Machine Learning & Data Mining
Set 1

Caltech CS/CNS/EE 155
January 6^{th}, 2023

Policies

- Due 9 PM PST, January 13^{th} 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.

Submission Instructions

- Submit your report as a single .pdf file to Gradescope (entry code K3RPGE), 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.

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 $\rightarrow$ 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: *The set of functions that our training algorithm can produce as output.*

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

Solution B: *The set of all possible linear functions of inputs. For example* $f(x_1, x_2, \dots) = w_0 + w_1 x_1 + w_2 x_2 + \dots$ *where* x_i *are the inputs and* w_i *are arbitrary constants.*

Problem C [2 points]: What is overfitting?

Solution C: *Overfitting occurs when out-of-sample error increases while in-sample error decreases, usually when too few samples are used relative to the size of the hypothesis set.*

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

Solution D: *Use more training samples, or restrict the hypothesis set (for example by using regularization).*

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 used to produce/pick a hypothesis, and test data is used to evaluate the hypothesis. You shouldn't change your model based on test data because the purpose of testing is to estimate performance outside the training set, and testing on data used in training would not be representative of that.*

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

Solution F: *We assume that our dataset (training and test) is sampled identically and independently from the same distribution of inputs we will see in the real world.*

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: *The input space could be a "bag of words" vector, with a 0 or 1 indicating whether or not each word in the dictionary appears in the email. The output space could be real numbers from 0 to 1 indicating probability that the email is spam.*

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

Solution H: *Divide the training/test data into k sections. Train k models, where the i-th model trains on every section but i and tests on i. This allows us to effectively use all of our data for testing while avoiding the problem of training on test data.*

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:

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

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

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.

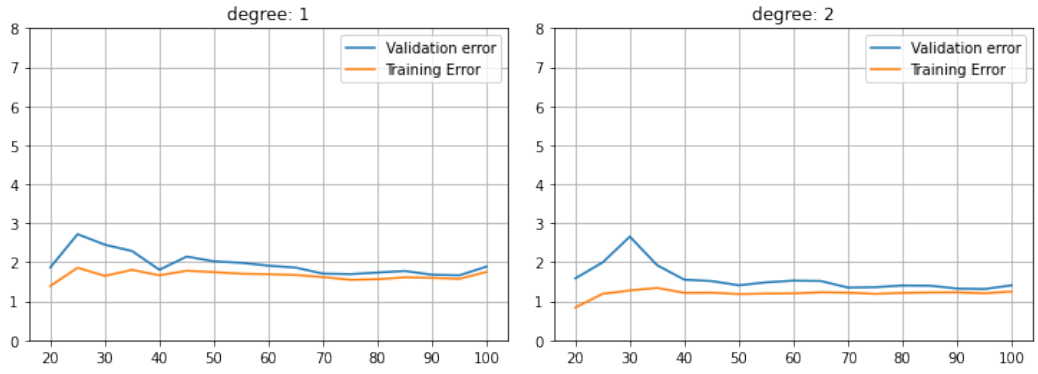
Polynomial regression is a type of regression that models the target y as a degree–d polynomial function of the input x. (The modeler chooses d.) You don't need to know how it works for this problem, just know that it produces a polynomial that attempts to fit the data.

