Caltech CS/CNS/EE 155
Jayden Nyamiaka

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: A hypothesis set is the set of mathematical models/functions we could use to model our data given some training set distribution.

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 models.functions including all the different possible weights that could be used to model our data. Using some optimization, we can narrow down which specific linear model best approximates our data.

Problem C [2 points]: What is overfitting?

Solution C: Overfitting is when a model fits the training set data so well, but in practice, it functions poorly on some new data it hasn't seen before. In these cases, the model would show very little error in approximating data points from the training set but reveal error when approximating data points outside the training set. This often happens if the distribution of the training set poorly represents the true distribution such that patterns in the training data that aren't present in the real world are trained into the model.

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

Solution D: Overfitting can be prevented by limiting variance within our model classification. This often means choosing a simpler mathematical model like a linear model opposed to a cubic model because with less degrees of freedom, overfitting is less possible.
Additionally, if the training set distribution is biased such that it's misrepresentative of the true distribution, there will be cases that the model was never trained for, causing overfitting. Thus, by carefully making sure our training data reflects the true distribution or perhaps using more data in training, overfitting can be prevented.
Lastly, there are training techniques that can be used to prevent overfitting like cross validation; specifically, in class we discussed K-Fold Cross-Validation. By splitting up our training data into K different training partitions, we can reuse all our training data as test data. This would give the effect of more data than we actually have and let us validata/test our model on unseen data, thus limiting overfitting.

Problem E [2 points]: What are training data and test data, and how are they used differently? Why should you never change your model based on information from test data?

Solution E: Training data is used to mathematically get the weights and parameters of your model by updating them based on the error your model exhibits when approximating points incorrectly. Test data is new data the model has never encountered before used to assess the correctness/bias of your model once it has been well trained on the training data.
If we use the test data to train the model, then the model would be optimized so that it approximates the test data and would obviously perform well during the "test", so we wouldn't actually be "testing" anything. To actually assess our model with the test data, it must be data the model has never encountered.
Additionally, the training data is supposed to be representative of the true distribution so that after training the model with the training data, the model can accurately predict data over the true distribution. So, altering the model anymore after that point would only make the model biased towards the test data and less accurate since the test data may not reflect the true distribution.

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

Solution F: We assume that the distribution of our data set is representative of the true distribution of data, and we assume that our data set is large enough where each data point is independent of other data points. We also assume that our data set contains a relationship between our identified independent and dependent variables that can be modeled with a mathematical function we can optimize.

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 is governed by any possible email that could be received, so for the purposes of our model, we can view any possible email as any formatted text file. Consider that we have a long list of "relevant words" that our model cares about. We could construct the input for our model by parsing a text file into a list of integers for our relevant words where each integer in the list matches to a relevant word and its value is the number of times that relevant word appears in the email. Since we parse any email into this list before feeding it to our model, the input space X would be any possible list of integers of length equal to the number of relevant words. The model needs to predict if the email is spam or not, so output space could just be either a 1 or 0, where 1 indicates the email is spam and 0 means not spam.

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

Solution H: The K-Fold Cross-Validation procedure is a way to train and test our model that prevents overfitting. First, you shuffle and split our training data into K partitions. Then, for each combination of K - 1 partitions, you train the model on all K - 1 partitions and then test the model using the remaining partition. Once done, the model would have been tested using every single partition, and it seems that we utilized more data than we actually have compared to if we used a simpler procedure. We then have a better tested model.

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,

$$\mathrm{E}_S[\,E_{\text{out}}(f_S)\,] = \mathrm{E}_x[\,\text{Bias}(x) + \text{Var}(x)\,]$$

given the following definitions:

$$F(x) = \mathrm{E}_S[\,f_S(x)\,]$$

$$E_{\text{out}}(f_S) = \mathrm{E}_x[\,(f_S(x) - y(x))^2\,]$$

$$\text{Bias}(x) = (F(x) - y(x))^2$$

$$\text{Var}(x) = \mathrm{E}_S[\,(f_S(x) - F(x))^2\,]$$

Solution A:

Show $E_s[E_{out}(f_s)] = E_x[Bias(x) + Var(x)]$

Given $F(x) = E_s[f_s(x)]$

$E_{out}(f_s) = E_x[(f_s(x) - y(x))^2]$ $\qquad E_x[Bias(x) + Var(x)]$

$Bias(x) = (F(x) - y(x))^2$ $\qquad E_x[(F(x) - y(x))^2 + E_s[(f_s(x) - F(x))^2]]$

