1. Note that both $XX^\top$ and $X^\top X$ are symmetric positive-semidefinite, so all of their eigenvalues λ_i satisfy $\lambda_i \geq 0$ (they share the same eigenvalues, with the larger of the two matrices having 0 as a repeated eigenvalue with higher algebraic multiplicity). The eigenvalues of $X^\top X + \lambda$ and $XX^\top + \lambda I$ are therefore $\lambda_i + \lambda$. If $\lambda > 0$ then these eigenvalues are strictly positive, in which case it follows that both matrices are invertible. Hence
$$\begin{array}{rrcl} & X^\top Y &=& X^\top Y \\ \iff & X^\top Y &=& X^\top I Y \\ \iff & X^\top Y &=& X^\top(XX^\top + \lambda I)(XX^\top + \lambda I)^{-1}Y \\ \iff & X^\top Y &=& (X^\top X + \lambda I)X^\top(XX^\top + \lambda I)^{-1}Y \\ \iff & (X^\top X + \lambda I)^{-1}X^\top Y &=& X^\top(XX^\top + \lambda I)^{-1}Y. \end{array}$$
This shows that the second equality in the expression is true.

2. Differentiating G with respect to α_i yields
$$\begin{aligned} \frac{\partial G}{\partial \alpha_i} &= [(I + \tfrac{1}{\lambda}XX^\top)\alpha]_i - y_i \\ &= (I + \tfrac{1}{\lambda}XX^\top)_i \cdot \alpha - y_i \\ &= \sum_{j\neq i}^n \left(\frac{x_i \cdot x_j}{\lambda}\alpha_j\right) + \left(1 + \frac{x_i \cdot x_i}{\lambda}\right)\alpha_i - y_i \\ &= \tfrac{1}{\lambda}\left(\sum_{j\neq i}^n \alpha_j x_j\right) \cdot x_i + \left(1 + \frac{\|x_i\|^2}{\lambda}\right)\alpha_i - y_i. \end{aligned}$$

where $(I + \frac{1}{\lambda}XX^\top)_i$ is the i-th row of $I + \frac{1}{\lambda}XX^\top$. Note that we have used that $(XX^\top)_{ij} = x_i \cdot x_j$. To find the optima we set the above expression equal to zero and solve for α_i

$$\begin{aligned} & \tfrac{1}{\lambda}\left(\textstyle\sum_{j\neq i}^n \alpha_j x_j\right)\cdot x_i + \left(1+\tfrac{\|x_i\|^2}{\lambda}\right)\alpha_i - y_i &=& \; 0 \\ \Longrightarrow \quad & \left(1+\tfrac{\|x_i\|^2}{\lambda}\right)\alpha_i &=& \; y_i - \tfrac{1}{\lambda}\left(\textstyle\sum_{j\neq i}^n \alpha_j x_j\right)\cdot x_i \\ \Longrightarrow \quad & \alpha_i &=& \; \tfrac{y_i - \frac{1}{\lambda}\left(\sum_{j\neq i}^n \alpha_j x_j\right)\cdot x_i}{1+\|x_i\|^2/\lambda}. \end{aligned} \tag{3}$$

3. As it is stated the most expensive part of the update is the computation of the sum $\sum_{j\neq i}^n \alpha_j x_j$. For each of the $n-1$ terms one must perform d multiplications, totalling $\mathcal{O}(nd)$ operations. Adding these terms together costs us another $d(n-1) = \mathcal{O}(nd)$ operations. The other computations are relatively cheap compared to this one, so the total complexity is $\mathcal{O}(nd)$.

4. An update of stochastic gradient descent is of the form

$$w \leftarrow w + \eta\left(-\lambda w + x^j\left(y^j - \frac{\exp(w_0 + w\cdot x^j)}{1+\exp(w_0+w\cdot x^j)}\right)\right)$$

which only costs $\mathcal{O}(d)$ operations (the vector additions, dot products, and multiplication of x^j by a scalar all cost $\mathcal{O}(d)$ operations).

5. First we observe that

$$\begin{aligned} \Delta\alpha_i &= \frac{(y_i - w\cdot x_i) - \alpha_i}{1+\|x_i\|^2/\lambda} \\ &= \frac{y_i - \frac{1}{\lambda}\left(\sum_j \alpha_j x_j\right)\cdot x_i - \alpha_i}{1+\|x_i\|^2/\lambda} \\ &= \frac{y_i - \frac{1}{\lambda}\left(\sum_{j\neq i}\alpha_j x_j\right)\cdot x_i - \alpha_i\left(1+\frac{1}{\lambda}\|x_i\|^2\right)}{1+\|x_i\|^2/\lambda} \\ &= \frac{y_i - \frac{1}{\lambda}\left(\sum_{j\neq i}\alpha_j x_j\right)\cdot x_i}{1+\|x_i\|^2/\lambda} - \alpha_i. \end{aligned}$$

Hence the update (2) reduces to

$$\alpha_i + \Delta\alpha_i \;=\; \alpha_i + \frac{y_i - \frac{1}{\lambda}\left(\sum_{j\neq i}\alpha_j x_j\right)\cdot x_i}{1+\|x_i\|^2/\lambda} - \alpha_i \;=\; \frac{y_i - \frac{1}{\lambda}\left(\sum_{j\neq i}\alpha_j x_j\right)\cdot x_i}{1+\|x_i\|^2/\lambda},$$

which is the solution to the inner optimization problem for α_i as demonstrated in (3).

