ForestFire Documentation

Release 1.1.5

Marlon Weinert

Dec 22, 2017

CONTENTS

CHAPTER

ONE

__INIT__

ForestFire is a Python tool that aims to enhance the performance of machine learning algorithms. It utilises the Random Forest algorithm - which is itself a machine learning technique - to determine the importance of features in a given set of data and make new predictions which featuresets are most likely to yield the best results. After building a Random Forest only the most promising feature sets are presented to the machine learning algorithm to gain a better result. The Random Forest is burnt down and a new one is grown until the defined maximum number of forests is reached. The results can be compared against random search.

The value of *ForestFire* lies in the selection of a *feature set* that - when computed by the designated *MLA* - yields better results than using all of the features or a random selection of features.

ForestFire is most usefull in data sets with a number of features greater than 10 where a single run of a *MLA* has a high computational cost. In such data sets the problem arises that some features are more significant than the rest. Others may even distort the performance of the underlying *MLA* in a negative fashion. With a rising number of features the number of possible combinations (= feature sets) emerges and converges towards infinity. In those cases ForestFire can help to choose those feature sets that are most promising to yield good results. By predicting the performance of new feature sets according to their importance in a Random Forest built from previous runs it is more likely to find a feature set with a higher performance after a shorter period of time than randomly choosing new feature sets.

Possible benefits:

- Increase overall precision (higher accuracy / lower Error Rate)
- Reduce overall computational cost (Finding a good solution earlier)
- Gain knowledge about importance of single features

1.1 How to use

In order to use *ForestFire* it is required to provide raw data in the form of two numpy arrays:

- **X.npy** - contains the values of the features for each data set
- **y.npy** - contains the corresponding performance of those feature sets as a single value

The *MLA* and the way the raw data is split are configured in two seperate files:

- *import_data.py* - X and y are loaded from the numpy files in the same folder. It is possible (yet not required) to apply data splitting methods here and return the train and test data sets.
- *compute.py* - Set up the *MLA* that you want to supply with promising selections of feature sets generated by *ForestFire*.

After *ForestFire* is supplied with the raw Data in X.npy and y.npy (import_data.py) and the designated *MLA* (compute.py) the default setup is complete. By executing **run_ForestFire.py** the tool can be started with default or adjusted hyperparameters.

ForestFire will execute an initial *n_start* number of *MLA* runs to set up an internal database. From this database single Decision Trees are built and grouped into a Random Forest. The Random Forest is evaluated to determine the importance of each feature. *ForestFire* will next predict the performance of possible new feature sets (chosen both randomly and deliberately). The two feature sets with the highest predicted performance (for mean and for variance) are selected, computed by the original *MLA* and their result is added to the database. The Random Forest is burnt down and a new one is built, taking into account the two newly generated data points. A total number of n_forests is built. *ForestFire* will print the current best 5 feature sets as soon as a new top 5 feature set emerges. After all Random Forests are built, the results are stored in descending order both as a .txt file and a .npy file.

In *Demo mode*, the performance of *ForestFire* is compared to randomly picking new featuresets. This can be used to make sure that the algorithm does not only exploit local maxima, but keeps exploring the whole solution space. The results can be plotted.

Quickstart: Clone Repository and run ForestFire-master/Source/ForestFire/run_ForestFire.py

CHAPTER

TWO

USING FORESTFIRE

2.1 Overview

ForestFire takes as input raw data and an unsupervised machine learning algorithm provided by the user and tries to optimize the feature selection for the given classifier.

It uses Decision Trees and Random Forests to predict the performance of untested feature sets. Only those which are most likely to perform well are used during future runs of the *Machine Learning Algorithm*.

2.1.1 Utilized Modules

The following Modules are imported during the execution of ForestFire:

```
#  Imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
```

2.1.2 Abbreviations

DT *Decision Tree*

RF *Random Forest*

MLA *Machine Learning Algorithm*

2.1.3 Glossary

branch Junction in a *Decision Tree*. Each *node* has a true and a false branch leading away from it.

Decision Tree Consists of at least one *node* and represents a treelike structure that can be used for classification of new observations

feature Unique property of the raw data set. Typically all entries in a specific column of the raw data.

feature set Combination of several single features. At least one, at most all of the available features. Used to present the *Machine Learning Algorithm* with a selection of features on which it performs with better results. Synonym to *Observation*.

ForestFire Subject of this documentation. Tool that can imporove performance and efficiency of *MLAs*.

leaf Last point of a *branch* in a :term‘Decision Tree‘

Machine Learning Algorithm Specified by the user in *Generate Database*. Can basically be any existing unsupervised machine learning algorithm that classifies the raw data. Results can be improved by *ForestFire*

node A point in a *Decision Tree* where a decision is made (either true or false)

Observation Synonym to *feature set*.

pruning Cutting back *branches* of a *Decision Tree* with little information gain. See *prune*

Random Forest Cumulation of *Decision Trees* that can be used for classification of new observations

Raw data set Data set that is provided by the user in *Import Data*. The *MLA* will run on this data set.

Synonyms

- Feature Importance = Feature Probability

2.1.4 References

2.1.5 About the author

Marlon Weinert is currently graduating from the University of Stuttgart, Germany. After finishing a B.Sc. degree in Renewable Energies in 2015 he enroled in the M.Sc. program for Energy Technology. ForestFire is his first python and machine learning project.

2.2 Import Data

corresponding file: import_data.py

In this step the raw data is imported. It must consist of two numpy arrays **X** and **y** which are located in the same directory as *import_data.py*. **X** contains the data sets in rows and the features in columns. For example, X[0:12] is the value of the 13th feature in the first data set. **y** contains the corresponding result for all data sets in a single column. It must be of the same length as X. For example y[19] is the result of the 20th data set.

After loading the data apply how it should be splitted into train and test data sets and set **X_train / X_test and y_train / y_test** accordingly.

Note: The train/test split in *import_data.py* will only be done once! Use it if a fix split is desired. If a split should be done in every future calculation (e.g. with shufflesplit), set **X = X_test = X_train and y= y_test = y_train** and configure the splitting routine in the next step (*Generate Database*).

Important: Functions used in this chapter

`ForestFire.import_data.`**`import_data`**`()`
import the raw data from two numpy arrays.

Import raw data from two numpy arrays X.npy and y.npy. Set how train and test data are to be split for fix splits. Returns train/test splits as well as number of features.

Returns:

•X_test {np.array} – result training data

•X_train {np.array} – feature training data

•y_test {np.array} – result test data

•y_train {np.array} – result training data

2.3 Generate Database

corresponding file: compute.py

In this step the underlying machine learning algorithm can be configured from scratch or inserted from an existing file. Required imports can be put at the top of the file. The default algorithm can be replaced. As inputs the train / test split data from *Import Data* can be used.

Note: If no train / test split has been configured in *Import Data* it has to be done here.

The result of the *MLA* is stored in the variable *score* and returned to the main file.

Important: Functions used in this chapter

`ForestFire.compute.`**`compute`**(*X_train*, *y_train*, *mask_sub_features*, *X_test*, *y_test*)

Computes a new dataset for the Random Forest with the underlying machine learning algorithm.

Configure your machine learning algorithm here. Add imports at the top of the file. If no train / test split is done during import, X_train and X_test are equal (y_train and y_test as well). In this case define your own splits with your machine learning algorithm.

Arguments:

•X_train {np.array} – feature training data

•y_train {np.array} – result training data

•mask_sub_features {np.array} – feature set = dedicated part of all features

•X_test {np.array} – result training data

•y_test {np.array} – result test data

Returns: score {np.float64} – score of the selected feature set

`ForestFire.Main.`**`gen_database`**(*n_start*, *X*, *y*, *X_test*, *y_test*)

Runs the underlying *MLA* *n_start* times to generate a database from which Random Forests can be built.

Arguments:

- n_start {int} – number of times the underlying *MLA* is executed
- X {numpy.array} – raw data
- y {numpy.array} – raw data
- X_test {numpy.array} – test data
- y_test {numpy.array} – test data

Returns: [numpy.array] – data set containing feature sets and corresponding results

2.4 Execution

corresponding file: run_ForestFire.py

After *importing the raw data* and *configuring the MLA*, ForestFire can be executed.

2.4.1 Hyperparameters

There is a number of hyperparameters that can be changed or left at default:

```
# Hyperparameters #

# number of runs before building first Random Forest = number of data points in first␣
↪RF; minimum = 4, default = 50
# adjust according to computational capabilities and demands of the underlying␣
↪machine learning algorithm
n_start = 30  # default = 30e
# if pruning is greater than zero, branches of a Decision Tree will be pruned␣
↪proportional to pruning value; default = 0
# advanced parameter. If set too high, all trees will be cut down to stumps. Increase␣
↪carefully. Start with values between 0 and 1.
pruning = 0.3
# minimum percentage of Datasets that is used in RF generation; default = 0.2
min_data = 0.1
# number of forests; minimum=1;  default = 25
# adjust according to computational capabilities. For each forest two new␣
↪computational runs are done. default = 20
n_forests = 85
```

These parameters should be chosen according to computational demand of the *MLA*. It makes sense to start with a small number of runs and increase it carefully. Pruning is an advanced parameter. If it is set to high, every single branch will be cut and only a tree stump with a single node is left. If this parameter is used at all it should be incremented carefully to find a good balance between merging branches and keeping the tree significant.

The following parameters can be left at default since they adapt to the raw data automatically. But changing them can tweak the performance.

```
# number of trees that stand in a forest; min = 3; default = number of features * 3
n_trees = 'default'
# number of deliberately chosen feature sets that get predicted in each forest;␣
↪default = n_trees * 5
n_configs_biased = 'default'
# number of randomly chosen feature sets that get predicted in each forest; default =␣
↪n_configs_biased * 0.2
n_configs_unbiased = 'default'
# sets how aggressively the feature importance changes; default = 0.25
# higher values will increase pressure on how often promising features will be␣
↪selected.
# advanced parameter, adjust carefully. If set too high the risk of runnning into␣
↪local extrema rises.
multiplier_stepup = 'default'
# number of recent forests that are taken into acount for generating probability of␣
↪the chosen feature sets default = 0.01
seen_forests = 'default'
# the chosen feature sets default = 4

# weight of the mean in calculating the new probability for selecting future feature␣
↪sets; default = 0.2
```

```
weight_mean = 'default'
# weight of the gradient in calculating the new probability for selecting future␣
↪feature sets; default = 0.8
weight_gradient = 'default'

# which scoring metric should be used in the Decision Tree (available: entropy,␣
↪giniimpurity and variance); default = entropy
# select variance for numerical values in y only
scoref = 'entropy'
# set random seed for repeatabilit; comment out if no repeatability is required;␣
↪default = 1
np.random.seed(10)
```

2.4.2 Demo Mode & Plot

In order to compare and plot the performance of ForestFire vs. a randomized search there are two more hyperparameters that can be used:

```
# if true a comparison between the Random Forest driven Search and a random search is␣
↪done
demo_mode = True
# decide if at the end a plot should be generated , only valid in demo mode
```

This mode can be usefull when trying to make sure that ForestFire doesn't get caught in a local extremum. In general ForestFire should always find solutions that are at least as good as a random search - otherwise there is no sense in using it at all - or better. If that's not the case it might be "stuck" at a dominant feature set that seems to perform well, but there are even better feature sets that never get chosen.

2.4.3 Output

By Executing run_ForestFire.py the algorithm starts. When a new feature set with good performance (top 5) is found, the current 5 best feature sets and the according performance are printed to the console. For each feature either 1 or 0 is displayed. 1 means that the underlying *MLA* did "see" the feature, 0 means this feature was left out

Naturally in the first runs there will be more new best feature sets. The longer the algorithm continues the harder it gets to find better values.

The importance of a feature can be interpreted by looking at the feature sets that had the best results. If for example a feature is included in all best feature sets it has a high importance. If on the other hand a feature is never included, this indicates that the feature is either not important or is even a distortion to the *MLA*.

After all Random Forests are built, the results are stored both as a .txt (human readable) and a .npy (binary, for further use with python) file. In the results file the rows contain all feature set combinations calculated by the *MLA*. The last column contains the corresponding results to the preceding information wether a feature has been selected in the particular run. Example: [1 0 0 1 0.9432] means that feature 1 and 4 were presented to the *MLA* whereas feature 2 and 3 were not. The result corresponding result is 94.32%.

