ITI 1121. Introduction to Computing II
Winter 2021

Assignments 2 and 3

Mehrdad Sabetzadeh and Guy-Vincent Jourdan – Copyrighted material

(Last modified on February 8, 2021)

Deadline for Assignment 2: February 26, 2021, 11:30 pm

Deadline for Assignment 3: March 19, 2021, 11:30 pm

Learning objectives

- Inheritance
- Interfaces
- Abstract Methods
- Polymorphism
- Experimentation with Lists

Introduction

In Assignments 2 and 3, we will take one step further towards building decision trees. Since the two assignments are closely related, we provide a **combined description of the two**. Towards the end of this description, we specify what needs be submitted for each of the two assignments. **Please note that Assignment 2 and Assignment 3 have different deadlines (February 26 and March 19, respectively).**

In these two assignments, we are going to consider all possible "splits" of the input data that we already read into a matrix (Assignment 1) and determine which split yields the best results. Before we explain what splitting means and how to measure the quality of a split, let us see an example of a decision tree and make our way from there. Consider the `weather-nominal` dataset shown in Figure 1.

outlook	temperature	humidity	windy	class
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
overcast	hot	high	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
overcast	cool	normal	TRUE	yes
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
rainy	mild	normal	FALSE	yes
sunny	mild	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	TRUE	no

Figure 1: The `weather-nominal` dataset

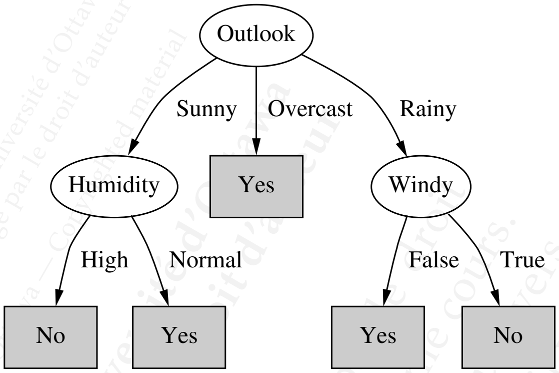
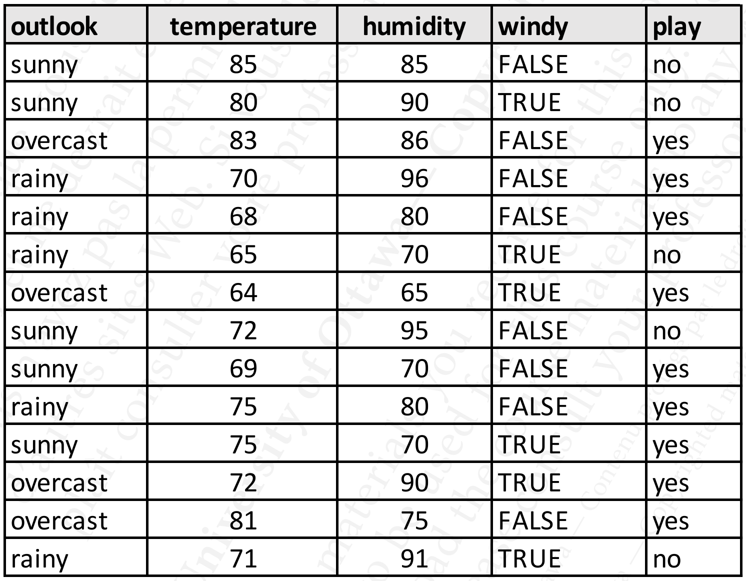


Figure 2: A decision tree (borrowed from [1]) learned from the dataset of Figure 1

Our goal is to eventually (that is, in Assignment 4) be able to build decision trees like the one shown in Figure 2. A decision tree uses a tree model to explain possible consequences or provide predictions, given a set of known parameters. For instance, in our weather example, we want our decision tree to take outlook, temperature, humidity, and whether it is windy or not as parameters and predict the "class" attribute. This attribute indicates whether a certain sports tournament (say, football or tennis) is feasible to play, given the weather conditions of the day. Obviously, we want our prediction to be more accurate than a coin toss! For this, we need to *train* a model – in our context, a decision tree – based on the data that we have observed previously. In our weather example, the previously observed data would be what is shown in Figure 1.

The reason why the last column of the data in Figure 1 is called "class" is because we are dealing with a *classification* problem, with the possible outcomes being `yes` or `no`. Since there are two outcomes only, the problem is a *binary classification* problem. **In the assignments for this course, we are concerned exclusively with binary classification. Furthermore, we assume that the "class" attribute is always the last attribute, irrespective of what the attribute is actually named.** For example, in the `weather-numeric` dataset, shown in Figure 3, the last attribute is named "play". For this dataset, we take "play" (the last column) to have exactly the same role as "class".



outlook	temperature	humidity	windy	play
sunny	85	85	FALSE	no
sunny	80	90	TRUE	no
overcast	83	86	FALSE	yes
rainy	70	96	FALSE	yes
rainy	68	80	FALSE	yes
rainy	65	70	TRUE	no
overcast	64	65	TRUE	yes
sunny	72	95	FALSE	no
sunny	69	70	FALSE	yes
rainy	75	80	FALSE	yes
sunny	75	70	TRUE	yes
overcast	72	90	TRUE	yes
overcast	81	75	FALSE	yes
rainy	71	91	TRUE	no

Figure 3: The `weather-numeric` dataset; the "play" attribute has the same role as "class" in the dataset of Figure 1

Semantically, the decision tree of Figure 2 is equivalent to the `if-else` block shown in Figure 4. The nice thing about our decision tree (and the corresponding `if-else` block) is that it is *predictive* and can project an outcome for weather conditions that have not been observed in the past. For example, our model has learned that "if the outlook is overcast, no matter what the other conditions are, we are good to play". Interestingly, there are several

```
if (outlook.equals("sunny")) {
    if (humidity.equals("high")) {
        class = "no";
    }
    else if (humidity.equals("normal")) {
        class = "yes";
    }
} else if (outlook.equals("overcast")) {
    class = "yes"
} else if (outlook.equals("rainy")) {
    if (windy.equals("FALSE")) {
        class = "yes";
    }
    else if (windy.equals("TRUE")) {
        class = "no";
  }
}
```

Figure 4: The decision tree of Figure 2 represented as `if-else` statements

combinations that have not been seen in historical observations. For example, the historical data does not tell us what would happen if the outlook is overcast, the temperature is hot, the humidity is normal, and it is windy. Indeed, if the outcome for all possible combinations were known, learning would be useless; all we would need to do was looking up the outcome in a table! In contrast, the model of Figure 2 makes an informed guess, without necessarily having seen the exact conditions int the past. This is the magic of machine learning: extrapolating from existing (training) data and projecting conclusions about situations that have not been seen before.