Next we consider the update for w. Before updating, $w = \frac{1}{\lambda}X^\top\alpha$. Then α is updated as in (2), which can also be expressed as $\alpha \leftarrow \alpha + e_i\Delta\alpha_i$, where e_i is the vector with 0's in all but the i-th entry where it has a 1. Thus the new w should be

$$\begin{aligned} & w \leftarrow \tfrac{1}{\lambda}X^\top(\alpha + e_i\Delta\alpha_i) \\ \Longrightarrow \quad & w \leftarrow \tfrac{1}{\lambda}X^\top\alpha + \tfrac{1}{\lambda}X^\top e_i\Delta\alpha_i \\ \Longrightarrow \quad & w \leftarrow w + \tfrac{1}{\lambda}x_i\cdot\Delta\alpha_i \end{aligned}$$

which agrees with the update given in (2).

6. The computational complexity of the update in Equation (2) is of complexity $\mathcal{O}(d)$ since computing $\Delta\alpha_i$ requires two d-dimensional dot products, each costing $2d-1 = \mathcal{O}(d)$ operations. Updating w also costs $\mathcal{O}(d)$ operations because it requires multiplying a d-dimensional vector by a scalar and adding two d-dimensional vectors.

2 Let's try it out!

Dimension Reduction with PCA

Project each image onto the top 50 PCA directions. This reduces the dimension of each image from 784 to 50. The reason for doing this is to speed up the computation.

Feature generation: Random Fourier Features to approximate the RBF Kernel

Now we will map each x to a k-dimensional feature vector as follows:

$$x \to (h_1(x), h_2(x), \dots h_k(x))$$

We will use $k = N = 30,000$. Each h_j will be of the form:

$$h_j(x) = sin\left(\frac{v_j \cdot x}{\sigma}\right)$$

To construct all these vectors $v_1, v_2, \dots v_k$, you will independently sample every coordinate for every vector from a standard normal distribution (with unit variance). Here, you can think of σ as a bandwidth parameter.

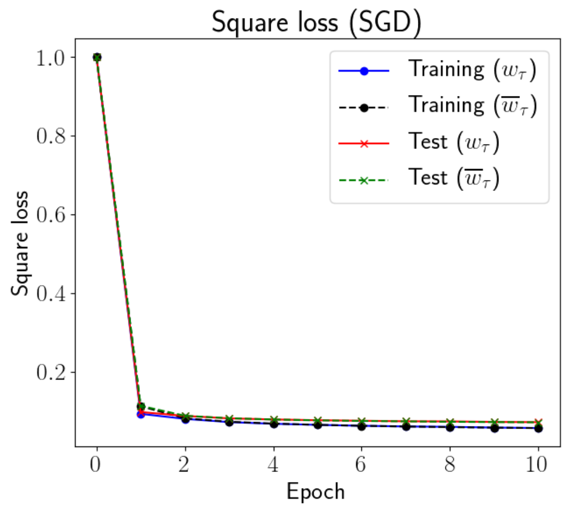
2.1 SGD and Averaging

Let us optimize the square loss. We do this as a surrogate to obtain good classification accuracy (on the test set). In practice, instead of complete randomization, we often run in *epochs*, where we randomly permute our dataset and then pass through out dataset in this permuted order. Each time we start another epoch, we permute our dataset again.
Let us plot the losses of two quantities. Let us keep around your parameter vector w_t (your weight vector at iteration t). Let us also track the average weight vector $\overline{w}_\tau$ over the last epoch, i.e. $\overline{w}_\tau$ is the average parameter vector over the τ-th epoch. The average parameter is what is suggested to be used by the Polyak-Juditsky averaging method.
Define the total square loss as the *sum* square loss over the 10 classes (so you do not divide by 10 here).
Throughout this question you will use a mini-batch size of 1.

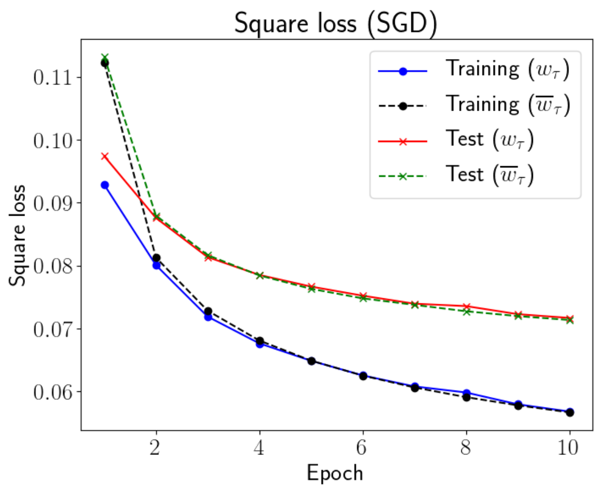
1. Specify your parameter choices: your learning rate (or learning rate scheme) and the kernel bandwidth.
2. You should have one plot showing the both the squared loss after every epoch (starting with your initial squared error). Please label your axes in a readable way. Plot the loss of both w_t and the average weight vector $\overline{w}_\tau$. You should have both the training and test losses on the same plot (so there should be four curves).
3. For the classification loss, do the same, except start your plots at the end of the first epoch. Again, there should be four curves (for the training and test loss of both parameters).
4. What is your final training squared loss and classification loss for both parameters? What is the total number of mistakes that are made on the training set of your final point? Comment on the performance of the average iterate vs. the last point.
5. What is your final test squared loss and classification loss for both parameters? What is the total number of mistakes that are made on the test set of your final point? Comment on the performance of the average iterate vs. the last point.

