Instituto Superior Técnico

Object Oriented Programming

Electrical and Computer Engineering

Tree augmented naive Bayes classifier
Final Report

Group 9

Authors	*Students number*
David Souto	86966
Gonçalo Tomás	87009
Marcelo Jacinto	87063

Year 2019/2020 - 2nd semester

1 Introduction

The aim of this project was to develop a Tree Augmented Naive Bayes Classifier using a polynomial-time bounded approach focusing only on tree-like network structures. The project was developed in java making use of only the resources that the language provides. The main goal was to make the project as modular and extensible as possible without compromising on performance.

1.1 Notes on the notation and figures used

In this report we use some figures to better visualize the though process behind the implemented algorithm. These figures do not replace in any way the project's UML, and their sole purpose is to help visualize how the project is divided.

During this project we make use of the name **category**, any time we refer to the attributes or features, such as X1, being the **labels** of this category the different values that it can assume, for example, 0, 1, 2... In addition we consider that a **classification** also has labels, i.e, the values that a class can assume, for example 0, 1, 2 in case of numeric features or "cat"/"dog" in the case of categorical features.

2 Project Solution and Extensibility

The classification algorithm proposed was targeted only at categorical features represented by numbers that ranged from 0 to N. This approach can be limiting in real life problems, so in our implementation we accept labels that are not necessarily numbers but also letters or any other value that can be represented by a string.

As suggested by figure 1, the project implementation was divided into the following big main parts:

- A parser to read data from csv files with training and test data;
- A data-model responsible for saving the useful data for the classifier based on real world observation data;
- A graph library to be used when defining the structure of the classifier;
- A score to be used to evaluate the relation between features;
- A classifier to make the predictions;
- Metrics to evaluate the performance of the classifier.

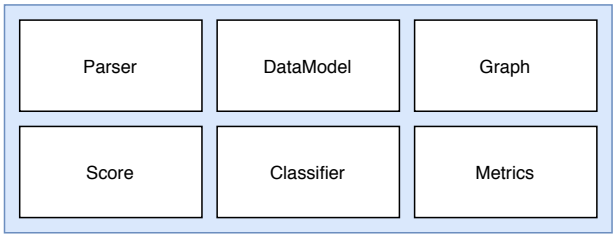

Figure 1: Global view of the solution

2.1 CSV Parser

The CSV parser is implemented in the **parser package**. The **DataParser** class is abstract and contains only the static **readFile** method. The goal was to make the parser only responsible for reading the data from the file without making any decision on how to store that data. Therefore it would not make sense having to instantiate a class to be able to invoke the **readFile** method, when it does not save any data.

In order for this approach to work, the parser must receive a string with the name of the file to open, a string with a split symbol and a **DataHolder** object that is responsible for implementing the methods: **readLabels** and **readData**.

Upon reading the first line of a file, the **readLabels** method provided by the **DataHolder** object is called, passing an array of String with the content of the line read as arguments. For every other single line read, the method **readData** is then called.

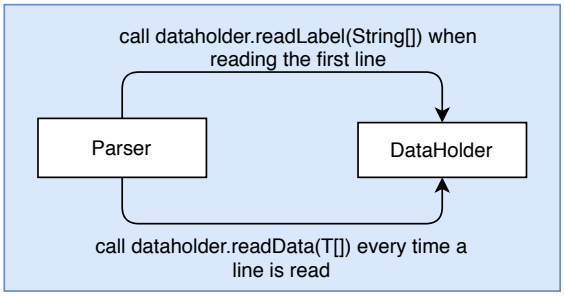

Figure 2: Parser Overview

This model was inspired by the **Bridge**, **Facade** and **Adapter** design patterns. Inside a subpackage **facade** we have 4 different implementations for the **DataHolder** interface. Such implementations comprise:

- **TrainingModelFacade** - a **DataHolder** that updates a model with counts every time a line is read from the file, without the need to store the actual read data;
- **OnlinePredictingFacade** - call a classifier to make a prediction every time a line is read;
- **TrainingDataHolderFacade** - a "container" that saves the data being read into a list of arrays of String of data and labels (like a typical CSV reader);
- **TesteDataHolderFacade** - also a "container" that saves the data being read into a list of arrays of String but separates the last column from the rest of the data (the column that contains the classifications);

