CS 229, Winter 2025
Problem Set #3

Due Wednesday, February 19 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/70557/discussion`. (3) This quarter, Winter 2025, all homework assignments must be submitted individually. However, students may work in groups. If you do so, make sure both names are attached to the submission. 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) To account for late days, the due date is Wednesday, February 19 at 11:59 pm. If you submit after Wednesday, February 19 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Wednesday, February 19 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX, and we will award one bonus point for typeset submissions. To edit on Overleaf, upload the `pset3.zip` folder as new project. In the project view, click on the "Menu" button located on the top left corner. In the "Settings" subsection, you will need to set the "Main Document" to be "tex/ps3.tex". Also, make sure your **cursor is selected on ps3.tex** before your press the **Recompile button**. Type your response in the `-sol.tex` file version for every problem. Inside `text/ps3.tex` file, you will need to change `\def\solutions{0}` to `\def\solutions{1}` and then re-compile. To export the PDF file, click the downward arrow next to the **Recompile button**.

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.

1. **[20 points] AdaBoost Performance**

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.

(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

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

(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.

2. **[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.

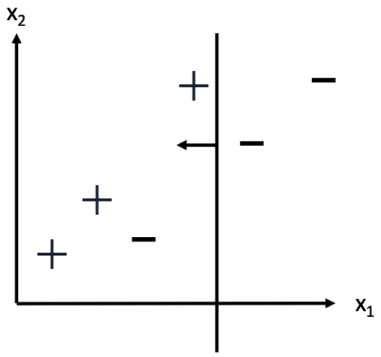

Figure 1: 2-dimensional labeled data, where '+' corresponds to class $y = +1$ and '-' corresponds to class $y = -1$. The decision boundary for the first decision stump is shown. The arrow points in the positive direction from this decision boundary.

(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).

(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.

(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.)

3. **[30 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. To greedily select our split, we choose the feature and threshold that maximize the gain function:

$$j, \theta = \arg\max_{j,\theta} G(j, \theta).$$

In other words, we find the feature j and the split value θ such that we maximize our gain function $G(j, \theta)$. Provide a list of all splits and the classification error reduction at each split.

(b) [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.

(c) [5 points] Multivariate decision trees have practical advantages and disadvantages. List two advantages and two disadvantages multivariate decision trees have compared to univariate decision trees.

(d) [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. 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.

4. **[25 points] Transformers in PyTorch** In lecture, we were introduced to the concepts underlying language models, like transformers. In this problem, you will use PyTorch to implement a simplified transformer model that can generate Shakespeare-like text. The transformer architecture relies on the **self-attention mechanism**, which allows the model to focus on relevant parts of the input sequence when making predictions. Self-attention compares a query (current token) to keys (all tokens in the sequence) and generates attention scores that determine how much weight each token should have in the final representation. Each vector is computed from the input sequence X using linear projections:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

Here, W_Q, W_K, W_V are learned weight matrices. Once these are computed, attention scores are derived by scaling the dot product of Q and K:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

This equation calculates how much focus (attention) each token in the sequence should give to other tokens. The **softmax** function ensures that the attention weights sum to 1. In **multi-head attention**, this process is repeated across multiple "heads", allowing the model to learn different attention patterns simultaneously. The outputs from each head are then concatenated and projected through another linear layer. You will implement this attention mechanism by following the mathematical formulas above.

Rather than implementing from scratch, you will build on top of code inspired by **this excellent teaching demo by Andrej Karpathy**. Specifically, you will complete the skeleton code for the attention mechanism, transformer model, and encoder in **this Colab notebook**. The data loading, training, and evaluation functions are already implemented for you. You will not need to change any of this code. Do NOT modify the hyperparameters. Fill out the missing code by following the instructions in the comments, and reference PyTorch documentation online for syntax. If implemented correctly, the model will train in approximately 20 to 30 minutes. (Note: The tqdm estimate can be unreliable, so instead confirm that the training is on average at least 5 iterations per second).

You will be training and evaluating your model with the Tiny Shakespeare dataset, a text corpus frequently used in language modeling tasks. The dataset contains a collection of Shakespeare's plays in plain text form. Instructions to download the dataset are provided in the colab.

Report the final validation loss, which we expect to be below 1.7, and submit the generated output for the provided input sample (you may need to run it a few times to get a response that you like, as it is not deterministic). Also download the pdf for the .ipynb file and attach it as a part of your writeup. Connect to a T4 GPU in Colab to ensure efficient training. Remember to make a copy of the colab before editing cells, otherwise you will not be able to save your work!