CS 229, Winter 2024
Problem Set #3 Solutions

YOUR NAME HERE (`YOUR SUNET HERE`)

Due Wednesday, February 21 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be as concise as possible.

(2) If you have a question about this homework, we encourage you to post your question on our Ed forum, at `https://edstem.org/us/courses/51342/discussion/`.

(3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work.

(4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted.

(5) The due date is Wednesday, February 21 at 11:59 pm. If you submit after Wednesday, February 21 at 11:59 pm, you will begin consuming your late days. The late day policy can be found in the course website: Course Logistics and FAQ.

All students must submit an electronic PDF version of the written question including plots generated from the codes. We highly recommend typesetting your solutions via LaTeX. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup. Please make sure that your PDF file and zip file are submitted to the corresponding Gradescope assignments respectively. We reserve the right to not give any points to the written solutions if the associated code is not submitted.

Honor code: We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solution independently, and without referring to written notes from the joint session. Each student must understand the solution well enough in order to reconstruct it by him/herself. It is an honor code violation to copy, refer to, or look at written or code solutions from a previous year, including but not limited to: official solutions from a previous year, solutions posted online, and solutions you or someone else may have written up in a previous year. Furthermore, it is an honor code violation to post your assignment solutions online, such as on a public git repo. We run plagiarism-detection software on your code against past solutions as well as student submissions from previous years. Please take the time to familiarize yourself with the Stanford Honor Code[1] and the Stanford Honor Code[2] as it pertains to CS courses.

Honor code: We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he

[1] https://communitystandards.stanford.edu/policies-and-guidance/honor-code
[2] https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1164/handouts/honor-code.pdf

collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

Regarding Notation: The notation used in this problem set matches the notation used in the lecture notes. Some notable differences from lecture notation:

- The superscript "(i)" represents an index into the training set – for example, $x^{(i)}$ is the i-th feature vector and $y^{(i)}$ is the i-th output variable. In lecture notation, these would instead be expressed as x_i and y_i.
- The subscript j represents an index in a vector – in particular, $x_j^{(i)}$ represents the j-th feature in the feature vector $x^{(i)}$. In lecture notation, $x_j^{(i)}$ would be $h_j(x_i)$ or $x_i[j]$.
- The vector that contains the weights parameterizing a linear regression is expressed by the variable θ, whereas lectures use the variable $\mathbf{w}$. As such, $\theta_0 = w_0$, $\theta_1 = w_1$, and so on.

An overview of this notation is also given at the beginning of the lecture notes (pages 6-7).

1. **[35 points] Decision trees**

Consider the problem of predicting if a person has a college degree based on age and salary. Table 1 contains training data for 10 individuals.

Age	Salary ($1k)	College degree
24	40	Yes
53	52	No
23	25	No
25	77	Yes
32	48	Yes
52	110	Yes
22	38	Yes
43	44	No
52	27	No
48	65	Yes

Table 1: Training data for predicting college degree.

For questions below, the answers may not be unique. Any plausible solution is acceptable. Keep two significant decimals in part (a) and (c).

(a) [5 points] Build a decision tree for classifying whether a person has a college degree by greedily choosing threshold splits that minimize the classification error. Provide a list of all splits and the classification error reduction at each split.

Answer: There are 2 splits.

i. Salary $> 30k$. Classification error reduction

$$0.4 - 0.2 = 0.2.$$

ii. Given salary $> 30k$, the new split age > 52.5. Classification error reduction

$$0.2 - 0.1 = 0.1.$$

Since there is no split that can further bring down the classification error, we will stop the splitting process here and end up with 2 splits.

(b) [15 points] Now let's implement a classification, univariate decision tree with misclassification loss (mentioned in equation 1). The starter code is provided in `src/decision_trees_general/decision_tree.py`. Fill in the functions marked with **#TODO**. You are not allowed to use any package other than NumPy. You **cannot** assume there are only two classes. **Deliverables**: report the accuracy output when running the Python script. For reference, the staff solution gives the same expected accuracy in part (a) for the college degree dataset (Table 1) and 93.33% for the iris dataset.

Answer: Accuracy for college degree dataset: 0.9
Accuracy for Iris dataset: 0.9333

(c) [5 points] A multivariate decision tree is a generalization of univariate decision trees, where more than one attribute can be used in the decision rule for each split. For the same data, learn a multivariate decision tree where each decision rule is a linear classifier that makes decisions based on the sign of $\alpha x_{\text{age}} + \beta x_{\text{income}} - 1$. Provide a list of all splits with the

classification error reduction at each split, as well as α, β. For α and β, keep two significant decimals.