Solution:

The code for this problem is contained in `hw3-2-1.py`.



(a) The square loss of SGD on the training and test sets as a function of the epoch



(b) The square loss of SGD on the training and test sets as a function of the epoch (starting after the first epoch)

Figure 1: Two views of the square loss of SGD on the training and test sets after each epoch.

1. For this problem I used no regularization, a learning rate of $10^{-5}/(2\sqrt{t+1})$, where t is the epoch, and set all initial weights to 0. My kernel bandwidth was half the approximate mean distance between points in the dataset. In order to approximate this distance I randomly sampled 100 pairs of points from the training data and took the mean of their Euclidean distances from one another. This parameter was on the order of 10^3. To obtain the plots below I ran SGD for 10 epochs. The code took about two hours to run (including the time it took to compute the square and classification loss each epoch).

2. Figure 1a shows the square loss after each epoch on the training and test sets for both w_t and the average weight vector $\overline{w}_\tau$. Figure 1b shows the same square loss, but omits the initial loss.

3. Figure 2 gives the classification loss as a function of iterations for w_t and $\overline{w}_\tau$ on the training and test sets.

	Square loss	**Classification loss**	**Total mistakes**
w_t	0.056793	0.009250	555
$\overline{w}_\tau$	0.056665	0.009433	566

Table 1: Final losses of SGD on the training set

4. The final squared and classification losses on the training set are summarized in Table 1. The average iterate and the final weights produced by SGD give similar performance on the training set with the final weights doing marginally better.

	Square loss	**Classification loss**	**Total mistakes**
w_t	0.071672	0.017400	174
$\overline{w}_\tau$	0.071346	0.016500	165

Table 2: Final losses of SGD on the test set

5. The final squared and classification losses on the test set are summarized in Table 2. Again the performance of the averaged weights and the final weights are similar. On the test set the average weights do a bit better

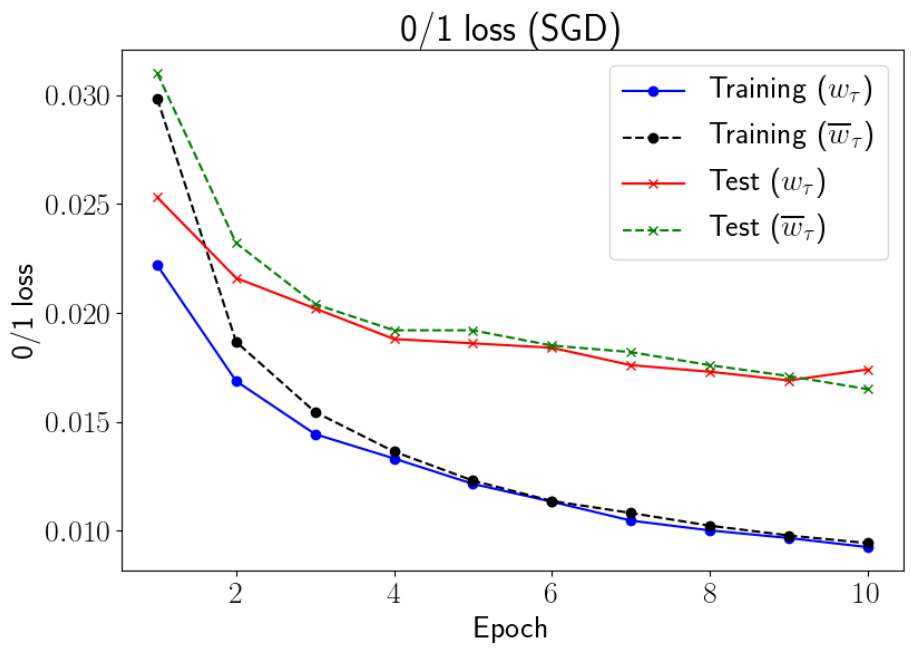


Figure 2: The classification error of SGD on the training and test sets as a function of the epoch

than the final weights.

2.2 SDCA

Now let us try out our (dual) coordinate ascent method. Again, instead of complete randomization, we often run in *epochs*, where we randomly permute our dataset and then pass through out dataset in this permuted order. Each time we start another epoch, we permute our dataset again.
Choose a regularization parameter. Do this appropriately so you obtain a good test accuracy.

