Challenge

Classification Trees

Background

Computer scientists often refer to the principle of *divide and conquer*: by recursively splitting a problem into smaller problems that are easier to solve, we can solve a large difficult problem by putting together solutions to many small easy problems. Common applications are tasks like sorting, where mergesort recursively splits arrays in half and sorts each half, or data storage and retrieval, like k-d trees and R-trees for efficiently storing and querying high-dimensional data.

Divide and conquer can work for statistical tasks as well. Their appeal is their simplicity: a conceptually simple algorithm can, by putting together many simple pieces, model quite complex patterns in data.

One such algorithm is called Classification and Regression Trees (CART). Suppose we have a vector Y of n observed outcomes—either categorical outcomes, for classification, or continuous outcomes for regression—and a $n \times p$ matrix X of observed covariates for each observation. We would like to make a prediction $\widehat{y}$ for a given set of newly observed predictors x. CART does this first by *dividing:* splitting up the space of possible values of X into many regions. It then *conquers* by producing a single estimate $\widehat{y}$ for each of these regions, usually just a constant value in each region. The trick, of course, is all in how we divide the space.

For simplicity, we'll focus on binary classification here, and save regression for later. In binary classification, Y is always either 0 or 1 (e.g. "survived" vs. "didn't survive", "spam" vs. "not spam", and so on). Given a new observed x, our classifier should predict 0 or 1. CART can, as its name suggests, also be used for regression problems where Y can take on a continuous range of values, and it can also handle classification when there are more than two possible classes, but we will ignore these possibilities for now.

For CART to divide the space of X values into regions, we need to define what makes a "good" region and what makes a "bad" region. We define something called the *impurity* of each region, and pick splits that try to reduce the impurity the most. In classification, a pure region would be one where Y is always the same value—always 0 or 1—and the most impure region would be one where half the observations are 0 and half are 1, since we would make the worst predictions in this case. Let $p(y = 1 \mid A)$ be the probability that the observed Y in region A is 1. The impurity $I(A)$ will be some function ϕ of this:

$$I(A) = \phi\left(p(Y = 1 \mid A)\right).$$

There are several common choices of ϕ:

$$\phi(p) = \min(p, 1 - p) \qquad \text{(Bayes error)}$$

$$\phi(p) = -p\log(p) - (1 - p)\log(1 - p) \qquad \text{(cross-entropy)}$$

$$\phi(p) = p(1 - p) \qquad \text{(Gini index)}$$

Each of these functions has its maximum at $p = 1/2$ and its minimum at $p = 0$ and $p = 1$, so pure regions have all their data points in the same class and impure regions have half and half.

CART picks regions by recursively splitting regions, forming a tree. We start with a tree with only one node, the root node: we predict $\widehat{y}$ to be the most common value of Y, regardless of the x we are given. This is obviously not a very good model, so we want to split the region up. We do this as follows. For a region A, consider a possible covariate X_j, for $1 \leq j \leq p$. Then,

1. Consider splitting A into two regions, A_L and A_R, where A_L contains all observations in A with $X_j < s$ and A_R contains all observations in A with $X_j \geq s$, for some value s. The fraction of observations in A that falls into A_L is p_L, and similarly the fraction falling into A_R is p_R.

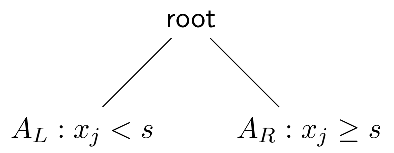
2. Calculate the reduction in impurity,

$$\Delta I(s, A) = I(A) - p_L I(A_L) - p_R I(A_R).$$

   Pick the value of s that maximizes $\Delta I(s, A)$.

