Machine Learning & Data Mining
Set 1

Caltech CS/CNS/EE 155
January 7th, 2025

Policies

- Due 11:59 PM PST, January 15th 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.
- You are allowed to use up to a total of 48 late hours throughout the term. Late hours must be used in units of whole hours. Specify the total number of hours you have ever used when turning in the assignment.
- Students are expected to complete homework assignments based on their understanding of the course material. Student can use LLMs as a resource (e.g., helping with debugging, or grammar checking), but the assignments (including code) should be principally authored by the student.

Submission Instructions

- Submit your report as a single .pdf file to Gradescope (entry code 932378), under "Problem Set 1".
- 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. Upload all the files found in the 'code' folder from the set1.zip to your Google drive (put them all in one folder).
2. Edit the .ipynb file names 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 $\mathcal{H}$ is the set that contains all possible hypotheses we can pick from to model our target function. For example, if g is our final hypothesis for f, such that $g \approx f$, then $g \in \mathcal{H}$ necessarily.*

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

Solution B: *The hypothesis set for a linear model is the set of all lines, i.e. $m^T x + b$.*

Problem C [2 points]: What is overfitting?

Solution C: *When a model is trained on a sample, and its performance "in-sample" is dramatically better than its performance on sets outside of sample, it is said that the model has overfit. I.e. overfitting is when the model has low generalizability.*

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

Solution D: *Overfitting tends to happen when the model starts looking for things to learn that simply don't exist. A good way to avoid overfitting is by picking a hypothesis set of appropriate complexity given the dataset (this often requires domain-specific knowledge). Another good way is by repeatedly (and randomly) dividing your dataset into training-validation splits and picking a hypothesis with the lowest difference in training and validation sample errors.*

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 the subset of the dataset that is employed to train the learning model (i.e. picking a hypothesis from the hypothesis set). Testing data is the subset that is used to measure the performance of the hypothesis picked. Tailoring on the basis of the test-data performance defeats the purpose of the split because the model's performance is best measured on "unseen data," and in this case, we'd be "seeing" the data on its behalf.*

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

Solution F: *The sampling of data is assumed to be independent and obey a probability distribution. This means that the inclusion of (x_k, y_k) does not impact the inclusion of (x_j, y_j) and that the probability of both of their inclusions is derived from some properly defined distribution.*

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 could be vectors of length n, where n "features" are thought to be related to spam, such as the frequency of some words in the email, the no. of grammatical errors, the sender's address etc. and the output space could be $\{0, 1\}$, for not spam/spam.*

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

Solution H: *A k-fold cross-validation procedure essentially divides the data into k equal and disjoint subsets and for each set, it takes this set as the training set and trains the model on the $k - 1$ remaining subsets, then it calculates the validation error, and averages the k errors to yield an overall generalization error for the 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,

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

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: *Since* $\mathbb{E}_{out}(f_S) = \mathbb{E}_x\left[\left(f_S(x) - y(x)\right)^2\right]$*, we can say by changing the order that*

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

Now note that

$$\begin{aligned} \left(f_S(x) - y(x)\right)^2 &= \left(f_S(x) - F(x) + F(x) + y(x)\right)^2 \\ &= \left(f_S(x) - F(x)\right)^2 + \left(-y(x) + F(x)\right)^2 + 2\left(f_S(x) - F(x)\right)\left(-y(x) + F(x)\right) \end{aligned}$$

After removing the term whose expectation in S *is zero, we get*

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

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.

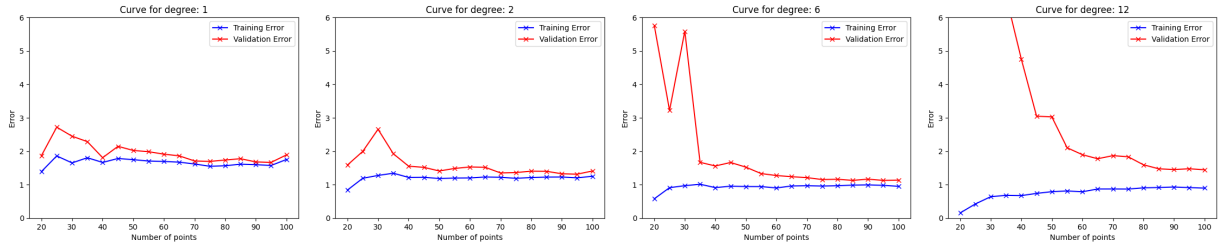
Polynomial regression is a type of regression that models the target y as a degree–d polynomial function of the input x. (The modeler chooses d.) You don't need to know how it works for this problem, just know that it produces a polynomial that attempts to fit the data.

