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 hypothesis set is the set of all possible hypotheses (map input data to output data) for the target function.*

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

Solution B: *The hypothesis set, usually represented by H, is the set of all candidate formulas for a linear formula of the form $w^T x + b$.*

Problem C [2 points]: What is overfitting?

Solution C: *Overfitting occurs with the model has a much higher error in the test data than the error in training data, test error is much larger $>>$ training error. Overfitting means that the model learns the specifics and noise of the training data too well to the extent that the performance in the test data is negatively impacted (high test error). Overfitting implies high variance.*

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

Solution D: *One way is to train the model on a larger set of training data in order to have a more accurate representation of the true distribution and to reduce variance. In addition, methods using a validation set, such as k-fold cross validation can also reduce overfitting. Methods like these include a process to evaluate and update the model using different portions of the training data, thereby resulting in a final hypothesis that yields lower errors on the validation set.*

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 set is used to "learn" the model as our learning algorithm uses this data set to fit the parameters of the target function. This helps use find a hypothesis. The test data is an independent data set that is used to evaluate the performance of the final hypothesis/fully trained model.*

You should not change your model based on information from the test data because this is a form of data snooping,

which can result in the final hypothesis being (over)fit/biased towards the test set. If a data set has affected any step in the learning process, the models ability to assess the outcome has been compromised. This means lower out-of-sample performance, meaning the model does not perform well on data is has not seen.

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

Solution F: *The first assumption is that the data is sampled independently of the "true" distribution.*

The second assumption is that the results/outputs of the training data should be diverse, i.e. should have all/wide variety of the possible outcomes represented. For example, in our email spam learning algorithm, the data set should have emails that are both spam and not spam.

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

Solution G: *X, the input space, is the set of emails in some parameterized vector form. As explained in class, the input space could be the set of a number of emails broken down into a vector based on its "bag of words" representation. If our model simply reports whether the email is or is not, then Y, is $0, 1$. 0 means that email is not spam and 1 means that the email is spam.*

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

Solution H: *Begin by splitting the dataset into k equal-sized partitions/groups. Then, for each group of the k groups, let one group be a test data set and the remaining $k-1$ groups be the training data sets. Fit/train the model on the training data set and the evaluate it on the test set. After this process (i.e. k rounds of training), the resulting model is the outcome.*

In this way, each group is held out of the set 1 time and used to train the model $k-1$ times, allowing for the re-using of training data as 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:

$$\begin{aligned}
E_{out}(f_s)\big] &= \mathbb{E}_x\Big[\left(f_S(x) - y(x)\right)^2\Big]\Big] && (1)\\
\mathbb{E}_S\big[E_{out}(f_s)\big] &= \mathbb{E}_S\Big[\mathbb{E}_x\Big[\left(f_S(x) - y(x)\right)^2\Big]\Big] && (2)\\
&= \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(f_S(x) - y(x)\right)^2\Big]\Big] && (3)\\
&= \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(f_S(x) - F(x) + F(x) - y(x)\right)^2\Big]\Big] && (4)\\
&= \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(\left(f_S(x) - F(x)\right) + \left(F(x) - y(x)\right)\right)^2\Big]\Big] && (5)\\
&= \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(f_S(x) - F(x)\right)^2 + \left(F(x) - y(x)\right)^2 + 2(f_S(x) - F(x))(F(x) - y(x))\Big]\Big] && (6)\\
&= \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(f_S(x) - F(x)\right)^2\Big]\Big] + \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(F(x) - y(x)^2\right]\Big] + \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(2(f_S(x) - F(x))(F(x) - y(x))\right]\Big] && (7)\\
&= \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(f_S(x) - F(x)\right)^2\Big]\Big] + \mathbb{E}_x\Big[\mathbb{E}_S\Big[\left(F(x) - y(x)^2\right]\Big] + 0 && (8)\\
&= \mathbb{E}_x\Big[\mathbb{E}_s\Big[\left(f_S(x) - F(x)\right)^2\Big]\Big] + \mathbb{E}_x\Big[\left(F(x) - y(x)\right)^2\Big] && (9)\\
&= \mathbb{E}_x\Big[\text{Var}(x)\Big] + \mathbb{E}_x\Big[\text{Bias}(x)\Big] && (10)\\
&= \mathbb{E}_x\Big[Bias(x) + Var(x)\Big] && (11)\\
& && (12)
\end{aligned}$$

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: *https://drive.google.com/file/d/1Hvx3v6T1LMqtOEcc4RudmHFTh-aHh9Zg/view?usp=sharing*

