



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Practical Work 2: Decision Forests

Victor Badenas Crespo

Supervised and Experiene Learning
Master in Artificial Intelligence (FIB-MAI)

May 2021

Contents

1	Introduction	1
1.1	Introduction	1
2	Classifiers	2
2.1	Leaf	2
2.2	Node/Tree	3
2.2.1	fit	3
2.2.2	predict	4
2.3	ForestInterpreter	4
2.4	BaseForest	4
2.5	DecisionForestClassifier	5
2.6	RandomForestClassifier	5
3	Datasets	6
3.1	Datasets	6
3.1.1	Car Evaluation Data Set	6
3.1.2	Chess (King-Rook vs. King-Pawn) Data Set	6
3.1.3	Iris Data Set	7
4	Results Tables	8
5	Instructions	11
5.1	Execution instructions	11
5.1.1	Linux	11
5.1.2	Other OS	12

Chapter 1

Introduction

1.1 Introduction

In this work, we have implemented and tested a version of *RandomForestClassifier* and *DecisionForestClassifier*. Both approaches are quite similar and have a lot of common concepts, that is why they have been developed in an OOP style following some SOLID principles. The algorithms have been tested for the required configurations which as a reminder are:

1. Random Forest Classifier

- (a) $NT = 1, 10, 25, 50, 75, 100$

- (b) $F = 1, 3, \log_2(M + 1), \sqrt{M}$

2. Decision Forest Classifier

- (a) $NT = 1, 10, 25, 50, 75, 100$

- (b) $F = \text{int}(M/4), \text{int}(M/2), \text{int}(3 * M/4), RU(1, M)$

Where RU stands for a random uniform value between 1 and M. The previous combinations have been tested on UCI Datasets. The datasets chosen for this experiments are:

1. iris (< 500)
2. car ($500 < n_instances < 2000$)
3. kr-vs-kp (> 2000)

The datasets will be explained later on in the report. Finally we will comment the results extracted from the models, the times for each task, accuracies and the most relevant features.

Chapter 2

Classifiers

In this chapter we will discuss the base classes for all Classifiers as well as the particularities of each of the two classifiers implemented for this work. The implementation of the Classifiers is segmented in several parts:

1. BaseClassifier (*./src/base_classifier.py*)
2. ForestInterpreter (*./src/forest_interpreter.py*)
3. Node, Tree, Leaf: all of them inherit from BaseClassifier (*./src/tree_units.py*)
4. BaseForest: inherits from BaseClassifier and ForestInterpreter (*./src/base_forest.py*)
5. RandomForestClassifier: inherits from BaseForest (*./src/random_forest.py*)
6. DecisionForestClassifier: inherits from BaseForest (*./src/decision_forest.py*)

The *BaseClassifier* is an abstract class implementing the `fit`, `predict` and `fit_predict` methods for the classes that inherit from it. *ForestInterpreter* implements the loading and inference methods to create a Forest from a model previously saved as json. The *Node*, *Tree* and *Leaf* classes are the main units of the building of a tree structure and they all inherit from *BaseClassifier* as they need `fit` and `predict` methods. Then the *BaseForest* class inherits from *BaseClassifier* and *ForestInterpreter*. The first for the basic train methods and the later for inference. Finally the two *ForestClassifiers* inherit from *BaseForest* and will each implement their distinguishable features.

2.1 Leaf

The class *Leaf* implements the basic termination of a Tree structure. The main concept behind it is that once a branch of the tree has been terminated, will imply that a prediction has to be made for that instance and so, when in training we reach a leaf, we need to compute the probability for the instance to be of a class. For that we store the probability computed as the count of class instances for the instances that reach the Leaf in the node divided by the number of instances in the leaf. When predicting, a dictionary containing the class names and the probability will be returned.

