CSCI567: Machine Learning	USC, Spring 2024
Homework 1	
Instructor: Vatsal Sharan	*Due: February 7 by 11:59 pm PST*

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 Gradescope. **While submitting on Gradescope, please make sure to select your project partner and correctly mark the pages for the answer to each question.**

Theory-based Questions

Problem 1: Perceptron Convergence (20pts)

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 (defined later), then the perceptron algorithm will provably *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 $(\mathbf{x}_i, y_i)$ uniformly 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 $y_i \cdot (\mathbf{w}_{\text{opt}}^T \mathbf{x}_i) \geq \gamma, \forall i \in \{1, 2, \ldots, n\}$. Additionally, assume $\|\mathbf{x}_i\|_2 \leq R$, $\forall i \in \{1, 2, ..., n\}$ for some $R > 0$. 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 (7pts) Show that if the algorithm makes a mistake, the update rule moves the parameter $\mathbf{w}$ 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 $y_i(\mathbf{w}_k^T\mathbf{x}_i) < 0$ where $(\mathbf{x}_i, y_i)$ is the instance selected in this iteration, we have

$$\mathbf{w}_{k+1}^T\mathbf{w}_{\text{opt}} \geq \mathbf{w}_k^T\mathbf{w}_{\text{opt}} + \gamma\|\mathbf{w}_{\text{opt}}\|_2. \tag{1}$$

1.2 (5pts) 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

$$\|\mathbf{w}_{k+1}\|_2^2 \leq \|\mathbf{w}_k\|_2^2 + R^2. \tag{2}$$

1.3 (6pts) 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

$$\gamma M \leq \|\mathbf{w}_{k+1}\|_2 \leq R\sqrt{M}. \tag{3}$$

Hint: use the Cauchy-Schwartz inequality: $\mathbf{a}^T\mathbf{b} \leq \|\mathbf{a}\|_2\|\mathbf{b}\|_2$.

1.4 (2pts) 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 data points).

Problem 2: Learning rectangles (16pts + 8 pts Bonus)

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

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

We will assume that the true labels y of the data points $(\mathbf{x}, y)$ 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 (meaning that for all $(\mathbf{x}, y)$ drawn from distribution D, $f_{(a_1^*,b_1^*,a_2^*,b_2^*)}(\mathbf{x}) = y$). 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. 1 for an example.

2.1 (4pts) We will follow the general supervised learning framework from class. For a function $f_{(a_1,b_1,a_2,b_2)}$, define the empirical risk with respect to 0-1 loss as $\widehat{\mathcal{R}}(f_{(a_1,b_1,a_2,b_2)}) = \sum_{i=1}^n \mathbb{I}\{f_{(a_1,b_1,a_2,b_2)}(\mathbf{x}_i) \neq y_i\}$. 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, meaning that it minimize $\widehat{\mathcal{R}}(f)$ over all f in $\mathcal{F}_{\text{rec}}^2$. (*Hint: use the realizability assumption.*)

2.2 (4pts) 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 **2.1** 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.

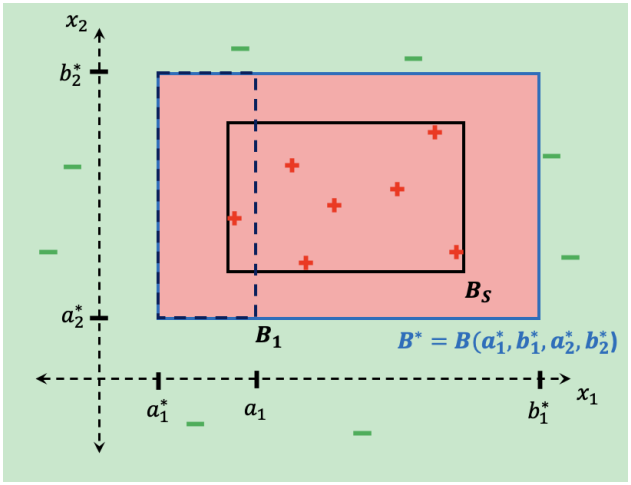

Figure 1: Learning axis-aligned rectangles in two dimensions, here $+$ (red) denotes datapoints in training set S with label 1 and $-$ (green) denotes datapoints with label -1. The true labels are given by rectangle B^* (solid blue line), with everything outside B^* being labelled negative and inside 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).