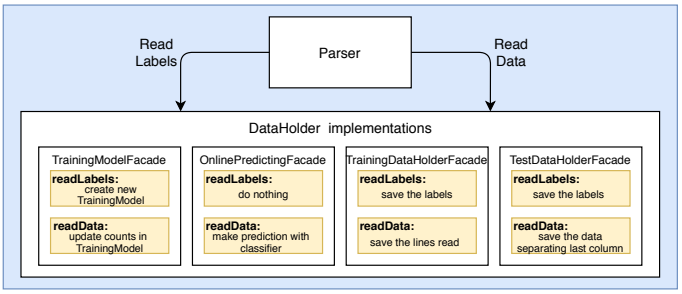

Figure 3: Parser Facades overview

The first 2 facades enable training and making predictions in an online modality. The last 2 enable the user to have a "batch" of data that can later be used to train/test a classifier.

By default, all the "facades" provided make use of the data read from the parser in String form and no conversions to other data-types are made. This implies that the **DataModel** and **Classifier** used are instantiated with String. The downside of using String is the increase in memory usage, but the upside is the ability to use features/categories that are not necessarily numeric.

Advantages of this approach

- Using this approach, we can separate the act of reading from a file from the act of saving/interpreting the data being read. This provides flexibility to the implementation as we have built the foundations for the creation of "wrapper classes" that can be used to have different behaviors as lines are read.

What could be done differently

- Instead of implementing the "facades" we had the option of just making the **Classifier** and **DataModel** extend the **DataHolder** interface. We believe this approach would not be as clean as using "wrapper classes" because the **Classifier** and **DataHolder** interfaces would lose some of their independence from the parser implementation.

2.2 Data-model

The **datamodel package** implements the classes necessary to save the counts extracted from training data. All the interfaces and classes in this package make use of generic types to save the labels of data for each category. Since we are considering that the categories/features might not be numeric we must save, besides the number of combinations in which they appear in, their own values (labels). Using this approach we are able to save labels that are numbers, characters or strings leaving the model general enough for usage with different types of training data.

We start by implementing a **Model** interface responsible for providing the functionality of saving labels and getting their index, without enforcing the usage of any particular data structure to save them. We provide both a **Category** interface and a **Classification** interface, specialized at saving labels for a given category/feature and at saving the classifications. Some of the methods provided by these interfaces are to increment/get the N_{ijkc}, N_{ikc} and N_{ijc} counts as well as increment/get the N_c count respectively.

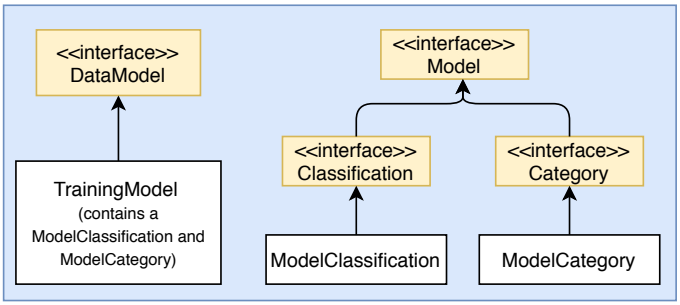

Figure 4: DataModel overview

This package also provides implementations for the interfaces described above: **ModelCategory** and **ModelClassification**. The **ModelCategory** extends the **Category** interface. In this implementation, if a label read never appeared before, it is saved in a list of "labels". There are also 2 arrays of hash-maps to save the N_{ijkc} and N_{ijc} counts. In this case, each position of the array a different category as being the parent of the current category. In this case, arrays can be used because it is known a-priori how many features/categories the data has. In addition to these data-structures there is also an hash-map to save the N_{ikc} counts. These hash-maps use as keys a String that result from the concatenation of "i,j,k,c", "i,j,c" and "i,k,c" values. The **ModelClassification** implements the **Classification** interface and similar to the **ModelCategory class**, it saves the classifications that never appeared before in a list and saves the counts for each of those classifications in an hash-map.

