Machine Learning & Data Mining - Set 1 Caltech CS/CNS/EE 155
Ben Juarez January $4^{th}$, 2022

---

# 1 Basics [16 Points]

Answer each of the following problems with 1-2 short sentences.

**Question A [2 points]:** What is a hypothesis set?

**Solution A:** *A hypothesis set involves all of the hypotheses such the process will essentially select the best hypothesis to approximate the target function that would map our inputs to our outputs.*

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

**Solution B:** *A hypothesis set involves all of the hypotheses in the form* $f(x|w) = w^t x$ *such the process will essentially select the best hypothesis to approximate the target function that would map our inputs to our outputs.*

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

**Solution C:** *Overfitting occurs when the test error is much greater than the training error.*

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

**Solution D:** *One way to prevent overfitting is to use more data for training. Another solution could be to reduce the complexity of the model.*

**Question 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:** *Being of the same form, training data is used to "train" the model in order to select the appropriate hypothesis, while the test data is used to check the accuracy of the model. You should never change your model based on information from test data because the purpose of the model is to be able to perform accurately with the test data without any prior knowledge of the test data, so we cannot trust the accuracy of a model that was "pre-fitted" to the test data.*

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

**Solution F:** *We assume that our dataset is sampled such that each datapoint is (1) independent and identically distributed and that (2) the true probability distribution over all possible data is captured by our sampling.*

**Question 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 $X$ could be words within the subject line (and/or body) such that each word is assigned an integer value according to a certain vocabulary (AKA bag of words). The output space $Y$ could be $\pm 1$ in order to denote whether or not an email is spam (+1) or not a spam (-1).*

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

**Solution H:** *This validation procedure involves splitting the original dataset into $k$ parts of equal size such that $k-1$ partitions are used the training set while the remaining partition is used as the test set. This process is repeated such that each partition is used as a test set while the validation errors at each step are averaged.*

## 2 Bias-Variance Tradeoff [34 Points]

**Question 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:**

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

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.

**Question 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

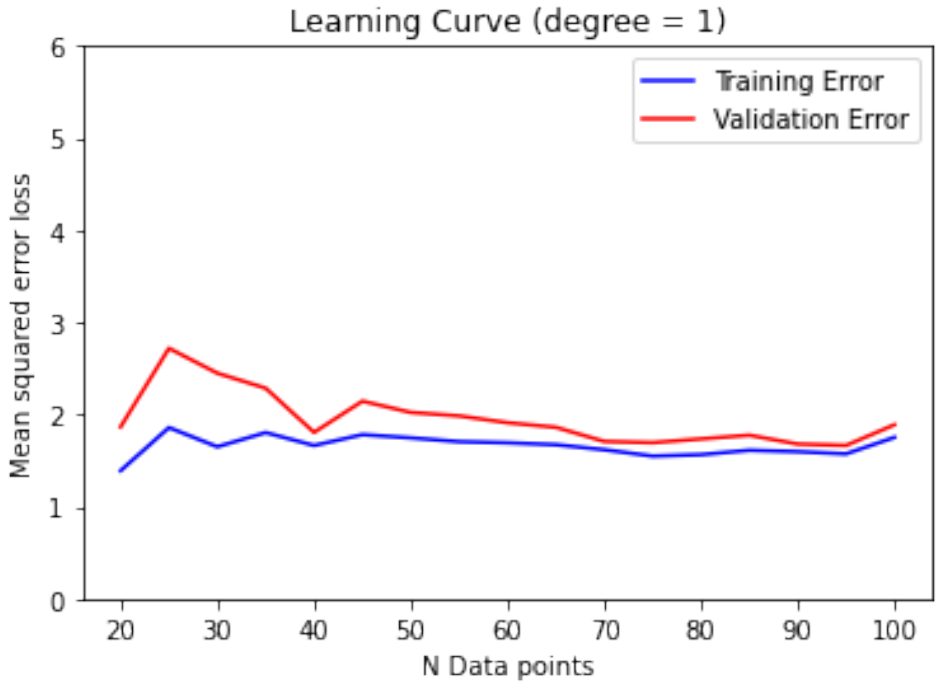
scikit-learn's KFold method; you may find it helpful to read through and run this example code prior to continuing with this problem. Additionally, you may find it helpful to look at the documentation for scikit-learn's learning_curve method for some guidance.

