Homework 3: Decision Trees and Random Forests

Principles of Machine Learning (Wentworth Institute of Technology)

Fall 2024 Semester

Decision tree models are easy to understand, but prone to overfitting; random forests are the exact opposite. Addressing the root of these problems involves finding the "best" trade-off of hyperparameter choices (get it, because tree data structures have a root node?). Please respond to each item below, which can be a question to answer and/or task to perform. *You must show your work in order to receive full credit for a correct response, and your work will determine partial credit for incorrect responses.* If you choose to work with others in the class, please remember that everyone must turn in their own version of the responses to the items in this assignment. You must mention with whom you collaborated as well (and for that matter, please remember to include your name on the submission).

When working on problems in the wild, be it for work or pleasure, the world will neither tell you the specific problem nor what to do to solve this problem. Customers and/or clients come to you because they do not know how to do it themselves, which means they can only give a lot of context attempting to describe what they want you to achieve. The world itself has lots of things you can observe and loves to throw life challenges at us. Some of the information from these sources will be useful, and the rest is either a red herring or irrelevant. These problems are phrased in a similar way, and your solutions will need you to identify both the problem(s) and which information is relevant to solve them.

1. Continuing to grow our suite of ML algorithms, we will implement a basic decision tree learner that assumes discrete features as inputs and discrete class labels as outputs. This means our decision tree models will only serve as classifiers because regression has continuous outputs. Continuing with your matrix-based programming language (Matlab, Octave, Julia, R, etc.) or package (Python's numpy and scipy, C++'s Boost, etc.) of choice, grab the general classes you prepared from Homework 2 as your starting point. It is highly encouraged you TEST that your code works as intended after each step so that any problems in the next step are easier to diagnose. *Good programming practices you should use in this assignment are listed in the back as an appendix.*

   (a) ***(15 points)*** Before we start to implement the `train` and `predict` functions, we will need a data structure for our decision tree nodes. Traditionally, a tree node only stores some data item and a list/array of pointers to other tree nodes (the parent's children). We need something a bit more sophisticated for decision trees because there is additional context to track. Let's break it down and think about what we need. This is the thought process you should apply when given programming specifications to determine how to implement effecient and effective code that solves the problem, from considerations of tasks to questions to ask to what to actually do:

      i. The first thing is the data that the node stores. There are two different types of nodes with their own purpose: ***query nodes*** have a parameter that stores the feature in the sub-dataset to inspect, and ***decision nodes*** have a class label that returns a prediction.

         TO-DO: Keep track of the node's type via a boolean/binary variable in your decision tree node class code. Answer these questions: How can we keep track of the node's purpose in the code with this boolean/binary variable? Because decision nodes are always the leaves of the tree, how could we check the type of node without a specific variable?

      ii. Regardless of the node's purpose, it stores a single integer value that represents the feature (query node) or label (decision node).

TO-DO: Keep track of the stored data via a numerical variable that your language of choice supports type-wise. Answer this question: Why is it sufficient to only have one variable store either value rather than have two variables for the feature and label, and how does this help save memory?

iii. The second thing is organizing the child node pointers. A query node will have a different child node per response (value of the inspected feature), which means a list or array of pointers is not enough to make sure the correct response pairs with the correct child node pointer.

TO-DO: Keep track of the child node pointers via a hash map/dictionary or related data structure that your language of choice supports. Answer these questions: What do the key and value for this hash map/dictionary represent with respect to the decision tree node's children? How does the hash map/dictionary data structure save runtime during decision tree prediction when we have the response from the input features (passed in as an argument)?

(b) ***(50 points)*** Now that we have a decision tree node class in addition to a general class or data structure that should describe the properties and capabilities of any machine learning algorithm, we can implement a more specific (sub)class or data structure for decision trees. Its training and prediction functions should follow these specifications:

TO-DO: Implement a decision tree class as specified below in the other TO-DO segments. When following the good programming practices, remember that you should include comments that document what parts of the code do. Because the TO-DO segments only describe what code to write, use the context provided before the TO-DO segments to determine what comments to provide that explain what the code does.

Entropy Function

i. The input to any of the error functions is a vector of labels $\overrightarrow{y}$ corresponding to some training set S.