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: See Code



(a) (b) (c) (d)

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: *Model using polynomial of degree 1 has approximately the highest error for both training and validation, hence it has the highest bias. Clear cut sign of underfitting.*

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

Solution D: *Model using polynomial of degree 12 has the highest variance because it has the highest validation error while some the lowest training errors. Clear cut sign of overfitting.*

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

Solution E: *It does not look like an increase in the number of data points will cause a difference in the quadratic model's performance. Beyond $N = 70$, both errors remain almost constant.*

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

Solution F: *Training error is lower because the model is literally trained on this dataset. It is biased towards this dataset because it is this dataset that has provided its basis as opposed to validation set which it has never seen.*

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 expect the model using polynomial of degree 6 to have the best performance on unseen data because it has the best training **and** validation errors (for large N), which means it performs well without overfitting.*

3 Stochastic Gradient Descent [36 Points]

Relevant materials: lecture 2

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

Linear regression learns a model of the form:

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

Problem A [2 points]: We can make our algebra and coding simpler by writing $f(x_1, x_2, \cdots, x_d) = \mathbf{w}^T \mathbf{x}$ for vectors $\mathbf{w}$ and $\mathbf{x}$. But at first glance, this formulation seems to be missing the bias term b from the equation above. How should we define $\mathbf{x}$ and $\mathbf{w}$ such that the model includes the bias term?

Hint: *Include an additional element in* $\mathbf{w}$ *and* $\mathbf{x}$*.*

Solution A: *This is done by including the trivial entry* $\mathbf{x}^{(0)} = 1$*, and setting* $\mathbf{w}^{(0)} = b$*, such that when they're multiplied, we get a "*$+b$ *term" overall.*

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

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

Problem B [2 points]: SGD uses the gradient of the loss function to make incremental adjustments to the weight vector $\mathbf{w}$. Derive the gradient of the squared loss function with respect to $\mathbf{w}$ for linear regression.

Solution B: *Given that*

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

we can say for the gradient that

$$\frac{dL}{d\mathbf{w}} = \sum_{i=1}^{N} \frac{d}{d\mathbf{w}} \left((y_i - \mathbf{w}^T \mathbf{x}_i)^2 \right) = \sum_{i=1}^{N} (-2\mathbf{x}_i)(y_i - \mathbf{w}^T \mathbf{x}_i)$$

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.

Problem C [8 points]: Implement the `loss`, `gradient`, and `SGD` functions, defined in the notebook, to perform SGD, using the guidelines below:

- Use a squared loss function.
- Terminate the SGD process after a specified number of epochs, where each epoch performs one SGD iteration for each point in the dataset.
- It is recommended, but not required, that you shuffle the order of the points before each epoch such that you go through the points in a random order. You can use `numpy.random.permutation`.
- Measure the loss after each epoch. Your `SGD` function should output a vector with the loss after each epoch, and a matrix of the weights after each epoch (one row per epoch). Note that the weights from all epochs are stored in order to run subsequent visualization code to illustrate SGD.

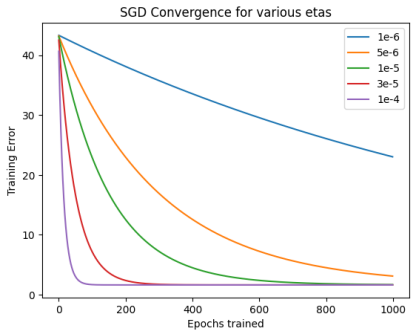
Solution C: *See Code*

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

Solution D: *The convergence behavior of the SGD does not change as the starting point varies in that it manages to converge in all cases, however, naturally, since points higher up have a larger downward gradient, the rate of convergence for these points is higher than points that are lower on the surface. The SGD method manages to reach the global minimum of the loss function in both of the datasets, as desired, although the loss-minimizing points in both datasets appear to be different.*

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

Solution E: *From the plot below, we can see that as η goes up, the training error reduces, and the rate of converge increases (the absolute derivative is greater).*



The following problems consider SGD with the larger, higher-dimensional dataset, `sgd_data.csv`. The file has a header denoting which columns correspond to which values. For these problems, use the Jupyter notebook `3_notebook_part2.ipynb`.

For your implementation of problems F-H, **do** consider the bias term using your answer to problem A.

Problem F [6 points]: Use your SGD code with the given dataset, and report your final weights. Follow the guidelines below for your implementation:

- Use $\eta = e^{-15}$ as the step size.
- Use $\mathbf{w} = [0.001, 0.001, 0.001, 0.001]$ as the initial weight vector and $b = 0.001$ as the initial bias.
- Use at least 800 epochs.
- You should incorporate the bias term in your implementation of SGD and do so in the vector style of problem A.
- Note that for these problems, it is no longer necessary for the `SGD` function to store the weights after all epochs; you may change your code to only return the final weights.