2.2 Node/Tree

The *Node* and *Tree* class are equivalent. However, the tree class is instantiated by the *Classifiers* while the *Node* is only instantiated by the *Tree* or another *Node*. The class is responsible for multiple actions as defined by the following methods:

2.2.1 fit

The goal of this method is to determine the best split of the node and determine the two branches if it is not feasible to split the data anymore, a return statement forces the parent node to terminate that *Node* attempt with a *Leaf*. The main process for the training of the node is as follows:

Algorithm 1: Node/Tree fit method

```

Result: self
X := data for the node;
F := number of random features;
attributes := dataset attributes;
node_gini := gini_index(X);
best_gain := 0;
best_feature, best_value := None, None;
if  $F < 0$  or  $\text{len}(\text{attributes}) < F$  then
    | features := attributes
else
    | features := K random attributes
end
for feature in features do
    for unique value in feature column in X do
        true_split, false_split = try to split by the condition. feature == value for
            categorical and feature >= value for numerical;
        if  $\text{len}(\text{true\_split}) == 0$  or  $\text{len}(\text{false\_split}) == 0$  then
            | skip to next value as it does not split the data;
        end
        gain := node_gini - split_gain(true_split, false_split);
        if  $\text{gain} > \text{best\_gain}$  then
            | best_feature := feature;
            | best_value := value;
            | best_gain := gain;
        end
    end
end
true_split, false_split := split(X, best_value, best_feature);
// initialize both branches from the node. If the split only contains one item, create leaf
    instead and finally call recursively the nodefit method. If the method returns a None,
    replace the newly created node with a Leaf instead to terminate the process.;
branches[True] := initialize_branch(true_split);
branches[False] := initialize_branch(false_split);

```

The algorithm above is implemented on the *Node/Tree*. First, the data for the node X , the number of random features to F and dataset attributes *attributes*. Then we compute the *gini_index* for the instances in X . The *gini_index* is computed efficiently using *pandas* following the following

expression:

$$Gini(X) = 1 - \sum_{i=0}^N \left(\frac{X_{x \in C_i}}{len(X)} \right)^2$$

after computing the gini_index of the data in the node, a set of F features are chosen from the dataset attributes. If there are not enough attributes, all attributes will be used. Also, if $F < 0$, all attributes will be considered. For each feature considered, all (feature, value) pairs in the dataset are tested and the gini index gain is computed for all and the split condition with the highest gain is chosen to be the condition for this node.

Once the condition is set, the branches are then initialized. There are 3 scenarios:

1. the number of instances is 1 for a given split. Then the branch is initialized to a Leaf.
2. the branch is initialized as a Node but no gain is found in further splitting the data. Then the branch is initialized to a Leaf.
3. the branch is initialized to a Node and it fits appropriately. We then continue creating nodes recursively.

2.2.2 predict

The predict method in the Node will evaluate the condition and then return whatever the branch returns recursively. This way we can return the dictionary containing the probabilities provided by the Leaf of the tree.

2.3 ForestInterpreter

The forest interpreter class contains the main loading and prediction methods for any BaseForest type of Classifier. The most interesting method for the class is the *predict* method. The predict method will ask the prediction for each instance to each tree and then average the probabilities for each of the class in order to get the most probable class for each instance of the dataset. It will call the predict method for each tree in the classifier. The class was optimized using the python package *multiprocessing* which allows us to generate a pool of threads that run a function's code for each item in a list. This way we can paralellize the computation of the predictions for each three among the desired number of jobs using:

```
1 import multiprocessing as mp
2 from functools import partial
3 with mp.Pool(self.n_jobs) as p:
4     predictions = list(p.map(partial(self._predict_tree, X=X), self.trees))
```

2.4 BaseForest

The main class for the ForestClassifiers. Implements the main fit methods for generating the trees from the train dataset. Also implements the methods to save the Classifiers as json objects. The fit method initializes the dataset characteristics and then creates NT trees and calls the fit method for each one of them. The same optimization that was done in the predict function in the ForestInterpreter class was done here, where each tree is fitted in a different thread to be able to paralellize the computation.

2.5 DecisionForestClassifier

The first final object is the `DecisionForestClassifier`, which inherits from `BaseForest` all the base methods for inference and training. The method is based on the [1] paper. The functions that the class implements is the `_fit_tree` method. In the case of the `DecisionForestClassifier`, the method first generates a dataset with F columns chosen at random from the dataset attributes. Once this dataset is generated, a tree is fitted with the newly constructed dataset where the per-node selection parameter F is set to -1 to exhaustively search for all (feature, value) pairs of the instances at each node.

2.6 RandomForestClassifier

Finally the `RandomForestClassifier`, which also inherits from `BaseForest` for the same reasons as `DecisionForestClassifier`, is implemented. The method is based on the [2] paper. Similarly to the `DecisionForestClassifier`, it only implements the `_fit_tree` method, which generates a bootstrapped dataset with the same number of instances as the original dataset. Then initializes the tree where the F value is set to the F value of the `RandomForestClassifier` to explore only F features at each node.

