Machine Learning & Data Mining — Caltech CS/CNS/EE 155
Set 1 — January 5th, 2024

Policies

- Due 5 PM PST, January 13th 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`.

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 the set of all possible functions mapping from input data to output values / predictions. The hypothesis set is determined by the desired model complexity and intuition / knowledge about the process being simulated.*

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

Solution B: *The hypothesis set of a linear model is the set of all functions of the form* $f(x|w, b) = w^T x + b$. *Therefore, the set of these functions is uniquely determined by our choices of weights* w *and bias* b.

Problem C [2 points]: What is overfitting?

Solution C: *Overfitting is what happens when your training model develops a bias to the training data and no longer generalizes out of sample to the testing data. This usually happens when model complexity is too high and it starts to fit the noise instead of the underlying trends.*

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

Solution D: *Performing cross-validation can prevent overfitting because splitting a dataset into training and validation sets and selecting a model with the lowest validation error allows us to have an unbiased measure of the model's performance over the entire input space. Another solution is to increase the size and diversity of the dataset such that the input space is well-represented (eliminates training bias).*

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 train the model and generate a hypothesis function while testing data is used to measure the objective accuracy of the model over the input space. We can never change our model based on the test data because it eliminates the entire purpose of the testing set (an unbiased estimate of the model's performance).*

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

Solution F: *The first assumption is that our dataset is sampled independently, meaning that the occurrence of one data point does not change the probability that a different data point is sampled. The second assumption is that all of the points in our dataset are sampled with the same underlying probability 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: *The input space could be a tuple (Bag of Words) that marks the occurrence of a select list of "features" (in this case keywords from the email). The output space could be the integers +1 and -1 to indicate the classification of an email as spam or not spam.*

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

Solution H: *k-fold cross-validation starts by splitting the data set into k partitions where $k-1$ partitions are used as the training set and the last partition is used as the testing set. The test errors from each permutation of possible training sets and testing sets are averaged to create a validation error for the given model and dataset.*

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:

$$E_S[E_{out}(f_S)]$$

$$= E_S[E_x[(f_S(x) - y(x))^2]]$$

$$= E_x[E_S[(f_S(x) - F(x) + F(x) - y(x))^2]]$$

$$= E_x[E_S[(f_S(x) - F(x))^2 + (F(x) - y(x))^2 + 2(f_S(x) - F(x))(F(x) - y(x))]]$$

$$= E_x[E_S[(f_S(x) - F(x))^2] + E_S[(F(x) - y(x))^2] + \underbrace{E_S[2(f_S(x) - F(x))(F(x) - y(x))]}_{0}]$$

$$= E_x[E_S[(f_S(x) - F(x))^2] + [F(x) - y(x)]^2]$$

$$= E_x[\text{Bias}(x) + \text{Var}(x)] \checkmark$$

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.

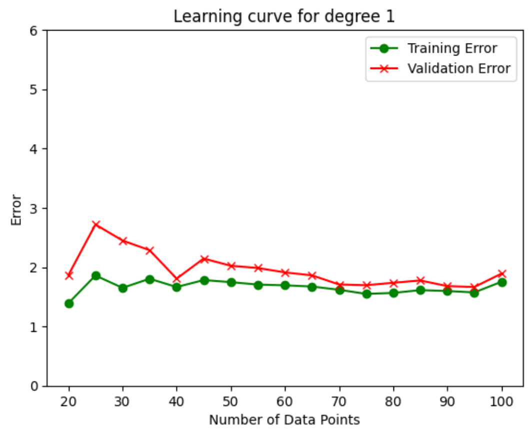
The dataset `bv_data.csv` is provided and has a header denoting which columns correspond to which values. Using this dataset, plot learning curves for 1st–, 2nd–, 6th–, and 12th–degree polynomial regression (4 separate plots) by following these steps for each degree $d \in \{1, 2, 6, 12\}$:

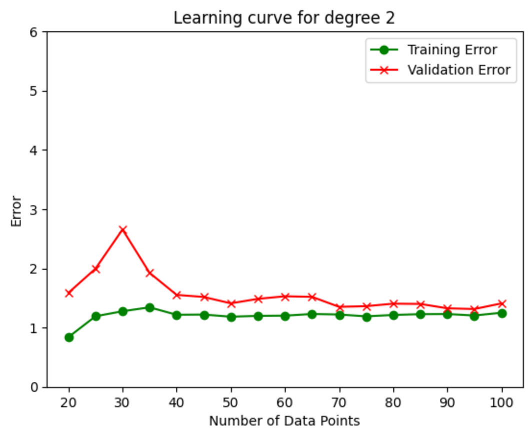
1. For each $N \in \{20, 25, 30, 35, \cdots, 100\}$:
   i. Perform 5-fold cross-validation on the first N points in the dataset (setting aside the other points), computing the both the training and validation error for each fold.
      - Use the mean squared error loss as the error function.
      - 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.

