Artificial Intelligence – Programming assignment 3

UNIZG FER, academic year 2019/20.

Handed out: 12. May 2020 Due: 31. May 2020 at 23.59.

Decision forests (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.

Note: 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). Together with the attached files, your implementation will also be tested with an *autograder* using examples which are not published, so make sure that you cover all the edge cases that might introduce issues to your implementation. Together with the lab assignment, you will get access to a sample of test examples, so you can try running them locally.

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++`. For the random number generator which the bonus assignment requires, use the builtin `random` library for python, `java.util.Random` for Java and `<random>` for `c++`.

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 purposes 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**.

Along with the dataset file we will also use a **configuration file** which will contain the **hyperparameters** of our algorithms. This file will contain the (1) **execution mode** of your impementation, (2) the **algorithm** used (which will be different from ID3 only for the bonus assignment), (3) the **maximum depth of the tree** for the third task in the lab assignment and (4) the **number of trees**, (5) **example ratio** and (6) **feature ratio** used in the bonus assignment. The configuration file will always contain a single value per line, labeled with a keyword. In case a value is not defined, you can assume the tree depth to be unlimited, the number of trees to be 1 and the feature and example ratios to be 1. (100%). The remaining values will always be defined. An example of the configuration file can be seen as follows:

```
mode=test
model=ID3
max_depth=3
num_trees=1
feature_ratio=1.
example_ratio=1.
```

The value with the **mode** keyword will not affect the performance of your algorithm – only its output. In case of `mode=test`, your algorithm should **only** print the required lines for each task. The output of your implementation in this mode will be evaluated with the autograder. The output for any other mode can be freely formatted. The value with the **model** keyword will be different from `ID3` only in the bonus assignment. For the bonus assignment, this value will be `RF` and will indicate that you should use the random forest algorithm. The order of the hyperparameters **will not** necessarily be the same, but they will always be labeled with the same keyword and separated with an equals symbol. When being ran, your implementation will always receive **exactly three arguments** via the command line: **(1)** the path to the train dataset, **(2)** the path to the test dataset and **(3)** the path to the configuration file.

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, which has 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!). While 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 **current node**. In case there are multiple values of the class label with the same frequency, choose the first one according to alphabetical order (if the choice is between A, B and C, we will choose A).

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
```

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. The order of the nodes on the same level of the decision tree does not need to be the same in your implementation. In case there are multiple features which maximize information gain, 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, in the test mode your model **should** print the **(1) depth and feature name** based on which the nodes in the decision tree were built. The output should be formatted as follows:

```
0:weather, 1:wind, 1:humidity
```

This output will be **evaluated with the autograder**. The output elements are in the `depth:feature_name` format, where each element is separated by a comma and a whitespace `, `. The elements do not need to be ordered by depth, although this is aesthetically helpful for the oral examination. The first line which your algorithm prints to the *stdout* should be the aforementioned line.

Apart from this, your model should print the predictions for all **test** set examples in the second row. The order of the predicted class labels should be the same as in the test file. Separate the value of the predicted class label with a whitespace as follows:

```
yes yes yes yes no yes yes yes no yes yes no yes no no yes yes yes yes
```

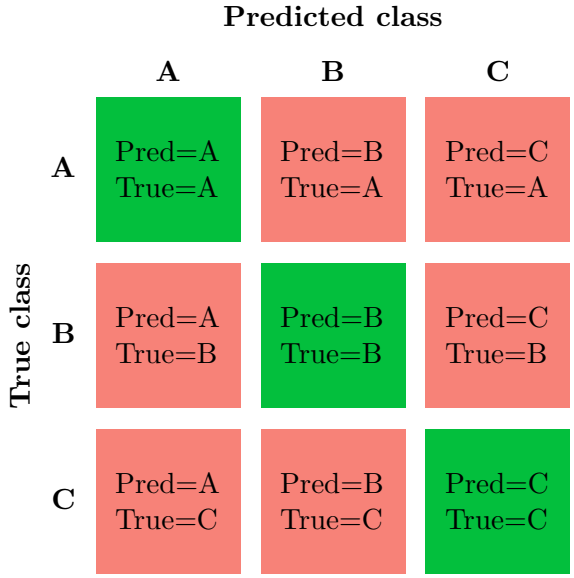
3. Evaluating model performance (4 points)

Once we have learned our model, we would like to summarize its performance on data. To do this, we will implement the **accuracy** score and **confusion matrix**. Accuracy is

defined as the ratio of correctly classified examples over the total number of examples:

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

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 **confusion matrix**. A detailed description of the confusion matrix can be found online, but a brief explanation is as follows:



Where in our example `A, B and C` are 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 instance for which we have obtained the mentioned combination of predicted and actual value. While printing the confusion matrix, sort the values of the class label **alphabetically** as in the example.

In the **test** mode, your implementation should print the **(3.)** accurcacy on the **test** dataset in the third output line. This line should contain a single number a sfollows:

```
0.57895
```

The remaining Y lines of your test mode output should contain the **confusion matrix**. While printing the confusion matrix, print only the values of each cell of the confusion matrix, where the columns are separated with a single space, as follows:

```
4 7
1 7
```

In case your output does not contain these lines, the autograder will count that you have not implemented this 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 set via the configuration file, and if its value is not defined or set to -1, the depth of the tree is not limited.

A consequence of limiting the depth of our decision tree is that we will not always converge to a single class label in every 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 in the **current node**. In case there are multiple values of the class label with the same 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 same as the output for the ID3 tree with unlimited depth. An example of the **full test output** for a decision tree with its depth limited to 1:

```
0:weather
yes no no yes yes no yes yes yes yes yes yes yes no no yes yes yes yes
0.36842
2 9
3 5
```

Bonus points assignment: Random forest (+6 points)

Note: Solutions to all bonus points assignments must be uploaded to Ferko in the same archive together with the rest of the lab assignment.

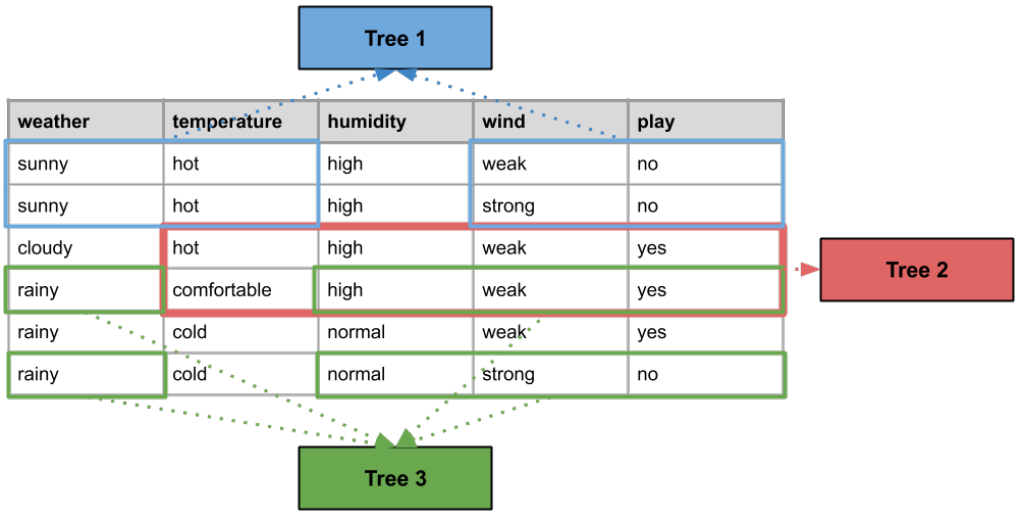
When we set a limited depth for our decision tree algorithm, we severely constrain its expressiveness as it can only represent a fixed number of decisions. To solve this problem, we will implement the **random forest** algorithm.s

Random forests use the idea of *strength in numbers* through ensembling and instead of a single tree – construct multiple. To ensore that all trees we generate are not identical, we need to apply ideas of data subsampling (*bagging*) and feature subsampling. Through data and feature subsampling we intend to obtain smaller trees which focus on separate aspects of input data and features.

This concept can easily be visualised:

Our algorithm will construct **num_trees** separate ID3 decision trees whose depth is limited **max_depth**. For each tree and the assigned **example_ratio** (in $(0, 1]$) we randomly sample `instance_subset = round(example_ratio * dataset_size)` samples to train that specific tree. For each tree and the assigned **feature_ratio** (in $(0, 1]$) we randomly sample `feature_subset = round(example_ratio * num_features)` features which will be used to train that specific tree. When subsampling features, we **always include** the class label but don't count it towards the feature limit.

The random forest algorithm uses a separate **sample** of input data of shape instance_subset$\times$



feature_subset+1. When each of the **num_trees** is learned, we again apply the democratic voting approach where each tree *votes* (generates its prediction) for the value of the class label. The final value of the class label for the random forest algorithm is the value with the most votes. In case multiple values have the same number of votes, we choose the first one according to alphabetic ordering.

As it is not possible to easily check code which uses random number generators across programming languages and platforms, your output in this assignment should have two additional lines at the start **for each tree**. For each tree, print the **feature names** selected and **the indices** (starting from 0) of the examples used to train that tree. **Do not print** the names of features for the nodes of each tree. An example of the output for `num_trees=5, example_ratio=0.5, feature_ratio=0.5, max_depth=1` can be seen as follows:

```
weather temperature
8 2 10 13 6 13 7
temperature wind
11 12 5 2 2 6 9
weather temperature
13 8 10 11 6 11 4
temperature humidity
11 9 1 13 3 13 0
temperature humidity
3 7 8 3 0 11 8
no yes yes yes no yes yes yes no no yes no no yes yes yes yes yes yes
0.52632
4 7
2 6
```

Autograder instructions

Uploaded archive structure

The archive that you will upload to Ferko **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. Path to the configuration file

All three arguments will be used in every run of your solution.

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 (for Python):

```
>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv
config/id3.cfg
```

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 (from the `lab3java` folder):

```
>>> mvn compile
>>> java -cp target/classes ui.Solution datasets/volleyball.csv
    datasets/volleyball_test.csv config/id3.cfg
```

Info regarding the Maven and Java versions:

```
>>> mvn -version
Apache Maven 3.6.1
Maven home: /usr/share/maven
Java version: 13.0.2, vendor: Oracle Corporation, runtime: /opt/jdk-13.0.2
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "5.3.0-45-generic", arch: "amd64", family: "unix"

>>> java -version
java version "13.0.2" 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, 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 (from the `lab3cpp` folder):

```
>>> make
>>> ./solution datasets/volleyball.csv datasets/volleyball_test.csv
    config/id3.cfg
```

Info regarding the gcc version:

```
>>> gcc --version
gcc (Ubuntu 9.2.1-9ubuntu2) 9.2.1 20191008
Copyright (C) 2019 Free Software Foundation, Inc.
```

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. The instructions will be updated with output examples on multiple datasets during this week. For now, the outputs and attached datasets will only be ran on `volleyball.csv` and `volleyball_test.csv` with different configuration files.

2. ID3 decision tree & 3. Evaluating model performance

The first two lines of the output are the outputs from the ID3 decision tree implementation, while the next two lines are generated through model performance evaluation.

1. Volleyball

`>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv config/id3.cfg`

```
0:weather, 1:wind, 1:humidity
yes yes yes yes no yes yes yes no yes yes no yes no no yes yes yes yes
0.57895
4 7
1 7
```

2. Logical formula

The second dataset on which we will evaluate our models will be a logical formula. Based on the values of features and the class label we need to predict the value of an unknown logical function. For this task we will use the `logic_small.csv` and `logic_small_test.csv` files.

