Project 1 – Recommender

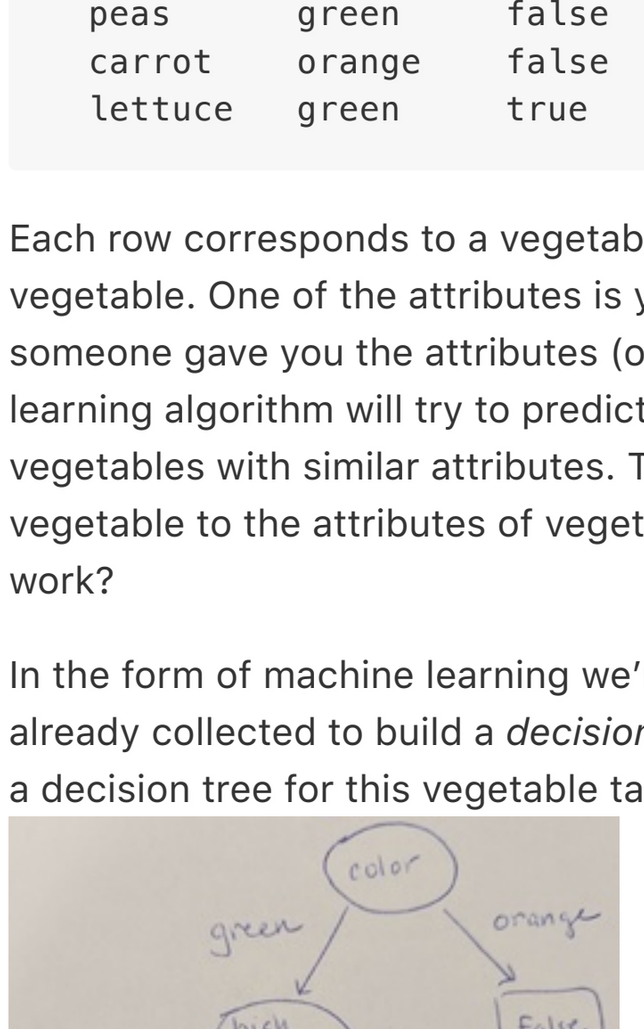
Overview

We hear a lot about machine learning, but how does it work underneath the hood? As a concrete example, imagine that you wanted a machine learning algorithm to tell you whether you were likely to enjoy a particular vegetable. You've been tracking your vegetable preferences, and have the following table:

```
name      color    lowCarb     highFiber     likeToEat
------------------------------------------------------
spinach   green     true        true          false
kale      green     true        true          true
peas      green     false       true          true
carrot    orange    false       false         false
lettuce   green     true        false         true
```

Each row corresponds to a vegetable and each column corresponds to an attribute of that vegetable. One of the attributes is your decision about whether you liked the vegetable. If someone gave you the attributes (other than the decision) for a new vegetable, the machine learning algorithm will try to predict whether you would like it, based on whether you liked other vegetables with similar attributes. To do this, it has to compare the attributes of the new vegetable to the attributes of vegetables you've already tried. But how does such a comparison work?

In the form of machine learning we're doing in this project, we're going to use the data you have already collected to build a *decision tree* to summarize your preferences. Here is an example of a decision tree for this vegetable table:



The nodes of the tree are labeled with attributes of the data (except for the name, which we are skipping in this example). The edges are labeled with values of attributes. The tree summarizes which values of `likeToEat` correspond to values of the other attributes. The root node considers the color attribute, for which there are two values in the dataset (green and orange). For the orange vegetables, `likeToEat` is always false. But `likeToEat` differs among the green vegetables. The tree next looks at whether a vegetable is `highFiber`. For green, low fiber vegetables, the decision in the dataset is always false, but decisions vary for high fiber vegetables. The tree thus looks at the `lowCarb` attribute. If `lowCarb` is true, you are just as likely to like or dislike the vegetable, so we pick one decision to use as the result.

If you wanted to use this tree to predict whether you would like a new vegetable, you would use the attributes of the new vegetable to traverse the tree from the root until you get to a decision leaf. At each node, you follow the edge whose label corresponds to the value of the attribute about the new vegetable. For example, if you wondered whether you would like acorn squash (which is orange), this tree would predict that you would not like it. If you wondered about green beans, which are low in fiber, this tree would predict that you would like it.

Here are things to note about the decision tree:

- each non-leaf node is labeled with an attribute from the dataset
- the same attribute arises only once along any path from the root to a decision
- a node for an attribute has outgoing edges corresponding to the possible values of that attribute from within the table
- the decisions are the predicted value for one attribute (here, `likeToEat`), based on the values of the other attributes. All decisions are for the same attribute.