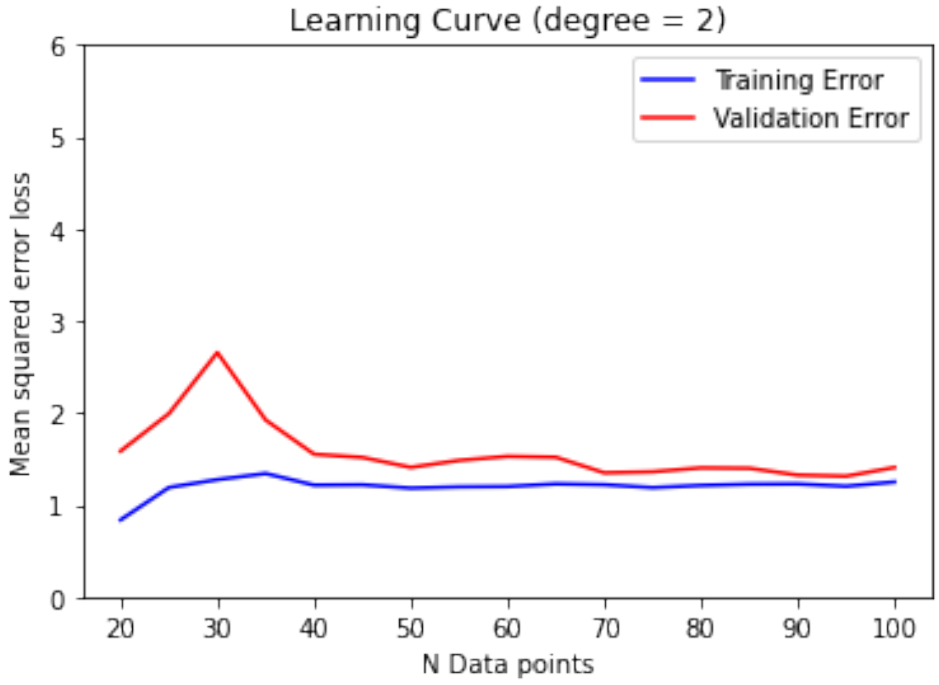
The dataset `bv_data.csv` is provided and has a header denoting which columns correspond to which values. Using this dataset, plot learning curves for 1st–, 2nd–, 6th–, and 12th–degree polynomial regression (4 separate plots) by following these steps for each degree $d \in \{1, 2, 6, 12\}$:

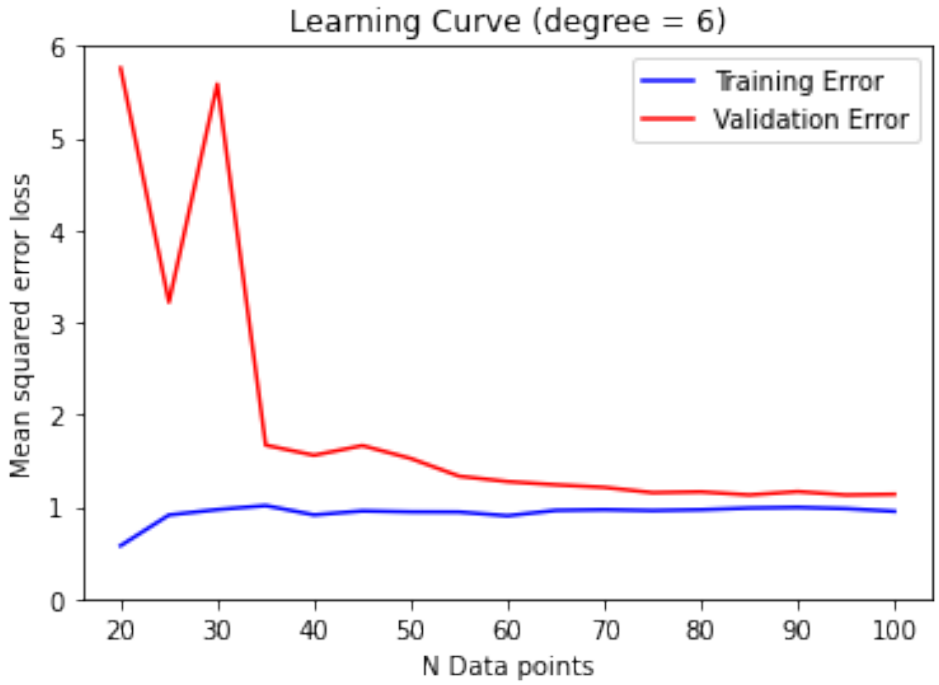
1. For each $N \in \{20, 25, 30, 35, \cdots, 100\}$:
   i. Perform 5-fold cross-validation on the first $N$ points in the dataset (setting aside the other points), computing the both the training and validation error for each fold.
      - Use the mean squared error loss as the error function.
     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

