1 Basics [16 Points]

Relevant materials: lecture 1

Answer each of the following problems with 1-2 short sentences.

Problem A [2 points]: What is a hypothesis set?

Solution A: *The hypothesis set is the set of all hypothesis that exist for a given problem.*

Problem B [2 points]: What is the hypothesis set of a linear model?

Solution B: *The hypothesis set of a linear model is all possible lines that interact with the data, of the form $w^T x$.*

Problem C [2 points]: What is overfitting?

Solution C: *Overfitting occurs when the machine learning model learns to match to the data directly, rather than learning. In overfitting, the model has a difficult time predicting testing data.*

Problem D [2 points]: What are two ways to prevent overfitting?

Solution D: *One way to prevent overfitting is more data. Another way is to simplify the model.*

Problem E [2 points]: What are training data and test data, and how are they used differently? Why should you never change your model based on information from test data?

Solution E: *Training and testing data are both data sets used to help the model reach optimization. Training data is data used solely to train the model, and test data is data used only to test the accuracy of the model. The test data should only be used to test the accuracy of the model, and should not have any effects on its training to prevent a biased model.*

Problem F [2 points]: What are the two assumptions we make about how our dataset is sampled?

Solution F: *We assume that our dataset is sampled independently and identically distributed.*

Problem G [2 points]: Consider the machine learning problem of deciding whether or not an email is spam. What could X, the input space, be? What could Y, the output space, be?

Solution G: *X, the input space, could be all emails put into some form of vector featuring the words of the emails. Y, the input space, could be whether the email is spam or not spam, representing in binary* $[0, 1]$.

Problem H [2 points]: What is the k-fold cross-validation procedure?

Solution H: *The procedure to k-fold cross-validation is to first split the data set into k subsets. Then, you train the model on $k-1$ of the subsets and test the model on the final subset, storing the validation error. You do this k times, so each of the subsets becomes the test data, and then average all of the validation errors.*

2 Bias-Variance Tradeoff [34 Points]

Relevant materials: lecture 1

Problem A [5 points]: Derive the bias-variance decomposition for the squared error loss function. That is, show that for a model f_S trained on a dataset S to predict a target $y(x)$ for each x,

$$\mathbb{E}_S\left[E_{\text{out}}\left(f_S\right)\right] = \mathbb{E}_x[\text{Bias}(x) + \text{Var}(x)]$$

given the following definitions:

$$\begin{aligned}
F(x) &= \mathbb{E}_S\left[f_S(x)\right] \\
E_{\text{out}}(f_S) &= \mathbb{E}_x\left[\left(f_S(x) - y(x)\right)^2\right] \\
\text{Bias}(x) &= \left(F(x) - y(x)\right)^2 \\
\text{Var}(x) &= \mathbb{E}_S\left[\left(f_S(x) - F(x)\right)^2\right]
\end{aligned}$$

Solution A:

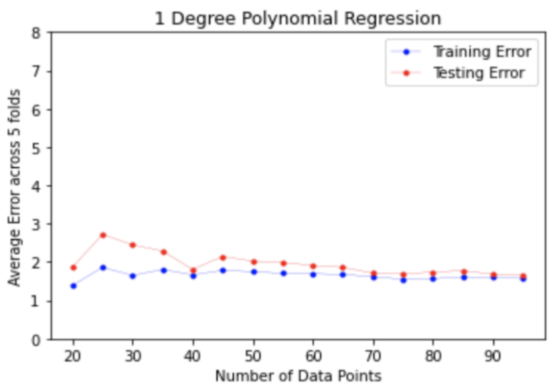
$$\begin{aligned}
E_{out}(f_S) &= \mathbb{E}_x\left[\ (f_S(x) - y(x))^2\right. \\
\mathbb{E}_S[E_{out}(f_S)] &= \mathbb{E}_S\left[\mathbb{E}_x\left[(\ f_S(x) - y(x))^2\right]\right] \\
&= \mathbb{E}_S[\mathbb{E}_x[(f_S(x) + F(x) - F(x) - y(x))^2]] \\
&= \mathbb{E}_S[\mathbb{E}_x[(f_S(x) - F(x))^2 + (F(x) - y(x))^2 + 2(f_S(x) - F(x)) + 2(F(x) - y(x))]] \\
&= \mathbb{E}_x[\mathbb{E}_S[(f_S(x) - F(x))^2 + (F(x) - y(x))^2 + 2(f_S(x) - F(x)) + 2(F(x) - y(x))]] \\
&= \mathbb{E}_x[\mathbb{E}_S[(f_S(x) - F(x))^2 + (F(x) - y(x))^2 + 2(f_S(x) - F(x))(F(x) - y(x))]] \\
&= \mathbb{E}_x[\mathbb{E}_S[(f_S(x) - F(x))^2] + \mathbb{E}_x[\mathbb{E}_S[(F(x) - y(x))^2] + \mathbb{E}_x[\mathbb{E}_S[2(f_S(x) - F(x))(F(x) - y(x))]] \\
&= \mathbb{E}_x[\mathbb{E}_S[(f_S(x) - F(x))^2] + \mathbb{E}_x[(F(x) - y(x))^2] + \\
&\qquad \mathbb{E}_x[2\mathbb{E}_S f_S(x)F(x) - 2F(x)^2 - 2\mathbb{E}_S f_S(x)y(x) + 2F(x)y(x)]] \\
&= \mathbb{E}_x[\mathbb{E}_S[(f_S(x) - F(x))^2] + \mathbb{E}_x[(F(x) - y(x))^2] + \mathbb{E}_x[2F(x) - 2F(x) + 2F(x)y(x) - 2F(x)y(x)]] \\
&= \mathbb{E}_x[\text{Var}(x)] + \mathbb{E}_x[\text{Bias}(x)] + 0 \\
&= \mathbb{E}_x[\text{Bias}(x) + \text{Var}(x)]
\end{aligned}$$