Chapter 3

Datasets

3.1 Datasets

The datasets used for the comparison were retrieved from the UCI dataset repository [3]. The datasets chosen are all exclusively categorical without any missing values, as those are the two aspects that prism cannot handle well. The datasets chosen are the Car Evaluation Data Set, kr-vs-kp and hayes-roth datasets.

3.1.1 Car Evaluation Data Set

Car Evaluation Database[4] was derived from a simple hierarchical decision model originally developed for the demonstration of DEX, M. Bohanec, V. Rajkovic: Expert system for decision making. Sistemica 1(1), pp. 145-157, 1990.).

Input attributes are printed in lowercase. Besides the target concept (CAR), the model includes three intermediate concepts: PRICE, TECH, COMFORT. Every concept is in the original model related to its lower level descendants by a set of examples.

The Car Evaluation Database contains examples with the structural information removed, i.e., directly relates CAR to the six input attributes: buying, maint, doors, persons, lug_boot, safety.

Because of known underlying concept structure, this database may be particularly useful for testing constructive induction and structure discovery methods.

The characteristics of the dataset are as shown in 3.1

Data Set Characteristics:	Multivariate	Number of Instances:	1728	Area:	N/A
Attribute Characteristics:	Categorical	Number of Attributes:	6	Date Donated	1997-06-01
Associated Tasks:	Classification	Missing Values	No	Number of Web Hits:	1347720

Table 3.1: Car Evaluation Characteristics

3.1.2 Chess (King-Rook vs. King-Pawn) Data Set

The last dataset used in the project is the Chess (King-Rook vs. King-Pawn) Data Set [5], which consists of chess data from the match. The Dataset characteristics are shown in 3.2.

Data Set Characteristics:	Multivariate	Number of Instances:	3196	Area:	Game
Attribute Characteristics:	Categorical	Number of Attributes:	36	Date Donated	1989-08-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	125000

Table 3.2: Chess (King-Rook vs. King-Pawn) Characteristics

3.1.3 Iris Data Set

The last dataset used in the project is the Iris Data Set [6], which consists of measurements of iris flowers' petal and sepals. The Dataset characteristics are not shown because the website at the time of writing was down.

Chapter 4

Results Tables

Car DF

F	NT	fit_time	predict_time	fit_accuracy	predict_accuracy	buying	maint	doors	persons	lug_boot	safety
1	1	0.041	0.047	0.701	0.698	0	0	0	0	0	2
1	10	0.063	0.05	0.701	0.698	3	0	3	4	6	6
1	25	0.118	0.082	0.701	0.698	9	12	9	6	10	14
1	50	0.101	0.155	0.701	0.698	15	18	27	20	16	24
1	75	0.13	0.17	0.701	0.698	33	36	39	20	28	30
1	100	0.166	0.14	0.701	0.698	51	51	57	26	30	38
3	1	0.362	0.058	0.714	0.662	0	35	8	0	0	4
3	10	0.51	0.073	0.767	0.748	53	68	71	52	69	67
3	25	0.644	0.155	0.735	0.717	135	165	194	173	159	151
3	50	1.236	0.476	0.729	0.71	334	395	492	259	250	357
3	75	1.768	0.959	0.714	0.7	616	603	761	422	359	452
3	100	2.309	1.028	0.709	0.698	844	903	917	501	552	565
4	1	1.068	0.05	0.826	0.775	0	56	9	74	0	4
4	10	1.691	0.116	0.914	0.829	274	385	279	276	111	93
4	25	2.528	0.543	0.935	0.854	857	705	575	586	462	366
4	50	4.359	1.835	0.917	0.813	1505	1399	1284	1024	1128	1014
4	75	6.41	3.855	0.922	0.813	2042	1910	1884	1967	1615	1643
4	100	8.45	3.951	0.93	0.819	2732	2710	2326	2582	2166	2216
random	1	10.739	0.077	1.0	0.838	401	78	10	147	567	4
random	10	11.849	0.293	1.0	0.887	1547	572	908	394	639	341
random	25	12.286	1.339	1.0	0.871	2615	928	2526	2046	1030	1367
random	50	15.031	6.183	1.0	0.856	4282	2244	3419	4895	2752	2701
random	75	21.818	13.499	1.0	0.817	5294	3386	4369	6730	3775	3993
random	100	32.096	12.662	1.0	0.81	6788	5235	5708	8016	5284	5776

