SD201 TP01

1 Introduction

1.1 General Information on the Project

The project consists in implementing an algorithm for the construction of a decision tree, both in the case when the dataset is static (i.e. it does not change over time) and dynamic. To this end, please refer to the slides on decision trees. In particular, for the lab we shall consider the slides called "Lec4a-Intro.pdf" on *ecampus*, slide 32 of for the definition of Gini gain, slides 79-95 for the algorithm building a decision tree, slides 104-110 for the definition of F1-score. The slides have been updated recently, please make sure you use the most recent version.

This is the first of three lab sessions that lead to such a task. The deadline for the entire project (all three parts) is at the end of the course, i.e. **November 15th at 11.45am**. You should submit the code in Python to *ecampus* as well as the results you produced (more info will be provided later) until that date. Once you have completed a task you can check whether it is correct or not by running the corresponding script (more info below). Observe that you can work on the project even outside the lab hours, except for the last lab session where you are supposed to finish within the end of that session. This said, you are strongly encouraged to keep the pace and try to finish all tasks in class.

You should write your own code, if we suspect that you copy the code from somebody else or used any tool that wrote the code for you, all students involved will fail the course! We will use various tools to detect any kind of plagiarism.

1.2 Folder overview

This folder consist in 3 main parts, that work together to help you implement and test the algorithms of this lab session : the code, the *input_data* folder and the *results* folder.

1.2.1 The code

main.py This is the main script of the lab, that runs the necessary computations to produce the results expected for each question. The syntax to run it as an executable is `python main.py [eval or debug] [question number]`. You are not expected to modify this file and, if you do, we will restore it to its original version when testing your results.

check_results.py This script aims at checking that the results produced by main.py match the expected results. In debug mode, it will indicate for each input file and for each results if the achieved results match what is expected. In eval mode, it will only check that the checksum of your achieved results file match the expected checksum (see subsection 1.2.3). The syntax to run it as an executable in `python check_results.py [eval or debug] [question number]`. You are not expected to modify this file and, if you do, we will restore it to its original version when testing your results.

read_write.py This file contains a list of technical functions to read the input data and to write your result data with the right formatting in the results files. You are not expected to modify this file and, if you do, we will restore it to its original version when testing your results.

PointSet.py This file defines a class meant to handle a set of points and how to split them at best to build a decision Tree. You are expected to modify this file as part of the lab.

Tree.py This file defines a class meant to represent a decision tree. You are expected to modify this file as part of the lab.

evaluation.py This file defines functions that are useful for evaluating the results of a classification algorithm. You are expected to modify this file as part of the lab.

1.2.2 Input Data folder

This folder contains the data used for evaluating the algorithms you implemented. In general, the files containing "debug" in their names are used for checking the results of your algorithms against known results, while the files containing "eval" in their names are used for checking that your algorithms give results consistent with an unknonwn expected result, of which only the checksum is known.

The data files are csv files, in which each line is a data point, the first column is the label of the point and the following columns are features. The first line is the types of the data ('l' for "label", 'b' for "binary", 'c' for "categorical" and 'r' for "real").

1.2.3 Results

This folder contains two separated subfolders : *expected* and *achieved*. The *expected* folder contains the results that we expect your implementations to produce.

There are two types of files, the first type is the *debug* files that give you explicitly the expected results so that you can debug your code and understand what is wrong if you don't get the expected results, the second type is the *eval* files that contain only the checksum of the results files you are expected to

achieve. Your algorithms will ultimatly be evaluated only using this eval mode and files.

The other subfolder is *achieved*. That is the file to which your results will be automatically written.

1.3 Lab procedure

1.3.1 General instructions

The lab consists in a few questions, each of which asks you to implement a method or function and explains what are the results that will be produced and evaluated. You are expected to follow this procedure :

1. Implement the function/method that the question asks for
2. Run `main.py debug [question number]` and then `check_results.py debug [question number]` to see if your results match the expected one. If not, you are encourage to use `pdb` to debug your implementation. This step is not required but strongly advised.
3. Run `main.py eval [question number]` and then `check_results.py eval [question number]` to see if your implementation will allow you to have the points for the question.
4. You are encourage to run again the `eval` part when implementing the following questions to make sure that you don't break the completion of some previous questions.

Keep in mind that the feature asked for at a given question should stay available while you implement the features of the following questions, that is, the process of running `main.py` and then *check_results.py* for a previous question should always give positive results.

1.3.2 Expected, allowed and not allowed changes to the code

You are expected to edit the body of some methods. You are allowed to change the body of any method of function in the files that you can edit, but you should keep the return type and semantic of those methods or functions, both the ones already implemented or the ones specified in the questions. You are expected not to remove any parameter to any method or function and, if you add parameters, those should be placed after any already-existing parameter and have default values so that, when the function/method is called with the default values, its behaviour is the same as before the addition of the parameter.