Example

A generic output (with demo mode on) can look like this:

```
Starting ForestFire
Loading Raw Data
setting Hyperparameters
Generate Data Base for Random Forest
Starting ForestFire

Building Random Forest Nr. 1
wrongs: 9/39
max Probability: None
picked biased feature set for mean
picked biased feature set for var
found new best 5 feature sets: [[ 1.          1.          1.          1.          1.
 ↪         0.          1.
   1.          0.          1.          0.          1.          0.          0.74       ]
 [ 1.          0.          1.          0.          0.          1.          1.
   1.          0.          0.          0.          0.          0.
   0.72666667]
 [ 0.          0.          0.          1.          1.          1.          1.
   1.          1.          1.          0.          1.          0.          0.71       ]
 [ 1.          1.          1.          0.          1.          0.          1.
   1.          0.          1.          0.          0.          1.
   0.68666667]
 [ 0.          0.          1.          0.          1.          1.          0.
   1.          1.          1.          0.          1.          0.
   0.67666667]]

Building Random Forest Nr. 2
wrongs: 2/39
max Probability: None
picked biased feature set for mean
picked unbiased feature set for var
found new best 5 feature sets: [[ 1.          1.          1.          1.          1.
 ↪         0.          1.
   1.          0.          1.          0.          1.          0.          0.74       ]
 [ 1.          0.          1.          0.          0.          1.          1.
   1.          0.          0.          0.          0.          0.
   0.72666667]
 [ 1.          1.          1.          0.          1.          1.          1.
   1.          0.          1.          0.          0.          1.
   0.71333333]
 [ 0.          0.          0.          1.          1.          1.          1.
   1.          1.          1.          0.          1.          0.          0.71       ]
 [ 1.          1.          1.          1.          1.          1.          1.
   1.          1.          1.          1.          0.          1.          0.7
 ↪]]

   ...
   ...
   ...

Building Random Forest Nr. 8
wrongs: 4/39
max Probability: 0.133463620284
raised multiplier to 1.03
picked biased feature set for mean
picked biased feature set for var
found new best 5 feature sets: [[ 1.          0.          1.          1.          1.
 ↪         1.          1.
```

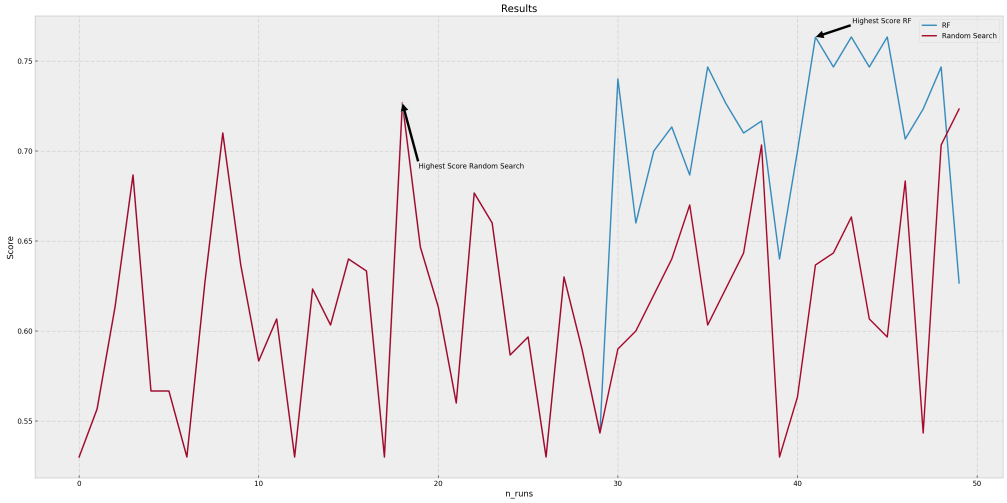
```
   1.          1.          1.          1.          1.          1.
   0.76333333]
 [ 1.          0.          1.          1.          1.          1.          1.
   1.          1.          1.          1.          1.          1.
   0.76333333]
 [ 1.          0.          1.          1.          1.          1.          1.
   1.          1.          1.          1.          1.          1.
   0.76333333]
 [ 1.          1.          1.          1.          1.          1.          1.
   1.          1.          1.          1.          1.          1.
   0.74666667]
 [ 1.          1.          1.          1.          0.          1.          1.
   1.          1.          1.          1.          1.          1.
   0.74666667]]

Building Random Forest Nr. 9
wrongs: 5/39
max Probability: 0.16963581418
picked biased feature set for mean
picked biased feature set for var

Building Random Forest Nr. 10
wrongs: 2/39

max Probability: 0.130904237306
raised multiplier to 1.04
picked biased feature set for mean
picked biased feature set for var

ForestFire finished

Generating more randomly selected feature sets for comparison
best 5 feature sets of random selection: [[ 1.          0.          1.          0.
 ↪      0.          1.          1.
   1.          0.          0.          0.          0.          0.
   0.72666667]
 [ 1.          1.          1.          0.          0.          1.          0.
   1.          1.          0.          1.          1.          0.
   0.72333333]
 [ 0.          0.          0.          1.          1.          1.          1.
   1.          1.          1.          0.          1.          0.          0.71      ]
 [ 1.          1.          0.          0.          0.          0.          1.
   1.          1.          0.          0.          1.          1.
   0.70333333]
 [ 1.          0.          1.          0.          0.          1.          1.
   1.          1.          1.          1.          0.          1.
   0.70333333]]

Lowest MSE after 50 random SVM runs: 0.726666666667
Lowest MSE of ForestFire after 30 initial random runs and 20 guided runs: 0.
 ↪763333333333
Performance with ForestFire improved by 5.04587155963%
Execution finished

Found Best value for Random Forest Search after 30 initial runs and 11/20 smart runs
Best value with RF: 0.763333333333

Found Best value for Random Search after 18 random runs
```

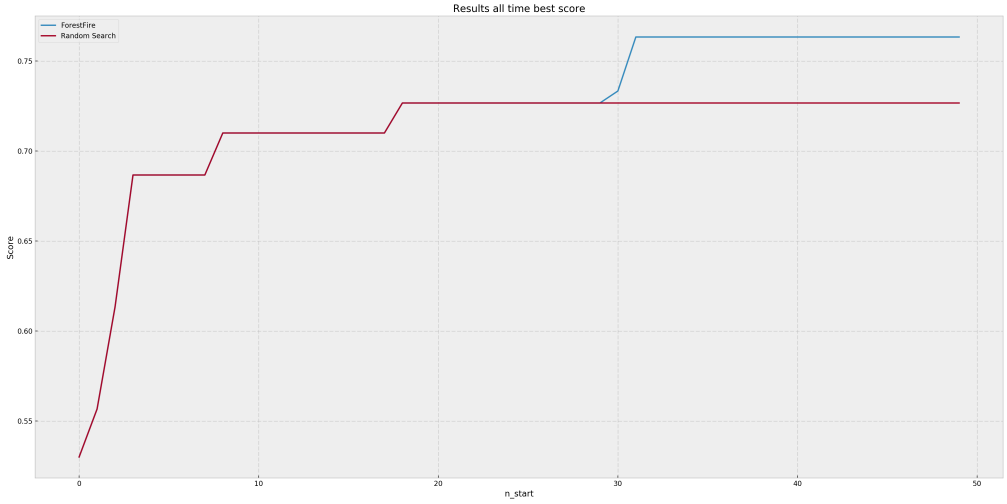
```
Best value with Random Search: 0.726666666667

Creating Plots

[Finished in xxx s]
```

Interpretation:

In this example ForestFire was able to find the best solution of 76,3% accuracy after 30 random and 11 guided runs. Compared to random search accuracy could be improved by ~5%. The best *MLA* run did "see" all features but the second.

Since Demo mode was turned on at the end two plots are produced:





Important: Functions used in this chapter

ForestFire.Main.**main_loop**(*n_start*, *pruning*, *min_data*, *n_forests*, *n_trees*, *n_configs_biased*, *n_configs_unbiased*, *multiplier_stepup*, *seen_forests*, *weight_mean*, *weight_gradient*, *scoref*, *demo_mode*, *plot_enable*)

Load raw data and Generate database for Random Forest. Iteratively build and burn down new Random Forests, predict the performance of new feature sets and compute two new feature sets per round.

Arguments:

- n_start {int} – number of runs before building first RF = number of data points in first RF; minimum = 4, default = 50
- pruning {float} – if greater than zero, branches of a Decision Tree will be pruned proportional to pruning value; default = 0
- min_data {float} – minimum percentage of Datasets that is used in RF generation; default = 0.2
- n_forests {int} – number of forests; minimum=1; default = 25
- n_trees {int} – # number of trees that stand in a forest; min = 3; default = number of features x 3 x
- n_configs_biased {int} – # number of deliberately chosen feature sets that get predicted in each forest; default = n_trees x 5
- n_configs_unbiased {int} – # number of randomly chosen feature sets that get predicted in each forest; default = n_configs_biased x0.2
- multiplier_stepup {float} – # sets how aggressively the feature importance changes; default = 0.25
- seen_forests {int} – # number of recent forests that are taken into acount for generating probability of the chosen feature sets default = 4
- weight_mean {float} – # weight of the mean in calculating the new probability for selecting future feature sets; default = 0.2
- weight_gradient {bool} – # weight of the gradient in calculating the new probability for selecting future feature sets; default = 0.8
- scoref {function} – # which scoring metric should be used in the Decision Tree (available: entropy and giniimpurity); default = entropy
- demo_mode bool – # if true a comparison between the Random Forest driven Search and a random search is done
- plot_enable bool – # decide if at the end a plot should be generated , only possible in demo mode

CHAPTER

THREE

BUILDING AND BURNING RANDOM FORESTS

3.1 Decision Tree

corresponding file: Main.py

The principle of building decision trees is based on the implementation of decision trees in *[Collective_Intelligence]* by Toby Segaran.

3.1.1 Base Class

At the foundation of the ForestFire algorithm stands the *decisionnode class*. It represents a node in a *DT* at which the decision is made into which branch (true or false) to proceed. The whole tree is built up of nodes. Each node itself can contain two more nodes - the true and false branch - which are themselves decisionnodes. In this way a tree is constructed in which a set of data takes a certain path along the tree to get classified. At each node it either enters the true or the false branch. When a branch is reached with no further branches below, this is called a leaf node. The leaf node contains the results which represent the classification a data set receives. The results can be a single value - in this case the classification is 100% this single value. It can also consist of several values, e.g. value1 with 2 instances and value2 with 1 instance. The result of this classification is ambiguous, so it is expressed as a probability: the classification is 1/3 value2 and 2/3 value1.

3.1.2 Helper Functions

At each node two questions have to be answered:

- **By which feature (=column) should the next decision be made?** The feature that is chosen at the first node should be the one feature that separates the data set in the best possible way. Latter features are of less importance
- By which value should the decision be made?

To answer those questions the data is iteratively split in every possible way. This means it is split for every feature and within every feature it is split for every single value.

See *divideset*

Each of the resulting splits has to be evaluted with respect to "how well" the split separates the big list into two smaller lists. For this three evaluation metrics can be chosen from:

- ***Gini Impurity*** "Probability that a randomly placed item will be in the wrong category"
- ***Entropy*** "How mixed is a list"
- ***Variance*** "How far apart do the numbers lie"

The evaluation metric returns the gini coefficient / entropy / variance of the list that it is presented with. Both methods need information about how many unique elements are in one list. See *uniquecounts*.

After a tree is built its width and depth can be examined by *getdepth* and *getwidth*. A tree's depth is the maximum number of decisions that can be made before reaching a leaf node plus 1 (A tree stump that has no branches by definition still has a depth of 1). A tree's width is the number of leaves it contains, i.e. number of nodes that have entries in their results property.

3.1.3 Building a tree

Starting with a root node and the whole provided data set the *buildtree* function recursively loops through the following steps and builds up the tree structure:

1. create a decisionnode
2. calculate score (entropy / gini coefficient / variance) of current list
3. divide list into every possible split
4. evaluate each split according to evaluation metric
5. split the list into true and false branches according to best evaluated split
6. If no split is better than the current list no split is performed and results are stored, tree is returned
7. If true and false branches are created, start at 1.

An example tree can look like *this*. The first node checks if the value of the third column is >= 21. If yes it continues to the right and checks column 0 if the value is equal to 'slashdot'. If yes the prediction for the new data set will be 50% None and 50% Premium since both values have appeared 1 time during trainging/building of the tree.

If the value of column 0 is instead not equal to 'slashdot', there is another query at the next node for colum 0 wether it is equal to 'google' and so on.

Pruning a tree

At the deeper levels of a tree there might be splits that further reduce the entropy / gini coefficient / variance of the data, but only to a minor degree. These further splits are not productive since they make the tree more complex but yield only small improvements. There are two ways of tackling this problem.

One is to stop splitting the data if the split does not produce a significant reduction in entropy / gini coefficient / variance. The danger in doing this is that there is a possibility that at an even later split there might be a significant reduction, but the algorithm can not forsee this. This would lead to an premature stop.

The better way of dealing with the subject of overly complex trees is *pruning*. The pruning approach builds up the whole complex tree and then starts from its leaves going up. It takes a look at the information gain that is made by the preceding split. If the gain is lower than a threshold specified by the *pruning* hyperparameter in *Execution* it will reunite the two leaves into one single leaf. This way no meaningful splits are abandoned but complexity can be reduced

In the *above example tree* the rightmost leaf is the only place where pruning might have hapenned. Before pruning 'None' and 'Premium' could have been located in separate leaves. If the information gain from splitting the two was below the defined threshold, those two leaves would get pruned into one single leaf. Still, only by looking at the finished tree one cannot tell if the tree was pruned or if it has been built this way (meaning that already during building there was no benefit in creating another split).