TO-DO: Setup the entropy function within the decision tree class. For now, it is an empty function with only a name, the parameter list, and any additional syntax your programming language of choice requires to write a function (examples are curly braces, colons, keywords like `pass` or `return`, etc.).

ii. We compute the entropy using the proportions of $\overrightarrow{y}$ assigned to each label p_i in:

$$H(S) = -\sum_{i=1}^{\#\text{labels}} p_i \log_2(p_i).$$

TO-DO: Define a hash map/dictionary variable whose keys are classification labels (numbers or strings, depending on your implementation, but numbers will be easier and need less memory) and values are numbers representing the proportion of $\overrightarrow{y}$ that contains the corresponding label. After defining this hash map/dictionary variable, implement a loop over all the entries in $\overrightarrow{y}$ that uses the current entry as a key to the hash map/dictionary and increments it as a counter. After this loop, define a numeric variable h and set it to 0. Then, implement a new loop over the keys of the hash map/dictionary that divides the mapped value by the number of entries in $\overrightarrow{y}$ and then add $-1 \cdot p \cdot \log_2(p)$ to h via the `+=` operator where p is the computed quotient. After this loop, return h.

Gini Impurity Function

i. The input to any of the error functions is a vector of labels $\overrightarrow{y}$ corresponding to some training set S.

TO-DO: Setup the Gini impurity function within the decision tree class. For now, it is an empty function with only a name, the parameter list, and any additional syntax your programming language of choice requires to write a function (examples are curly braces, colons, keywords like `pass` or `return`, etc.).

ii. We compute the Gini impurity using the proportions of $\overrightarrow{y}$ assigned to each label p_i in:

$$I_G(S) = \sum_{i=1}^{\#\text{labels}} \left(1 - p_i^2\right).$$

TO-DO: Define a hash map/dictionary variable whose keys are classification labels (numbers or strings, depending on your implementation, but numbers will be easier and need less memory) and values are numbers representing the proportion of $\overrightarrow{y}$ that contains the corresponding label. After defining this hash map/dictionary variable, implement a loop over all the entries in $\overrightarrow{y}$ that uses the current entry as a key to the hash map/dictionary and increments it as a counter. After this loop, define a numeric variable g and set it to 0. Then, implement a new loop over the keys of the hash map/dictionary that divides the mapped value by the number of entries in $\overrightarrow{y}$ and then add $(1 - (p \cdot p))$ to g via the `+=` operator where p is the computed quotient. After this loop, return g.

Train Function

i. The inputs to the train function should always be the hyperparameters (as an array or hash map based on your implementation in Homework 2) and the training data as inputs X (a matrix) and ground truth outputs $\overrightarrow{y}$ (a vector, which is effectively a matrix with exactly one column).

TO-DO: Implement the decision tree class as a subclass of the generic ML algorithm class, and setup the train function that the subclass will overwrite. For now, this overwrite is an empty function with only a name, the parameter list, and any additional syntax your programming language of choice requires to write a function (examples are curly braces, colons, keywords like `pass` or `return`, etc.).

ii. Let one of the hyperparameters the user sets be a choice of error function that determines the best feature to inspect at a decision tree query node. Check this hyperparameter to determine which specific helper function to call for the actual training process.

TO-DO: Implement an if-then-else check within the train function (and its recursive helper function described in the next part) as needed that checks for the value assigned to a hyperparameter defining the choice of error function: entropy or Gini impurity. Notice that "as needed" means this is not simply done one time at the start of the function for three separate control flows. Instead, when the error function choice matters, we will use this split in control flow. Answer this question: How does changing the control flow only when the choice of a hyperparameter matters reduce the amount of code we write if the rest the control flow is identical regardless of hyperparameter choice?

iii. Because tree data structures are recursive in nature, we will implement a helper function that performs the recursive function. In this case, it will generate a decision tree node for a given dataset, and the dataset will change per recursive call as we partition the points by those that have specific values assigned to chosen features (as answers to the queries).