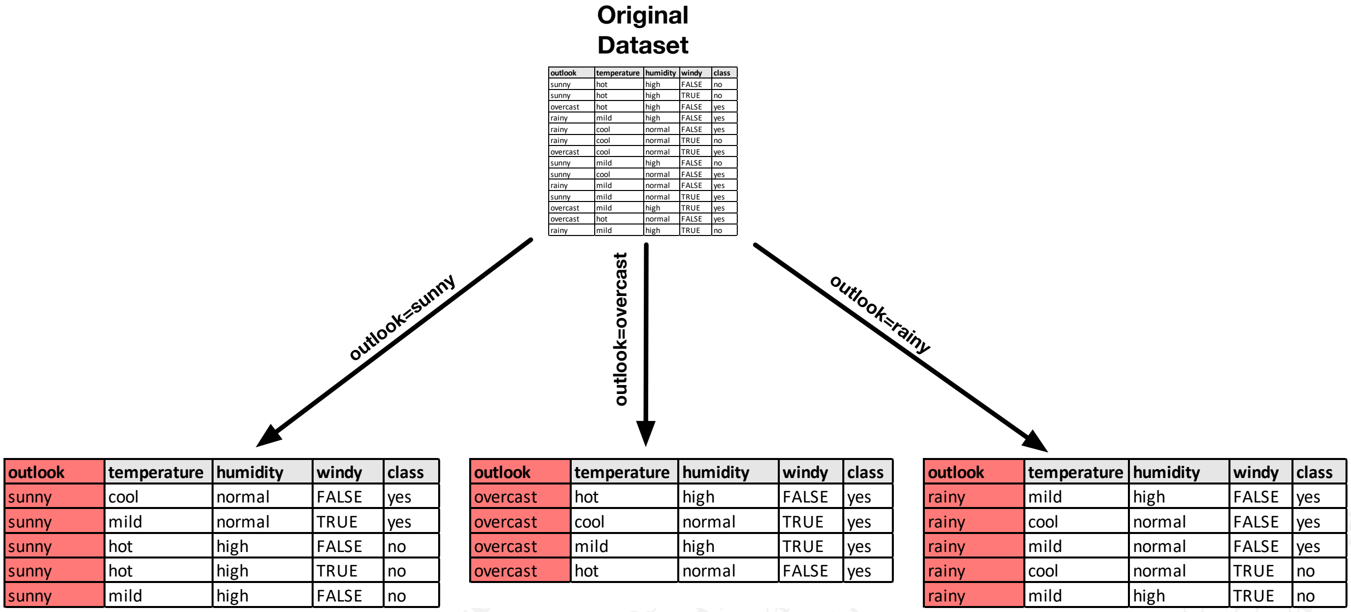
What will we do in Assignments 2 and 3?

These two assignments have a simple goal: **deciding which attribute should be the *root* of the decision tree.** The root is the attribute according to which a given dataset is split (partitioned) into smaller datasets. Once we see the concept of recursion later in the course, we will be able (in Assignment 4) to build a complete decision tree just by knowing how to determine the root at each level and splitting the data accordingly. For Assignments 2 and 3, we are concerned only with the root at the very top.

For the moment, assume that our dataset has only nominal attributes (we will deal with numeric attributes later). Going back to our example of Figure 1, let us first see what splitting means, before we determine which attribute is best to split over. Suppose the attribute we want to split over is "outlook". This attribute has three distinct values: {sunny, overcast, rainy}. These three values partition our original dataset into three smaller datasets. The three resulting partitions are shown in Figure 5.

The outlook column is shaded red in the partitions. By this, we mean that the outlook attribute is no longer part of the partitions. In other words, outlook has been *dropped* from the partitions. The reason for doing so is that, once we split over a nominal attribute, the partitions will each carry a single value for the split attribute. In general, any attribute that has a single value across a given dataset can be removed from the dataset, since such an attribute does not provide any useful information to learn from.

Now that we know how to split a dataset over a (nominal) attribute, let us see how we can measure the *quality* of the split. In our dataset of Figure 1, aside from the class attribute, we have four other attributes. These are: outlook, temperature, humidity, and windy. There are therefore four possible ways to split the dataset. In Figure 6, we show the possible splits. Notice that, for each split, Figure 6 provides only the "class" column of the resulting partitions. For example, Figure 6 (A) is a compact representation of what we already saw in in Figure 5. As we see next, our strategy for measuring the quality of a split requires only knowledge of the class columns.



outlook	temperature	humidity	windy	class
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
overcast	hot	high	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
overcast	cool	normal	TRUE	yes
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
rainy	mild	normal	FALSE	yes
sunny	mild	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	TRUE	no

outlook	temperature	humidity	windy	class
sunny	cool	normal	FALSE	yes
sunny	mild	normal	TRUE	yes
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
sunny	mild	high	FALSE	no

outlook	temperature	humidity	windy	class
overcast	hot	high	FALSE	yes
overcast	cool	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes

outlook	temperature	humidity	windy	class
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	mild	normal	FALSE	yes
rainy	cool	normal	TRUE	no
rainy	mild	high	TRUE	no

Figure 5: The result of splitting the dataset of Figure 1 over “outlook”; in the three resulting partitions, the outlook column is shaded red, indicating that the outlook attribute is *dropped* from the partitions

Measuring the quality of a split

The main quality that we seek in a decision tree is for the tree to be as small (and thus as general) as possible. Motivated by this intuition, we now need to define a quantitative metric to decide which split leads to the smallest tree. To this end, if we have a measure of *purity* for each partition, we can split over the attribute that, on average, yields the purest partitions. At this point, you may want to examine the possible choices in Figure 6 and contemplate over which attribute presents our best choice. What is your opinion?

The measure of purity that we use is called *entropy*. Entropy is one of the most fundamental concepts in information theory. A detailed mathematical understanding of entropy is unnecessary for our purposes and is beyond the scope of this course. As far as entropy goes, all we need are two things: (a) the overall intuition and (b) a formula that quantifies entropy for binary classification. The intuition for entropy is as follows: **The more the entropy, the more uncertain (unpredictable) the data is.** When we are building a classification tree, we would want to split over the attribute that reduces entropy the most (more illustration will come momentarily). As for the formula, we use *Shannon’s logarithmic entropy*.

Suppose that the two classes of interest (in our examples: no and yes) are denoted 0 and 1. For a given dataset D, let p_0 be the probability of observing 0 as class and let $p_1 = (1 - p_0)$ be the probability of observing 1 as class. The binary entropy function is computed as follows:

$$\text{Entropy}(D) = -p_0 \log_2 p_0 - p_1 \log_2 p_1 \qquad ;\text{ where } \log_2 \text{ is logarithm to base 2, and } 0 \log_2 0 \text{ is taken to be } 0$$

The value of the entropy function is 1 when unpredictability is the highest and 0 when unpredictability is absent (that is, we have full predictability). To better understand the formula, you can try it out for the following two scenarios: (i) when 0s and 1s occur equally frequently in D (the equivalent of a coin toss!), and (ii) when you have only 0s or only 1s in D. For case (i), you get: $(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}) = 1$, that is, maximum entropy. And, for case (ii), you get $(\cancel{-0\log_2 0} - 1\log_2 1) = 0$, that is, minimum entropy.

All that is left is to adapt entropy to the splitting process for decision trees. The metric that we choose for this purpose is called *information gain*. Intuitively, information gain measures how much we have been able to reduce entropy by splitting the dataset over a given attribute. More precisely, let D be a dataset, let t be an attribute of D, and let $P_1, \ldots, P_n$ be the partitions resulting from splitting D over t[1].

$$\text{Gain}(t) = \text{Entropy}(D) - \sum_{i=1}^{n} \frac{|P_i|}{|D|} \times \text{Entropy}(P_i)$$

[1] Notice that, here, n will be the number of distinct (unique) attribute values that attribute t has.

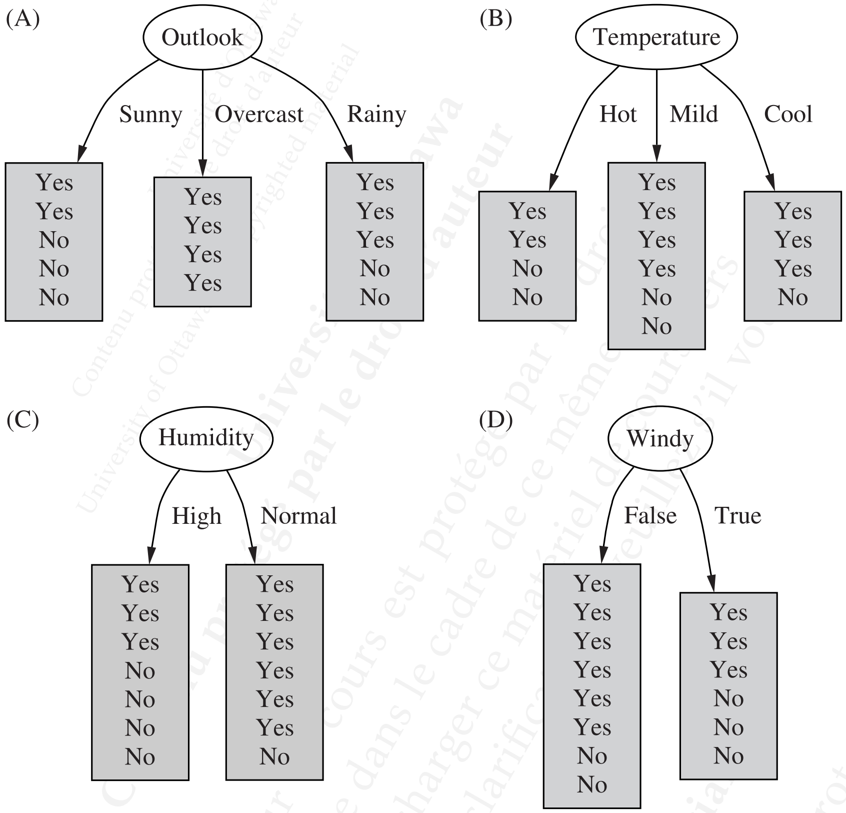


Figure 6: The result of splitting the dataset of Figure 1 over all its non-class attributes (borrowed from [1])

