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 functions that we could use to approximate the target function under the specified model.*

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 linear functions in the given dimension. These functions are of the form $w^T x + b$ for input x, weight vector w, and bias vector b. For example, in 2 dimensions, the hypothesis set for a linear model is the set of all lines. In 3 dimensions, the hypothesis set for a linear model is the set of all planes.*

Problem C [2 points]: What is overfitting?

Solution C: *Overfitting is when the error on the test dataset is significantly larger than the error on the training dataset.*

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

Solution D: *One way is to pick a simpler model to use, another is to use more training data. A simpler model will have less variance and will thus be less likely to overfit the training data. More training data will reduce variance since the overall dataset will be smoother and less likely to overfit specific points.*

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: *The training data is fed to the model to optimize the hypothesis function parameters. The test data is only used to approximate the performance of the model on* all *data. If the model is changed based on the test data, then there is the risk of overfitting the model based on only this set of test data, while worsening the performance on all other data.*

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

Solution F: *That the sampled data is independent 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: *The input space would be the set of all possible emails (text), and the output space would be a binary spam/not spam. Emails can be defined as vectors with each dimension corresponding to the presence of a word and the binary labels can simply be* $-1/1$

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

Solution H: *In k-fold cross-validation, the original dataset is split into k subsets. Then, over k iterations, all but one of the subsets is used as the training dataset, and the performance of the model after training on these subsets is evaluated on the final subset (validation set). The cross validation error, which is an estimate of the test error, is then computed by averaging the error from each of the k iterations.*

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: *Substituting values for error, bias, and variance,*

$$\mathbb{E}_S\left[\mathbb{E}_x\left[\left(f_S(x) - y(x)\right)^2\right]\right] = \mathbb{E}_x\left[\left(\mathbb{E}_S\left[f_S(x)\right] - y(x)\right)^2 + \mathbb{E}_S\left[\left(f_S(x) - \mathbb{E}_S\left[f_S(x)\right]\right)^2\right]\right]$$

Expanding, we get

$$\begin{aligned} &\mathbb{E}_S[\mathbb{E}_x[f_S(x)^2]] - 2\mathbb{E}_S[\mathbb{E}_x[f_S(x)y(x)]] + \mathbb{E}_S[\mathbb{E}_x[y(x)^2]] = \\ &\quad \mathbb{E}_x[F(x)^2] - 2E_x[F(x)y(x)] + \mathbb{E}_x[y(x)^2] + \mathbb{E}_x[\mathbb{E}_S[f_S(x)^2]] - 2\mathbb{E}_x[\mathbb{E}_S[f_S(x)F(x)]] + \mathbb{E}_x[\mathbb{E}_S[F(x)^2]] \end{aligned}$$

Cancelling terms and simplifying, we get

$$-2\mathbb{E}_x[\mathbb{E}_S[f_S(x)]y(x)] = \mathbb{E}_x[F(x)^2] - 2\mathbb{E}_x[F(x)y(x)] - 2\mathbb{E}_x[\mathbb{E}_S[f_s(x)]F(x)] + \mathbb{E}_x[F(x)^2]$$

which yields

$$-2\mathbb{E}_x[F(x)y(x)] = 2\mathbb{E}_x[F(x)^2] - 2\mathbb{E}_x[F(x)y(x)] - 2\mathbb{E}_x[F(x)^2] = -2\mathbb{E}_x[F(x)y(x)]$$

which is clearly true.

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: *Code link*

https://colab.research.google.com/drive/1r_qRYIX9K6ilAok0OEwl_GLSQ_snjagv?usp=sharing