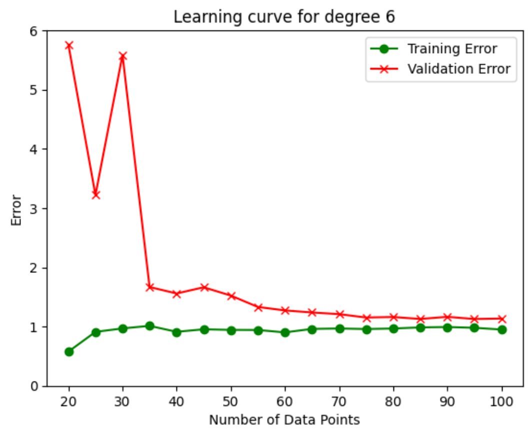
   ii. Compute the average of the training and validation errors from the 5 folds.
2. Create a learning curve by plotting both the average training and validation error as functions of N.
   Hint: Have same y-axis scale for all degrees d.

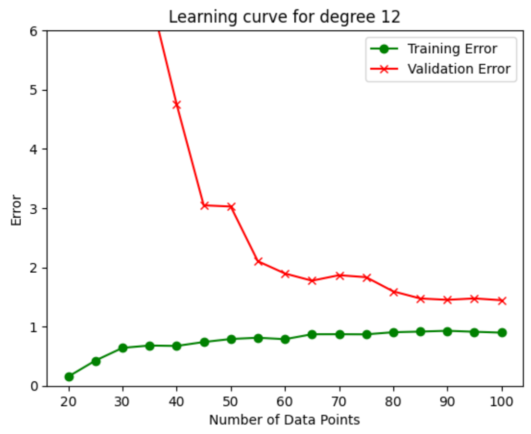
Solution B:

https://colab.research.google.com/drive/1o_78Gc9aUixgLWpCZ1l3sdjh-X0N8ZD4?usp=sharing









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 has the highest bias. We can tell because both training error and validation error are high which is a sign of underfitting. Also, the errors are largely independent of training set size.*

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

Solution D: *The model with degree 12 has the highest variance. You can tell because the validation error is much higher than the training error until the number of data points in the training set is increased sufficiently. This is an example of 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: *Based on the behavior exhibited in the learning curve of the quadratic model, we can see that additional training points will not improve training or validation error by much, if anything. After around N = 50 data points, the model has the same performance.*

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

Solution F: *The model is trained directly on the training data but it never interacts with the validation data until the end. It develops a bias toward the training data and will always fit the training data better compared to the testing data because it picks up on the noise in the training data which may not reflect the actual underlying trends of the input space.*

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 has the lowest validation error as we approach large N (Number of Data Points). Since the validation error tests the ability of the model to correctly classify unseen data, it would be most appropriate.*

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 will include an additional element in w and x. If we assume our starting weight vector w has $N zeros$ in it, then we simply prepend another 0 to the starting vector. To x, we prepend an artificial feature $x^{(0)} = 1$ to the data point. $w^{(0)}$ is our bias term.*

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:

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

$$\frac{dL}{dw} = \sum_{i=1}^{N} 2(y_i - w^T x_i) \overset{\text{chain rule}}{\frac{d}{dw}} (y_i - w^T x_i)$$