1. Specify your regularization parameter. Are there any other parameter choices?
2. Now plot the dual loss G. Your plot should show the G-loss after every epoch (starting with your initial squared error). Please label your axes in a readable way.
3. Does it make sense to plot the G-loss on the test set? Why not?
4. Now plot the squared loss after every epoch (starting with your initial squared error). Please label your axes in a readable way. You should have both the training and test losses on the same plot (so there should be two curves).
5. For the classification loss, do the same, except start your plots at the end of the first epoch. Again, there should be two curves (for the training and test loss).
6. What is your best training squared loss and classification loss? What is the total number of mistakes that are made on the training set of your final point?
7. What is your best test squared loss and classification loss? What is the total number of mistakes that are made on the test set of your final point?
8. Compare the speed and accuracy to what you obtained with SGD?

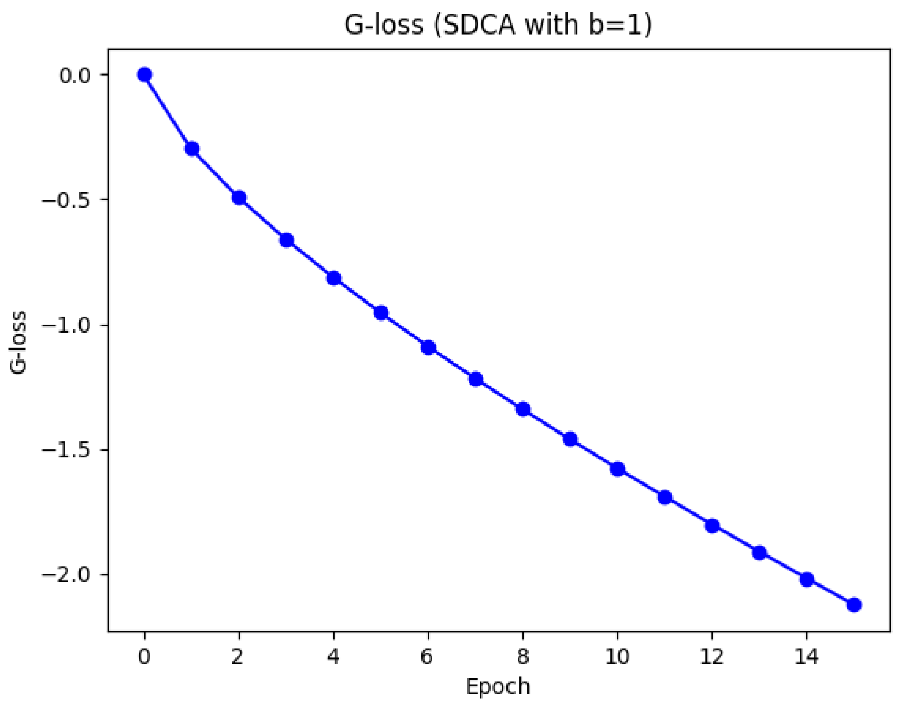
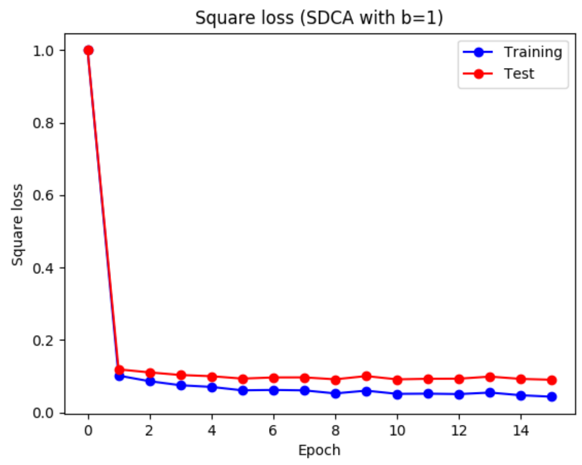


Figure 3: G-loss on training set using SDCA with batch size $b = 1$

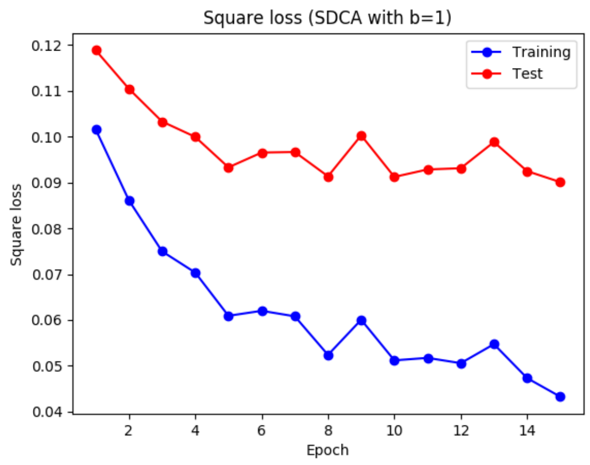
Solution:

The code for this problem is contained in `hw3-2-2.py`.