TO-DO: Implement a function in the decision tree class that will perform the recursive training process. It should have the same parameters as the train function for consistency of information needed to perform the algorithm. It will have one additional parameter input that lists the features already queried (we will see why later[1]). For now, this recursive function is an empty function with only a name, the parameter list, and any additional

[1]If it makes anyone feel better, Rick Freedman (the instructor who wrote this assignment) had to add this parameter in later due to not considering this case until it was needed. Writing code can involve going back to change stuff if you failed to plan ahead, and we all miss important details no matter how hard we try to think of everything in advance.

syntax your programming language of choice requires to write a function (examples are curly braces, colons, keywords like `pass` or `return`, etc.); if you declare a return type in your language of choice, then this recursive function returns a pointer to a decision tree node. In the train function, set the learned decision tree[2] to the output of the recursive function, and pass in all the same arguments that the train function received. Pass in an empty list for the features already queried because no features have been considered yet. Answer this question: How does this recursive function build a tree when it only returns a pointer to one decision tree node?

Helper Function: Recursive Decision Tree Node Generator

i. The input to the recursive decision tree node generator function should always be the hyperparameters and the training data as inputs X and ground truth outputs $\overrightarrow{y}$. (The train function stated this already, but we repeat it here because the details are relevant to this function's implementation).

   TO-DO: Sanity check: did you setup this helper function correctly?

ii. As a recursive function, we need a base case that simply terminates when the function inputs satisfy certain conditions. Generally, regression hyperparameters for the decision tree will determine when to stop, and they need to be implemented throughout the function once the correct details become computed. Without those details yet, the simplest stopping criteria is having no more features to query.

   TO-DO: Check whether the list of queried features is the same size as the number of columns in X. If this is the case, then return an instance of a decision tree node whose type is a decision node and label is the majority label found in $\overrightarrow{y}$. To compute this majority label, define a hash map/dictionary whose key is a label and value is a number corresponding to the frequency of a label (starting with 0 counts). Then, iterate over the entries in $\overrightarrow{y}$ in a loop; this loop will increment the count obtained from the map using the current entry as the key. Make sure to place the code computing the majority label *before* initializing the decision tree node.

iii. When the base case does not apply, then simply begin the recursive step without needing an else statement because the control flow does not change further based on the stopping conditions. In order to determine the impact of a chosen feature for partitioning the training dataset, we need to compute the baseline 'error' with the entire training dataset available before splitting.

   TO-DO: Define a local numeric variable that stores the error over the passed-in $\overrightarrow{y}$. Because the error function is a hyperparameter, use an if-then-else check to assign the value to the variable. Keep in mind that scoping requires the variable's declaration *before* the if-then-else block so that the variable may be used after the block.

iv. In order to determine the best feature over which to split the data, we need to try each possible feature and compute the change in error. Doing this efficiently is more complicated than it sounds, and we will instead do something simple that wastes a some computation time for everyone's sanity. As we iterate over features, we will keep track of the dataset row indeces that evaluate to each value of that feature (splitting with respect to the current feature), build new label vectors $\overrightarrow{y_{f,v}}$ where f is the feature and v is the value, compute the error for each $\overrightarrow{y_{f,v}}$ as an iteration over the possible v since f is fixed in the outer loop, and accumulate the change in error. Despite this looking like a few steps, it is many more in code. We break down the bigger steps into smaller ones, and it will be good practice to think about how each high-level step became all these smaller steps because you will rarely have someone breaking it down for you.

[2]If your language of choice has duck typing, then you can simply use the same variable that normally stores the learned parameters. If your language of choice has a strict type checking system, then you will need to create a new variable in the decision tree class that stores a pointer to the root node of the learned decision tree.

TO-DO: (Initial Setup) Define a hash map/dictionary variable that stores integers (representing column indeces of X) as keys, and it stores values that are hash maps/dictionaries that store values corresponding to some feature (usually numbers since they are entries in a matrix) as keys and a list of integers (representing the row indeces of X) as values—yes, this is a nested hash map/dictionary. Define another hash map/dictionary variable that stores integers (representing the column indeces of X again) as keys, but this one stores values that are numeric values for the accumulated change in error. As in the previous step, scoping requires these variables' definitions to appear before other blocks such as loops if we plan to use them after that block of code executes.

TO-DO: (Splitting the Dataset per Feature) After these variable declarations, implement a loop that iterates over all the features (column indeces) c of X; this loop will immediately continue if the current feature happens to be in the list of already-queried features[3]. Within this outer loop, implement an inner loop that iterates over all the samples (row indeces) r of X. To keep track of splitting the dataset, the nested hash map/dictionary will take key c for the outer map and key $X_{r,c}$ for the inner map, and that value's list will append r to it.