At a higher level we define a **DataModel** interface responsible for providing the user with some functionality regarding updating counts related to categories, classes and retrieving them, once again, without enforcing any particular structure for the data. We also provide a **TrainingModel** class that implements the **DataModel** interface. This class contains a **ModelClassification** and **ModelCategory** attributes in order to save data related to both the categories and the classifications.

Advantages of this approach

- The implementation makes extensive use of generic types and has many layers of abstraction that makes it easy to create classes that implement **Classification** and **Category** using different data structure than the ones chosen. It also gives the possibility of combining them in different ways to create a **DataModel**.

What could be done differently

- Constructing the key to access an hash-map involves getting the index of a given label in the list of labels. For this purpose, the method indexOf provided by java lists is used. This approach leads to a simple implementation but leads to a time complexity in the worst case scenario of O(N) being N the labels each category can assume. This is not a critical factor as we are not considering problems with many features/categories with many labels per

feature, but it should be subject for improvement if that was the case. A possible solution would be using 2 hash-maps in place of the list of labels. With that approach access times would be on average O(1) but it would require more memory to save the labels.

2.3 Graph library

The **graph package** contains a very simple implementation of a weighted graph using adjacency lists. The implementation can be summarized by figure 5. While implementing this package, the focus was on creating graphs that are weighted and do not need to save data inside the nodes. This choice was deliberate as the focus of the project was only on the graph structure. We could have made a more general graph implementation that saved "generic" data in the nodes, but that would add unneeded complexity for the problem.

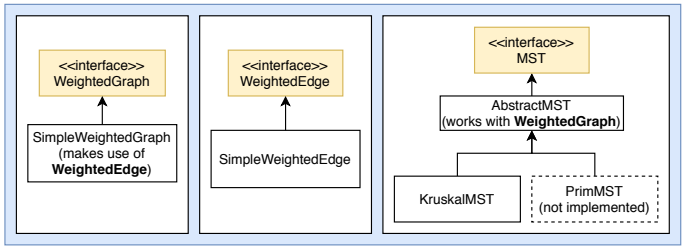

Figure 5: Graph Model overview

Even thought this graph library is very limited it provides an interface (**WeightedGraph**) that exposes the methods that a graph should offer such as *addEdge, removeEdge, getAllEdges*, among others. This graph interface is suited toward a graph implemented using adjacency lists, as it offers a method that returns all the edges that it contains. In order to counterbalance this limitation we try to make the graph more open in other areas such as the weights of the edges. In this interface we assume that the weight used can be of **generic type** as long as it **extends the Comparable interface**. This gives some freedom for developers to create weighted graph with heuristics where weights can be something more complex than just a simple value.

This package also contains a **search sub-package** in which there is an interface, **MST** and the class **AbstractMST**. Given a **WeightedGraph**, the class provides the functionally of calculating the Minimum/Maximum spanning tree and return it in an array form. It also contains a class that extends **AbstractMST** that calculates the Maximum Spanning Tree of the graph that is passed to the constructor using the Kruskal algorithm. Following this approach makes it possible to easily implement other MST algorithms such as Prim. The biggest downside regarding this implementation is the fact that the MST algorithms must work with graph implementations that extends **WeightedGraph** if they want to extend from **AbstractMST**. This is a limitation of the search algorithm as we need to infer which kind of structure the graph is built on top of (adjacency list vs matrix). Otherwise, they can always implement the interface **MST** which makes no assumption of the type of graph used.

Advantages of this approach

- Allows for graphs with weights that are more complex than just a number (as long as the weight Object extends Comparable);
- Separates the Maximum Spanning Tree from the graph implementation itself, allowing for different algorithms of MST calculation;

What could be done differently

- It would be useful to also have the possibility of having a graph that is not weighted;
- The MST algorithm only works with Graph classes that implement the **WeightedGraph** interface, which can be limiting as this interface provides a method that returns a list of **WeightedEdges**. This choice is limiting but we must keep in mind that the MST algorithm must assume something about the graph structure - if it is using a matrix or a list to save its edges.