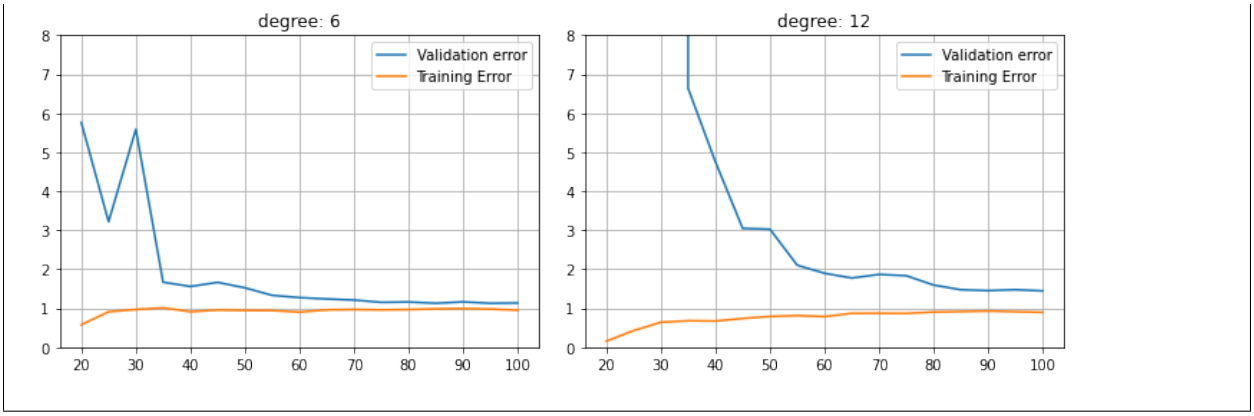
Problem B [14 points]: Use the provided `2_notebook.ipynb` Jupyter notebook to enter your code for this question. This notebook contains examples of using NumPy's polyfit and polyval methods, and scikit-learn's KFold method; you may find it helpful to read through and run this example code prior to continuing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

Solution B: *Colab notebook*





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 1st degree polynomial because its validation error is the largest for large N.*

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

Solution D: *The 12th degree polynomial, because its training error is the most different from its validation error.*

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 training error and validation error are already close together and flat at $N = 100$, so additional training points probably won't help.*

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

Solution F: *The learning algorithm optimizes to make training error as low as possible, but it doesn't optimize for validation error directly since validation is done with different 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: *The 6th degree polynomial (with $N = 100$), since it has the lowest validation error.*

3 Stochastic Gradient Descent [34 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 analyze gradient descent and implement SGD for linear regression using the squared loss function. Then, you will analyze how several parameters affect the learning process.

Problem A [3 points]: To verify the convergence of our gradient descent algorithm, consider the task of minimizing a function f (assume that f is continuously differentiable). Using Taylor's theorem, show that if x' is a local minimum of f, then $\nabla f(x') = 0$.

Hint: *First-order Taylor expansion gives that for any $x, h \in \mathbb{R}^n$, there exists $c \in (0,1)$ such that $f(x+h) = f(x) + \nabla f(x + c \cdot h)^T h$.*

Solution A: *If x' is a local minimum of f, then there is some $\varepsilon > 0$ such that $f(x+h) \geq f(x)$, so $f(x'+h) - f(x') \geq 0$ for $|h| < \varepsilon$. Dividing by $|h|$ gives $\frac{f(x'+h)-f(x')}{|h|} \geq 0$ for $|h| > 0$.*

Taylor's theorem gives us $f(x'+h) = f(x') + \nabla f(x' + c \cdot h)^T h$, where $0 < c < 1$, for any $h \in \mathbb{R}^n$. We can rewrite as $\frac{f(x'+h)-f(x')}{|h|} = \nabla f(x' + c \cdot h)^T \hat{h}$ for $|h| > 0$, where $\hat{h} = h/|h|$.

For $0 < |h| < \varepsilon$ we can combine both statements to get

$$\nabla f(x' + c \cdot h)^T \hat{h} \geq 0 \qquad (*)$$

Taking the limit of (*) as $|h| \to 0$ gives $\nabla f(x') \geq 0$.

Plugging $-h$ into (*) gives $-\nabla f(x' - c \cdot h)^T \hat{h} \geq 0$, so $\nabla f(x' - c \cdot h)^T \hat{h} \leq 0$. Taking this limit as $|h| \to 0$ gives $\nabla f(x') \leq 0$.

Thus $\nabla f(x') = 0$. □

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 B [1 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 B:

$$\mathbf{x} = (1, x_1, x_2, \dots)$$
$$\mathbf{w} = (b, w_1, w_2, \dots)$$

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 C [2 points]: Both GD and 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. Explain the difference in computational complexity in 1 update of the weight vector between GD and SGD.