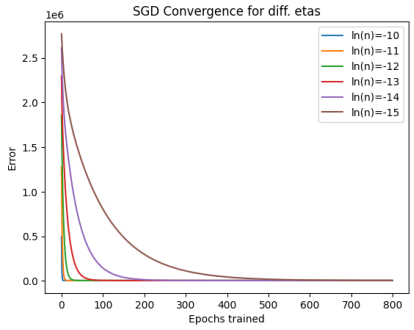
Solution F: *See Code, the weight vector* $\mathbf{w} =$
`[-0.22717856, -5.94207942, 3.94393238, -11.72380778, 8.78570763]`

Problem G [2 points]: Perform SGD as in the previous problem for each learning rate η in

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

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

Solution G: *From the plot below, we can see that as expected, when η goes up, the rate of convergence increases and the error drops more dramatically.*



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

$$\mathbf{w} = \left(\sum_{i=1}^{N} \mathbf{x_i} \mathbf{x_i}^T\right)^{-1} \left(\sum_{i=1}^{N} \mathbf{x_i} y_i\right).$$

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

Solution H: *Final weight vector* $\mathbf{w} =$
`[-0.31644251, -5.99157048, 4.01509955, -11.93325972, 8.99061096]`

Answer the remaining questions in 1-2 short sentences.

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

Solution I: *SGD serves as a computationally less expensive way to deducing the solution with reasonable accuracy, specially for large n, this method is preferred to raw computation of the closed form solution.*

Problem J [2 points]: Based on the SGD convergence plots that you generated earlier, describe a stopping condition that is more sophisticated than a pre-defined number of epochs.

Solution J: *Based on the plots, instead of an absolute stoploss, or a limit on the number of epochs, a good stopping point is when subsequent iterations do not yield a significant difference in error, i.e. if going from the 401th epoch to the 402nd epoch did not yield an error measurable in say, 10 digits of precision, then we stop.*

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

Solution K: *The SGD is smarter in its approach to the minimum. As long as the loss function casts a smooth surface with a well defined minimum, the SGD is guaranteed to converge whereas the perceptron could display unideal convergence behavior where it is not guaranteed to converge, as it picks its points not on the basis of improvement, but at random.*

4 The Perceptron [14 Points]

Relevant materials: lecture 2

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

$$f(\mathbf{x}) = \text{sign}\left(\left(\sum_{i=1}^{d} w_i x_i\right) + b\right)$$

The weights and bias of a perceptron can be thought of as defining a hyperplane that divides $\mathbb{R}^d$ such that each side represents an output class. For example, for a two dimensional dataset, a perceptron could be drawn as a line that separates all points of class $+1$ from all points of class -1.

The PLA (or the Perceptron Learning Algorithm) is a simple method of training a perceptron. First, an initial guess is made for the weight vector $\mathbf{w}$. Then, one misclassified point is chosen arbitrarily and the $\mathbf{w}$ vector is updated by

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

where $\mathbf{x}(t)$ and $y(t)$ correspond to the misclassified point selected at the t^{th} iteration. This process continues until all points are classified correctly.

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

Problem A [8 points]: 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			

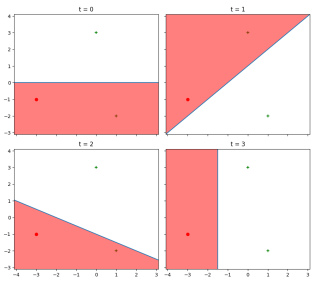
Include in your report both: the table that your code outputs, as well as the plots showing the perceptron's classifier at each step (see notebook for more detail).

Solution A: See Code

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

(a)



(b)

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

In a 2D dataset, how many data points are in the smallest dataset that is not linearly separable, such that no three points are collinear? How about for a 3D dataset such that no four points are coplanar? Please limit your solution to a few lines - you should justify but not prove your answer.

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

Solution B: *In 2D space, we can always separate all possible classifications of three non collinear points. However, with 4 points, this is not possible in some cases, like the XOR case. In general the perceptron can shatter a $d+1$ sized dataset in d space. Thus, for a 3D dataset, it is a guarantee that there is some configuration of 5 points that the perceptron cannot classify correctly. Since $d_{VC}(PLA) = d + 1$, for N dimensions, there exists a $N + 2$ sized dataset that the PLA is unable to correctly label.*

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: *No, if the dataset is not linearly separable, then without a max_iter threshold, the perceptron will keep going on. This is because every time it selects a new hypothesis correcting for a misclassified point, it is guaranteed to generate new misclassified points.*