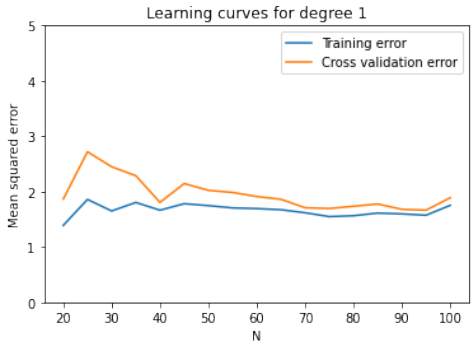

Figure 1: Errors for $d = 1$

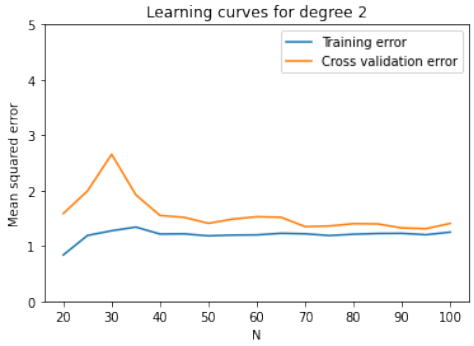

Figure 2: Errors for $d = 2$

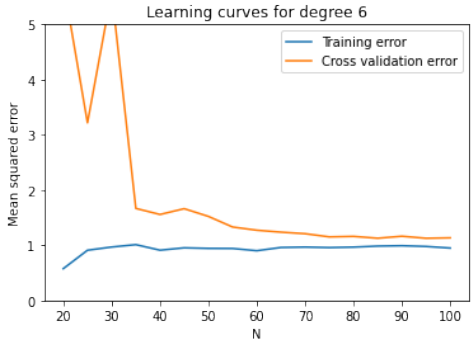

Figure 3: Errors for $d = 6$

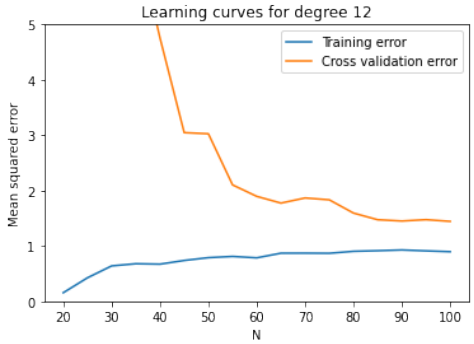

Figure 4: Errors for $d = 12$

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 had the highest bias, since the training error was the highest for that model. High training error is a characteristic of high bias, or underfitting.*

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

Solution D: *The polynomial regression model of degree 12 had the highest variance, since it was characterized by low training error, but high cross validation error, which is indicative 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: *The cross validation error of the model will decrease as additional training points are added and it will approach the training error, but not decrease below it.*

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

Solution F: *The model is optimized on the training dataset, so the training error will be lowered by the optimization algorithm. The validation error, however, is error on data the model is not optimized on, so the model is more likely to incorrectly predict those values, leading to higher 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 degree 2 polynomial to perform best on some unseen data drawn from the same distribution, since it achieves a low training and cross validation error while not 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:

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

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 accomplish this by defining* $\vec{x} = \langle 1, x_1, x_2, ...x_d \rangle$ *and* $\vec{w} = \langle b, w_1, w_2, ...w_d \rangle$

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:

$$\begin{aligned} \frac{\partial}{\partial w} L(f) &= \frac{\partial}{\partial w} \sum_{i=1}^{N} (y_i - w^T x_i)^2 \\ &= -2 \sum_{i=1}^{N} (y_i - w^T x_i) x_i \end{aligned}$$

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.*

Code link

https://colab.research.google.com/drive/1AKwDLc0pRTGClwumJgWCJ41gEGIU4FHs?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:

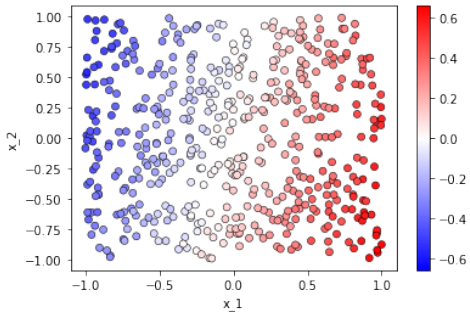

Figure 5: Plot of first dataset

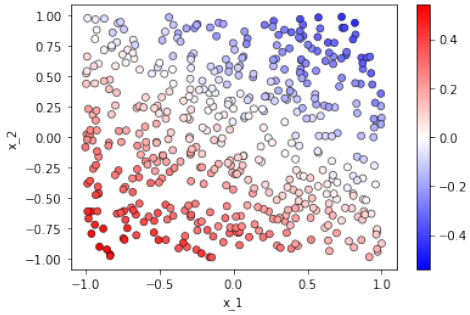

Figure 6: Plot of second dataset

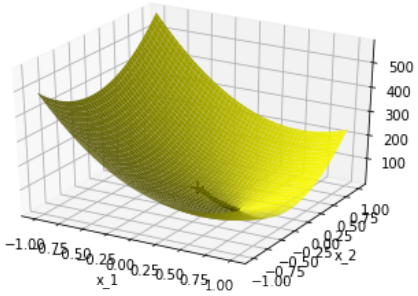

Figure 7: Convergence on first data point from single starting point

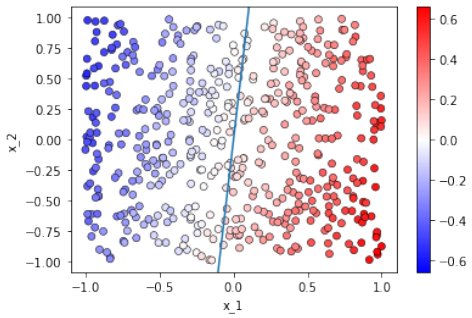

Figure 8: Convergence of weight vector on first dataset

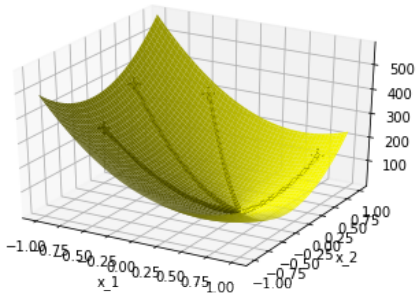

Figure 9: Convergence on first dataset from different starting points

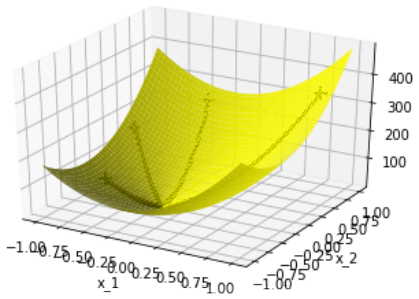

Figure 10: Convergence on second dataset from different starting points

Regardless of the starting point, the SGD algorithm converges towards the minimum nearly directly. Even when comparing the convergence across different datasets, different starting points still results in convergence of the algorithm. Each path starts from a different point but traverses towards the minimum.

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:

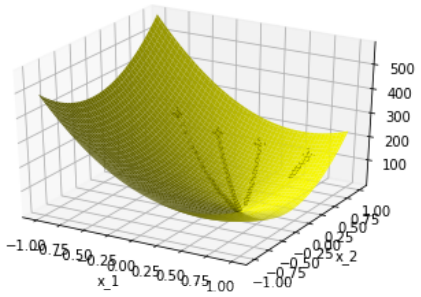

Figure 11: Convergence paths for different learning rates

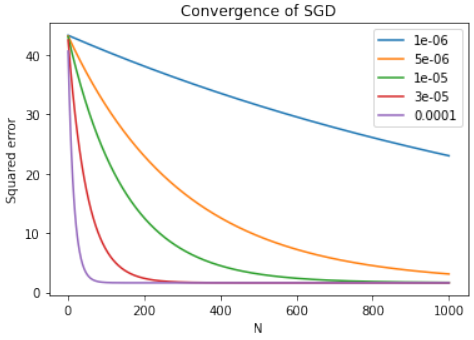

Figure 12: SGD error convergence for different learning rates

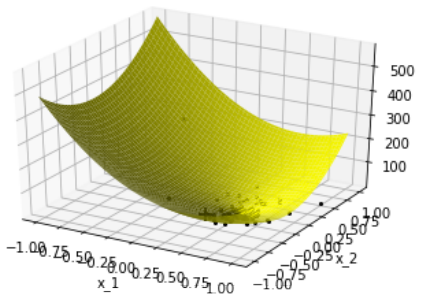

Figure 13: SGD with large learning rate

As η becomes smaller, the loss decreases at a slower rate and the loss after the same amount of epochs is larger for smaller learning rate, since the model has not converged as well. Meanwhile, the largest learning rate converged very quickly and arrived at a very low error rate.

However, if η becomes too large, then the model never converges, since the updates for each iteration are too large, the weight vector jumps around the minimum and can never reach the point of minimum error.

