Assignment 1
CSE 446: Machine Learning

University of Washington

1 Policies [0 points]

Please read these policies. **Please answer the three questions below and include your answers marked in a "problem 0" in your solution set.** Homeworks which do not include these answers will not be graded.

Gradescope submission: When submitting your HW, please tag your pages correctly as is requested in gradescope. Untagged homeworks will not be graded, until the tagging is fixed.

Readings: Read the required material.

Submission format: Submit your report as a *single* pdf file. Also, please include all your code in the PDF file in a section at the end of your document, marked "Code"; also specify which problem(s) the code corresponds to. The report (in a single pdf file) must include all the plots and explanations for programming questions (if required). Homework solutions must be organized in order, with all plots arranged in the correct location in your submitted solutions. We highly recommend typesetting your scientific writing using LaTeX(see the website for references for free tools). Writing solutions by hand will be accepted provided they are neat; written solutions need to be scanned and included into a single pdf.

Written work: Please provide succinct answers *along with succinct reasoning for all your answers*. Points may be deducted if long answers demonstrate a lack of clarity. Similarly, when discussing the experimental results, concisely create tables and figures to organize the experimental results. In other words, all your explanations, tables, and figures for any particular part of a question must be grouped together.

Python source code: for the programming assignment. Please note that we will not accept Jupyter notebooks. Submit your code together with a neatly written README file to instruct how to run your code with different settings (if applicable). We assume that you always follow good practice of coding (commenting, structuring); these factors are not central to your grade.

Coding policies: You must write your own code. You are welcome to use any Python libraries for data munging, visualization, and numerical linear algebra. Examples includes Numpy, Pandas, and Matplotlib. You may **not**, however, use any machine learning libraries such as Scikit-Learn, TensorFlow, or PyTorch, unless explicitly specified for that question. If in doubt, post to the message boards.

Collaboration: It is acceptable for you to discuss problems with other students; it is not acceptable for students to look at another students written answers. It is acceptable for you to discuss coding questions with others; it is not acceptable for students to look at another students code. Each student must understand, write, and hand in their own answers. In addition, each student must write and submit their own code in the programming part of the assignment.

Acknowledgments: We expect the students not to refer to or seek out solutions in published material from previous years, on the web, or from other textbooks. Students are certainly encouraged to read extra material for a deeper understanding.

Extra Credit Policy: Before you do the extra credit problems, do all the regular questions. Extra credit points will only be awarded if there are (honest attempts at) answers to *all* the regular questions. This is because they are not designed to be alternative questions to the regular questions.

1.1 List of Collaborators

List the names of all people you have collaborated with and for which question(s).

1.2 List of Acknowledgements

If you do inadvertently find an assignment's answer, acknowledge for which question and provide an appropriate citation (there is no penalty, provided you include the acknowledgement). If not, then write "none".

1.3 Certify that you have read the instructions

Please make sure to read and follow these instructions. Write "I have read and understood these policies" to certify this.

2 Decision Stumps and Trees *[40 points]*

2.1 Criteria for Choosing a Feature to Split *[8 points]*

When choosing a single feature to use to classify—for example, if we are greedily choosing the next feature to use for a subset of the data inside of a decision tree—we proposed in lecture to choose the feature that will give the greatest reduction in total error.

Let D be the dataset under consideration, n be the number of examples in D with label 0 (negative examples), and p be the number of examples in D with label 1 (positive examples).

1. *[4 points]* (Not Splitting) Suppose we have reached the maximum depth allowed by our learning algorithm; we may not use any more features. At this point, we have a partition of the data, D' (which is a subset of D) and it contains n' negative examples and p' positive examples. Assuming that our algorithm is deterministic, what's the smallest number of mistakes (in D') we can make, and how do we obtain it? (Give a formula in terms of the symbols we have introduced above.)
2. *[4 points]* (Splitting) Consider a binary feature ϕ with the following contingency table for D':

y	ϕ	
	0	1
0	n_0	n_1
1	p_0	p_1

Note that $p' = p_0 + p_1$ and $n' = n_0 + n_1$. If we split D' based on feature ϕ, how many mistakes will we make (in D')? (Give a formula in terms of the symbols we have introduced.)