Solution C:

$$\begin{aligned}
\nabla L &= \nabla \sum_{i=1}^{N} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \\
&= \sum_{i=1}^{N} \nabla (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \\
&= \sum_{i=1}^{N} 2(y_i - \mathbf{w}^T \mathbf{x}_i) \nabla (y_i - \mathbf{w}^T \mathbf{x}_i) \\
&= \sum_{i=1}^{N} -2(y_i - \mathbf{w}^T \mathbf{x}_i) \mathbf{x}_i
\end{aligned}$$

GD uses the average (or sum) of the loss function over all data points on every step, whereas SGD uses the loss function at a single random data point. Therefore GD does roughly N times more computations than SGD for one update.

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 D-F, **do not** consider the bias term.

Problem D [6 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.

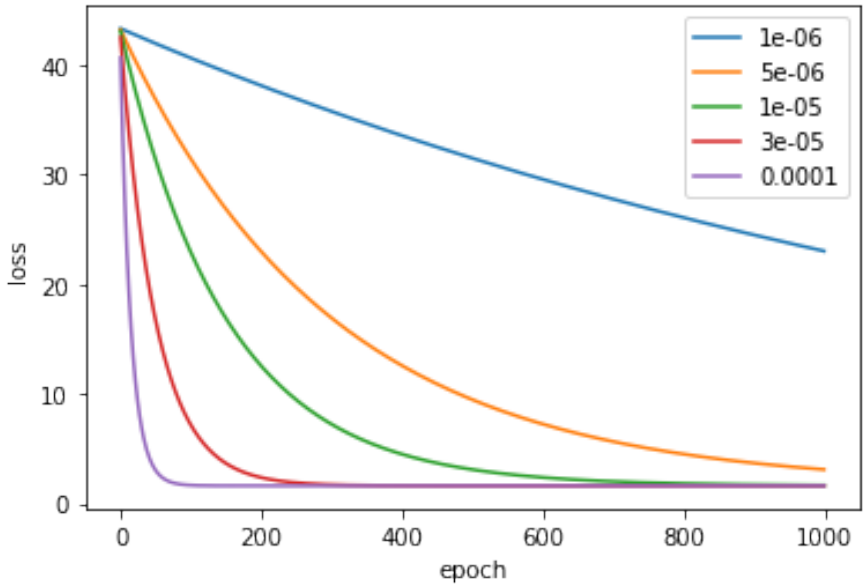
Solution D: *Colab notebook*

Problem E [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 E: *For both datasets, SGD converges at about the same rate from every starting point. This makes sense because the starting points farther from the minimum have a steeper gradient, so they'll be moved faster by SGD.*

Problem F [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 $\eta \in \{10^{-6}, 5 \cdot 10^{-6}, 10^{-5}, 3 \cdot 10^{-5}, 10^{-4}\}$. 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 F:



Convergence gets faster as η increases.

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 G-I, **do** consider the bias term using your answer to problem A.

Problem G [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 G: *Colab notebook*

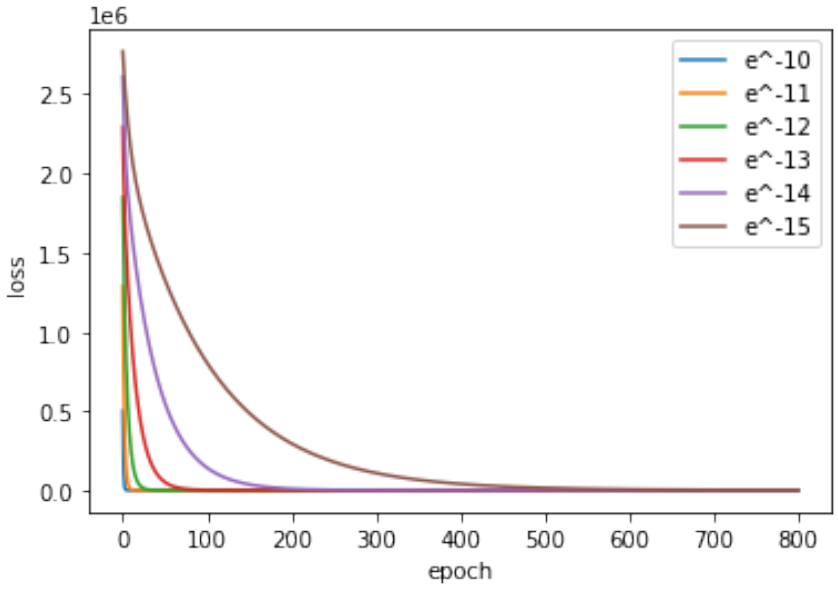
Final weights (starting with bias term): $\{-0.22789155, -5.9785225, 3.98840241, -11.85699312, 8.9113101\}$