`>>> python solution.py datasets/logic_small.csv datasets/logic_small_test.csv config/id3.cfg`

Control output (IG):

```
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
```

Test output:

```
0:A, 1:C
False False True False False True
0.50000
3 2
1 0
```

3. Titanic

The third and last public dataset we will evaluate our models on will be a real-world example of a machine learning dataset. The goal of this task is to predict whether a person has survived the sinking of the RMS Titanic.

For this task we will use the `titanic_train_categorical.csv` and `titanic_test_categorical.csv` files.

```
>>> python solution.py datasets/titanic_train_categorical.csv datasets/titanic_test_ca
config/id3.cfg
```

Control output0 **only shown for the first level** due to length:

```
IG(sex)=0.2180 IG(fare)=0.0888 IG(passenger_class)=0.0712
    IG(cabin_letter)=0.0682 IG(age)=0.0204
...
```

Test output:

```
0:sex, 1:passenger_class, 2:age, 3:fare, 4:cabin_letter, 4:cabin_letter,
    4:cabin_letter, 3:fare, 4:cabin_letter, 4:cabin_letter,
    4:cabin_letter, 3:fare, 4:cabin_letter, 4:cabin_letter, 3:fare,
    4:cabin_letter, 4:cabin_letter, 4:cabin_letter, 3:fare,
    4:cabin_letter, 4:cabin_letter, 2:age, 3:fare, 4:cabin_letter,
    3:fare, 4:cabin_letter, 4:cabin_letter, 2:age, 3:fare,
    4:cabin_letter, 3:cabin_letter, 4:fare, 1:cabin_letter, 2:age,
    3:fare, 4:passenger_class, 3:fare, 4:passenger_class, 2:fare, 3:age,
    3:age, 4:passenger_class, 3:age, 4:passenger_class, 3:age, 2:fare,
    3:age, 4:passenger_class, 3:age, 4:passenger_class,
    4:passenger_class, 3:age, 4:passenger_class, 3:age,
    4:passenger_class, 2:fare, 3:age, 4:passenger_class, 3:age,
    4:passenger_class, 2:age, 3:fare, 4:passenger_class, 2:age, 2:age,
    3:passenger_class, 4:fare, 4:fare, 4:fare, 3:fare, 4:passenger_class,
    4:passenger_class, 3:fare, 3:fare, 4:passenger_class, 3:fare, 3:fare,
    4:passenger_class, 4:passenger_class, 3:fare, 4:passenger_class,
    4:passenger_class, 4:passenger_class, 4:passenger_class,
    3:passenger_class, 4:fare, 4:fare
no no no no no no no no no no yes yes no no no yes yes no yes no no no
    no no no yes no no no yes no yes yes yes no no no no yes no no no no
    no yes no no yes no no no yes no no no no no no yes yes no no yes yes
    yes yes no yes no no no no no no yes yes yes no yes no yes no no yes
    yes no no no yes yes no yes no no no no yes no no no
0.78218
56 9
13 23
```

4. Limiting the tree depth

The first two lines of the output are the outputs from the ID3 decision tree implementation, while the next two lines are generated through model performance evaluation.

1. Volleyball

```
>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv
config/id3_maxd1.cfg
```

```
0:weather
yes no no yes yes no yes yes yes yes yes yes yes no no yes yes yes yes
0.36842
2 9
3 5
```

2. Logical formula

```
>>> python solution.py datasets/logic_small.csv datasets/logic_small_test.csv
config/id3_maxd1.cfg

0:A
False False False False False False
0.83333
5 0
1 0
```

3. Titanic

```
>>> python solution.py datasets/titanic_train_categorical.csv datasets/titanic_test_ca
config/id3_maxd1.cfg

0:sex
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
0.77228
54 11
12 24
```

Bonus: Random forest

The first `num_trees * 2` lines of the output are generated by each tree from the random forest, the next line contains the predictions and the final two lanes are generated through model performance evaluation.