1. I chose $\lambda = 1.0$ as my regularization parameter. I ran the method for one epoch using a variety of values of λ and found this one to give the best 0/1 loss on the test set. The only other parameters to set were hyperparameters such as the number of epochs to run.
2. Figure 3 shows the G-loss after each of the 15 epochs.
3. It does not make sense to plot the G-loss on the test set because the G-loss is meant to give information about how close our current α is to the true dual coordinates, α^*. But α^* is associated with the solution to the least squares problem involving only the *training set*. Furthermore, α does not even have the right number of entries for us to define a G-loss on the test set (unless the training and test sets were the same size). Each entry of α_i corresponds to an example in the training set.
4. Figure 4 shows two views of the square loss of SDCA on the training and test data. One subfigure omits the initial square error.
5. Figure 5 plots the classification (0/1) loss on the training and test sets as a function of epochs of SDCA.



(a) The square loss for SDCA using batch size $b = 1$



(b) The square loss after 1st epoch for SDCA using batch size $b = 1$

Figure 4: Two views of the square loss for SDCA using batch size $b = 1$

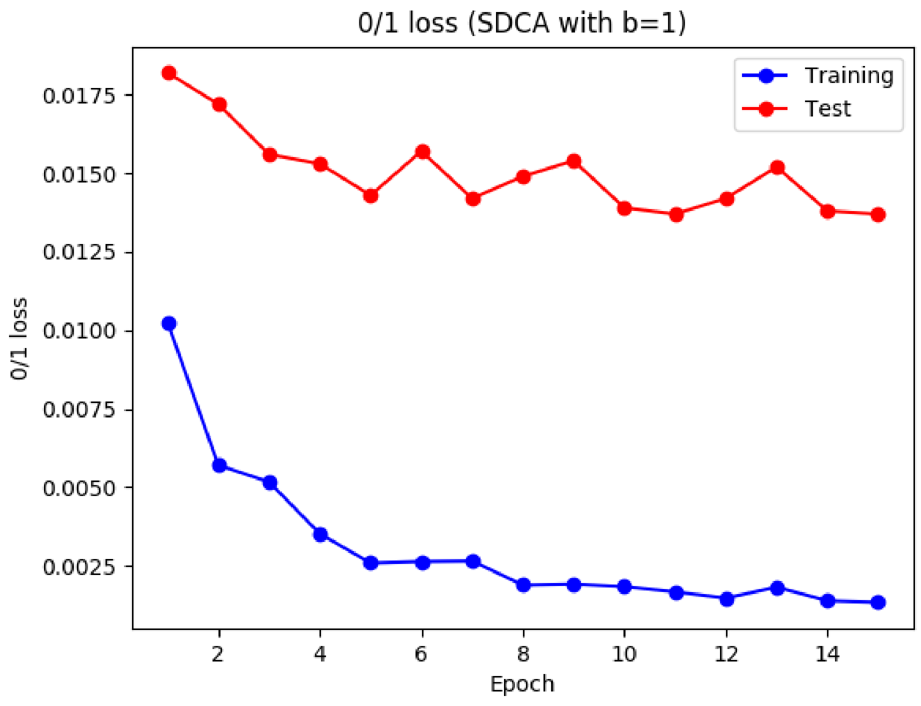


Figure 5: The classification error of SDCA using $b = 1$ on the training and test sets as a function of the epoch

	Square loss	**Classification loss**	**Total mistakes**
Training	0.043270	0.001333	80
Test	0.090068	0.013700	137

Table 3: Final losses of SDCA on the training and test sets using batch size $b = 1$

6. Table 3 summarizes the best square and classification losses on the training and test sets, as well as the total number of mistakes made on each.

7. See Table 3.

8. SDCA is able to produce a solution with slightly better accuracy than SGD on the test set (and much better accuracy on the training set) in roughly the same amount of time (actually almost the exact same amount of time).

3 Mini-batching (in the dual) and parallelization!

Now, let us understand parallelization issues with mini-batching.
Here instead of updating one coordinate at a time, we will update multiple coordinates. In particular, we will choose a batch of b coordinates and compute the Δ_i's of these coordinates as specified in Equation (1).
Let us choose a batch size $b = 100$.

1. One possibility is to just update the α_i's as specified in Equation (2). Is this ok?

2. Argue that the following update never increases the value of G:

$$\alpha_i \leftarrow \alpha_i + \frac{1}{b}\Delta\alpha_i, \quad w \leftarrow w + \frac{1}{b}\frac{1}{\lambda}x_i \cdot \Delta\alpha_i$$

   where the update is performed on all of the chosen coordinates in the mini-batch.

3. Now use this algorithm with $b = 100$. And provide the same training and test loss plots, for the square loss and classification loss as before. (Again, use a permuted order rather than complete randomization). Plot this with your earlier curves for $b = 1$ and have the x axis be the total number of points used. Note that b steps on your x-axis will correspond to one update.

4. In terms of the *total number of data points touched*, does this perform notably better or worse than what you had before, with the $b = 1$?

