CSCI567: Machine Learning	USC, Fall 2022
Homework 1	
Instructor: Vatsal Sharan	*Due: September 14 by 2:00 pm PST*

We would like to thank previous 567 staff, and Gregory Valiant (Stanford) for kindly sharing many of the problems with us.

A reminder on collaboration policy and academic integrity: Our goal is to maintain an optimal learning environment. You can discuss the homework problems at a high level with other groups, but you should not look at any other group's solutions. Trying to find solutions online or from any other sources for any homework or project is prohibited, will result in zero grade and will be reported. To prevent any future plagiarism, uploading any material from the course (your solutions, quizzes etc.) on the internet is prohibited, and any violations will also be reported. Please be considerate, and help us help everyone get the best out of this course.

Please remember the Student Conduct Code (Section 11.00 of the USC Student Guidebook). General principles of academic honesty include the concept of respect for the intellectual property of others, the expectation that individual work will be submitted unless otherwise allowed by an instructor, and the obligations both to protect one's own academic work from misuse by others as well as to avoid using another's work as one's own. All students are expected to understand and abide by these principles. Students will be referred to the Office of Student Judicial Affairs and Community Standards for further review, should there be any suspicion of academic dishonesty.

Instructions

We recommend that you use LaTeX to write up your homework solution. However, you can also scan handwritten notes. The homework will need to be submitted on D2L. We will announce detailed submission instructions later.

Theory-based Questions

Problem 1: Perceptron Convergence (15pts)

Recall the perceptron algorithm that we saw in class. The perceptron algorithm comes with strong theory, and you will explore some of this theory in this problem. We begin with some remarks related to notation which are valid throughout the homework. Unless stated otherwise, scalars are denoted by small letters in normal font, vectors are denoted by small letters in bold font, and matrices are denoted by capital letters in bold font.

Problem 1 asks you to show that when the two classes in a binary classification problem are linearly separable, then the perceptron algorithm will *converge*. For the sake of this problem, we define convergence as predicting the labels of all training instances perfectly. The perceptron algorithm is described in Algorithm 1. It gets access to a dataset of n instances $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$. It outputs a linear classifier $\mathbf{w}$.

Algorithm 1 Perceptron

while not converged **do**
  Pick a data point $(\mathbf{x}_i, y_i)$ randomly
  Make a prediction $\hat{y} = \text{sgn}(\mathbf{w}^T\mathbf{x}_i)$ using current $\mathbf{w}$
  if $\hat{y} \neq y_i$ **then**
    $\mathbf{w} \leftarrow \mathbf{w} + y_i\mathbf{x}_i$
  end if
end while

Assume there exists an optimal hyperplane $\mathbf{w}_{\text{opt}}$, $\|\mathbf{w}_{\text{opt}}\|_2 = 1$ and some $\gamma > 0$ such that $y_i(\mathbf{w}_{\text{opt}}^T\mathbf{x}_i) \geq \gamma, \forall i \in \{1, 2, \ldots, n\}$. Additionally, assume $\|\mathbf{x}_i\|_2 \leq R, \forall i \in \{1, 2, ..., n\}$. Following the steps below, show that the perceptron algorithm makes at most $\dfrac{R^2}{\gamma^2}$ errors, and therefore the algorithm must converge.

1.1 (5pts) Show that if the algorithm makes a mistake, the update rule moves it towards the direction of the optimal weights $\mathbf{w}_{\text{opt}}$. Specifically, denoting explicitly the updating iteration index by k, the current weight vector by $\mathbf{w}_k$, and the updated weight vector by $\mathbf{w}_{k+1}$, show that, if $y_i(\mathbf{w}_k^T\mathbf{x}_i) < 0$, we have

$$\mathbf{w}_{k+1}^T\mathbf{w}_{\text{opt}} \geq \mathbf{w}_k^T\mathbf{w}_{\text{opt}} + \gamma\|\mathbf{w}_{\text{opt}}\|. \tag{1}$$

Answer: According to the algorithm, if $y_i(\mathbf{w}_k^T\mathbf{x}_i) < 0$, we have $\mathbf{w}_{k+1} = \mathbf{w}_k + y_i\mathbf{x}_i$, replace $\mathbf{w}_{k+1}$ in the inequality, we have

$$\mathbf{w}_k^T\mathbf{w}_{\text{opt}} + \mathbf{x}_i^T y_i\mathbf{w}_{\text{opt}} \geq \mathbf{w}_k^T\mathbf{w}_{\text{opt}} + \gamma\|\mathbf{w}_{\text{opt}}\|$$

Then, eliminate same term and utilize $\mathbf{x}_i^T\mathbf{w}_{\text{opt}} = \mathbf{w}_{\text{opt}}^T\mathbf{x}_i$, we have

$$y_i\mathbf{w}_{\text{opt}}^T\mathbf{x}_i \geq \gamma\|\mathbf{w}_{\text{opt}}\|$$

Finally, utilize $\|\mathbf{w}_{\text{opt}}\|_2 = 1$, we have

$$y_i(\mathbf{w}_{\text{opt}}^T\mathbf{x}_i) \geq \gamma$$

This is given in the assumption. So we proved the inequality.

