

## Policies

- Due 5 PM PST, January 13<sup>th</sup> on Gradescope.
- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- If you have trouble with this homework, it may be an indication that you should drop the class.
- In this course, we will be using Google Colab for code submissions. You will need a Google account.
- You are allowed to use up to a total of 48 late hours throughout the term. Late hours must be used in units of whole hours. Specify the total number of hours you have ever used when turning in the assignment.
- **No use of large language models is allowed.** Students are expected to complete homework assignments based on their understanding of the course material.

## Submission Instructions

- Submit your report as a single .pdf file to Gradescope (entry code 2P8P28), under "Set 1 Report".
- In the report, **include any images generated by your code** along with your answers to the questions.
- Submit your code by **sharing a link in your report** to your Google Colab notebook for each problem (see naming instructions below). Make sure to set sharing permissions to at least "Anyone with the link can view". **Links that can not be run by TAs will not be counted as turned in.** Check your links in an incognito window before submitting to be sure.
- For instructions specifically pertaining to the Gradescope submission process, see [https://www.gradescope.com/get\\_started#student-submission](https://www.gradescope.com/get_started#student-submission).

## Google Colab Instructions

For each notebook, you need to save a copy to your drive.

1. Open the github preview of the notebook, and click the icon to open the colab preview.
2. On the colab preview, go to File → Save a copy in Drive.
3. Edit your file name to "lastname\_firstname\_originaltitle", e.g. "yue\_yisong\_3\_notebook\_part1.ipynb"

## 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:** *A hypothesis set is a set of all hypotheses that can be learned by a machine learning algorithm.*

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

**Solution B:** *The hypothesis set of a linear model is all hypotheses in the form  $\omega^T x - b$ .*

**Problem C [2 points]:** What is overfitting?

**Solution C:** *Overfitting is when the training error is much larger than the testing error.*

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

**Solution D:** *Two ways to prevent overfitting are to reduce the model complexity and to train on a larger amount of data.*

**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 data is the data that the model uses to learn how to map certain inputs to their correct outputs while the testing data is used to test the accuracy of this learning. You should never change the model based on information from the test data since the purpose of the testing data is to test the true accuracy of the model. If the testing data was introduced to the model beforehand, the model would already know how to map its inputs to the respective output.*

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

**Solution F:** *Two assumptions that we make about our dataset is that it is sampled identically and independently from the true distribution.*

**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:** *In this particular problem,  $X$  could be the subject of an email and  $Y$  could be the classification of "spam" or "not spam" with +1 being spam and -1 being not spam.*

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

**Solution H:** *The  $k$ -fold cross-validation procedure is when the dataset given is split up evenly into  $k$  chunks. The model is then given  $k - 1$  of those chunks to train on while using the last chunk to test. This process is repeated such that every chunk ends up being a testing chunk.*

## 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  $x$ ,

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

given the following definitions:

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

**Solution A:**

#2 Bias-Variance Tradeoff

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

In the following problems you will explore the bias-variance tradeoff by producing learning curves for polynomial regression models.

A *learning curve* for a model is a plot showing both the training error and the cross-validation error as a function of the number of points in the training set. These plots provide valuable information regarding the bias and variance of a model and can help determine whether a model is over- or under-fitting.

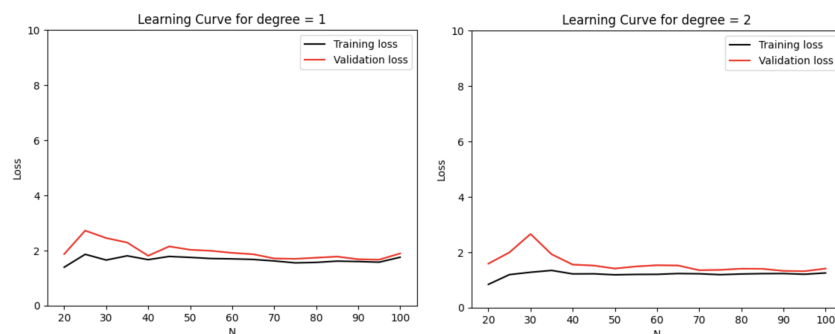
*Polynomial regression* is a type of regression that models the target  $y$  as a degree- $d$  polynomial function of the input  $x$ . (The modeler chooses  $d$ .) You don't need to know how it works for this problem, just know that it produces a polynomial that attempts to fit the data.