$$= \sum_{i=1}^{N} -2(y_i - w^T x_i)\, x_i$$

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.*

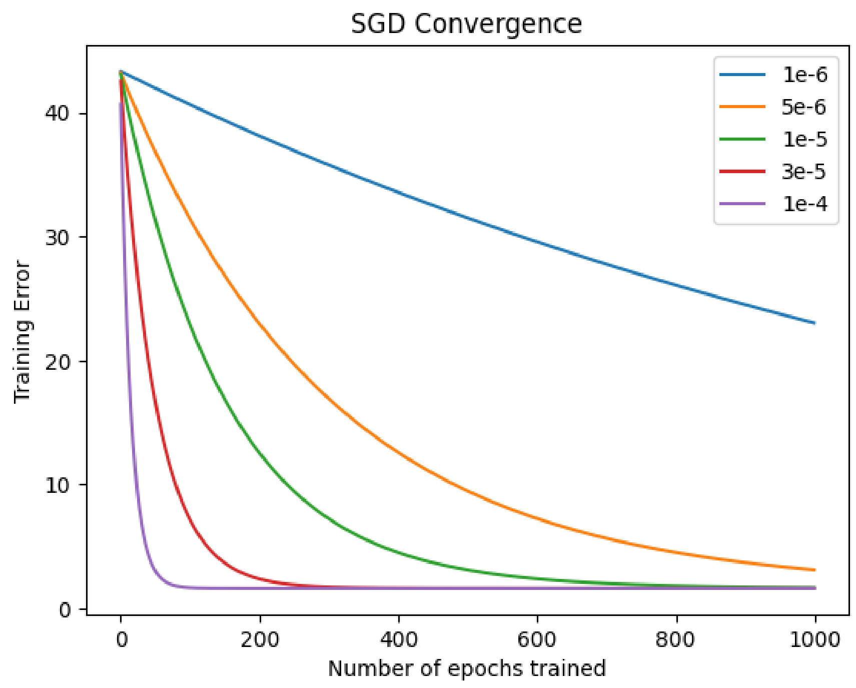
https://colab.research.google.com/drive/1Sq1l4bSeMqC2kFACV8Cdep-1Iva4ZvKG?usp=sharing

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: *We can see that no matter the starting point, the SGD converges to the same global minima of the loss function. However, the speed of convergence does change depending on the starting point. Starting points that are on steeper slopes of the loss function will progress toward the minima faster than other less steep points. Datasets 1 and 2 exhibit the same behavior but they converge towards different minima.*

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 $\eta \in \{1e\text{-}6, 5e\text{-}6, 1e\text{-}5, 3e\text{-}5, 1e\text{-}4\}$. 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 in magnitude, we see that training error decreases much quicker (i.e. SGD convergence occurs 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:

https://colab.research.google.com/drive/161B9pfm8IAZur61N4JmA-ZPrPwr47kpC?usp=sharing

Final weights:

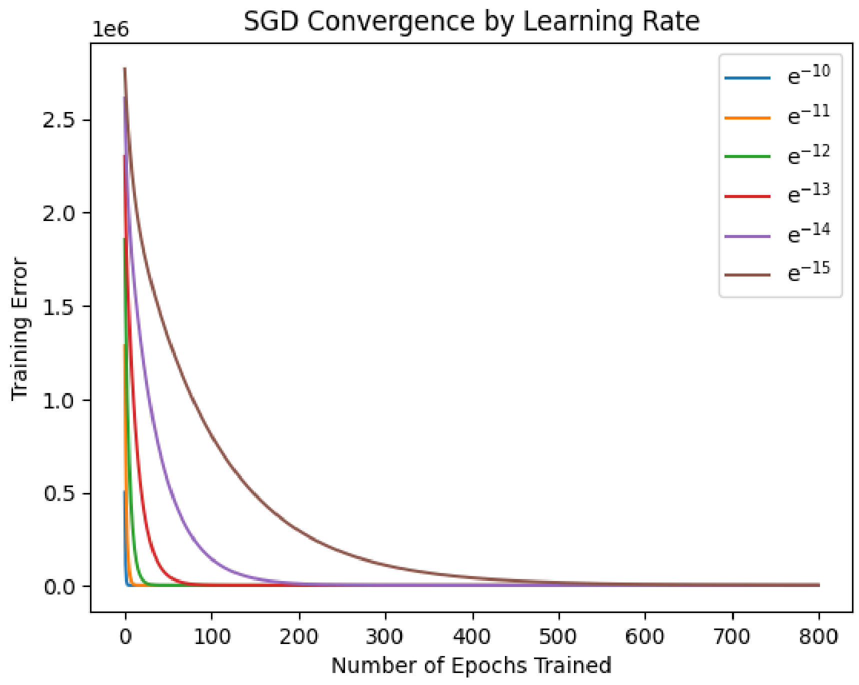
$$w = [-0.22719156, -5.94211358, 3.94389987, -11.72384116, 8.78567424]$$

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:



This is the same trend as in Problem 3E. We see that as η increases in magnitude, training error decreases much quicker and SGD convergence occurs faster (Less Epochs needed).

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 final weight vector is* $[-0.31644251, -5.99157048, 4.01509955, -11.93325972, 8.99061096]$, *where* $w^{(0)}$ *is the bias term. These values are very close to what we got 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, a large reason why we use SGD instead of closed-form solutions is because SGD is less computationally expensive. Computing gradients of the loss function 1 point at a time is less taxing than computing a large pseudo-inverse when the dataset is large (N) and has many dimensions (d).*

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 pre-defining a number of epochs to run, we can stop our SGD iterations when the deficit in loss between epochs is negligible. For example, if the loss from 1 epoch is within* 10^{-14} *of the next epoch, then we could stop iterating.*

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

Solution K: *There is no guarantee that the perceptron algorithm will ever converge however SGD will necessarily converge given a smooth differentiable loss function. This is because the weight vector in the perceptron algorithm is updated using a random misclassified point through every iteration. However, there is no guarantee that the weight vector will correctly classify all points so the perceptron's weight vector might update continuously.*

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

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

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.