1.2 (4pts) Show that the length of updated weights does not increase by a large amount. In particular, show that if $y_i(\mathbf{w}_k^T\mathbf{x}_i) < 0$, then

$$\|\mathbf{w}_{k+1}\|^2 \leq \|\mathbf{w}_k\|^2 + R^2. \tag{2}$$

Answer: According to the algorithm, if $y_i(\mathbf{w}_k^T\mathbf{x}_i) < 0$, then $\mathbf{w}_{k+1} = \mathbf{w}_k + y_i\mathbf{x}_i$, we have

$$\begin{aligned}
\|\mathbf{w}_{k+1}\|^2 &= \|\mathbf{w}_k + y_i\mathbf{x}_i\|^2 \\
&\leq \|\mathbf{w}_k\|^2 + |y_i|\|\mathbf{x}_i\|^2 \\
&\leq \|\mathbf{w}_k\|^2 + |y_i|R^2 \\
&\leq \|\mathbf{w}_k\|^2 + R^2
\end{aligned}$$

1.3 (5pts) Assume that the initial weight vector $\mathbf{w}_0 = \mathbf{0}$ (an all-zero vector). Using results from the previous two parts, show that for any iteration $k+1$, with M being the total number of mistakes the algorithm has made for the first k iterations, we have

$$\gamma M \leq \|\mathbf{w}_{k+1}\| \leq R\sqrt{M}. \tag{3}$$

Hint: use the Cauchy-Schwartz inequality: $\mathbf{a}^T\mathbf{b} \leq \|\mathbf{a}\|\|\mathbf{b}\|$.

Answer: Let i be the i-th mistake the algorithm has made, then we have $\mathbf{w}_{k+1} = \mathbf{w}_0 + \sum_{i=1}^M y_i\mathbf{x}_i = \sum_{i=1}^M y_i\mathbf{x}_i$, so,

$$\begin{aligned}
\|\mathbf{w}_{k+1}\| &= \|\sum_{i=1}^M y_i\mathbf{x}_i\| \\
&= \|\sum_{i=1}^M y_i\mathbf{x}_i\|\|\mathbf{w}_{\text{opt}}\| \\
&\geq \sum_{i=1}^M y_i\mathbf{w}_{\text{opt}}^T\mathbf{x}_i \\
&\geq M\gamma
\end{aligned}$$

Then, for the right part, we know

$$\|\mathbf{w}_{k+1}\|^2 \begin{cases} = \|\mathbf{w}_k\|^2 \text{ , if } y_i(\mathbf{w}_k^T\mathbf{x}_i) \geq 0 \\ \leq \|\mathbf{w}_k\|^2 + R^2 \text{ , if } y_i(\mathbf{w}_k^T\mathbf{x}_i) < 0 \end{cases} .$$

Thus,

$$\|\mathbf{w}_{k+1}\|^2 \leq \|\mathbf{w}_0\|^2 + MR^2 = MR^2$$
$$\|\mathbf{w}_{k+1}\| \leq \sqrt{M}R$$

Now, we proved that $\gamma M \leq \|\mathbf{w}_{k+1}\| \leq R\sqrt{M}$

1.4 (1pts) Use **1.3** to conclude that $M \leq R^2/\gamma^2$. Therefore the algorithm can make at most R^2/γ^2 mistakes (note that there is no direct dependence on the dimensionality of the datapoints).

Answer: According to previous question, we know $0 < \gamma M \leq R\sqrt{M}$, so $\gamma^2 M^2 \leq R^2 M$, then we have

$$M \leq \frac{R^2}{\gamma^2}$$

So the algorithm can make at most R^2/γ^2 mistakes.

Problem 2: Logistic Regression (10pts)

This problems explores some properties of the logistic function to get you more comfortable with it. Consider the following univariate function first:

$$F(x; A, k, b) = \frac{A}{1 + e^{-k(x-b)}} \tag{4}$$

2.1 (3pts) Describe with words how A, k and b affect or are related to the shape of $F(x; A, k, b)$ by using the plot at `https://www.geogebra.org/graphing/mxw7wp8b`. Note: We are trying to learn the parameters of a logistic function that can fit the dataset. For example, see the figure below and guess which of the four functions fit well.

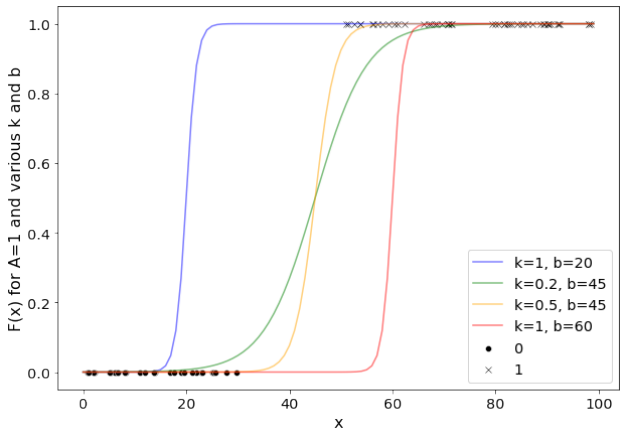


Figure 1: Learning the parameters of logistic function for a dataset.

Answer:
A: affect the magnitude of the function, the value will be with $(0, A)$
k: affect the slope of the function, a bigger k causes a steeper curve.
b: affect the position of symmetric point $x_0 = b$ where $F(x_0) = A/2$

2.2 (1pt) For what values of A, k and b is $F(x; A, k, b)$ a cumulative distribution function (CDF)?