You are free to add any function, method or class that is not already implemented. You are also free to add any file you deem necessary.

Keep in mind that your implementation is automatically executed by the `main.py` script and therefore, all those rules aim at keeping the code consistent with this script.

1.3.3 Libraries

You are only allowed to use the `numpy` library as well as any module from the standard library.

1.3.4 Communication and plagiarism

You are allowed and even encouraged to communicate and help each other for understanding the questions or debugging your implementations. However, you should write your own code! You are not allowed to copy code written by somebody else, found on the internet, or use any tool that would write the code for you. We will use various methods to check that your submission is entirely from you. If we suspect that you did not write your own code you will **fail** the course!

2 Questions

2.1 Gini score and gain

For the whole lab, we will work with binary labels (`True` or `False`). For the first part of the lab, all the features are also binary. All the features will be represented as floats, but binary features will always get value 0.0 (`False`) or 1.0 (`True`).

Question 1 (3/20) In the file *PointSet.py*, implements the `get_gini(self)` method. This method should return a float representing the Gini score (called Gini in slide 31) of the points contained in the object. The results for each test file will be the Gini score of the points contained in the file. Note that the `PointSet` constructor gets an input parameter `types` that you don't need for those questions. This parameter will be explained when needed.

Question 2 (2/20) In the file *PointSet.py*, implement `get_best_gain(self)` method. This method should return the index of the feature along which splitting provides the best Gini gain (the index of the first feature, that is on the second column in the file, is 0), as well as the value of this gain. We use the definition of Gini gain provided in slide 32. When the gain is ill-defined, both return values should be None. If a feature has the same value in all the points, both return values should also be None. The results for each test file will be the index of the feature with best Gini gain and the related gain value.

2.2 Evaluation metrics

Question 3 (1/20) In the file *evaluation.py*, implement the function that is named `precision_recall(expected_results, actual_results)` that, given two arrays containing respectively the expected labels of a set of points and the labels achieved by a classification algorithm for this same set, returns two float values that are the precision and the recall of the algorithm for this set of points. Then, implement the `F1_score(expected_results, actual_results)`

function that, given the same inputs, returns the F1-score of the algorithm for this set of points.

For this question, the input data files are a set of pairs of expected and achieved results. The results for each test file are the precision, the recall and the F1-score associated with those data. The definition of F1-score is provided in slides 104-110.

2.3 Trees for binary features

Question 4 (2/20) In the file *Tree.py* implement the constructor and the `decide` method. The constructor is expected to create a binary tree of height 1 (that is, one root and two children, which are leaves) by splitting the points of its `PointSet` along the feature that provides best gain. Note that the constructor has parameters `types` and `h` that you don't need for now. Note however that `types` is needed for constructing the `PointSet`. Those parameters will be explained when needed. The `decide(self, features)` method is expected to, given a set of features, return the class (`True` or `False`) in which the algorithm would put a point with those features. The result for each test file will be the F1-score of the Tree built using the 80% first points of the file, and tested using the 20% last points.

Question 5 (1/20) Adapt the `Tree` class so that the `h` parameter is the maximal height of the Tree, that is the maximal distance between the root and any leaf. Note that, for completing this question, the size of each branch should always be `h` except if, at some point before reaching this height, it is not relevent to split the `PointSet` anymore. The result for each test file will be the F1-score of the Tree built using the 80% first points of the file, and tested using the 20% last points.

2.4 Trees for categorical features

So far, all the features of the data have been Boolean (aka binary). However, some of the Boolean features in our datasets might be better represented by one categorical feature, whose values are the identifiers of the Boolean features. Note that, as those values are identifiers, it is not relevant to compare them using an order (i.e. it is not relevant to assume that the identifier `0` is less than the identifier `1`).

For efficiency issues, we distinguish Boolean features from general categorical features. To do that, we use the `FeaturesTypes` enum class defined in the `PointSet.py`. Note that there is a third type of features named `REAL` that we will use in the next lab. You don't need it so far.

The `types` parameters of the `PointSet` and `Tree` constructor are arrays of `FeaturesTypes` that indicates if each feature is Boolean or class.

Question 6 (1/20) Adapt the implementations of `Tree` and `PointSet` classes so that it can accept Boolean and categorical features. Try to compute the Gini gain as efficiently as possible, taking advantage of fact that some

Boolean features (those mentioned above) can all be represented by one categorical feature. You should avoid for example to re-split those categories into separate Boolean features. The result for each test file will be the F1-score of the Tree built using the first 80% of the points of the file, and tested using the remaining 20% points.