Policies:

Instructions: Completed assignments should be submitted via Canvas on time.

Coding: You must write your own code in this HW set. You may use any numerical linear algebra package, but you may not use machine learning libraries (e.g. sklearn, tensorflow). In addition, each student must write and submit their own code in the programming part of the assignment (we may run your code).

Acknowledgments: We expect you to make an honest effort to solve the problems individually. As we sometimes reuse problem set questions from previous years, covered by papers and webpages, we expect the students not to copy, refer to, or look at the solutions in preparing their answers (referring to unauthorized material is considered a violation of the honor code). Similarly, we expect to not to google directly for answers (though you are free to google for knowledge about the topic). If you do happen to use other material, it must be acknowledged here, with a citation on the submitted solution.

Required HW submission format:

The following is the required HW submission format: Submit all the answers to the HW as one single typed pdf document (not handwritten). This document must contain all plots. It is encouraged you latex all your work, though you may use another comparable typesetting method. Additionally, submit your code as a separate archive file (a .tar or .zip file). All code must be submitted. The code should be in a runnable file format like .py files or .txt files. Jupyter notebooks are not acceptable.

Readings:

Read the required material.

Collaborators:

In a separate section (before your answers), list the names of all people you collaborated with and for which question(s). If you did the HW entirely on your, please state this. Each student must understand, write, and hand in their own answers. Also, each student must understand, write, and hand in their own code.
1 (Dual) Coordinate Ascent [25 Points]

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:

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

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

1. (5 points) (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:

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

Now let us consider the coordinate ascent algorithm:

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

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

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

2. (7 points) Show that the solution to the inner optimization problem for $\alpha_i$ 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. (2 points) What is the computational complexity of this update, as it is stated? (Assume that scalar multiplication and scalar addition is order one.)

4. (1 points) 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:
    \[ \Delta \alpha_i = \frac{(y_i - w \cdot x_i) - \alpha_i}{1 + \|x_i\|^2 / \lambda} \]  
    \[ (1) \]
  - Update the parameters as follows:
    \[ \alpha_i \leftarrow \alpha_i + \Delta \alpha_i, \quad w \leftarrow w + \frac{1}{\lambda} x_i \cdot \Delta \alpha_i \]
    \[ (2) \]

  - 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. (7 points) Prove that the above update is valid.

6. (3 points) What is the computational complexity of the update, Equation 2 in the above implementation?

2 Let’s try it out! [40 Points]

Let’s try this out on MNIST, without “distortions” or “pre-processing”. The table in [http://yann.lecun.com/exdb/mnist/](http://yann.lecun.com/exdb/mnist/) shows which strategies use this pre-processing.

If we wanted even higher accuracy, we should really move to convolutional methods, which we will discuss later in the class.

**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:

\[
x \rightarrow (h_1(x), h_2(x), \ldots, h_k(x))
\]

We will use \( k = 30,000 \). 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 \( v_1, v_2, \ldots, v_k \), you will independently sample every coordinate for every vector from a standard normal distribution (with unit variance). Here, you can think of \( \sigma \) as a bandwidth parameter.

In practice, setting \( \sigma \) is often done with the 'median trick', which is the median of the pairwise distances (between the \( x \)'s) in your dataset. As this a scalar parameter, I often set \( \sigma \) by: First, I just randomly grab a few pairs of points and estimate the mean distance between a random pair of points (I often use the mean rather than the median). Second, I usually multiplicatively cut it down by some factor (maybe 2, 4, 8, ... depending on the problem).

*SK: You can read more about this at: https://people.eecs.berkeley.edu/~ brecht/kitchensinks.html. Technically, you should really use sin’s and cosine’s (or a random phase). For this problem, just sin’s works fine.*

*SK: For this problem, I found cutting the mean of \( X^2 \) down by a factor of 2 works well for the bandwidth.*

**Some Tips**

Let’s first optimize with SGD.

Some hints:

• (memory) As the dimension is large in this problem, we seek to avoid explicitly computing and storing the full data matrix, which is of size \( k \times N \). Instead, you can compute the feature vector \( h(x) \) ‘on the fly’, i.e. you recompute the vector \( h(x) \) whenever you access image \( x \).

• (learning rates, square loss case) With the square loss, I like to set my learning rates pretty large. I actually start by finding where things diverge (for the square loss, it really does diverge. for logistic, one will see that performance doesn’t improve if the rate is too large). Then I usually just cut this down roughly by some factor (maybe 2 or 4).
2.1 SGD and Averaging [15 points]

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 our 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 \( \tau \)-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. (1 point) Specify your parameter choices: your learning rate (or learning rate scheme) and the kernel bandwidth.

2. (5 points) 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. (5 points) 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. (2 points) 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. (2 points) 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.

*SK: Averaging often is reasonable even for non-convex optimization problems (provided you are averaging on one serial run, as opposed to computations done on multiple machines). If one mini-batches, the benefits of averaging often diminish.*
2.2 SDCA [25 points]

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 our 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. *(1 point)* Specify your regularization parameter. Are there any other parameter choices?
2. *(5 points)* Now plot the sum dual loss $G$ (where the sum is over all the classes). 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. *(4 points)* 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. *(4 points)* 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. *(4 points)* 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. *(6 points)* 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. *(1 point)* Compare the speed and accuracy to what you obtained with SGD?

3 Mini-batching (in the dual) and parallelization! [35 Points]

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 $\Delta_i$’s of these coordinates as specified in Equation 1.

Let us choose a batch size $b = 100$.

1. *(1 points)* One possibility is to just update the $\alpha_i$’s as specified in Equation 2. Is this ok?
2. (4 points) Argue that the following update will never increase the value of $G$:

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

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

3. (10 points) 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. (2 points) In terms of the total number of data points touched, does this perform notably better or worse than what you had before, with the $b = 1$?

SK: Note that while each iteration touches 100 points, it is not 100 time slower than using point for an iteration. This is due to that “for loops” are often costly and due to computer architecture choices.

Now let us improve upon this algorithm. Consider the update rule with parameter $\gamma$:

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

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

1. (3 points) Give an argument as to why choosing $\gamma = 1/b$ is pessimistic. (Either give a concise argument or give an example.)

2. (3 points) Now empirically (on our $b = 100$ experiments) search a little to find a good choice for gamma. What value of $\gamma$ did you choose? What value of $\gamma$ do things diverge?

3. (10 points) 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. (2 points) 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?

SK: Typically, one would not expect that mini-batching will improve upon the total number of data points touched. However, if this does number is not much worse, than you will find your code to be much faster, essentially due to the parallelization speedup, in that your computer is better optimized to do simultaneous matrix multiply operations.