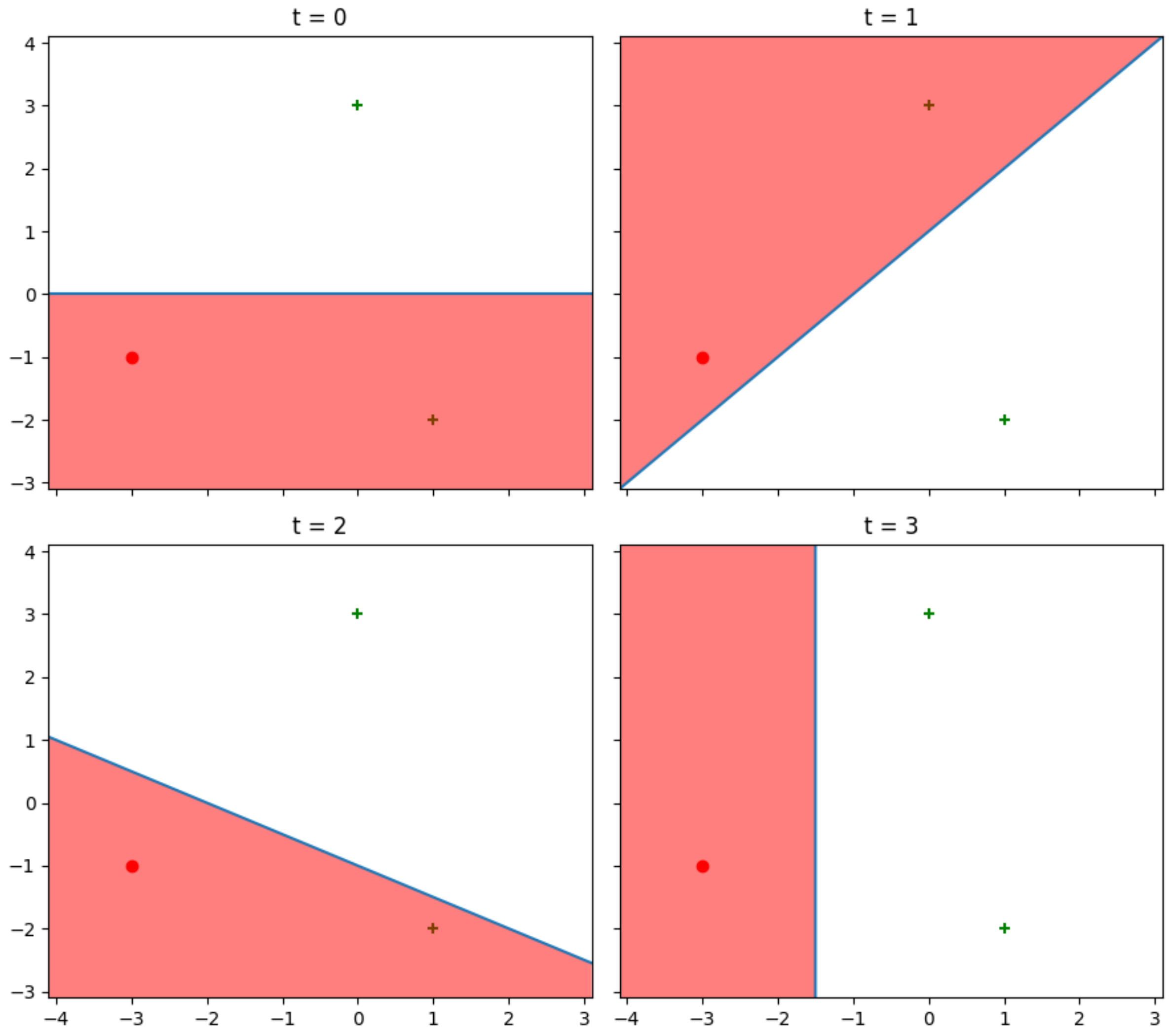
t	b	w_1	w_2	x_1	x_2	y
0	0	0	1	1	-2	+1
1	1	1	-1	0	3	+1
2	2	1	2	1	-2	+1
3	3	2	0			