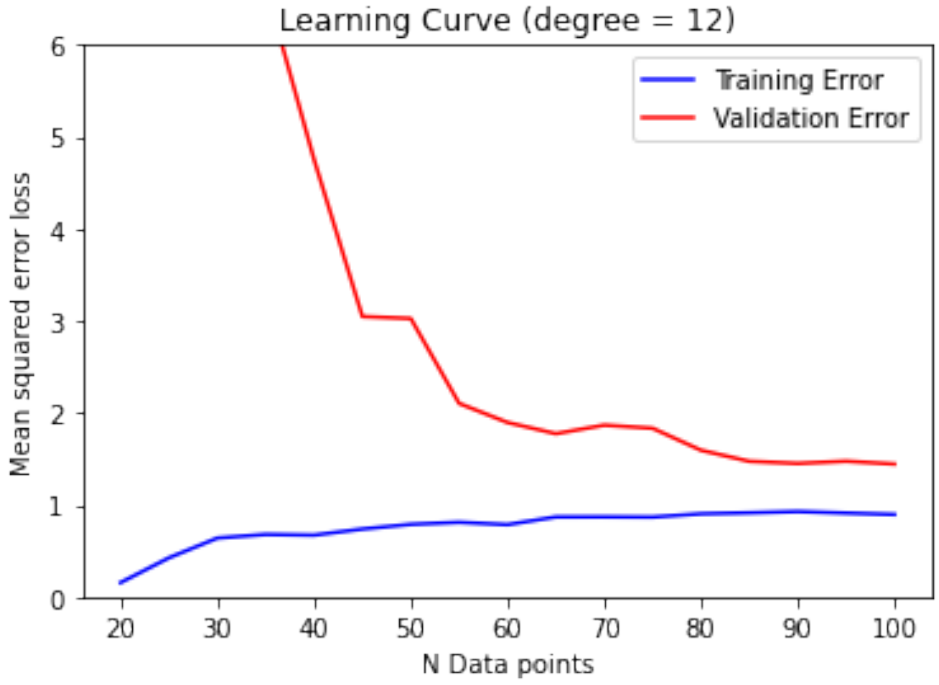
**Solution B:**

*Problem 2 Code*


Learning Curve (degree = 1)
Training Error
Validation Error
Mean squared error loss
N Data points
0
1
2
3
4
5
6
20
30
40
50
60
70
80
90
100


Learning Curve (degree = 2)
Training Error
Validation Error
Mean squared error loss
N Data points
0
1
2
3
4
5
6
20
30
40
50
60
70
80
90
100




**Question 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 for degree 1 seems to have the highest bias because we see evidence of underfitting as the training error and validation error converge to the highest error value relative to the other models.*

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

**Solution D:** *The polynomial regression model for degree 12 seems to have the highest variance because of the high validation error relative to the training error (overfitting), particularly with a lower number of data points. OVERFITTING*

**Question E [3 points]:** What does the learning curve of the quadratic model tell you about how much the model will improve if we had additional training points?