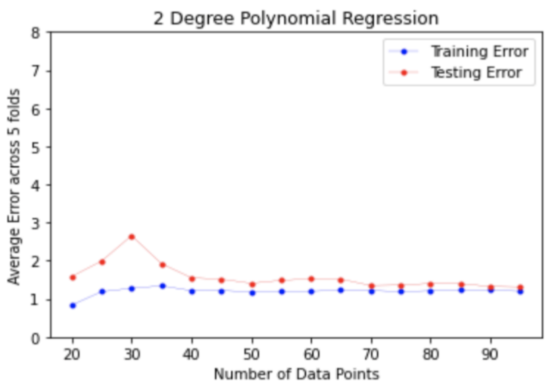
Problem B [14 points]: Use the provided `2_notebook.ipynb` Jupyter notebook to enter your code for this question. This notebook contains examples of using NumPy's polyfit and polyval methods, and scikit-learn's KFold method; you may find it helpful to read through and run this example code prior to continuing with this problem. Additionally, you may find it helpful to look at the documentation for scikit-learn's learning_curve method for some guidance.

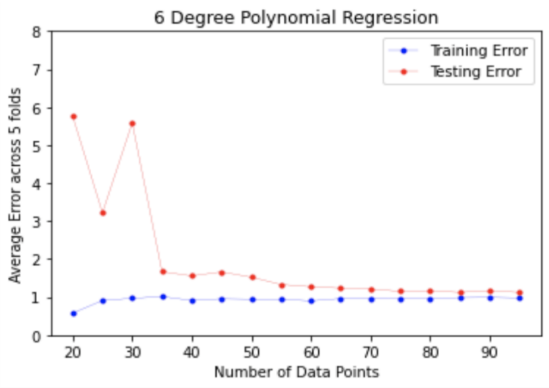
The dataset `bv_data.csv` is provided and has a header denoting which columns correspond to which values. Using this dataset, plot learning curves for 1st–, 2nd–, 6th–, and 12th–degree polynomial regression (4 separate plots) by following these steps for each degree $d \in \{1, 2, 6, 12\}$:

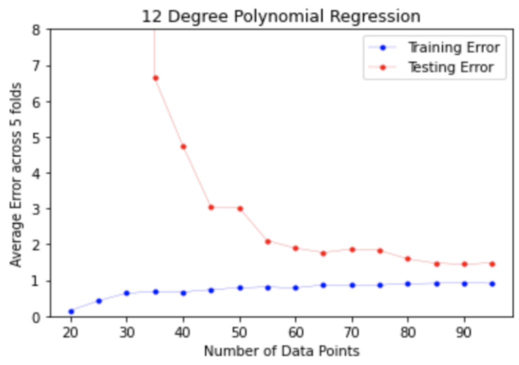
1. For each $N \in \{20, 25, 30, 35, \cdots, 100\}$:
   i. Perform 5-fold cross-validation on the first N points in the dataset (setting aside the other points), computing the both the training and validation error for each fold.
      - Use the mean squared error loss as the error function.
      - Use NumPy's polyfit method to perform the degree–d polynomial regression and NumPy's polyval method to help compute the errors. (See the example code and NumPy documentation for details.)
      - When partitioning your data into folds, although in practice you should randomize your partitions, for the purposes of this set, simply divide the data into K contiguous blocks.

   ii. Compute the average of the training and validation errors from the 5 folds.
2. Create a learning curve by plotting both the average training and validation error as functions of N. *Hint: Have same y-axis scale for all degrees d.*









Problem C [3 points]: Based on the learning curves, which polynomial regression model (i.e. which degree polynomial) has the highest bias? How can you tell?

Solution C: *The polynomial regression model of degree 1 has the highest bias. We can tell, since the degree 1 model had the highest average error, thus was unable to accurately fit the data.*

Problem D [3 points]: Which model has the highest variance? How can you tell?

