# Introduction to Artificial Intelligence – Programming assignment 3

UNIZG FER, academic year 2023/24.
v2.0 (updated: May 12th, 2021)

Handed out: March 7, 2024 Due: May 23, 2024 at 23:59.

## Decision tree (24 points)

In the third lab assignment we will implement the decision tree algorithm and analyse the problems and extensions of this machine learning algorithm.

Despite the fact that we are using a single example throughout the instructions, your implementation **must** work for **all** files attached with this assignment in reasonable time period (max. 2 minutes). The evaluation of your implementation will be conducted using *autograder*. It is expected that your know how to run your solution with *autograder* in front of the Teaching Assistant during the examinaton. "Autograder instructions" were written by taking into account that **we** will run your solution, which we won't do this year. **You** need to make sure that your solution is runnable and can be evaluated with *autograder* in front of the Teaching Assistant. Together with the lab assignment, you will get access to all test samples.

Deadline for solution submission on Moodle **does not include** the last minute in the day, so make sure that you upload the archive with your solution **before** 23:59. Submissions uploaded exactly at 23:59 or after will be considered as late submissions.

Please read, in detail, the instructions for input and output formatting in the Section "Autograder instructions" as well as the sample outputs in chapter "Output examples". In case your submitted lab assignment cannot be compiled or executed with the autograder it will be graded with 0 points **without exception**. Your code may not use **any external libraries**. If you are not sure whether you are allowed to use some library, check whether it is part of the standard library package for that language. To compute the logarithm in base 2 when computing entropy you can use the built-in library `math` in python, `java.lang.Math` for Java and `<math.h>` for `c++`.

The total number of points in this lab assignment is 24. The number of points that you obtain will be scaled by percentage to 7.5 points.

### 1. Data loading

In order to apply our decision tree algorithm to various tasks, we will first define a standardised input format for the **dataset file**. A classic dataset format used throughout machine learning is `csv` (comma separated values). Files in `csv` format contain values separated with commas. Each line of the file contains the same number of values. In our case, the values represent features used in our machine learning algorithm. The first line of the file will always be the *header*, which contains the names of the features found in each column. Even though in practice this might be different, for the purpose of this lab assignment the comma symbol **will not be found** in the feature values.

An example of the first two lines for the dataset file for the example from the lectures (the first line is the header):

```
weather,temperature,humidity,wind,play
sunny,hot,high,weak,no
```

A convention in machine learning is to use the last column for the class label. All other columns will contain features. In the scope of this lab assignment we will always have **a single label**, which can have an **arbitrary number of values**.

## 2. ID3 decision tree (12 points)

In this part of the lab assignment we will implement the ID3 algorithm for learning a decision tree. We recommend that when writing your code, you adhere to the machine learning algorithm design principles similar to the *scikit-learn* machine learning library:

Each machine learning agorithm should be implemented as a separate class, with the following functionalities:

1. A constructor which accepts and stores the algorithm hyperparameters
2. The `fit` method, which obtains a dataset as an argument and performs **model learning**
3. The `predict` method, which obtains a dataset as an argument and performs **prediction** of the class label based on a trained model

For the case of our basic version of the ID3 decision tree, our algorithm will **not use** any hyperparameters, but this will change in subsequent tasks. An important distinction between the functionalities of the methods our machine learning algorithm should implement is that the model is **trained** only in the `fit` method, while the `predict` method only serves to produce the predictions based on an already trained model (the model is **not trained** on the test data!).

The pseudocode of our ID3 algorithm usage could look as follows:

```
model = ID3() # construct model instance
model.fit(train_dataset) # learn the model
predictions = model.predict(test_dataset) # generate predictions on unseen data
```

For implementing the ID3 algorithm you can use the pseudocode of ID3 algorithm from the slides in the presentation "10. Machine learning". **Note**: Pseudocode from the slides is different from that in Croatian version of the video presentation for the lecture "10. Machine learning", so make sure that you follow the code from the slides and implement the algorithm according to that.