**Solution E:** *The learning curve of the quadratic model tells us that the model's improvement greatly slows as the number of training points reaches around 40-50. Basically, the model will not improve that much more with additional training points after the* $N \approx 45$ *range.*

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

**Solution F:** *The training error is generally lower than the validation error because the model is constructed and fitted according to the training data, so it inherently fits the training data better in general, thus producing lower error relative to the testing data. On the other hand, the testing data is not used to train the model, explaining why the model is generally more accurate for the training data.*

**Question 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 polynomial regression model for degree 6 to perform best in this case because as the number of data points increases towards* $N = 100$*, the validation error is the lowest relative to the other models. This implies that it would perform the most accurately for more unseen data assuming it is from the same distribution as the training data.*

## 3 Stochastic Gradient Descent [36 Points]

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:

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

**Question 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 should include an additional element in both* $\boldsymbol{w}$ *and* $\boldsymbol{x}$*. To further explain, we can add a "dummy feature" such that* $x^{(0)} = 1$ *for all inputs in* $\boldsymbol{x}$ *such that* $w^{(0)}$ *represents the bias.*

Linear regression learns a model by minimizing the squared loss function $L$, which is the sum across all training data $\{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)\}$ of the squared difference between actual and predicted output values:

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

**Question 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:**

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

$$= \sum_{i=1}^{N} 2(y_i - \vec{w}^T\vec{x}_i)\cdot(-\vec{x}_i)$$

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

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.

**Question 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.*

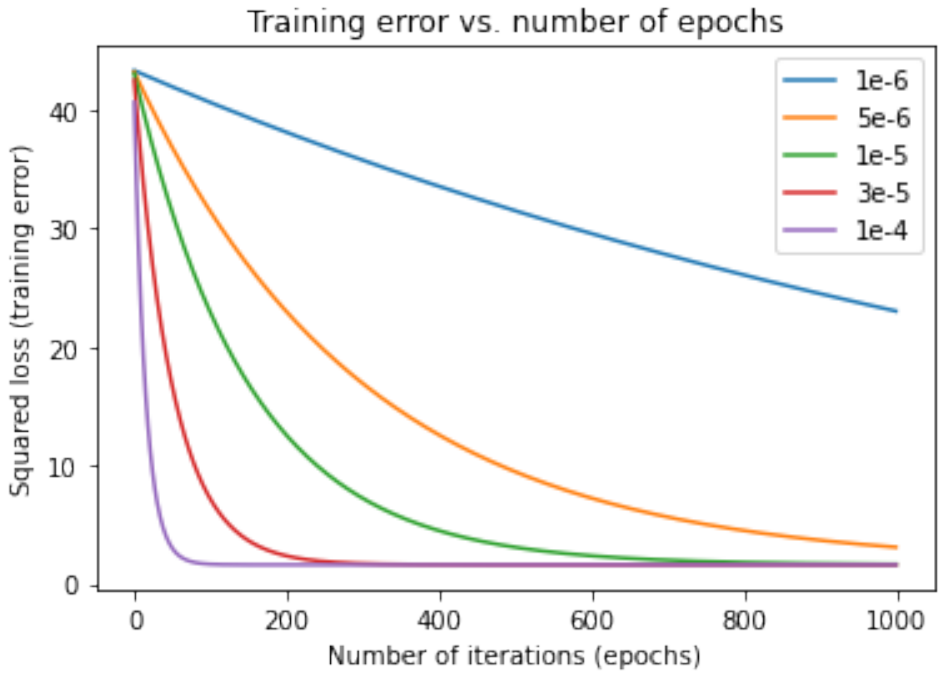
*Problem 3 Code (part 1)*

**Question D [2 points]:** Run the visualization code in the notebook corresponding to problem D. How does the convergence behavior of SGD change as the starting point varies? How does this differ between datasets 1 and 2? Please answer in 2-3 sentences.