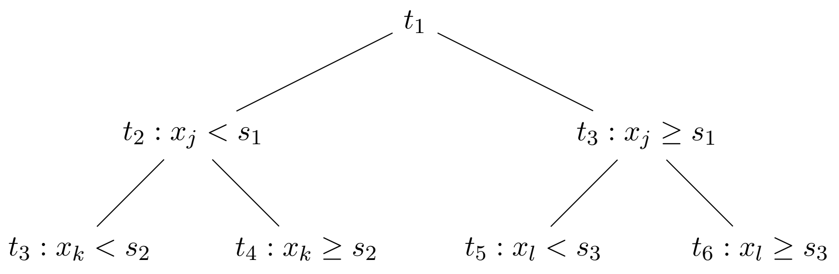
3. Repeat these steps for each possible covariate j. Pick the covariate j and split s that reduce the impurity of A the most.

Once we have split A, we now have a tree with two nodes, A_L and A_R:



To predict $\widehat{y}$ for a new x, we start at the root of the tree and work down. If $x_j < s$ we go to the left child, and our prediction $\widehat{y}$ is the most common value of Y for the observed data points in that node. Otherwise we go to $x_j \geq s$ and make our prediction there.

The tree doesn't stop there, of course. We repeat this procedure on both A_L and A_R, splitting them up as well, and continue recursively splitting nodes until we stop—usually when the node only has a few data points in it, or when all data points in the node are of the same class. We might end up with a tree that looks like this:

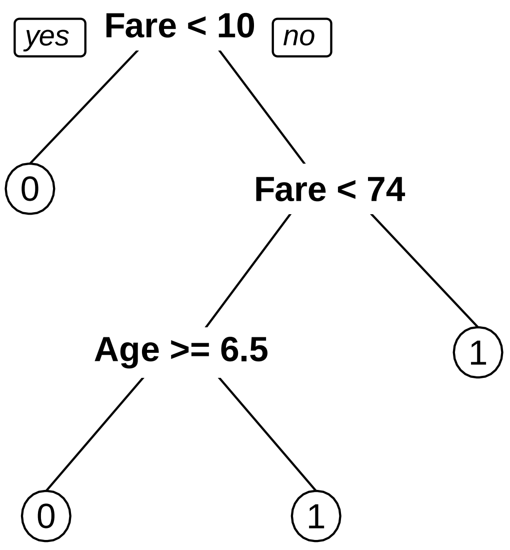


Notice I have labeled the nodes $t_1, t_2, \ldots$, so we can refer to them more easily. Also note that t_2 was split along variable x_k, but t_3 was split along x_l, which need not be the same variable; each node is split based on the variable that would reduce its own impurity the most. A node can be split on the same variable its parent was split upon—every variable is considered as a possible split at every level of the tree.

Continuing the tree metaphor, we'll call nodes t_3, t_4, t_5, t_6 the *leaves* of the tree, since they are where we make predictions and they are not split any further. The leaves are mutually exclusive: t_3, for example, is the region $\{X : X_j < s_1, X_k < s_2\}$, which does not intersect with any other leaf. We'll write $\mathsf{leaves}(T)$ to refer to the set of leaves for a given tree T.

A Quick Example

Let's look at an easy-to-interpret dataset: the passengers on the *RMS Titanic*, classified by whether or not they survived its sinking in 1912. Here Y is 1 if the passenger survived and 0 otherwise, and X is a set of covariates for each passenger. We'll consider only $p = 2$ covariates: the passenger's age and the fare they paid, which reflects what class of service they bought (such as first-class cabins or third-class shared rooms). Using 891 passengers as training data, we can grow a tree:



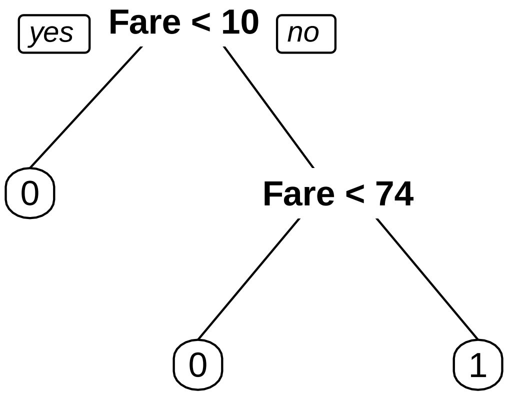
The interpretation is quite brutal. If your fare was cheap, we predict you did not survive. If the fare was mid-range (between \$10 and \$74), you survived if you were under 6.5 years old. ("Women and children first" was the rule as they filled the lifeboats.) And if you had an expensive ticket, you survived.