An example of the control output for the ID3 decision tree construction for the `volleyball.csv` dataset:

```
IG(weather)=0.2467 IG(humidity)=0.1518 IG(wind)=0.0481 IG(temperature)=0.0292
IG(wind)=0.9710 IG(temperature)=0.0200 IG(humidity)=0.0200
IG(humidity)=0.9710 IG(temperature)=0.5710 IG(wind)=0.0200
```

Your algorithm **does not** have to produce this output, we are adding it purely for debugging and validity checking purposes. The control output is sorted according to information gain. A line in control output corresponds to information gain calculation for a single node, starting from the root down to the leaves. If there are multiple features with the maximum information gain value, choose the first feature according to the alphabetical order (if the choice is between A, B and C, we will choose A).

When predicting (the `predict` method), if the trained model is faced with a yet unseen feature value, it should return the **most frequent** observed value of the class label in the subset of training dataset for that node. In case there are multiple values of the class label with the same highest frequency, choose the first one according to alphabetical order (if the choice is between A, B and C, we will choose A).

As a result of learning the decision tree, your algorithm **must** print **(1)** all the branches in the decision tree that lead to each node. Each branch must be printed in a separate line. Output for each branch should contain the **name** of feature used for splitting the dataset in each node in the branch and the feature **value** for which the classification continues on the current branch. Finally, at the end of each branch you should output the value of the class label for that branch (leaf value). Before printing all the branches, it is necessary to print the keyword "`[BRANCHES]:`" in a separate line, after which all the branches should be printed.

Output values for each branch should be formatted according to the following rules:

- output for each node in the branch should be in the format `level:feature_name=feature_value`, where `level` represents the depth of the node, `feature_name` the name of the feature used for splitting in that node, and `feature_value` the value of the feature for which classification continues in the current branch,
- nodes should be ordered by depth, from the smallest to the largest, and separated by a single whitespace,
- output for the leaf node should contain only the value of the class label in that leaf.

An example of such output for the training set from `volleyball.csv`:

```
[BRANCHES]:
1:weather=cloudy yes
1:weather=rainy 2:wind=strong no
1:weather=rainy 2:wind=weak yes
1:weather=sunny 2:humidity=high no
1:weather=sunny 2:humidity=normal yes
```

This output will be **evaluated with the autograder**. The order of the branches does not have to be the same as in the provided examples, but it is necessary to output all the branches.

After printing all the branches in the tree, your algorithm **must** output **(2)** predictions for all the examples in the **test** set. The order of predictions for the examples in the test set must be the same as their order in the input file. Before the predictions, you should output the keyword "`[PREDICTIONS]:`", and after it (in the same line) predictions separated by a single whitespace. An example of such output for the test set from `volleyball_test.csv`:

```
[PREDICTIONS]: yes yes yes yes no yes yes yes no yes yes no yes no no yes yes
    yes yes
```

## 3. Evaluating model performance (4 points)

Once we have learned our model, we would like to summarize its performance on data. To do this, we will implement the calculation of **accuracy** score and **confusion matrix**. Accuracy is defined as the ratio of correctly classified examples over the total number of examples:

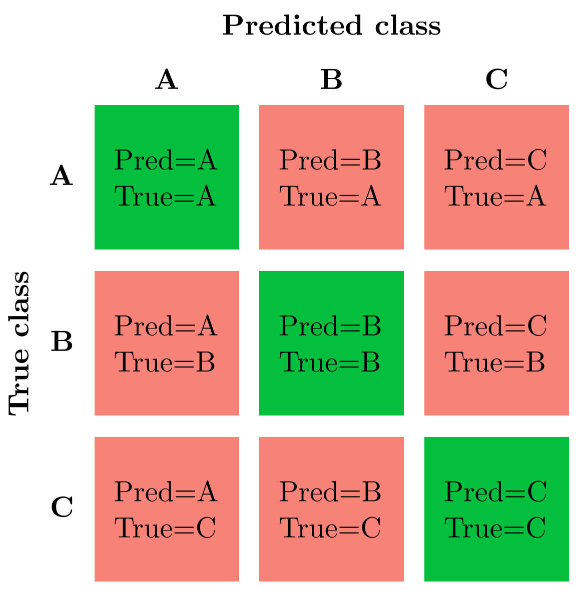
$$\text{accuracy} = \frac{\text{correct}}{\text{total}} \tag{1}$$

Together with the previously mentioned information about the branches in the tree and predictions, your algorithm must also output **(3)** accuracy on the **test** set. This line must be formatted as follows (an example for the test set from `volleyball_test.csv`):

```
[ACCURACY]: 0.57895
```

Numerical value of the accuracy must be rounded to five digits.