**Solution D:** *Even with varying starting points, we see that there is always convergence in a direct line to the global minimum in this case. Thus, we can see that general convergence behavior is very similar and this trend is consistent between datasets 1 and 2 except for the fact that each dataset will follow its own "path".*

**Question E [6 points]:** Run the visualization code in the notebook corresponding to problem E. One of the cells—titled "Plotting SGD Convergence"—must be filled in as follows. Perform SGD on dataset 1 for each of the learning rates $\eta \in \{$1e-6, 5e-6, 1e-5, 3e-5, 1e-4$\}$. On a single plot, show the training error vs. number of epochs trained for each of these values of $\eta$. What happens as $\eta$ changes?

**Solution E:**




*We can see that as $\eta$ increases, SGD converges earlier/faster as it takes less epochs. Our plot also shows that convergence does not occur within 1000 epochs for our lowest range of $\eta$.*

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.

**Question 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:**

*Problem 3 Code (part 2)*

*We have that the bias is* $-0.22720591$ *and the final weight vector is*

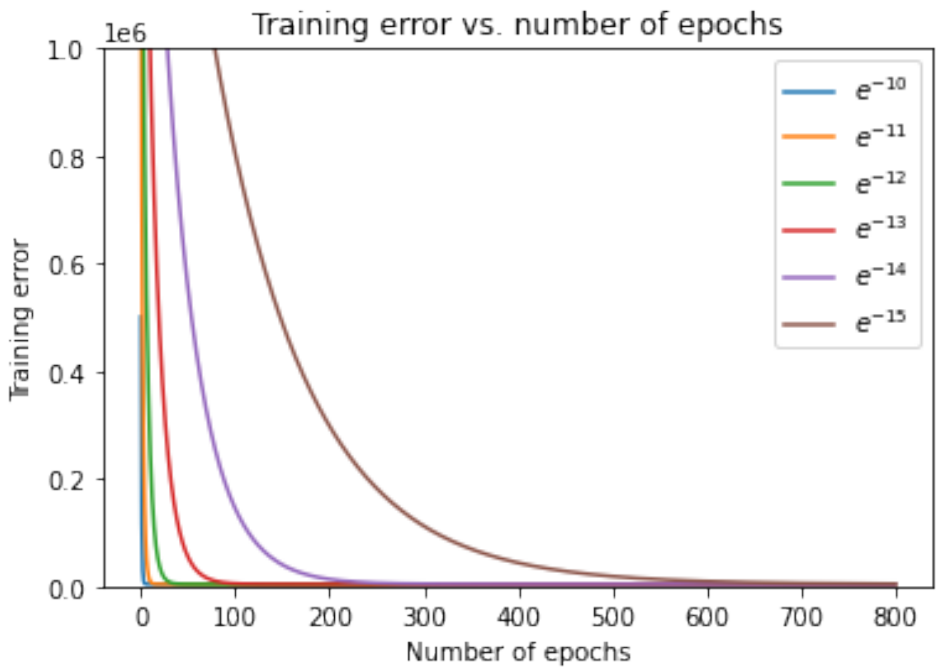
$$[-5.94229011 \quad 3.94369494 \quad -11.72402388 \quad 8.78549375]$$

**Question G [2 points]:** Perform SGD as in the previous problem for each learning rate $\eta$ in

$$\{e^{-10}, e^{-11}, e^{-12}, e^{-13}, e^{-14}, e^{-15}\},$$

and calculate the training error at the beginning of each epoch during training. On a single plot, show training error vs. number of epochs trained for each of these values of $\eta$. Explain what is happening.

**Solution G:**




*We can see that again as* $\eta$ *increases, SGD converges earlier/faster as it takes less epochs. We also see no divergence. To further explain, with smaller step sizes* $(\eta)$*, it takes more iterations for SGD to converge.*

**Question 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 bias is* $-0.31644251$ *and the final weight vector is*

$$[-5.99157048 \quad 4.01509955 \quad -11.93325972 \quad 8.99061096]$$

*We can see that these values match up very well from what we got with SGD.*

Answer the remaining questions in 1-2 short sentences.