Another way to think about the which-feature-if-any-to-split-on decision is to *maximize* the number of mistakes that we would *correct* by splitting (as opposed to not splitting). This is given by subtracting the answer to 2 from the answer to 1.

Aside: Sometimes it is also helpful to consider other criterion for splits, such as relative error conditions (rather than absolute error conditions) or information theoretic conditions. For example, if we divide this value by $|D'|$ (the number of examples in D'), then we have the absolute reduction in error *rate* achieved by making the split (as opposed to not splitting). This value is often referred to as "error rate reduction."

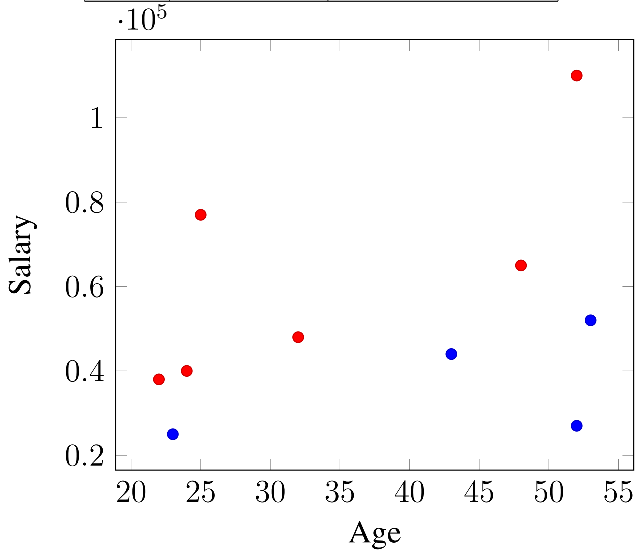
Before proceeding, think carefully about this question: is it possible for a split to *increase* the error rate on the training set? (No points for this, but you should have a confident answer in your mind.)

2.2 Decision Stumps and Trees *[20 points]*

It might be helpful for you to draw the decision boundary, in the figure, of each classifier in each part.

Consider the problem of predicting whether a person has a college degree based on age and salary. The table and graph below contain training data for 10 individuals.

Age	Salary ($)	College Degree
24	40,000	Yes
53	52,000	No
23	25,000	No
25	77,000	Yes
32	48,000	Yes
52	110,000	Yes
22	38,000	Yes
43	44,000	No
52	27,000	No
48	65,000	Yes



1. *[10 points]* (Build Two Decision Stumps) In class, we focused on discrete features. One way to deal with continuous (or ordinal) data is to define binary features based on thresholding of continuous features like Age and Salary. For example, you might let:

$$\phi(x) = \begin{cases} 0 & \text{if Age} \leq 50 \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

The choice of the threshold 50 is, on the surface, arbitrary. In practice, though, for a given feature, you should only consider the best threshold for that feature. While the search for a threshold might seem daunting, the only thresholds worth considering will be midpoints between consecutive feature values, after sorting.[1]

For each of the two features in this dataset (Age and Salary), generate a plot showing the training set accuracy (on the y-axis) as a function of the threshold value (on the x-axis). You will

[1]Imagine moving a threshold just a little bit, but not enough to change the decision for any data point. For example, the threshold in Equation 1 could be reduced to 49 or increased to 51, and there would be no effect on the classification decisions in our dataset.

probably want to automate the calculations. Report the training-set error rates on both decision stumps.

2. *[10 points]* (Build a Decision Tree) Build a decision tree for classifying whether a person has a college degree by greedily choosing threshold splits. Do this by counting errors (as shown in lecture). Show the decision tree and report the training set error rate.