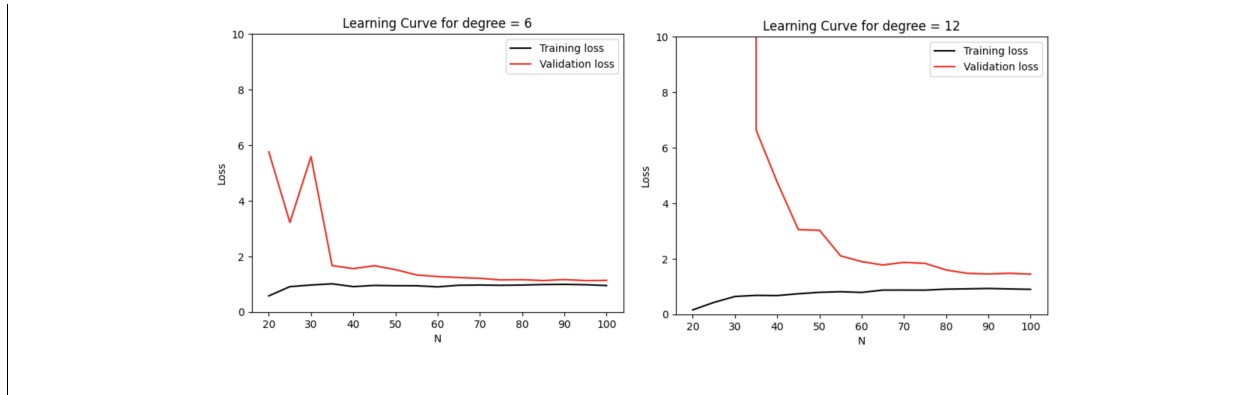
**Problem B [14 points]:** Use the provided `2_notebook.ipynb` Jupyter notebook to enter your code for this question. This notebook contains examples of using NumPy's `polyfit` and `polyval` methods, and scikit-learn's `KFold` method; you may find it helpful to read through and run this example code 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  $N \in \{20, 25, 30, 35, \dots, 100\}$ :
  - 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  $d$ .*

**Solution B:** code: [mantripragada\\_ishaan\\_2\\_notebook.ipynb](#)





**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 with degree 1 seems to have the highest bias because the training error and validation error indicate underfitting.*

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

**Solution D:** *The polynomial with degree 12 seems to have the highest variance because it has the largest difference between the training error and validation error which may indicate overfitting.*

**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 tells us that even with additional training data points ( $\geq 40$ ), its performance will not significantly improve.*

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

**Solution F:** *The training error is generally lower than the validation error because the model learns based on the training data. The validation data is completely hidden from the model, so when it tries to make predictions, it will tend to have a higher error than the training data.*

**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 model with degree 6 to perform the best because it seems to have the lowest difference between the validation and training error without showing signs of overfitting.*

### 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:

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

**Problem A [2 points]:** We can make our algebra and coding simpler by writing  $f(x_1, x_2, \dots, x_d) = \mathbf{w}^T \mathbf{x}$  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  $\mathbf{x}$ .

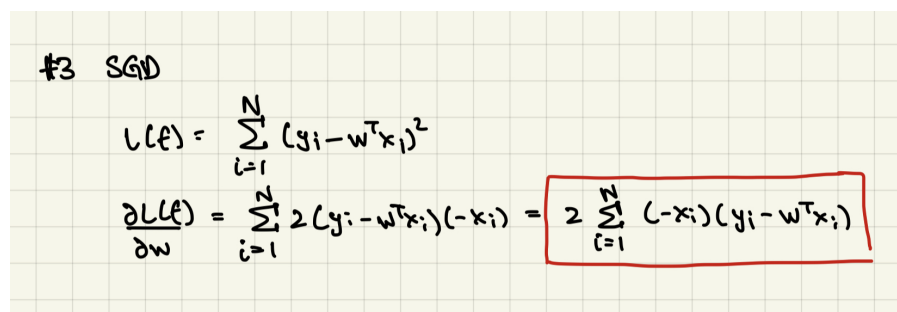
**Solution A:** We can include an additional element,  $\mathbf{w}^0 \mathbf{x}$  such that  $\mathbf{x} = 1$ . Therefore, we are left with a  $\mathbf{w}^0$  term that represents the bias.

Linear regression learns a model by minimizing the squared loss function  $L$ , which is the sum across all training data  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$  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:**



Handwritten solution for Problem B:

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

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



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:** code: [\*mantripragada\\_ishaan\\_3\\_notebook\\_part1.ipynb\*](#)

**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, the convergence behavior stays as a direct line to the minimum point. The same convergence pattern is shown for both datasets. The difference between them is that each point takes its own convergence path to the minimum instead of a set way.*

**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  $\eta$ . What happens as  $\eta$  changes?

**Solution E:** *As the learning rate gets larger, the model converges faster.*

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:** code: [mantripragada\\_ishaan\\_3\\_notebook\\_part2.ipynb](#)

Final weights: `[-0.22791757, -5.9787249, 3.98816869, -11.85719653, 8.91110894]`

**Problem G [2 points]:** Perform SGD as in the previous problem for each learning rate  $\eta$  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  $\eta$ . Explain what is happening.

