Shirley Li
1/13/2023

**Turned In One hour Late **

Policies

- Due 9 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.

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 → 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 a space for mapping inputs to outputs that can be searched. This is often constrained by the choice of the framing of the problem, the choice of model and the choice of model configuration.*

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 formulas in the form* $f(x|w,b) = w^t x - b$.

Problem C [2 points]: What is overfitting?

Solution C: *Overfitting occurs when the test error is much larger than the training error. In other words, the machine leaerning model gives accurate predictions for training data but not for new data.*

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

Solution D: *Overfitting could be prevented by having more training data. Another way is by early-stopping using validation.*

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 used to learn a hypothesis from whereas the test data is used to check how the learned hypothesis performs. You should never change your model based on information from test data because it would lead to bad generalization instead of improving the model. It would actually fit the validation set even less if you fit to specific points in the test data.*

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

Solution F: *The two assumptions that we make about how our data set is sampled is that number one it approximates the true distribution of the population that we are trying to learn from and number two we assume that the data set is sampled randomly.*

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 X could be the bag of words which breaks the email down into a vector. The output space Y could be a binary indicator [0,1] to show whether email is spam or not spam.*

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

Solution H: *The k-fold cross-validation procedure is when data is split into 5 equal partitions and we train on 4 partitions and 1 partition is used as the validation set. This method allows for re-using training data as test data and allows using all data as validation. The training and validation set are sampled independently from same distribution.*

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:

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

Solution A:

$E\_S\left[E_{out}\left(f_S\right)\right]$
$= \mathbb{E}_S[\mathbb{E}_x\left[\left(f_S(x) - y(x)\right)^2\right]]$
$= \mathbb{E}_x[\mathbb{E}_S\left[\left(f_S(x) - y(x)\right)^2\right]]$
$= \mathbb{E}_x[\mathbb{E}_S\left[\left(f_S(x) - F(x) + F(x) + y(x)\right)^2\right]]$
$= \mathbb{E}_x[\mathbb{E}_S[(f_S(x) - F(x))^2]] + (F(x) - y(x))^2 + 2(f_s(x) - F(x))(F(x) - y(x))]$
$= \mathbb{E}_x[\mathbb{E}_S[(f_S(x) - F(x))^2]] + \mathbb{E}_x[(F(x) - y(x))^2] + 0$
$= \mathbb{E}_x[Var(x) + Bias(x)]$

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.

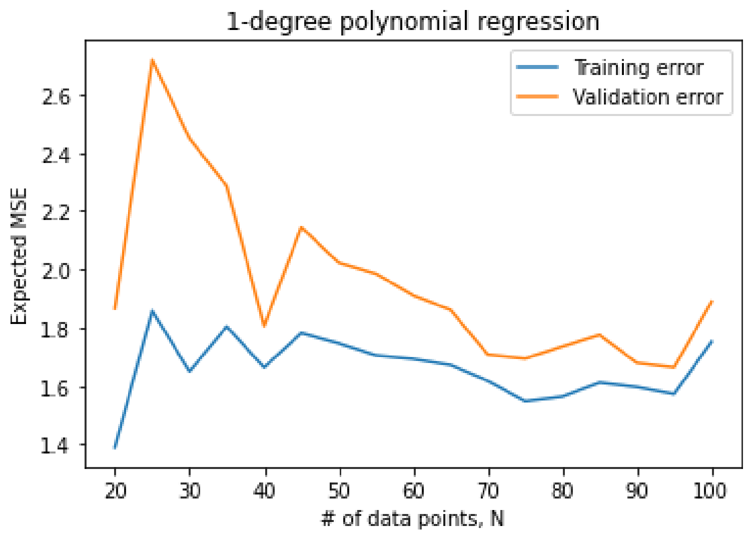
Problem B [14 points]: Use the provided `2_notebook.ipynb` Jupyter notebook to enter your code for this question. This notebook contains examples of using NumPy's polyfit and polyval methods, and scikit-learn's KFold method; you may find it helpful to read through and run this example code prior to continuing with this problem. Additionally, you may find it helpful to look at the documentation for scikit-learn's learning_curve method for some guidance.