- It is still possible to implement a graph that uses a matrix to save the edges and implements the **WeightedEdge** interface, but it would also require to translate "translate" that matrix into a list form when implementing the getConnected method.

2.4 Score

The model used for this Tree Augmented Naive Bayes classifier contains a structure learning step. This involves calculating a graph where each node is related to each other by a weight. There were 2 proposed ways of calculating those weights given counts extracted from training data: using a log-likelihood method or using a minimum-description length method. In order to have multiple ways of calculating a score, we believe that it was key to have a package specific for that need. Therefore, in the **score package**, one can find an interface called **Score**. That interface provides a method called score that receives 2 integer numbers referring the node to be considered, it's parent and a **DataModel** object. This **DataModel** object implements the necessary methods to retrieve the counts used in the score formula. This package also provides 2 classes: LLScore and MDLScore. The LLScore implements the Score interface and is extended by the MDLScore as it's formula uses parts of the LLScore formula.

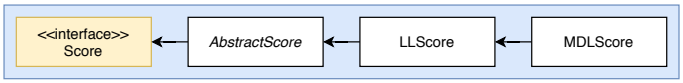

Figure 6: Score model overview

Advantages of this approach

- We have a solid foundation to implement other kinds of scores just by extending the **AbstractScore** class or by implementing the **Score** interface.
- It is possible to use the formula of LL or MDL in the calculation of a new score just by extending these classes.

What could be done differently

- The **Score** interface assumes that a given node has only one parent node in its score method - it is not extensible enough to consider multiple parents. Unfortunately we do not have a clear answer to this problem as we do not have enough knowledge regarding which other types of scores exist and which data they need for their calculations.
- A possible improvement to this implementation would be to use the design pattern of dependency injection on the constructor of the classes that implemented the Score interface. In this way it would not be necessary to pass the **DataModel** object through the score method every time we want to calculate a score but rather only once upon the construction of the Score object. This was not done because it would force the user to have a **DataModel** object by the time a **Score** object is instantiated. This would remove some of the flexibility this model has.

2.5 Classifier

In the **classifier package** is where the implementation of the classifier can be found. There are several layers of abstraction in this implementation in order to enable someone to implement new types of classifiers or extend the existing ones in a streamlined way.

Figure 7: Classifier model overview

We start by providing a **classifier** interface that provides only 3 methods that are common to any multi-variable classifier: *train, predictOne and predictMultiple*. In this interface the *predictOne* and *PredictMultiple* receive generic type and list of generic type as parameters and return generic type and list of generic type respectively, in order to leave the interface as open as possible.

In the next step we provide an abstract class called **Abstract Classifier** with the goal of providing an implementation for the *PredictMultiple* method by just calling the *predictOne* method for every element of the the list we want to make a prediction on. The goal here was avoid having every classifier implementing this method that should be the same for every single one.

The next layer is an abstract class named **AbstractBayes** responsible for layering the foundation for our classifier. This class has attributes such as a **Score** and a **DataModel**. These are set by constructor injection. That is, this particular classifier will never receive the training data directly. It receives a **DataModel** which already provides the functionality to access the relevant counts previously extracted from training data. The classifier itself also never defines the **score** that will be used.

At its lowest level, this package provides the **TreeAugmentedNayveBayes** class that extends the **AbstractBayes** class. In this concrete implementation we enforce that this algorithm will use an array of integers to save the tree structure of the classifier and hash-maps to save the values of θ_c and θ_{ijkc}. In this class the methods of predict and train are actually implemented according to the algorithm described in the assignment. We use the weighted graph package to develop a graph with the same number of nodes as the number of features. In addiction, the score package was used to calculate the scores between the nodes. Then, the KruskalMST is used to calculate the maximum spanning tree for the graph, defining the structure of the classifier. After this step we calculate the probabilities. In this class the predictOne method is also implemented, according to the data-structures defined.