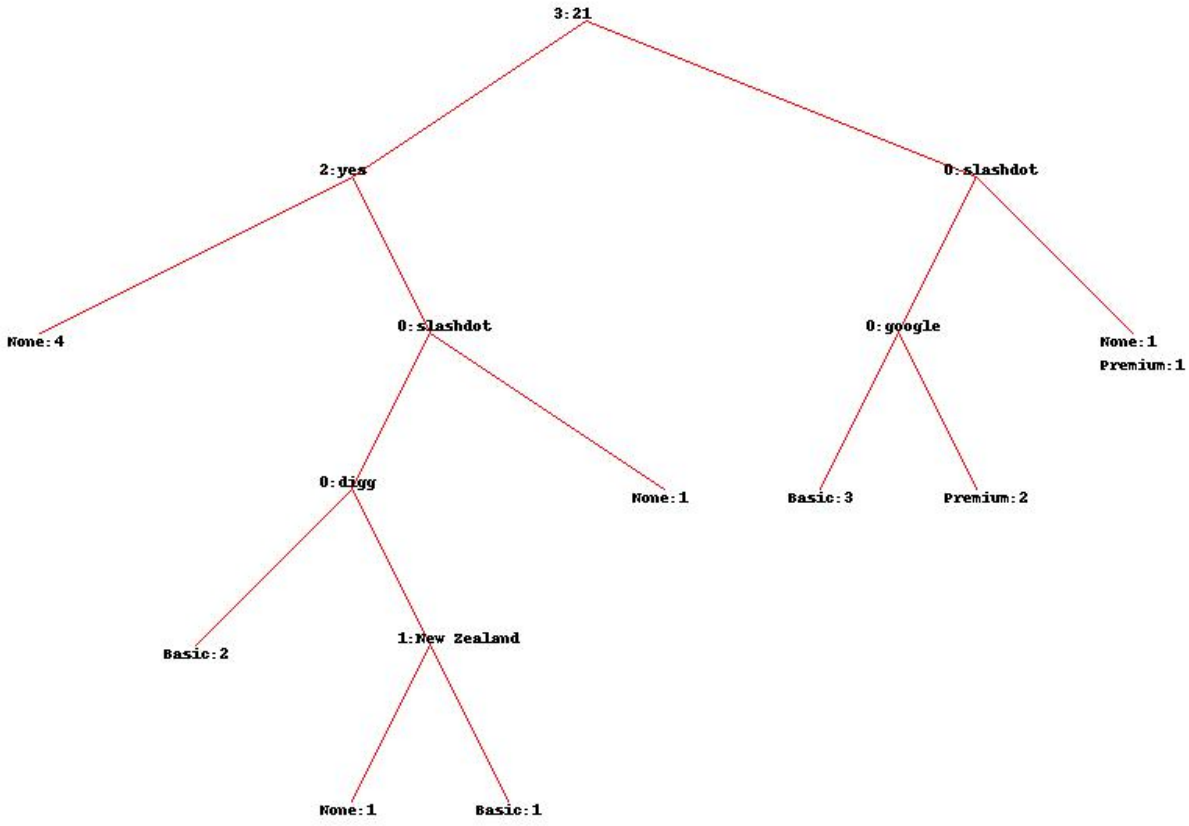


Fig. 3.1: Treeview.jpg

Warning: By default pruning is disabled (set to 0). A reasonable value for pruning depends on the raw data. Observe the output for "wrongs" on the console. By default it should be quite small (<10% of the total number of trees at most). Try a value for pruning between 0 and 1 and only increase above 1 if the "wrongs" output does not get too big.

A "wrong" tree is a tree "stump" consisting of only one node. Such a tree has no informational benefit.

Being an advanced hyperparameter pruning can greatly improve overall results as well as the number of runs it takes to find a good result. But it also increases the risk of getting stuck in a local extremum or ending up with a lot of tree 'stumps' that are useless for further information retrieval.

3.1.4 Classifying new observations

After a *DT* is built new observations can be classified. This process can vividly be explained by starting at the top node and asking a simple yes or no question about the corresponding feature and value that is stored in the node. If the answer for the new observastion is yes, the path follows the true branch of the node. In case of a negated answer the false branch is pursued.

See *Tree Image* as an example. Visually the true branch is on the right hand side of the parent node, the false branch on the left.

The classification of new data is done with the help of the *classify function*.

Note: *classify* is also able to handle missing data entries. In this case both branches are followed and the result is weighted according to the number of entries they contain. Since the ForestFire algorithm produces its own database from the raw data and the underlying *MLA* it is made sure that there are always entries present and the case of missing entries does not come to pass.

3.1.5 Visualizing a tree

The following functions are for debugging purposes only.

The structure of the tree can be output to the console with the help of *printtree*.

An image of the tree can be created with the *drawtree* function. It makes use of *drawnode*.

3.1.6 Storing the tree structure

To *grow a Random Forest from single Decision Trees* there must be a way to store whole trees and their structure in an array. Unlike *printtree* and *drawtree* where the tree is printed / drawn recursively by looping through the nodes.

This is done with the help of *path_gen* and *path_gen2*. By examining the last column of the path matrix that is returned by *path_gen* all results of the different leaf nodes can be reached.

Another usefull function is *check_path*. It takes as input a tree and a result (typically extracted from a path matrix) and checks wether the result is in that tree. This way it is possible to move along the branches of a tree and at each node check if it (still) contains a certain result, e.g. the best result of the whole tree. This is used for determining the importance of features in the following chapter about *growing a Random Forest*

Important: Functions used in this chapter

class ForestFire.Main.**decisionnode**(*col=-1*, *value=None*, *results=None*, *tb=None*, *fb=None*)
Base class that a decision tree is built of.

Keyword Arguments:

- col {integer} – column number = decision criterium for splitting data (default: {-1})
- value {integer/float/string} – value by which data gets split (default: {None})
- results {integer/float/string} – if node is an end node (=leaf) it contains the results (default: {None})
- tb {decisionnode} – next smaller node containing the true branch (default: {None})
- fb {decisionnode} – next smaller node containing the false branch (default: {None})

ForestFire.Main.**divideset**(*rows*, *column*, *value*)
splits a data set into two separate sets according to the column and the value that is passed into.

If value is a number the comparison is done with <= and >=. If value is not a number the exact value is compared

Arguments:

- rows {list} – data set that is split
- column{integer} – column by which data gets split
- value {number/string} – value by which data gets split

Returns: [list] – two listso

ForestFire.Main.**giniimpurity**(*rows*)
Probability that a randomly placed item will be in the wrong category

Calculates the probability of each possible outcome by dividing the number of times that outcome occurs by the total number of rows in the set. It then adds up the products of all these probabilities. This gives the overall chance that a row would be randomly assigned to the wrong outcome. The higher this probability, the worse the split.

Returns: float – probability of being in the wrong category

ForestFire.Main.**entropy**(*rows*)
Entropy is the sum of p(x)log(p(x)) across all the different possible results –> how mixed is a list

Funciton calculates the frequency of each item (the number of times it appears divided by the total number of rows) and applies these formulas:

$$p(i) = frequency(outcome) = \frac{count(outcome)}{count(totalrows)}$$

$$Entropy = \sum(p(i)) \cdot \log(p(i)) \; for\ all\ outcomes$$

The higher the entropy, the worse the split.

Arguments: rows {list} – list to evaluate

Returns: [float] – entropy of the list

ForestFire.Main.**variance**(*rows*)
Evaluates how close together numerical values lie

Calculates mean and variance for given list

$$mean = \frac{\sum(entries)}{number\ of\ entries}$$

$$variance = \sum(entry - mean)^2$$

Arguments: rows {list} – list to evaluate

Returns: number – variance of the list

ForestFire.Main.**uniquecounts**(*rows*)
evaluate how many unique elements are in a given list

Arguments: rows {list} – evaluated list

Returns: integer – number of unique elements

ForestFire.Main.**getdepth**(*tree*)
returns the maximum number of consecutive nodes

Arguments: tree {decisionnode} – tree to examine

Returns: number – maximum number of consecutive nodes

ForestFire.Main.**getwidth**(*tree*)
returns the number of leaves = endnodes in the tree

Arguments: tree {decisionnode} – tree to examine

Returns: number – number of endnodes

ForestFire.Main.**buildtree**(*rows*, *scoref*)
recursively builds decisionnode objects that form a decision tree

At each node the best possible split is calculated (depending on the evaluation metric). If no further split is neccessary the remaining items and their number of occurence are written in the results property.

Arguments: rows {list} – dataset from which to build the tree scoref {function} – evaluation metric (entropy / gini coefficient)

Returns: decisionnode – either two decisionnodes for true and false branch or one decisionnode with results (leaf node)

ForestFire.Main.**prune**(*tree*, *mingain*)
prunes the leaves of a tree in order to reduce complexity

By looking at the information gain that is achieved by splitting data further and further and checking if it is above the mingain threshold, neighbouring leaves can be collapsed to a single leaf.

Arguments: tree {decisionnode} – tree that gets pruned mingain {number} – threshold for pruning

ForestFire.Main.**printtree**(*tree*, *indent=' '*)
prints out the tree on the command line

Arguments: tree {decisionnode} – tree that gets printed

ForestFire.Main.**drawtree**(*tree*, *jpeg='tree.jpg'*)
visualization of the tree in a jpeg

Arguments: tree {decisionnode} – tree to draw

Keyword Arguments: jpeg {str} – Name of the .jpg (default: {'tree.jpg'})

ForestFire.Main.**drawnode**(*draw*, *tree*, *x*, *y*)
Helper Function for drawtree, draws a single node

Arguments: draw {img} – node to be drawn tree {decisionnode} – tree that the node belongs to x {number} – x location y {number} – y location

ForestFire.Main.**classify**(*observation*, *tree*)
takes a new data set that gets classified and the tree that determines the classification and returns the estimated result.

Arguments: observation {numpy.array} – the new data set that gets classified, e.g. test data set tree {decisionnode} – tree that observation gets classified in

Returns: data – expected result

ForestFire.Main.**path_gen**(*tree*)

Create a path Matrix which contains the structure of the tree. Calls path_gen2 to do so.

Arguments: tree {decisionnode} – tree of which the data structure is stored

Returns: numpy.array – data structure of the tree, NaN means there is no more branch

ForestFire.Main.**path_gen2**(*tree*, *width*, *depth*, *path*, *z2*, *z1*)

Create a path Matrix which contains the structure of the tree.

creates a matrix 'path' that represents the structure of the tree and the decisions made at each node, last column contains the average MSE at that leaf the sooner a feature gets chosen as a split feature the more important it is (the farther on the left it appears in path matrix) order that leaves are written in (top to bottom): function will crawl to the rightmost leaf first (positive side), then jump back up one level and move one step to the left (loop)

Arguments: tree {decisionnode} – tree of which the data structure is stored width {int} – width of the tree depth {int} – depth of the tree path {[type]} – current path matrix, gets updated during function calls z2 {int} – control variable for current depth z1 {int} – control variable for current width

Returns: numpy.array – the structure of the tree

ForestFire.Main.**check_path**(*tree*, *result*)

Check if a tree contains MSE_min (= True) or not (= False)

Arguments: tree {decisionnode} – tree that gets searched for result result {data} – result that the tree is searched for

Returns: bool – True if result is in the tree, false if not

ForestFire.Main.**main_loop**(*n_start*, *pruning*, *min_data*, *n_forests*, *n_trees*, *n_configs_biased*, *n_configs_unbiased*, *multiplier_stepup*, *seen_forests*, *weight_mean*, *weight_gradient*, *scoref*, *demo_mode*, *plot_enable*)

Load raw data and Generate database for Random Forest. Iteratively build and burn down new Random Forests, predict the performance of new feature sets and compute two new feature sets per round.

Arguments:

- n_start {int} – number of runs before building first RF = number of data points in first RF; minimum = 4, default = 50
- pruning {float} – if greater than zero, branches of a Decision Tree will be pruned proportional to pruning value; default = 0
- min_data {float} – minimum percentage of Datasets that is used in RF generation; default = 0.2
- n_forests {int} – number of forests; minimum=1; default = 25
- n_trees {int} – # number of trees that stand in a forest; min = 3; default = number of features x 3 x
- n_configs_biased {int} – # number of deliberately chosen feature sets that get predicted in each forest; default = n_trees x 5
- n_configs_unbiased {int} – # number of randomly chosen feature sets that get predicted in each forest; default = n_configs_biased x0.2
- multiplier_stepup {float} – # sets how aggressively the feature importance changes; default = 0.25
- seen_forests {int} – # number of recent forests that are taken into acount for generating probability of the chosen feature sets default = 4

•weight_mean {float} – # weight of the mean in calculating the new probability for selecting future feature sets; default = 0.2

•weight_gradient {bool} – # weight of the gradient in calculating the new probability for selecting future feature sets; default = 0.8

•scoref {function} – # which scoring metric should be used in the Decision Tree (available: entropy and giniimpurity); default = entropy

•demo_mode bool – # if true a comparison between the Random Forest driven Search and a random search is done

•plot_enable bool – # decide if at the end a plot should be generated , only possible in demo mode

3.2 Random Forest

corresponding file: Main.py

3.2.1 Why a single Decision Tree is not enough

A single Decision Tree is already a fully fledged classifier that can be used to determine which features are of more importance than the rest. The higher up in the hirarchy of the tree a feature stands the more decisive it is with regard to how well it splits the data in two separate lists. The feature at the top node of a tree can be considered the most important one, a feature that appears on the lower levels is not as importnant. Consequently a feature that is not at all appearing in the tree is even less important - it is even possible it distorts performance of the *MLA*. As a consequence it might be reasonable to leave features with little importance out of the *MLA* and only present it with the important ones.