$Var(x) = E_s[(f_s(x) - F(x))^2]$

$$E_{out}(f_s) = E_x[(f_s(x) - y(x))^2]$$

$$E_s[E_{out}(f_s)] = \underline{E_s}[E_x[(f_s(x) - y(x))^2]]$$

$$E_s[E_{out}(f_s)] = E_x[E_s[(f_s(x) - y(x))^2]]$$

$$E_s[E_{out}(f_s)] = E_x[E_s[(f_s(x) - y(x) + \underline{F(x) - F(x)})^2]]$$

$$E_s[E_{out}(f_s)] = E_x\left[E_s\left[\begin{matrix}\cancel{(f_s(x))^2} - \cancel{f_s(x)y(x)} + \cancel{f_s(x)F_x} - \cancel{f_s(x)F_x} + \cancel{(F(x))^2} + \cancel{f_s(x)F_x} - \cancel{y(x)F(x)} - \cancel{(F(x))^2} + \\ \cancel{(y(x))^2} - \cancel{f_s(x)y(x)} - \cancel{y(x)F_x} + \cancel{y(x)F_x} + \cancel{(F(x))^2} - \cancel{f_s(x)F(x)} + \cancel{y(x)F(x)} - \cancel{(F(x))^2}\end{matrix}\right]\right]$$

$$E_s[E_{out}(f_s)] = E_x\left[E_s\left[\begin{matrix}[(F(x))^2 - 2y(x)F(x) + (y(x))^2] + [(f_s(x))^2 - 2f_sF(x) + (F(x))^2] \\ -2[(F(x))^2 - y(x)F(x) - f_sF(x) + f_s(x)y(x)]\end{matrix}\right]\right]$$

$$E_s[E_{out}(f_s)] = E_x[E_s[(F(x) - y(x))^2 + (f_s(x) - F(x))^2 - 2(F(x) - y(x))(F(x) - f_s(x))]]$$

$$E_s[E_{out}(f_s)] = E_x[\cancel{E_s}[(F(x) - y(x))^2] + E_s[(f_s(x) - F(x))^2] - 2\cancel{E_s[(F(x) - y(x))(F(x) - f_s(x))]}]$$

$$E_s[E_{out}(f_s)] = E_x[Bias(x) + Var(x)] \quad \square$$

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 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 the same y-axis scale for all degrees d.*

Solution B:
https://colab.research.google.com/drive/18hoWBFudGpHia7LlhHXeQM2ka4D8GEtf?usp=sharing

Problem C [3 points]: Based on the learning curves, which polynomial regression model (i.e. which degree polynomial) has the highest bias? How can you tell?

Solution C: The linear model (degree = 1) has the highest bias because it has the highest training and validation error. You can see that as we test the graph with more points, the training and validation error of the linear model at 100 points is the highest (approximately E = 1.8 using average squared loss) compared to the other models. Its high bias also shows because unlike other models, the linear model is the only one to show increasing error when adding more points. This is likely because the linear model doesn't have enough degrees of freedom, so it's oversimplifying the relationship.

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

Solution D: The degree 12 model has the highest variance because it has the lowest training error but highest validation error. You can tell because the testing error is always much higher than the training error, indicating that overfitting is occurring, and overfitting happens with models with too much variance. Additionally, when training with fewer points (20-40), you can see on the graph that the training error is super low, but the testing error is super high. This is because the many degrees of freedom of the model allow it to capture all the patterns of the training data too perfectly such that it also captures noise, and then these noise patterns make it perform very poorly on testing data it's never seen before.

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: It shows the model will likely do worse if fed more training points. After 95 points, the model shows more training and validation error given more points, which is likely because the degrees of freedom / variance of the quadratic model aren't enough to capture the relationship. Because they aren't enough, training with more points will show the model does poorly.

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

Solution F: The model was made to optimize the training data specifically, so of course, the error is going to be lower. Validation error from testing data is higher because that testing data is data the model has never seen before. Although both the training and testing data correspond to the same relationship, the distribution of the training data is most likely going to be different than the distribution of the testing data such that some patterns are more emphasized in one set of data than the other. This happens in sampling because realistically it's impossible to get data that perfectly represents the true distribution. Thus, the model is optimized for the distribution and patterns of the training data, so when it comes across the different distribution of the unseen testing data, there will be more 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 6 model to perform best on unseen data from the same distribution as the training data. This is because it has the lowest training and validation error as we converge on more and more points. The fact that the training and validation errors of the linear and quadratic models increase after getting more points show they do not have enough variance and will oversimplify the relationship. In the degree 12 model, the testing error is always much higher than the training error showing overfitting and that the model has too much variance for the relationship. In the degree 6 model, however, the training and testing error keep getting closer and closer to each other with more training points, and they never go up after a given number of points; this shows the model has just the right variance, right in the middle, so it would perform best.

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^0 is a local minimum of *f*, then $\nabla f(x^0) = 0$.

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: *I Don't Know, Sorry*

Linear regression learns a model of the form: See GitHub

Problem B [1 points]: We can make our algebra and coding simpler by writing $f(x_1, x_2, \cdots, x_d) = \mathrm{w}^T \mathrm{x}$ for vectors w and x. But at first glance, this formulation seems to be missing the bias term *b* from the equation above. How should we define x and w such that the model includes the bias term?

Hint: *Include an additional element in* w *and* x.

