Instituto Superior Técnico - UL

Object Oriented Programming

2019/20

Tree augmented naive Bayes classifier

Authors:	*IST ID:*
Francisco Quelincho	86995
Guilherme Mascarenhas	87011
Ricardo Antão	87107

Group 14

Contents

1 Introduction

The Object Oriented Programming paradigm lies on the four pillars that are Abstraction, Encapsulation, Inheritance and Polymorphism. These four concepts aim at fixing the problems that "regular" programming can have if done improperly, which range from code duplication and possible security issues to being difficult to read and maintain. Abstraction keeps hidden what's meant to be hidden, Encapsulation bundles up attributes and methods in classes, Inheritance allows for easy code reusability and Polymorphism allows for easy functionality expansion.

1.1 Objectives

There are a list objectives we want to achieve in the end of this project

- Build a tree augmented Naive Bayes Classifier
- Use and understand all the concepts of OOP (mentioned in the Introduction)
- Have the code be extensible and robust at the same time.

2 Extensibility of solution

Extensibility is the measure of the ability to extend the capabilities of a system and the level of effort required without impairing the system. either by adding code to the already existing source code or altering the existing source code.

2.1 Places of extensibility

The code is extensible in 2 areas.

The first one is the scoring criterion used to calculate the weight of a connection between 2 nodes. If another criterion is to be used the user should create a class that extends the abstract class Edges and override the method *calcScore()* to calculate the desired criterion. It should also be added a new entry in the list of modes with the desired name so that the *InputHandler* creates an object of the right type.

The second area is the type of classifier to be used. Like in the last case the user should create a class that extends the abstract class *Classifiers* and override the method *predict()* so that it will classify using the right processes. It is to note that this new classifier might not even use the scoring package, it only takes the training and test data from the *InputHandler*.

2.2 Problems found

The program can run into some bugs all dependent on the input files.

- When the input files have invalid instances (for example having a name instead of a number in an instance)
- When the feature doesn't have instances for each value from 0 to (r_i -1) (for example it skips value 1 while having values of 2 and 0)
- If there's a value in the test data which doesn't appear in the training data (values outside range r_i -1)
- If the file are large enough that they do not fit in memory, our program will not work

2.3 Possible Improvements

The first problem would be solved by ignoring the invalid instances.

The second problem would involve altering the method from the class *InputHandler* called *getValuesUnique()* so that it returns a list with every value between 0 and (r_i -1) instead of a list with every unique value in the data.

Another improvement would be add methods to the *InputHandler* class so that he would be able to read files other than CSV files.

The last problem could also be addressed, although at a necessary cost. Instead of reading the file just once and storing the data in memory, we'd need to read the file twice. At first, we'd need to compute the unique values for each feature, in order to determine the size of N_{ijkc}. The second time reading the file would then be used to update the N_{ijkc} for each alpha, solving our problem. This would involve changes mainly to the InputHandler class, where it's functionality would need to be more closely linked to the alpha's computation step. This solution would be easier on memory, but would drastically slow down the program, given that access times to the hard disk is much slower than main memory. Ideally, the input files would come pre-processed, with the necessary unique values information, so that we would only need to read the file once.

Finally, yet another solution to the last problem would be to read the file just once, and grow the N_{ijkc} space as we see new, different values. This would benefit from the speed of our implementation, while being easier on memory and tolerant to large input data file, but it's a more complex implementation, which could lead to it being less modular and extensible.

3 Critical Performance Analysis

3.1 Main Data Structures

- InputHandler - Saves the input data into a $Map < String, ArrayList < Integer >>$ this was chosen due to being a really easy and intuitive way to access the data. An $ArrayList <>$ was chosen over a conventional array given it's robustness to variable size, since we don't need to know the final size it will take before having to allocate it.
- alpha - Structure that represents a connection between nodes (it possesses weight and N_{ijkc} counts)
- Edges - Saves the matrix of alphas (fully connected graph)
  An adjacency matrix was chosen for its simplicity.
- connection - Represents a connection in the graph (it possesses θ_{ijkc}, parent node an son node)
- NodeC - Represents the class node (it possesses θ_c)
- Graph - List of connections and a NodeC representing the class node.
  The $ArrayList <>$ was chosen due to possessing methods for iterating over it.
- BayesClassifier - Represents the classifier (it possesses the graph)

3.2 Time performance

Here we'll discuss the performance of the program in terms of time complexity during the various stages of it.