Of course, there are other covariates we could use, and this very simple tree only predicts 70% of the training cases correctly (though 70% is impressive for such a simple tree!). The dataset includes the specific cabin each passenger was in, the size of their families and whether they were traveling single or together, and so on, and we could include these covariates, and perhaps prune the tree less severely, to improve classification accuracy. But this demonstrates the conceptual simplicity of a classification tree: divide the covariate space up and make single 0/1 predictions in each portion.

Pruning Trees

When exactly should we stop building a tree? This is a tricky question. If we split the space until every node has only a single data point in it, then when we predict Y for a new observation, we will essentially predict the value of Y for the training data point with the most similar value of X, similar to a nearest-neighbor approach. This is a very flexible model but has a lot of variance, and of course the classification tree would be very large and difficult to interpret. A better approach is to *prune* the tree. The Titanic tree above was pruned to make it simple.

Pruning a tree means removing a node and all of its descendants, replacing them with a single leaf node. (We can only prune nodes with descendants, not leaf nodes. It doesn't make sense to remove a leaf node, because what would we predict in that case?) For example, we could prune the Titanic tree above by removing the Age ≥ 6.5 distinction, instead using



We no longer distinguish between children and adults, replacing that node with a single leaf node predicting 0—death. This reduces the classification accuracy to about 67%, but makes the tree simpler. This is the bias–variance tradeoff: a very large tree can have low bias, but has high variance because it overfits; a very small tree can have low variance, but high bias because it can't fit real patterns in the data. Finding the right size tree is tricky, and the first step is to prune it.

To prune a tree, we start with a regularization parameter $\alpha \geq 0$. This parameter is chosen by the user before the pruning starts. This parameter is similar to the penalty λ in the lasso: $\alpha = 0$ means no pruning happens at all, and a large α prunes the tree a lot. In lasso, we penalize $\|\beta\|_1$, the L_1 norm of the parameter vector; when pruning a tree, we penalize $|\,\mathsf{leaves}(T)|$, the number of leaves in the tree T. The *cost-complexity measure* of the tree T is

$$R_\alpha(T) = R(T) + \alpha|\,\mathsf{leaves}(T)|,$$

where $R(T)$ is the total misclassification cost of the tree T, calculated by summing the misclassification cost $R(t)$ of all the leaves t of the tree:

$$R(T) = \sum_{t \in \mathsf{leaves}(T)} R(t) = \sum_{t \in \mathsf{leaves}(T)} r(t)p(t),$$

where

$$\begin{aligned} R(t) &= r(t)p(t) \\ r(t) &= \text{fraction of points in leaf } t \text{ that are misclassified} \\ p(t) &= \text{fraction of all data points that are in leaf } t. \end{aligned}$$

A useful property of the misclassification cost $R(T)$ is that it can only decrease when you're growing the tree—it can't increase.

Let $T' \preceq T$ be a subtree of T. (You can obtain a subtree by removing any node and all of its descendants.) We want to find a T' that is *optimally pruned*, meaning that its cost-complexity is smaller than any other:

$$R_\alpha(T') = \min_{T^* \preceq T} R_\alpha(T^*).$$

One can prove that, for a specific choice of α, there is a unique smallest optimally pruned tree. We can also prove that increasing the penalty α can only decrease the size of the tree, not increase it, so by increasing α, we get a sequence of trees that gradually get smaller and smaller until eventually the tree is only a root node.