Solution B: Given that there are d elements in vectors w and x, we can include an additional element in the d + 1 spot of both vectors where w_{d+1} = b and x_{d+1} = 1, so when the transpose multiplication / dot product occurs between the two vectors $w_{d+1}d_{d+1}$ = b is added to the final result adding the bias term into the our model.

Linear regression learns a model by minimizing the squared loss function *L*, which is the sum across all training data $\{(x_1, y_1), \cdots, (x_N, y_N)\}$ of the squared difference between actual and predicted output values: See GitHub

Problem C [2 points]: Both GD and SGD uses the gradient of the loss function to make incremental adjustments to the weight vector w. Derive the gradient of the squared loss function with respect to w for linear regression. Explain the difference in computational complexity in 1 update of the weight vector between GD and SGD.

Solution C:

GD	SGD
$\partial_w L(f)$	$\partial_w L(f)$ *Breaks $L(f)$ into components $L_i(f) = (y_i - \vec{w}^T \vec{x}_i)^2$ since Σ is additive
$\partial_w \sum_{i=1}^{N} (y_i - \vec{w}^T \vec{x}_i)^2$	$\partial_w E_i[N(y_i - \vec{w}^T \vec{x}_i)^2]$
$\sum_{i=1}^{N} \partial_w (y_i - \vec{w}^T \vec{x}_i)^2$	$E_i[\partial_w N(y_i - \vec{w}^T \vec{x}_i)^2]$
$\sum_{i=1}^{N} 2(y_i - \vec{w}^T \vec{x}_i)(-\vec{x}_i)$	$E_i[N \partial_w (y_i - \vec{w}^T \vec{x}_i)^2]$
$\sum_{i=1}^{N} -2\vec{x}_i(y_i - \vec{w}^T \vec{x}_i)$	$E_i[2N(y_i - \vec{w}^T \vec{x}_i)(-\vec{x}_i)]$
$-2\sum_{i=1}^{N} \vec{x}_i(y_i - \vec{w}^T \vec{x}_i)$	$-2N\vec{x}_i(y_i - \vec{w}^T \vec{x}_i)$

In one update of GD, the summation shows that you have to pass over and compute for every single point in the training data set. This is very computationally expensive. Conversely, using SGD, due to the assumption that the expected loss of every additive piece of the loss function equals the loss function, we only need to do the computation for one arbitrary training point per update iteration. Note that in the SGD derivation, the multiplied by N term is present to reflect that N terms would be summed to get the full loss function.

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:
https://colab.research.google.com/drive/14jAGZ5wuwAhmDOxp3QcmcuLMoIlNz21P?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: With the use of different starting points and data points, the convergence behavior is relatively the same. Despite the differences in where the weights start on the graph and how far they are from the lowest (optimal) point, they all take the same amount of computations to get there. The points that are farther seem to have larger changes in their weights for each update such that all SGD initializations converge to the optimum at the same time.

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: As the value of learning rate η increases, the SGD converges faster. The slope of the error becomes steeper such that the loss reaches (or gets close enough to) the optimal in less epochs. This occurs because the magnitude of change to the weights during every update is larger with larger step sizes. It doesn't show with these step sizes, but if we chose a step size that was too large, we would also see the loss diverge getting higher and higher, never reaching the optimal model.

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 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://colab.research.google.com/drive/1a5I8btc36_a_IDVxY_6qY5LPmlx2_t_8?usp=sharing

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: For each larger step size, it takes significantly less time to train the data because there is more change occurring to the weights and bias term with each update. Specifically, for each step size increasing by multiplying by e, it seems to take half the computations to reach convergence. For example, it looks like it takes about 400 epochs to reach convergence with $\eta = e^{-15}$ and about 200 epochs with $\eta = e^{-14}$.

Problem I [2 points]: The closed form solution for linear regression with least squares is

See GitHub

Compute this analytical solution. Does the result match up with what you got from SGD?

Solution I: The results match don't match exactly, but they are very close. They are the same up to 0.1 precision. When computing loss for these models however, the analytical model shows less loss than the SGD model, showing that the differences of the analytical model are due to it being a better model. This is due to the fact that SGD converges to the best weight vector stopping training after a certain number of iterations, yielding a good enough model whereas the analytical model simply computes the ideal weight vector.

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: For linear regression with least squares regression specifically, it might be best to use the analytical computation since it's faster and yields a better model; however, SGD is still very useful because many times we don't know what model class works best for our function and we're usually not using a linear model. There are many different loss functions and model classes that don't have a closed form analytical solution, so in those cases, SGD is very helpful for finding the best model. This analytical model only works for simple unique cases. Additionally, the analytically computed model may be too exact such that it overfits the data, which we wouldn't want.

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: We can stop once the model stops improving by a certain margin. Specifically, we can analyze the weight vector and see by how much the weights of the weight vector are changing each iteration by taking the gradient of the function. And once the change each iteration isn't that much, we deem it a good enough model and stop training.