Answer: To make sure $F(x; A, k, b)$ a CDF, $F(x; A, k, b)$ should be monotonically non-decreasing, right-continuous and within [0, 1].
Because F is a combination of basic functions, F is right-continuous.
Ensure the value is within [0, 1]: $A = 1$

Ensure monotonically non-decreasing:
$\forall x_1, x_2 \in (-\infty, \infty)$, when $x_1 \leq x_2$, $F(x_1) \leq F(x_2)$. Thus,

$$\begin{aligned} \frac{F(x_1)}{F(x_2)} &\leq 1 \\ \frac{1+e^{-k(x_2-b)}}{1+e^{-k(x_1-b)}} &\leq 1 \\ e^{-k(x_2-b)} &\leq e^{-k(x_1-b)} \\ -k(x_2-b) &\leq -k(x_1-b) \\ kx_2 &\geq kx_1 \\ k &\geq 0 \end{aligned}$$

Then, CDF should make sure that $\lim_{x\to+\infty} F(x) = 1$ and $\lim_{x\to-\infty} F(x) = 0$, we can get $k > 0$.
In conclusion, to make $F(x; A, k, b)$ a CDF, $A = 1, k > 0, \forall b \in \mathbb{R}$

2.3 (2pts) When $F(x; A, k, b)$ is a CDF, we can interpret $F(x; A, k, b)$ as the probability of x belonging to the class 1 (in a binary classification problem). Suppose we know $F(x; A, k, b)$ and want to predict the label of a datapoint x. We need to decide on a threshold value for $F(x; A, k, b)$, above which we will predict the label 1 and below which we will predict -1. Show that setting the threshold to be $F(x; A, k, b) \geq 1/2$ minimizes the classification error.

The value $x = x_0$ for which $F(x_0) = 0.5$ is called the decision boundary. In the univariate case, it is a point in on the x-axis, in the multivariate case (next question), it is a hyperplane. There is a rich literature on decision theory which studies how to make decisions if we know the probabilities of the underlying events (see `https://mlstory.org/pdf/prediction.pdf` to learn more if you're interested).

Answer:
Let $l_{i,j}$ be the loss for predicting a j label for a i label. If we predict $\hat{y} = 1$, to minimize the classification error, we have

$$l_{1,1}P(y=1) + l_{-1,1}P(y=-1) \leq l_{-1,-1}P(y=-1) + l_{1,-1}P(y=1)$$

Consider a 0-1 loss, where $l_{-1,-1} = l_{1,1} = 0$ and $l_{-1,1} = l_{1,-1} = 1$, we have $P(y=1) = F(x)$ and $P(y=-1) = 1 - F(x)$, thus,

$$1 - F(x) \leq F(x)$$

We can get that, to minimize error when predicting label 1, $F(x; A, k, b) \geq 1/2$. Similarly, we can get that $F(x; A, k, b) < 1/2$ when predicting label -1. In conclusion, setting the threshold to $1/2$ minimizing the error.

2.4 (4pts) We now consider the multivariate version. Consider the function:

$$F(\mathbf{x}; \mathbf{w}, b) = \frac{1}{1 + e^{-\mathbf{w}^\mathbf{T}\mathbf{x}+b}}. \tag{5}$$

You can explore a 2D version of this function at `https://www.geogebra.org/3d/g9fjtdeh` where $\mathbf{w} = (w_1, w_2)$. Match the items in the left column with those on the right. It will help if you can get the expression for the gradient of $F(\mathbf{x}; \mathbf{w}, b)$ w.r.t $\mathbf{x}$. Provide a short explanation of your answers.

1) level sets of $F(\mathbf{x}; \mathbf{w}, b)$	a) parallel to $\mathbf{w}$
2) direction of gradient at any point	b) $\|\|\mathbf{w}\|\|/4$
3) distance of the level set $F(\mathbf{x}) = 1/2$ from the origin	c) $\|b\|/\|\|\mathbf{w}\|\|$
	d) $\|b\|$
	e) orthogonal to $\mathbf{w}$

Answer:
1) The level set of $F(\mathbf{x}; \mathbf{w}, b)$ is $\{\mathbf{x} \mid F(\mathbf{x}; \mathbf{w}, b) = c\}$, that is, $\{\mathbf{x} \mid \mathbf{w}^\mathbf{T}\mathbf{x} = c\}$. It's orthogonal to $\mathbf{w}$ only when $c = 0$. So there is no match for 1).
2) The gradient of $F(\mathbf{x}; \mathbf{w}, b)$ is

$$\frac{\mathrm{d}F(\mathbf{x})}{\mathrm{d}\mathbf{x}} = \frac{\mathbf{w}(e^{-\mathbf{w}^\mathbf{T}\mathbf{x}+b})}{(1 + e^{-\mathbf{w}^\mathbf{T}\mathbf{x}+b})^2}$$

Thus, it's parallel to $\mathbf{w}$ and 2) is matched with a).
3) When $F(\mathbf{x}) = 1/2$, we can derive $\mathbf{w}^\mathbf{T}\mathbf{x} = b$. Meanwhile, the distance from the origin is $\|\mathbf{x}\|_2$. It's equal to $|b|/||\mathbf{w}||$ only when $\mathbf{x}$ is parallel to $\mathbf{w}$. So there is no match for 3).

Problem 3: Learning rectangles (15pts)

An axis aligned rectangle classifier is a classifier than assigns the value 1 to a point if and only if it is inside a certain rectangle. Formally, given real numbers $a_1 \leq b_1, a_2 \leq b_2$, define the classifier $f_{(a_1,b_1,a_2,b_2)}$ on an input $\mathbf{x}$ with coordinates (x_1, x_2) by

$$f_{(a_1,b_1,a_2,b_2)}(x_1, x_2) = \begin{cases} 1 & \text{if } a_1 \leq x_1 \leq b_1 \text{ and } a_2 \leq x_2 \leq b_2 \\ -1 & \text{otherwise.} \end{cases}$$

The function class of all axis-aligned rectangles in the plane is defined as

$$\mathcal{F}^2_{\text{rec}} = \{f_{(a_1,b_1,a_2,b_2)}(x_1, x_2) : a_1 \leq b_1, a_2 \leq b_2\}.$$

We will assume that the true labels y of the datapoints $(\mathbf{x}, y)$ are given by some axis-aligned rectangle (this is the realizability assumption discussed in class). The goal of this question is to come up with an algorithm which gets small classification error with respect to any distribution D over $(\mathbf{x}, y)$ with good probability.