Solution D: *The polynomial regression model of degree 12 has the highest variance. We can tell by the extremely large testing error for low numbers of data points, which eventually drops as N increases, revealing that with low data, the complex 12 degree model is morphing to exactly the data set.*

Problem E [3 points]: What does the learning curve of the quadratic model tell you about how much the model will improve if we had additional training points?

Solution E: *The learning curve of the quadratic model for training seems to maintain a similar mean squared error of about 1.2, but the testing error seems to be continuously decrease as the number of points increase. Thus, we can predict from the learning curve that if the model had additional training points, the model would generally do better.*

Problem F [3 points]: Why is training error generally lower than validation error?

Solution F: *Training error is generally lower than validation error because the training of the model is based on the training data, so the model will attempt to optimize on the training data. However, since the validation data is not used in training but only for testing, the model is not accustomed to the validation data, and will only use what it has learned from the training data on the validation data. Thus, the model will havea higher validation error than it will training error.*

Problem G [3 points]: Based on the learning curves, which model would you expect to perform best on some unseen data drawn from the same distribution as the training data, and why?

Solution G: *I would expect the 6th degree regression model to perform the best. The 6th degree model, for $n > 35$, outperforms every other model in minimizing testing error. Thus, the 6th degree model has the best generalization for unseen data and will be expected to perform the best.*

https://colab.research.google.com/drive/1kYQefWu0UUV3UNRiQRzuvVYslODo9w7W?usp=sharing

3 Stochastic Gradient Descent [36 Points]

Relevant materials: lecture 2

Stochastic gradient descent (SGD) is an important optimization method in machine learning, used everywhere from logistic regression to training neural networks. In this problem, you will be asked to first implement SGD for linear regression using the squared loss function. Then, you will analyze how several parameters affect the learning process.

Linear regression learns a model of the form:

$$f(x_1, x_2, \cdots, x_d) = \left(\sum_{i=1}^{d} w_i x_i\right) + b$$

Problem A [2 points]: We can make our algebra and coding simpler by writing $f(x_1, x_2, \cdots, x_d) = \mathbf{w}^T\mathbf{x}$ for vectors $\mathbf{w}$ and $\mathbf{x}$. But at first glance, this formulation seems to be missing the bias term b from the equation above. How should we define $\mathbf{x}$ and $\mathbf{w}$ such that the model includes the bias term?

Hint: *Include an additional element in* $\mathbf{w}$ *and* $\mathbf{x}$*.*

Solution A: *We can redefine w and x such that $w_0 = b$ and $x_0 = 1$.*

Linear regression learns a model by minimizing the squared loss function L, which is the sum across all training data $\{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)\}$ of the squared difference between actual and predicted output values:

$$L(f) = \sum_{i=1}^{N} (y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

Problem B [2 points]: SGD uses the gradient of the loss function to make incremental adjustments to the weight vector $\mathbf{w}$. Derive the gradient of the squared loss function with respect to $\mathbf{w}$ for linear regression.

Solution B:

$$L(f) = \sum_{i=1}^{N} (y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

$$\nabla_{\mathbf{w}} L(f) = \sum_{i=1}^{N} (-2\mathbf{x}_i(y_i - \mathbf{w}^T\mathbf{x}_i))$$

The following few problems ask you to work with the first of two provided Jupyter notebooks for this problem, `3_notebook_part1.ipynb`, which includes tools for gradient descent visualization. This notebook utilizes the files `sgd_helper.py` and `multiopt.mp4`, but you should not need to modify either of these files.

For your implementation of problems C-E, **do not** consider the bias term.

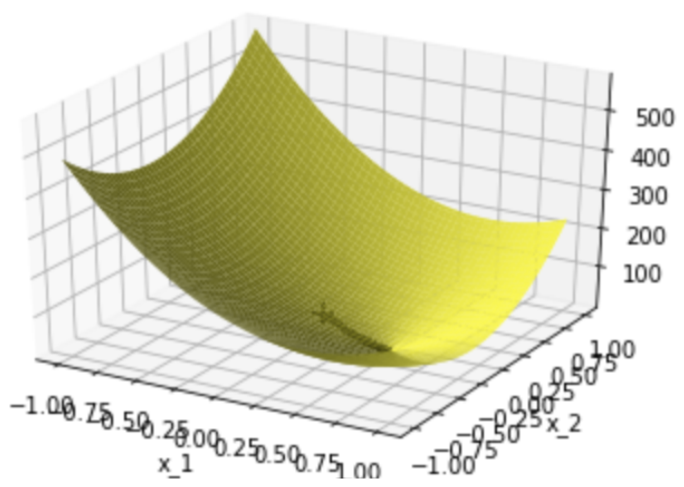
Problem C [8 points]: Implement the `loss`, `gradient`, and `SGD` functions, defined in the notebook, to perform SGD, using the guidelines below:

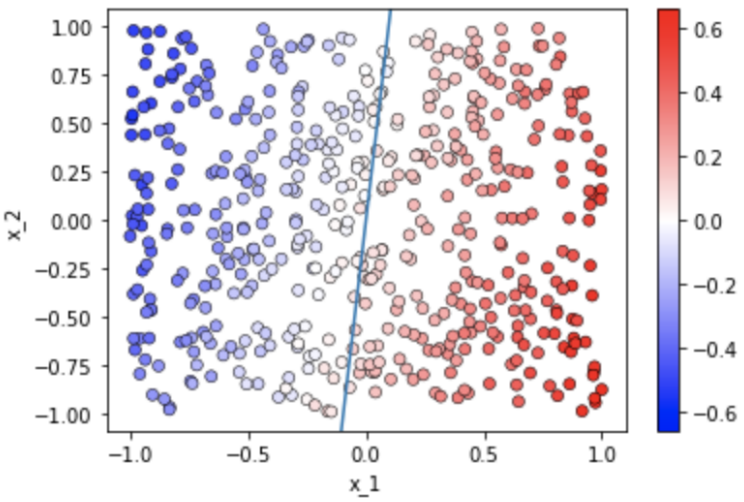
- Use a squared loss function.
- Terminate the SGD process after a specified number of epochs, where each epoch performs one SGD iteration for each point in the dataset.
- It is recommended, but not required, that you shuffle the order of the points before each epoch such that you go through the points in a random order. You can use `numpy.random.permutation`.
- Measure the loss after each epoch. Your `SGD` function should output a vector with the loss after each epoch, and a matrix of the weights after each epoch (one row per epoch). Note that the weights from all epochs are stored in order to run subsequent visualization code to illustrate SGD.

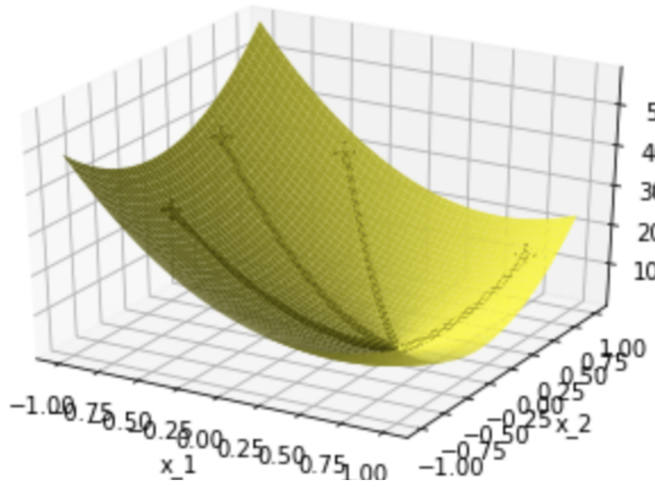
Solution C: *See code.*

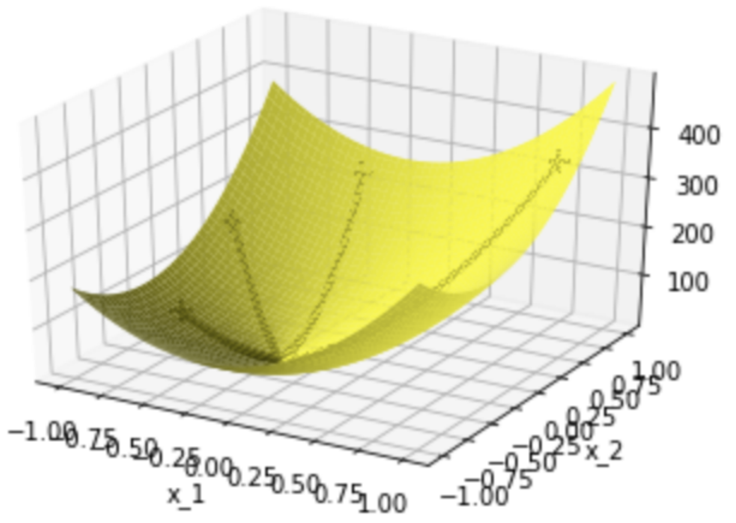
Problem D [2 points]: Run the visualization code in the notebook corresponding to problem D. How does the convergence behavior of SGD change as the starting point varies? How does this differ between datasets 1 and 2? Please answer in 2-3 sentences.