Do not be daunted by the gain formula! All the formula does it to deduct the entropy of D from the *weighted-average* entropy of P_i. Note that when we write $|X|$ for a set X, we mean the cardinality of the set, that is, the number of elements in X. Here, we are weighting each P_i's entropy by P_i's size, that is, by the number of data points in P_i.

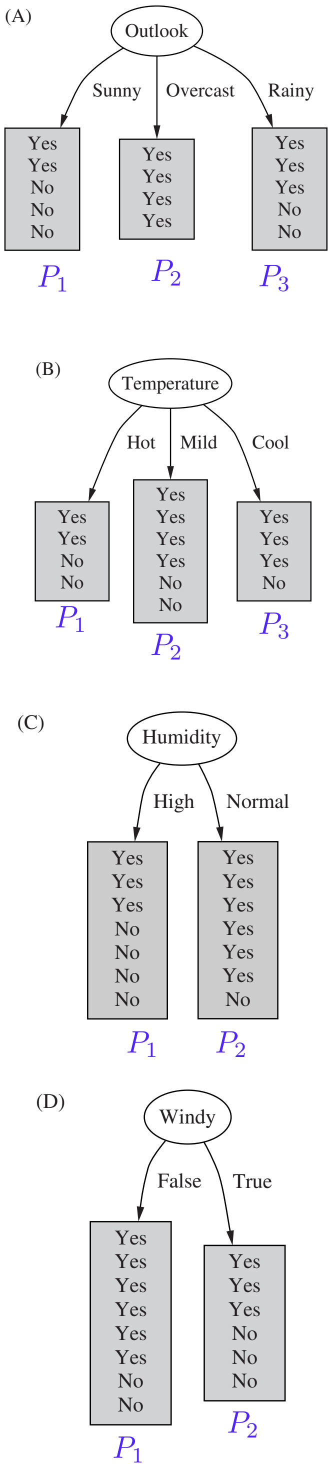
We are now ready to calculate the information gains for our four attributes in Figure 6. Let's start with the entropy of the entire dataset. Looking at Figure 1, we see that we have 5 occurrences of no and 9 occurrences of yes out of a total of 14 data points. The entropy of the whole dataset is therefore $(-\frac{5}{14}\log_2\frac{5}{14}-\frac{9}{14}\log_2\frac{9}{14}) \simeq 0.940$. With this baseline entropy calculated, we spell out the remainder of the gain calculations in Figure 7.

Splitting over numeric attributes

The splitting process that we saw so far does not work well for numeric attributes. To illustrate the issue that arises with numeric attributes, consider the `weather-numeric` dataset in Figure 3. We recall in Figure 8 (from Assignment 1) the unique value sets for the different attributes of the `weather-numeric` dataset.

Now, suppose we split over "humidity" using the same process that we just saw for nominal attributes. What will the result look like? We will get 10 distinct partitions (one partition for each value in the value set of humidity)! These partitions would be way too small, containing one, two, or at most three data points (for humidity = 70). The split is a nice fit for our dataset, but it is way too tailor-made for it (this phenomenon is known as *overfitting* in machine learning). If you try to use a decision tree with these 10 humidity branches to predict whether the sports tournament can be played today, and it just so happens that today's humidity reading is 72 – a value that you have not seen before – you will not get any prediction! This is at odds with commonsense which tells you "well, I have never seen exactly 72, but this is pretty close to 70 and 75, which I have seen before".

$$|D| = 14$$



$$|P_1| = 5 \quad |P_2| = 4 \quad |P_3| = 5$$
$$\text{Entropy}(P_1) = -\tfrac{2}{5}\log_2 \tfrac{2}{5} - \tfrac{3}{5}\log_2 \tfrac{3}{5} \simeq 0.971$$
$$\text{Entropy}(P_2) = 0$$
$$\text{Entropy}(P_3) = -\tfrac{3}{5}\log_2 \tfrac{3}{5} - \tfrac{2}{5}\log_2 \tfrac{2}{5} \simeq 0.971$$
$$\text{Gain}(\text{outlook}) = 0.940 - (\tfrac{5}{14} \times 0.971 + \tfrac{4}{14} \times 0 + \tfrac{5}{14} \times 0.971) \simeq 0.247$$

$$|P_1| = 4 \quad |P_2| = 6 \quad |P_3| = 4$$
$$\text{Entropy}(P_1) = -\tfrac{2}{4}\log_2 \tfrac{2}{4} - \tfrac{2}{4}\log_2 \tfrac{2}{4} = 1$$
$$\text{Entropy}(P_2) = -\tfrac{4}{6}\log_2 \tfrac{4}{6} - \tfrac{2}{6}\log_2 \tfrac{2}{6} \simeq 0.918$$
$$\text{Entropy}(P_3) = -\tfrac{3}{4}\log_2 \tfrac{3}{4} - \tfrac{1}{4}\log_2 \tfrac{1}{4} \simeq 0.811$$
$$\text{Gain}(\text{temperature}) = 0.940 - (\tfrac{4}{14} \times 1 + \tfrac{6}{14} \times 0.918 + \tfrac{4}{14} \times 0.811) \simeq 0.029$$

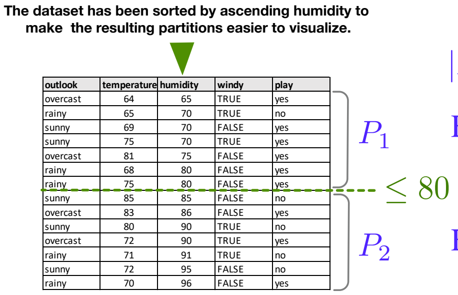
$$|P_1| = 7 \quad |P_2| = 7$$
$$\text{Entropy}(P_1) = -\tfrac{3}{7}\log_2 \tfrac{3}{7} - \tfrac{4}{7}\log_2 \tfrac{4}{7} \simeq 0.985$$
$$\text{Entropy}(P_2) = -\tfrac{6}{7}\log_2 \tfrac{6}{7} - \tfrac{1}{7}\log_2 \tfrac{1}{7} \simeq 0.592$$
$$\text{Gain}(\text{humidity}) = 0.940 - (\tfrac{7}{14} \times 0.985 + \tfrac{7}{14} \times 0.592) \simeq 0.152$$

$$|P_1| = 8 \quad |P_2| = 6$$
$$\text{Entropy}(P_1) = -\tfrac{6}{8}\log_2 \tfrac{6}{8} - \tfrac{2}{8}\log_2 \tfrac{2}{8} \simeq 0.811$$
$$\text{Entropy}(P_2) = -\tfrac{3}{6}\log_2 \tfrac{3}{6} - \tfrac{3}{6}\log_2 \tfrac{3}{6} = 1$$
$$\text{Gain}(\text{windy}) = 0.940 - (\tfrac{8}{14} \times 0.811 + \tfrac{6}{14} \times 1) \simeq 0.048$$