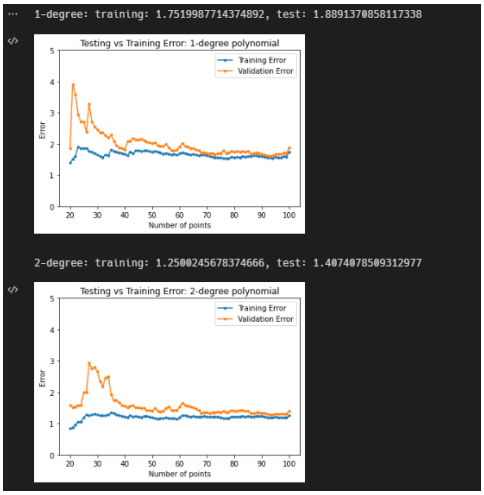

Figure 1: Graph of N v Training and Test Error

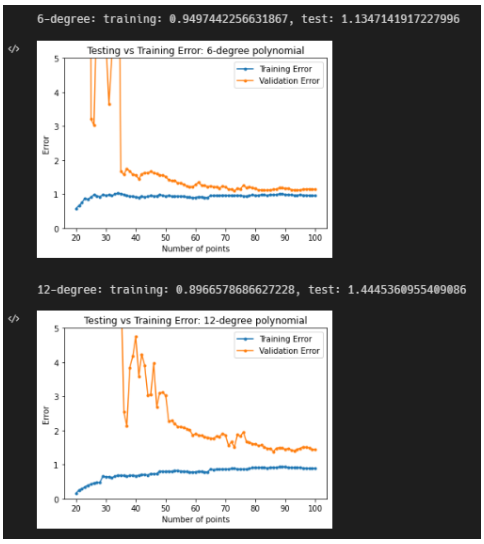

Figure 2: Graph of N v Training and Test Error

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 regression model has the highest bias (closer to 2 as compared to 1 for the other degrees). From the graphs, we see that the 1st-degree polynomial model has the highest average training*

error for the data. This make sense since bias typically decreases with increasing model complexity since low complexity models are not able to fit the data well enough to model the underlying pattern. In addition, we know that a high bias implies underfitting and lower model complexities often imply high bias (1st-degree polynomial model is the lowest model complexity of our models).

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

Solution D: *The 12-degree polynomial model has the highest variance because with low numbers of data points, the validation error is extremely high but as the number of data points increases, the validation error decreases. In addition, we know that variance increase with model complexity so it makes since that the 12th-degree polynomial has high variance, implying 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: *Since the learning curve of the quadratic model plateaus/does not significantly decrease after 50 points, this tells us that the model would not gain much improvement even if we had additional training points. The main issue is not variance because the error (both training and test) does not significantly decreases with more data points.*

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

Solution F: *Training error is lower than validation error because our model is trained on this data so it is meant to fit this data better. Since it is not possible for the training data to be perfectly representative of the entire data set (i.e accurately match the entire data set's distribution), the model will perform better on the data is it trained on. In k-fold cross validation, the validation error is calculated from the "untouched" test data in each fold, so it the error on this set of data will be larger than the training error (the margin of this higher error depends on the model's generalizing ability).*

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 2-degree model to perform best since its validation error (variance) is the low while also having a relatively low bias. As compared to the 1st-degree polynomial, 2nd degree polynomial has the lowest error that both training and test converge too. In addition, compared to the 6th and 12th degree polynomial, the 2nd degree has much lower variance. This balance between bias and variance indicates that it*

would have good generalizing ability on other sets of data with the same distribution.

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 some $x, h \in \mathbb{R}^n$, there exists $c \in (0, 1)$ such that $f(x + h) = f(x) + \nabla f(x + c \cdot h)^T h$.*

Solution A:

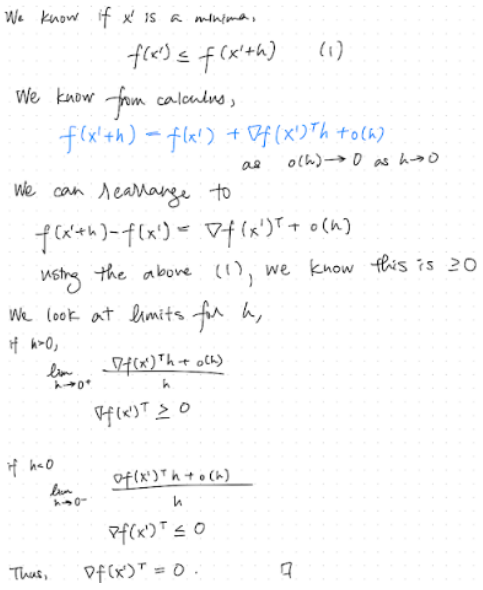

Figure 3: Graph of Learning Rate vs. Loss

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: *Add* $w_0 = b$ *and* $x_0 = 1$*, so that* $\mathbf{w} = (w_0, w_1, ..., w_d)$ *and* $\mathbf{x} = (1, x_1, x_2..., x_d)$*.*

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:

$$L(f) = \sum_{i=1}^{N} (y_i - \mathbf{w}^T\mathbf{x}_i)^2 \tag{13}$$

$$\partial_w L(f) = \partial_w \sum_{i=1}^{N} (y_i - \mathbf{w}^T\mathbf{x}_i)^2 \tag{14}$$

$$= \sum_{i=1}^{N} \partial_w (y_i - \mathbf{w}^T\mathbf{x}_i)^2 \tag{15}$$

$$= \sum_{i=1}^{N} -2(y_i - \mathbf{w}^T\mathbf{x}_i)\mathbf{x}_i \tag{16}$$

$$\tag{17}$$

Difference in computational complexity in 1 update of the weight vector: Stochastic gradient descent uses random sample (we used np.random.permutation) to use only one point/a subset of the training sample from the training set to update the weight parameter. In comparison, gradient descent runs through all the samples in the training set in a single update of a parameter(weight/bias) in a particular iteration. This means that the computational complexity is much higher in GD compared to SGD because in every iteration we are going through the entire training set. SGD is faster as only a single training subset is used and the updates to the parameters (weight/bias) begin in the first iteration. between GD and SGD

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: *See code. https://drive.google.com/file/d/1L1rsLMx_foUzFb0o30htE4MRdbSZoCGq/view?usp=share_link*

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 all starting points, SGD always converges to the same point and all at the same. This means for the points farther away the gradient changes faster. In both cases, it reaches a global minimum in a straight line (along the curve) without diverging. The animation shows the same behavior and timing in both data sets.*

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:

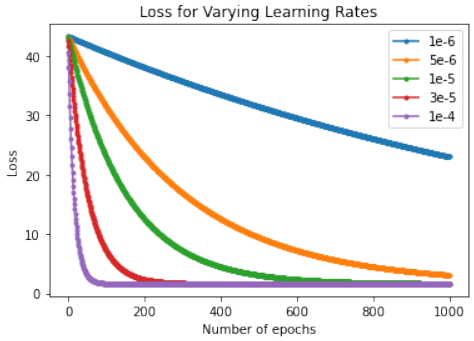

Figure 4: Graph of Learning Rate vs. Loss

Looking at the graph, we see that SGD converges fastest – taking less epochs – at larger values of η. At low values of η, $1e-6$, SGD takes many many more epochs to converge.

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: *https://drive.google.com/file/d/1HckLy3qX8VJZBoOxrap9puijD9Mc4wZw/view?usp=share_link After implementation, my final weights are* $[-0.22718437 - 5.942093573.94391876 - 11.723825168.78569577]$.

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:

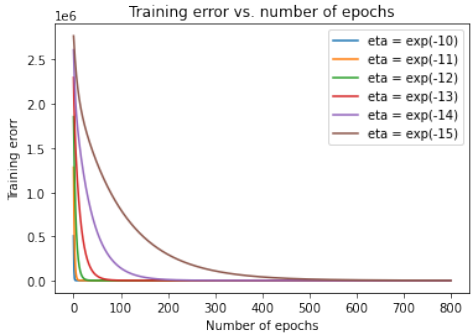

Figure 5: Graph of Learning Rate vs. Loss

The smaller η is the longer it take for our training error to decrease. Larger values of η take fewer epochs to converge (for example exp(-10), exp(-11), exp(-12) converge in much fewer epochs)

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: *The analytical solution I get is* $[-0.31644251 - 5.991570484.01509955 - 11.933259728.99061096]$ *which is relatively close to the solution of* $[-0.22718437 - 5.942093573.94391876 - 11.723825168.78569577]$ *via SGD.*

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: *In some cases the closed form solution could be very computationally expensive because of the a very large data set size or the input dimension is increasingly large (time complexity is large on taking an matrix inverse, for example). In comparison, parallelization is (relatively) easily implemented with SGD which can reduce the time it takes to compute a solution.*

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: *Since the training error asymptotically reaches 0 as the number of epochs increases, we can define a threshold for the change in training error that is sufficiently small for our purpose. This would indicate that the improvement in each following iteration is sufficiently small, so there's no purpose in calculating further epochs.*

We can calculate this change in the weight vector, by using the norm of the existing weight vector and the new vector vector is smaller than a threshold (for example $\|(w^{t-1} - w^t < 0.01\|$*).*

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.