The loss function we use throughout the question is the 0-1 loss. It will be convenient to denote a rectangle marked by corners (a_1, b_1, a_2, b_2) as $B(a_1, b_1, a_2, b_2)$. Let $B^* = B(a_1^*, b_1^*, a_2^*, b_2^*)$ be the rectangle corresponding to the function $f_{(a_1^*,b_1^*,a_2^*,b_2^*)}$ which labels the datapoints. Let $S = \{(\mathbf{x}_i, y_i), i \in [n]\}$ be a training set of n samples drawn i.i.d. from D. Please see Fig. 2 for an example.

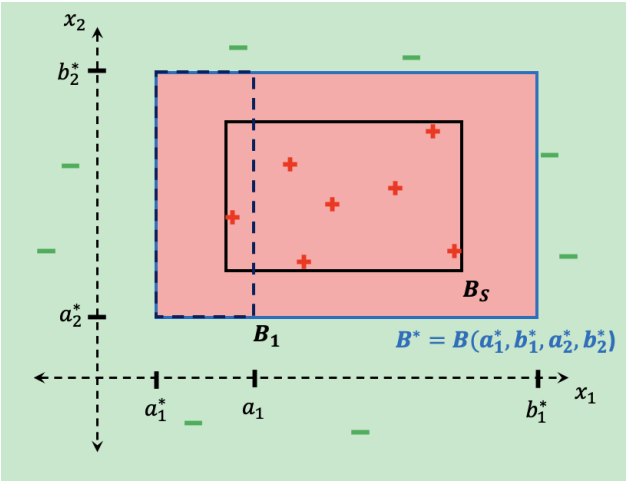


Figure 2: Learning axis-aligned rectangles in two dimensions, here $+$ (red) denotes datapoints in training set S with label 1 and $-$ (green) denotes datapoints with label -1. The true labels are given by rectangle B^* (solid blue line), with everything outside B^* being labelled negative and inside being labelled positive. B_S (solid black line) is the rectangle learned by the algorithm in Part (a). B_1 (dashed black line) is defined in Part (c).

3.1 (3pts) We will follow the general supervised learning framework from class. Given the 0-1 loss, and the function class of axis-aligned rectangles, we want to find an empirical risk minimizer. Consider the algorithm which returns the smallest rectangle enclosing all positive examples in the training set. Prove that this algorithm is an empirical risk minimizer.

Answer:
Observe that by the algorithm's definition, the returned rectangle labels all the positive instances in the training set positively. Then, because of the given realizability assumption, and since the smallest rectangle enclosing all positive

examples is returned, all the negative instances are also labeled correctly. So this algorithm is an ERM.

3.2 (2pts) Our next task is to show that the algorithm from the previous part not only does well on the training data, but also gets small classification error with respect to the distribution D. Let B_S be the rectangle returned by the algorithm in the previous part on training set S, and let f_S^{ERM} be the corresponding hypothesis. First, we will convince ourselves that generalization is inherently a probabilistic statement. Let a *bad* training set S' be a training set such that $R(f_{S'}^{ERM}) \geq 0.5$. Pick some simple distribution D and ground-truth rectangle B^*, and give a short explanation for why there is a non-zero probability of seeing a bad training set.

Answer:
Consider an axis-aligned rectangle $B^* = B(a_1^*, b_1^*, a_2^*, b_2^*)$ and a distribution D, where the probability mass inside B^* is $\frac{3}{4}$ and outside is $\frac{1}{4}$. Both probability mass are evenly distributed inside their own region. Now, we always draw the positive training samples from a $B_S = B(a_1, b_1, a_2, b_2)$, which is inside B^* ($a_1^* \leq a_1 \leq b_1 \leq b_1^*$ and $a_2^* \leq a_2 \leq b_2 \leq b_2^*$) and the area size is no more than $\frac{1}{3}$ of B^*. Then, if we apply the previous algorithm, B_S will be the returned rectangle and $R(f_{S'}^{ERM}) = \frac{3}{4}(1 - \text{Area of } B_S / \text{Area of } B^*) \geq \frac{3}{4}(1 - \frac{1}{3}) = 0.5$. Thus, it's possible to see a bad training set.

3.3 (5pts) We will now show that with good probability over the training dataset S, f_S^{ERM} does get small error. Show that if $n \geq \frac{4\log(4/\delta)}{\epsilon}$ then with probability at least $1 - \delta$, $R(f_S^{ERM}) \leq \epsilon$.

To prove this follow the following steps. Let $a_1 \geq a_1^*$ be such that the probability mass (with respect to D) of the rectangle $B_1 = B(a_1^*, a_1, a_2^*, b_2^*)$ is exactly $\epsilon/4$. Similarly, let b_1, a_2, b_2 be numbers such that the probability mass (with respect to D) of the rectangles $B_2 = B(b_1, b_1^*, a_2^*, b_2^*), B_3 = B(a_1^*, b_1^*, a_2^*, a_2), B_4 = B(a_1^*, b_1^*, b_2, b_2^*)$ are all exactly $\epsilon/4$.