In this project, you will write a program that generates a decision tree from a dataset and uses that tree to make predictions. We'll exercise various object-oriented practices to make our code robust. We'll evaluate the performance and social impacts of our implementation.

Architecture

Your implementation will consist of three main parts:

- Classes and interfaces for a decision tree
- Classes and interfaces for a dataset
- Classes and interfaces to generate and use a decision tree given a dataset

We will give you the interfaces. It is up to you to design and implement the corresponding classes. In addition, you will have to provide test cases and prose answers to reflective questions.

Your implementation may only use built-in datatypes that we have already covered in CS18 *(LinkedList, ArrayList, arrays, and any classes you define for yourself). In particular, those with prior Java experience may know other data structures (such as hashmaps/ dictionaries) which we have not yet covered in 18. To keep the workload even for everyone, and to maintain consistency in considering runtimes and space usage, you* ***cannot*** *use these other data structures on this project.*

Decision Trees

Decision trees differ from binary trees in two ways: (1) nodes can have any number of children and (2) edges have labels on them. You must decide how to implement each of these features. The only constraint is that your nodes (including leaves) must implement the following interface:

```
public interface INode {
    // traverse the tree based on attribute values to retrieve decision
    public Object lookupDecision(IAttributeDatum attrVals);

    // print tree
    public void printNode(String leadspace);
}
```

The `lookupDecision` method returns Object because the type of decisions can vary across datasets and individual attributes to be predicted.

Each non-leaf node will have an associated attribute from the dataset. The edges will be labeled with the possible values for that attribute.

Datasets

For purposes of this project, we will assume that each dataset corresponds to a table. The columns of the table are the attributes, while the rows are the individual items or observations for which we have values of attributes. Each attribute should have a fixed (enumerated) number of possible values (such as booleans or a set of specific numbers or strings), rather than numbers in general. (Real-world decision trees do support general numbers, but this restriction simplifies the project).

The following interface captures one item with attributes (corresponding to a row in the table):

```
public interface IAttributeDatum {
    // lookup the value associated with the named attribute
    public Object getValueOf(String attributeName);
}
```

The following interface captures a set of *IAttributeDatum*. The generic type `T` for the interface is the type corresponding to a single datum in the set. The role of these specific methods will be clear once we explain the tree-generation algorithm.

```
public interface IAttributeDataset<T extends IAttributeDatum> {
    // all the attributes in the dataset
    public LinkedList<String> getAttributes();

    // does every row/datum have the same value for the given attribute/column
    public Boolean allSameValue(String ofAttribute);

    // number of data/rows in the dataset
    public Integer size();

    // partition rows into subsets such that each subset has same value of onAttr
    public LinkedList<IAttributeDataset<T>> partition(String onAttribute);

    // get the value for ofAttribute, which is assumed to be common across all ro
    public Object getSharedValue(String ofAttribute);

    // get the most common value for ofAttribute in the dataset
    public Object mostCommonValue(String ofAttribute);
}
```

Decision-tree Generator

The generator supports building a decision tree for a dataset and making predictions with the generated tree.

```
public interface IGenerator {
   // build a decision tree to predict the named attribute
   public INode buildClassifier(String targetAttr);

   // produce the decision predicted for the given datum
   public Object lookupRecommendation(IAttributeDatum forVals);

   // print the decision tree
   public void printTree();
}
```

Your implementation will provide a class named `TreeGenerator` that implements this interface. The type parameter `T` is for the type of individual items (rows) in the dataset (a specific dataset is provided as a constructor argument to this class).

```
public class TreeGenerator<T extends IAttributeDatum> implements IGenerator {
   public TreeGenerator (IAttributeDataset<T> initTrainingData) {
       ...
   }
}
```

Building the Tree

A decision tree gets generated for a dataset and an attribute in the dataset that we want to predict. The values of the attribute to predict will be at the leaves of the decision tree.

To generate a tree for a subset of the dataset:

1. if the subset is empty, create a decision node that returns the most common value for the attribute to predict within the most recent non-empty subset of the dataset.
2. if all of the rows in the subset have the same value for the attribute to predict, create a decision node that returns the common value
3. otherwise, pick one attribute that has not been used during generation so far (i.e., it is not on the path from the root to the current spot in the tree). Create a new node for the chosen attribute. Partition the subset into further subsets, one for each value of the chosen attribute. For each new subset, create an edge for the corresponding attribute value and generate the tree for the subset.

The algorithm starts with the full dataset as the subset, and repeats until there are no further nodes to generate.

How do we choose the attribute to split on in step 3? This is one of the most interesting aspects of the algorithm. There are many ways one could do this. For example, you could:

- take the attributes in the order that they are in the table,
- choose an unused attribute at random, or
- choose an unused attribute that will minimize the size of the produced tree.