Problem H [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 H:



As seen in 3F, SGD converges faster for higher learning rates.

Problem I [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 I: *Analytical solution:* $\{-0.31644251, -5.99157048, 4.01509955, -11.93325972, 8.99061096\}$

This is very close to what I got from SGD, and it has slightly lower loss (4071 vs 4176).

Answer the remaining questions in 1-2 short sentences.

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

Solution J: *For very large data sets, computing the closed form solution is too costly. SGD iteratively approaches the optimal weights rather than computing them directly, so we can stop once we're close enough.*

Problem K [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 K: *You could stop when the difference in loss from the previous epoch falls below some threshold.*

4 The Perceptron [16 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

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

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]: The graph below shows an example 2D dataset. The $+$ points are in the $+1$ class and the $\circ$ point is in the -1 class.

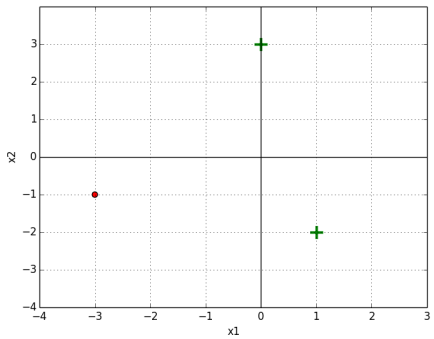


Figure 1: The green $+$ are positive and the red $\circ$ is negative

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.