How do we produce the optimally pruned tree? Suppose we have a tree T and are considering pruning a node t. The node t and its descendants form their own tree T_t; pruning t means removing all its descendants and turning it into a leaf node. We can calculate

$$g(t) = \frac{R(t) - R(T_t)}{|\,\mathsf{leaves}(T_t)| - 1},$$

where here $R(t)$ refers to the misclassification cost of the node t *if it were turned into a leaf node* by pruning T_t. We then define

$$\alpha^* = \min_{t \in T - \mathsf{leaves}(T)} g(t).$$

($T - \mathsf{leaves}(T)$ refers to all the nodes in T except the leaf nodes.) If $\alpha^* > \alpha$, it is not possible to lower $R_\alpha(T)$ by pruning T, so T is optimally pruned. Otherwise, we prune all nodes for which $g(t) = \alpha^*$. We then repeat the process, calculating $g(t)$ on the nodes of the freshly pruned tree, until we cannot find a node to prune for which $\alpha^* \leq \alpha$.

For a derivation demonstrating why this procedure works, consult David Austin's essay "How to Grow and Prune a Classification Tree": `http://www.ams.org/publicoutreach/feature-column/fc-2014-12`

Choosing the Best Prune

Question: how do we pick the best penalty α? What is the "best", anyway?

One approach is to pick the α that results in the smallest generalization error—the best accuracy when predicting Y for new observations that we *didn't* use when building the tree. To estimate the generalization error, then, we need to set aside some data and not use it to build the tree. One way to do this is *cross-validation*.

In cross validation, we build the tree using only a subset of the data, and use the left-out data to evaluate the classification accuracy of our tree. In k-fold cross-validation, the data is split into k folds, and $k-1$ are used to fit the tree and the remaining used to test it. Typically, one rotates through the folds in turn, using each fold as test set for a tree built with the other $k-1$, and calculates the average classification error over all k test sets.

We can use the cross-validation concept to choose α. Choose a reasonable value of k, such as 5 or 10. Fix a value of α. Build and prune trees using this value of α and calculate the average error on the test set across the k test folds. Repeat this procedure for a range of possible values of α, and choose the value of α that minimizes the test set error.

Forests of Trees

Random forests use the idea of ensemble classifiers: instead of building a single classifier, like a single tree, we build *many* classifiers in slightly different ways. The ensemble then votes on the classification of each data point. In random forests, we build ensembles of *many* classification trees, each with slightly different data. When we get a new x and want to predict $\widehat{y}$, we ask each tree to make its prediction, then pick the most common answer as our prediction.

What do I mean by "slightly different data"? To be specific, a random forest is built following this procedure:

1. Take a random sample of size n, with replacement, from the observations X. Also take the corresponding Y values. (This is a bootstrap sample.)

2. Build a classification tree using this sample, *except* that at each split, instead of choosing the best covariate x_j to split out of all covariates, choose from a random sample of k of the covariates. Use a new random sample for every split in the tree. Do not prune these trees.

3. Repeat steps 1 and 2 many times (say, 500 times). Store the collection of classification trees as your random forest.

A random forest, then, requires two tuning parameters: the number of trees in the forest and the number of randomly selected features k to consider at each split. No pruning is done, so no α is necessary.

You may want to consult the paper `classification-tree/Resources/breiman.pdf`, which introduces random forests, though rather abstractly.

Task

Part 1: Plans and Tests

First, you must plan out your implementation of classification trees. There are a lot of moving parts here—the tree, the splitting method, the pruning method, cross-validation—so careful design is important to prevent your code from becoming complicated and hard to work with.

In this part, you should

☐ Determine how you will represent your tree data structure. A tree consists of many nodes, each of which represents a split along a particular variable at a particular split point; all of this has to be stored in the tree.

You might use a **`Node`** class containing data fields like *`split_var`* and *`split_point`*, or a **`Tree`** class containing many nodes. How will you represent this in your programming language?

Write definitions of the classes/data structures you will use. Be specific about what data will be stored where.