- Show that $B_S \subseteq B^*$.
- Show that if S contains (positive) examples in all of the rectangles B_1, B_2, B_3, B_4 then $R(f_S^{ERM}) \leq \epsilon$.
- For each $i \in \{1, \ldots, 4\}$ upper bound the probability that S does not contain an example from B_i.
- Use the union bound to conclude the argument.

Answer:
According to the definition of the algorithm, it's obvious that $B_S \subseteq B^*$. Then, we can conclude that

$$R(f_S^{ERM}) = \mathcal{D}(B^* \setminus B_S) \leq \mathcal{D}(\bigcup_{i=1}^{4} B_i) \leq \sum_{i=1}^{4} \mathcal{D}(B_i) = \epsilon$$

Thus, if $R(f_S^{ERM}) > \epsilon$, samples in the training set should not appear in at least one B_i. That is,

$$\begin{aligned} \underset{S\sim D^n}{P}(R(f_S^{ERM}) > \epsilon) &\leq \underset{S\sim D^n}{P}(\bigcup_{i=1}^{4}\{B_S \cap B_i = \varnothing\}) \\ &\leq \sum_{i=1}^{4} \underset{S\sim D^n}{P}(\{B_S \cap B_i = \varnothing\}) \\ &= \sum_{i=1}^{4}(1 - \frac{\epsilon}{4})^n \\ &\leq 4e^{-n\frac{\epsilon}{4}} \end{aligned}$$

Let $4e^{-n\frac{\epsilon}{4}} \leq \delta$, we can get that when $n \geq \frac{4\log(4/\delta)}{\epsilon}$, $\underset{S\sim D^n}{P}(R(f_S^{ERM}) > \epsilon) \leq \delta$, that is, with the probability of at least $1 - \delta$, $R(f_S^{ERM}) \leq \epsilon$.

3.4 (5pts) Repeat the previous question for the function class of axis-aligned rectangles in $\mathbb{R}^d$. To show this, try to generalize all the steps in the above question to higher dimensions, and find the number of training samples n required to guarantee that $R(f_S^{ERM}) \leq \epsilon$ with probability at least $1 - \delta$.

Answer:
The axis-aligned rectangle in $\mathbb{R}^d$ could be defined as follows:

$$\mathcal{F}^d_{\text{rec}} = \{f_{(a_1,b_1,...,a_d,b_d)}(x_1, x_2, ..., x_d) : \forall\ 1 \leq i \leq d,\ a_i \leq b_i\}$$

The classifier could be:

$$f_{(a_1,b_1,...,a_d,b_d)}(x_1, x_2, ..., x_d) = \begin{cases} 1 & \text{if } \forall\ 1 \leq i \leq d,\ a_i \leq x_i \leq b_i \\ -1 & \text{otherwise.} \end{cases}$$

It's obvious that the above algorithm is still ERM and we can follow the steps in 3.3 to get n as well. Instead of 4 B_i, we now have 2d B_i, each with a probability mass of $\epsilon/2d$. Then, we can get when $n \geq \frac{2d\log(2d/\delta)}{\epsilon}$, $R(f_S^{ERM}) \leq \epsilon$ with probability at least $1 - \delta$

Programming-based Questions

Before you start to conduct homework in this part, you need to first set up the coding environment. We use python3 (version ≥ 3.7) in our programming-based questions. There are multiple ways you can install python3, for example:

- You can use conda to configure a python3 environment for all programming assignments.
- Alternatively, you can also use virtualenv to configure a python3 environment for all programming assignments

After you have a python3 environment, you will need to install the following python packages:

- numpy
- matplotlib (for you plotting figures)

Note: You are **not allowed** to use other packages, such as *tensorflow, pytorch, keras, scikit-learn, scipy*, etc. to help you implement the algorithms you learned. If you have other package requests, please ask first before using them.

Problem 4: k-nearest neighbor classification and the ML pipeline (25pts)

In class, we talked about how we need to do training/test split to make sure that our model is generalizing well. We also discussed how we should not reuse a test set too much, because otherwise the test accuracy may not be an accurate measure of the accuracy on the data distribution. In reality, a ML model often has many *hyper-parameters* which need to be tuned (we will see an example in this question). We don't want to use the test set over and over to see what the best value of this hyper-parameter is. The solution is to have a third split of the data, and create a *validation set*. The idea is to use the validation set to evaluate results from the training set and tune any hyper-parameters. Then, use the test set to double-check your evaluation after the model has "passed" the validation set. Please see this nice explanation for more discussion: `https://developers.google.com/machine-learning/crash-course/validation/another-partition`.

With this final piece, we are now ready to build a real ML pipeline. It usually conducts three parts. (1) Load and pre-process the data. (2) Train a model on the training set and use the validation set to tune hyper-parameters. (3) Evaluate the final model on the test set and report the result.

In this problem, you will implement the *k-nearest neighbor (k-NN) algorithm* to conduct classification tasks. We provide the bootstrap code and you are expected to complete the classes and functions. You can find the code and data on `https://vatsalsharan.github.io/fall22/knn_hw1.zip`.

k-NN algorithm

The k-nearest neighbor (k-NN) algorithm is one of the simplest machine learning algorithms in the supervised learning paradigm. The idea behind k-NN is simple, and we explain it first for the case of $k = 1$. The 1-NN algorithm predicts the label of any new datapoint $\mathbf{x}$ by finding its closest neighbor $\mathbf{x}'$ in the training set, and then predicts the label of $\mathbf{x}$ to be the same as the label of $\mathbf{x}'$. For general k, the k-NN algorithm predicts the label by taking a majority vote on the k nearest neighbors.