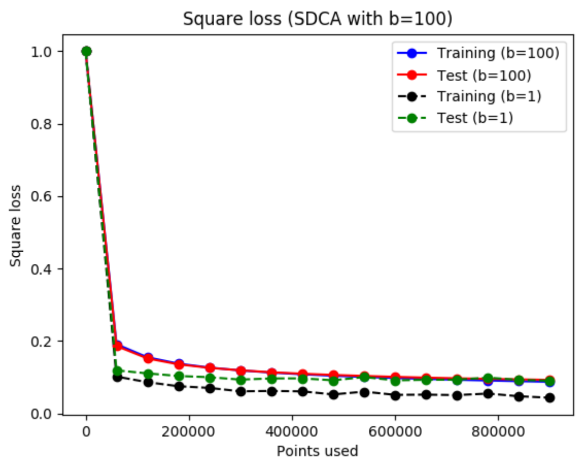
Solution:

The code for this problem is contained in `hw3-2-2.py`.

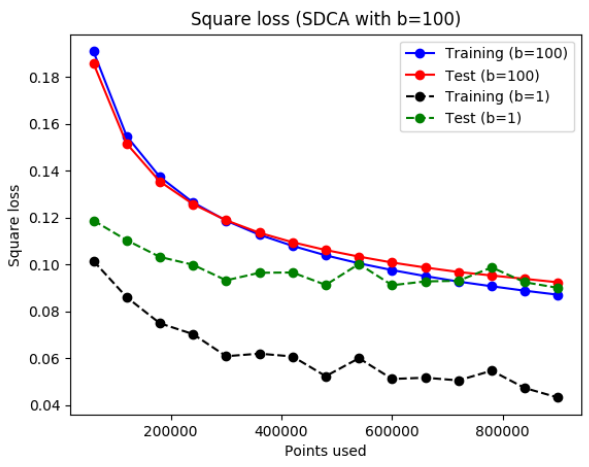
1. No. Consider the extreme case when all b x_i's we select are the same (or are collinear). Since we compute all the $\Delta\alpha_i$'s without updating w, they could all end up having roughly the same value, say $\Delta\alpha$. When we go to update w, instead of modifying it how we would have before, our update is approximately

$$w \leftarrow w + \tfrac{b}{\lambda}x_i \cdot \Delta\alpha.$$

   This pushes w a factor of b too far in the direction x_i (remember that all the b samples are assumed to be roughly x_i).



(a) The square loss for SDCA



(b) The square loss for SDCA after 1st epoch

Figure 6: Two views of the square loss for SDCA with different batch sizes

2. Let I be the set of indices which are to be updated in a given batch so that $|I| = b$. Note that for the same reasons (2) was shown to be valid, updating any one component of α using the rule $\alpha_i \leftarrow \alpha_i + \Delta\alpha_i$ results in a decrease in G. Where we can run into problems is when we update many such components of α in this way before updating w. Observe that we can express all the updates to α in one batch as

$$\alpha \leftarrow \alpha + \tfrac{1}{b}\sum_{i\in I}\Delta\alpha_i e_i,$$

where e_i is again the i-th standard basis vector. Hence the new value of G after such a series of updates is

$$G\left(\alpha + \tfrac{1}{b}\sum_{i\in I}\Delta\alpha_i e_i\right).$$

Since $|I| = b$ this can be rewritten as

$$G\left(\tfrac{1}{b}\sum_{i\in I}(\alpha + \Delta\alpha_i e_i)\right),$$

i.e. as a convex combination of points. Note that $\alpha + \Delta\alpha_i e_i$ is equivalent to the update for α_i given in (2). Then, as G is convex, we have

$$\begin{aligned} G\left(\alpha + \tfrac{1}{b}\sum_{i\in I}\Delta\alpha_i e_i\right) &= G\left(\tfrac{1}{b}\sum_{i\in I}(\alpha + \Delta\alpha_i e_i)\right) \\ &\le \tfrac{1}{b}\sum_{i\in I} G(\alpha + \Delta\alpha_i e_i) \\ &\le \tfrac{1}{b}\sum_{i\in I} G(\alpha) \\ &= G(\alpha). \end{aligned}$$

Hence the proposed update never increases the value of G. The update for w simply changes w to reflect the updates made in the b components of α.

3. Figure 6 gives two perspectives of the square loss of SDCA on the training and testing sets when different batch sizes are used. The horizontal axis is the number of points used by the two variants of SDCA. Similarly, Figure