Solution D: *Even as the starting point varies, points still converge to the same location for the SGD. For points where they started further from the point of convergence, they seemed to converge to the point at the same time, thus the rate of convergence was higher for these points. This is true for both datasets 1 and 2, but there is a different point of convergence for each dataset.*



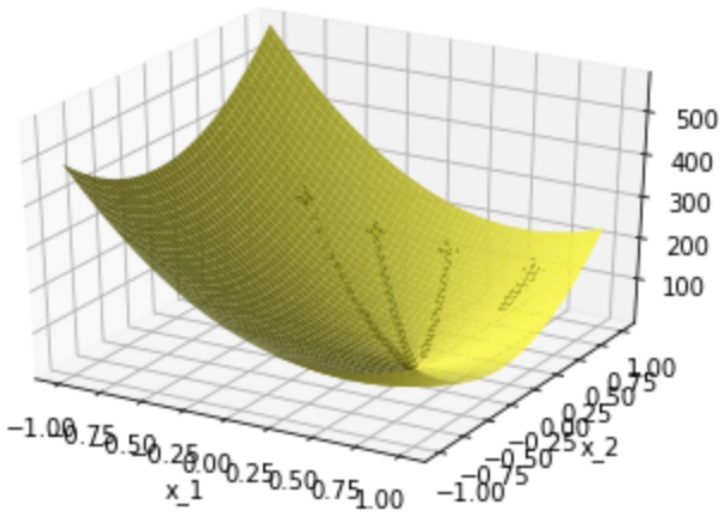

x_1
x_2
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

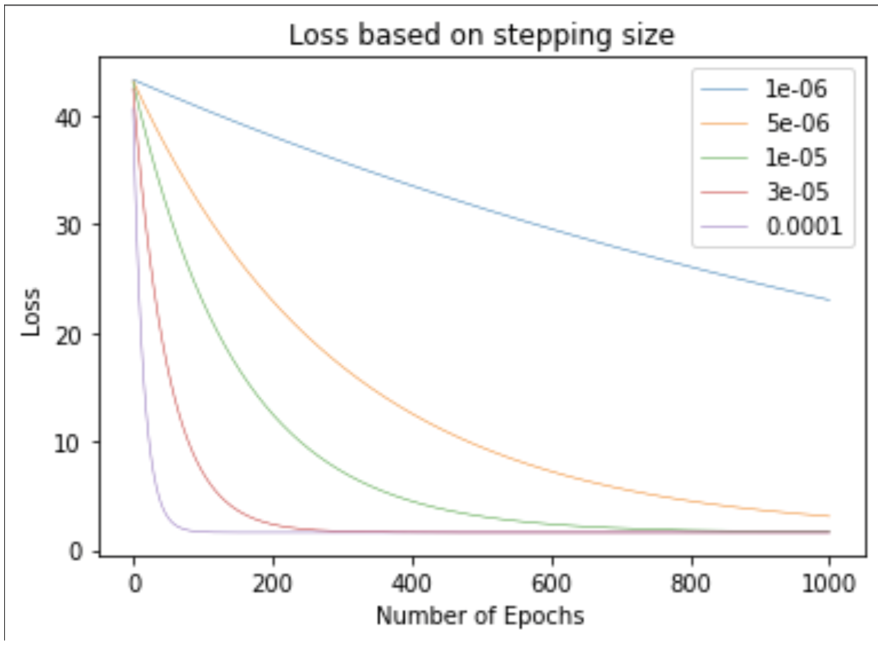

x_1
x_2
500
400
300
200
100

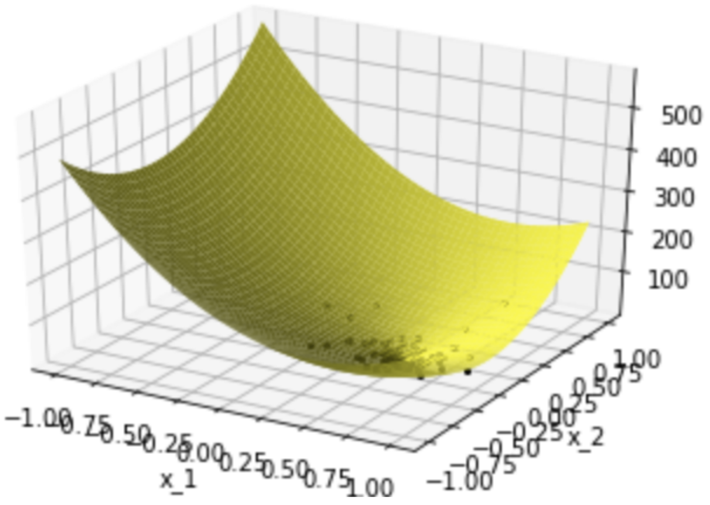


Problem E [6 points]: Run the visualization code in the notebook corresponding to problem E. One of the cells—titled "Plotting SGD Convergence"—must be filled in as follows. Perform SGD on dataset 1 for each of the learning rates $\eta \in \{$1e-6, 5e-6, 1e-5, 3e-5, 1e-4$\}$. On a single plot, show the training error vs. number of epochs trained for each of these values of η. What happens as η changes?

Solution E: *As η increases, the loss converges to 0 quicker.*







https://colab.research.google.com/drive/1wP92VnQ0ktUSh7YvYAMXZrEicnH8P9IZ?usp=sharing

The following problems consider SGD with the larger, higher-dimensional dataset, `sgd_data.csv`. The file has a header denoting which columns correspond to which values. For these problems, use the Jupyter notebook `3_notebook_part2.ipynb`.

For your implementation of problems F-H, **do** consider the bias term using your answer to problem A.