2.3 Multivariate Decision Tree *[12 points]*

Multivariate decision trees are a generalization of the "univariate" decision trees that we have discussed so far. In a multivariate decision tree, more than one attribute can be used in the decision rule for each split. That is, from a geometric point of view, splits need not be orthogonal to a feature's axis.

1. *[9 points]* For the same data, learn a multivariate decision tree where each split makes decisions based on the sign of the function $\alpha x_{\text{Age}} + \beta x_{\text{Income}} - 1$. Draw your tree, including the α, β, and give the final error rate. Specify the criterion you used to design your splits. Again, your objective here is to minimize the training error, in a manner that demonstrates that you are utilizing the extra flexibility of this model.
2. *[3 points]* (Discussion) Multivariate decision trees have practical advantages and disadvantages. List an advantage and a disadvantage multivariate decision trees have compared to univariate decision trees.

3 Estimating the True Loss *[24 points]*

Suppose we obtain N labeled samples from our underlying distribution $\mathcal{D}$. Suppose we break this into N_{train}, N_{dev}, and N_{test} samples for our training, dev, and test set.

Recall our definition of the true error, for the "0/1" loss,

$$\epsilon(f) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathbf{1}\{y \neq f(x)\}]$$

(the subscript $(x, y) \sim \mathcal{D}$ makes clear that our input-output pairs are sampled according to $\mathcal{D}$). Our training, dev, and test losses are defined as:

$$\widehat{\epsilon}_{\text{train}}(f) = \frac{1}{N_{\text{train}}} \sum_{(x,y)\in\text{Training Set}} \mathbf{1}\{y \neq f(x)\}$$

$$\widehat{\epsilon}_{\text{dev}}(f) = \frac{1}{N_{\text{dev}}} \sum_{(x,y)\in\text{Dev Set}} \mathbf{1}\{y \neq f(x)\}$$

$$\widehat{\epsilon}_{\text{test}}(f) = \frac{1}{N_{\text{test}}} \sum_{(x,y)\in\text{Test Set}} \mathbf{1}\{y \neq f(x)\}$$

We then train our algorithm (say a decision tree), where we use our training set to build the tree and the dev set for hyperparameter tuning (e.g. we use it to determine when to stop our algorithm). Let $\widehat{f}$ be the hypothesis returned by our learning procedure.

1. *[4 points]* (bias: the test error) Show the test error is an unbiased estimate of our true error. Specifically, show that:
$$\mathbb{E}[\widehat{\epsilon}_{\text{test}}(\widehat{f})] = \epsilon(\widehat{f})$$
2. *[4 points]* (bias: the train/dev error) Is the above equation true (in general) with regards to the training or dev loss? If so, why? If not, give a clear argument as to where your previous argument breaks down.
3. *[4 points]* (variance) Show that the variance of $\widehat{\epsilon}_{\text{test}}(\widehat{f})$ is bounded by $\frac{1}{4N_{\text{test}}}$. (Hint: See HW0. You may use basic properties of the variance to simply your argument from HW0, if you are not interested in re-deriving things for this setting.)
4. *[6 points]* (the central limit theorem) What do we expect the distribution of $\widehat{\epsilon}_{\text{test}}(\widehat{f})$ to look like, for large N_{test}? Why can we invoke the central limit theorem?
5. *[6 points]* (confidence intervals) Now let us show that our estimate of the true error of $\widehat{f}$ using the test set is $\widehat{\epsilon}_{\text{test}}(\widehat{f}) \pm \frac{1}{\sqrt{N_{\text{test}}}}$ (with 95% probability). Specifically, show that with probability greater than 95%, that the following will hold:
$$|\widehat{\epsilon}_{\text{test}}(\widehat{f}) - \epsilon(\widehat{f})| \leq \frac{1}{\sqrt{N_{\text{test}}}}$$
You may assume N large enough so that the central limit theorem is accurate. (In practice, the central limit theorem kicks in pretty quickly, as we saw in HW0. In theory, there are techniques to make the above precise for any N, which leads to minor adjustments in the above bound).