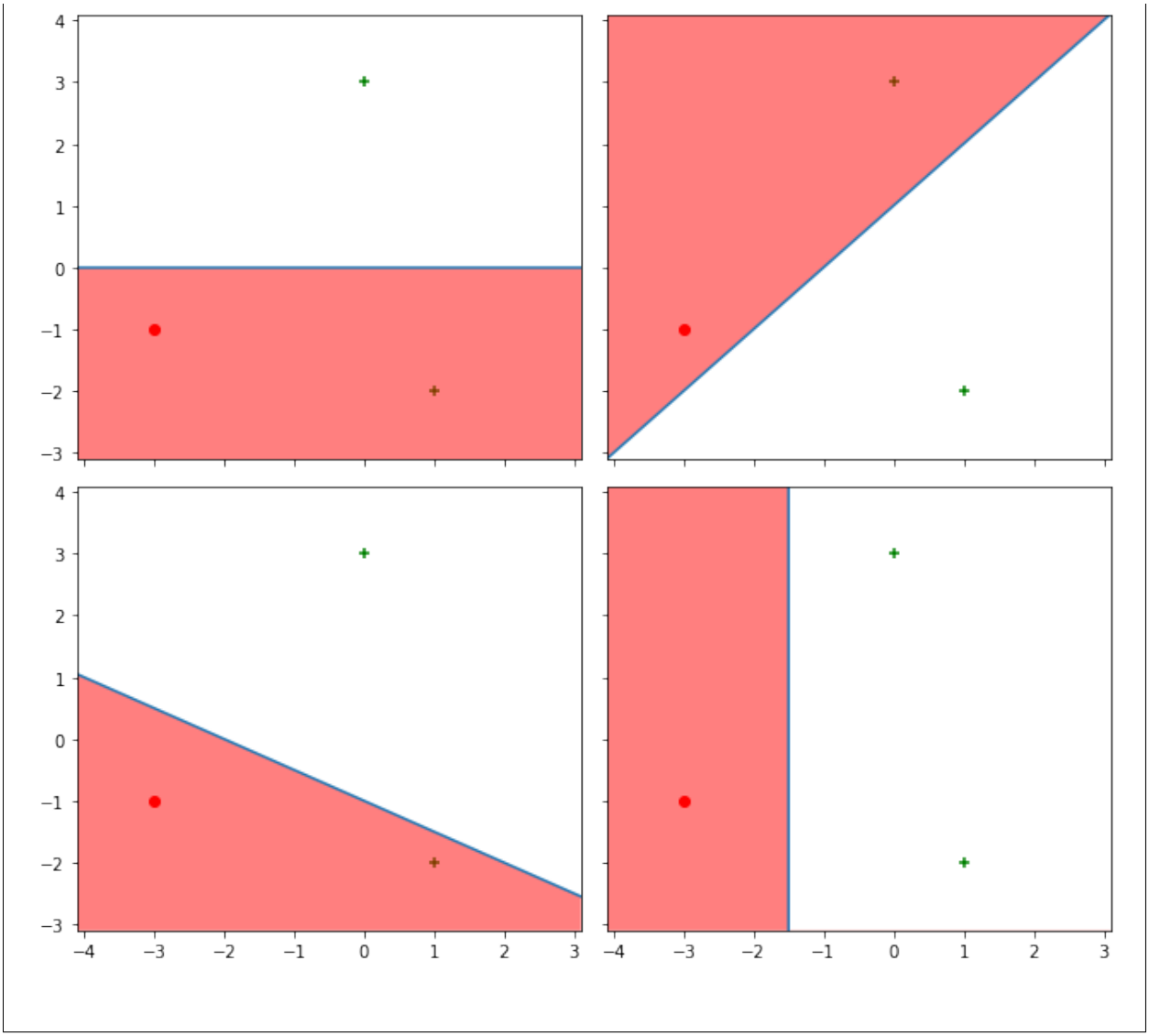
t	b	w_1	w_2	x_1	x_2	y
0	0	0	1	1	-2	+1
1	1	1	-1	0	3	+1
2	2	1	2	1	-2	+1
3	3	2	0			