Applied to a convenient data set this approach can work. But there are challenges that arise in most real world data sets. The data can be clustered, i.e. a number of subsequent data sets might follow a certain pattern that is overlooked by a single Decision Tree because it is presented with all the data sets combined. In addition a single tree that sees all features of the data set tends to be biased towards the most dominant features.

A logical implication to these two challenges is to introduce randomness:

- present a single tree with only a random subset of all data sets
- present a single tree with only a random subsmet of all features

This will reduce the bias of the tree, but increase its variance. By building multiple trees and averaging their results the variance can again be reduced. The term for this construct is *Random Forest*.

3.2.2 Growing a Random Forest

In *buildforest* the Random Forest is built according to the following steps:

1. select random data and feature sets from the *generated Database*
2. build a single tree with “limited view”
3. *prune* the tree (if *enabled*)
4. reward features that lead to the best result
5. punish features that don’t lead to the best result
6. Build next Tree

After a new tree is built the feature importance for the whole Forest is *updated* according to the number of appearances in the single trees. The higher up a feature gets selected in a tree the higher it is rated. The punishment for features that don't lead to the best results is weaker than the reward for leading to the best results. Features that are not included in the tree get neither a positive nor a negative rating. Instead their probability of getting chosen as a biased feature set in *Predicting new feature sets* is set to zero.

3.2.3 Extracting the feature Importance from a Random Forest

From the so far completed Random Forests the resulting importance of each single feature can be extracted. The terms "importance" and "probability" of a feature are used synonymously, since this value will be used for selecting new feature sets in *Predicting new feature sets*.

In *update_prob* the current feature importance / probability is calculated. Therefore several parameters are taken into account:

- seen_forests: Only a fix number of recent Forest is taken into consideration
- weight_mean: From the last seen_forest Forests the mean of each feature is calculated and weighed accordingly
- weight_gradient: From the last seen_forest Forests the gradient of each feature is calculated and weighed accordingly
- multiplier: each feature probability is potentized by the current multiplier in order to achieve a more distinct distribution of the probabilites
- prob_current: the resulting probability for a feature is a combination of its recent trends for both gradient and mean (for details see *update_prob*)

Multiplier Stepup

The multiplier that is *applied* as an exponent to all single feature probabilities is a quantity that is scaled dynamically. Depending on the *Raw data set* set it is possible that the feature importances in a Random Forest are all very close to the average importance, hence resembling nothing more than a randomly chosen distribution. In order to avoid this *ForestFire* examines the importances of every single feature after a Random Forest is built. If the highest feature importance does not lie above a certain threshold (default: 2 times the average importance) the multiplier is raised by the *hyperparameter multiplier_stepup*.

3.2.4 Predicting new feature sets

After the forest is built it can be used to make predictions (see *forest_predict*) about the performance of arbitrary feature sets. A new feature set candidate gets classified in every single forest. The results are averaged. From the vast amount of possible feature sets two different groups of feature sets are considered:

- feature sets biased according to the average importance of each feature (prob_current from *update_prob*)
- entirely randomly chosen feature sets

The two *hyperparameters* *n_configs_biased* and *n_configs_unbiased* determine the amount of feature sets that get tested.

For selecting the biased feature sets the probability of choosing a particular feature depends on its rating calculated in *buildforest*. The unbiased feature sets are chosen randomly.

Every candidate for future computation in the *MLA* gets predicted in every tree that stands in the *Random Forest*. The results are incorporated by their average (mean) and variance.

Of all predicted feature sets two are chosen for the next computing run with the *MLA*. One with a high average (mean) and one with a high variance (respectively a combination of both, for details see *forest_predict*).

If a feature set has already been computed before, it will not be computed again. Instead its result is copied to the database.

The *Updating of the database* depicts the last step in the ForestFire Loop.

Important: Functions used in this chapter

`ForestFire.Main.`**`buildforest`**(*data*, *n_trees*, *scoref*, *n_feat*, *min_data*, *pruning*)

Growing the Random Forest

The Random Forest consists of n_trees. Each tree sees only a subset of the data and a subset of the features. Important: a tree never sees the original data set, only the performance of the classifying algorithm For significant conclusions enough trees must be generated in order to gain the statistical benefits that overcome bad outputs

Arguments:

- data {numpy.array} – data set the Forest is built upon
- n_trees {int} – number of trees in a Decision tree
- scoref {function} – scoring metric for finding new nodes
- n_feat {int} – number of features in data
- min_data {float} – minimum percentage of all data sets that a tree will see
- pruning {bool} – pruning enabled (>0) / disabled(=0)

Returns:

- RF – dictionary = importances of single features in the forest
- prob_current – single value for importance, used for generating new biased feature sets
- trees – contains all single trees that stand in the Forest

`ForestFire.Main.`**`update_RF`**(*RF*, *path*, *tree*, *rand_feat*)

for each tree the features that lead to the leaf with the lowest Error will get rewarded. Features that don't lead to the leaf with the lowest Error will get punished (only by 20% of the amount the "good" featurtes get rewarded).

RF is a dictionary that gets updated after a new tree is built and thus contains the cummulation of all feature appearences in the whole forest.

Arguments:

- RF {dict} – dictionary that counts occurrence / absence of different features
- path {numpy.array} – structure of the current tree
- tree {decisionnode} – tree that gets examined
- rand_feat {list} – boolean mask of selected features (1 = selected, 0 = not selected)

Returns:

- RF – updated dictionary that counts occurrence / absence of different features

`ForestFire.Main.`**`update_prob`**(*Probability*, *i*, *weight_mean*, *weight_gradient*, *multiplier*, *seen_forests*)

Calculates the current Importance / Probability of the single features

Based on the probabilities of each feature in past Forests a new current_prob is calculated that takes into account the mean and the gradient of the prior feature importances.

Arguments:

- Probability {numpy array} – contains Importances of single features for all past Random Forests
- i {integer} – number of current Forest
- weight_mean {float} – weight of the mean in calculating resulting probability
- weight_gradient {float} – weight of the var in calculating resulting probability
- multiplier {float} – exponent for amplifying probabilities
- seen_forests {integer} – number of before built forest that are considered

Returns: prob_current – list of floats representing the calculated aggregation of recent feature importances

ForestFire.Main.**forest_predict**(*data*, *trees*, *prob*, *n_configs*, *biased*)

Predict performance of new feature sets

Predicts biased and unbiased feature sets in the before constructed Random Forest. Feature sets are predicted in every single Decision Tree in the Random Forest. Results are represented as (mean+0.1*var) and (variance+0.1*mean) for each feature set. The two best feature sets are selected to be sent into the *MLA*.

Arguments:

- data {numpy.array} – contains all previous computing runs
- trees {decisionnodes} – the trees that make up the Random Forest
- prob {array of floats} – probability that a feature gets chosen into a feature set
- n_configs {int} – number of feature sets to be generated
- biased {bool} – true for biased feature selection, false for unbiased feature selection

Returns:

- best mean – highest average of all predicted feature sets
- best feature set mean – corresponding boolean list of features (0=feature not chosen, 1=feature chosen)
- best var – highest variance of all predicted feature sets
- best feature set var – corresponding boolean list of features (0=feature not chosen, 1=feature chosen)

ForestFire.Main.**main_loop**(*n_start*, *pruning*, *min_data*, *n_forests*, *n_trees*, *n_configs_biased*, *n_configs_unbiased*, *multiplier_stepup*, *seen_forests*, *weight_mean*, *weight_gradient*, *scoref*, *demo_mode*, *plot_enable*)

Load raw data and Generate database for Random Forest. Iteratively build and burn down new Random Forests, predict the performance of new feature sets and compute two new feature sets per round.

Arguments:

•n_start {int} – number of runs before building first RF = number of data points in first RF; minimum = 4, default = 50

•pruning {float} – if greater than zero, branches of a Decision Tree will be pruned proportional to pruning value; default = 0

•min_data {float} – minimum percentage of Datasets that is used in RF generation; default = 0.2

•n_forests {int} – number of forests; minimum=1; default = 25

•n_trees {int} – # number of trees that stand in a forest; min = 3; default = number of features x 3 x

•n_configs_biased {int} – # number of deliberately chosen feature sets that get predicted in each forest; default = n_trees x 5

- n_configs_unbiased {int} – # number of randomly chosen feature sets that get predicted in each forest; default = n_configs_biased x0.2
- multiplier_stepup {float} – # sets how aggressively the feature importance changes; default = 0.25
- seen_forests {int} – # number of recent forests that are taken into acount for generating probability of the chosen feature sets default = 4
- weight_mean {float} – # weight of the mean in calculating the new probability for selecting future feature sets; default = 0.2
- weight_gradient {bool} – # weight of the gradient in calculating the new probability for selecting future feature sets; default = 0.8
- scoref {function} – # which scoring metric should be used in the Decision Tree (available: entropy and giniimpurity); default = entropy
- demo_mode bool – # if true a comparison between the Random Forest driven Search and a random search is done
- plot_enable bool – # decide if at the end a plot should be generated , only possible in demo mode

3.3 Update Database

In chapter *Random Forest* the process of choosing two new feature sets is described. Those newly chosen feature sets and their performance are added to the initially *generated database*.

See *update_database* for details.

Important: Functions used in this chapter

`ForestFire.Main.`**`update_database`**(*X*, *y*, *data*, *mask_best_featureset*, *X_test*, *y_test*)

Appends newly tested feature sets and their result to the already calculated feature sets

Arguments:

- X {numpy array} – X rat data sets
- y {numpy array} – y raw data sets
- data {[type]} – data set the Forest is built upon
- mask_best_featureset {bool} – feature set (1: feature contained, 0: feature not contained)
- X_test {numpy array} – test data set
- y_test {numpy array} – test data set

Returns: data – updated data base

`ForestFire.Main.`**`main_loop`**(*n_start*, *pruning*, *min_data*, *n_forests*, *n_trees*, *n_configs_biased*, *n_configs_unbiased*, *multiplier_stepup*, *seen_forests*, *weight_mean*, *weight_gradient*, *scoref*, *demo_mode*, *plot_enable*)

Load raw data and Generate database for Random Forest. Iteratively build and burn down new Random Forests, predict the performance of new feature sets and compute two new feature sets per round.

Arguments:

- n_start {int} – number of runs before building first RF = number of data points in first RF; minimum = 4, default = 50

- pruning {float} – if greater than zero, branches of a Decision Tree will be pruned proportional to pruning value; default = 0
- min_data {float} – minimum percentage of Datasets that is used in RF generation; default = 0.2
- n_forests {int} – number of forests; minimum=1; default = 25
- n_trees {int} – # number of trees that stand in a forest; min = 3; default = number of features x 3 x
- n_configs_biased {int} – # number of deliberately chosen feature sets that get predicted in each forest; default = n_trees x 5
- n_configs_unbiased {int} – # number of randomly chosen feature sets that get predicted in each forest; default = n_configs_biased x0.2
- multiplier_stepup {float} – # sets how aggressively the feature importance changes; default = 0.25
- seen_forests {int} – # number of recent forests that are taken into acount for generating probability of the chosen feature sets default = 4
- weight_mean {float} – # weight of the mean in calculating the new probability for selecting future feature sets; default = 0.2
- weight_gradient {bool} – # weight of the gradient in calculating the new probability for selecting future feature sets; default = 0.8
- scoref {function} – # which scoring metric should be used in the Decision Tree (available: entropy and giniimpurity); default = entropy
- demo_mode bool – # if true a comparison between the Random Forest driven Search and a random search is done
- plot_enable bool – # decide if at the end a plot should be generated , only possible in demo mode

CHAPTER

FOUR

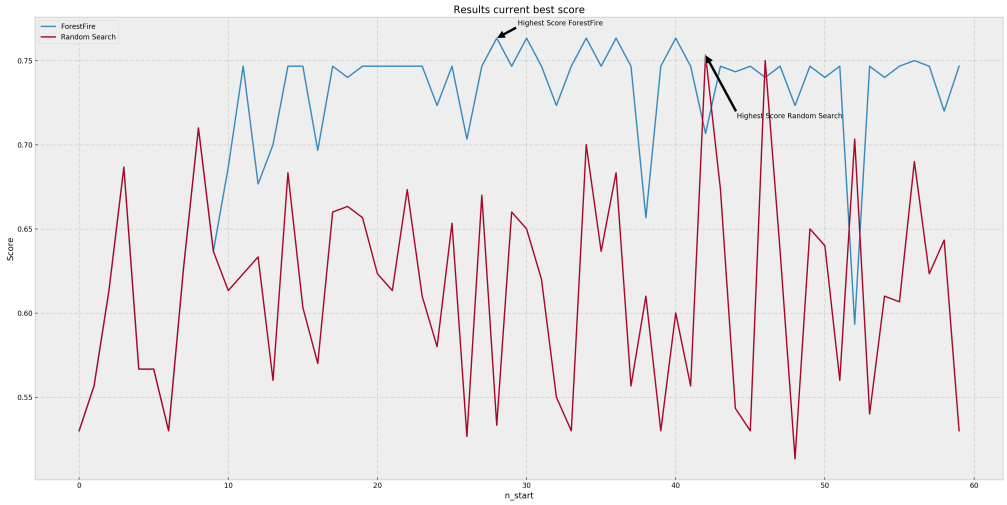
EVALUATION MODE

4.1 Evaluation Mode

If the *hyperparameter* *demo_mode* is set to *True* the performance of ForestFire can be compared to a randomized search of new feature sets. For every new feature set that gets calculated with ForestFire a randomly generated feature set is calculated. Information about the performance is printed out after both ForestFire and the randomized search are done.