Problem F [6 points]: Use your SGD code with the given dataset, and report your final weights. Follow the guidelines below for your implementation:

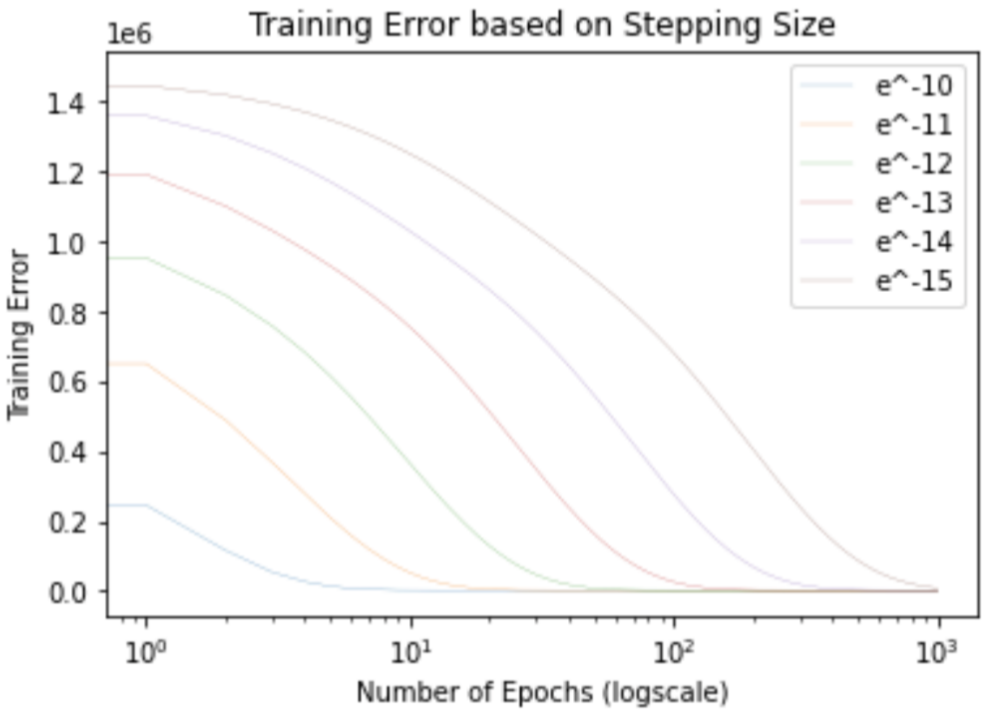
- Use $\eta = e^{-15}$ as the step size.
- Use $\mathbf{w} = [0.001, 0.001, 0.001, 0.001]$ as the initial weight vector and $b = 0.001$ as the initial bias.
- Use at least 800 epochs.
- You should incorporate the bias term in your implementation of SGD and do so in the vector style of problem A.
- Note that for these problems, it is no longer necessary for the `SGD` function to store the weights after all epochs; you may change your code to only return the final weights.

Solution F: $[-0.22790041, -5.97855428, 3.98836539, -11.85702688, 8.91127797]$

Problem G [2 points]: Perform SGD as in the previous problem for each learning rate η in

$$\{e^{-10}, e^{-11}, e^{-12}, e^{-13}, e^{-14}, e^{-15}\},$$

and calculate the training error at the beginning of each epoch during training. On a single plot, show training error vs. number of epochs trained for each of these values of η. Explain what is happening.



Solution G: *For all of the step sizes, the training error eventually converges to* 0*. Again, the for smaller values of* η*, the training error converges to* 0 *slower, while for larger values of* η*, it converges quicker to* 0*.*

Problem H [2 points]: The closed form solution for linear regression with least squares is

$$\mathbf{w} = \left(\sum_{i=1}^{N} \mathbf{x_i}\mathbf{x_i}^T\right)^{-1} \left(\sum_{i=1}^{N} \mathbf{x_i} y_i\right).$$

Compute this analytical solution. Does the result match up with what you got from SGD?

Solution H: $[-0.31644251, -5.99157048, 4.01509955, -11.93325972, 8.99061096]$. *This closed form analytical solution for linear regression is very similar to the final weight vector we got from our SGD.*

Answer the remaining questions in 1-2 short sentences.

Problem I [2 points]: Is there any reason to use SGD when a closed form solution exists?

Solution I: *One reason to use SGD is because it is computationally cheaper; the closed form solution is very expensive in the calculations of the matrices and inverting the matrices.*

Problem J [2 points]: Based on the SGD convergence plots that you generated earlier, describe a stopping condition that is more sophisticated than a pre-defined number of epochs.

Solution J: *Rather than a pre-defined number of epochs, we can implement a stopping condition which stops the for loop when the training error drops below a specified* ϵ*.*

Problem K [2 points]: How does the convergence behavior of the weight vector differ between the perceptron and SGD algorithms?