Car RF

F	NT	fit_time	predict_time	fit_accuracy	predict_accuracy	buying	maint	doors	persons	lug_boot	safety
1	1	0.357	0.047	0.718	0.71	10	18	11	15	10	12
1	10	0.612	0.087	0.723	0.712	130	130	106	139	92	107
1	25	0.693	0.244	0.706	0.702	292	271	249	277	220	237
1	50	1.221	0.594	0.711	0.7	564	551	525	494	426	488
1	75	1.701	1.498	0.708	0.702	831	846	807	727	654	723
1	100	2.342	1.381	0.716	0.7	1101	1109	1079	967	899	986
2	1	1.424	0.054	0.776	0.723	52	51	64	2	45	45
2	10	2.158	0.21	0.95	0.844	601	519	501	431	407	464
2	25	3.829	0.901	0.971	0.844	1454	1146	1269	1037	1065	1187
2	50	6.119	3.88	0.988	0.829	2790	2458	2581	2027	2000	2161
2	75	8.699	8.701	0.988	0.837	4074	3820	3837	3085	3020	3205
2	100	11.838	9.678	0.988	0.848	5468	4980	5111	4130	3987	4253
3	1	3.054	0.057	0.819	0.675	46	114	71	106	42	90
3	10	3.794	0.299	0.985	0.819	863	950	787	900	569	811
3	25	7.765	1.656	0.998	0.858	2351	2153	1979	2152	1698	1979
3	50	13.191	7.71	1.0	0.883	4637	4600	4318	3946	3594	3711
3	75	18.113	17.485	1.0	0.888	7059	7153	6496	5506	5371	5547
3	100	26.213	16.234	1.0	0.888	9524	9360	8830	7221	7101	7482

Iris DF

F	NT	fit_time	predict_time	fit_accuracy	predict_accuracy	sepal_length	sepal_width	petal_length	petal_width
1	1	0.385	0.038	0.962	0.911	0	0	38	0
1	10	0.394	0.044	0.99	0.889	96	21	152	36
1	25	0.467	0.145	0.981	0.867	256	84	266	108
1	50	0.877	0.316	0.981	0.867	448	210	532	216
1	75	1.218	0.604	0.981	0.889	736	294	912	252
1	100	1.716	0.563	0.981	0.867	960	462	1140	324
2	1	0.701	0.032	0.99	0.911	0	0	43	34
2	10	1.016	0.075	1.0	0.889	104	281	224	204
2	25	1.748	0.179	1.0	0.911	374	593	566	500
2	50	2.919	0.788	1.0	0.933	949	989	1284	892
2	75	4.23	1.919	1.0	0.911	1549	1514	1815	1320
2	100	6.303	1.851	1.0	0.911	2173	2084	2342	1692
3	1	1.289	0.032	1.0	0.933	0	23	43	35
3	10	1.755	0.075	1.0	0.911	193	343	232	224
3	25	3.125	0.357	1.0	0.911	604	687	641	548
3	50	5.181	1.017	1.0	0.889	1283	1191	1459	1039
3	75	7.213	2.142	1.0	0.933	1978	1817	2095	1566
3	100	10.577	2.484	1.0	0.889	2708	2485	2695	2047
random	1	1.688	0.037	1.0	0.933	0	23	43	35
random	10	1.962	0.065	1.0	0.933	139	325	235	168
random	25	2.451	0.233	1.0	0.933	467	643	569	436
random	50	4.983	0.884	1.0	0.911	960	1048	1301	790
random	75	6.91	1.721	1.0	0.911	1529	1496	1878	1164
random	100	9.341	1.921	1.0	0.889	2135	2073	2391	1617