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:

https://colab.research.google.com/drive/1jkd2gm4z-zUyHYeYkDYMz0uF2ImSj5EH?usp=sharing

```
t | b w1 w2 | x1 x2 | 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
```

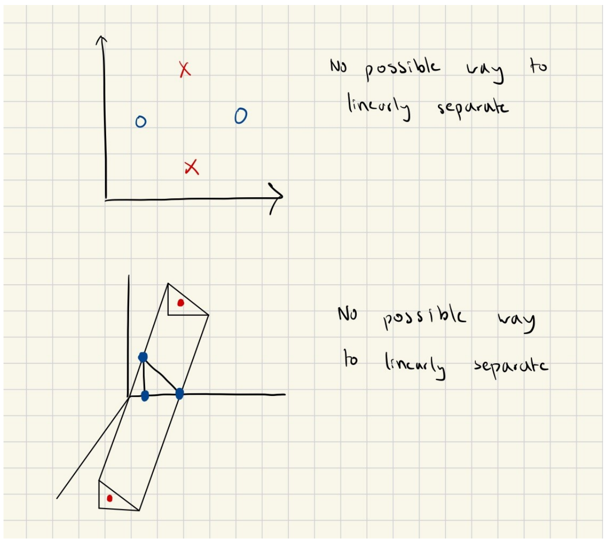


Problem B [4 points]: A dataset $S = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)\} \subset \mathbb{R}^d \times \mathbb{R}$ is *linearly separable* if there exists a perceptron that correctly classifies all data points in the set. In other words, there exists a hyperplane that separates positive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

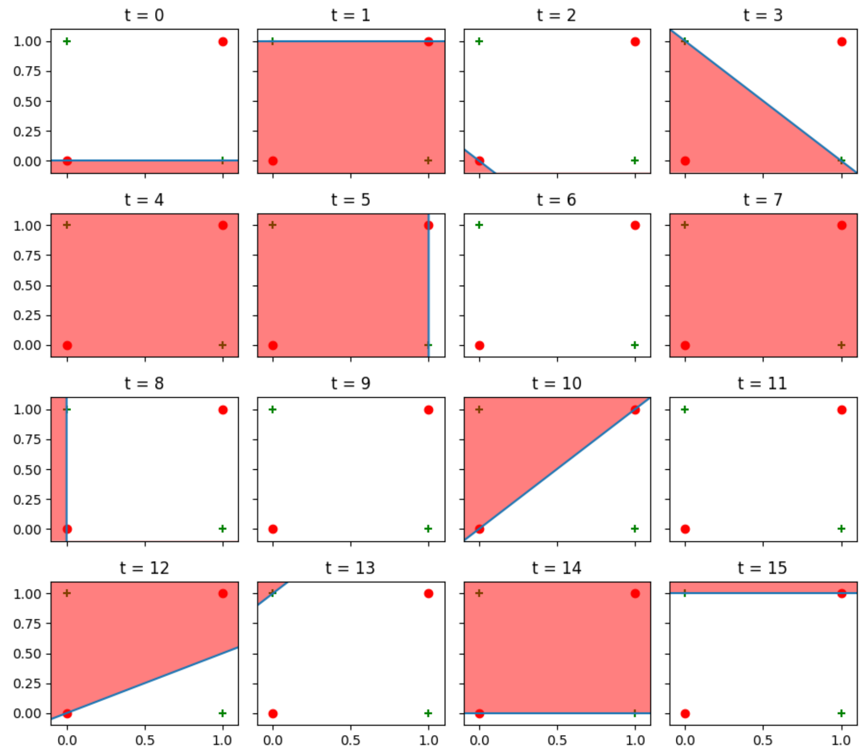
Solution B: *In a 2D dataset, the minimum number of data points in a dataset that can't be linearly separated is 4. Consider two colinear points of one classification and two other colinear points of another classification. Imagine these points set in a structure where the lines connecting the colinear points intersect with one another. There is no way to linearly separate these points. In a 3D dataset, the minimum number of points that can't be linearly separated is 5. Consider 3 coplanar points forming a right triangle of one classification. Extend a triangular prism in both directions and place points of another classification. There are no more than 3 points coplanar at any given time, but we cannot linearly separate the points. Both of these situations are illustrated below.*


X
O
O
X
No possible way to linearly separate
No possible way to linearly separate

Following this pattern, we see that for the N-dimensional set, the minimum number of points in a dataset that are not linearly separable is $N + 2$.

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 PLA algorithm will never converge because there will always be at least one misclassified point in every iteration. The PLA algorithm does not stop until all points are classified correctly.