Figure 7: Calculating information gain for the different possible splits of the dataset in Figure 1

```
1) outlook (nominal): ['sunny', 'overcast', 'rainy']
2) temperature (numeric): [85, 80, 83, 70, 68, 65, 64, 72, 69, 75, 81, 71]
3) humidity (numeric): [85, 90, 86, 96, 80, 70, 65, 95, 75, 91]
4) windy (nominal): ['FALSE', 'TRUE']
5) play (nominal): ['no', 'yes']
```

Figure 8: Unique values sets for the weather-numeric dataset (from Assignment 1)



The dataset has been sorted by ascending humidity to make the resulting partitions easier to visualize.

outlook	temperature	humidity	windy	play
overcast	64	65	TRUE	yes
rainy	65	70	TRUE	no
sunny	69	70	FALSE	yes
sunny	75	70	TRUE	yes
overcast	81	75	FALSE	yes
rainy	68	80	FALSE	yes
rainy	75	80	FALSE	yes
sunny	85	85	FALSE	no
overcast	83	86	FALSE	yes
sunny	80	90	TRUE	no
overcast	72	90	TRUE	yes
rainy	71	91	TRUE	no
sunny	72	95	FALSE	no
rainy	70	96	FALSE	yes

$$|D| = 14$$

$$|P_1| = 7 \quad |P_2| = 7$$

$$\text{Entropy}(P_1) = -\tfrac{6}{7}\log_2\tfrac{6}{7} - \tfrac{1}{7}\log_2\tfrac{1}{7} \simeq 0.592$$

$$\text{Entropy}(P_2) = -\tfrac{3}{7}\log_2\tfrac{3}{7} - \tfrac{4}{7}\log_2\tfrac{4}{7} \simeq 0.985$$

$$\text{Gain}(\text{humidity} \leq 80) = 0.940 - (\tfrac{7}{14} \times 0.592 + \tfrac{7}{14} \times 0.985) \simeq 0.152$$

Figure 9: Example of splitting over the humidity attribute in weather-numeric (criterion: $x \leq 80$ versus $x > 80$)

For numeric attributes, we observe that, most often, there is a natural *order*.[2] We therefore define split around the notion of "less than or equal" ($\leq$). Specifically, our numeric split algorithm works as follows:

1. For every value v_i from the value set of a numeric attribute t do:
   (a) partition the dataset into two subsets P_1 and P_2. P_1 will contain the data points where the value of t is $\leq v_i$. P_2 will contain rest of the dataset (that is, the data points where the value of t is $> v_i$);
   (b) Calculate and store an information gain score Gain(v_i) for this two-way split;
2. The gain for t, that is Gain(t), is the *largest* calculated Gain(v_i)

In Figure 9, we illustrate sub-steps (a) and (b) where t is the humidity attribute and $v_i = 80$. To be able to better visualize P_1 and P_2, we have sorted the weather-numeric dataset (D) in ascending order of humidity. Notice that, whereas splitting over a nominal attribute leads to that attribute being dropped from the resulting partitions, a numeric attribute will *persist* in the generated partitions (as long as the numeric attribute in question has more than one value left in a partition). For example, the humidity attribute still remains part of P_1 and P_2 in Figure 9, although the original value set of humidity is now split between P_1 and P_2.

For humidity, the gain computed at $v_i = 80$ turns out to be the maximum gain for any of the 10 elements in the value set. The information gain from splitting over humidity is therefore 0.152. We reached this conclusion only after calculating the gains for *all* elements in the value set of humidity. For completeness, we provide in Figure 10 the gains calculated for all the elements in the value set of humidity.

```
Gain from splitting at <=85 is: 0.048126996
Gain from splitting at <=90 is: 0.07930398
Gain from splitting at <=86 is: 0.1022436
Gain from splitting at <=96 is: 0.0
Gain from splitting at <=80 is: 0.1518355
Gain from splitting at <=70 is: 0.014956057
Gain from splitting at <=65 is: 0.047709167
Gain from splitting at <=95 is: 0.047709167
Gain from splitting at <=75 is: 0.04533428
Gain from splitting at <=91 is: 0.0103181
```

Figure 10: Gain scores for the whole value set of the humidity attribute in the weather-numeric dataset

[2] There are situations where numeric attributes have no order and are, in fact, nominals. An example would be bus route numbers. It does not make sense to say that "route number 98" is less than "route number 100". For the purposes of this course's assignments, we do not concern ourselves with nominal numerics. In other words, for simplicity, we assume that numeric values always have a natural order.

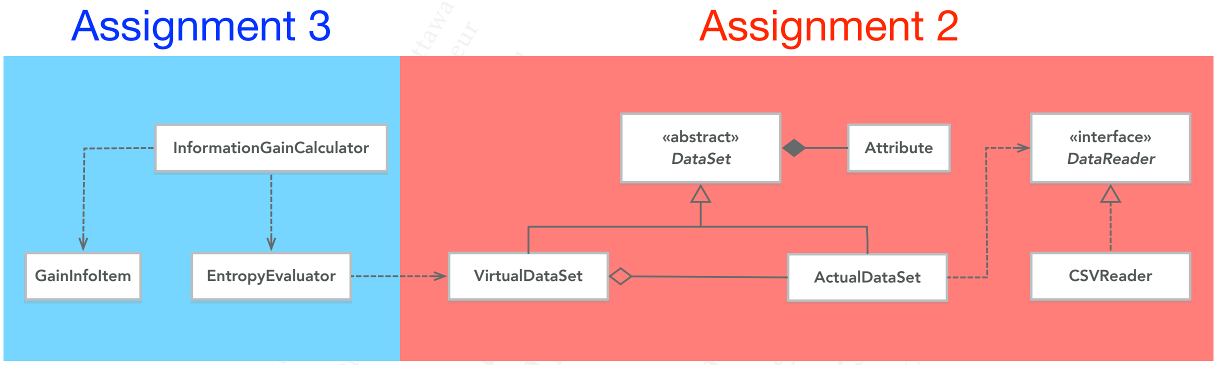


Figure 11: Implementation overview

Implementation

We are now ready to program our solution. An overview of our **object-oriented design is shown in Figure 11**, using a UML class diagram. The diagram further shows the separation between Assignment 2 and Assignment 3. For both assignments, you need to follow the patterns that we provide. **You cannot change any of the method signatures. Moreover, you cannot add any new *public* methods or variables.** You can, nevertheless, add new *private* methods to improve the readability and/or organization of your code.

Before moving further, let us go through the rationale for the design of Figure 11. The main considerations that went into this design are as follows:

- In Assignment 1, the function of reading a CSV file was implemented in the `DataSet` class itself. This choice is not particularly good. Why? Because, CSV is not the only possible format in which a dataset can be provided. For example, a dataset may be stored in a database or as an excel sheet (.xls or .xlsx formats). To decouple `DataSet` from the CSV format, our design defines a `CSVReader` class. This class is dedicated to dealing with the CSV format. Ultimately, we expect that the attribute names and the data matrix should be read from the data source, be the data source in CSV or any other format. To ensure that datasets are decoupled from the details of their storage, we have our readers implement the `DataReader` interface. Here, we have only one class implementing this interface and that is `CSVReader`. In the future, if we need to deal with another file format, say .xlsx, we will need to have the corresponding reader implement `DataReader`.

- In Assignment 1, the `metadataToString()` method derived some important information (metadata) about the attributes of a dataset. However, all we did with the metadata in `metadataToString()` was to write it out to the standard output and then forget about it. In Assignments 2 and 3, we need to retain and manipulate the metadata that we derive for the attributes of a dataset. To be able to do so, we define the `Attribute` class.

- In Assignments 2 and 3, we need to decompose a given dataset into subsets (more specifically, partitions). The original dataset may have a very large data matrix. To be efficient, we need to implement the subsets *without* duplicating the information in the original dataset's matrix. So, what can we do to avoid the duplication of data? One way is to distinguish between an `ActualDataSet` and a `VirtualDataSet`. Instances of `ActualDataSet` will carry a data matrix in them. **In contrast, instances of `VirtualDataSet` have no data matrix in them.** Instead, `VirtualDataSet` instances are *views* over an actual dataset. These instances will carry a *map* to keep track of how their (virtual) rows map onto the (actual) rows in an instance of `ActualDataSet`.