4.1.1 Plots

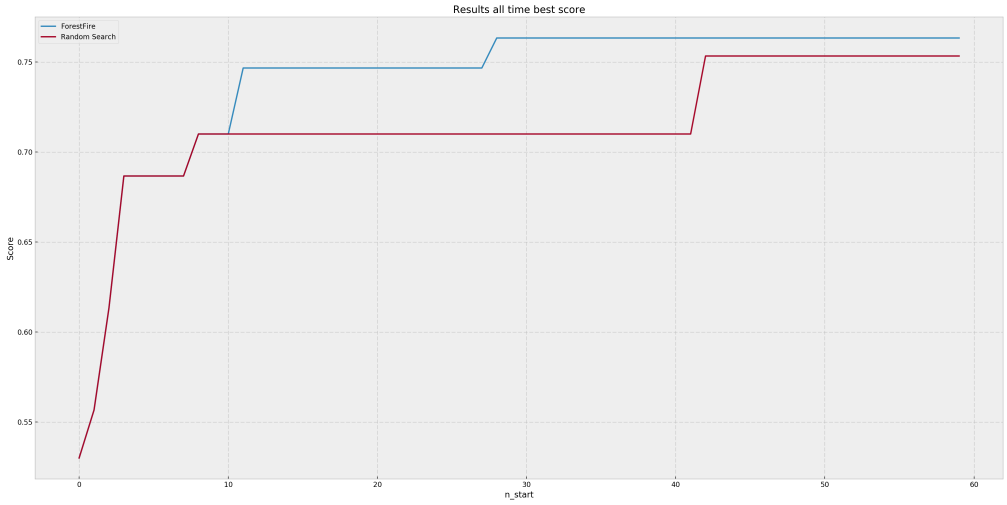
If the *hyperparameter* *plot_enable* is set to *True*, the performance of both are plotted over the number of generated feature sets. Example Plots look like this:



Important: Functions used in this chapter

`ForestFire.Main.`**`main_loop`**(*n_start*, *pruning*, *min_data*, *n_forests*, *n_trees*, *n_configs_biased*, *n_configs_unbiased*, *multiplier_stepup*, *seen_forests*, *weight_mean*, *weight_gradient*, *scoref*, *demo_mode*, *plot_enable*)

Load raw data and Generate database for Random Forest. Iteratively build and burn down new Random Forests, predict the performance of new feature sets and compute two new feature sets per round.



Arguments:

- n_start {int} – number of runs before building first RF = number of data points in first RF; minimum = 4, default = 50
- pruning {float} – if greater than zero, branches of a Decision Tree will be pruned proportional to pruning value; default = 0
- min_data {float} – minimum percentage of Datasets that is used in RF generation; default = 0.2
- n_forests {int} – number of forests; minimum=1; default = 25
- n_trees {int} – # number of trees that stand in a forest; min = 3; default = number of features x 3 x
- n_configs_biased {int} – # number of deliberately chosen feature sets that get predicted in each forest; default = n_trees x 5
- n_configs_unbiased {int} – # number of randomly chosen feature sets that get predicted in each forest; default = n_configs_biased x0.2
- multiplier_stepup {float} – # sets how aggressively the feature importance changes; default = 0.25
- seen_forests {int} – # number of recent forests that are taken into acount for generating probability of the chosen feature sets default = 4
- weight_mean {float} – # weight of the mean in calculating the new probability for selecting future feature sets; default = 0.2
- weight_gradient {bool} – # weight of the gradient in calculating the new probability for selecting future feature sets; default = 0.8
- scoref {function} – # which scoring metric should be used in the Decision Tree (available: entropy and giniimpurity); default = entropy
- demo_mode bool – # if true a comparison between the Random Forest driven Search and a random search is done
- plot_enable bool – # decide if at the end a plot should be generated , only possible in demo mode

CHAPTER

FIVE

SOURCE CODE

5.1 Source Code

In this section the complete source Code is presented in html friendly style.

5.1.1 import_data.py

```
import numpy as np


def import_data():
    """import the raw data from two numpy arrays.

    Import raw data from two numpy arrays X.npy and y.npy.
    Set how train and test data are to be split for fix splits.
    Returns train/test splits as well as number of features.

    Returns:

        * X_test {np.array} -- result training data
        * X_train {np.array} -- feature training data
        * y_test {np.array} -- result test data
        * y_train {np.array} -- result training data

    """
    X = np.load('X.npy')
    X_train = X[:500]
    X_test = X[500:800]
    y = np.load('y.npy')
    y_train = y[:500]
    y_test = y[500:800]
    n_feat = len(X[0])

    return X_test, X_train, y_test, y_train, n_feat
```

5.1.2 compute.py

```
# Imports
import numpy as np
from sklearn import svm
from sklearn.metrics import accuracy_score
```

```
from sklearn.model_selection import GridSearchCV

# make sure that a high score is better than a low score! If you use accuracy, a high␣
↪accuracy is better than a low
# one. If you use Error (e.g. MSE) make sure it is negative (negative MSE)!


def compute(X_train, y_train, mask_sub_features, X_test, y_test):
    """Computes a new dataset for the Random Forest with the underlying machine␣
↪learning algorithm.

    Configure your machine learning algorithm here.
    Add imports at the top of the file.
    If no train / test split is done during import, X_train and X_test are equal (y_
↪train and y_test as well).
    In this case define your own splits with your machine learning algorithm.

    Arguments:

        * X_train {np.array} -- feature training data
        * y_train {np.array} -- result training data
        * mask_sub_features {np.array} -- feature set = dedicated part of all features
        * X_test {np.array} -- result training data
        * y_test {np.array} -- result test data

    Returns:
        score {np.float64} -- score of the selected feature set
    """

    # insert your own machine learning algorithm #
    param_grid = [{'C': np.logspace(-1, 1, 16), 'gamma': np.logspace(-1, 1, 16)}]
    clf = svm.SVC()  # SVR for regression, SVC for classification
    grid = GridSearchCV(clf, param_grid, cv=None, n_jobs=-1, scoring='neg_mean_
↪squared_error', pre_dispatch=8)
    grid.fit(X_train, y_train)
    y_pred = grid.predict(X_test[:, mask_sub_features])

    # store the result in score #
    score = accuracy_score(y_test, y_pred)
    # print score
    return score
    # print grid.cv_results_
    # print (grid.grid_scores_)
    # print(grid.best_score_)
    # print(grid.best_params_)
    # return grid.best_score_
```

5.1.3 run_ForestFire.py

```
import Main
import numpy as np

name = '__main__'
# Hyperparameters #

# number of runs before building first Random Forest = number of data points in first␣
↪RF; minimum = 4, default = 50
```

```
# adjust according to computational capabilities and demands of the underlying␣
↪machine learning algorithm
n_start = 30  # default = 30e
# if pruning is greater than zero, branches of a Decision Tree will be pruned␣
↪proportional to pruning value; default = 0
# advanced parameter. If set too high, all trees will be cut down to stumps. Increase␣
↪carefully. Start with values between 0 and 1.
pruning = 0.3
# minimum percentage of Datasets that is used in RF generation; default = 0.2
min_data = 0.1
# number of forests; minimum=1;  default = 25
# adjust according to computational capabilities. For each forest two new␣
↪computational runs are done. default = 20
n_forests = 85

# number of trees that stand in a forest; min = 3; default = number of features * 3
n_trees = 'default'
# number of deliberately chosen feature sets that get predicted in each forest;␣
↪default = n_trees * 5
n_configs_biased = 'default'
# number of randomly chosen feature sets that get predicted in each forest; default =␣
↪n_configs_biased * 0.2
n_configs_unbiased = 'default'
# sets how aggressively the feature importance changes; default = 0.25
# higher values will increase pressure on how often promising features will be␣
↪selected.
# advanced parameter, adjust carefully. If set too high the risk of runnning into␣
↪local extrema rises.
multiplier_stepup = 'default'
# number of recent forests that are taken into acount for generating probability of␣
↪the chosen feature sets default = 0.01
seen_forests = 'default'
# the chosen feature sets default = 4

# weight of the mean in calculating the new probability for selecting future feature␣
↪sets; default = 0.2
weight_mean = 'default'
# weight of the gradient in calculating the new probability for selecting future␣
↪feature sets; default = 0.8
weight_gradient = 'default'

# which scoring metric should be used in the Decision Tree (available: entropy,␣
↪giniimpurity and variance); default = entropy
# select variance for numerical values in y only
scoref = 'entropy'
# set random seed for repeatabilit; comment out if no repeatability is required;␣
↪default = 1
np.random.seed(10)

# if true a comparison between the Random Forest driven Search and a random search is␣
↪done
demo_mode = True
# decide if at the end a plot should be generated , only valid in demo mode
plot_enable = True

if name == '__main__':
    Main.main_loop(n_start, pruning, min_data, n_forests, n_trees, n_configs_biased,␣
↪n_configs_unbiased, multiplier_stepup, seen_forests,
```

```
                    weight_mean, weight_gradient, scoref, demo_mode, plot_enable)
```