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 link*

https://colab.research.google.com/drive/1l3GNg4qN64dDfjl43ocTblSOg5K5rk2B?usp=sharing

$$w = \langle -0.22720591, -5.94229011, 3.94369494, -11.72402388, 8.78549375 \rangle$$

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:

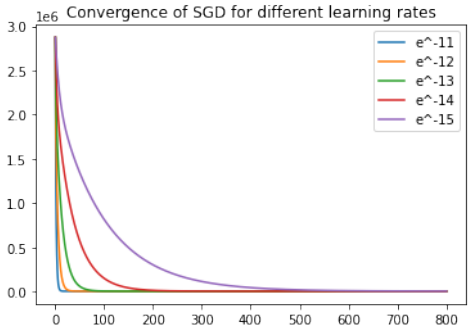

Figure 14: Training error vs. number of epochs for different learning rates

As the learning rate increases, the training error decreases very quickly in early epochs, then the error levels off and stops decreasing. For lower learning rates, the training error decreases more slowly.

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 resulting weight vector returned by the analytical solution is*

$$w = \langle -0.31644251, -5.99157048, 4.01509955, -11.93325972, 8.99061096 \rangle$$

This answer is very near the answer produced by 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: *If the number of dimensions of the vectors x_i is large, then the matrix that must be inverted is of size $d \times d$. As this matrix grows larger, it becomes more expensive to invert. Then, it is easier to compute the gradient, which does not require matrix inversion.*

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: *Since we want to stop the algorithm after it has converged, one stopping condition definition could be to compute the change in loss across epochs, or the derivative of the loss. If the average change in loss becomes very small, we could stop the algorithm. Another method would be to stop the algorithm after the error reaches below a specified threshold.*

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

Solution K: *The perceptron algorithm only converges if it achieves total accuracy, which can only occur if the dataset is linearly separable. If the data can not be separated linearly, then the weight vector for the perceptron algorithm continues to change. Meanwhile, the SGD weight vector approaches the line-of-best-fit, which minimizes the squared error, so the weight vector can converge to the value that minimizes the squared error. The SGD convergence is smoother because the error is real valued, while perceptron error is generally discrete (classification accuracy).*

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

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

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.

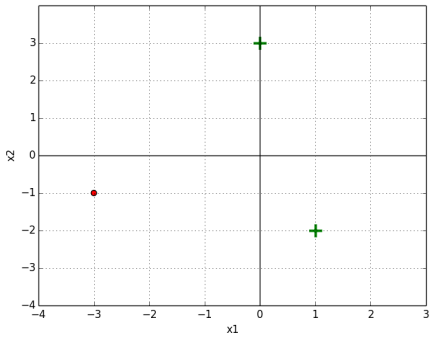

Figure 15: The green $+$ are positive and the red $\circ$ is negative

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: *Code link*

https://colab.research.google.com/drive/10Cua0-AkiVtcIZ3m8gMi6ZSZVOIFFWd_?usp=sharing