Answer: The data can be split perfectly with a linear classifier, i.e., one split. One such split is formed by the boundary line

$$-0.30x_{\text{age}} + 0.32x_{\text{income}} - 1 = 0.$$

Classification error reduction of the only split

$$0.4 - 0 = 0.4.$$

(d) [4 points] Multivariate decision trees have practical advantages and disadvantages. List two advantages and two disadvantages multivariate decision trees have compared to univariate decision trees.

Answer: Advantages.

- Multivariate decision trees may result in trees with smaller depth.
- Multivariate decision trees are less biased.

Disadvantages.

- Multivariate decision trees are more difficult to interpret.
- Multivariate decision trees are more computationally heavy.
- Multivariate decision trees are more prone to overfitting.

(e) [6 points]

```
1:  function DecisionTree(Data)
2:      if all points in Data have same label y or max height reached then
3:          return Leaf(majority vote for y in Data)
4:      else
5:          for each feature h_i do
6:              for each value v of feature h_i in Data do
7:                  Data_1, Data_2 = Split(Data, h_i ≤ v)
8:                  Error_{i,v} = ClassificationError(Data_1) + ClassificationError(Data_2)
9:              end for
10:         end for
11:         h*, v* = choose feature h_i and split v that has smallest Error_{i,v}
12:         Data_1, Data_2 = Split(Data, h* ≤ v*)
13:         return Branch(h* ≤ v*, DecisionTree(Data_1), DecisionTree(Data_2))
14:     end if
15: end function
```

Now imagine we want to predict a person's salary from their age and whether or not they have a college degree, which is a regression task. Being the lazy coder you are, you decide to reuse your existing classification tree code above, modifying as few lines as possible to implement a regression tree.

(i) [3 points] Recall that at a leaf node, the decision tree for classification returns the majority vote of training datapoints at that leaf. For regression, what would an appropriate choice of output be? Provide the line of code that needs to be modified and write psuedocode for the suggested modification.

(ii) [3 points] Recall that the decision tree for classification chooses the split that minimizes classification error. For regression, what would we aim to minimize? Provide the line of code that needs to be modified and write psuedocode for the suggested modification.

Answer:

(i) Output would be mean of data at node. Line 3 needs to be modified to "Leaf(average value for y in $Data$)".

(ii) We would aim to minimize residual sum of squares (RSS). Line 8 needs to be modified to "$Error_{i,v}$ =RSS($Data_1$)+RSS($Data_2$)."

2. **[20 points] Decision Trees and Gini Loss**

When growing a decision tree, we split the input space in a greedy, top-down, recursive manner. Given a parent region R_p, we can choose a split $s_p(j,t)$ which yields two child regions $R_1 = \{X \mid x_j < t, X \in R_p\}$ and $R_2 = \{X \mid x_j \geq t, X \in R_p\}$. Assuming we have defined a per region loss $L(R)$, at each branch we select the split that minimizes the weighted loss of the children:

$$\min_{j,t} \frac{|R_1|L(R_1) + |R_2|L(R_2)}{|R_1| + |R_2|}$$

When performing classification, a commonly used loss is the Gini loss, defined for the K-class classification problem as:

$$G(R_m) = G(\vec{p}_m) = \sum_{k=1}^{K} p_{mk}(1 - p_{mk})$$

Where $\vec{p}_m = \begin{bmatrix} p_{m1} & p_{m2} & \dots & p_{mK} \end{bmatrix}$ and p_{mk} is the proportion of examples of class k that are are present in region R_m. However, we are oftentimes more interested in optimizing the final misclassification loss:

$$M(R_m) = M(\vec{p}_m) = 1 - \max_k p_{mk} \tag{1}$$

For the problems below, assume we are dealing with binary classification and that there are no degenerate cases where positive and negative datapoints overlap in the feature space.