☐ Design functions for operating on your tree. To implement classification trees, you'll probably need functions for adding nodes to a tree, removing nodes (to prune them), searching the tree, and so on. Design these functions.

It should be possible, for a given node, to obtain all the data points that fall into that node of the tree. For example, for the root node, this would be all the data; for a node midway through the tree, it should be the subset of the data obtained by splitting the dataset to get to this node. Design a function *`get_data`* that can do this. (It may have to traverse the tree to do this.)

(By "design", we mean you should write out the function names, arguments, and return values, with comments explaining their purpose, but ***do not*** write the actual code inside the functions. That will come in Part 2.)

☐ Design functions for building and pruning a classification tree. Consider how the data should be provided—a data frame? a matrix? how is the response variable specified?—and make sure that it's possible for the user to pick different values of ϕ, so they can use different impurity definitions. It should be possible for the user to specify a new ϕ that isn't built in to your code.

☐ Design functions for querying a classification tree—given a new X, they must determine the Y predicted by the tree.

☐ Design functions to build an ensemble of classification trees—a forest of many trees, using resampled data. These functions should reuse your classification tree functions in straightforward ways, rather than re-implementing the tree algorithms from scratch.

☐ Write an *`is_valid`* function that checks that a classification tree is valid. (Write the actual implementation, not just a design; your implementation will call all the functions you designed above.)

What does "valid" mean? Several things:

- No nodes are empty (contain no data)
- If a node is split on x_j, all data points in the left child should have $x_j < s$ and all points in the right child should have $x_j \geq s$.
- Applying the *`get_data`* function to every leaf node and combining the results should yield the same dataset as applying it to the root node.

  - Other properties you might think of while designing your code.

- ☐ Write tests for the functions you designed. The tests will not pass, of course, since the functions are not implemented, but the tests will help you in Part 2.

  Be sure to test thoroughly. Consider testing edge cases like:

  - One variable in the dataset has the same value for every data point (or only two possible values). Will your splitting code handle it correctly?
  - Two possible splits reduce the impurity by the same amount. Which is picked?

  Include randomized tests. There are many kinds of randomized tests you could write. One test, for example, would generate random data, build a tree, and call `is_valid` to ensure the tree is correctly built. Other randomized tests might ensure that the classifier yields the correct answer on contrived datasets where all Ys are equal in a certain region of the space.

Part 2: Trees and Forests

In this part, you should

- ☐ Implement all the functions you designed in Part 1. If you write any new functions you hadn't planned for in Part 1, make sure they have tests too. If you discover unexpected bugs or behavior you hadn't planned for, add tests accordingly. Ensure all your tests pass.

  You cannot use an existing package, in any programming language, that implements classification trees. Your implementation must be written from scratch.

- ☐ Write a benchmark system for your code that times how long it takes to build trees and how long it takes to query them. You should separately time loading the data (which only has to be done once) and querying, which can happen many times after the data is loaded. (The `microbenchmark` package for R may also be useful; Python provides a `timeit` module that's similar.)

  Write a script that calculates the times and prints out the results. Provide source code for any benchmarks or tests you run. Testing on a single input isn't sufficient—you must test on a range of inputs to be sure your results are representative. Analyze performance like a statistician! Summarize your results in your GitHub pull request text.

- ☐ Analyze the performance of your code. Test it on various input sizes and graph the results. Try to estimate the complexity (in big O notation) of your classification algorithm, in terms of the amount of data in the tree. Use the profiling tools in your programming language (like R's `Rprof` or Python's `profile`) to identify specific functions or portions of code that take a disproportionate time to run.

  Write up the results in a text file included with your submission. What could you improve? Are you satisfied with the speed of your classification tree?

☐ Sometimes we want to build classification trees for very large datasets: many millions of observations, far more than we can comfortably hold in a data frame or matrix in memory. Or perhaps we want to use a dataset that's also used by other applications for other analyses, so it needs to be shared and updated and changed over time.