We now describe the algorithm more rigorously. Given a hyper-parameter k, training instances $(\mathbf{x}_i, y_i)$ ($\mathbf{x}_i \in \mathbb{R}^d$ and y_i is the label), and a test example $\mathbf{x}$, the k-NN algorithm can be executed based on the following steps,

1. Calculate the *distances* between the test example and each of the training examples.
2. Take the k nearest neighbors based on the distances calculated in the previous step.
3. Among these k nearest neighbors, count the number of the data points in each class.
4. Predict the label $\hat{y}$ of $\mathbf{x}$ to be the most frequent class among these neighbors (we describe how to break any ties later).

You are asked to implement the missing functions in `knn.py` following each of the steps.

Part 4.1 Report 4-nearest neighbor accuracy (8pts)

Euclidean distance calculation Compute the distance between the data points based on the following equation:

$$d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_2 = \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2}. \tag{6}$$

Task: Fill in the code for the function `compute_l2_distances`.

k-NN classifier Implement a k-NN classifier based on the above steps. Your algorithm should output the predictions for the test set. *Note:* You do not need to worry about ties in the distance when finding the k nearest neighbor set. However, when there are ties in the majority label of the k nearest neighbor set, you should return the label with the smallest index. For example, when $k = 4$, if the labels of the 4 nearest neighbors happen to be 0, 0, 1, 1, your prediction should be the label 0.

Task: Fill in the code for the function `predict_labels`.

Report the error rate We want you to report the error rate for the classification task. The error rate is defined as:

$$\text{error rate} = \frac{\text{\# of wrongly classified examples}}{\text{\# of total examples}}. \tag{7}$$

Task: Fill in the code for the function `compute_error_rate`.

Report Item: Report the error rate of your k nearest neighbor algorithm in the validation set when $k = 4$ using Euclidean distance.

Answer: The error rate in the validation set when $k = 4$ using Euclidean distance is 0.3466666666666667.

Part 4.2 Data transformation (2+2pts)

We are going to add one more step (data transformation) in the `data_processing` part and see how it works. Data transformations and other feature engineering steps often play a crucial role to make a machine learning model work. Here, we take two different data transformation approaches. This link might be helpful: `https://en.wikipedia.org/wiki/Feature_scaling`.

Normalizing the feature vector This one is simple but sometimes may work well. Given a feature vector $\mathbf{x}$, the normalized feature vector is given by $\tilde{\mathbf{x}} = \frac{\mathbf{x}}{\|\mathbf{x}\|_2}$. If a vector is an all-zero vector, define the normalized vector to also be the all-zero vector (in practice, a useful trick is to add a very very small value to the norm of $\mathbf{x}$, so that we do not need to worry about the division by zero error).

Min-max scaling for each feature The above normalization is independent of the rest of the data. On the other hand, min-max normalization scales each sample in a way that depends on the rest of the data. More specifically, for each feature in the training set, we normalize it linearly so that its value is between 0 and 1 across all samples, and in addition, the largest value becomes exactly 1 while the smallest becomes exactly 0. Then, we will apply the same scaling parameters we get from training data to our testing instances.

Task: Fill in the code for the function `data_processing_with_transformation`.

Report Item: Report the error rate of your k nearest neighbor algorithm in the validation set for $k = 4$ using Euclidean distance when data is using (1) Normalized featured vector, and (2) Min-max scaling featured vector.

Answer: The error rate in the validation set for $k = 4$ using Euclidean distance when data is using (1) Normalized featured vector is 0.3333333333333333, and (2) Min-max scaling featured vector is 0.30666666666666664.

Part 4.3 Different distance measurement (3pts)

In this part, we will change the way that we measure distances. We will work with the original (unnormalized) data for simplicity, and continue the experiments from Part 4.1.

Cosine distance calculation Compute the distance between data points based on the following equation:

$$d(\mathbf{x}, \mathbf{x}') = \begin{cases} 1, & \text{if } \|\mathbf{x}\|_2 = 0 \text{ or } \|\mathbf{x}'\|_2 = 0 \\ 1 - \dfrac{\mathbf{x} \cdot \mathbf{x}'}{\|\mathbf{x}\|_2 \|\mathbf{x}'\|_2} & \text{otherwise.} \end{cases}$$

Similar to when we are normalizing the feature vector, a useful trick in practice is to add a very very small value to the norm of $\mathbf{x}$, so that we do not need to worry about the division by zero issues.

Task: Fill in the code for the function `compute_cosine_distances` and change distance function used in the main function in the code to get results.

Report Item: Report the error rate of your k nearest neighbor algorithm in the validation set for $k = 4$ using cosine distance for *original data*.

Answer: The error rate in the validation set for $k = 4$ using cosine distance is 0.3333333333333333.

Part 4.4 Tuning the hyper-parameter k (10pts)

Again, follow Part 4.1, however, this time we conduct experiments with $k = \{1, 2, 4, 6, 8, 10, 12, 14, 16, 18\}$.