- As we will see later, `VirtualDataSet` will implement the methods that can split a dataset over nominal and numeric attributes. Entropy calculations will be performed by the `EntropyEvaluator` class using the partitions produced by the `VirtualDataSet` class.

- Finally, the `InformationGainCalculator` class will calculate and sort the information-gain score that result from splitting over different attributes. To properly structure and retain the calculated gain information[3], we define the `GainInfoItem` class. Instances of `GainInfoItem` store the gain scores calculated for different attributes. For numeric attributes, `GainInfoItem` instances additionally store the numeric value at which splitting produces the best gain. In our example of Figure 10, the best gain for the humidity attribute is obtained when the split is at humidity ≤ 80.

[3] We will need this information in Assignment 4.

To assist you with the implementation, **the work is broken down into seven tasks, numbered 1 to 7. Tasks 1 to 4 are to be completed for Assignment 2. Tasks 5, 6, and 7 are to be completed for Assignment 3**. Please bear in mind that the tasks are *not* of equal sizes. Some of these tasks are easy; others, marked with an asterisk (*), involve more thinking and implementation. The tasks with a * will take more time to complete. Starting early and managing your time would therefore be paramount. This is particularly so for Assignment 2: **Anticipate 60% to 75% of the Assignment 2 work to be concentrated in Task 4. Make a plan and manage your time wisely.**

As long as you complete Tasks 1-4 by the deadline of Assignment 2 and Tasks 5-7 by the deadline of Assignment 3, you are permitted to tackle the tasks in any order you like. However, **we encourage you to follow the order that we recommend** (that is, Task 1, then Task 2, and so on) to ensure smooth progression.

Like in Assignment 1, guidance is provided through comments in the template code you receive. The locations where you need to write your code are clearly indicated with an inline comment which reads as follows:

```
// WRITE YOUR CODE HERE!
```

Task 1. Complete the `CSVReader` class

This is an easy task: `CSVReader` is an adaptation of the CSV parsing function you already developed in Assignment 1. There is little additional code to write here; all you need to do is to ensure that `CSVReader` properly implements the methods in the `DataReader` interface. To start, please consult the javadoc documentation of `DataReader`. Should you so wish, you can reuse code from the instructor-provided implementation of Assignment 1. Alternatively, you can base `CSVReader` on your own implementation of Assignment 1. If you choose to do the latter, please note the following: if `CSVReader` does not work properly due to issues in your own A1 implementation, these issues may propagate to A2 and A3, potentially resulting in lost marks in these assignments as well.

Task 2. Complete the `Attribute` class

You need to implement all the methods in the `Attribute` class, except for the `toString()` method which is being provided to you. There is adequate documentation in the template code to guide you through this. You may also want to examine the `getAttribute(...)` methods in the `DataSet` class to better understand what is expected of `Attribute`. The `DataSet` class is being provided to you in full. You do not need to modify the `DataSet` class in Assignments 2 and 3. Notice that, in contrast to Assignment 1, we are no longer treating attributes just as name labels. Instead, attributes are now objects that, in addition to a name, hold an index[4] (column number), a type, and the set of unique values for the attribute.

An important consideration when implementing the `Attribute` class is to ensure that there is no object sharing taking place between the attributes of different datasets. In particular, when we build a virtual dataset over an actual dataset, the virtual dataset is often a proper subset of the actual one. This means that the attributes of the virtual dataset can have a smaller set of unique values. Now, if we let the virtual dataset "reuse" the attribute objects of the actual dataset, trouble may ensue. Why? Because the virtual dataset is going to adjust the value sets of its attributes. If the attribute objects are shared with another dataset, that other dataset will be affected too, which is undesirable.

To avoid the sharing of attribute objects, we need to make sure that `DataSet` instances do not expose references to any of their own attribute objects. Without such provision, our implementations of A2, A3 and (later A4) are prone to bugs, because we will be creating many virtual datasets over an actual dataset.

If you look at the `getAttribute(...)` methods in `DataSet`, you see that the methods return a reference to a copy (*clone*) of an attribute as opposed to a reference to the attribute that the dataset maintains for its own use. For the `clone` method in `Attribute`, you need to perform a deep copy, meaning that `clone` will have its own copy of the `values` array as well. Have a look at Appendix A to better understand what is required to achieve this.

In addition to the above, you need to make sure that:

- any string array (`String[]`) passed as parameter to an instance method of `Attribute` is duplicated before being stored. Why? Because you cannot be sure what other objects know about that string array and whether these object may alter the array.

- `getValues()` in `Attribute` returns a reference to a copy of the `values` array (as opposed to "`return values;`".

[4] Notice that the attribute is named `absoluteIndex`. We come back to what this means when we discuss the `VirtualDataSet` class.

Task 3. Complete the ActualDataSet class

There are four sub-tasks here:

- Task 3.1. Complete the constructor of ActualDataSet. First, you need to initialize dataSourceId, numAttributes, numRows, matrix using the DataReader instance passed as parameter. Next, you need to create the attributes for the dataset, that is, initialize the (protected) attributes variable defined in DataSet.

- Task 3.2. Complete the getValueAt(...) method. Notice that this will be a concrete implementation for an abstract method of DataSet. Hint: getValueAt for ActualDataSet is really easy! Look at what you did in A1. This is a warm-up for implementing getValueAt(...) in the ActualDataSet, which requires a little bit of extra thinking.

- Task 3.3. Override the toString() method of DataSet, adding extra information, followed by a call to super.toString(). Example output is provided in Figure 13.

- Task 3.4. Complete the toVirtual() method. This method creates and returns a (virtual) dataset that covers the entire data in the current (actual) dataset. You may want to postpone Tasks 3.4 to *after* you have completed Task 4.

Task 4*. Complete the VirtualDataSet class

- Task 4.1. Implement the constructor of VirtualDataSet. As you can see from the template code, VirtualDataSet has an instance variable named map (an array of integers). This variable will store the indices for the rows of the actual dataset that are included in the virtual dataset. **You are prohibited from copying the data matrix of an actual dataset into a virtual dataset.** Doing so is not only time-consuming but can also cause the Java Virtual Machine to quickly run out of memory if your data matrix is large. Important details about the implementation of the VirtualDataSet constructor have been provided as comments in the template code. Please read the comments carefully. A key detail is that the VirtualDataSet constructor is in charge of checking the value sets of its attributes and pruning these value sets (if necessary) according to the data rows contained in the dataset. This is illustrated in the example output of Figure 13.

- Task 4.2. Complete getValueAt(...). Hint: you simply need to call source.getValueAt(...) with suitable parameters.

- Task 4.3*. Complete partitionByNominalAttribute(...). This method splits a (virtual) dataset over a given nominal attribute. We elaborated what splitting over an attribute means earlier in the description. Notice that, for a nominal attribute, you will have as many partitions as there are values in the (unique) value set of the given attribute. This was already illustrated in Figure 5.

- Task 4.4*. Complete partitionByNumericAttribute(...). This method splits a (virtual) dataset over a given numeric attribute and at a given value. This too was elaborated earlier in the description. Notice that, once you have implemented the partitionByNominalAttribute(...) method above, partitionByNumericAttribute(...) is going to be a relatively easy adaptation. Here, there will always be two partitions, as already illustrated in Figure 9.

Note: ***For Tasks 4.3 and 4.4, you are allowed to use Java Collections, for example, java.util.LinkedList.***

Task 5*. Complete the EntropyEvaluator class

Following the explanation given earlier about the calculation of entropy, complete the static EntropyEvaluator.evaluate(...) method. In this assignment, you need to use binary logarithm (that is, logarithm to base 2) for calculating entropy. For this, you can use the static EntropyEvaluator.log2(...) method provided to you.