F	NT	fit_time	predict_time	fit_accuracy	predict_accuracy	sepal_length	sepal_width	petal_length	petal_width
1	1	0.317	0.032	0.962	0.756	14	13	13	12
1	10	0.52	0.054	1.0	0.911	145	112	153	118
1	25	0.882	0.197	1.0	0.956	352	317	350	310
1	50	1.344	0.771	1.0	0.933	724	652	677	604
1	75	1.992	1.268	1.0	0.933	1051	1003	1023	924
1	100	2.806	1.139	1.0	0.911	1403	1364	1344	1234
2	1	0.634	0.04	0.971	0.911	17	17	17	13
2	10	1.005	0.063	1.0	0.933	154	149	158	149
2	25	1.606	0.184	1.0	0.911	401	396	398	361
2	50	2.576	0.624	1.0	0.889	818	779	812	712
2	75	3.7	1.446	1.0	0.911	1220	1174	1247	1060
2	100	5.387	1.245	1.0	0.911	1650	1578	1646	1388
3	1	0.856	0.04	1.0	0.911	20	12	18	14
3	10	1.195	0.055	0.99	0.956	175	154	161	135
3	25	2.454	0.248	1.0	0.956	421	416	416	335
3	50	3.979	0.611	1.0	0.956	867	814	841	673
3	75	5.316	1.823	1.0	0.911	1306	1222	1277	1012
3	100	7.694	1.387	1.0	0.911	1719	1627	1713	1347

[illegible][illegible]

Chapter 5

Instructions

In this chapter we will discuss the instructions to run the code.

5.1 Execution instructions

5.1.1 Linux

Environment install with conda:

```
1 conda create --name sel3.9 python=3.9
2 conda activate sel3.9
3 pip install -r requirements.txt
```

To get the data in the right format:

```
1 ./getData.sh
```

The scripts contains the following:

```
1 #!/bin/sh
2
3 get_data () {
4     url=$1
5     headers=$2
6     filename=data/${basename ${url}}
7     echo ${headers} > ${filename}
8     curl ${url} >> ${filename}
9 }
10
11 mkdir data/
12
13 get_data https://archive.ics.uci.edu/ml/machine-learning-databases/car/car.data 'buying,
    maint,doors,persons,lug_boot,safety,class'
14
15 get_data https://archive.ics.uci.edu/ml/machine-learning-databases/chess/king-rook-vs-king-
    pawn/kr-vs-kp.data 'bkblk,bknwy,bkon8,bkona,bkspr,bkxbq,bkxcr,bkxwp,blxwp,bxqsq,cntxt,
    dsopp,dwipd,hdchk,katri,mulch,qxmsq,r2ar8,reskd,reskr,rimmx,rkxwp,rxmsq,simpl,skach,
    skewr,skrxp,spcop,stlmt,thrsk,wkcti,wkna8,wknck,wkovl,wkpos,wtoeg,class'
16
17 get_data https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data '
    sepal_length,sepal_width,petal_length,petal_width,class'
18
19
20 for fullfile in data/*; do
21
22     filename=$(basename -- "$fullfile")
23     extension="${filename##*}"
24     filename="${filename%.*}"
```

```

25
26     if [ "$extension" = "data" ]; then
27         python scripts/split_train_test.py -i $fullfile
28         rm $fullfile
29     fi
30 done

```

which downloads the data, adds the corresponding header and splits the data into train and test.

To run all the training and testing:

```
1 ./runAll.sh
```

the script contains the following:

```

1 #!/bin/sh
2
3 python scripts/train_all.py -i data/iris.train -i data/car.train -i data/kr-vs-kp.train -f
   models/ -t -1 -m RF
4 [ $? -ne 0 ] && exit 1
5 python scripts/train_all.py -i data/iris.train -i data/car.train -i data/kr-vs-kp.train -f
   models/ -t -1 -m DF
6 [ $? -ne 0 ] && exit 1
7 python scripts/test_all.py -i data/iris.test -i data/car.test -i data/kr-vs-kp.test -md
   models/ -t -1
8 [ $? -ne 0 ] && exit 1

```

5.1.2 Other OS

Environment install with conda:

```

1 conda create --name sel3.9 python=3.9
2 conda activate sel3.9
3 pip install -r requirements.txt

```

if you are on a UNIX based system with bash you can run './getData.sh' from CLI or follow the following steps:

1. Download the datasets from UCI repository:
 - (a) <https://archive.ics.uci.edu/ml/machine-learning-databases/car/car.data>
 - (b) <https://archive.ics.uci.edu/ml/machine-learning-databases/chess/king-rook-vs-king-pawn/kr-vs-kp.data>
 - (c) <https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>
2. move them to the *data/* folder
3. manually add the headers for the datasets:
 - (a) kr-vs-kp.data: bkbk, bknwy, bkon8, bkona, bkspr, bkxbq, bkxcr, bkxwp, blxwp, bxqsq, cntxt, dsopp, dwipd, hdchk, katri, mulch, qxmsq, r2ar8, reskd, reskr, rimmx, rkxwp, rxmsq, simpl, skach, skewr, skrxp, spcop, stlmt, thrsk, wkcti, wkna8, wknc, wkovl, wkpos, wtoeg, class
 - (b) car.data: buying, maint, doors, persons, lug_boot, safety, class
 - (c) iris.data: sepal_length, sepal_width, petal_length, petal_width, class
4. split the data into train and test with the following commands:

```

1 python scripts/split\_train\_test.py -i data/car.data
2 python scripts/split\_train\_test.py -i data/kr-vs-kp.data
3 python scripts/split\_train\_test.py -i data/iris.data

```

Once the data is formatted correctly into the ‘data/’ subfolder, run the training and inference with the following commands:

```

1 # Train random forest
2 python scripts/train\_all.py \
3     -i data/iris.train \
4     -i data/car.train \
5     -i data/kr-vs-kp.train \
6     -f models/ \
7     -t -1 \
8     -m RF
9 # Train Decision forest
10 python scripts/train\_all.py \
11     -i data/iris.train \
12     -i data/car.train \
13     -i data/kr-vs-kp.train \
14     -f models/ \
15     -t -1 \
16     -m DF
17 # Test all the datasets
18 python scripts/test\_all.py \
19     -i data/iris.test \
20     -i data/car.test \
21     -i data/kr-vs-kp.test \
22     -md models/ \
23     -t -1

```

finally to build the csvs of the report:

```

1 python scripts/create\_csv\_all.py

```

In the end, the models will be saved to ‘models/dataset/*model_config*/{model.json, feat_counts.json, model.trees, results.txt}’. Where the ‘model.json’ contains the serialized object to be loaded by ‘ForestInterpreter’, the ‘feat_counts.json’ file contains the occurrences of each of the features in a node. The ‘results.txt’ contains the train and test accuracies for the model. The ‘model.trees’ contains a text representation of the trees for that model. f.i:

Tree 0:

```

(safety == low)
|t- {'unacc': 1.0}
|f- (safety == high)
    |t- {'unacc': 0.48, 'acc': 0.36, 'vgood': 0.11, 'good': 0.05}
    |f- {'unacc': 0.61, 'acc': 0.31, 'good': 0.07}

```

Tree 1:

```

(doors == 3)
|t- {'unacc': 0.71, 'acc': 0.23, 'good': 0.03, 'vgood': 0.03}
|f- (doors == 2)
    |t- {'unacc': 0.75, 'acc': 0.19, 'good': 0.04, 'vgood': 0.02}
    |f- (doors == 5more)
        |t- {'unacc': 0.66, 'acc': 0.25, 'good': 0.04, 'vgood': 0.04}
        |f- {'unacc': 0.69, 'acc': 0.22, 'vgood': 0.05, 'good': 0.04}

```

Tree 2:

```
(lug_boot == small)
|t- {'unacc': 0.77, 'acc': 0.21, 'good': 0.03}
|f- (lug_boot == med)
    |t- {'unacc': 0.68, 'acc': 0.22, 'good': 0.05, 'vgood': 0.04}
    |f- {'unacc': 0.65, 'acc': 0.24, 'vgood': 0.07, 'good': 0.04}
```


Bibliography

- [1] Tin Kam Ho. “The random subspace method for constructing decision forests”. In: *IEEE transactions on pattern analysis and machine intelligence* 20.8 (1998), pp. 832–844.
- [2] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32.
- [3] *UCI Machine Learning Repository*. URL: <https://archive.ics.uci.edu/ml/index.php>.
- [4] *Car Evaluation Data Set*. URL: <https://archive.ics.uci.edu/ml/datasets/car+evaluation>.
- [5] *Chess (King-Rook vs. King-Pawn) Data Set*. URL: [https://archive.ics.uci.edu/ml/datasets/Chess+\(King-Rook+vs.+King-Pawn\)](https://archive.ics.uci.edu/ml/datasets/Chess+(King-Rook+vs.+King-Pawn)).
- [6] *Iris Data Set*. URL: <https://archive.ics.uci.edu/ml/datasets/iris>.