4 The Perceptron [16 Points]

Relevant materials: lecture 2

The perceptron is a simple linear model used for binary classification. For an input vector $x \in R^d$, weights $w \in R^d$, and bias $b \in R$, a perceptron $f : R^d \to \{-1, 1\}$ takes the form

See GitHub

The weights and bias of a perceptron can be thought of as defining a hyperplane that divides 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 w. Then, one misclassified point is chosen arbitrarily and the w vector is updated by

$$w_{t+1} = w_t + y(t)x(t)$$
$$b_{t+1} = b_t + y(t),$$

where $x(t)$ 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 ◦ point is in the −1 class.

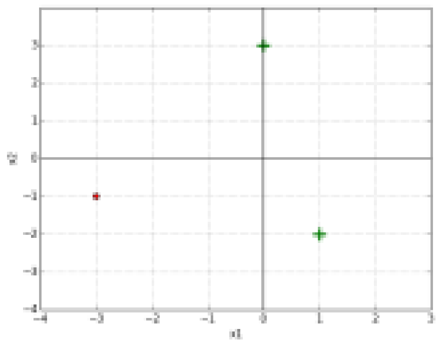


Figure 1: The green + are positive and the red ◦ is negative

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.

See GitHub

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

Solution A:
https://colab.research.google.com/drive/1CmgGinXH7ubOHXy7GaggkgI7vbdUnfP4?usp=sharing

Problem B [4 points]: A dataset $S = \{(x_1, y_1), \cdots, (x_N, y_N)\} \subset \mathbb{R}^d \times \mathbb{R}$ 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: In 2D, 3 data points along the same line are not linearly separable. However, when you add in the requirement that the data points cannot be collinear, any data set with 3 non-collinear points can be completely linearly separated.
As we can see with the graphs, however, once you add that fourth point, the dataset is no longer guaranteed to be linearly separable, showing that the minimum size of a dataset that is not linearly separable in 2D space is 4 points. In this case, there are simply not enough dimensions to split the data anymore.
In 3D, you add one more dimension to control the boundary of separation. With this one more dimension, we have another way to separate the data, and now we can handle 4 points, bringing up the minimum size of a dataset in 3D that isn't linearly separable to 5 points.
Notice here that a pattern starts to arise where as you continue to add more points and dimensions, the minimum size of a dataset in N-dimensions that can't be linearly separable is N + 2 points. This pattern holds in an induction-like way. Given the previous minimum dataset size that can't be linearly separated in N-dimensions (which would be N + 2 points), adding one more dimension (now N + 1 dimensions) allows you to now account for the degree of separation to handle those N + 2 points. Then, once you add in the N + 3 point to the dataset, the N + 1 dimensions are not guaranteed to be 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?

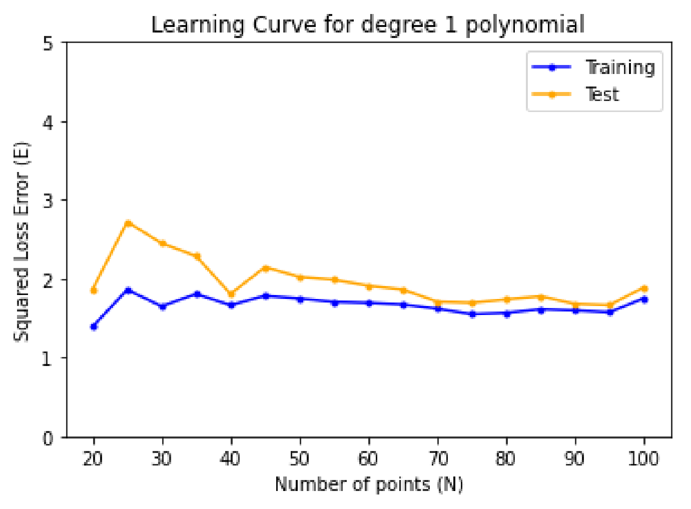
Solution C: The Perceptron Learning Algorithm will never converge because there will always be points that are misclassified. By construction of the algorithm, it stops updating the weights and bias term (AKA converges) once the model correctly classifies all the points in the dataset. If the dataset is not linearly separable, there is no model that can correctly classify all the points, so the algorithm will continue running endlessly trying to fit data that it can't model.

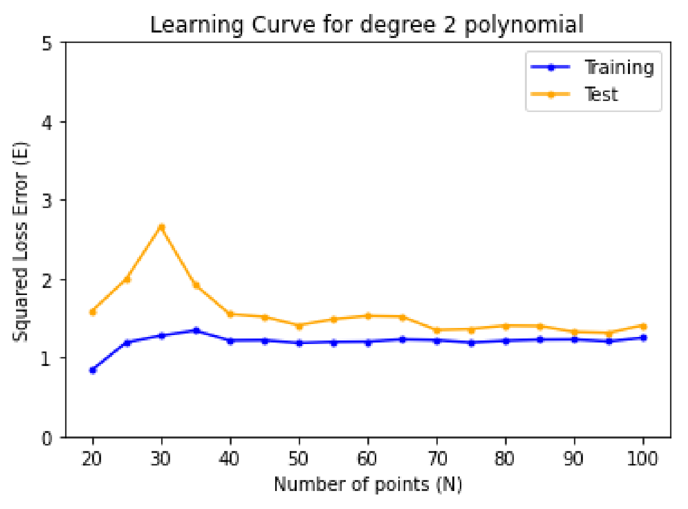
Problem D [2 points]: How does the convergence behavior of the weight vector differ between the perceptron and SGD algorithms? Think of comparing, at a high level, their smoothness and whether they always converge (You don't need to implement any code for this problem.)