Accuracy is a measure which summarizes the algorithm performance with a single number – but, it is possible that our model performs better for one label value than for the others. To find out the performance of our model for each distinct class label, we will also implement the calculation of **confusion matrix**. A detailed description of the confusion matrix can be found online, but a brief explanation is as follows:



In our example `A`, `B` and `C` are the values of the class label. In the confusion matrix we make a distinction between errors based on the true and predicted value of the class label. The shape of the confusion matrix is $Y \times Y$, where $Y$ is the number of distinct values of our class label. Each cell of the confusion matrix contains the **number** of instances for which we have obtained the mentioned combination of predicted and actual value.

After accuracy, your algorithm must output **(4)** the **confusion matrix** for the **test** set. Before printing the matrix, it is necessary to print the keyword "`[CONFUSION_MATRIX]:`" in a separate line. In the output of the confusion matrix you should output only the values from each cell of the matrix (not the values of the class label), with columns separated by a single whitespace. An example of such output for test set from `volleyball_test.csv`:

```
[CONFUSION_MATRIX]:
```

```
4 7
1 7
```

When printing the confusion matrix, rows and columns representing class label values must be sorted **alphabetically** (as in the previous example).

Set of class label values displayed in the confusion matrix should contain only (1) true labels of examples from the test set and (2) predicted values for the test set. Class label values that appeared in the training set, but not in the true or predicted values of the test set, must not be included in the confusion matrix. For example, if the train set contained class label values `A`, `B` and `C`, but the test set contains only true labels `A` and `B`, and prediction on the test also returned only values `A` and `B`, row and column for class value `C` must not be printed in the confusion matrix.

In case your output does not contain some of the four elements required in the output ((1) branches, (2) predictions, (3) accuracy, and (4) confusion matrix), it will be considered that you have not implemented that part of the lab assignment.

## 4. Limiting the tree depth (8 points)

The ID3 algorithm often runs into problems with overfitting due to the tree depth growing until all the examples are correctly classified or all the features are exhausted. Along with pruning, another method of regularizing our decision tree model is to limit its maximum depth. In this task, we will extend our implementation of the ID3 algorithm to include the **maximum depth hyperparameter**. This hyperparameter will be passed as the third argument in the command line when running your code. If the argument is not passed, the depth of the tree is considered to be unlimited.

A consequence of limiting the depth of our decision tree is that we will not always converge to a single class label in every leaf node (the subset of data which is incorrectly classified in that node may contain multiple possible values of the class label). To solve this, we will implement a democratic solution. In the nodes in which we don't have a single class label, our algorithm should return the **most frequent** observed value of the class label for that node. In case there are multiple values of the class label with the same highest frequency, choose the first one according to alphabetical order (if the choice is between A, B and C, we will choose A).

The output of the ID3 decision tree with limited depth should be formatted the same as the output for the ID3 tree with unlimited depth. In case when maximum depth of the tree is set to 0, the root node of the tree is considered as leaf node, and the output in the field "BRANCHES" should contain only the output for the leaf node (i.e., the value of the class label obtained using the described democratic solution). An example of output for such case can be found in the chapter "Output examples".

An example of the **full test output** for a decision tree with its depth limited to 1 for training set `volleyball.csv` and test set `volleyball_test.csv`:

```
[BRANCHES]:
1:weather=cloudy yes
1:weather=rainy yes
1:weather=sunny no
[PREDICTIONS]: yes no no yes yes no yes yes yes yes yes yes yes no no yes yes
    yes yes
[ACCURACY]: 0.36842
[CONFUSION_MATRIX]:
2 9
```

```
3 5
```

## Autograder instructions

The instructions are given below about uploaded archive structure. Detailed instructions on how to run *autograder* using given structure can be found in `autograder.zip` archive in file `README.md`.