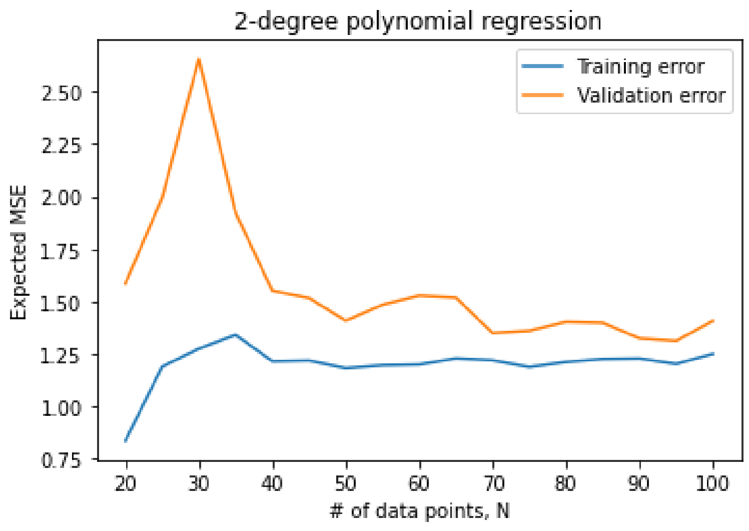
The dataset `bv_data.csv` is provided and has a header denoting which columns correspond to which values. Using this dataset, plot learning curves for 1st–, 2nd–, 6th–, and 12th–degree polynomial regression (4 separate plots) by following these steps for each degree $d \in \{1, 2, 6, 12\}$:

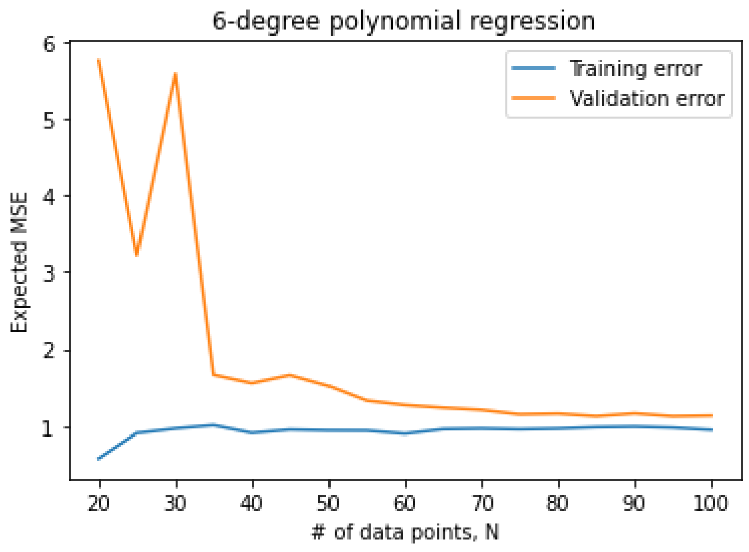
1. For each $N \in \{20, 25, 30, 35, \cdots, 100\}$:
   i. Perform 5-fold cross-validation on the first N points in the dataset (setting aside the other points), computing the both the training and validation error for each fold.
      - Use the mean squared error loss as the error function.
      -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

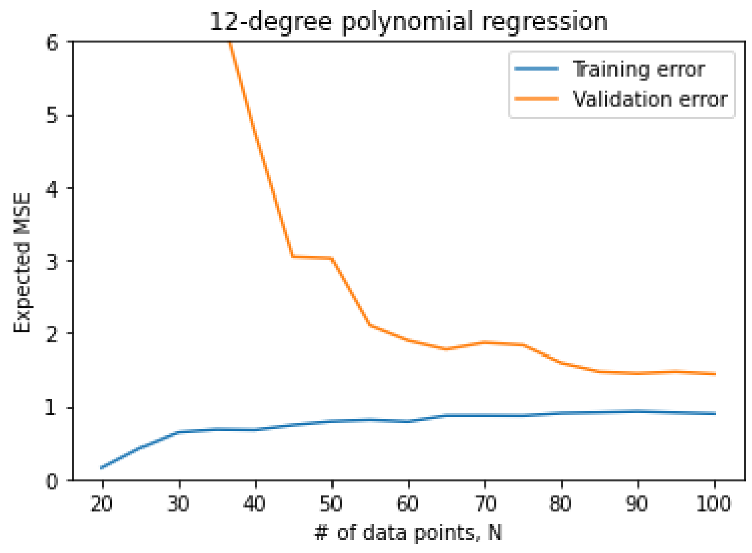
   ii. Compute the average of the training and validation errors from the 5 folds.
2. Create a learning curve by plotting both the average training and validation error as functions of N.
   Hint: Have same y-axis scale for all degrees d.