**Question I [2 points]:** Is there any reason to use SGD when a closed form solution exists?

**Solution I:** *Yes, there is still a reason to use SGD even though a closed form solution exists because it is less complex/expensive to use SGD especially as the complexity/size of the data increases.*

**Question 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 see that training error reaches 0 essentially as SGD converges with the number of epochs increases, so instead of using defined number of epochs as a stopping condition, perhaps we could use a threshold which tracks the change in error at every epoch. To further explain, at each epoch, we can compare the change in error to the initial error, and choosing the appropriate threshold will allow stopping to occur properly by using error.*

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

**Solution K:** *With the SGD algorithm, the weight vector gradually converges to a certain point (except in case where step size gets too large in which it will diverge). On the other hand, with the perceptron algorithm, the weight vector basically oscillates back and forth until it converges (if it will even reach convergence).*

# 4 The Perceptron [14 Points]

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 $t^{\text{th}}$ iteration. This process continues until all points are classified correctly.

The following few problems ask you to work with the provided Jupyter notebook for this problem, titled `4_notebook.ipynb`. This notebook utilizes the file `perceptron_helper.py`, but you should not need to modify this file.

**Question 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.

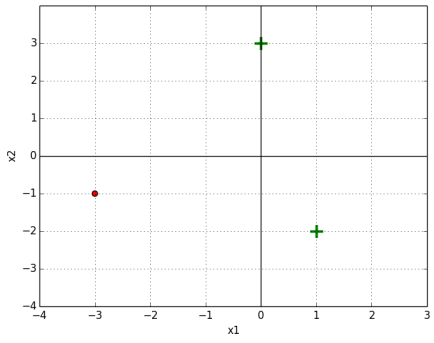



Figure 1: The green $+$ are positive and the red $\circ$ is negative

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.

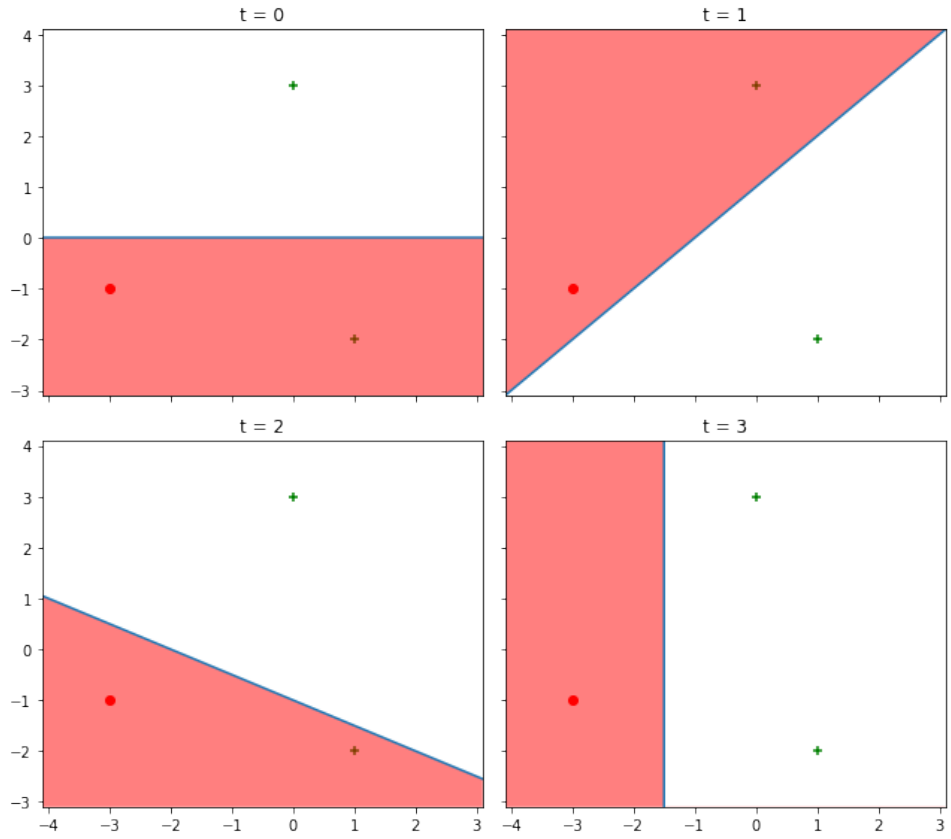
| $t$ | $b$ | $w_1$ | $w_2$ | $x_1$ | $x_2$ | $y$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | -2 | +1 |
| 1 | 1 | 1 | -1 | 0 | 3 | +1 |
| 2 | 2 | 1 | 2 | 1 | -2 | +1 |
| 3 | 3 | 2 | 0 | | | |