(a) [5 points] Show that for any given split, the weighted Gini loss of the children can not exceed that of the parent. (**Hint**: first show that the Gini loss is strictly concave. And then use the fact that G is strictly concave meaning:

$$\forall p_1 \neq p_2, \forall t \in (0,1) : G(tp_1 + (1-t)p_2) > tG(p_1) + (1-t)G(p_2)$$

Answer: First note that we can re-write the Gini loss in the binary class in terms of a single proportion p_m, which we define as the proportion of positive examples in the region R_m:

$$\begin{aligned} G(\vec{p}_m) = G(p_m) &= p_m(1-p_m) + (1-p_m)(1-(1-p_m)) \\ &= 2p_m - 2p_m^2 \end{aligned}$$

Thus, $\frac{d^2G}{dp_m^2}(p_m) = -4$ and the Gini loss is strongly concave.
Defining $t = \frac{|R_1|}{|R_1|+|R_2|}$, we can write the parent region R_p proportion p_p as:

$$p_p = t * p_1 + (1-t) * p_2$$

Thus, we can use the definition of concavity to show that:

$$\begin{aligned} G(R_p) &= G(p_p) \\ &= G(tp_1 + (1-t)p_2) \\ &\geq tG(p_1) + (1-t)G(p_2) \\ &= \frac{|R_1|G(R_1) + |R_2|G(R_2)}{|R_1| + |R_2|} \end{aligned}$$

(b) [5 points] List out the cases where Gini loss will stay the same after a split. Show why these do not violate the strong concavity of the Gini loss. Briefly explain why these cases do not prevent a fully grown tree from achieving zero Gini loss. (**Hint**: Recall the definition of strict concavity).

Answer: There are two cases where the loss of the children will not be lower than the loss of the parent. The first is when one of the two children are empty. Then either $t = 0$ or $t = 1$, which is outside the range where the inequality is forced to be strict. The second case is when the proportion of positive and negative examples in each child remains the same as that of the parent. Then we have that $p_p = p_1 = p_2$, which again allows the inequality to not be strict.

These cases do not prevent a tree from fully fitting the data, as the lack of degenerate examples will always allow us to pick splits that have non-zero cardinality children, and the constant diminishing of the cardinality of the children will eventually force an uneven split where $p_1 \neq p_2$.

(c) [4 points] If instead we use misclassification loss, what additional case causes the loss to stay the same after a split? Show why this is (hint: you may find it useful to define $N_m = |R_m|$ and N_{mk} as the number of examples of class k present in R_m).

Answer: If the majority class of the two children match that of the parent, the misclassification loss will remain the same. To show this, first define class c as the majority class prediction. We then have that:

$$\begin{aligned}
\frac{|R_1|M(R_1) + |R_2|M(R_2)}{|R_1| + |R_2|} &= \frac{N_1(1 - p_{1c}) + N_2(1 - p_{2c})}{N_1 + N_2} \\
&= \frac{N_1(1 - \frac{N_{1c}}{N_1}) + N_2(1 - \frac{N_{2c}}{N_2})}{N_1 + N_2} \\
&= 1 - \frac{N_{1c} + N_{2c}}{N_1 + N_2} \\
&= 1 - \frac{N_{pc}}{N_p} = M(R_p)
\end{aligned}$$

(d) [4 points]

Bagging, short for "bootstrap aggregating," is a powerful ensemble learning technique that aims to improve the stability and accuracy of machine learning algorithms. It leverages the concept of bootstrapping, which involves simulating the drawing of a new sample from the true underlying distribution of the training set, as the training set is presumed to be a representative sample of the true distribution. In practice, this is done by generating new datasets through uniform sampling with replacement from the original dataset.

The "aggregating" component of bagging comes into play by repeating this bootstrapping process for each model in the ensemble, allowing each to be trained independently on a unique dataset. When considering decision trees, the method's utility becomes evident as it mitigates overfitting by ensuring that each tree in the ensemble is exposed to different subsets of the training data. This reduces the likelihood that the ensemble will fixate on particular data points, thus lowering overall variance. Statistically, each bootstrapped sample will contain, in expectation, about $1 - \frac{1}{e} \approx 63.2\%$ of unique data points from the original dataset.

However, the effectiveness of bagging depends on the characteristics of the underlying models. For models with low variance (and typically high bias), bagging may produce very

similar models, which diminishes its benefits. On the other hand, with high-variance models such as decision trees, bagging capitalizes on the models' instability to promote diversity in the ensemble, thereby enhancing its performance. This results in an ensemble that maintains low bias while reducing variance, leading to a robust aggregate model.