Task 6. Complete the GainInfoItem class

Implement all the methods in GainInfoItem, except for the toString() method which is already provided to you. Implementing GainInfoItem is an easy task.

Task 7*. Complete the `InformationGainEvaluator` class

Follow the instructions given earlier about the calculation of information gain scores to complete the static `InformationGainCalculator.calculateAndSortInformationGains(...)`. In the `InformationGainEvaluator` class, you will also find a `main(...)` method. This `main(...)` method is provided in full. You do not need to make any changes to it. The name of the CSV file to process is given as a command-line parameter. Specifically, the following command line should be used to run A3:

```
java InformationGainCalculator <name of your file>
```

Example usage:

```
java InformationGainCalculator weather-nominal.csv
```

Please feel free to alter the `main(...)` method as you debug and test different parts of your implementation. **However, when you submit Assignment 3, please make sure to include the `main(...)` method that we provide.**

Example Output

Example Output for Assignment 2

Assignment 2 is an intermediate step towards Assignment 3. In Figure 12, we show the `VirtualDataSet.main(...)` method, recreating the conceptual illustrations of Figure 5 and Figure 9.

The output generated by `VirtualDataSet.main(...)` is shown in Figure 13. Before you start your own implementation, please carefully examine this output as well as the remarks made over the figure.

Example Output for Assignment 3

For A3, we provide four datasets. Three of these (`weather-nominal.csv`, `weather-numeric.csv`, and `credit-info.csv`) you have seen before. There is one new dataset, `diabetes.csv`, which is concerned with the prediction of diabetes in patients. We expect that your implementation of `InformationGainCalculator.main(...)` will produce results for all four datasets. The outputs for `weather-nominal.csv` and `weather-numeric.csv` are shown in Figures 14 and 15, respectively. The other two datasets are provided to you so that you can validate the performance of your solution. We will use `credit-info.csv` and `diabetes.csv` during the marking of A3. The output for these two datasets is thus withheld.

When we mark Assignment 3, the teams whose implementations produce a fully correct output for `credit-info.csv` and `diabetes.csv` will get a 10% bonus on their Assignment 3 mark!

Academic Integrity

This part of the assignment is meant to raise awareness concerning plagiarism and academic integrity. Please read the following documents.

- Academic regulation I-14 - Academic fraud
- Academic integrity

Cases of plagiarism will be dealt with according to the university regulations. By submitting this assignment, you acknowledge:

1. I have read the academic regulations regarding academic fraud.

2. I understand the consequences of plagiarism.

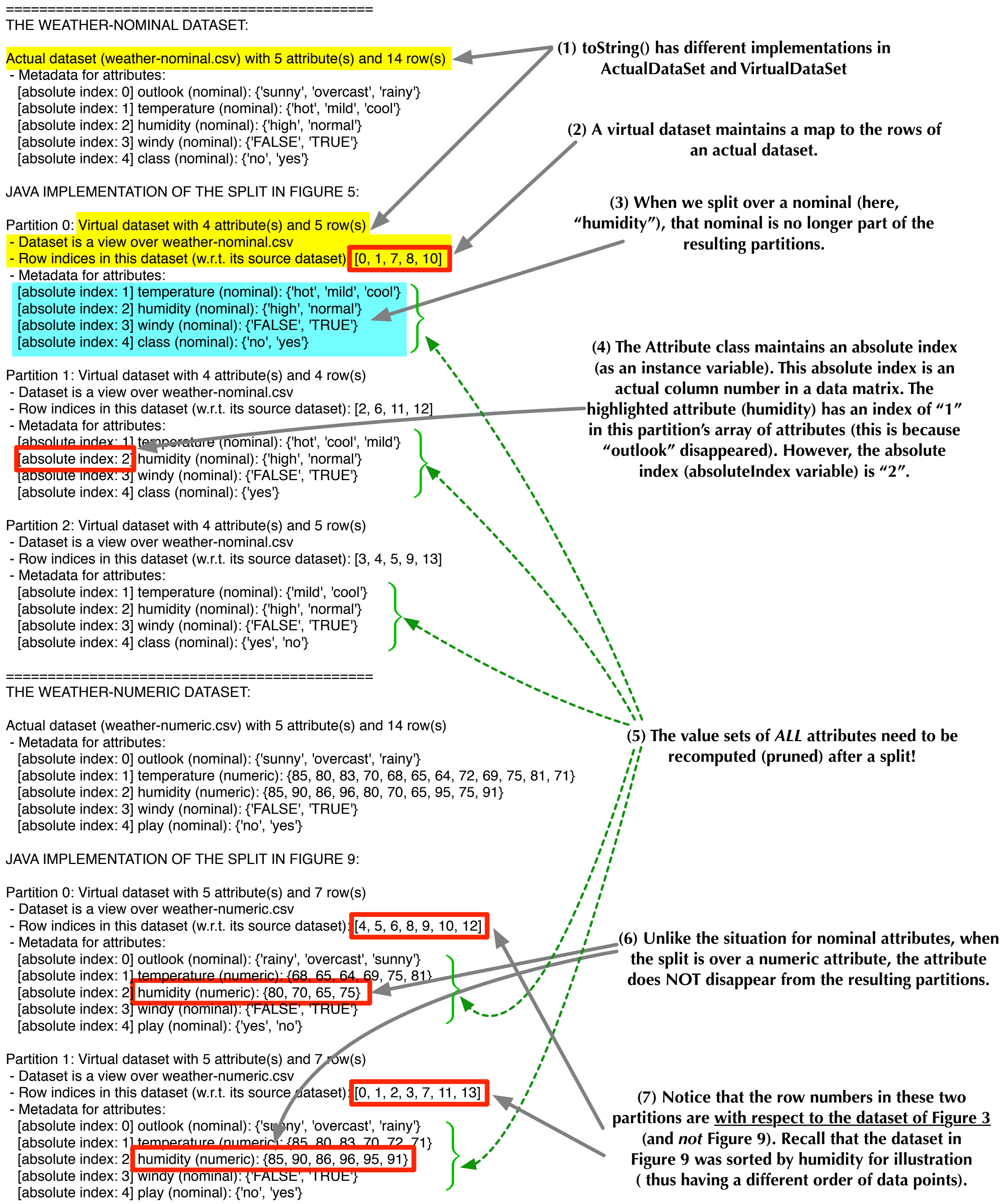
3. With the exception of the source code provided by the instructors for this course, all the source code is mine.

4. I did not collaborate with any other person, with the exception of my partner in the case of team work.

    - If you did collaborate with others or obtained source code from the Web, then please list the names of your collaborators or the source of the information, as well as the nature of the collaboration. Put this information in the submitted README.txt file. Marks will be deducted proportional to the level of help provided (from 0 to 100%).

```
public static void main(String[] args) throws Exception {
        // StudentInfo.display();
        System.out.println("===========================================");
        System.out.println("THE WEATHER-NOMINAL DATASET:");
        System.out.println();

        ActualDataSet figure5Actual = new ActualDataSet(new CSVReader("weather-nominal.csv"));

        System.out.println(figure5Actual);

        VirtualDataSet figure5Virtual = figure5Actual.toVirtual();

        System.out.println("JAVA IMPLEMENTATION OF THE SPLIT IN FIGURE 5:");
        System.out.println();

        VirtualDataSet[] figure5Partitions = figure5Virtual
                        .partitionByNominallAttribute(figure5Virtual.getAttributeIndex("outlook"));

        for (int i = 0; i < figure5Partitions.length; i++)
                System.out.println("Partition " + i + ": " + figure5Partitions[i]);

        System.out.println("===========================================");
        System.out.println("THE WEATHER-NUMERIC DATASET:");
        System.out.println();

        ActualDataSet figure9Actual = new ActualDataSet(new CSVReader("weather-numeric.csv"));

        System.out.println(figure9Actual);

        VirtualDataSet figure9Virtual = figure9Actual.toVirtual();

        // Now let's figure out what is the index for humidity in figure9Virtual and
        // what is the index for "80" in the value set of humidity!

        int indexForHumidity = figure9Virtual.getAttributeIndex("humidity");
        Attribute humidity = figure9Virtual.getAttribute(indexForHumidity);
        String[] values = humidity.getValues();

        int indexFor80 = -1;
        for (int i = 0; i < values.length; i++) {
                if (values[i].equals("80")) {
                        indexFor80 = i;
                        break;
                }
        }

        if (indexFor80 == -1) {
                System.out.println("Houston, we have a problem!");
                return;
        }

        VirtualDataSet[] figure9Partitions = figure9Virtual
             .partitionByNumericAttribute(indexForHumidity, indexFor80);

        System.out.println("JAVA IMPLEMENTATION OF THE SPLIT IN FIGURE 9:");
        System.out.println();

        for (int i = 0; i < figure9Partitions.length; i++)
                System.out.println("Partition " + i + ": " + figure9Partitions[i]);
}
```