While the third is what gets used in practice, for this project we will instead use the second (pick an attribute at random) to simplify the problem.

Making Predictions

The goal of machine learning is to predict decisions on new data based on information about prior data. The `lookupRecommendation` method in the generator will take a collection of attributes and their values, use them to traverse the tree, then return the decision at the reached leaf.

What if the new data uses an attribute value that is not already in the decision tree? An example of this might lie in predicting whether you will like a yellow vegetable (like corn), when the initial data had no yellow vegetables. During traversal, if the value of an attribute is not in the tree, the most common decision across all data (rows) remaining at that point in the tree should be returned (this is already included in the generation algorithm; we're just reinforcing it here).

In particular, you cannot assume that you will have all of the attributes values up front. You will only know about those values that are represented in the initial dataset.

Ethics and Experemental Datasets

The recommender system you created can be used to make predictions in a variety of real-world contexts. However, applying your system to real-world problems could introduce unanticipated and harmful social consequences. Algorithmic bias, the systematic creation of unfair outcomes for one group of users, is one way that technological systems can perpetuate injustice. Before deploying any predictive tool, it is important to test for algorithmic bias in its predictions over time. In this section, you will test and discuss some of the potential social consequences of your recommender system in an algorithmic hiring context.

At many companies, algorithmic systems help hiring managers flag candidates for recruitment or screen large volumes of resumes. However, there are increasing examples of these tools unintentionally reproducing and amplifying bias in the hiring process. If you're interested, read about gender bias in Amazon's scrapped hiring tool here or other forms of bias in hiring algorithms here. (Note: A lot of the examples here and in our datasets are for cis female and cis male, even if not explicitly specified. We understand that there are other gender identities that also face bias and that are not covered in these datasets.)

Imagine that a large technology company uses your recommender system to help screen applications for software engineering positions. A parsing tool identifies the same set of attributes for each application. The attributes of each application are input in your recommender system to predict whether or not the application should be reviewed or discarded. Hiring managers trained the system on a collection of resumes from previously hired and rejected applicants.

Follow the instructions below to explore the potential social consequences of using your recommender system in this context. Write your answers to questions listed below in your README file.

Look at the files train_candidates_unequal.csv and train_candidates_equal.csv. These correspond to the datasets contained in the multiple tabs of the Google sheet linked here. These datasets contain attributes about job candidates determined by the parsing tool. While doing this section of the assignment, refer to the different hire percentages listed for training dataset, both equal and unequal, in the sheet. Note that the cis male and cis female testing datasets contain identical candidates, except for the gender attribute.

1. Copy `BiasTest.java`, `Candidate.java`, and `RecommenderCSVParser.java` into your `sol` directory by running `cp /course/cs0180/sol/recommender/sol/*.java ~/course/cs0180/workspace/javaproject/sol/recommender/sol/` (all one command) on a department machine. **Note:** when you install the `RecommenderCSVParser.java` file, you may get errors on the `import` statements at the top. To fix these, follow the steps in Section 5 of Lab 5.
2. Copy the dataset csv files into your `javaproject` directory by running `cp /course/cs0180/sol/recommender/sol/*.csv ~/course/cs0180/workspace/javaproject/` (all one command) on a department machine.
3. Follow the TODO comment in BiasTest.java, i.e first change the filepath variable in line 40 to `"train_candidates_unequal.csv"`
4. Look through the file `BiasTest.java` comments to understand what is going on in the file. Do not dwell on the CSVParser or the different loops. Just try to get the general idea of what is going on in this file. Run the file `BiasTest.java` (a few times if you get both the cis male and cis female hired ratio as zero). Run the file several times and write down the first five hire percentages you get (where both percentages are not zero).
   What do you notice in the differences between cis male and cis female hired ratios printed out?
5. Now, change filepath to `"train_candidates_equal.csv"`. Run the file several times and write down the first five hire percentages you get (where both percentages are not zero). Do you still see the bias in the different hiring ratio?
6. How do you think each training dataset (also presented on the Google sheet) affects the hiring bias?

Then, answer the following questions in your README (write a couple of sentences for each one):

- How could you modify your recommender system to prevent gender bias from arising? Identify an improvement to the training data and/or algorithm that could help prevent unfair outcomes.
- How does the approach your code used to choose which attribute to split on impact the resulting bias, if at all?
- If your hiring rates vary each time you build a new classifier, why does this occur? If we fix the algorithm to choose the same attribute to split on each time, could we eliminate the bias?
- Given the company's current skewed demographics, outline the extent to which ensuring fairness in this hiring system is the responsibility of the engineers, hiring manager, or someone else.