5.1.4 Main.py

```
#  Imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing

# For Debugging only
from PIL import Image, ImageDraw

# Import files
from compute import compute
from import_data import import_data

# matplotlib.use('TkAgg')  # set Backend


# change settings
np.set_printoptions(threshold=np.inf)    # print whole numpy array in console
np.seterr(divide='ignore', invalid='ignore')  # ignore warnings if dividing by zero␣
↪or NaN
plt.style.use('bmh')
plt.rcParams.update({'font.size': 25})

# Definitions #


def gen_database(n_start, X, y, X_test, y_test):
    """Runs the underlying :ref:`MLA <MLA>` *n_start* times to generate a database␣
↪from which Random Forests can be built.

    Arguments:
        * n_start {int} -- number of times the underlying :ref:`MLA <MLA>` is executed
        * X {numpy.array} -- raw data
        * y {numpy.array} -- raw data
        * X_test {numpy.array} -- test data
        * y_test {numpy.array} -- test data

    Returns:
        [numpy.array] -- data set containing feature sets and corresponding results
    """
    X_DT = np.zeros((n_start, len(X[0])), dtype=bool)  # Prelocate Memory
    # print X_DT
    y_DT = np.zeros((n_start, 1))  # Prelocate Memory

    # create SVMs that can only see subset of features
    for i in range(n_start):
        # create random mask to select subgroup of features
        mask_sub_features = np.zeros(len(X[0]), dtype=bool)  # Prelocate Memory
        # mask_sub_data = np.zeros(len(X), dtype=bool)  # Prelocate Memory
        # selecting features: any number between 1 and all features are selected
        size = np.random.choice(range(len(X[0]) - 1)) + 1
        rand_feat = np.random.choice(range(len(X[0])), size=size, replace=True,␣
↪p=None)  # in first run prob is None --> all features are equally selected, in␣
↪later runs prob is result of previous RF results
```

```
        mask_sub_features[rand_feat] = True  # set chosen features to True

        # Select Train and Test Data for subgroup
        # print X
        X_sub = X[:, mask_sub_features]  # select only chosen features (still all␣
↪datasets)
        # print len(X_sub[0])
        # print X_sub[0]

        # compute subgroup
        # print X_sub
        y_DT[i] = compute(X_sub, y, mask_sub_features, X_test, y_test)

        # Save Data
        X_DT[i] = mask_sub_features  # for the Decision Tree / Random Forest the X␣
↪values are the information about whether an SVM has seen a certain feature or not
    # print X_DT
    # print y_DT

    # merge X and y values
    Data = np.concatenate((X_DT, y_DT), axis=1)  # this Dataset goes into the␣
↪Decision Tree / Random Forest
    return Data


# Functions for Generating Database for RF


# Decision Tree


# class definition
class decisionnode:
    """Base class that a decision tree is built of.


    Keyword Arguments:
        * col {integer} -- column number = decision criterium for splitting data␣
↪(default: {-1})
        * value {integer/float/string} -- value by which data gets split (default:
↪{None})
        * results {integer/float/string} -- if node is an end node (=leaf) it␣
↪contains the results (default: {None})
        * tb {decisionnode} -- next smaller node containing the true branch (default:
↪{None})
        * fb {decisionnode} -- next smaller node containing the false branch␣
↪(default: {None})
    """

    def __init__(self, col=-1, value=None, results=None, tb=None, fb=None):
        self.col = col
        self.value = value
        self. results = results
        self.tb = tb
        self.fb = fb

# Functions for DT
```

```
# Divides a set on a specific column. Can handle numeric
# or nominal vlaues
def divideset(rows, column, value):
    """ splits a data set into two separate sets according to the column and the␣
↪value that is passed into.

    If value is a number the comparison is done with <= and >=.
    If value is not a number the exact value is compared

    Arguments:
        * rows {list} -- data set that is split
        * column{integer} -- column by which data gets split
        * value {number/string} -- value by which data gets split

    Returns:
        [list] -- two listso
    """
    split_function = None  # Prelocate
    if isinstance(value, int) or isinstance(value, float):
        def split_function(row):
            return row[column] >= value  # quick function definition
    else:
        def split_function(row):
            return row[column] == value
    # divide the rows into two sets and return them
    set1 = [row for row in rows if split_function(row)]  # positive side >= or ==
    set2 = [row for row in rows if not split_function(row)]  # negative side True or␣
↪False
    return (set1, set2)


# Create counts of possible results (the last column of each row is the result) = how␣
↪many different results are in a list
def uniquecounts(rows):
    """evaluate how many unique elements are in a given list

    Arguments:
        rows {list} -- evaluated list

    Returns:
        integer -- number of unique elements
    """
    results = {}
    for row in rows:
        # The result is the last column
        r = row[len(row) - 1]
        # if r not already in results, entry will be generated
        if r not in results:
            results[r] = 0
        # increase count of r by one
        results[r] += 1
    return results


def giniimpurity(rows):
    """ Probability that a randomly placed item will be in the wrong category
```

```
    Calculates the probability of each possible outcome by dividing the number of␣
↪times that outcome occurs
    by the total number of rows in the set.
    It then adds up the products of all these probabilities.
    This gives the overall chance that a row would be randomly assigned to the wrong␣
↪outcome.
    The higher this probability, the worse the split.

    Returns:
        float -- probability of being in the wrong category
    """
    total = len(rows)
    counts = uniquecounts(rows)
    imp = 0
    for k1 in counts:
        p1 = float(counts[k1]) / total
        for k2 in counts:
            if k1 == k2:
                continue
            p2 = float(counts[k2]) / total
            imp += p1 * p2
    return imp


def entropy(rows):
    """Entropy is the sum of p(x)log(p(x)) across all the different possible results -
↪-> how mixed is a list

    Funciton calculates the frequency of each item (the number of times it appears␣
↪divided by the total number of rows)
    and applies these formulas:

    .. math::
        p(i) = frequency(outcome) = \dfrac{count(outcome)}{count(total rows)}

        Entropy = \sum(p(i)) \cdot  \log(p(i)) \ for \ all \ outcomes


    The higher the entropy, the worse the split.

    Arguments:
        rows {list} -- list to evaluate

    Returns:
        [float] -- entropy of the list
    """
    from math import log

    def log2(x):
        return log(x) / log(2)
    results = uniquecounts(rows)
    # calculate Entropy
    ent = 0.0
    for r in results.keys():
        p = float(results[r]) / len(rows)
        ent -= p * log2(p)
    return ent
```

```
def variance(rows):
    """Evaluates how close together numerical values lie

    Calculates mean and variance for given list

    .. math::
        mean = \dfrac{\sum(entries)}{number \ of \ entries}

        variance = \sum(entry - mean) ^ 2

    Arguments:
        rows {list} -- list to evaluate

    Returns:
        number -- variance of the list
    """
    if len(rows) == 0:
        return 0
    data = [float(row[len(row) - 1]) for row in rows]
    mean = sum(data) / len(data)
    variance = sum([(d - mean) ** 2 for d in data]) / len(data)
    return variance


# building the tree
def buildtree(rows, scoref):
    """recursively builds decisionnode objects that form a decision tree

    At each node the best possible split is calculated (depending on the evaluation␣
↪metric).
    If no further split is neccessary the remaining items and their number of␣
↪occurence
    are written in the results property.

    Arguments:
        rows {list} -- dataset from which to build the tree
        scoref {function} -- evaluation metric (entropy / gini coefficient)

    Returns:
        decisionnode -- either two decisionnodes for true and false branch or one␣
↪decisionnode with results (leaf node)
    """
    if len(rows) == 0:
        return decisionnode()
    current_score = scoref(rows)

    # Set up variables to track the best criteria
    best_gain = 0.0
    best_criteria = None
    best_sets = None

    column_count = len(rows[0]) - 1  # number of columns minus last one (result)
    for col in range(0, column_count):
        # Generate the list of different values in this column
        column_values = {}
        for row in rows:
            column_values[row[col]] = 1
```

```
        # Try dividing the rows up for each value in this column
        for value in column_values.keys():
            (set1, set2) = divideset(rows, col, value)

            # Information Gain
            p = float(len(set1)) / len(rows)  # = ration(Anteil) of list 1 against␣
↪whole list (list1+list2)
            gain = current_score - p * scoref(set1) - (1 - p) * scoref(set2)  # set1␣
↪and set2 can be exchanged
            if gain > best_gain and len(set1) > 0 and len(set2) > 0:
                best_gain = gain
                best_criteria = (col, value)
                best_sets = (set1, set2)
    # print "Best Gain = " + str(best_gain)
    # print "Best criteria = " + str(best_criteria)

    # Create subbranches
    if best_gain > 0:
        trueBranch = buildtree(best_sets[0], scoref)
        falseBranch = buildtree(best_sets[1], scoref)
        return decisionnode(col=best_criteria[0], value=best_criteria[1],␣
↪tb=trueBranch, fb=falseBranch)
    else:
        return decisionnode(results=uniquecounts(rows))


def printtree(tree, indent=' '):
    """prints out the tree on the command line

    Arguments:
        tree {decisionnode} -- tree that gets printed

    """
    if tree.results is not None:
        print str(tree.results)
    else:
        print str(tree.col) + ': ' + str(tree.value) + '?'
        print indent + 'T-->',
        printtree(tree.tb, indent + '   ')
        print indent + 'F-->',
        printtree(tree.fb, indent + '   ')


def getwidth(tree):
    """returns the number of leaves = endnodes in the tree

    Arguments:
        tree {decisionnode} -- tree to examine

    Returns:
        number -- number of endnodes
    """
    if tree.tb is None and tree.fb is None:
        return 1
    return getwidth(tree.tb) + getwidth(tree.fb)


def getdepth(tree):
```

```
    """returns the maximum number of consecutive nodes

    Arguments:
        tree {decisionnode} -- tree to examine

    Returns:
        number -- maximum number of consecutive nodes
    """
    if tree.tb is None and tree.fb is None:
        return 0
    return max(getdepth(tree.tb), getdepth(tree.fb)) + 1


def drawtree(tree, jpeg='tree.jpg'):
    """visualization of the tree in a jpeg

    Arguments:
        tree {decisionnode} -- tree to draw

    Keyword Arguments:
        jpeg {str} -- Name of the .jpg (default: {'tree.jpg'})
    """
    w = getwidth(tree) * 100
    h = getdepth(tree) * 100 + 120

    img = Image.new('RGB', (w, h), (255, 255, 255))
    draw = ImageDraw.Draw(img)

    drawnode(draw, tree, w / 2, 20)
    img.save(jpeg, 'JPEG')


def drawnode(draw, tree, x, y):
    """Helper Function for drawtree, draws a single node

    Arguments:
        draw {img} -- node to be drawn
        tree {decisionnode} -- tree that the node belongs to
        x {number} -- x location
        y {number} -- y location
    """
    if tree.results is None:
        # Get the width of each branch
        w1 = getwidth(tree.fb) * 100
        w2 = getwidth(tree.tb) * 100

        # Determine the total space required by this node
        left = x - (w1 + w2) / 2
        right = x + (w1 + w2) / 2

        # Draw the condition string
        draw.text((x - 20, y - 10), str(tree.col) + ':' + str(tree.value), (0, 0, 0))

        # Draw links to the branches
        draw.line((x, y, left + w1 / 2, y + 100), fill=(255, 0, 0))
        draw. line((x, y, right - w2 / 2, y + 100), fill=(255, 0, 0))

        # Draw the branch nodes
```

```
        drawnode(draw, tree.fb, left + w1 / 2, y + 100)
        drawnode(draw, tree.tb, right - w2 / 2, y + 100)
    else:
        txt = ' \n'.join(['%s:%d' % v for v in tree.results.items()])
        draw.text((x - 20, y), txt, (0, 0, 0))


def prune(tree, mingain):
    """prunes the leaves of a tree in order to reduce complexity

    By looking at the information gain that is achieved by splitting data further and
↪further and checking if
    it is above the mingain threshold, neighbouring leaves can be collapsed to a
↪single leaf.

    Arguments:
        tree {decisionnode} -- tree that gets pruned
        mingain {number} -- threshold for pruning
    """
    if getdepth(tree) == 0:
        return
    # If the branches aren't leaves, then prune them
    if tree.tb.results is None:
        prune(tree.tb, mingain)
    if tree.fb.results is None:
        prune(tree.fb, mingain)

    # If both the subbranches are now leaves, see if they should be merged
    if tree.tb.results is not None and tree.fb.results is not None:
        # Build a combined dataset
        tb, fb = [], []
        # v equals key, c equals value, results in a list of the different values
↪each added up
        for v, c in tree.tb.results.items():
            tb += [[v]] * c
        for v, c in tree.fb.results.items():
            fb += [[v]] * c

        # Test the reduction in entropy
        delta = entropy(tb + fb) - (entropy(tb) + entropy(fb)) / 2
        # print delta
        if delta < mingain:
            # Merge the branches
            tree.tb, tree.fb = None, None
            tree.results = uniquecounts(tb + fb)
            # print "tree pruned"


def classify(observation, tree):
    """takes a new data set that gets classified and the tree that determines the
↪classification and returns the estimated result.

    Arguments:
        observation {numpy.array} -- the new data set that gets classified, e.g. test
↪data set
        tree {decisionnode} -- tree that observation gets classified in

    Returns:
```

```
        data -- expected result
    """
    if tree.results is not None:
        return tree.results
    else:
        v = observation[tree.col]
        if v is None:
            tr, fr = classify(observation, tree.tb), classify(observation, tree.fb)
            tcount = sum(tr.values())
            fcount = sum(fr.values())
            tw = float(tcount) / (tcount + fcount)
            fw = 1 - tw
            result = {}
            for k, v in tr.items():  # k is name, v is value
                result[k] = v * tw
            for k, v in fr.items():
                result[k] = result.setdefault(k, 0) + (v * fw)
            return result
        else:
            if isinstance(v, int) or isinstance(v, float):
                if v >= tree.value:
                    branch = tree.tb
                else:
                    branch = tree.fb
            else:
                if v == tree.value:
                    branch = tree.tb
                else:
                    branch = tree.fb
        return classify(observation, branch)


def path_gen(tree):
    """Create a path Matrix which contains the structure of the tree. Calls path_gen2␣
↪to do so.

    Arguments:
        tree {decisionnode} -- tree of which the data structure is stored

    Returns:
        numpy.array -- data structure of the tree, NaN means there is no more branch
    """
    z1 = 0  # equals number of leafs, increases during creation of path
    z2 = 0  # equals depth, fluctuates during creation of path
    width = getwidth(tree)
    depth = getdepth(tree) + 1  # +1 for target values
    path = np.zeros((width, depth))  # Prelocate Memory
    path[::] = None  # NaN in final result means branch is shorter than total depth
    path, z1 = path_gen2(tree, width, depth, path, z2, z1)
    return path