**Solution G:** *As the learning rate gets smaller, the convergence takes more epochs to converge. This is because with smaller step sizes, the convergence process takes longer.*

**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:** *The result is  $[-0.31644251, -5.99157048, 4.01509955, -11.93325972, 8.99061096]$ . This does match up with my weights from 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:** *Yes, even if a closed form solution exists, SGD can still prove useful for larger and more complex datasets.*

**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:** *We can define some threshold value such that if the training error drops below that threshold, we stop the training. This is more sophisticated than a pre-defined number of epochs as it saves time and may prevent instances of overfitting.*

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

**Solution K:** *For SGD, the convergence behavior is gradual. The loss will continue to decrease until convergence. For the perceptron, the loss fluctuates back and forth until convergence.*

## 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  $f : \mathbb{R}^d \rightarrow \{-1, 1\}$  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  $t^{\text{th}}$  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  $w_1 = 0, w_2 = 1, b = 0$ .

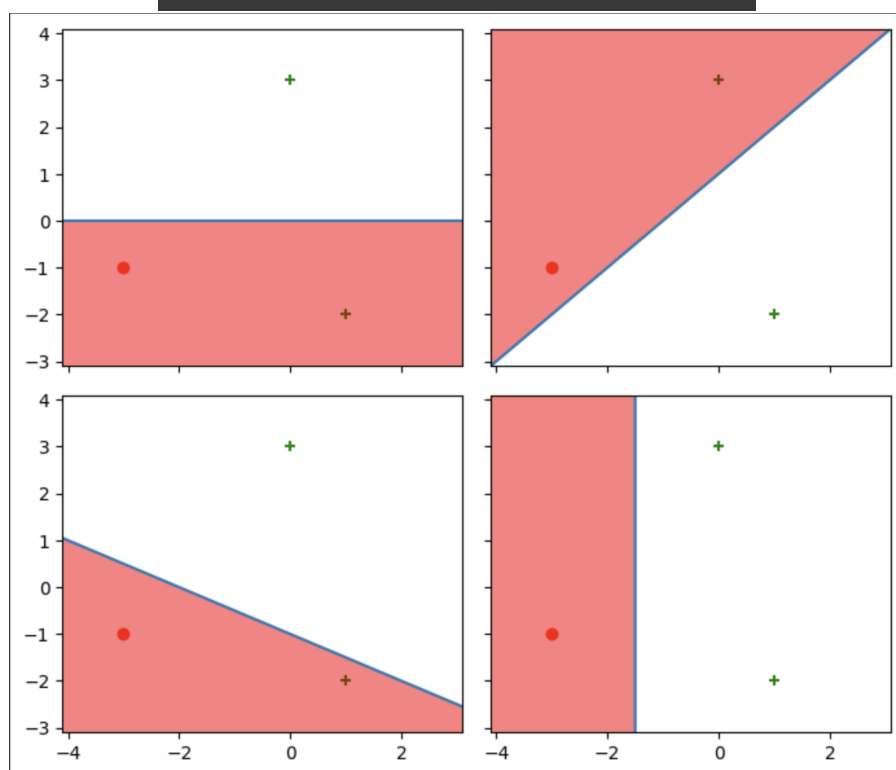
Give your solution in the form a table showing the weights and bias at each timestep and the misclassified point  $([x_1, x_2], y)$  that is chosen for the next iteration's update. You can iterate through the three points in any order. Your code should output the values in the table below; cross-check 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:** code: [mantripragada\\_ishaan\\_4\\_notebook.ipynb](#)

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



**Problem B [4 points]:** A dataset  $S = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\} \subset \mathbb{R}^d \times \mathbb{R}$  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:** *4 data points are in the smallest 2D dataset that is not linearly separable. If the points are arranged in a quadrilateral configuration with one set of opposite corners  $b$  as  $-1$  while the other set of corners is  $1$ , no separation is possible. In a 3D dataset, 5 points is the smallest amount of points that is not linearly separable. In this example if four of the points are arranged as a tetrahedron with the fifth point inside, such that alternating labels are used for the corners, no single plane can separate the positives and negatives. For  $N$  dimensions, a  $N+2$  points is the smallest number of points that would not be linearly separable.*

**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:** *The Perceptron Learning Algorithm will never converge because there will always be at least one point that is misclassified. This would cause the perceptron to continue to fluctuate back and forth without reaching convergence.*