### Uploaded archive structure

The archive that you will upload to Moodle **has to** be named `JMBAG.zip`, while the structure of the unpacked archive **has to** be as in the following example (the following example is for solutions written in Python, while examples for other languages are given in subsequent sections):

```
|JMBAG.zip
|-- lab3py
|----solution.py [!]
|----decisiontree.py (optional)
|----...
```

Uploaded archives that are not structured in the given format will **not be graded**. Your code must be able to execute with the following arguments from command line:

1. Path to the train dataset
2. Path to the test dataset
3. Depth of ID3 tree (optional)

First two arguments will always be passed to your solution. Third argument might appear, but if it is not given, tree's depth is considered unlimited.

Your code will be executed on linux. This does not affect your code in any way except if you hardcode the paths to files (which in any way, **you should not do**). Your code should **not use any external libraries**. Use the `UTF-8` encoding for all your source code files.

An example of running your code with tree depth limited to 1 (for Python):

```
>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv 1
```

### Instructions for Python

The entry point for your code **must** be in the `solution.py` file. You can structure the rest of your code using additional files and folders, or you can leave all of it inside `solution.py` file. Your code will always be executed from the folder of your assignment (`lab3py`).

The directory structure and execution example can be seen at the end of the previous chapter. Your code will be executed with python version `3.7.4`.

### Instructions for Java

Along with the lab assignment, we will publish a project template which should be imported in your IDE. The structure inside your archive `JMBAG.zip` is defined in the template and has to be as in the following example:

```
|JMBAG.zip
|--lab3java
|----src
|------main.java.ui
|--------Solution.java [!]
|--------DecisionTree.java (optional)
|--------...
|----target
|----pom.xml
```

The entry point for your code **must** be in the file `Solution.java`. You can structure the rest of the code using additional files and folders, or you can leave all of it inside the `Solution.java` file. Your code will be compiled using Maven.

An example of running your code with the autograder with the tree depth limited to 1 (from the `lab3java` folder):

```
>>> mvn compile
>>> java -cp target/classes ui.Solution datasets/volleyball.csv
    datasets/volleyball_test.csv 1
```

Info regarding the Maven and Java versions:

```
>>> mvn -version
Apache Maven 3.6.3
Maven home: /opt/maven
Java version: 15.0.2, vendor: Oracle Corporation, runtime: /opt/jdk-15.0.2
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.15.0-139-generic", arch: "amd64", family: "unix"

>>> java -version
openjdk version "15.0.2" 2021-01-19
OpenJDK Runtime Environment (build 15.0.2+7-27)
OpenJDK 64-Bit Server VM (build 15.0.2+7-27, mixed mode, sharing)
```

**Important:** please check that your implementation can be compiled with the predefined `pom.xml` file.

### Instructions for C++

The structure inside your archive `JMBAG.zip` has to be as in the following example:

```
|JMBAG.zip
|--lab3cpp
|----solution.cpp [!]
|----decisiontree.cpp (optional)
|----decisiontree.h (optional)
|----Makefile (optional)
|----...
```

**If** your submitted archive does not contain a `Makefile`, we will use the Makefile template available along with the assignment. **If** you submit a `Makefile` in the archive (which we don't suggest, unless you really know what you're doing), we expect it to work.

An example of compiling and running your code with the autograder with the tree depth limited to 1 (from the `lab3cpp` folder):

```
>>> make
>>> ./solution datasets/volleyball.csv datasets/volleyball_test.csv 1
```

Info regarding the gcc version:

```
>>> gcc --version
gcc (Ubuntu 9.3.0-11ubuntu0~18.04.1) 9.3.0
```

## Output examples

Each output example will also contain the running command that was used to produce that output. Running command assumes Python implementation, but the arguments will be the same for other languages as well.

For each output we also provide the control output containing information gain for each feature when deciding which feature to use for splitting the data. As mentioned previously, your algorithm doesn't have to output this information, but only the four requested fields ("BRANCHES", "PREDICTIONS", "ACCURACY" and "CONFUSION_MATRIX"). Due to line length, the output for "PREDICTIONS" field is split into two lines in some examples. As mentioned in the instructions, your output for that field should in a **sinle** line.