TO-DO: (Calculating the Change in Error) After this inner loop finishes, still within the outer loop, implement a new inner loop over the keys of the inner hash map/dictionary v (you can access the current inner hash map via the key c with the outer hash map). In this new inner loop, define a list variable that will create $\overrightarrow{y_{f,v}}$ where $f = c$ in this case, and then create yet another inner loop that iterates over the list of row indeces r' (you can access this list using the keys c for the outer hash map and v for the inner hash map). Within this inner-most loop, append the r' entry of $\overrightarrow{y}$ to $\overrightarrow{y_{f,v}}$ to create the split dataset's labels. After this inner-most loop, still within the second inner loop you implemented, compute the error (do not forget the if-then-else block for checking hyperparameters) over $\overrightarrow{y_{f,v}}$ and its difference from the 'baseline' error you stored at the start of this function; be sure to store it via accumulation[4] into the other hash map/dictionary we have not touched yet since defining it (the key will be c). After the second inner loop, we need to divide the accumulated value by the total number of added values to get the average (this is the number of keys in the inner hash map/dictionary that you can access with key c to the outer hash map). Then, the outer loop is also complete.

TO-DO: (Sanity-Check) Because a lot happens, it is not a bad idea to try a small dataset (as in, 3 or 4 samples with 2 features and 2 labels) by hand and make sure everything works as intended. Better yet, inserting a few print statements[5] will enable you to compare to your by-hand calculations to make sure things go as expected.

v. With all the changes in error computed per feature, we can now select the best query for the current decision tree node. This not only creates the node that we will return, but also sets up the recursive function calls that continue to partition the remaining data subsets.

TO-DO: (Creating the parent node) Define a numeric variable that stores the best error change observed so far, and define another numeric variable that stores the column index corresponding to that best error change observed so far. Because some of the error functions prefer to maximize while others prefer to minimize, selecting the best error change will vary per hyperparameter choice; set up the if-then-else statement and identify the maximum/minimum error change within each case. To find the maximum/minimum,

[3] This is when Rick Freedman realized this information was missing from the input parameters.

[4] Accumulation is available in most languages via the `+=` operator, and this means you need to initialize the sum variable to 0 before beginning the loop that accumulates the values.

[5] Do not forget to put them in conditional checks with a debug toggle as suggested in the good programming practices!

set the variable storing the best error change observed so far to a very large positive (for the minimum) or very small negative (for the maximum) number, and then iterate in a loop over the hash map/dictionary computed above with values corresponding to the computed error changes per feature. Within this loop, when a value in the map is appropriately greater (for maximum) or lesser (for minimum) than the best error observed so far, set the variable for the best seen error so far to this map's value and the variable for the corresponding feature to map's key. After the loop (and if-then-else) completes, initialize a new instance of a decision tree node whose type is a query node and feature (parameter) is the same as the variable storing the column index corresponding to the best error change observed.

TO-DO: (Creating the child nodes) Thanks to keeping the nested hash map/dictionary variable in scope, we have the list of all values that the queried feature can be (use the queried feature's column index as the key to the outer map). Iterating in a loop over these possible feature values v' as the keys of the inner map, we first generate the partitioned dataset inputs $X_{f,v'}$ and labels $\overrightarrow{y_{f,v'}}$ using a similar method to the one used for Calculating the Change in Error above: define matrix variables for each of $X_{f,v'}$ and $\overrightarrow{y_{f,v'}}$, and then implement an inner loop that iterates over the list of row indeces (you can get this with key v' to the inner map) to append the rows of X and $\overrightarrow{y}$ into $X_{f,v'}$ and $\overrightarrow{y_{f,v'}}$, respectively. After this loop, still within the outer loop, the decision tree node will add the following key-value pair to its hash map/dictionary of child nodes: the key is v', and the value is the returned pointer from a recursive call to this function that passes in the same hyperparameters, $X_{f,v'}$, $\overrightarrow{y_{f,v'}}$, and list of queried features that appends the newest queried feature (assigned to the decision tree node).

TO-DO: (Final Wrap-Up) After the outer loop completes, return the pointer to the initialized decision tree node.