Solution K: *The convergence behavior of the weight vector of the perceptron will start in a general defined starting position and then slowly adjust until it meets the point of convergence. In SGD, the convergence behavior of the weight vector starts on either side of the point of convergence and will slide towards it. SGD is also much smoother in approaching the point of convergence, while in the perceptron it depends a lot on what misclassified point is chosen.*

https://colab.research.google.com/drive/1uOW4Ox5GOAYL22JGAD2UzdVVfiNGxzKT?usp=sharing

4 The Perceptron [14 Points]

Relevant materials: lecture 2

The perceptron is a simple linear model used for binary classification. For an input vector $\mathbf{x} \in \mathbb{R}^d$, weights $\mathbf{w} \in \mathbb{R}^d$, and bias $b \in \mathbb{R}$, a perceptron $f : \mathbb{R}^d \to \{-1, 1\}$ takes the form

$$f(\mathbf{x}) = \text{sign}\left(\left(\sum_{i=1}^{d} w_i x_i\right) + b\right)$$

The weights and bias of a perceptron can be thought of as defining a hyperplane that divides $\mathbb{R}^d$ such that each side represents an output class. For example, for a two dimensional dataset, a perceptron could be drawn as a line that separates all points of class $+1$ from all points of class -1.

The PLA (or the Perceptron Learning Algorithm) is a simple method of training a perceptron. First, an initial guess is made for the weight vector $\mathbf{w}$. Then, one misclassified point is chosen arbitrarily and the $\mathbf{w}$ vector is updated by

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + y(t)\mathbf{x}(t) \\ b_{t+1} &= b_t + y(t), \end{aligned}$$

where $\mathbf{x}(t)$ and $y(t)$ correspond to the misclassified point selected at the t^{th} iteration. This process continues until all points are classified correctly.

The following few problems ask you to work with the provided Jupyter notebook for this problem, titled `4_notebook.ipynb`. This notebook utilizes the file `perceptron_helper.py`, but you should not need to modify this file.

Problem A [8 points]:

Implement the `update_perceptron` and `run_perceptron` methods in the notebook, and perform the perceptron algorithm with initial weights $w_1 = 0, w_2 = 1, b = 0$.

Give your solution in the form a table showing the weights and bias at each timestep and the misclassified point $([x_1, x_2], y)$ that is chosen for the next iteration's update. You can iterate through the three points in any order. Your code should output the values in the table below; cross-check your answer with the table to confirm that your perceptron code is operating correctly.

t	b	w_1	w_2	x_1	x_2	y
0	0	0	1	1	-2	+1
1	1	1	-1	0	3	+1
2	2	1	2	1	-2	+1
3	3	2	0			

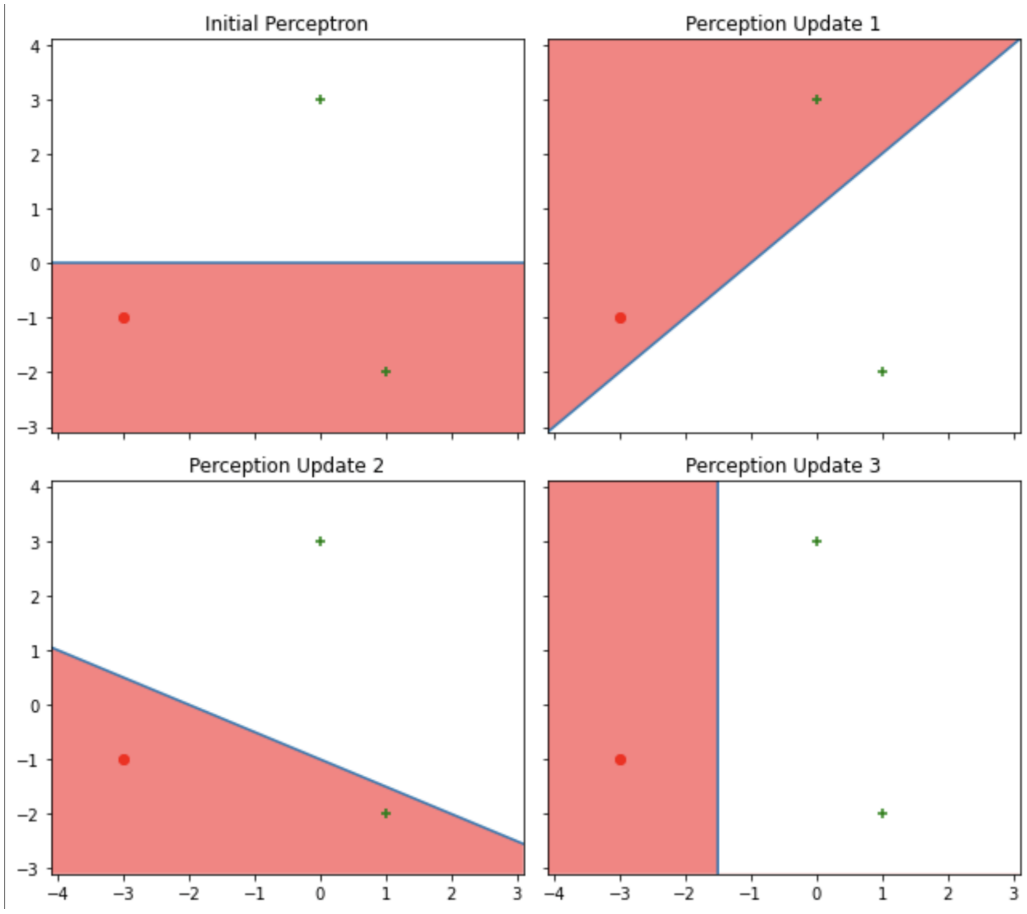
Include in your report both: the table that your code outputs, as well as the plots showing the perceptron's classifier at each step (see notebook for more detail).