Algorithmic bias is easier to see in these examples because gender was included as an attribute in this dataset. However, algorithmic bias still exists in systems where protected attributes (race, gender, religion, ability, etc.) have been removed from training datasets. The inclusion of attributes that are correlated with protected attributes produces similar bias patterns in the results.

Let's take a look at our Candidate class, specifically two attributes: leadership experience and last position duration. Leadership experience is an important skill for recruiters but is also a correlated variable in the sense that currently, only 37 percent of companies have at least one woman on the board of directors. This means that the vast majority of executive positions are held by men.

If we take a look at last position duration, there are a variety of factors, unrelated to a candidate's competency, that may cause women to have more job turnover. Women may have shorter durations due to maternity leave or leaving workplaces environments that are hostile due to sexism or harassment. According to a report from the Center for Talent Innovation, over time, 52 percent of women working for science, engineering, and technology companies leave their jobs as a result of daily bias.
The algorithm we implement for this class does not take these factors into account when considering these attributes.

The train_candidates_correlated dataset reflects real life disparities like above. It is the same dataset as the train_candidates_unequal, but with the gender attribute removed. Run the BiasTest with train_candidates_correlated file following step 6 from above. In addition to that, follow the TODO from line 76 and line 83. Do you still see evidence of hiring bias?

Lastly, answer the following questions in the README (a few sentences for each):

- Proponents of algorithmic hiring argue that these systems have the potential to make the hiring process more efficient and remove some human biases from hiring decisions. Other than potential bias, what criteria do you think an organization should consider when deciding whether to use an algorithmic system for hiring?

- The recommender system you created could be used in a wide variety of contexts. You've observed the potential of the system to generate biased results. Do you the programmer have any responsibility to prevent potentially harmful uses of your code? What is one step that you (the programmer) could take to prevent negative consequences of someone using your general-purpose recommender system? You can propose a technical, non-technical, or policy idea. (Note that there is no "correct" answer being sought here – we are curious to see what ideas emerge from the class).

Testing

You may notice when you build your decision tree that the shape is different every time. This makes the actual tree generation difficult to write tests for. What you can do is write tests for any methods you write other than those that are actually constructing the tree. This means any methods in classes that implement `IAttributeDatum`, `IAttributeDataset` or `INode` as well as any other helpers you might write. **These are the only tests you are required to write for this project.**

The following paragraph describes a way of testing that we don't require you to do, it is here only as a resource.

The reason the shape of the tree is different every time is because we randomly pick the attribute to split on when building the tree. If you wish to check that your tree generation is working correctly, one way would be to replace the code that randomly picks an attribtue to split on and replace it with code that cycles through the attributes in a deterministic way, which will make your tree output the same every time. Just remember to change it back to random selection before you hand in!

Project Tasks

Source Code

Source code can be copied from the course directory by running `cp /course/cs0180/src/recommender/src/* ~/course/cs0180/workspace/javaproject/src/recommender/src` (all one command). You can also make the classes yourself if you want and copy the code from this handout into your project. **It is essential that you put all of the interfaces from this handout in your `src/recommender/src` directory in `javaproject`, not your `sol` direcory.** All of the code *you* write, including the `TreeGenerator` class will go in your `sol/recommender/sol` directory.

Design Check

At the design check, we want to see the following:

- Two hand-drawn examples of decision trees for the vegetables example that have different sizes by virtue of considering attributes in different orders. Both examples should be predicting the same attribute.
- Class outlines (names, fields, types on fields – no methods) that will implement the decision tree and dataset interfaces.
- An implementation of the Vegetable class from the above example.
- Your plan for handling attribute values that arise during prediction but weren't in the original dataset: are you building this into the nodes and edges of the tree, into the method that predicts the outcome, something else?

Design checks will be held Wednesday the 19th through Friday the 21st. We will send out a spreadsheet for you and your partner to select a slot on Sunday, please find a time before 3 PM on Tuesday the 18th. Before your design check, please upload your `Vegetable` class to Gradescope.

Final Submission

For your final submission, you will implement the algorithm as described in this handout. You will also experiment with your algorithm on a more realistic dataset. We will provide you with that dataset (and some code to read in datasets from csv files) after the design checks. When we release the experimental dataset, we will also release questions that have you think about the consequences of recommender systems in practice. The answers to these questions should go in your README along with a brief description of your code structure and any known bugs. Thus, your final submission will have four components: code, results of experimentation, written answers to these questions, and a description of your code.

FAQ

- **Do all of the paths have to consider the attributes in the same order?** No, as long as each attribute is only considered once per path, they can be considered in different orders on different paths.

- **Do decisions need to be booleans?** No, the decisions can be from any finite set of values (city names, color preferences, booleans, etc)