Solution B: *link:*
https://colab.research.google.com/drive/1CCkC5NPvVAEN9vMP8p6vhLOx6sJyI9gQ?usp=sharing




2-degree polynomial regression
Training error
Validation error
Expected MSE
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50
20
30
40
50
60
70
80
90
100
# of data points, N


6-degree polynomial regression
Training error
Validation error
Expected MSE
1
2
3
4
5
6
20
30
40
50
60
70
80
90
100
# of data points, N



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: *Based on the learning curves, the 1st degree polynomial has the highest bias because it has the highest error for training data as compared to the other three models. This is likely because bias decreases with model complexity. Since this polynomial has the lowest model complexity, it shows that this model is under-fitted.*

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

Solution D: *Polynomial with degree 12 has the highest variance because the validation error is the highest as compared to the other three models. This means that it is over-fitted. In other words, the model is fitting too much to things such as noise, random error, etc. that it stranded away from estimating the true function. This also agree with the fact that high variance comes with increase model complexity.*

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: *What the learning curve of the quadratic model tells me about how much the model will improve if we had additional training points is that: training data points seem to plateau more and more after 50 training* *points. This means that it does not necessarily get too much better approximation after 50 training points.*

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

Solution F: *The training error is generally lower than validation error because the model is trained on the training set so it fits the training set better since it is optimized on it. Since the model has not been trained on the validation set, it makes sense that the validation error would be higher than the training error since the model has not been optimized for the validation set.*

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: *Based on the learning curves, the model that would perform best on unseen data drawn from the same distribution as the training data would be the 6 degree model since it has the lowest validation error with enough data points.*

3 Stochastic Gradient Descent [34 Points]

Relevant materials: lecture 2

Stochastic gradient descent (SGD) is an important optimization method in machine learning, used everywhere from logistic regression to training neural networks. In this problem, you will be asked to analyze gradient descent and implement SGD for linear regression using the squared loss function. Then, you will analyze how several parameters affect the learning process.

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

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

Solution A:

Assume not. So, let's assume that $\nabla f(x') \neq 0$. Then $\lim_{h \to^+ 0} \frac{f(x+h)-f(x)}{h} = f'(x)$.

Assume, $f'(x) > 0$. In this part, we go from the left to the right, i.e. $h < 0$. Therefore, by the definition of the limit, there should exist some δ, s.t. for h sufficiently small enough, $\frac{f(x+h)-f(x)}{h}$ is not that far away from $f'(x)$. In other words, for h sufficiently close to 0, we should have $\frac{f(x+h)-f(x)}{h} > \frac{9}{10} \cdot f'(x)$. It is equal to the statement that for h sufficiently close to 0, we should have $f(x+h) < \frac{9}{10} f'(x) \cdot h + f(x)$. Now, let's notice that $h < 0$, $f'(x) > 0$ by the assumption in this paragraph. So, we basically have shown that $f(x+h) <$(smth negative)$+f(x)$. In particular, $f(x+h) < f(x)$. Contradiction! $f(x)$ was assumed to be a local minimum.

Assume, $f'(x) < 0$. In this part, we go from the right to the left, i.e. $h > 0$. By the same idea as above, starting some h sufficiently close to 0, $\frac{f(x+h)-f(x)}{h} < \frac{9}{10} \cdot f'(x)$. So, $f(x+h) < \frac{9}{10} \cdot h \cdot f'(x) + f(x)$. Note, $h > 0$, $f'(x) < 0$. Therefore, $f(x+h) <$ (smth negative) $+ f(x)$. In particular, $f(x+h) < f(x)$. Contradiction! $f(x)$ was assumed to be a local min.