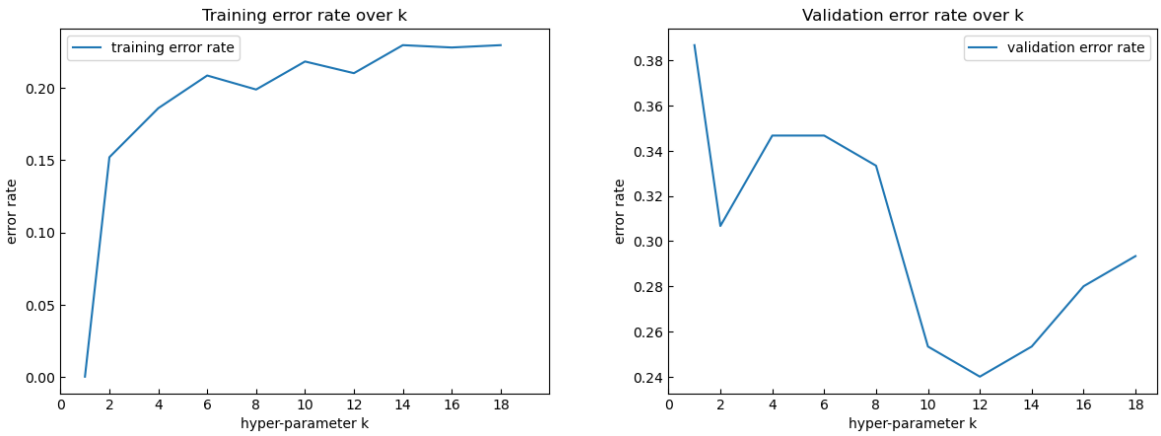
Task: Fill in the code for the function `find_best_k`.

Report Item: (1) Report and draw a curve based on the error rate of your model on the *training set* for each k. What do you observe? (2pts) (2) Report and draw a curve based on the error rate of your model on the *validation set* for each k. What is your best k? (2pts) (3) What do you observe by comparing the difference between the two curves? (2pts) (4) What's the final test set error rate you get using your best-k? (1pt) (5) Comment on these results (3pts).

Answer:
(1) Training error rate for each k: $\{0.000, 0.152, 0.186, 0.209, 0.199, 0.218, 0.210, 0.230, 0.228, 0.230\}$. The curve is shown in Fig. 3(a). Observe that the overall trend of training error rate is increasing when k gets bigger and when $k = 1$, the error rate is zero.
(2) Validation error rate for each k: $\{0.387, 0.307, 0.347, 0.347, 0.333, 0.253, 0.240, 0.253, 0.280, 0.293\}$. The curve is shown in Fig. 3(b). The best k is 12.
(3) The overall training error rate is going up when k gets bigger. The overall validation error rate will go down first and reach a minima, then it will go up.
(4) The final test set error is 0.21333333333333335.
(5) The training error rate is zero when $k = 1$. That's because the one nearest neighbor is itself. It's an overfitting model for the training set. When k gets bigger, because it will predict one point's label with many other neighbor belonging to different groups. So the training error rate will increase. For the validation error, because initially the model is overfitting, the error will decrease when k gets bigger and reach a minima, then, it will increase. That's because the model becomes underfitting when you take too more points into account. Thus, we could use validation set to find best-k and we see the final test error is smaller than that with a not tuned k.

Part 4.5 (0pts)

Report Item: Include your filled in code in the submitted pdf file.
See the code in `knn.py`.



(a) Training error rate over k

(b) Validation error rate over k

Figure 3: Error rate over k

Problem 5: Linear Regression (20pts)

We will consider the problem of fitting a linear model. Given d-dimensional input data $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$ with real-valued labels $y_1, \cdots, y_n \in \mathbb{R}$, the goal is to find the coefficient vector $\mathbf{w}$ that minimizes the sum of the squared errors. The total squared error of $\mathbf{w}$ can be written as $F(\mathbf{w}) = \sum_{i=1}^{n} f_i(\mathbf{w})$, where $f_i(\mathbf{w}) = (\mathbf{w}^\mathbf{T}\mathbf{x_i} - y_i)^2$ denotes the squared error of the ith data point. We will refer to $F(\mathbf{w})$ as the *objective function* for the problem.

The data in this problem will be drawn from the following linear model. For the training data, we select n data points $\mathbf{x}_1, \cdots, \mathbf{x}_n$, each drawn independently from a d-dimensional Gaussian distribution. We then pick the "true" coefficient vector $\mathbf{w}^*$ (again from a d-dimensional Gaussian), and give each training point $\mathbf{x}_i$ a label equal to $(\mathbf{w}^*)^\top \mathbf{x}_i$ plus some noise (which is drawn from a 1-dimensional Gaussian distribution).

The following Python code will generate the data used in this problem.

```
d = 100 # dimensions of data
n = 1000 # number of data points
X = np.random.normal(0,1, size=(n,d))
w_true = np.random.normal(0,1, size=(d,1))
y = X.dot(w_true) + np.random.normal(0,0.5,size=(n,1))
```

5.1 (6pts) Least-squares regression has the closed form solution $\mathbf{w}_{LS} = (\mathbf{X}^\mathbf{T}\mathbf{X})^{-1}\mathbf{X}^\mathbf{T}\mathbf{y}$, which minimizes the squared error on the data. (Here $\mathbf{X}$ is the $n \times d$ data matrix as in the code above, with one row per data point, and $\mathbf{y}$ is the n-vector of their labels.) Solve for $\mathbf{w}_{LS}$ on the training data and report the value of the objective function $F(\mathbf{w}_{LS})$. For comparison, what is the total squared error $F(\mathbf{w})$ if you just set $\mathbf{w}$ to be the all 0's vector? Using similar Python code, draw $n = 1000$ test data points from the same distribution, and report the total squared error of $\mathbf{w}_{LS}$ on these test points. What is the gap in the training and test objective function values? Comment on the result.

Note: Computing the closed-form solution requires time $O(nd^2 + d^3)$, which is slow for large d. Although gradient descent methods will not yield an exact solution, they do give a close approximation in much less time. For the purpose of this assignment, you can use the closed form solution as a good sanity check in the following parts.