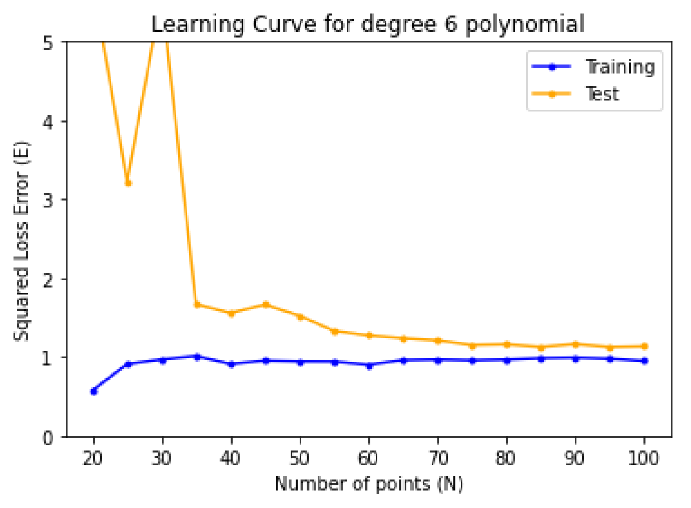
Solution D: Convergence behavior of the SGD and the perceptron mainly in the way they reach convergence. For the perceptron, the model will continue modeling and updating until the model is a perfect fit and has all points correctly classified such that it will go on forever if there is no possible correct classification and thus never converge. SGD, assuming a learning rate that isn't too large, will continuously generate a better and better model, and even if it can't perfectly model the data, it will converge at the best "good enough" point. So, even though it's not perfect, the model is the best it can possibly be compared to other models in the model class, and that's where it'll converge. The main difference is the way that SGD converges at this best model if there's no perfect model whereas the perceptron will never converge and there isn't a best model if it can't perfectly model the data.