Predict Function

i. The input to the predict function should always be the inputs X.

   TO-DO: Setup the predict function that the subclass will overwrite. For now, this overwrite is an empty function with only a name, the parameter list, and any additional syntax your programming language of choice requires to write a function (examples are curly braces, colons, keywords like `pass` or `return`, etc.).

ii. An important sanity check, if you did not implement this for the generalized template, is that the parameters have been learned already. If not, be sure to inform the user and terminate the program gracefully (throw an error/exception or trigger an assertion violation).

   TO-DO: At the start of the predict function, check whether the class instance's value for the parameters is set to a non-default value (typically null). If it is the default value, then throw an error/exception or trigger an assertion violation that provides a message about needing to train the model first.

iii. Without any formal matrix operations to perform in decision tree classification, we must predict each row of X independently. For a single input row $\overrightarrow{x}$, navigate the decision tree from the root node until reaching a leaf node based on matching the queried features' values to the correct children. Include all the predicted labels in a vector $\overrightarrow{z}$ and return it.

   TO-DO: After the check of the parameters, define a vector or matrix variable that will represent $\overrightarrow{z}$. Scoping requires this variable's definitions to appear before other blocks such as loops if we plan to use them after that block of code executes. Implement a loop over the rows $\overrightarrow{x}$ of X, and define a decision tree node pointer variable at the start of this loop assigned to the root of the learned decision tree. Within this loop after defining this variable, implement another loop that repeats as long as the decision tree node pointer variable just defined is a query node[6]. In this inner loop, access the decision tree node pointer variable you defined's query feature/parameter, which should be a column index, and store the value of $\overrightarrow{x}$ at that column index into a variable. Still within the inner loop, reassign the decision tree node pointer variable to its child node pointer whose key is the stored value from $\overrightarrow{x}$ (if this key does not exist, then consider throwing an error explaining that the input contains new data not seen during training). After this loop, still within the outer loop, append the label stored in the decision-type decision tree node pointer variable to $\overrightarrow{z}$. After the outer loop, return $\overrightarrow{z}$. Answer this question: If we had access to a GPU, then could we perform each prediction in parallel; why or why not?

2. ***25 points*** Using the decision tree you programmed, let us implement a random forest classifier. The method we use here can also create bagging ensembles for other ML algorithms. *Good programming practices you should use in this assignment are listed in the back as an appendix.*

   TO-DO: Implement a random forest class as specified below in the other TO-DO segments. When following the good programming practices, remember that you should include comments that document what parts of the code do. Because the TO-DO segments only describe what code to write, use the context provided before the TO-DO segments to determine what comments to provide that explain what the code does.

   Additional Properties

   (a) A bagging ensemble has many ML models, usually of the same algorithm. They will all be trained at once as well as used for prediction, which means we need to keep track of them.

   TO-DO: Implement a list variable that will store decision tree object pointers.

   Train Function

   (a) The inputs to the train function should always be the hyperparameters (as an array or hash map based on your implementation in Homework 2) and the training data as inputs X (a matrix) and ground truth outputs $\overrightarrow{y}$ (a vector, which is effectively a matrix with exactly one column). The hyperparameters will be the same as the decision tree class's hyperparameters, but there will be one additional hyperparameter for the number of decision trees to learn in the forest.

   TO-DO: Implement the random forest class as a subclass of the generic ML algorithm class, and setup the train function that the subclass will overwrite. For now, this overwrite is an empty function with only a name, the parameter list, and any additional syntax your programming language of choice requires to write a function (examples are curly braces, colons, keywords like `pass` or `return`, etc.).

   (b) Each of the decision trees in the random forest classifier will receive a subset of the input dataset for training. There are various ways to obtain this subset, but we will simply shuffle the rows of the dataset (both sample and label) and then train with only the top 80% of the rows from the shuffled dataset. Although there are many algorithms for shuffling, we will keep it simple and use lots of row swaps[7].

[6]If you want to be extra careful, then check that the the decision tree node pointer variable is not set to null before checking its type property. Depending on the language of choice, a null value could crash the program's execution or enter undefined behavior states where it reads whatever is in memory at that address regardless of whether it makes sense.

[7]In mathematics, the permutation group studies properties of swapping two things at a time in an ordered set in order to get various reorderings of the set.