Consider a training set X. In bootstrap sampling, each time we draw a random sample Z of size N from the training data and obtain $Z_1, Z_2, ..., Z_B$ after B times, i.e. we generate B different bootstrapped training data sets. If we apply bagging to regression trees, each time a tree $T_i (i = 1, 2, ..., B)$ is grown based on the bootstrapped data Z_i, and we average all the predictions to get:

$$\hat{T(x)} = \frac{1}{B}\sum_{i=1}^{B} T_i(x)$$

Now, if $T_1, T_2, ..., T_B$ is independent from each other, but each has the same variance σ^2, the variance of the average $\hat{T}$ is σ^2/B. However, in practice, the bagged trees could be similar to each other, resulting in correlated predictions. Assume $T_1, T_2, ..., T_B$ still share the same variance σ^2, but have a positive pair-wise correlation ρ. We define the correlation between two random variables as:

$$Corr(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var(X)}\sqrt{Var(Y)}}$$

Thus, we have $\rho = Corr(T_i(x), T_j(x)), i \neq j$.

Show that in this case, the variance of the average is given by:

$$Var(\frac{1}{B}\sum_{i=1}^{B} T_i(x)) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

Answer:

$$\begin{aligned}
Var(\frac{1}{B}\sum_{i=1}^{B} T_i(x)) &= \frac{1}{B^2}\sum_{i=1}^{B}\sum_{j=1}^{B} Cov(T_i(x), T_j(x)) \\
&= \frac{1}{B^2}\sum_{i=1}^{B}(Cov(T_i(x), T_i(x)) + \sum_{j\neq i}^{B} Cov(T_i(x), T_j(x))) \\
&= \frac{1}{B^2}\sum_{i=1}^{B}(\sigma^2 + (B-1)\sigma^2\rho) \\
&= \frac{B(\sigma^2 + (B-1)\sigma^2\rho)}{B^2} \\
&= \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2
\end{aligned}$$

3. **[12 points] AdaBoost**

Consider building an ensemble of decision stumps f_t with the AdaBoost algorithm,

$$F(x) = \text{sign}\Big(\sum_{t=1}^{T} \hat{w}_t f_t(x)\Big).$$

Figure 1 displays a 2-dimensional training dataset, as well as the first stump chosen. A stump predicts binary $+1/-1$ values, and depends only on one coordinate value (the split point). The little arrow indicates the positive side where the stump predicts $+1$. All points start with uniform weights.

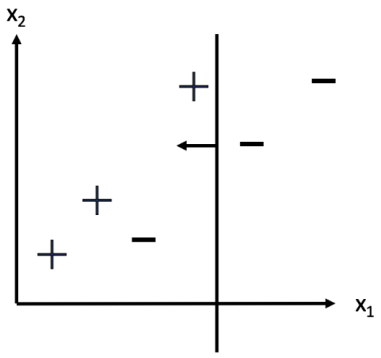


Figure 1: 2-dimensional labeled data, where '+' corresponds to class $y = +1$ and '-' corresponds to class $y = -1$. The decision boundary for the first decision stump is shown. The arrow points in the positive direction from this decision boundary.

(a) [4 points] **Circle all the point(s)** in Figure 1 whose weight(s) will increase as a result of incorporating the first stump (the weight update due to the first stump).

Answer: We will refer to the model importances $\hat{w}_t$ as coefficients and the data point importances α_i as weights. We are given that all points start with uniform weights, i.e., $\alpha_i = \frac{1}{6}$. If we predict using the first decision stump shown in Figure 1, then the model makes 1 mistake (highlighted in Figure 2) and the weight of this point increases when the weights are recomputed using Eq. (2).

$$\alpha_i \leftarrow \begin{cases} \alpha_i e^{-\hat{w}_1}, & f_t(x_i) = y_i \\ \alpha_i e^{\hat{w}_1}, & f_t(x_i) \neq y_i \end{cases} \tag{2}$$

The above equation states that the weight of correctly classified points will be decreased, while the weight of incorrectly classified points will be increased.

(b) [4 points] Draw a possible stump that we could select at the next boosting iteration. You need to draw both the decision boundary and its positive orientation. The answer may not be unique and any plausible solution is acceptable.

Answer: A possible stump is shown in Figure 3. This stump will have the same number of mistakes as stump 1: stump 1 mis-classified point 6, while stump 2 mis-classified point 3. Please note that the weight corresponding to the point misclassified (point 3 in Figure 3) by stump 2 is *less* than the weight corresponding to the point misclassified (point 6 in 2) by stump 1. This is because point 3 was classified correctly by stump 1 and when we updated the weights using Eq. (2), the weight of point 3 was reduced while the weight of point 6 was increased.