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: *We can add* 1 *additional element to the vector* w *and to the vector* x. *So, instead of* $w_1, w_2, ...$ *we will have* $w_0, w_1, w_2, ...$, *and instead of* $x_1, w_2, ...$, *we will have* $x_0, x_1, x_2, ...$. *Also, we set* $x_0 = 1$. *Therefore, our* $w_0 \cdot x_0$ *term will play the same role as* b *played before.*

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: $\frac{dL}{dw} = \sum_{N}^{i=1} 2 \cdot (y_i - w^T x_i) \cdot (-x_i)$. *It is because when I take a derivative w.r.t. a matrix,* $\frac{d(x'b)}{dx} = b$. *In this case,* $x = w$, $b = x_i$.
No doubts, SGD is much cheaper than the GD. The reason is that for one iteration of the GD algorithm, we need to find $(y_i - \mathbf{w}^T \mathbf{x}_i)^2$ N *times, and then to sum all it together. However, for the SGD we need to generate a random number* j *between* 1 *and* n, *which is cost-less and to find only once* $L_j(w, b)$. *So, SGD should be approximately* N *times more time-efficient (i.e. it should take approx.* N *times less time to run an algorithm).*

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.

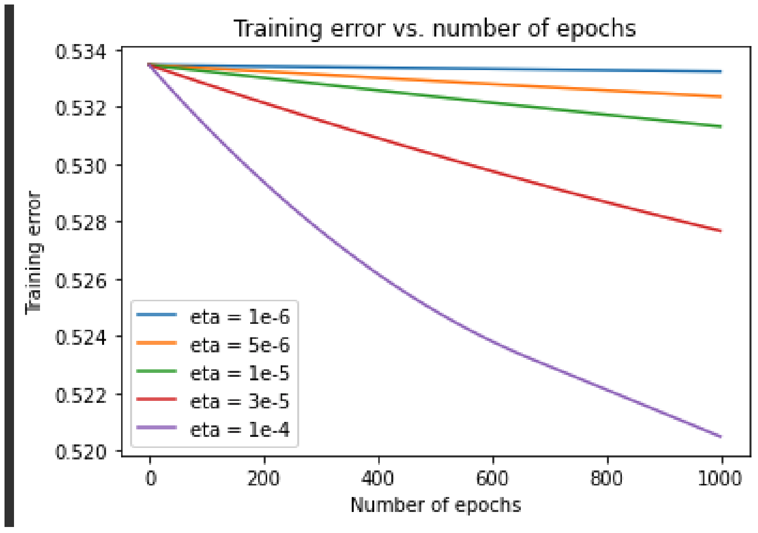
Solution D: *See code. Google Colab link:*
https://colab.research.google.com/drive/1dSJmmsjJD4LWDBW9EcwIBltFtFp9F8gK?usp=sharing

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: *It seems like regardless of the starting point, the convergence behavior of SGD is that it always converges to the global minimum. This behavior is the same between datasets 1 and 2.*

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:



SGD shows to converge faster as eta increases. And it does not converge within 1000 epochs with low values 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 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.

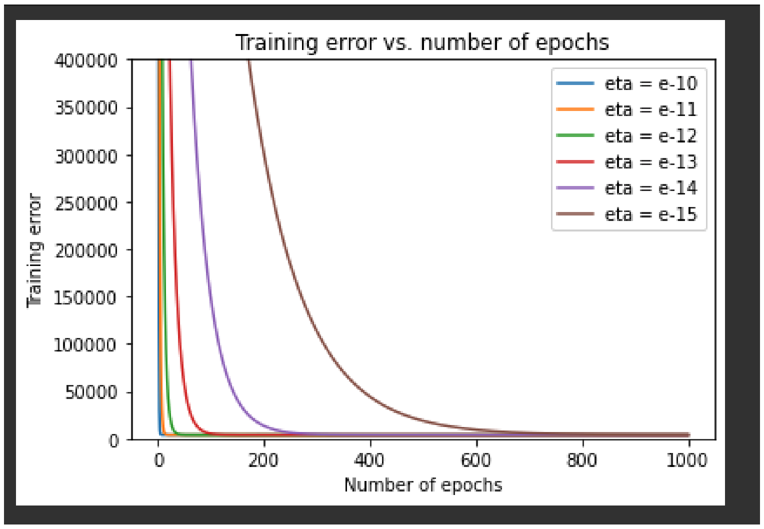
Solution G: *Google Colab:*
https://colab.research.google.com/drive/1Yb-NvP4jgqL4oFHA-YudOvkDOYVFowji?usp=sharing
My final weight vector is [-0.31644251 -5.99157048 4.01509955 -11.93325972 8.99061096].