Figure 12: Code for performing the splits shown in Figures 5 and 9



```
============================================
THE WEATHER-NOMINAL DATASET:

Actual dataset (weather-nominal.csv) with 5 attribute(s) and 14 row(s)
 - Metadata for attributes:
   [absolute index: 0] outlook (nominal): {'sunny', 'overcast', 'rainy'}
   [absolute index: 1] temperature (nominal): {'hot', 'mild', 'cool'}
   [absolute index: 2] humidity (nominal): {'high', 'normal'}
   [absolute index: 3] windy (nominal): {'FALSE', 'TRUE'}
   [absolute index: 4] class (nominal): {'no', 'yes'}

JAVA IMPLEMENTATION OF THE SPLIT IN FIGURE 5:

Partition 0: Virtual dataset with 4 attribute(s) and 5 row(s)
 - Dataset is a view over weather-nominal.csv
 - Row indices in this dataset (w.r.t. its source dataset): [0, 1, 7, 8, 10]
 - Metadata for attributes:
   [absolute index: 1] temperature (nominal): {'hot', 'mild', 'cool'}
   [absolute index: 2] humidity (nominal): {'high', 'normal'}
   [absolute index: 3] windy (nominal): {'FALSE', 'TRUE'}
   [absolute index: 4] class (nominal): {'no', 'yes'}

Partition 1: Virtual dataset with 4 attribute(s) and 4 row(s)
 - Dataset is a view over weather-nominal.csv
 - Row indices in this dataset (w.r.t. its source dataset): [2, 6, 11, 12]
 - Metadata for attributes:
   [absolute index: 1] temperature (nominal): {'hot', 'cool', 'mild'}
   [absolute index: 2] humidity (nominal): {'high', 'normal'}
   [absolute index: 3] windy (nominal): {'FALSE', 'TRUE'}
   [absolute index: 4] class (nominal): {'yes'}

Partition 2: Virtual dataset with 4 attribute(s) and 5 row(s)
 - Dataset is a view over weather-nominal.csv
 - Row indices in this dataset (w.r.t. its source dataset): [3, 4, 5, 9, 13]
 - Metadata for attributes:
   [absolute index: 1] temperature (nominal): {'mild', 'cool'}
   [absolute index: 2] humidity (nominal): {'high', 'normal'}
   [absolute index: 3] windy (nominal): {'FALSE', 'TRUE'}
   [absolute index: 4] class (nominal): {'yes', 'no'}

============================================
THE WEATHER-NUMERIC DATASET:

Actual dataset (weather-numeric.csv) with 5 attribute(s) and 14 row(s)
 - Metadata for attributes:
   [absolute index: 0] outlook (nominal): {'sunny', 'overcast', 'rainy'}
   [absolute index: 1] temperature (numeric): {85, 80, 83, 70, 68, 65, 64, 72, 69, 75, 81, 71}
   [absolute index: 2] humidity (numeric): {85, 90, 86, 96, 80, 70, 65, 95, 75, 91}
   [absolute index: 3] windy (nominal): {'FALSE', 'TRUE'}
   [absolute index: 4] play (nominal): {'no', 'yes'}

JAVA IMPLEMENTATION OF THE SPLIT IN FIGURE 9:

Partition 0: Virtual dataset with 5 attribute(s) and 7 row(s)
 - Dataset is a view over weather-numeric.csv
 - Row indices in this dataset (w.r.t. its source dataset): [4, 5, 6, 8, 9, 10, 12]
 - Metadata for attributes:
   [absolute index: 0] outlook (nominal): {'rainy', 'overcast', 'sunny'}
   [absolute index: 1] temperature (numeric): {68, 65, 64, 69, 75, 81}
   [absolute index: 2] humidity (numeric): {80, 70, 65, 75}
   [absolute index: 3] windy (nominal): {'FALSE', 'TRUE'}
   [absolute index: 4] play (nominal): {'yes', 'no'}

Partition 1: Virtual dataset with 5 attribute(s) and 7 row(s)
 - Dataset is a view over weather-numeric.csv
 - Row indices in this dataset (w.r.t. its source dataset): [0, 1, 2, 3, 7, 11, 13]
 - Metadata for attributes:
   [absolute index: 0] outlook (nominal): {'sunny', 'overcast', 'rainy'}
   [absolute index: 1] temperature (numeric): {85, 80, 83, 70, 72, 71}
   [absolute index: 2] humidity (numeric): {85, 90, 86, 96, 95, 91}
   [absolute index: 3] windy (nominal): {'FALSE', 'TRUE'}
   [absolute index: 4] play (nominal): {'no', 'yes'}
```

(1) toString() has different implementations in ActualDataSet and VirtualDataSet

(2) A virtual dataset maintains a map to the rows of an actual dataset.

(3) When we split over a nominal (here, "humidity"), that nominal is no longer part of the resulting partitions.

(4) The Attribute class maintains an absolute index (as an instance variable). This absolute index is an actual column number in a data matrix. The highlighted attribute (humidity) has an index of "1" in this partition's array of attributes (this is because "outlook" disappeared). However, the absolute index (absoluteIndex variable) is "2".

(5) The value sets of *ALL* attributes need to be recomputed (pruned) after a split!

(6) Unlike the situation for nominal attributes, when the split is over a numeric attribute, the attribute does NOT disappear from the resulting partitions.

(7) Notice that the row numbers in these two partitions are <u>with respect to the dataset of Figure 3</u> (and *not* Figure 9). Recall that the dataset in Figure 9 was sorted by humidity for illustration (thus having a different order of data points).

Figure 13: Output from `VirtualDataSet.main(...)`; you are encouraged to carefully examine this output as well as the remarks on the right side of the figure before you start with your own implementation.

```
 *** items represent (attribute name, information gain) in descending order of gain value ***

(outlook, 0.246750)
(humidity, 0.151836)
(windy, 0.048127)
(temperature, 0.029223)
```

Figure 14: A3 Output for weather-nominal.csv

```
 *** items represent (attribute name, information gain) in descending order of gain value ***

(outlook, 0.246750)
(humidity[split at <= 80], 0.151836)
(temperature[split at <= 83], 0.113401)
(windy, 0.048127)
```

Figure 15: A3 Output for weather-numeric.csv

Rules and regulation

- Follow all the directives available on the assignment directives web page.
- Submit your assignment through the on-line submission system virtual campus.
- You must preferably do the assignment in teams of two, but you can also do the assignment individually.
- You must use the provided template classes below.
- If you do not follow the instructions, your program will make the automated tests fail and consequently your assignment will not be graded.
- We will be using an automated tool to compare all the assignments against each other (this includes both, the French and English sections). Submissions that are flagged by this tool will receive the grade of 0.
- It is your responsibility to make sure that BrightSpace has received your assignment. Late submissions will not be graded.

Submission of Assignment 2

You must hand in a zip file (no other file format will be accepted). The name of the top directory has to have the following form: **a2_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment (simply repeat the same number if your team has one member). The name of the folder starts with the letter "a" (lowercase), followed by the number of the assignment, here 2. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a2_3000000_3000001.zip contains the files that you can use as a starting point. Your A2 submission must contain the following files.