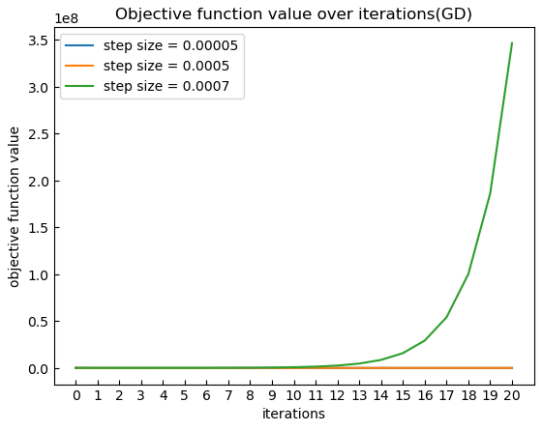
Answer:
if we set $\mathbf{w}$ to be the all 0's vector, the total squared error will be $\|\mathbf{y}\|^2$.
The value of objective function w.r.t $\mathbf{w}_{LS}$ is 207.52189068973843. For all-zero $\mathbf{w}$, the total error is 77161.83162003125.
The total squared error of $\mathbf{w}_{LS}$ on these test points is 258.19763849944593 and the gap is 50.6757478097075.
The total error of training set for $\mathbf{w}_{LS}$ is way smaller than all-zero $\mathbf{w}$. That's because setting $\mathbf{w}$ to all zero will cause underfitting. The total squared error of training set is closed to the error of test set. That's because we generate the test

data from the same distribution. The generalization gap is small but couldn't be zero because of the random used in generating data.

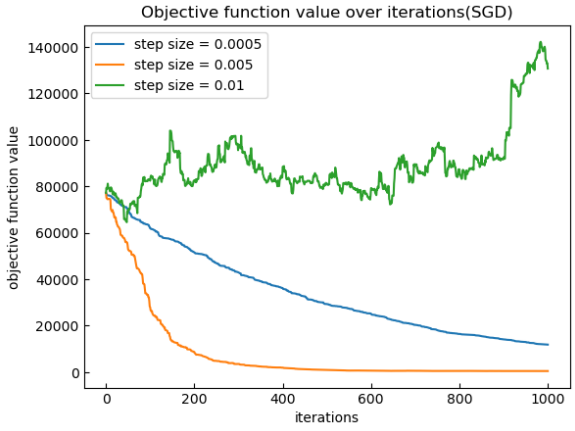
5.2 (7pts) In this part, you will solve the same problem via *gradient descent* on the squared-error objective function $F(\mathbf{w}) = \sum_{i=1}^{n} f_i(\mathbf{w})$. Recall that the gradient of a sum of functions is the sum of their gradients. Given a point $\mathbf{w}_t$, what is the gradient of f at $\mathbf{w}_t$?

Now use gradient descent to find a coefficient vector $\mathbf{w}$ that approximately minimizes the least squares objective function over the training data. Run gradient descent three times, once with each of the step sizes 0.00005, 0.0005, and 0.0007. You should initialize $\mathbf{w}$ to be the all-zero vector for all three runs. Plot the objective function value for 20 iterations for all 3 step sizes on the same graph. Comment in 3-4 sentences on how the step size can affect the convergence of gradient descent (feel free to experiment with other step sizes). Also report the step size that had the best final objective function value and the corresponding objective function value.

Answer:
The plot is shown in Fig. 4(a). The step size of 0.0005 has the best final objective function value of 207.52236695566353. When the step size is small, GD will converge slowly and need more iterations. As the step size gets bigger, GD will converge more quickly and use less iterations. However, when the step size continues to increase, GD will not converge because of "jumping over", a.k.a overstepping, a minima. It would cause oscillation around minima or divergence.



(a) Objective function value over 20 iterations(GD)



(b) Objective function value over 1000 iterations(SGD)

Figure 4: Objective function value over iterations (gradient descent and stochastic gradient descent)

5.3 (7pts) In this part you will run *stochastic gradient descent* to solve the same problem. Recall that in stochastic gradient descent, you pick one training datapoint at a time, say $(\mathbf{x}_i, y_i)$, and update your current value of $\mathbf{w}$ according to the gradient of $f_i(\mathbf{w}) = (\mathbf{w}^{\mathbf{T}}\mathbf{x_i} - y_i)^2$.

Run stochastic gradient descent using step sizes $\{0.0005, 0.005, 0.01\}$ and 1000 iterations. Plot the objective function value vs. the iteration number for all 3 step sizes on the same graph. Comment 3-4 sentences on how the step size can affect the convergence of stochastic gradient descent and how it compares to gradient descent. Compare the performance of the two methods. How do the best final objective function values compare? How many times does each algorithm use each data point? Also report the step size that had the best final objective function value and the corresponding objective function value.

Answer:
The plot is shown in Fig. 4(b). The step size of 0.005 has the best final objective function value of 449.47831140359267. When the step size is small, SGD will converge slowly and need more iterations. As the step size gets bigger, SGD will converge more quickly and use less iterations. However, when the step size is too big, SGD will not converge and would oscillate around minima or diverge. The effect of step size to SGD is very similar to GD. The best final objective function value of GD is 207.52, which is better than the 449.48 of SGD. In GD, each data point is used once for each iteration. Thus, 20 times in total. In SGD, each iteration uses a random data point. For 1000 data points,

the expectation for times of each data point used in 1000 iterations is 1. According to the Fig. 4, SGD need more iterations to converge compared to GD. GD performs better for a small number of iterations.

5.4 (0pts) Include the code for all the previous parts in the submitted pdf file.
See the code in `linear_regression.py`.