TO-DO: Implement a loop that repeats as many times as the hyperparameter for the number of decision trees specifies. Inside this loop, implement another loop that repeats as many times as ten times the number of rows in X. Within the inner loop, store two random integers that are within the bounds of X's rows and swap those two rows of *both* X and $\overrightarrow{y}$. Swapping simply stores one of the rows in a temporary matrix variable, replaces the copied row in the matrix with the other chosen row, and then stores the temporary matrix variable's row in the other chosen row. After this inner loop completes, still within the outer loop, define a matrix variable X' with the first 80% of X's rows as well as a matrix/vector variable $\overrightarrow{y}'$ with the first 80% of $\overrightarrow{y}$'s rows. After defining these variables; initialize a decision tree class instance; call its train function with arguments X', $\overrightarrow{y}'$, and the random forest's hyperparameters; and append the learned decision tree to the random foresst class instance's list of decision tree pointers.

Predict Function

(a) The input to the predict function should always be the inputs X.

TO-DO: Setup the predict function that the subclass will overwrite. For now, this overwrite is an empty function with only a name, the parameter list, and any additional syntax your programming language of choice requires to write a function (examples are curly braces, colons, keywords like `pass` or `return`, etc.).

(b) The random forest simply consults each of its decision trees with the inputs and selects the majority classification label per input as its own classification.

TO-DO: Define a matrix variable that has as many rows as X and columns as the number of stored decision tree pointers, and define a vector or matrix variable $\overrightarrow{z}$ that has one column and as many rows as X. After defining the variable, implement a loop that iterates over all the learned decision tree pointers. Inside this loop, call the current decision tree's predict function with argument X and store the predicted labels in the next available column of the defined matrix variable. After this loop completes, define a hash map/dictionary variable with keys that are labels (numeric or string depending on your implementation) and values that are integers representing counts for that label. Following the variable definition, implement a new loop that iterates over each row of the matrix defined at the start of the predict function. Inside this loop, implement another loop that iterates over the columns of the current row—for the current entry of the matrix at that row and column as key, increment the mapped hash map/dictionary value by 1. After this inner loop completes, still inside the outer loop, define two variables for the label with the greatest count and that count and then implement a loop over the keys in the hash map/dictionary that stores the key and value with the greatest value into those variables. After the inner loop completes, still inside the outer loop, store the variable with the label of the greatest count into $\overrightarrow{z}$ at the current row index. After the outer loop completes, return $\overrightarrow{z}$.

Before we move on, let us consider how we could implement our random forest class more generally as a bag for any ML algorithm.

(a) Due to the superclass we defined for all the ML algorithms, consider whether we can create a more general bagging class.

TO-DO: Answer the following questions: Would the random forest class still compile/interpret and run as expect if we changed the type the list stores from decision tree pointers to ML algorithm pointers; why or why not? If we added a hyperparameter selecting the specific ML algorithm, then what would change in the train function to get the desired classification models? Why would the test function not have to change despite the change in algorithm?

3. ***10 points*** Our naïve friend Flubsy noticed that we did not include any decision tree regression hyperparameters so that our decision trees will split until all the features are queried or some dataset partition has only samples corresponding to one class label. However, Flubsy does not quite understand

the overfitting risks with this setup nor realize that the random forest can counteract this overfitting risk. Please be kind to Flubsy, but let us help them understand these overfitting risks and mitigations.

(a) Starting with the decision tree, explain why the lack of regression hyperparameters lead to a greater risk of overfitting.

    TO-DO: Answer the following questions: How likely is a decision node within the decision tree to correctly predict a label for a random input in our dataset if its training data subset has only one class label? If all the features are queries before making a decision (regardless of the order in which the decision tree queried them), is it possible to gain any more information from a future query; why or why not? How do these two questions' responses contribute to overfitting on the training dataset?

(b) Continuing with the random forest, explain why many decision trees that might overfit on a subset of the dataset are still at a lower risk of overfitting.