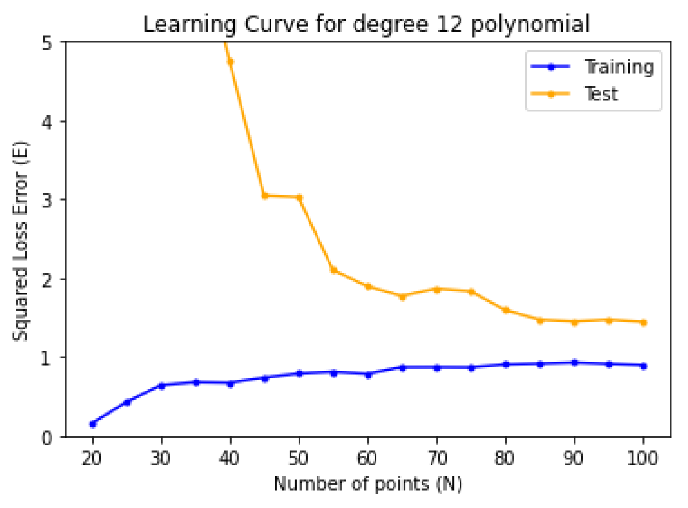
Generated Images

Part 2 B



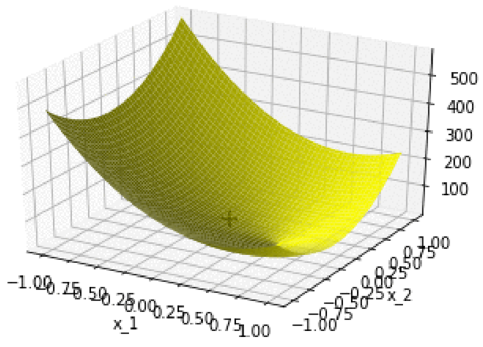




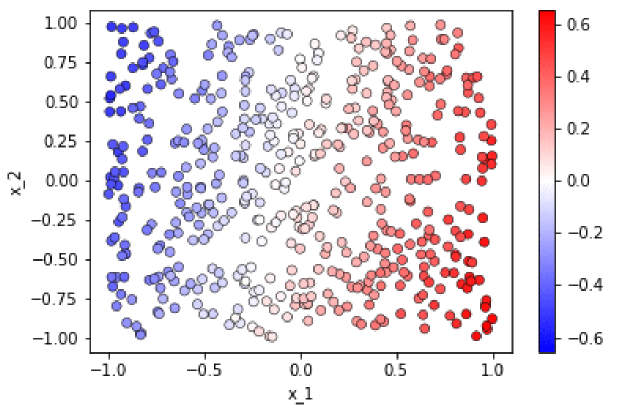


Part 3 D-F

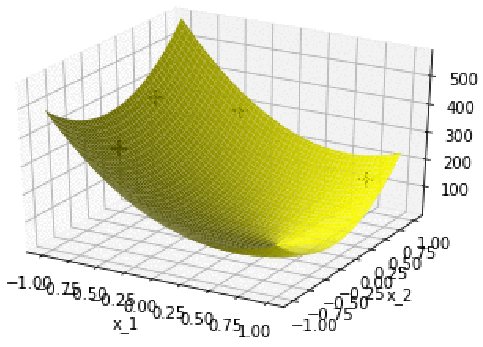
SGD from a Single Point



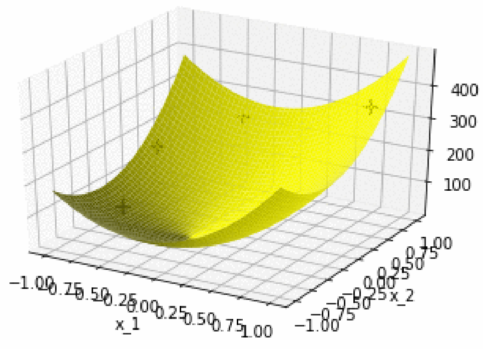
Animate Modeling of SGD from a Single Point



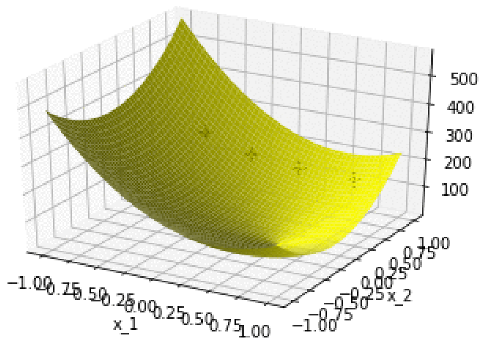
SGD from Multiple Points



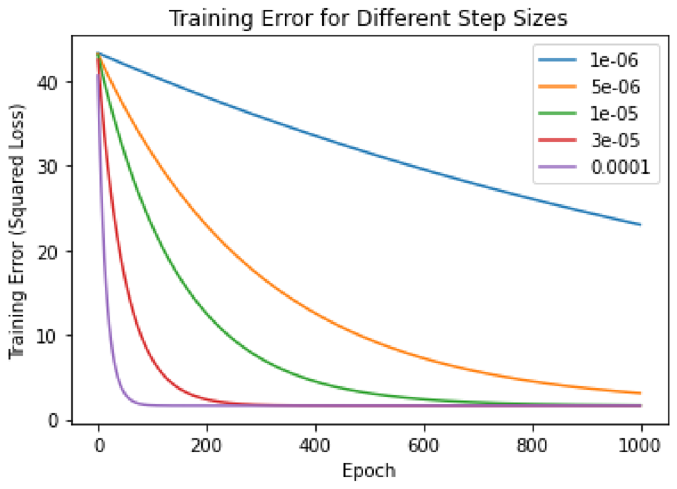
SGD from Multiple Points (Different Dataset)