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:**

*Problem 4 Code*

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




**Question 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.

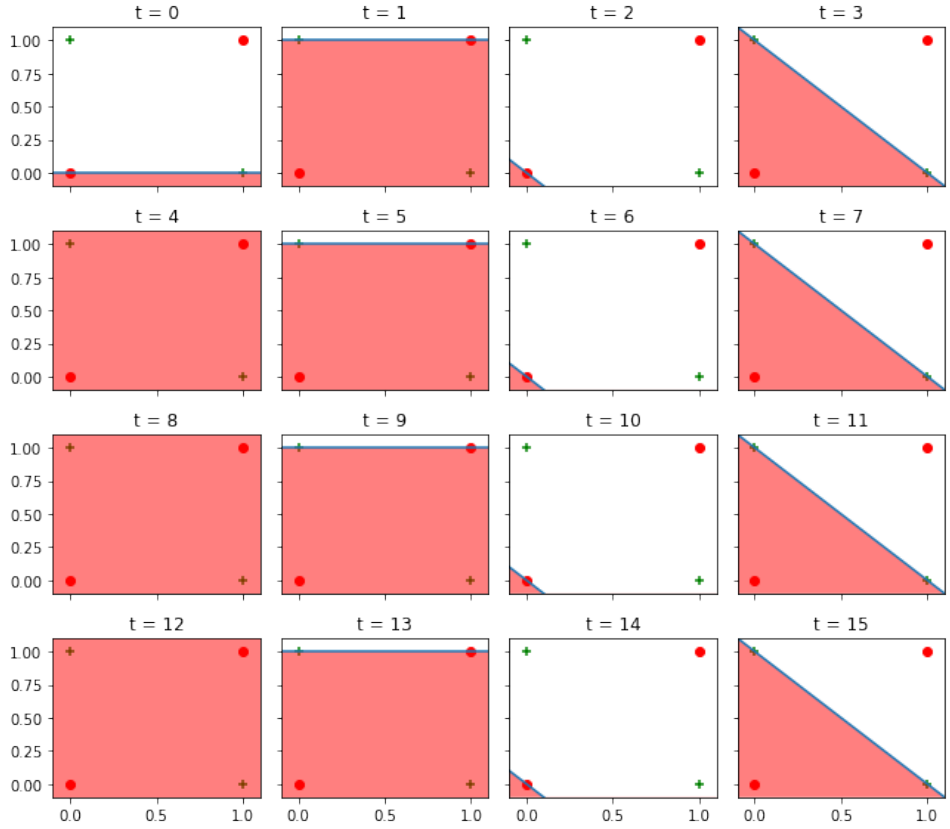
Finally, how does this generalize for an $N$-dimensional set, in which **no** $<N$-dimensional hyperplane contains a non-linearly-separable subset? For the $N$-dimensional case, you may state your answer without proof or justification.

**Solution B:** *In a 2D dataset, a dataset consisting of 4 points is the smallest dataset that is not linearly separable because any two points would be collinear, so we can imagine that a line between two positive points that "splits" the other two negative points would give us a scenario in which there does not exist a hyperplane that separates the positive and negative points from each other. In a 3D dataset, a dataset consisting of 5 points is the smallest dataset that is not linearly separable because any three points would be coplanar, so by the same logic, we can*

*imagine that a plane between 3 positive points that "splits" the other two negative points would give us a scenario in which there does not exist a hyperplane that separates the positive and negative points from each other. Thus, for an $N$-dimensional set, a dataset of $(N+2)$ points is the smallest dataset that is not linearly separable.*

**Question 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 will never converge because there will always be misclassified points since it is not possible to divide linearly inseparable points. Thus, the algorithm will continually update the weights and bias.*