    TO-DO: Answer the following questions: If each decision tree in the random forest trains on $q\%$ of the training dataset samples, then what is the greatest and least amount of training dataset overlap that two decision trees in the random forest could have? If two overfit decision trees each share $r\%$ of their training datasets, then what is the probability that exactly one, both, or neither will correctly classify some random input found in the union of their training datasets (combining both datasets into one larger dataset) when sampled uniformly (all rows in the dataset are equally likely to be chosen)? Which of the probabilites in these previous question's response will increase or decrease as the number of decision trees considered increases; why?

Appendix: Good Programming Practices

When writing code, you must follow these good programming practices:

- When turning in code, keep in mind that Rick Freedman (the instructor who wrote this assignment) has a golden rule: *Good code with bad comments is bad code, PERIOD.* Make sure comments are clear and describe what will happen per significant moment in the code. If the code does something trivial over several lines, explain the idea just before the first of those lines. If the code does something complicated on a single line, then explain what that line does. Use your judgement, but also remember you might need to reuse and understand this code later—your future self will thank you if you document it better now while the content is fresh in your mind!

- Documenting functions you create (purpose, inputs, outputs, and possible error throws) is also helpful. There is a reason most APIs that you read include these details as part of the manual.

- Avoid "magic numbers" when encoding choices like the linear regression approaches. It is tempting to use an integer such as 0 meaning MLE, etc., and the idea works well if done carefully. If you simply write 0 everywhere in the code, then how will you know which 0's mean the MLE choice, which 0's mean the number 0, and which 0's mean some other choice between different options? Instead, set a variable (ideally constant, if your language supports them, to avoid accidentally changing the assignment) to the number, and then use that variable everywhere in the code. It is easier to understand (compare `approach = 0` to `approach = MLE_LINREG`) and easier to update later if something changes in your code. New option to add in that slot, which means you have to change the previous option's assignment? Just change the variable to another number and do not worry because the variable effectively replaced it everywhere else in the code. Found a bug in the code that requires you to change the option you chose somewhere in the code? Now you know which 0's are for the option and which 0's are not so you do not accidentally change the wrong things (creating more errors that were not broken before).

- Be careful naming the output file to which you write because the program could overwrite another file that your program previously wrote by mistake! It is usually good practice to add a date+timestamp in the filename so that the output filenames are unique (unless you call the function really fast multiple times in a row).

- Store intermediate computations that you are able to reuse rather than compute everything from scratch in each line of code. Although you can compute things multiple times (such as X^T), it costs *real-world time* in an era where computer memory is much cheaper on your average computer (satellites and embedded systems are a different story, as is time spent retrieving cached data… these are all beyond the scope of this assignment). Save the time by storing the intermediate computation steps, and then call the variable storing it in every case that you use it afterwards. Storing the intermediate computation also lessens the chance of making a typo when recomputing it next time. This is similar to the benefits of avoiding "magic numbers."

- Break out sequences of computations that you will reuse frequently into functions rather than rewrite those lines of code from scratch each time you need it. Similar to storing intermediate computations, this lessens the chance of a typo when rewriting all that code each time. Furthermore, the function makes your code easier to read if it is descriptive—consider a function name replacing a lot of redundant code like an abridged comment placed above its block of code.

- Make debugging accessible even after your program is finished. You never know when you will need to look at the code or modify it again, and you do not want to have to redo your entire debugging setup. In particular, if you have any print statements in your code to explore the control flow ("entering function F," "exiting loop L in function F," etc.) or inspect variables ("the value of variable V is currently:," "compare variable V's value to expected E:," etc.), then those should stay in your code. However, to avoid those print statements cluttering the screen, put them all inside conditional blocks (if statements) where the condition to check is for a global constant variable such as DEBUG. Make sure to define this global constant variable DEBUG at the start of your code, and then set it appropriately before your compile and run your code. If DEBUG is boolean, then use TRUE when you want the debug information to print and FALSE when you do not want the debug information on the screen. If DEBUG is an integer, then define a hierarchy of verbosity/priorities where 0 means 'print nothing,' 1 means 'print the basic information,' 2 means 'print some additional details,' etc. (make sure to define each verbosity/priority level with its own constants to avoid "magic numbers"). You can also create more elaborate debugging variables in more complex code, such as separate DEBUG_X constant variables for different parts of the program X. For an assignment like this, one boolean variable should be sufficient.

Rick has lots of other rules and pet peeves about writing good code, but you can talk about those at office hours sometime if you are interested. They do not likely apply in this assignment.