def path_gen2(tree, width, depth, path, z2, z1):
    """Create a path Matrix which contains the structure of the tree.

    creates a matrix 'path' that represents the structure of the tree and the␣
↪decisions made at each node, last column contains the average MSE at that leaf
    the sooner a feature gets chosen as a split feature the more important it is (the␣
↪farther on the left it appears in path matrix)
```

```
    order that leaves are written in (top to bottom): function will crawl to the␣
↪rightmost leaf first (positive side), then jump back up one level and move one step␣
↪to the left (loop)

    Arguments:
        tree {decisionnode} -- tree of which the data structure is stored
        width {int} -- width of the tree
        depth {int} -- depth of the tree
        path {[type]} -- current path matrix, gets updated during function calls
        z2 {int} -- control variable for current depth
        z1 {int} -- control variable for current width

    Returns:
        numpy.array -- the structure of the tree
    """
    while z1 < width:  # continue until total number of leaves is reached
        if tree.results is None:  # = if current node is not a leaf
            path[z1, z2] = tree.col  # write split feature of that node into path␣
↪matrix
            z2 += 1  # increase depth counter
            path, z1 = path_gen2(tree.tb, width, depth, path, z2, z1)  # recursively␣
↪call path_gen function in order to proceed to next deeper node in direction of tb
            for x in range(z2):
                path[z1, x] = path[z1 - 1, x]  # assign the former columns the same␣
↪value as the leaf above
            path, z1 = path_gen2(tree.fb, width, depth, path, z2, z1)  # recursively␣
↪call path_gen function in order to proceed to next deeper node in direction of fb
            z2 -= 1  # after reaching the deepest fb leaf move up one level in depth
            break
        else:  # = if current node is a leaf
            path[z1, -1] = np.mean(tree.results.keys())  # put the average MSE in the␣
↪last column of path
            z1 += 1  # current leaf is completely written into path, proceeding to␣
↪next leaf
            break
    return path, z1  # return the path matrix and current leaf number


def check_path(tree, result):
    """Check if a tree contains MSE_min (= True) or not (= False)

    Arguments:
        tree {decisionnode} -- tree that gets searched for result
        result {data} -- result that the tree is searched for

    Returns:
        bool -- True if result is in the tree, false if not
    """
    path = path_gen(tree)
    if result in path[:, -1]:
        return True
    else:
        return False


def buildforest(data, n_trees, scoref, n_feat, min_data, pruning):
    """Growing the Random Forest
```

```
    The Random Forest consists of n_trees. Each tree sees only a subset of the data␣
↪and a subset of the features.
    Important: a tree never sees the original data set, only the performance of the␣
↪classifying algorithm
    For significant conclusions enough trees must be generated in order to gain the␣
↪statistical benefits that overcome bad outputs

    Arguments:
        * data {numpy.array} -- data set the Forest is built upon
        * n_trees {int} -- number of trees in a Decision tree
        * scoref {function} -- scoring metric for finding new nodes
        * n_feat {int} -- number of features in data
        * min_data {float} -- minimum percentage of all data sets that a tree will see
        * pruning {bool} -- pruning enabled (>0) / disabled(=0)

    Returns:
        * RF --  dictionary = importances of single features in the forest
        * prob_current -- single value for importance, used for generating new biased␣
↪feature sets
        * trees -- contains all single trees that stand in the Forest
    """

    # Initializations
    prob_current = None  # Prelocate
    RF = {}  # Prelocate dictionary for prioritizing important features
    trees = []  # Prelocate list that will contain the trees that stand in the␣
↪currently built forest
    total_best_result = None  # Prelocate
    current_best_result = None  # Prelocate
    path_min_current = []  # Prelocate
    wrongs = 0  # initialize number of (useless) trees that have only one node

    # build single trees
    for x in range(n_trees):  # n_trees is the number of trees in the forest

        # select only subset of available datasets
        # create mask for randomly choosing subset of available datasets
        mask_sub_data = np.zeros(data.shape[0], dtype=bool)  # Prelocate
        # randomly choose the random datasets
        rand_data = np.random.choice(range(data.shape[0]), size=int(np.amax((np.
↪around(len(data) * min_data, decimals=0),
                                                                             np.
↪random.choice(range(len(data) - 1)) + 1), axis=None)), replace=False, p=None)
        # translate to selected data sets
        mask_sub_data[rand_data] = True
        sub_data = data[mask_sub_data, :]  # random subset of datasets still␣
↪including all features

        # select only subset of features
        # create mask for randomly choosing subset of available features
        mask_sub_features = np.zeros(data.shape[1], dtype=bool)  # Prelocate
        # randomly choose the random features
        rand_feat = np.random.choice(range(data.shape[1] - 1), size=np.random.
↪choice(range(len(data[0]) - 1)) + 1, replace=False, p=None)
        # sort ascending
        rand_feat = np.sort(rand_feat)
        # append last column with result
        rand_feat = np.append(rand_feat, data.shape[1] - 1)
```

```
        # translate to selected features
        mask_sub_features[rand_feat] = True

        sub_data = sub_data[:, mask_sub_features]  # random subset of datasets and␣
↪random subset of features
        # print "sub_data = " + str(sub_data) # Debugging line

        # build the tree from the subset data, last column contains result
        # print "building tree" # Debugging Line
        tree = buildtree(sub_data, scoref)

        # pruning the tree (if hyperparameter is enabled)
        if pruning > 0:
            prune(tree, pruning)

        # draw the tree and create path matrix
        # drawtree(tree, jpeg='treeview_RF.jpg') # Debbuging Line

        if getdepth(tree) is 0:  # if the tree contains only a single node --> tree␣
↪is useless
            wrongs += 1
            # print "wrongs: " + str(wrongs) # Debugging Line
        else:  # only increment feature counter if tree has more than one leaf
            path = path_gen(tree)  # create path to current tree
            current_best_result = np.max(path[:, -1])
            path_min_current = path[np.argmax(path[:, -1])]

            # update best result and corresponding path
            if total_best_result is None or current_best_result > total_best_result:
                total_best_result = current_best_result

            # update the RF dictionary that rewards / punishes features
            update_RF(RF, path_min_current, tree, rand_feat)
            trees.append(tree)
    # print "RF: " + str(RF) # Debugging Line
    # print "Returning RF" # Debugging Line

    # a "wrong" tree is a tree with only one node that has no power to gain␣
↪additional insight and therefore is useless...
    print "wrongs: " + str(wrongs) + "/" + str(n_trees)

    # Transform the counter for rewarded / punished features from RF dictionary into␣
↪a proportionate number
    # set up scaler that projects accumulated values of RF onto a scale between 0 and␣
↪1
    min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
    # take only values of RF, reshape them (otherwise deprecation warning), make them␣
↪numpy array, and scale them (again) between 0 and 1
    temp = min_max_scaler.fit_transform(np.nan_to_num(np.array(RF.values())).reshape(-
↪1, 1))
    # sum up values of RF, divide each value of RF by sum to get percentage --> must␣
↪sum up to 1
    temp_sum = np.sum(temp)
    temp_percent = temp * (1.0 / temp_sum)
    # print temp_percent
    # update values in RF with scaled percentage values
    i = 0
    for key in RF:
```

```
        RF[key] = temp_percent[i][0]  # [0] because otherwise there would be an array␣
↪inside the dictionary RF
        i += 1
    # print "RF: " + str(RF) # Debugging Line

    # build a dictionary of most important features in a tree and how often they were␣
↪chosen
    # create weights of features
    weights = {}  # Prelocate
    weights_sorted = {}  # Prelocate
    # transfer values from dictionary into list
    for key, value in RF.items():
        weights[key] = float(value)  # create relative weight
    # some features might not get picked once, so their probability is set to zero
    if len(weights) < n_feat:
        for key in range(n_feat):
            if key not in weights:
                weights[key] = 0
    # print "weights = " + str(weights) # Debugging Line
    weights_sorted = dict(sorted(weights.items(), key=lambda value: value[0],␣
↪reverse=False))  # sort by frequency = importance

    # print "importance of features in random forest: " + str(weights_sorted) #␣
↪Debugging Line

    prob_current = np.array(weights_sorted.values())  # extract only the values of␣
↪feature importance

    return RF, prob_current, trees


def update_RF(RF, path, tree, rand_feat):
    """for each tree the features that lead to the leaf with the lowest Error will␣
↪get rewarded.
    Features that don't lead to the leaf with the lowest Error will get punished␣
↪(only by 20% of
    the amount the "good" featurtes get rewarded).


    RF is a dictionary that gets updated after a new tree is built and thus contains␣
↪the cummulation of all
    feature appearences in the whole forest.

    Arguments:
        * RF {dict} -- dictionary that counts occurrence / absence of different␣
↪features
        * path {numpy.array} -- structure of the current tree
        * tree {decisionnode} -- tree that gets examined
        * rand_feat {list} -- boolean mask of selected features (1 = selected, 0 =␣
↪not selected)

    Returns:
        * RF -- updated dictionary that counts occurrence / absence of different␣
↪features
    """
    current_depth = getdepth(tree)
    # print "current path: " + str(path) # Debugging Line
    # print  "current depth = " + str(getdepth(tree)) # Debugging Line
```

```
    # print "current col: " + str(tree.col) # Debugging Line
    if current_depth == 0:
        return RF
    MSE_min = path[-1]
    # print "MSE_min: " + str(MSE_min) # Debugging Line
    # print "Checking if MSE_min is in True branch" # Debugging Line
    if check_path(tree.tb, MSE_min) is True:
        # print "MSE_min is in True Branch" # Debugging Line

        # initialize the feature in dictionary RF if it appears for the first time
        if rand_feat[int(tree.col)] not in RF:
            RF[rand_feat[int(tree.col)]] = float(current_depth)
        else:  # if the feature is already present in dictionary RF, increase counter
            RF[rand_feat[int(tree.col)]] += float(current_depth)
        # print "added " + str(current_depth) + " to feature  " + str(tree.col) #␣
↪Debugging Line
        # print "current RF: " + str(RF) # Debugging Line
        update_RF(RF, path[1:], tree.tb, rand_feat)  # recursively jump into update_
↪RF again with shortened path at next level in true branch
    else:
        # print "MSE_min is not in True Branch" # Debugging Line
        # print "Checking if MSE_min is in False Branch" # Debugging Line
        if check_path(tree.fb, MSE_min) is True:
            # print "MSE_min is in False Branch" # Debugging Line
            if rand_feat[int(tree.col)] not in RF:  # initialize the feature in␣
↪dictionary RF if it appears for the first time
                RF[rand_feat[int(tree.col)]] = -0.2 * float(current_depth)
            else:  # if the feature is already present in dictionary RF, decrease␣
↪counter
                RF[rand_feat[int(tree.col)]] -= float(current_depth) * 0.2
            # print "subtracted " + str(current_depth*0.2) + " from feature " +␣
↪str(tree.col) # Debugging Line
            # print "current RF: " + str(RF) # Debugging Line
            update_RF(RF, path[1:], tree.fb, rand_feat)  # recursively jump into␣
↪update_RF with shortened path at next level in false branch


def forest_predict(data, trees, prob, n_configs, biased):
    """Predict performance of new feature sets

    Predicts biased and unbiased feature sets in the before constructed Random Forest.
    Feature sets are predicted in every single Decision Tree in the Random Forest.
    Results are represented as (mean+0.1*var) and (variance+0.1*mean) for each␣
↪feature set.
    The two best feature sets are selected to be sent into the :ref:`MLA <MLA>`.


    Arguments:
        * data {numpy.array} -- contains all previous computing runs
        * trees {decisionnodes} -- the trees that make up the Random Forest
        * prob {array of floats} -- probability that a feature gets chosen into a␣
↪feature set
        * n_configs {int} -- number of feature sets to be generated
        * biased {bool} -- true for biased feature selection, false for unbiased␣
↪feature selection

    Returns:
        * best mean -- highest average of all predicted feature sets
```

```
        * best feature set mean -- corresponding boolean list of features (0=feature␣
↪not chosen, 1=feature chosen)
        * best var -- highest variance of all predicted feature sets
        * best feature set var -- corresponding boolean list of features (0=feature␣
↪not chosen, 1=feature chosen)
    """
    if biased is not True:
        prob = None
    # print "prob: " + str(prob) # Debugging Line
    # Prelocate variables
    mean = np.zeros(n_configs)
    var = np.zeros(n_configs)
    best_mean = np.array([0])
    best_var = np.array([0])
    best_featureset_mean = np.array([0])
    best_featureset_var = np.array([0])

    # new config (=feature set) is generated
    for x in range(n_configs):  # n_configs_biased is hyperparameter
        # create mask for choosing subfeatures
        mask_sub_features = np.zeros(data.shape[1] - 1, dtype=bool)  # Prelocate␣
↪Memory
        if prob is not None:
            rand_feat = np.random.choice(range(data.shape[1] - 1), size=int(np.
↪min((np.random.choice(range(len(data[0]) - 1)) + 1, len(np.nonzero(prob)[0])))),
                                         replace=False, p=prob)  # size must be <=␣
↪nonzero values of p, otherwise one feature gets selected twice
        if prob is None:
            rand_feat = np.random.choice(range(data.shape[1] - 1), size=int(np.random.
↪choice(range(len(data[0]) - 1)) + 1), replace=False, p=None)  # size must be <=␣
↪nonzero values of p, otherwise one feature gets selected twice

        rand_feat = np.sort(rand_feat)  # sort ascending
        mask_sub_features[rand_feat] = True
        # print "current feature set: " + str(mask_sub_features) # Debugging Line

        # Predict the new feature set
        predictions = np.zeros(len(trees))  # Prelocate Memory
        i = 0  # set counter for going through all trees
        # classify the randomly chosen feature sets in each tree
        for tree in trees:
            predictions[i] = classify(mask_sub_features, tree).keys()[0]
            i += 1
        # print "predictions: " + str(predictions) # Debugging Line
        # print "best_mean = " + str(best_mean) # Debugging Line
        # calculate mean an std for all predictions in a tree
        mean[x] = np.mean(predictions)
        var[x] = np.var(predictions) / abs(mean[x])
        # check if current mean and var are better than best mean and var
        # calculation: best_mean = 1.0*mean + 0.1*var and vice versa
        if best_mean == [0] or mean[x] + var[x] * 0.75 > best_mean:
            best_mean = mean[x] + var[x] * 0.75

            # print "best_mean updated: " + str(best_mean) # Debugging Line
            best_featureset_mean = mask_sub_features
            # print "best_featureset_mean = " + str(best_featureset_mean) # Debugging␣
↪Line