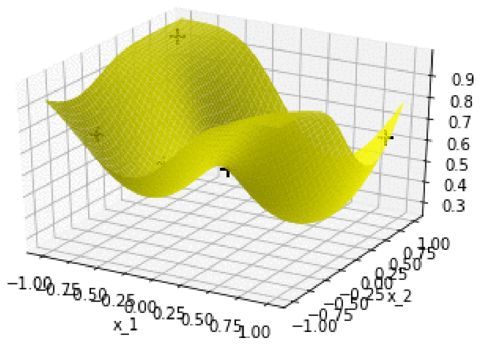
SGD with Different Step Sizes



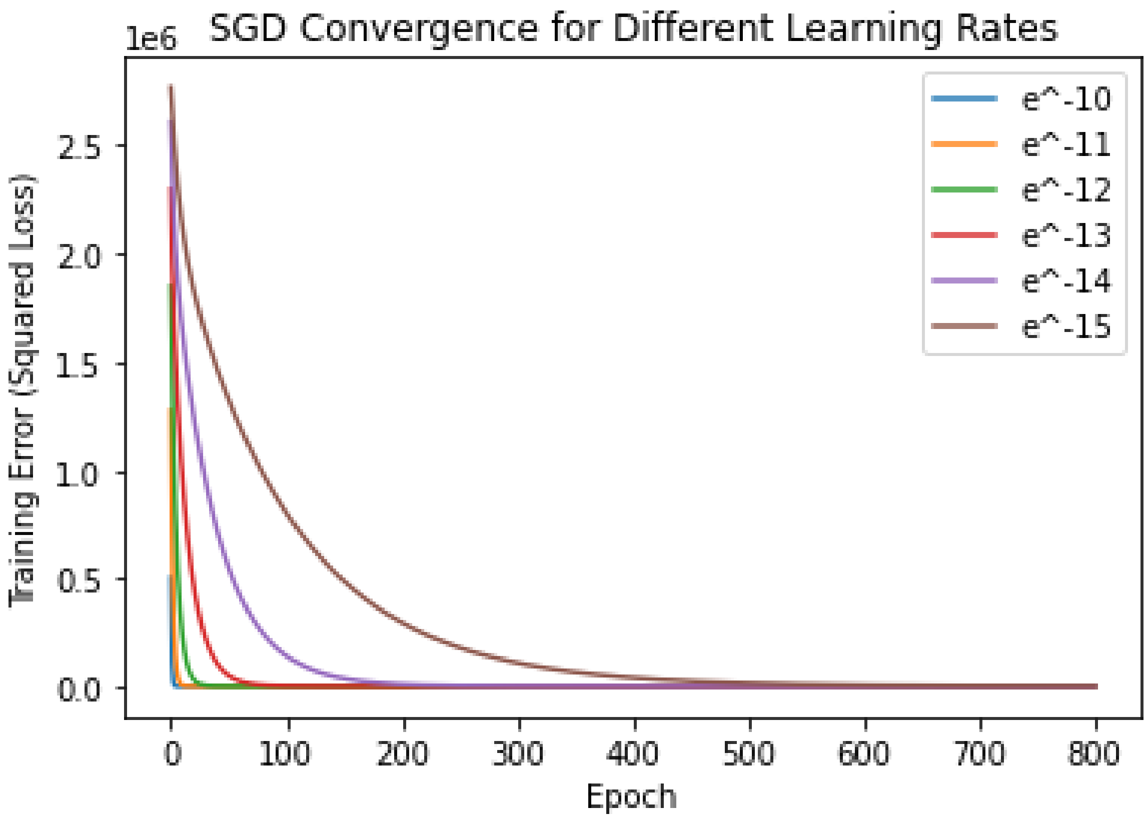
SGD Convergence for Different Step Sizes



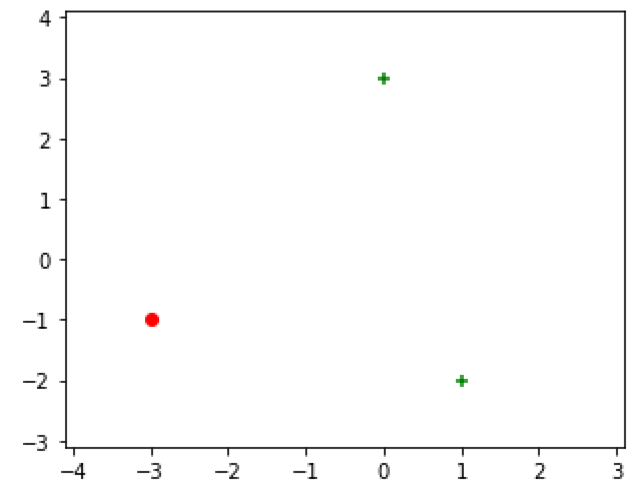
SGD with Multiple Optima (Extra Visualization)

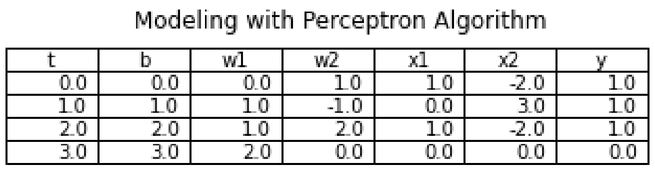


Part 3 G-I


SGD Convergence for Different Learning Rates
1e6
Training Error (Squared Loss)
0.0
0.5
1.0
1.5
2.0
2.5
0
100
200
300
400
500
600
700
800
Epoch
e^-10
e^-11
e^-12
e^-13
e^-14
e^-15