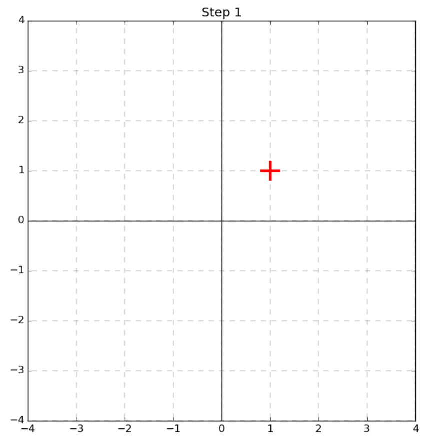
4 Perceptron Exercise *[20 points]*

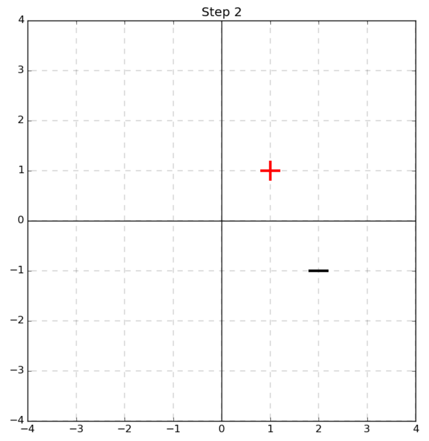
Recall that a perceptron learns a linear classifier with weight vector $\mathbf{w}$. It predicts
$$\widehat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$$
(We assume here that $\widehat{y} \in \{+1, -1\}$. Also, note that we are *not* using a bias weight b.) When the perceptron makes a mistake on the nth training example, it updates the weights using the formula
$$\mathbf{w} \leftarrow \mathbf{w} + y_n \mathbf{x}_n$$

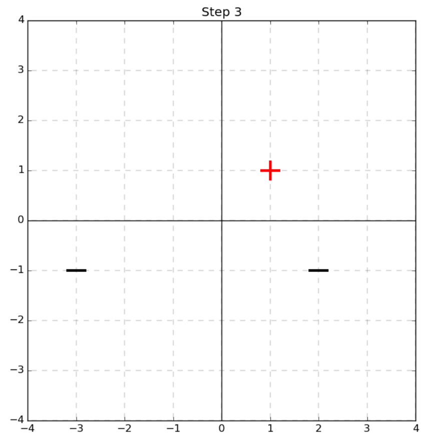
Imagine that we have each $\mathbf{x}_n \in \mathbb{R}^2$, and we encounter the following data points

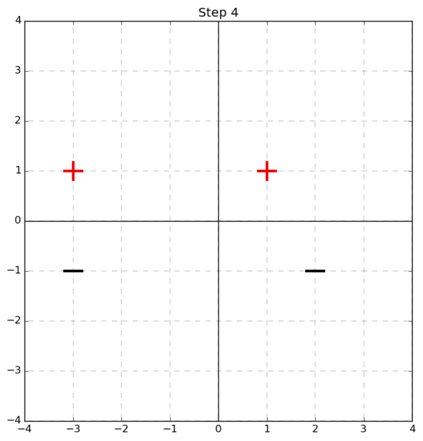
$\mathbf{x}[1]$	$\mathbf{x}[2]$	y
1	1	1
2	-1	-1
-3	-1	-1
-3	1	1

1. *[10 points]* Starting with $\mathbf{w} = [0 \quad 0]^\top$, use the perceptron algorithm to learn on the data points in the order from top to bottom. Show the perceptron's linear decision boundary after observing each data point in the graphs below. Be sure to show which side is classified as positive.