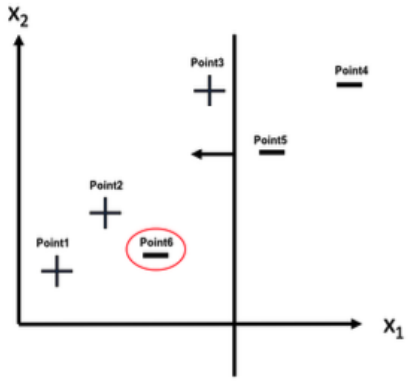


Figure 2: Point whose weight will increase is highlighted in red color.

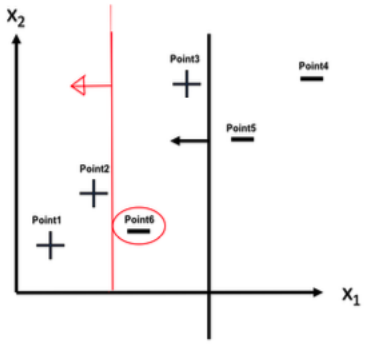


Figure 3: Figure showing second stump in red color.

(c) [4 points] Will the second stump receive higher coefficient in the ensemble than the first? In other words, will $\hat{w}_2 > \hat{w}_1$? Briefly explain your answer. (No calculation should be necessary.)

Answer: Yes. Observe that, although stumps 1 and 2 each mis-classify a single point, the point mis-classified by stump 1 has a higher weight than the point mis-classified by stump 2 in the second iteration. Therefore, the weighted mis-classification rate is lower for stump 2, and stump 2 is assigned a higher model weight to reflect its better (weighted) predictive performance.gn*

4. **[20 points] AdaBoost Performance**

We learned about boosting in lecture. Statistician Kevin Murphy claims that "It can be shown that, as long as each base learner has an accuracy that is better than chance (even on the weighted dataset), then the final ensemble of classifiers will have higher accuracy than any given component." We will now verify this in the AdaBoost framework.

(a) [3 points] Given a set of n observations (x_i, y_i) where y_i is the label $y_i \in \{-1, 1\}$, let $f_t(x)$ be the weak classifier at step t and let $\hat{w}_t$ be its weight. First we note that the final classifier after T steps is defined as

$$F(x) = \text{sign}\left\{\sum_{t=1}^{T} \hat{w}_t f_t(x)\right\} = \text{sign}\{f(x)\},$$

where

$$f(x) = \sum_{t=1}^{T} \hat{w}_t f_t(x).$$

We can assume that $f(x)$ is never exactly zero.
Show that

$$\varepsilon_{\text{training}} := \frac{1}{n}\sum_{i=1}^{n} 1_{\{F(x_i)\neq y_i\}} \leq \frac{1}{n}\sum_{i=1}^{n} \exp(-f(x_i)y_i),$$

where $1_{\{F(x_i)\neq y_i\}}$ is 1 if $F(x_i) \neq y_i$ and 0 otherwise.
Answer: It suffices to show that for any i,

$$1_{\{F(x_i)\neq y_i\}} \leq \exp(-f(x_i)y_i). \tag{3}$$

In fact, if $F(x_i) \neq y_i$, then the sign of $f(x_i)$ and y_i are different, and thus $f(x_i)y_i < 0$. This means that $\exp(-f(x_i)y_i) > 1 = 1_{\{F(x_i)\neq y_i\}}$. Similarly, we can show Eq. (3) is true if $F(x_i) = y_i$.

(b) [8 points] The weight for each data point i at step $t+1$ can be defined recursively by

$$\alpha_{i,(t+1)} = \frac{\alpha_{i,t}\exp(-\hat{w}_t f_t(x_i)y_i)}{Z_t},$$

where Z_t is a normalizing constant ensuring the weights sum to 1

$$Z_t = \sum_{i=1}^{n} \alpha_{i,t}\exp(-\hat{w}_t f_t(x_i)y_i).$$

Show that

$$\frac{1}{n}\sum_{i=1}^{n} \exp(-f(x_i)y_i) = \prod_{t=1}^{T} Z_t.$$

Answer: Notice that

$$\frac{\alpha_{i,(t+1)}}{\alpha_{i,t}} = \frac{\exp(-\hat{w}_t f_t(x_i)y_i)}{Z_t},$$

then

$$\frac{\alpha_{i,(T+1)}}{\alpha_{i,1}} = \prod_{t=1}^{T} \frac{\alpha_{i,(t+1)}}{\alpha_{i,t}} = \prod_{t=1}^{T} \frac{\exp(-\hat{w}_t f_t(x_i) y_i)}{Z_t} = \frac{\exp(-y_i \sum_{t=1}^{T} \hat{w}_t f_t(x_i))}{\prod_{t=1}^{T} Z_t} = \frac{\exp(-y_i f(x_i))}{\prod_{t=1}^{T} Z_t}.$$