One way to do this is to use a *database*. In class, we will introduce SQL, a query language for database servers like MySQL and PostgreSQL. A database server is very efficient at storing large quantities of data and searching it effectively—using clever data structures and indices to rapidly find the data you want, even if it doesn't all fit in RAM and has to be read from the hard disk.

In Part 3, you will alter your classification tree code to use a SQL database instead of a data frame or matrix. *Read the details below* for more information. In *this* part, you should make the preparatory changes you need.

Specifically: Plan any additional functions or classes you will need to read data from a SQL database. Figure out how the code you have written will need to change, and how you could modify it so it can accept either data frames *or* a SQL database. Should you extract out specific functions for, say, getting the rows of data whose x values are in certain ranges, so you can write those functions either to select from databases or to select with a SQL query? Should you make a new tree class, inheriting from your first class, which replaces the crucial methods with SQL-based ones?

Include a `DATABASE-PLAN.txt` file in your submission with your plan. Make any preparatory code changes that may be useful, but don't start writing the SQL code yet.

Part 3: Big Data

As mentioned in Part 2, we can't always fit all our data in memory, and SQL database servers are excellent systems for efficiently storing data in such cases. They use clever data structures so you can easily write queries like

```sql
SELECT SUM(is_evil) AS evilness, character_class FROM characters
WHERE franchise = 'marvel'
AND publication_year > 1964
GROUP BY character_class
ORDER BY evilness DESC;
```

and get the results *without* the database actually looking at every row in the `characters` table. Instead, databases usually use various indexes, often based on tree data structures, so they can search only rows meeting the requirements.

In this part, you'll use SQL to build a classification tree. Your tree-building code should take a connection to a SQL database (however that's represented in your programming language), a table name, and the names of the table columns to use as predictors and outcome variables. It will then query each of the predictor columns to find optimal split points, choose one to split on, and record the split; then, for each child, it will again use queries to pick the optimal split, and so on.

Crucially, your tree structure will *not* store the data, just the variable splits. To make a prediction for a specific x, you can find the matching leaf node of the tree, then query the database to find the data in that node.

To calculate impurities without having to load all the data into your code, we've provided a few SQL functions that calculate Bayes error, cross entropy, and the Gini index *inside PostgreSQL*, in `Resources/impurity.sql`. Once you run the commands in that file in your PostgreSQL database, you can run queries such as this:

```sql
SELECT bayes_error(fraction_ones(survived)) FROM titanic
WHERE fare >= 10
AND fare < 74
AND age >= 6.5;
```

to get the Bayes error. The `fraction_ones` function calculates p, the fraction of values that are 1, and each of the impurity functions takes this as its argument. Note that the column provided to `fraction_ones` must be declared in the `CREATE TABLE` statement as an `INTEGER` column, and it should contain only 0 or 1.

Warning: SQL tables do *not* have row numbers, like you're familiar with in R or pandas data frames. You cannot simply store the row numbers of the data in each node. Instead, remember that each leaf node represents a series of splits: in a certain leaf node, for example, we have that fare ≥ 10, fare < 74, and age ≥ 6.5, and you can use a `SELECT` query to find all the data corresponding to that node. You should **not** do a `SELECT` query to find rows with certain row numbers.

In this part, you should

☐ Build support for creating classification trees from SQL data *and* from data frames. ***Don't*** hardcode the methods, like this:

```python
if data_source == "data_frame":
    # get data from data frame
    data[column, :] = # ...
elif data_source == "sql":
    cur.query("SELECT data FROM ...")
```

Instead, it should be possible to add new data access methods without rewriting all the code. This could mean

- having a **`Tree`** class with a **`SQLTree`** subclass that overrides some methods
- extracting all data access out to functions that can be easily replaced
- having a **`DataFrame`** class and **`SQLTable`** class with common interfaces for accessing their data, and that can be given to your tree code to use to access data
- or some other suitable and flexible design.