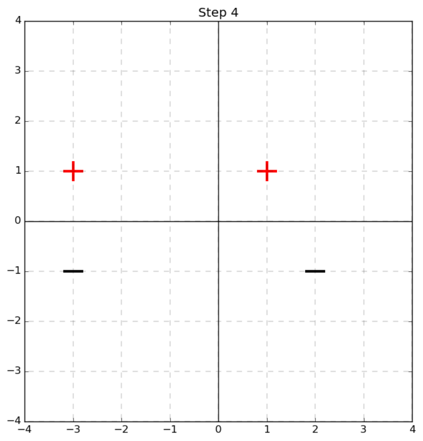

Step 1


Step 2


Step 3



2. *[5 points]* Does our learned perceptron maximize the margin between the training data and the decision boundary? If not, draw the maximum-margin decision boundary on the graph below.



3. *[5 points]* Assume that we continue to collect data and train the perceptron. Suppose all the data we see (including the previous data) are linearly separable, and there exists some vector w_* (of unit norm, i.e. $\|\mathbf{w}_*\| = 1$) such that for all i:

$$y_i(\mathbf{w}_* \cdot \mathbf{x}_i) \geq 0.5 = \gamma\,.$$

Also, suppose that our inputs (past and future) satisfy the maximum norm bound $\|\mathbf{x}_i\| \leq 8$ (for all i). Note this norm bound on inputs is not 1. What is the maximum number of mistakes we can make on future data?

5 Implementation: K-Means *[30 points]*

You will need files `kmeans_template.py`, `digit.txt`, and `labels.txt`, which are available on the website.

In class we discussed the K-means clustering algorithm. You will implement the K-means algorithm on digit data. A template has been prepared for you, and your job is to fill in the functions as described below.

The data are held in two files: one for the inputs and one for the output. The file `digit.txt` contains the inputs: namely, 1000 observations of 157 pixels (a randomly chosen subset of the original 784) from images containing handwritten digits. The file `labels.txt` contains the true digit label (**one**, **three**, **five**, or **seven**, each coded as a digit). The Python template handles reading in the data; just make sure that all the files live in the same directory as your code.

The labels correspond to the digit file, so the first line of `labels.txt` is the label for the digit in the first line of digit.txt.

1. *[10 points]* (Algorithm) Your algorithm will be implemented as follows:
   (a) Initialize k starting centers by randomly choosing k points from your data set. You should implement this in the `initialize` function.
   (b) Assign each data point to the cluster associated with the nearest of the k center points. Implement this by filling in the `assign` function.
   (c) Re-calculate the centers as the mean vector of each cluster from (2). Implement this by filling in the `update` function.
   (d) *[1 points]* Repeat steps (2) and (3) until convergence. This wrapping is already done for you in the `loop` function, which lives inside the `cluster` function.
2. *[5 points]* (Within-Group Sum of Squares) One way to formalize the goal of clustering is to minimize the variation *within* groups, while maximizing the variation *between* groups. Under this view, a good model has a low "sum of squares" within each group.
   We define sum of squares as follows. Let $\boldsymbol{\mu}_k$ (a vector) be the mean of the observations (each one a vector) in kth cluster. Then the within group sum of squares for kth cluster is defined as:
   $$SS(k) = \sum_{i \in k} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|_2^2$$
   where "$i \in k$" ranges over the set of indices i of observations assigned to the kth cluster. Note that the term $\|\mathbf{x}_i - \boldsymbol{\mu}_k\|_2$ is the Euclidean distance between $\mathbf{x}_i$ and $\boldsymbol{\mu}_k$, and therefore should be calculated as $\|\mathbf{x}_i - \boldsymbol{\mu}_k\|_2 = \sqrt{\sum_{j=1}^{d} (\mathbf{x}_i[j] - \boldsymbol{\mu}_k[j])^2}$, where d is the number of dimensions. Note that that term is squared in $SS(k)$, cancelling the square root in the formula for Euclidean distance.
   If there are K clusters total then the "sum of within-group sum of squares" is the sum of all K of these individual $SS(k)$ terms.
   In the code, the sum of within-group sum of squares is referred to (for brevity) as the *distortion*, i.e. $\sum_k SS(k)$. Your job is to fill in the function `compute_distortion`.
3. *[5 points]* (Mistake Rate) Typically K-means clustering is used as an exploration tool on data where true labels are not available, and might not even make sense. (For example, imagine clustering images of all of the foods you have eaten this year.)
   Given that, here, we know the *actual* assignment labels for each data point, we can analyze how well the clustering recovered the true labels. For kth cluster, let the observations assigned to