- README.txt
  - A text file that contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- AttributeType.java, DataReader.java, DataSet.java, Util.java[5]
- CSVReader.java
- Attribute.java
- ActualDataSet.java
- VirtualDataSet.java
- StudentInfo.java (update the file so that the display() method shows your personal information).

[5] You are **not** supposed to change any of these files; you simply resubmit what is given to you.

Submission of Assignment 3

You must hand in a zip file (no other file format will be accepted). The name of the top directory has to have the following form: **a3_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment (simply repeat the same number if your team has one member). The name of the folder starts with the letter “a” (lowercase), followed by the number of the assignment, here 3. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. **There is no separate template code provided for A3.** Your A3 submission must contain the following files.

- README.txt
  - A text file that contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- AttributeType.java, DataReader.java, DataSet.java, Util.java[5]
- CSVReader.java (from A2)
- Attribute.java (from A2)
- ActualDataSet.java (from A2)
- VirtualDataSet.java (from A2)
- EntropyEvaluator.java (**new in A3**)
- GainInfoItem.java (**new in A3**)
- InformationGainCalculator.java (**new in A3**)
- StudentInfo.java (update the file so that the `display()` method shows your personal information).

Questions

For all your questions, please visit the Piazza Web site for this course:

- `https://piazza.com/uottawa.ca/winter2021/iti1121/home`

References

[1] Ian Witten, Eibe Frank, Mark Hall, and Christopher Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 4th edition, 2016.

A Shallow copy versus Deep copy

As you know, objects have variables which are either a primitive type, or a reference type. Primitive variables hold a value from one of the language primitive type, while reference variables hold a reference (the address) of another object (including arrays, which are objects in Java).

If you are copying the current state of an object, in order to obtain a duplicate object, you will create a copy of each of the variables. By doing so, the value of each instance primitive variable will be duplicated (thus, modifying one of these values in one of the copy will not modify the value on the other copy). However, with reference variables, what will be copied is the actual reference, the address of the object that this variable is pointing at. Consequently, the reference variables in both the original object and the duplicated object will point at the same address, and the reference variables will refer to the same objects. This is known as a **shallow** copy: you indeed have two objects, but they share all the objects pointed at by their instance reference variables. The Figure A provides an example: the object referenced by variable **b** is a shallow copy of the object referenced by variable **a**: it has its own copies of the instances variables, but the references variables **title** and **time** are referencing the same objects.

Often, a shallow copy is not adequate: what is required is a so-called **deep** copy. A deep copy differs from a shallow copy in that objects referenced by reference variable must also be recursively duplicated, in such a way that when the initial object is (deep) copied, the copy does not share any reference with the initial object. The Figure A provides an example: this time, the object referenced by variable **b** is a deep copy of the object referenced by variable **a**: now, the references variables **title** and **time** are referencing different objects. Note that, in turn, the objects referenced by the variable **time** have also been deep-copied. The entire set of objects reachable from **a** have been duplicated.

You can read more about shallow versus deep copy on Wikipedia:

- Object copying

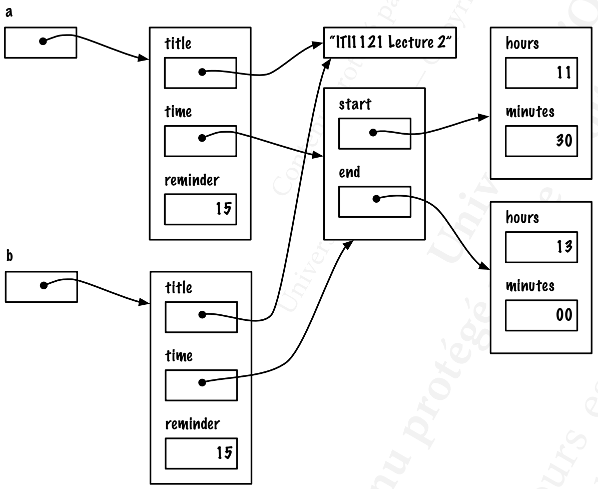


Figure 16: A example of a shallow copy of objects.

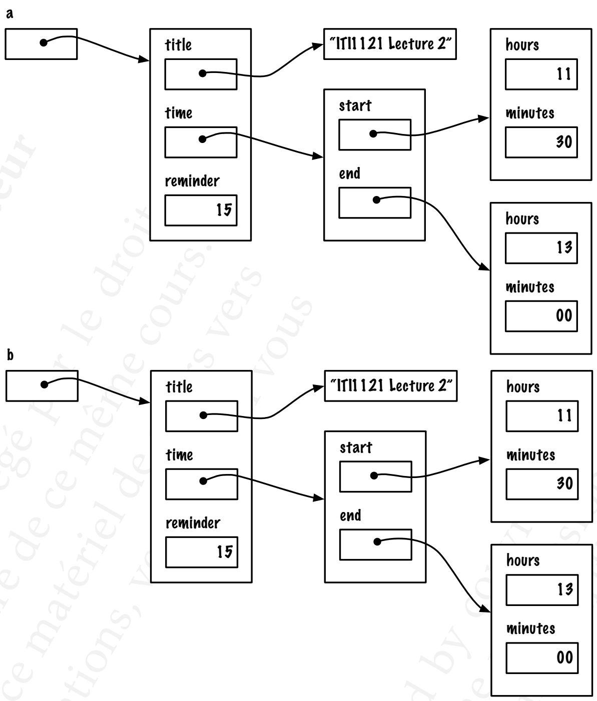


Figure 17: A example of a deep copy of objects.

B Enum

In this application, we have two types of attributes: *NOMINAL* and *NUMERIC*. There are several ways to achieve this, but in this assignment we are going to use Java's *Enum* type.

Some programmers use values of type **int** to represent symbolic constants in their programs. For example, to represent the day of the week or the month of the year.

```
public class E1 {

    public static final int MONDAY = 1;
    public static final int TUESDAY = 2;
    public static final int SUNDAY = 7;

    public static final int JANUARY = 1;
    public static final int FEBRUARY = 2;
    public static final int DECEMBER = 12;

    public static void main(String[] args) {

        int day = SUNDAY;

        switch (day) {
            case MONDAY:
            System.out.println("sleep");
            break;

            case SUNDAY:
            System.out.println("midterm test");
            break;

            default:
            System.out.println("study");
        }
    }
}
```

Using constants, such as MONDAY and JANUARY, improves the readability of the source code. Compare "if (day

== MONDAY) { . . . }” to “if (day == 1) { . . . }”. It is one step in the right direction.

However, since all the constants are integer values, there are several kinds of errors that the compiler cannot detect. For example, if the programmer uses the same number for two constants, the compiler would not be able to help, 7 is valid value for both SATURDAY and SUNDAY:

```
public static final int SATURDAY = 7;
public static final int SUNDAY = 7;
```

But also, assigning a value representing a month to variable representing a day of the week would not be detected by the compiler, both are of type int:

```
int day = JANUARY;
```

Enumerated types have the same benefits as the symbolic constants above, making the code more readable, but in a typesafe way.

```
public class E2 {

    public enum Day {
        MONDAY, TUESDAY, SUNDAY
    }

    public enum Month {
        JANUARY, FEBRUARY, DECEMBER
    }

    public static void main( String[] args ) {

        Day day = Day.MONDAY;

        switch (day) {
            case MONDAY:
            System.out.println( "sleep" );
            break;

            case SUNDAY:
            System.out.println( "midterm test" );
            break;

            default:
            System.out.println( "study" );
        }
    }
}
```

In the above program, each constant has a unique value. Furthermore, the statement below produces a compile time error, as it should:

```
Day day = Month.JANUARY;
```

```
Enum.java:36: incompatible types
found   : E2.Month
required: E2.Day
        Day day = Month.JANUARY;
                       ^
1 error
```

- https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

Last modified: February 8, 2021