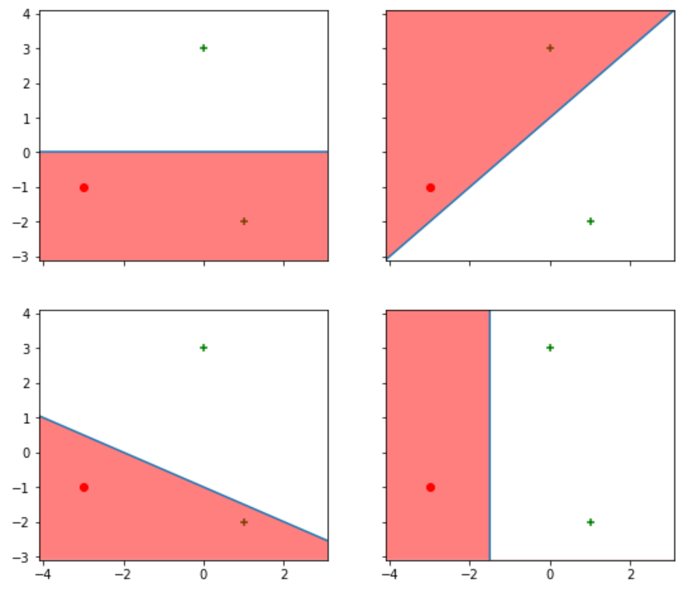
Part 4 A-C

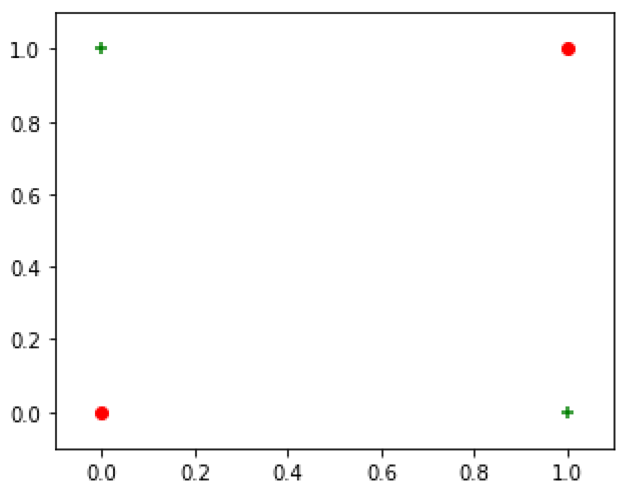


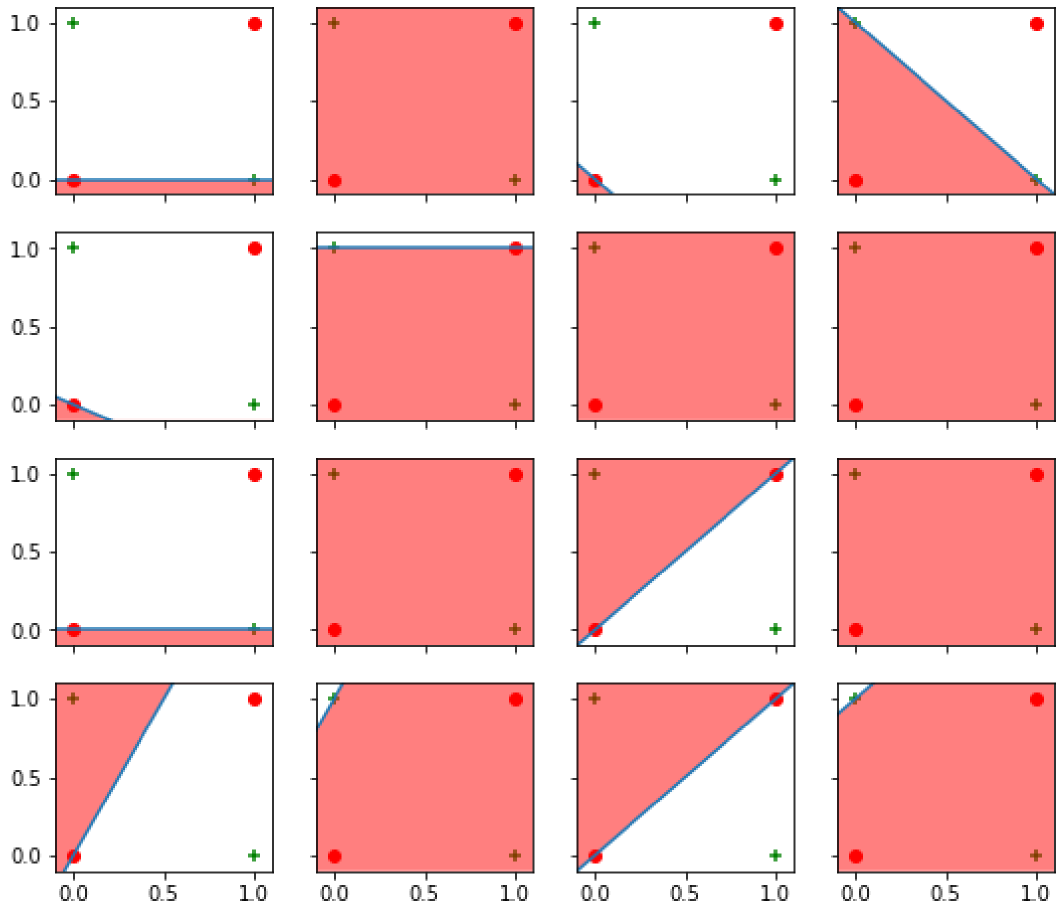


Modeling with Perceptron Algorithm

t	b	w1	w2	x1	x2	y
0.0	0.0	0.0	1.0	1.0	-2.0	1.0
1.0	1.0	1.0	-1.0	0.0	3.0	1.0
2.0	2.0	1.0	2.0	1.0	-2.0	1.0
3.0	3.0	2.0	0.0	0.0	0.0	0.0






1.0
0.5
0.0
0.0
0.5
1.0