Solution A:

t	w_1	w_2	b	x_1	x_2	y
0	0	1	0	1	-2	+1
1	1	-1	1	0	3	+1
2	1	2	2	1	-2	+1
3	2	0	3			

final w = [2. 0.], final b = 3.0

The table my code outputs matches the table above. Below are the plots showing the perceptron's classifier at each step.



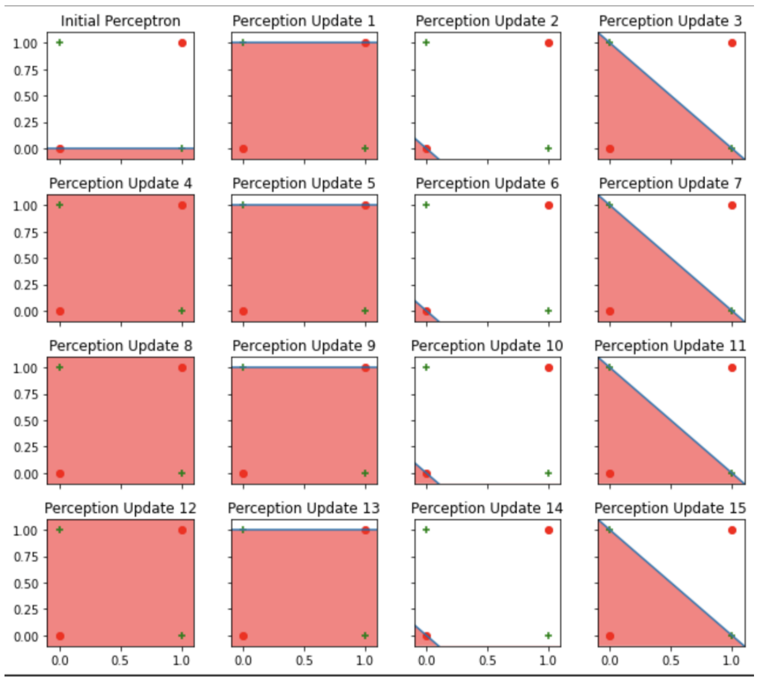
Problem B [4 points]: A dataset $S = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)\} \subset \mathbb{R}^d \times \mathbb{R}$ is *linearly separable* if there exists a perceptron that correctly classifies all data points in the set. In other words, there exists a hyperplane that separates positive data points and negative data points.

In a 2D dataset, how many data points are in the smallest dataset that is not linearly separable, such that no three points are collinear? How about for a 3D dataset such that no four points are coplanar? Please limit your solution to a few lines - you should justify but not prove your answer.

Finally, how does this generalize for an N-dimensional set, in which **no** $<N$-dimensional hyperplane contains a non-linearly-separable subset? For the N-dimensional case, you may state your answer without proof or justification.

Solution B: *In a 2D dataset, the smallest dataset that is not linearly separable in which no three points are collinear is 4 data points. Creating a rectangular plot with opposite corners being coordinates that are marked the same, we will never be able to linearly separate them. This is shown in Problem 4C. For a 3D dataset, the smallest dataset such that no 4 fours are coplanar is 5 data points. If arranged to be the vertices of a triangular bipyramid, it is ensured that no 4 points are on the same plane. Generalizing to the N-Dimensional case, a dataset of size* $N + 2$ *cannot be separated by any* $< N$*-dimensional hyperplane.*

Problem C [2 points]: Run the visualization code in the Jupyter notebook section corresponding to question C (report your plots). Assume a dataset is *not* linearly separable. Will the Perceptron Learning Algorithm ever converge? Why or why not?



Solution C: *If the dataset is not linearly separable, then a Perceptron Learning Algorithm attempting to linearly separate that dataset will never converge. In order for the PLA to converge, there must be no more misclassified points. However, since the dataset is not linearly separable, there will always be a misclassified point. Thus, the PLA will never converge. This is clearly show in the image above, where the PLA continuously loops through the same weight and bias.*

https://colab.research.google.com/drive/1pSmawk5ePCwaSgnV2OcKthpfv0pso12W?usp=sharing