Once the tree is built, it should be able to make predictions as well, and support everything that was possible in Part 2.

- ☐ Write tests for any new functions you've made, as needed. (It can be tricky to test SQL code, since it's annoying to write unit tests that access databases; extract out code into separate functions as much as possible, so the database access is limited to specific functions.)

- ☐ Make some test data tables in the SQL database you're using. Use this to test your SQL code, to ensure that the correct rows are returned when requested and the queries it uses are valid. (For example, if you had a **`DataFrame`** class and a **`SQLTable`** class, each with *`get_data_in_region`* functions that retrieve all rows with x values in a certain range, you should make sure the SQL version returns the correct results and matches the **`DataFrame`** version.)

- ☐ Load identical data into a SQL table and a data frame. (You can generate some test data for this purpose; be sure to include the script that generated the test data in your submission.) Build a randomized test that compares the results of your classifier on the SQL version and the data frame version of the data, making sure they give matching answers.

Part 4: Scrape and Classify

In this step, you'll use the classification tree you already built to classify some real data. We'll obtain the real data from the Internet, using a website's API to access real data and classify it. Ideally, you'll be able to leave your classification tree code unchanged, and write separate code to download the data, process it, and classify it.

We'll use the arXiv.org preprint server as our data source, since arXiv has a well-documented API that programs can use to access data in bulk. There is a helpful R package and a similar Python package for accessing the arXiv API. In this Part, we recommend you ***use a different programming language*** than you used in Parts 1–3, so you get practice integrating code in different languages.

Your task is:

- ☐ Write a script that can download abstracts from a specified arXiv category (like `stat.AP` or `econ.EM`) and date range. The script should be able to count the word frequencies for each abstract, in the style of the `information-retrieval-bow` homework exercise, and output each abstract's category and frequencies to a SQL database. It should be possible to run the script multiple times to keep updating the SQL database, e.g. to load successive months of data.

- ☐ Write a separate script that can select two specific arXiv categories from the database and identify the words whose frequencies differ the most between categories. Alternately, you may want to use more sophisticated dimension reduction on the word frequencies, for example using tf-idf to rescale the frequencies and then PCA to select a set of 10 or 20 linear combinations of frequencies that contain the most information.

  Once you have the final set of features, the script should divide the data into test and training sets (sizes to be determined by the user as an argument) and then train a classification tree to tell which of two categories an arXiv abstract came from. It should test the classifier on the test set and give a summary of its classification accuracy.

- ☐ Give a short report saying which categories you tried to classify, how much data you used, how you built the classifier, and how well it was able to distinguish the two categories in out-of-sample classification tests.

This Part is more open-ended than previous Parts: we will leave it to your judgment to decide how exactly to implement the classifier, how to store the data, and how to put the pieces together. You should aim for a simple and flexible structure with each piece clearly separated, rather than one big jumbled script that's hard to modify or reuse.

If you know of another website with interesting data that you'd like to classify, with an API or other way of downloading that data, you can use that instead of arXiv—but you must get approval from us at least *two weeks* before the *initial* Part 4 submission deadline, so we can verify the data source is suitable.

Requirements

A **Mastered** submission will meet the requirements described in each Part above. A **Sophisticated** submission will additionally:

- ☐ Use excellent variable names, code style, organization, and comments, so the code is clear and readable throughout.
- ☐ Be efficient, making the best use of its data structures with minimal copying, wasteful searching, or other overhead.
- ☐ Have a comprehensive suite of tests to ensure that the individual components function correctly on corner cases, error cases, and unexpected inputs.
- ☐ Give particularly thorough and detailed answers to the questions above.
- ☐ Supply a command-line driver that runs all your tests.

You should also remember that **peer review is an essential part of this assignment.** You will be asked to review another student's submission, and another student will review yours. You will then revise your work based on their comments. You should provide a clear, comprehensive, and helpful review to your classmate.