2.3 (8pts) Though there is non-zero probability seeing bad training set, we will now show that *with high 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 $b_1 \leq b_1^*, a_2 \geq a_2^*, b_2 \leq b_2^*$ 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 $R(f_S^{ERM}) \leq \epsilon$. (Hint: think about the geometric relationship between B_i and B_S, $i \in \{1,2,3,4\}$)
- 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.

(Bonus) 2.4 (8pts) Repeat the previous question for the function class of axis-aligned rectangles in $\mathbb{R}^d$. Specifically, the current function class is $\mathcal{F}_{\text{d-rec}} = \{f_{(a_1,b_1,a_2,b_2,\ldots,a_d,b_d)}(x_1, x_2, \ldots, x_d) : a_i \leq b_i, i \in \{1,2,\ldots,d\}\}$ and a datapoint $\mathbf{x} \in \mathbb{R}^d$ is predicted to be 1 by $f_{(a_1,b_1,a_2,b_2,\ldots,a_d,b_d)}(x_1, x_2, \ldots, x_d)$ if $a_i \leq x_i \leq b_i$ for all $i \in \{1,2,\ldots,d\}$ and -1 otherwise. We will still assume realizability. 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 $1-\delta$.

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
- json (for reading the dataset)
- collections
- 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.

You can find all the necessary starter code and data at `https://vatsalsharan.github.io/spring24/hw1_code.zip`.

Problem 3: 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 are providing starter bootstrap code and you are expected to complete the classes and functions. The dataset we will use is a subset of Breast Cancer Wisconsin (Diagnostic). This is a binary classification dataset. Each data point is represented by (x_i, y_i) with $x_i \in \mathbb{R}^{30}$ and $y_i \in \{0, 1\}$. A sample data point is (x_i, y_i) is

$$x_i = [19.17, 24.8, \ldots, 0.1767, 0.3176, 0.1023],\ y_i = 0.$$

More explanations are available on `https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic`.

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 3.1 Report 4-nearest neighbor accuracy (8pts)

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

$$d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_2 = \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2}. \tag{4}$$

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. If this is a tie in the majority label of the k nearest neighbor set, just return 0 as the label.

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:

$$\text{error rate} = \frac{\text{\# of wrongly classified examples}}{\text{\# of total examples}}. \tag{5}$$

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.

Part 3.2 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 division by zero).

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 test 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.

Part 3.3 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 3.1.

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 small value to the norm of $\mathbf{x}$, so that we do not need to worry about 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 without data transformation*.

Part 3.4 Tuning the hyper-parameter k (10pts)

Again, follow Part 3.1, however, this time we conduct experiments with $k = \{1, 2, 4, 6, 8, 10, 12, 14, 16, 18\}$ on the original dataset without data transformation.

Task: Fill in the code for the function `find_best_k`. Recall that the choice of best hyperparameter is based on validation set.

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 is the final test set error rate you get using your best-k? (1pt) (5) Comment on these results from the perspective of overfitting, generalization and hyper-parameter tuning. (3pts).

Code submission

Make sure to include your complete filled-in code in the single zip file that you upload for all your codes for the HW.

Problem 4: 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 can be found in your starter code and 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))
```

Part 4.1 (6pts) Least-squares regression has the closed form solution $\mathbf{w}_{LS} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$, 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.

Task: Fill in the code for the function `closed_form` and `square_loss`.

Part 4.2 (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. You can also have more graphs on separate step size if the curves with all 3 step sizes are hard to read. 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 among $\{0.00005, 0.0005, 0.0007\}$ that had the best final objective function value and the corresponding objective function value.

Task: Fill in the code for the function `gradient_descent`. You may want to plot the curves for a subset of the step sizes to make the curves distinguishable.

Part 4.3 (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 $f_i(\mathbf{w}) = (\mathbf{w^T x_i} - y_i)^2$.

Run stochastic gradient descent using step sizes $\{0.0005, 0.005, 0.01\}$ and 1000 iterations. You should initialize $\mathbf{w}$ to be the all-zero vector for all three runs. 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.

Task: Fill in the code for the function `stochastic_gradient_descent`. You may want to plot the curves for a subset of the step sizes to make the curves distinguishable.

Code submission

Make sure to include your complete filled-in code in the single zip file that you upload for all your codes for the HW.