Since $\alpha_{i,1} = 1/n$ and $\sum_{i=1}^{n} \alpha_{i,(T+1)} = 1$, we have

$$\sum_{i=1}^{n} \frac{\exp(-y_i f(x_i))}{\prod_{t=1}^{T} Z_t} = \sum_{i=1}^{n} \frac{\alpha_{i,(T+1)}}{\alpha_{i,1}} = n \sum_{i=1}^{n} \alpha_{i,(T+1)} = n.$$

(c) [9 points] We showed above that training error is bounded above by $\prod_{t=1}^{T} Z_t$. At step t the values Z_1, Z_2, ..., Z_{t-1} are already fixed therefore at step t we can choose α_t to minimize Z_t. Let

$$\varepsilon_t = \sum_{i=1}^{n} \alpha_{i,t} 1_{\{f_t(x_i) \neq y_i\}}$$

be the weighted training error for the weak classifier $f_t(x)$. Then we can re-write the formula for Z_t as

$$Z_t = (1 - \varepsilon_t) \exp(-\hat{w}_t) + \varepsilon_t \exp(\hat{w}_t).$$

(i) [3 points] First find the value of $\hat{w}_t$ that minimizes Z_t. Then show that the corresponding optimal value is

$$Z_t^{\text{opt}} = 2\sqrt{\varepsilon_t(1 - \varepsilon_t)}.$$

(ii) [3 points] Assume we choose Z_t this way. Then re-write $\varepsilon_t = 1/2 - \gamma_t$, where $\gamma_t > 0$ implies better than random and $\gamma_t < 0$ implies worse than random. Then show that

$$Z_t \leq \exp(-2\gamma_t^2).$$

(You may want to use the fact that $\log(1 - x) \leq -x$ for $0 \leq x < 1$.)

(iii) [3 points] Finally, show that if each classifier is better than random, i.e., $\gamma_t > \gamma$ for all t and $\gamma > 0$, then

$$\varepsilon_{\text{training}} \leq \exp(-2T\gamma^2),$$

which shows that the training error can be made arbitrarily small with enough steps.

Answer:

(i) We take the partial derivative of Z_t with regard to $\hat{w}_t$,

$$\frac{\partial Z_t}{\partial \hat{w}_t} = \varepsilon_t \exp(\hat{w}_t) - (1 - \varepsilon_t) \exp(-\hat{w}_t).$$

Set the partial derivative to zero and we obtain

$$\hat{w}_t^{\text{opt}} = \frac{1}{2} \log\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right).$$

Plug the above $\hat{w}_t^{\text{opt}}$ in Z_t,

$$Z_t^{\text{opt}} = (1 - \varepsilon_t)\sqrt{\frac{\varepsilon_t}{1 - \varepsilon_t}} + \varepsilon_t \sqrt{\frac{1 - \varepsilon_t}{\varepsilon_t}} = 2\sqrt{\varepsilon_t(1 - \varepsilon_t)}.$$

(ii) Plug $\varepsilon_t = 1/2 - \gamma_t$ into the formula of Z_t^{opt},

$$Z_t^{\text{opt}} = \sqrt{(1-2\gamma_t)(1+2\gamma_t)} = \sqrt{1-4\gamma_t^2}.$$

Since $1 - x \leq e^{-x}$ for $0 \leq x < 1$,

$$\sqrt{1-4\gamma_t^2} \leq \sqrt{\exp(-4\gamma_t^2)} = \exp(-2\gamma_t^2),$$

and we finish the proof.