Advantages of this approach

- Code is modular, and the user can decide which class to extend from, depending on how different the classifier is, from the one implemented. That is, if one would like to implement a Tree Augmented Naive Bayes that actually saved a tree structure with data, one could just extend the **AbstractBayes** class. But if one would like to make a classifier that had a graph structure, then one could simply extend the **AbstractClassifier** interface.
- The **DataModel** and **Score** to be used with this classifier is chosen by the user at the time of the object creation and not pre-defined a-priori.

What could be done differently

- There is no option to directly "feed" the classifier with a list with data to train. In order to do that, the only way would be through a "wrapper class" that would need to define a priori a **Score** and **DataModel**. We did not implement that, as it was not needed for the project but would be a nice addition.

2.6 Metrics

In order to evaluate the performance of any classifier it is necessary to have metrics that can compare predicted values vs real values. We can divide metrics into 2 different sets: one that consist of only one score for the combination of all classes (ex: accuracy score) and others that consider multiple scores - one score per class (ex: sensitivity, specificity scores and F1-scores). In addition, these last scores need a confusion matrix as an auxiliary structure to make the calculations.

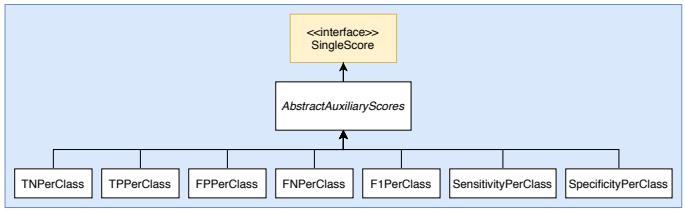

Figure 8: Single metric model overview

In order to model the problem above we start by creating a sub-package named **perclass** that has a **SingleScore** interface. This interface exposes a method that calculates a score for a given

class i. We then implement an abstract class named **AbstractAuxiliaryScores** that receives a confusion matrix. This confusion matrix fundamental to calculate the individual scores. Then we define a few classes that extend the latter, such as classes to calculate true positives, true negatives, false positives, false negatives, f1 score, sensitivity, specificity for individual classes i.

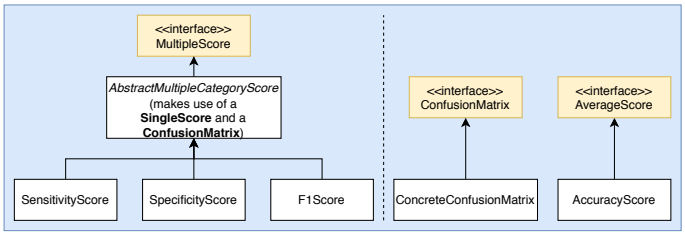

Figure 9: Multiple metric model overview

On a higher level we defined a **MultiScore** interface that exposes 2 methods: one that returns a list of scores and another that returns a list of the corresponding labels for each score. Then we define another layer - the **AbstractMultipleCategoryScore**, an abstract class responsible for receiving a **ConfusionMatrix** object and calculate the list of scores. This class also contains as attribute a SingleScore that will correspond to the one we want to use to calculate the score for all the classes. This class implements the method to calculate the score itself, making use of the methods provided by SingleScore and **ConfusionMatrix** objects.

In the end we implement the **SensitivityScore**, **SpecificityScore** and **F1Score** classes which are basically wrapper classes whose sole purpose are to "inject" the expected **SingleScore** and the **ConfusionMatrix** into the **AbstractMultipleCategoryScore** class.

Advantages of this approach

- We are able to implement the calculation of a score for a single class and then create a "wrapper" that is responsible for "feeding" that new score to an abstract class that is already prepared to make the calculations for all the classes that were observed.

What could be done differently

- Both the **SingleScore** and the **AbstractMultipleCategoryScore** need a confusion matrix to make their calculations. A "bad behaved" user might create its own "wrapper class" and provide a different confusion matrix to the **SingleScore** and **AbstractMultipleCategoryScore**, leading to inconsistencies in the results. Unfortunately we were not able to solve this problem, as it would mean that we had to instantiate the SingleScore directly into the AbstractMultipleCategoryScore, losing the ability to "inject" the desired score on demand, losing flexibility as a consequence.