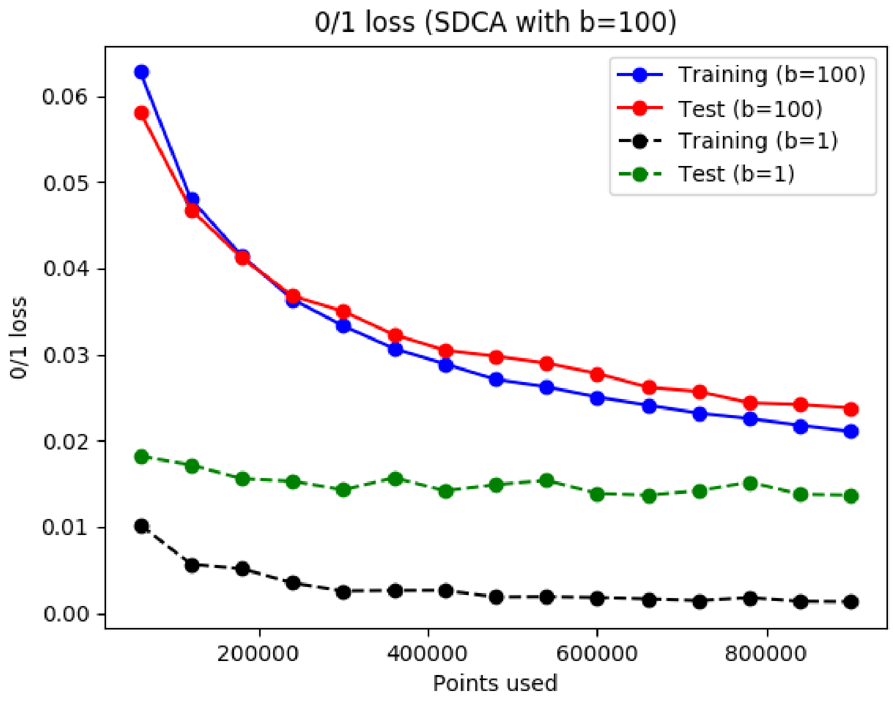


Figure 7: The classification error of SDCA on the training and test sets as a function of the epoch

	Square loss	**Classification loss**	**Total mistakes**
Training	0.087129	0.021067	1264
Test	0.092429	0.023800	238

Table 4: Final losses of SDCA on the training and test sets using batch size $b = 100$

7 plots the classification loss for these methods. Table 4 summarizes the final losses incurred by SDCA using batch size 100.

4. SDCA with batch size 100 performs noticeably worse than SDCA with batch size 1, when each is allowed to use the same number of data points. From Figure 7 we see that although the larger batch size seems to produce a steadier decay in the classificaiton loss, the smaller batch size gives superior results using many fewer examples. After only one epoch (60,000 data points touched) SDCA using batch size 1 has achieved better classification loss on the test set than SDCA with batch size 100 does on the training set after touching 900,000 points. In the next part of the problem we try to improve on this. It should be noted that SDCA with a larger batch size did run a few minutes faster than SDCA with batch size 1.

Now let us improve upon this algorithm. Consider the update rule with parameter γ:

$$\alpha_i \leftarrow \alpha_i + \gamma \Delta\alpha_i, \quad w \leftarrow w + \frac{\gamma}{\lambda} x_i \cdot \Delta\alpha_i$$

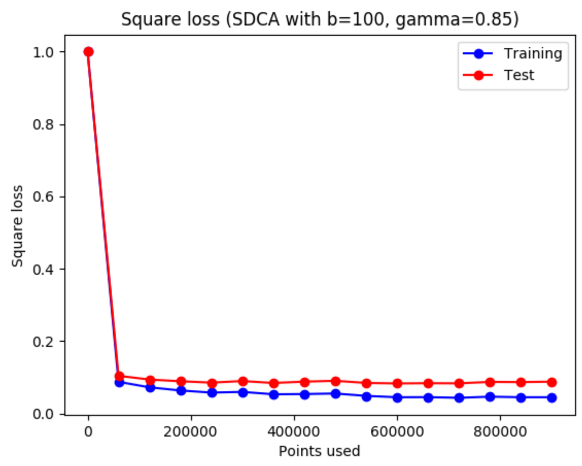
where the update is performed on all of the chosen coordinates in the mini-batch.

1. Give an argument as to why choosing $\gamma = 1/b$ is pessimistic. (Either give a concise argument or give an example.)
2. Now empirically (on our $b = 100$ experiments) search a little to find a good choice for gamma. What value of γ did you choose? What value of γ do things diverge?
3. Now use this choice to redo your experiments. Provide the same training and test loss plots, for the square loss and classification loss as before. (Again, use a permuted order rather than complete randomization).
4. In terms of the *total number of data points touched*, does this perform notably better than what you had before, for both SGD and for the $\gamma = 1/b$ case? What have noticed about your total runtime?

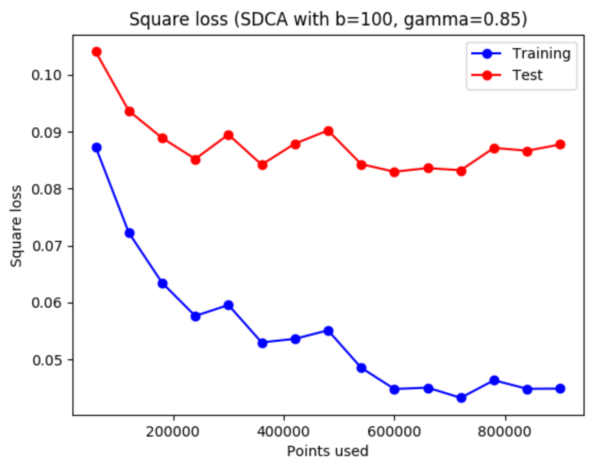
Solution:

The code for this problem is contained in `hw3-2-2.py`.