(iii) Combine (a), (b), and (c)(ii),

$$\varepsilon_{\text{training}} \leq \prod_{t=1}^{T} Z_t \leq \exp\left(-2\sum_{t=1}^{T}\gamma_t^2\right).$$

If $\gamma_t > \gamma$ for all t and $\gamma > 0$, then

$$\varepsilon_{\text{training}} \leq \exp\left(-2T\gamma^2\right).$$

5. **[25 points] A Simple Neural Network**

Let $X = \{x^{(1)}, \cdots, x^{(n)}\}$ be a dataset of n samples with 2 features, i.e $x^{(i)} \in \mathbb{R}^2$. The samples are classified into 2 categories with labels $y^{(i)} \in \{0, 1\}$. A scatter plot of the dataset is shown in Figure 4:

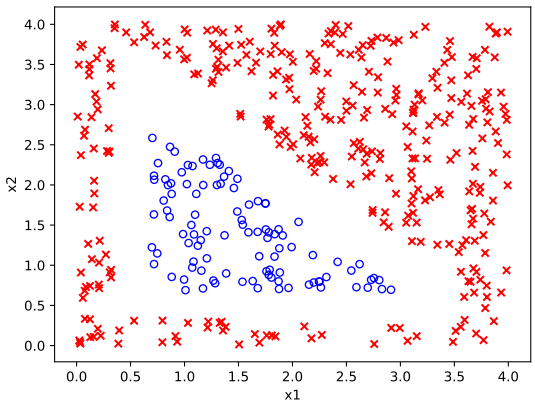


Figure 4: Plot of dataset X.

The examples in class 1 are marked as as "×" and examples in class 0 are marked as "○". We want to perform binary classification using a simple neural network with the architecture shown in Figure 5:

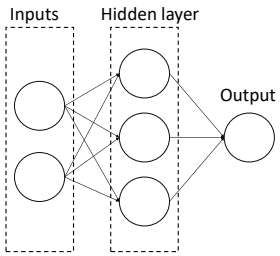


Figure 5: Architecture for our simple neural network.

Denote the two features x_1 and x_2, the three neurons in the hidden layer h_1, h_2, and h_3, and the output neuron as o. Let the weight from x_i to h_j be $w_{i,j}^{[1]}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w_j^{[2]}$. Finally, denote the intercept weight for h_j as $w_{0,j}^{[1]}$, and the intercept weight for o as $w_0^{[2]}$. For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

$$l = \frac{1}{n} \sum_{i=1}^{n} \left(o^{(i)} - y^{(i)} \right)^2,$$

where $o^{(i)}$ is the result of the output neuron for example i.

(a) [5 points] Suppose we use the sigmoid function as the activation function for h_1, h_2, h_3 and o. What is the gradient descent update to $w_{1,2}^{[1]}$, assuming we use a learning rate of α? Your answer should be written in terms of $x^{(i)}$, $o^{(i)}$, $y^{(i)}$, and the weights.

Answer: Let g denote the sigmoid function $g(x) = \frac{1}{1+e^{-x}}$, and $h_j^{(i)}$ denote the output of hidden neuron h_j for sample i. For shorthand, we denote $\vec{w_j}^T x^{(i)} = w_{1,j}^{[1]} x_1^{(i)} + w_{2,j}^{[1]} x_2^{(i)} + w_{0,j}^{[1]}$ for $j \in \{1,2,3\}$, and $\vec{w_o}^T h^{(i)} = w_1^{[2]} h_1^{(i)} + w_2^{[2]} h_2^{(i)} + w_3^{[2]} h_3^{(i)} + w_0^{[2]}$. Using chain rule, we have

$$\frac{\partial l}{\partial w_{1,2}^{[1]}} = \frac{2}{n} \sum_{i=1}^{n} (o^{(i)} - y^{(i)}) \frac{\partial (o^{(i)})}{\partial w_{1,2}^{[1]}}$$

$$\frac{\partial (o^{(i)})}{\partial w_{1,2}^{[1]}} = g(\vec{w_o}^T h^{(i)})(1 - g(\vec{w_o}^T h^{(i)})) w_2^{[2]} \frac{\partial (h_2^{(i)})}{\partial w_{1,2}^{[1]}}$$

$$\frac{\partial (h_2^{(i)})}{\partial w_{1,2}^{[1]}} = g(\vec{w_2}^T x^{(i)})(1 - g(\vec{w_2}^T x^{(i)})) x_1^{(i)}$$