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:



SGD shows to take less epochs and converges faster as η increases. With low values of η, it does not converge within 1000 epochs. For a comparison between learning rates, we understand that the smaller learning rates take more epochs to get the training error down.

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: *Using the closed form solution, the final weight vector is [-0.31644251 -5.99157048 4.01509955 -11.93325972 8.99061096]. This is fairly close to what I got from 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: *One reason to use SGD when a closed form solution exists is that computing the closed form solution could be very computationally expensive as the size of the data increases.*

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: *A stopping condition that is more sophisticated than a pre-defined number of epochs could be that we stop when the change in training error is small so that the progress is sufficiently small.*

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

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

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

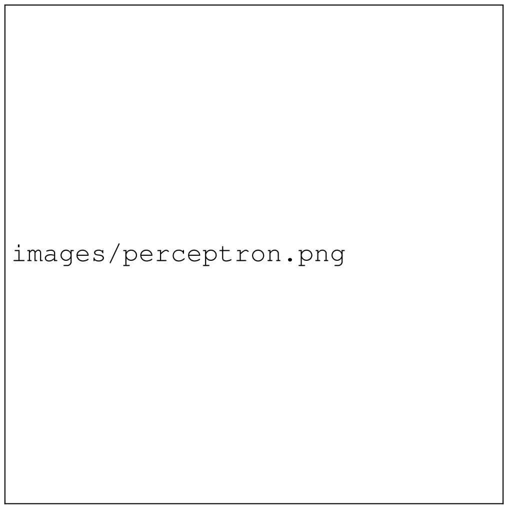


Figure 1: The green $+$ are positive and the red $\circ$ is negative

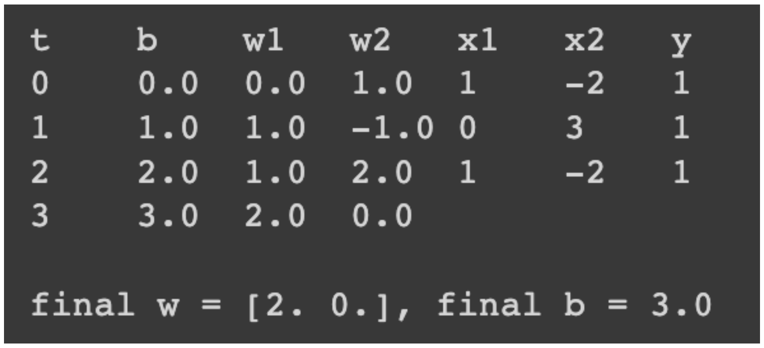
Implement the `update_perceptron` and `run_perceptron` methods in the notebook, and perform the perceptron algorithm with initial weights $w_1 = 0, w_2 = 1, b = 0$.