2.7 Exceptions and Error Handling

During the implementation of the program, 3 custom exceptions were created to signal potential errors that could occur during the program execution. The 3 main errors considered were empty Strings that should be considered as invalid labels, invalid number of categories/features and repeated categories/features (it does not make sense to have more than one X1 feature).

There are several method across the all program that throw exceptions if the parameters received do not follow certain conditions. For example: if the number of elements in a list of predictions and a list of true values is different, if the number of labels used to produce counts is different than the one specified in the beginning of the program,...

In the main method and the "facade"/"wrappers" for the parser some try/catch blocks are introduced to handle exceptions that might be thrown when training or classifying an object in order not to let the main program crash.

Expected behavior when data is corrupted

- If the program fails to open the training file, it will exit and display a message to inform the user.
- If the first line of the training file contains repeated features or labels (ex. X1 X1 X3), the program will also exit and display a message to inform the user.
- Empty lines in the file are completely ignored as well as lines only filled with commas.
- Lines with the wrong number of elements are ignored during training or evaluation and a message is shown in the console with the faulty line and it's content.
- The program will try to train/classify every single line of the files. If each line is corrupted, there will be a number of warnings to the user concordant with the number of lines corrupted.
- If the program fails to open the testing file, but succeed to open the training file, it will display the generated tree and, only after, an error message regarding the test file and how it could not be open.

2.8 Utilitarian functions

In this program we also provide an **utils package** that contains the **Utils** abstract class. This class implements some abstract method that do not really belong anywhere specific in the program and that are essentially auxiliary functions. An example is the findMaxIndex method that given an array of doubles finds the index with the highest value.

3 Putting All Together and Evaluating the Performance

In order to put the different pieces of the program together we decided to use a combination of **TrainingModelFacade** and the **OnlinePredictingFacade** with the parser. With this approach we do not need to store the training and test data in memory and can just update the counts/make classifications as each line is read by the parser. This makes the reading process slower but makes the algorithm less memory hungry.

Side-note: We also provide a batch reading example in the package main/tests named BatchMain.java. This class is not protected by try/catch blocks and is only a demonstration of a different way of using the parser in conjunction with the classifier.

EXTRA FEATURE: If the program is provided with an extra argument "c", it will show the confusion matrix in the end. For example:

```
java -jar Project.jar heart-train.csv heart-test.csv LL c
```

Regarding the performance, the biggest bottleneck that concerns our particular implementation, is the fact we are using a list to save the labels from the data and using the position of elements in the list to use as keys for hash-maps. This is a key area that would benefit from improvement. Just like discussed previously, this is not a massive problem as the access times would have a complexity of O(N), with N being the number of labels per category/feature. There are other zones of the program, for example the score calculation that it's formula inherently has a higher complexity than O(N), which means that the total program complexity is not globally limited by this particular implementation.

4 Conclusion

To sum up, we divided the problem in 6 major parts that were implemented in their respective packages. We then proceeded with an overall description of the implementation. For each part of the algorithm it was explained with a description of the problems that arise from it. For some of those problems we propose concrete solutions that could be used in a version 2.0 of this program.

Modularity was a key factor in our implementation, leading to many packages following the structure of implementing an interface, then followed by an abstract class that implements the interface and finally a class that extends the abstract class. In addition, some common design patterns such as dependency injection where used.

In general we believe that our program is solid enough to allow its extension even though there are some small nuances that we were not able to overcome, just as described in the sections: "What could have been done differently".
The provided code was structured with high flexibility in mind such that in the future any user can import some packages for their own purpose. This extensibility is offered by the JAVA language through the usage of interfaces, classes and methods that are perfectly interconnected allowing the developer to re-use some algorithms in future projects.

References

[1] Slides OOP, Alexandra Carvalho

[2] https://refactoring.guru/design-patterns/