**1. Volleyball**

```
>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv

IG(weather)=0.2467 IG(humidity)=0.1518 IG(wind)=0.0481 IG(temperature)=0.0292
IG(wind)=0.9710 IG(humidity)=0.0200 IG(temperature)=0.0200
IG(humidity)=0.9710 IG(temperature)=0.5710 IG(wind)=0.0200
[BRANCHES]:
1:weather=cloudy yes
1:weather=rainy 2:wind=strong no
1:weather=rainy 2:wind=weak yes
1:weather=sunny 2:humidity=high no
1:weather=sunny 2:humidity=normal yes
[PREDICTIONS]: yes yes yes yes no yes yes yes no yes yes no yes no no yes yes
    yes yes
[ACCURACY]: 0.57895
[CONFUSION_MATRIX]:
4 7
1 7
```

```
>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv 1

IG(weather)=0.2467 IG(humidity)=0.1518 IG(wind)=0.0481 IG(temperature)=0.0292
[BRANCHES]:
1:weather=cloudy yes
1:weather=rainy yes
1:weather=sunny no
[PREDICTIONS]: yes no no yes yes no yes yes yes yes yes yes yes no no yes yes
    yes yes
[ACCURACY]: 0.36842
[CONFUSION_MATRIX]:
2 9
3 5
```

```
>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv 0
```

```
[BRANCHES]:
yes
[PREDICTIONS]: yes yes yes yes yes yes yes yes yes yes yes yes yes yes yes yes
    yes yes yes
[ACCURACY]: 0.42105
[CONFUSION_MATRIX]:
0 11
0 8
```

### 2. Logic

```
>>> python solution.py datasets/logic_small.csv datasets/logic_small_test.csv
```

```
IG(A)=0.3219 IG(C)=0.2365 IG(D)=0.2365 IG(B)=0.0000
IG(C)=1.0000 IG(D)=1.0000 IG(B)=0.0000
[BRANCHES]:
1:A=False False
1:A=True 2:C=False False
1:A=True 2:C=True True
[PREDICTIONS]: False False True False False True
[ACCURACY]: 0.50000
[CONFUSION_MATRIX]:
3 2
1 0
```

```
>>> python solution.py datasets/logic_small.csv datasets/logic_small_test.csv 1
```

```
IG(A)=0.3219 IG(C)=0.2365 IG(D)=0.2365 IG(B)=0.0000
[BRANCHES]:
1:A=False False
1:A=True False
[PREDICTIONS]: False False False False False False
[ACCURACY]: 0.83333
[CONFUSION_MATRIX]:
5 0
1 0
```

### 3. Titanic

Due to the length of the output, here we provide only the output for the case in which tree depth is limited to 2. The rest of the outputs can be checked in the log file of the correct solution in the autograder archive.

```
>>> python solution.py datasets/titanic_train_categorical.csv
   datasets/titanic_test_categorical.csv 2
```

```
IG(sex)=0.2180 IG(fare)=0.0888 IG(passenger_class)=0.0712
    IG(cabin_letter)=0.0682 IG(age)=0.0204
IG(passenger_class)=0.1914 IG(cabin_letter)=0.1050 IG(fare)=0.0961 IG(age)=0.0533
IG(cabin_letter)=0.0492 IG(age)=0.0384 IG(fare)=0.0369 IG(passenger_class)=0.0330
[BRANCHES]:
1:sex=female 2:passenger_class=lower_class yes
1:sex=female 2:passenger_class=middle_class yes
```

```
1:sex=female 2:passenger_class=upper_class yes
1:sex=male 2:cabin_letter=A no
1:sex=male 2:cabin_letter=B no
1:sex=male 2:cabin_letter=C no
1:sex=male 2:cabin_letter=D no
1:sex=male 2:cabin_letter=E no
1:sex=male 2:cabin_letter=F no
1:sex=male 2:cabin_letter=T no
1:sex=male 2:cabin_letter=U no
[PREDICTIONS]: no no yes no no no yes yes no yes no yes no no no no no yes no
    yes no no no yes no no yes no no no yes no no yes no no no no no yes yes no
    no no no yes no no no no no no yes no no no no no no yes no no yes yes yes
    yes yes no yes no no no yes yes no yes yes no no no no yes no no yes yes no
    no no yes yes no yes no no yes no yes yes no no
[ACCURACY]: 0.77228
[CONFUSION_MATRIX]:
54 11
12 24
```