3.2.1 Scoring phase

In the scoring phase the program will first calculate the N_{ijkc} counts for every α. It will fill every entry of the adjacency matrix except the diagonal ($(\#features)^2 - \#features$ entries).
For each entry it will iterate through both features and the class instances simultaneously.

To calculate the weight of every α the program will run through the N_{ijkc} matrix which depends on the training data value range.

With the adjacency matrix constructed (fully connected graph created) the program will pass on to the next phase.

3.2.2 Building the Classifier phase

To build the DAG of the classifier we will use the Prim algorithm. For this we will be making use of the α matrix (adjacency matrix) and a Boolean vector, called *visited*, which indicates which nodes are already connected to the graph or not.

To add a node to the graph the program will run through the *visited* vector and for each node that is not connected to the graph the program checks weight of every α related to that node. In the end the connection with the biggest weight will be added to the graph.

When every node is inserted in the graph, a class node with the θ_c is created.With this, the DAG is finished. The root node is defined as the first node added to the graph. This ends the training process which varies from 20 ms to 50 ms. Logically this number will increase the more features and unique values are added to the training data.

3.2.3 Classification phase

In the classification phase for every instance the program will run through the graph as many times as there are possibilities of classes. While running through the graph it will pick the correspondent θ_{ijkc} to calculate the probability $P_B(C = i, X1, ..., Xn)$ .After, it will check which probability is higher and classify accordingly. The duration of this phase is of about 1 to 3 ms, which is to be expected since it only needs to run all the test iterations, a limited amount of times, which are much less than the training instances.

3.3 Space Performance

3.3.1 Scoring phase

In the scoring phase, like previously explained, the program will calculate the N_{ijkc} counts for every α. The number of existing α's is equal to $(\#features)^2$. The amount of memory used to store the N_{ijkc} counts depends on the number of possible parent configurations, the number of values the son can take and the number of classes. If by any chance one of these values increased the memory usage would also increase linearly.

3.3.2 Building the Classifier phase

In order to build the DAG, the program needs to store the values of θ_{ijkc} for each feature and the values of θ_c for the class node. Similarly to the Scoring phase, the memory usage depends on the

number of possible parent configurations, the number of values the son can take and the number of classes.

3.3.3 Classification phase

In the Classification phase the program receives data to test the classifier and elaborates a list of predictions for the class. The test data and the list of predictions are kept until the end of the program.

3.4 Possible Improvements

3.4.1 Time Performance Improvements

To improve the time complexity of the program we can eliminate the unnecessary iterations of the input data. One of this examples is while constructing the adjacency matrix. Due to the symmetry of the matrix, instead of calculating every entry of the matrix we only need to calculate the upper triangular half and copy it to the other half. Consequently reducing the computational time in half during the scoring phase.

The other improvement that could be made is in the Prim algorithm. The use of a adjacency matrix instead of adjacency list affects the computational time of the algorithm. Therefore a way to solve this problem would be to use the Prim algorithm with a adjacency list coupled with a binary heap, or even better, a Fibonacci heap.

There is another minor improvement that could be done in the classification phase. That is checking the biggest probability while calculating them.

3.4.2 Space Performance Improvements

As we've already described as a possible solution to one limitation of our implementation, we could change it so the input data wouldn't be stored in memory, such data that is easily what occupies the most space. The main disadvantage would be a loss in modularity, where the scoring and input handling would need to be more knitted together, as we've also mentioned before. Another possible improvement would be to break the test file into batches, and process one batch at a time.

4 Conclusion

We found this project to be a good introduction to OOP and a good way to use all the concepts of this programming branch.

Although our project is not perfect, it is still fairly efficient in terms of time performance, even though it takes its toll in space performance while being extensible to other types of classifiers or scoring criterion.

The problems in performance can be corrected but many of them will have advantages and disadvantages and we must seek balance on both types of performance.

In conclusion, the Naive Bayes classifier benefits immensely from an Object Oriented Programming approach, given the easy divisibility of the problem into sub-problems, that can be solved individually, bearing in mind the concept of encapsulation. The concept of abstraction helps us provide a framework where it's inner workings are hidden and secure as private methods and attributes, highlighting the high-level calls. Finally, inheritance and polymorphism allow for an easy redefinition and/or expansion of our current solution, for example, through the implementation of a more complex input data handler, or a new scoring method.