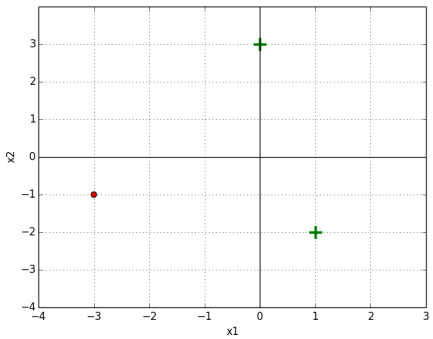

Figure 6: The green $+$ are positive and the red $\circ$ is negative

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: *https://drive.google.com/file/d/1CSVmfl-UO7_Qi739mJ4TcU3Bigco-qfz/view?usp=share_link*

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

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

Figure 7: Table from running PLA code

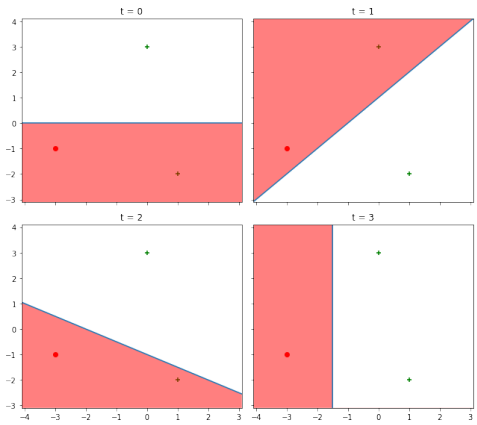

Figure 8: Graph from running PLA code, each graph is one iteration

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 points are on the same hyperplane? For the N-dimensional case, you may state your answer without proof or justification.

Solution B: *For 2D space, the minimum size of a dataset that is not linearly separable is 3. From linear algebra, we know that any two points in 2D are colinear and if we have a line that separates two points with the same sign intersecting with a line that separates two different point with the opposite sign (as we see in the plot in the ipynb), then we can't separate these points with a line.*

For 3D, 5 points would not be separable. Again using linear algebra, we know that 3 points define a plane and if we have a plane of 3 points of one sign that separates two opposite-signed points, we wouldn't be able to separate them linearly.

Extrapolating this, for a N-dimensional, a N + 2 sized dataset is the minimum sized set that isn't linearly separable.

Problem C [2 points]: Run the visualization code in the Jupyter notebook section corresponding to ques-

tion C (report your plots). Assume a dataset is *not* linearly separable. Will the Perceptron Learning Algorithm ever converge? Why or why not?

Solution C:

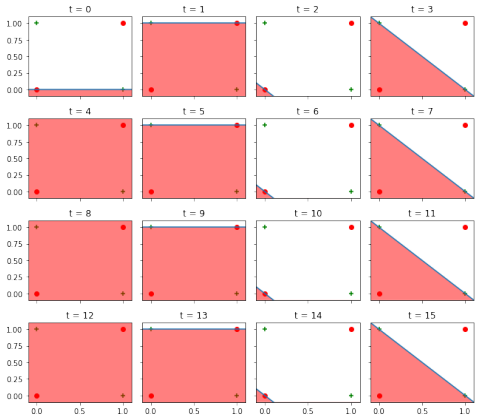

Figure 9: Graph from running PLA code on linearly inseparable

PLA will never converge because as we explained in the previous question, this dataset is makes it impossible to ever find a line/hyperplane that would divide these point(which are linearly inseparable). For every iteration, there will always be a misclassified point so the algorithm will iterate infinitely continuously updating the weights and biases and will never break from this loop.

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: *In the perceptron algorithm, the weight vector changes both increasing and decreasing (because of the randomness of choosing points), as it converges (it's also possible that it doesn't converge as explained in the cases for the previous answers). In comparison, in SGD, the weight vector is guaranteed to smoothly and gradually converge to a value without divergence (unless we have the case that the chosen step size is too large, then our weight vector may never converge).*