1. Volleyball

```
>>> python solution.py datasets/volleyball.csv datasets/volleyball_test.csv
config/rf_n5_sub05.cfg

weather temperature
8 2 10 13 6 13 7
temperature wind
11 12 5 2 2 6 9
weather temperature
13 8 10 11 6 11 4
temperature humidity
I11 9 1 13 3 13 0
temperature humidity
3 7 8 3 0 11 8
no yes yes yes no yes yes yes no no yes no no yes yes yes yes no yes
```

```
0.57895
5 6
2 6
```

2. Logical formula

```
>>> python solution.py datasets/logic_small.csv datasets/logic_small_test.csv
config/rf_n9_sub0705.cfg
```

If all features for a random forest subtree are labelled with the same class label, you **don't need** to print the first output row for that tree.

```
6 1 7 9 4 9 5
6 4 2 2 6 6 4
A B
1 6 4 8 5 4 0
A B
9 2 8 6 0 9 2
B C
6 3 2 5 6 2 0
A C
2 8 2 2 1 9 3
A C
7 8 6 0 2 2 9
B D
0 0 3 8 9 8 1
B D
8 8 6 0 3 0 4
False True False False True False
0.83333
4 1
0 1
```

3. Titanic

```
>>> python solution.py datasets/titanic_train_categorical.csv datasets/titanic_test_ca
config/rf_n9_sub05.cfg
```

The output is limited to **only one tree** due to length

```
fare cabin_letter
681 403 291 145 570 380 57 547 10 368 22 42 547 176 384 223 11 18 372 633
    641 635 81 264 186 320 402 433 290 536 27 386 244 412 99 172 229 420
    482 267 324 605 654 537 455 536 157 313 293 2 516 618 76 356 676 111
    341 105 133 286 682 470 76 612 593 386 612 453 189 401 464 391 576
    592 24 370 159 624 41 238 518 414 649 525 427 76 105 256 652 54 316
    684 688 633 386 308 219 97 400 211 166 123 585 580 420 99 698 650 484
    458 492 654 33 605 621 454 155 285 214 430 381 240 396 647 351 210 31
    111 421 354 124 69 408 154 443 34 128 498 489 11 597 439 43 145 168
    321 90 484 136 533 49 379 91 347 289 127 396 304 505 319 327 217 573
    537 69 167 505 228 54 94 572 634 580 469 62 560 428 504 226 661 318
    386 11 350 120 592 688 643 93 523 327 70 36 165 198 113 196 276 687
    150 623 81 201 267 257 650 205 637 39 404 274 389 304 24 241 354 7
    202 551 621 417 568 506 392 650 103 541 308 94 302 174 536 326 352 15
```

```
    104 698 593 31 488 161 237 570 431 696 381 676 362 599 240 166 157 23
    116 232 580 45 432 538 205 362 470 333 394 378 147 530 331 435 94 485
    176 14 574 354 349 171 120 140 680 283 508 136 641 337 637 619 393
    423 76 490 285 316 270 543 411 622 470 136 53 592 240 244 645 522 638
    502 264 272 189 686 505 318 151 398 562 304 312 580 490 191 284 581
    59 688 485 122 624 475 696 584 277 174 598 252 50 116 133 347 393 325
    497 266 558 241 65 88 396 358 376
no no no no no no yes no no no no yes yes no no no no no no yes no no no
    no no no no no no no yes no no yes no no no no no yes no no no no no
    yes no no no no no no yes no no no no no no yes no no no yes yes no
    yes no no no no no yes no no yes yes no no no no yes no no yes no no
    no no yes yes no no no no no no yes no no no
0.83168
64 1
16 20
```