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: *Colab Notebook*

```
t   b     w1     w2    x         y
0   0.0   0.0    1.0   [1 -2]    1
1   1.0   1.0   -1.0   [0  3]    1
2   2.0   1.0    2.0   [1 -2]    1
3   3.0   2.0    0.0

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



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 2D space, what is the minimum size of a dataset that is not linearly separable, such that no three points are collinear? How about the minimum size of a dataset in 3D that is not linearly separable, such that no four points are coplanar? Please limit your explanation to a few lines - you should justify but not prove your answer.

Finally, how does this generalize to N-dimension? More precisely, in N-dimensional space, what is the minimum size of a dataset that is not linearly separable, such that no $N + 1$ points are on the same hyperplane?

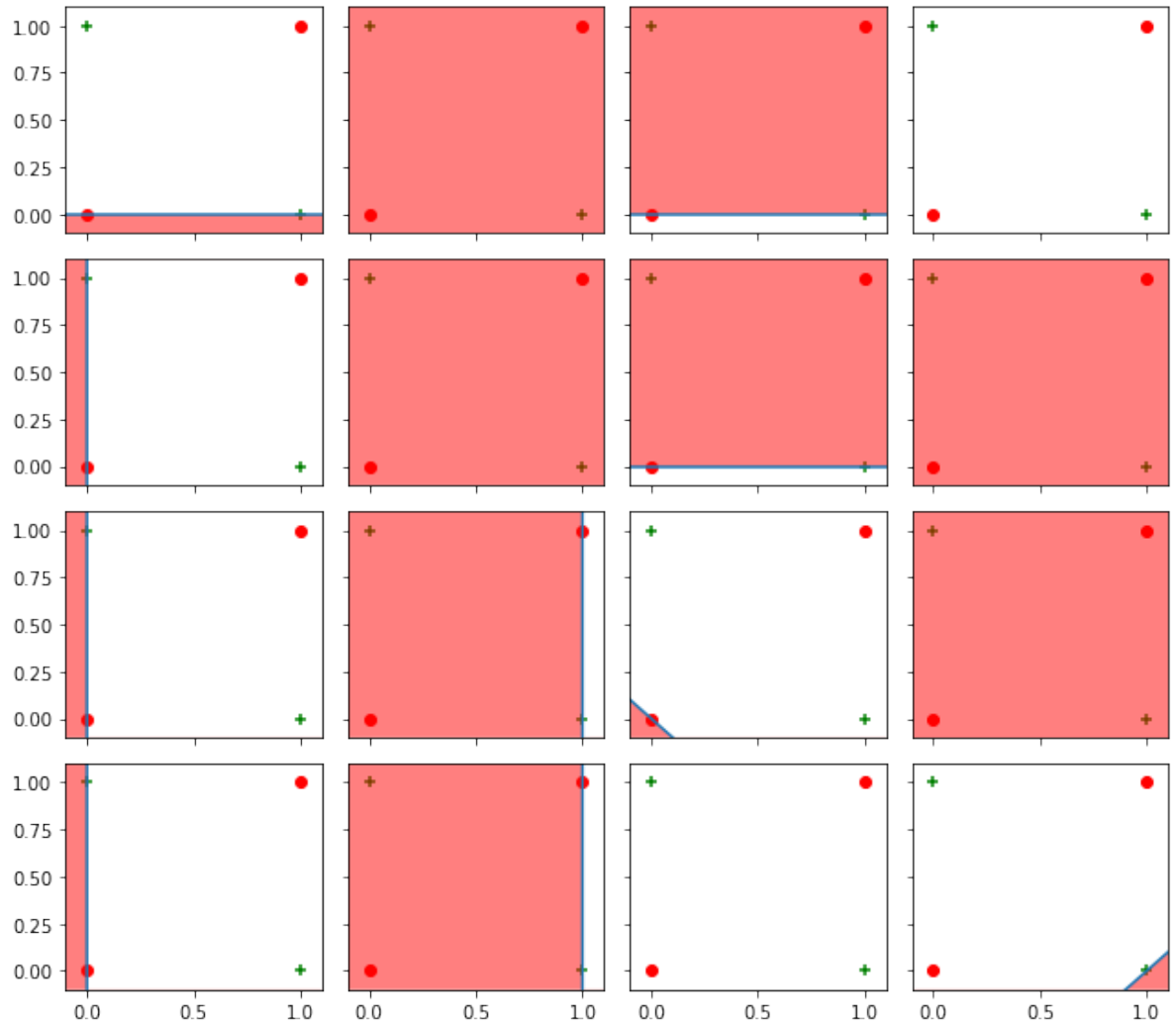
For the N-dimensional case, you may state your answer without proof or justification.

Solution B: *In 2D, the minimum size is 4 points arranged in an XOR or checkerboard shape. In 3D, the minimum size is 5 points. It can be done with 3 points labeled +1 making a plane and one point labeled -1 on either side of the plane.*

For N dimensions, the minimum is $N + 2$ points.

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:



It will never converge, because this pattern of points is not linearly separable, so there will always be at least one misclassified point.

Problem D [2 points]: How does the convergence behavior of the weight vector differ between the perceptron and SGD algorithms? Think of comparing, at a high level, their smoothness and whether they always converge (You don't need to implement any code for this problem.)

Solution D: *SGD is much smoother (it takes smaller steps than perceptron), and it will always converge because it simply looks for a local minimum, whereas perceptron looks for 0 loss.*