that cluster *vote* on a label (using their true labels as their votes). Then let its assignment be the label with the most votes. If there is a tie, choose the digit that is smaller numerically. This is equivalent to deciding that each cluster's members will all take the same label, and choosing the label for that cluster that lead to the fewest mistakes.
For example, if for cluster two we had 270 observations labeled **one**, 50 labeled **three**, 9 labeled **five**, and 0 labeled **seven** then that cluster will be assigned value **one** and had 50 + 9 + 0 = 59 mistakes.
If we add up the total number of mistakes for each cluster and divide by the total number of observations (1000) we will get our total mistake rate, between 0 and 1. Lower is better.
Your job is to fill in the function `label_clusters`, which implements the label assignment. Once you have implemented this, the function `compute_mistake_rate` (already written) will compute the mistake rate for you.

4. *[10 points]* (Putting it All Together) Once you have filled in all the functions, you can run the Python script from the command line with `python kmeans_template.py`. The script will loop over the values $K \in \{1, 2, \ldots, 10\}$. For each value of k, it will randomly initialize 5 times using your initialization scheme, and keep the results from the run that gave the lowest distortion. For the best run for each K, it will compute the mistake rate using your cluster labeling function. The code will output a figure named `kmeans.png`. The figure will have two plots. The top one shows distortion as a function of K. The bottom one shows the mistake rate, also as a function of K.
Write down your thoughts on these results. Are they what you expected? Did you think that within-group sum of squares and mistake rate would go up or decrease as K increased? Did the plots confirm your expectations?

6 A "Baby" No Free Lunch Theorem *[14 points]*

Suppose that Alice has a function $f : \{0,1\}^m \rightarrow \{0,1\}$ mapping binary sequences of length m to a label. Suppose Bob wants to learn this function. Alice will only provide labelled random examples. Specifically, Alice will provide Bob with a training set under the following data generating distribution: the input x is a uniformly at random binary string of length m and the corresponding label is $f(x)$ (the label is deterministic, based on Alice's function). Bob seeks to learn a function $\widehat{f}$ which has low mis-classification expected loss.

Remember to provide short explanations.

1. *[3 points]* (The # of inputs) How many possible inputs are there for m length binary inputs? And, specifically for $m = 5$, how many are there?
2. *[5 points]* (The # of functions) The function f is one function out of how many possible functions acting on m length binary sequences? And, specifically for $m = 5$, how many possible functions are there?
3. *[6 points]* (Obtaining low expected loss) Assume that Bob has no knowledge about how Alice chooses her function f. Suppose Bob wants to *guarantee* that he could learn with an expected loss that is non-trivially better than chance, e.g. suppose Bob wants an expected misclassification loss less than 25% (or any loss that is a constant less than 50%). Roughly (in

"Big-Oh" notation), how many training samples does Bob need to have in order to guarantee such an expected loss? (Hint: Note that the number of training samples will be almost the same as number of distinct inputs in the training set, due to the "coupon collectors" problem. You may assume these two are the same (there is a on log factor difference). In the coupon collectors problem, there is a uniform distribution over K outcomes. We know that with $O(K\log(K))$ samples, we will observe $O(K)$ distinct outcomes, with high probability (say greater than 95% chance).)

Interpretation: This "No Free Lunch" theorem essentially shows that, in general, we should not be able to learn without prior knowledge about how Alice is choosing her function. The readings CIML Ch 2 discusses these issues.