**The problem**

The main idea of the project is to create a chatbot able to classify some instances of a given problem, starting from some representative data. Given the data, the first task was to generate a decision tree that could be then used to establish the order of questions that need to be asked to the user in order to classify the given instance, where each attribute of the data set represents a different question.

For this task we analysed different algorithms, pointing our attention especially to the C4.5 algorithm and its improvement C5 version, two widely used data mining algorithms both deriving from the ID3 system. These algorithms are used to build a classification decision tree starting from a given training dataset. We focused primarily on them because they are able to deal with both numerical and categorical data and can also handle missing values. The C5 version improves performances and the size of the generated decision tree, adding also a boosted feature.

C4.5 has been already implemented in both the scikit-learn python library and in its java equivalent Weka under the J48 name. This already thoroughly tested implementation is the reason why we decided to use C4.5.

During its execution, in order to build the decision tree, the algorithm the algorithm chooses, for each node, which attribute to use in order to split the training data that reach the node using the information gain and entropy concepts. [*more detailed description of the algorithm*].

Since the given scenario was considering also a multi-class classification, we thought about using Meka, the multi label classification extension of the Weka framework. In Weka, C4.5 can handle classifications where multiple classes are present but only one is assigned to each instance during the process. In our case an instance could be labelled also with more than one class and this is considered by C4.5 as an entirely new class. Ultimately, the decision tree was correct also using this interpretation. On the other hand, Meka allows what is called multi-label classification, chaining a multi label classifier with a standard Weka one. Unfortunately, the multi label classifiers available did not match exactly the needs of the project since they often did not work correctly with the given data set, being too limited in terms of options, which data they can handle and, in some cases, the final result was the same achieved by J48. The “default interpretation” of J48 in the case of multiple classes labels was satisfiable and therefore we decided to keep the structure simple and apply just the original algorithm.

**How we used C4.5**

We started applying the algorithm to the training data, obtaining so in this way a decision tree. The tree is able to classify correctly all test instances in case all the attributes are categorical, while there is not this certainty in case of numerical attributes, since the chosen split point can misclassify some instances. [*more detailed description of the problem*]

After obtaining the decision tree we focused our effort in coding some export functions that allow us to export the tree in different formats (GraphML, dot and JSON), with an extra pruning ability, to use then the decision tree in the chatbot creation. The extra pruning ability was necessary because the algorithm, given an attribute, adds an edge labelled with any value that it is able to find for the given attribute in the whole dataset, although possibly no instances reach that node. The behaviour of the algorithm is intended to be exactly the one described above, since it would like to cover also cases not present in the dataset but that are a result of the combination of some values extracted from it. This is done to avoid overfitting, therefore a decision tree that is too specific to the training instances used and cannot handle cases that are not explicitly there. However, in our case, the training dataset contains exactly all possible combinations legal in the given domain and therefore cases that are not in it are impossible or not legal and therefore their deduction will only be an error. For this reason, the different export functions that were encoded, allow a parameter that enables the pruning of branches reached by no instances as soon as they are discovered. Summarizing, overfitting was for us a desired phenomenon.

At this point, the pruning feature introduced a new potential inconvenient. Some node could have most of their branches cut off by the pruning procedure with only one of the them surviving. The path that has to be taken at that point becomes trivial, but the chatbot application, that uses the nodes as “questions”, will still ask the question to the user although the possible suggested answer is only one. For this reason, the export functions were modified to include a “short circuiting feature” that skips node with only one branch surviving the pruning operation (in case the latter one is active, clearly) recurring directly on the grandchild instead.

In this way we were able to obtain, starting from the data, a decision tree able to classify some problem in a resulting class, pruning unnecessary branches and questions. At that point we were able to start with the chatbot creation.

**The chatbot implementation**

For the chatbot creation we considered at first a bot-creator Node.js application available on Github. The application has the ability to create and compile a bot, based on the Microsoft Bot Framework starting from a JSON specification file. The Microsoft Bot Framework is a framework that allows the creations of chatbots based on a dialog model. It is so possible to create different dialogs inside the same chat, each with different steps (questions/answers) executed in a waterfall flow. The framework allows also to ability to test the bot locally using the Microsoft Bot Emulator. In addition, bot developed with the framework can be easily deployed on the Azure platform.

Unfortunately, the application was not well maintained and compatible only with older versions of both Node.js and the Microsoft Bot Framework and therefore it was not possible to use it without rewriting many parts of it. For this reason, we decided to write a custom Node.js bot interpreter.

The bot interpreter

The bot interpreter we wrote is a Node.js application able to use a decision tree exported in the following JSON format as basis for the chatbot:

{label: “aaa”, children: [{ edgeLabel: “bbb”, label: “ccc” }, { edgeLabel: “ddd”, label: “eee”, children […] }] }

Each node is exported as a label attribute indicating the attribute of the dataset we are considering at that node and, if it is not the root, an edgeLabel attribute indicating which branch we have to follow to reach the current node from its parent, therefore the answer to the parent’s question. Moreover, a children attribute, which is an array of nodes, could be present.

The JSON input file is the only parameter required to the bot interpreter application which then creates a RESTful server on the local machine and a Microsoft “botbuilder” object. The application parses then the JSON specification file and presents the user a welcoming message. When the user is ready to start, it creates a dialog with the root node as parameter.

The general strategy of the interpreter is to create a new dialog for each node, replacing the older one to keep a low memory usage. The dialog reads the attribute that needs to be asked to the user and first checks if an answer to the current question was already given previously. This is important especially for numeric attributes. C4.5 creates for each numeric attribute a binary split, where a branch contains all instances where the attribute is less or equal to the split point, while the other branch all ones with a greater value.

In some cases, value ranges are needed to precisely decide between different cases and so the algorithm asks more than one time the same “question” with different split points. This could happen at different levels of the decision tree but also on consecutive levels. If the bot interpreter application would not memorize the questions and answers pairs, it will ask more than one time the same question, forcing the user to answer multiple times. For this reason, every time a new question is posed to the user, it is memorized along with the collected answer in a map. We are so able to check if an answer has already been asked and, in that case, retrieve the answer and use it to decide which branch to follow.

On the other hand, if the question has not already been asked, we pose it to the user. If the attribute is a numerical attribute, the user is just prompted with the question and is able to answer with whatever is a number (bot integers and floating points). Differently, if the attribute is categorical and therefore it can only take some specific values, the application prompts the questions and a list of clickable buttons representing the accepted values. The user can so directly click on one of the given options or reply with a message containing one of the options’ text.

When the interpreter reaches a leaf node, so a node with the children attribute missing, it presents the answer to the user and exits dumping on the server console all questions and answers retrieved that were memorized in the map.