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			

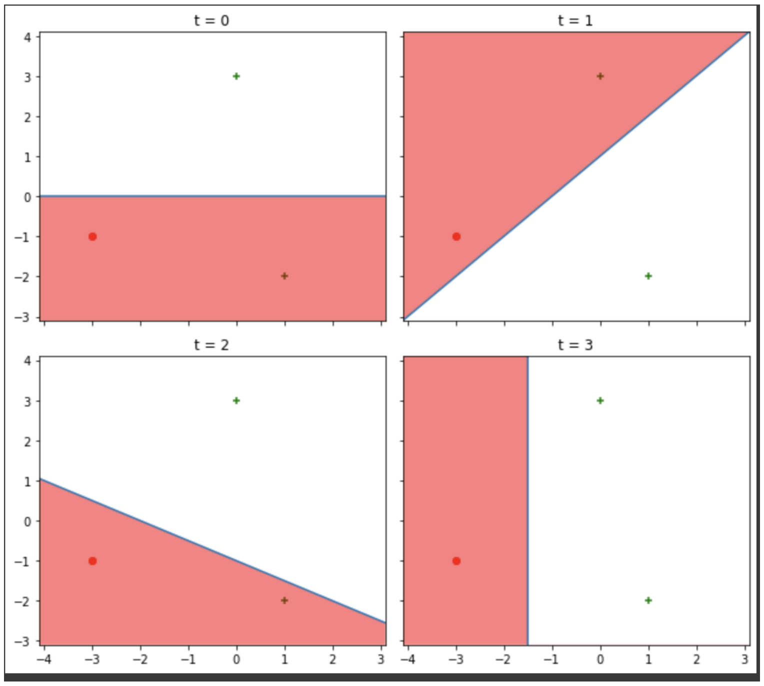
Include in your report both: the table that your code outputs, as well as the plots showing the perceptron's classifier at each step (see notebook for more detail).

Solution A: *Google colab:*
https://colab.research.google.com/drive/1Ra0gnzDf1AMRmTlMtpAdoAu_sJOIivci?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

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



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

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

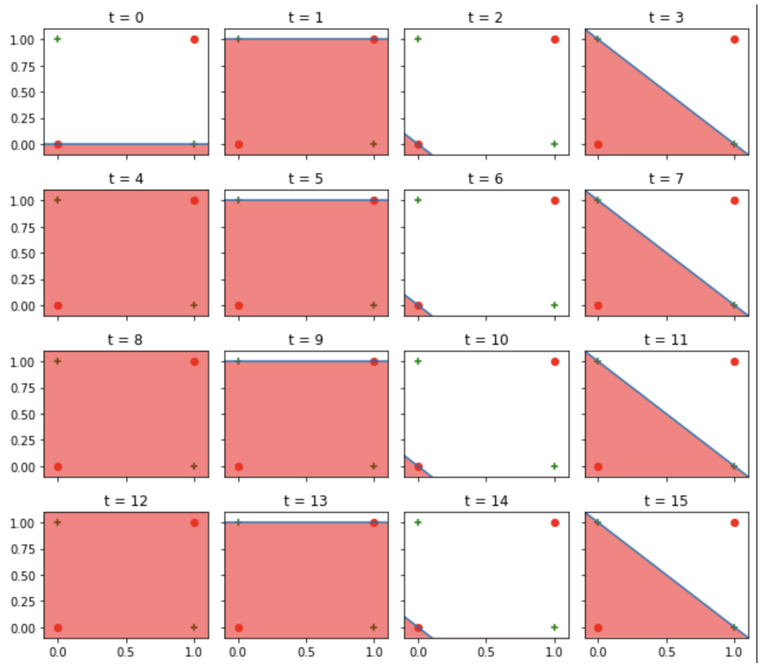
Finally, how does this generalize to N-dimension? More precisely, in N-dimensional space, what is the minimum size of a dataset that is not linearly separable, such that no $N + 1$ points are on the same hyperplane?

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

Solution B: *Any two points in a 2D space are collinear and a dataset of 4 points is the smallest that is not linearly separable in 2D input space. So if a line formed by 2 points of the same sign separates 2 points of a sign opposite from them, there is no way to separate such 4 points. A dataset of 5 points is the smallest that is not linearly separable in 3D input space, since any 3 points in a 3D space would be coplanar, and if a plane formed by 3 points of the same sign divides 2 points of a sign opposite from them, there would be no way to separate these 5 points. With this logical argument, a dataset of N+2 points is the smallest that is not linearly separable.*

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:



Assuming that a dataset is not linearly separable, the Perceptron Learning Algorithm will never converge because it is impossible to fit a hyperplane diving the linearly inseparable points.

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: *The weight vector fluctuates as the perceptron algo converges if at all, meanwhile the SGD is guaranteed to coverge smoothly wihtout diverging.*