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			

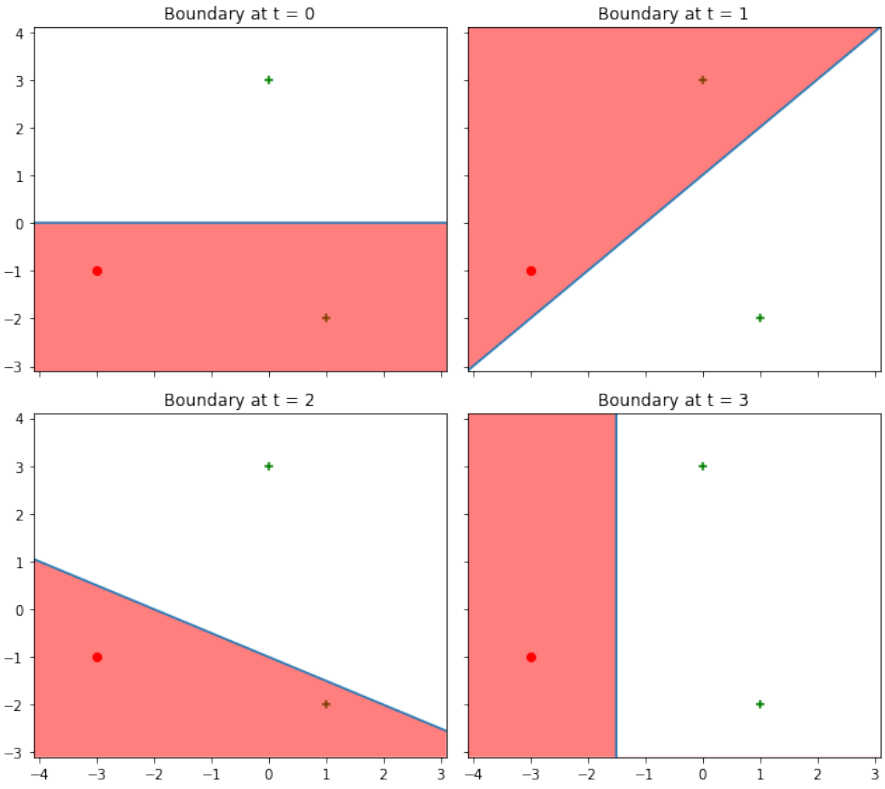

Figure 16: Perceptron Learning Algorithm boundary

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: *For a 2D dataset, the smallest set of points that is not linearly separable such that no 3 points are collinear is a set of 4 points. One example of an arrangement of points that satisfies this is 2 sets of 2 points with the same y value set in the shape of a cross. Another example is a triangle of points with 3 of a certain y value surrounding a 4th point of a different y value. For a 3D dataset, the smallest such set of points is 5 points, which can be shown with a tetrahedron and an interior different point.*

For the N-dimensional case, the smallest set of points is of size $N + 2$.

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:

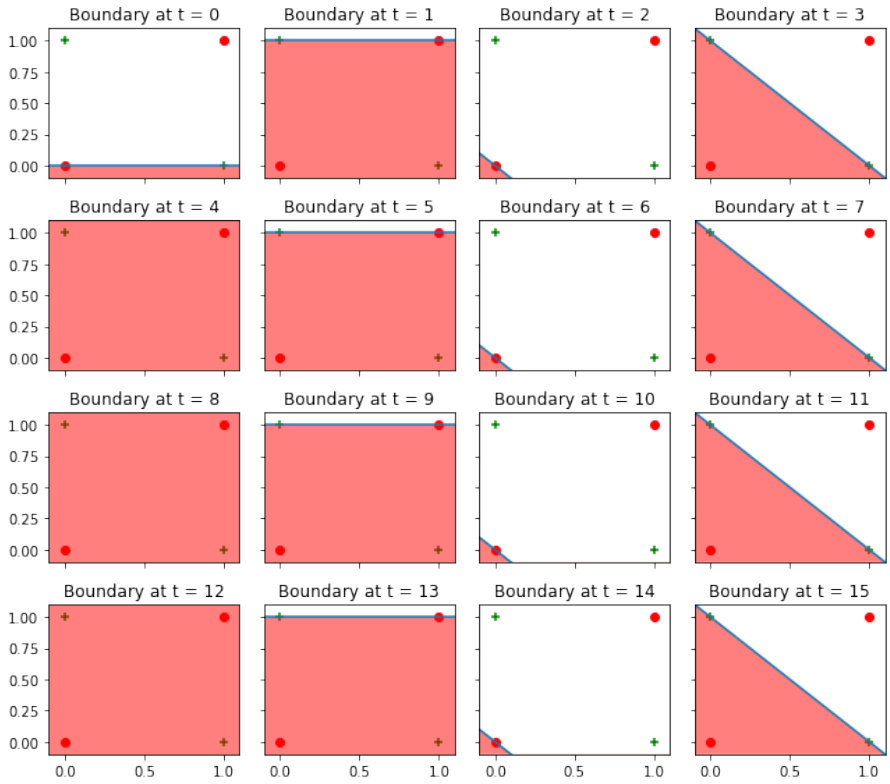

Figure 17: Perceptron Algorithm with no convergence

If the data is not linearly separable, then the Perceptron Learning Algorithm never converges, because no matter how it corrects the weight vector, there is at least one misclassified point, so the algorithm will continue to select points and update the weights.