1. Using the example from before where all of the samples in our batch are identical, i.e. they are all (x_1, y_1), we saw that if we did not divide our batch updates by a factor of b, each term would contribute the same amount in the same direction to w when the update $w \leftarrow w + \frac{\gamma}{\lambda} x_i \cdot \Delta\alpha_i$ was applied. w would be pushed b times too far in the direction of $x_i = x_1$. In this extreme we are forced to choose $\gamma = \frac{1}{b}$ to counteract the additive effect of all these updates. Of course the drawback of doing this is that we can only update each α_i by adding $\frac{1}{b}\Delta\alpha_i$ rather than adding the amount suggested by solving the optimization problem for a single coordinate, $\Delta\alpha_i$. In most circumstances, however, the examples will not be identical or even collinear, so the effects of the different updates will not accumulate in one direction when updating w. This should allow us to better optimize G by choosing a larger value of γ, giving us better approximate solutions to the b single-coordinate optimization problems for the α_i's.
2. To find a good value of γ I tested a set of uniformly spaced points in the interval $(1/b, 1]$ to see which led to the best losses after one epoch of SDCA. I also tested $\gamma = 1.5, 2$, and 4 In the end I chose to use $\gamma = 0.85$ since both $\gamma = 0.801$ and $\gamma = 0.902$ exhibited good performance on the training and test sets. In my experiments things started to diverge when $\gamma = 4$, but converged when $\gamma = 2$.
3. Figure 8 shows two perspectives of the square loss on the training and test sets as a function of number of points touched by SDCA. We used a batch size of $b = 100$ and $\gamma = 0.85$. Figure 9 plots the classification loss for this choice of paramters. Table 5 gives a summary of the losses after 15 epochs (i.e. 900,000 data points touched).



(a) The square loss for SDCA



(b) The square loss after 1st epoch for SDCA

Figure 8: Two views of the square loss for SDCA with batch size 100 and $\gamma = 0.85$

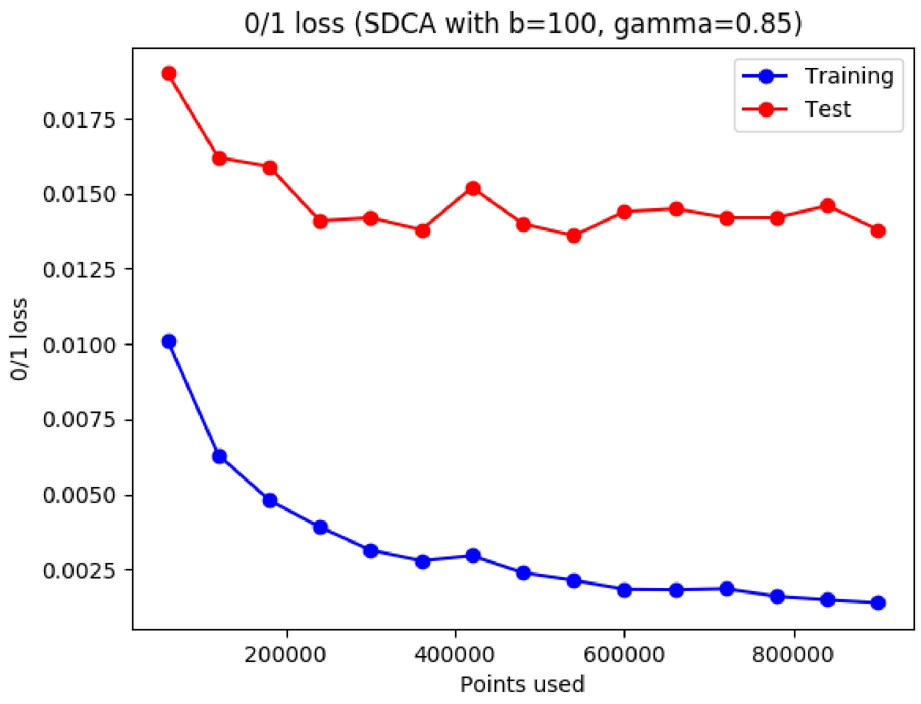


Figure 9: The classification error of SDCA with batch size 100 and $\gamma = 0.85$ on the training and test sets as a function of the epoch

	Square loss	**Classification loss**	**Total mistakes**
Training	0.044858	0.001400	84
Test	0.087752	0.013800	138

Table 5: Final losses of SDCA on the training and test sets using batch size $b = 100$ and $\gamma = 0.85$

4. In terms of the total number of data points visited, this version of batched SDCA performs much better than the version with $\gamma = 1/b$. Its performance, again with respect to examples used, is very similar to that of unbatched SDCA (compare Figure 9 with Figure 5). Both attain classification losses below 0.02 on the test set after only one pass through the data (60,000 points touched). If our goal were to run until a desired accuracy was obtained, this method would be the fastest of those studied. As noted before, batching decreases the time required to train for a fixed number of epochs. Since batching using $\gamma = 0.85$ produces as accurate a model as not batching, and does it faster, it is preferable in this case.