Combining everything, we have

$$\frac{\partial l}{\partial w_{1,2}^{[1]}} = \frac{2}{n} \sum_{i=1}^{n} (o^{(i)} - y^{(i)}) g(\vec{w_o}^T h^{(i)})(1 - g(\vec{w_o}^T h^{(i)})) w_2^{[2]} g(\vec{w_2}^T x^{(i)})(1 - g(\vec{w_2}^T x^{(i)})) x_1^{(i)}$$

and the update rule is

$$w_{1,2}^{[1]} := w_{1,2}^{[1]} - \alpha \frac{\partial l}{\partial w_{1,2}^{[1]}}$$

(b) [10 points] Now, suppose instead of using the sigmoid function for the activation function for h_1, h_2, h_3 and o, we instead used the step function $f(x)$, defined as

$$f(x) = \begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases}$$

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_step_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `step_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

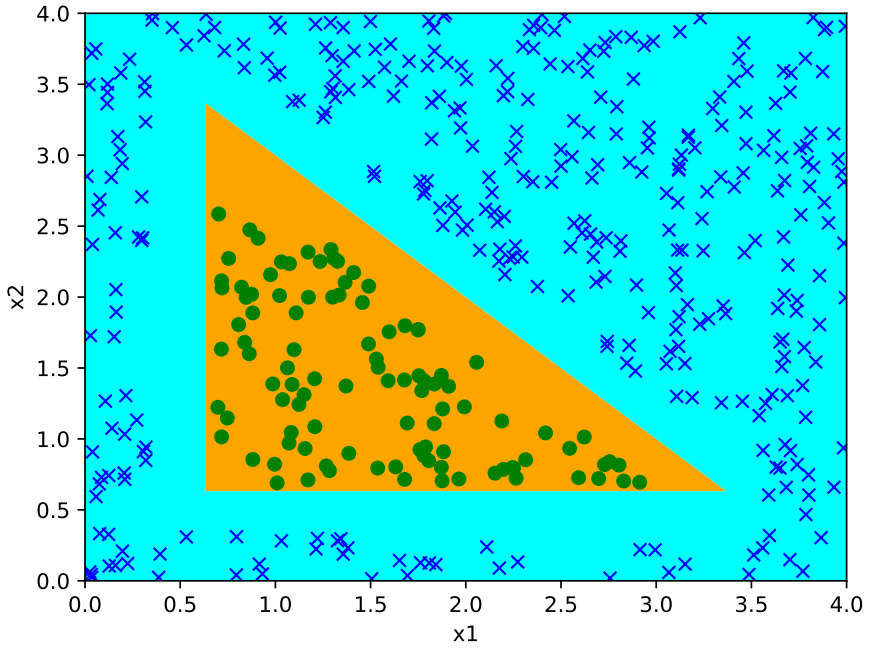
Hint 1: There are three sides to a triangle, and there are three neurons in the hidden layer.

Hint 2: A solution can be found where all weight and bias parameters take values only in $\{-1, -0.5, 0, 1, 3, 4\}$. You are free to come up with other solutions as well.

Answer: The key is to notice that the dataset is classifiable with a triangle. Each hidden neuron corresponds to if the data lies to the left/right of a line, and the output neuron determines if the data point is in the triangle. The three lines that form the boundary could be $x_1 = 0.5, x_2 = 0.5, x_1 + x_2 = 4$. One set of weights that works by looking at the plot would be

The corresponding plot for that solution is:

Neuron	Bias	Value	Weight	Value	Weight	Value	Weight	Value
h_1	$w_{0,1}^{[1]}$	-0.5	$w_{1,1}^{[1]}$	1	$w_{2,1}^{[1]}$	0		
h_2	$w_{0,2}^{[1]}$	-0.5	$w_{1,2}^{[1]}$	0	$w_{2,2}^{[1]}$	1		
h_3	$w_{0,3}^{[1]}$	4	$w_{1,3}^{[1]}$	-1	$w_{2,3}^{[1]}$	-1		
o	$w_0^{[2]}$	3	$w_1^{[2]}$	-1	$w_2^{[2]}$	-1	$w_3^{[2]}$	-1



(c) [10 points] Let the activation functions for h_1, h_2, h_3 be the linear function $f(x) = x$ and the activation function for o be the same step function as before.

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_linear_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `linear_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

Hint: The hints from the previous sub-question might or might not apply.

Answer: No there is no set of weights that could make the loss 0 (i.e, perfect classification). This is because composite linear functions are still linear, so the step function is essentially a step function of a linear function of the features. This would require the dataset to be linearly separable; from the plot it is evidently not.