        if best_var == [0] or var[x] + mean[x] * 0.1 > best_var:
```

```
            best_var = var[x] + mean[x] * 0.1
            # print "best_var updated: " + str(best_var) # Debugging Line
            best_featureset_var = mask_sub_features
            # print "best_featureset_var = " + str(best_featureset_var) # Debugging␣
↪Line
    # print "best mean for current forest: " + str(best_mean) # Debugging Line
    # print "best feature set for best mean: " + str(best_featureset_mean) #␣
↪Debugging Line
    # print "best var for current forest: " + str(best_var) # Debugging Line
    # print "best feature set for best var" + str(best_featureset_var) # Debugging␣
↪Line
    return best_mean, best_var, best_featureset_mean, best_featureset_var


# based on the probabilities of each feature in past Forests, a new current_prob is␣
↪calculated that takes into
# account the mean and the gradient of the prior feature importances
def update_prob(Probability, i, weight_mean, weight_gradient, multiplier, seen_
↪forests):
    """Calculates the current Importance / Probability of the single features

    Based on the probabilities of each feature in past Forests a new current_prob is␣
↪calculated that takes into
    account the mean and the gradient of the prior feature importances.


    Arguments:
        * Probability {numpy array} -- contains Importances of single features for␣
↪all past Random Forests
        * i {integer} -- number of current Forest
        * weight_mean {float} -- weight of the mean in calculating resulting␣
↪probability
        * weight_gradient {float} -- weight of the var in calculating resulting␣
↪probability
        * multiplier {float} -- exponent for amplifying probabilities
        * seen_forests {integer} -- number of before built forest that are considered

    Returns:
        prob_current -- list of floats representing the calculated aggregation of␣
↪recent feature importances
    """
    # print "Probability: " + str(Probability[0:i + 1]) # Debugging Line

    # if only one or two calculations of prob has been done so far, leave prob empty
    # (np.gradient needs 3 points and 3 random Forests to provide better statistical␣
↪insurance than only 1 Random Forest)
    if i <= 1:
        prob_current = None
    else:
        # gradients contains the current gradient for each feature
        # map: function list is applied to all zip(transposed(a)) (without list: zip␣
↪generatets tuple instead of list)
        if i < seen_forests:
            gradients = np.gradient(map(list, zip(*Probability[0:i + 1])), axis=1)
            mean = np.mean(map(list, zip(*Probability[0:i + 1])), axis=1)
        # only the last seen_forests values will be taken into account
        else:
            # print "consider only last " + str(seen_forests) + " forests for␣
↪calculation of probability"
```

```
            gradients = np.gradient(map(list, zip(*Probability[i - seen_forests:i +
↪1])), axis=1)
            mean = np.mean(map(list, zip(*Probability[i - seen_forests:i + 1])),
↪axis=1)

        # print "gradients: " + str(gradients) # Debugging Line

        # calculate the mean of the gradient for each feature
        gradients_mean = map(np.mean, gradients)
        # print "gradients_mean: " + str(gradients_mean) # Debugging Line

        # calculate the norm of the gradient for each feature
        gradients_norm = map(np.linalg.norm, gradients)
        # print "gradients_norm: " + str(gradients_norm) # Debugging Line

        # divide the mean by the norm(=length)
        # (to punish strongly fluctuating values and to reward values that change
↪only slightly over time)
        gradients = np.nan_to_num(np.divide(gradients_mean, gradients_norm))  # nan_
↪to_num: because division by zero leaves NaN
        # print "gradients mean / norm: " + str(gradients) # Debugging Line

        # scale values
        min_max_scaler = preprocessing.MinMaxScaler(feature_range=(1, 100))
        gradients = min_max_scaler.fit_transform(gradients.reshape(-1, 1))  #
↪reshape: otherwise deprecation warning
        mean = min_max_scaler.fit_transform(mean.reshape(-1, 1))  # reshape:
↪otherwise deprecation warning
        # print "gradients rescaled: " + str(gradients) # Debugging Line
        # print "mean rescaled: " + str(mean) # Debugging Line

        # calculate new probability for selection of new feature sets
        # weight_mean, weight_gradient and multiplier are hyperparameters
        prob_current = (mean * weight_mean + gradients * weight_gradient)**multiplier
        # print "prob_current: " + str(prob_current) # Debugging Line
        # print "gradients + mean: " + str(gradients) # Debugging Line

        # express values as percentage (because sum(prob) must equal 1)
        prob_current = np.divide(prob_current, np.sum(prob_current))
        # print "gradients percent: " + str(gradients)
        prob_current = np.array([item for sublist in prob_current for item in
↪sublist])  # convert nested list into usual list
        # print "prob_current: " + str(prob_current) # Debugging Line

        # in the last run print out the gradients
        if i + 1 == len(Probability):
            print " "
            # print "gradients mean: " + str(gradients_mean) # Debugging Line
            # print " " # Debugging Line
            # print "prob_current: " + str(prob_current) # Debugging Line
    return prob_current


def update_database(X, y, data, mask_best_featureset, X_test, y_test):
    """Appends newly tested feature sets and their result to the already calculated
↪feature sets

    Arguments:
```

```
        * X {numpy array} -- X rat data sets
        * y {numpy array} -- y raw data sets
        * data {[type]} -- data set the Forest is built upon
        * mask_best_featureset {bool} -- feature set (1: feature contained, 0:␣
↪feature not contained)
        * X_test {numpy array} -- test data set
        * y_test {numpy array} -- test data set

    Returns:
        data -- updated data base
    """
    # create the best mean feature set
    X_sub = X[:, mask_best_featureset]
    # compute the corresponding y values
    y_new = compute(X_sub, y, mask_best_featureset, X_test, y_test)
    # combine feature set and new y (result)
    new_dataset = np.append(mask_best_featureset, y_new)
    # print "new_dataset_mean: " + str(new_dataset_mean) # Debugging Line
    # print new_dataset_mean.shape # Debugging Line

    # append new feature sets and according result to dataset
    data = np.append(data, [new_dataset], axis=0)
    return data


# This is the main part of the program which uses the above made definitions
def main_loop(n_start, pruning, min_data, n_forests, n_trees, n_configs_biased, n_
↪configs_unbiased, multiplier_stepup, seen_forests,
              weight_mean, weight_gradient, scoref, demo_mode, plot_enable):
    """Load raw data and Generate database for Random Forest. Iteratively build and␣
↪burn down new Random Forests, predict the performance of new feature sets and␣
↪compute two new feature sets per round.

    Arguments:

        * n_start {int} -- number of runs before building first RF = number of data␣
↪points in first RF; minimum = 4, default = 50
        * pruning {float} -- if greater than zero, branches of a Decision Tree will␣
↪be pruned proportional to pruning value; default = 0
        * min_data {float} -- minimum percentage of Datasets that is used in RF␣
↪generation; default = 0.2
        * n_forests {int} -- number of forests; minimum=1;  default = 25
        * n_trees {int} -- # number of trees that stand in a forest; min = 3; default␣
↪= number of features x 3 x
        * n_configs_biased {int} -- # number of deliberately chosen feature sets that␣
↪get predicted in each forest; default = n_trees x 5
        * n_configs_unbiased {int} -- # number of randomly chosen feature sets that␣
↪get predicted in each forest; default = n_configs_biased x0.2
        * multiplier_stepup {float} -- # sets how aggressively the feature importance␣
↪changes; default = 0.25
        * seen_forests {int} -- # number of recent forests that are taken into acount␣
↪for generating probability of the chosen feature sets default = 4
        * weight_mean {float} -- # weight of the mean in calculating the new␣
↪probability for selecting future feature sets; default = 0.2
        * weight_gradient {bool} -- # weight of the gradient in calculating the new␣
↪probability for selecting future feature sets; default = 0.8
        * scoref {function} -- # which scoring metric should be used in the Decision␣
↪Tree (available: entropy and giniimpurity); default = entropy
```

```
        * demo_mode bool -- # if true a comparison between the Random Forest driven␣
↪Search and a random search is done
        * plot_enable bool -- # decide if at the end a plot should be generated ,␣
↪only possible in demo mode

    """
    print "Starting script"
    # Generate Test Data
    print "Loading Raw Data"
    X_test, X, y_test, y, n_feat = import_data()
    # set default hyperparameters
    print "Setting Hyperparameters"
    if n_trees is 'default':
        n_trees = n_feat * 3
    if seen_forests is 'default':
        seen_forests = 4
    if n_configs_biased is 'default':
        n_configs_biased = n_trees * 5  # number of biased configs that get predicted␣
↪in each forest
    if n_configs_unbiased is 'default':
        n_configs_unbiased = int(round(n_configs_biased * 0.2))  # number of unbiased␣
↪configs that get predicted in each forest
    if multiplier_stepup is 'default':
        multiplier_stepup = 0.01
    if weight_mean is 'default':
        weight_mean = 0.1
    if weight_gradient is 'default':
        weight_gradient = 0.9
    if scoref is 'default':
        scoref = entropy
    elif scoref is 'entropy':
        scoref = entropy
    elif scoref is 'giniimpurity':
        scoref = giniimpurity
    elif scoref is 'variance':
        scoref = variance
    if pruning > 0:
        print "Pruning enabled"

    multiplier = 1  # initialize value for multiplier

    Probability = np.zeros(shape=[n_forests, n_feat])  # Prelocate Memory:␣
↪probabilites for selecting features in svm

    # Generate database for RF
    print "Generate Data Base for Random Forest"
    data = gen_database(n_start, X, y, X_test, y_test)

    if demo_mode:
        data_start = data  # save starting data for later comparison with random␣
↪feature set selection
    # print "len(data): " + str(len(data)) # Debugging Line

    # ### Start of ForestFire ###
    print "Starting ForestFire"

    # Creating Random Forests: build n_trees, each sees only subs#et of data points␣
↪and subset of features of data
```

```
    for i in range(n_forests):

        # create the forest
        print " "
        print "Building Random Forest Nr. " + str(i + 1)
        RF, Probability[i], trees = buildforest(data, n_trees, scoref, n_feat, min_
↪data, pruning)
        # print "RF: " + str(RF) # Debugging Line

        # Update probability
        prob_current = update_prob(Probability, i, weight_mean, weight_gradient,␣
↪multiplier, seen_forests)
        print "max Probability: " + str(np.max(prob_current))
        # print np.multiply(np.divide(1.0, n_feat), 2)
        if i > 1 and np.max(prob_current) < np.multiply(np.divide(1.0, n_feat), 2):
            multiplier += multiplier_stepup
            print "raised multiplier to " + str(multiplier)
        # print RF # Debugging Line
        # print " " # Debugging Line
        # print "Predicting new possible configs" # Debugging Line
        # print "biased configs" # Debugging Line

        # test new biased and unbiased feature sets and extract the best feature sets
        best_mean_biased, best_var_biased, best_featureset_mean_biased, best_
↪featureset_var_biased = forest_predict(
            data, trees, prob_current, n_configs_biased, biased=True)
        # print " " # Debugging Line
        # print "unbiased configs" # Debugging Line
        best_mean_unbiased, best_var_unbiased, best_featureset_mean_unbiased, best_
↪featureset_var_unbiased = forest_predict(
            data, trees, prob_current, n_configs_unbiased, biased=False)
        # print "best mean_biased: " + str(best_mean_biased) # Debugging Line
        # print "best mean_unbiased: " + str(best_mean_unbiased) # Debugging Line
        # print " " # Debugging Line
        best_mean = np.max((best_mean_biased, best_mean_unbiased))
        if best_mean == best_mean_biased:
            best_featureset_mean = best_featureset_mean_biased
            print "picked biased feature set for mean"
        elif best_mean == best_mean_unbiased:
            best_featureset_mean = best_featureset_mean_unbiased
            print "picked unbiased feature set for mean"
        # print best_mean # Debugging Line
        # print best_featureset_mean # Debugging Line
        # print "best_var_biased: " + str(best_var_biased) # Debugging Line
        # print "best_var_unbiased: " + str(best_var_unbiased) # Debugging Line
        best_var = np.max((best_var_biased, best_var_unbiased))
        if best_var == best_var_biased:
            best_featureset_var = best_featureset_var_biased
            print "picked biased feature set for var"
        elif best_var == best_var_unbiased:
            best_featureset_var = best_featureset_var_unbiased
            print "picked unbiased feature set for var"

        # update database with two new feature sets
        # print "current feature sets:" + str(data[:, :-1]) # Debugging Line
        # print "best_var feature set:" + str(best_featureset_var) # Debugging Line
        # print "best_mean feature set:" + str(best_featureset_mean) # Debugging Line
```

```
        # check if newly selected feature sets  are already in data. if so, there is␣
↪no need to compute again
        check_mean = any(check for check in (np.array_equal(data[entry, :-1], best_
↪featureset_mean) for entry in range(len(data))))
        check_var = any(check for check in (np.array_equal(data[entry, :-1], best_
↪featureset_var) for entry in range(len(data))))

        # print "data len: " + str(len(data)) # Debugging Line

        double_var = np.all(np.all(data[x, :-1] == best_featureset_var for x in␣
↪range(len(data[:, -1]))))
        double_mean = np.all(np.all(data[x, :-1] == best_featureset_mean for x in␣
↪range(len(data[:, -1]))))

        if check_var:
            z = 0
            stopper = False
            for x in double_var:
                if x.all() and not stopper:
                    # print "Stopper: " + str(stopper) # Debugging Line
                    print "Variance feature set already computed. No need to do it␣
↪again"
                    data = np.append(data, [data[z]], axis=0)
                    stopper = True
                z += 1
        else:
            data = update_database(X, y, data, best_featureset_var, X_test, y_test)

        if check_mean:
            z = 0
            stopper = False
            for x in double_mean:
                if x.all() and not stopper:
                    # print "Stopper: " + str(stopper) # Debugging Line
                    print "Mean feature set already computed. No need to do it again!"
                    data = np.append(data, [data[z]], axis=0)
                    stopper = True
                z += 1
        else:
            data = update_database(X, y, data, best_featureset_mean, X_test, y_test)

        # check for current best feature sets
        best_featuresets_sorted = data[np.argsort(-data[:, -1])]
        if i == 0:
            best_featuresets_sorted_old = best_featuresets_sorted  # initialize␣
↪storage value
        # if the best 5 feature sets have improved, update the current best feature␣
↪sets
        if sum(best_featuresets_sorted[:5, -1]) > sum(best_featuresets_sorted_old[:5,␣
↪-1]) or i == 0:
            print "found new best 5 feature sets: " + str(best_featuresets_sorted[:5])
        # store values for comparison to later results
        best_featuresets_sorted_old = best_featuresets_sorted

    # ### End of ForestFire ###

    # store results
    print "Storing results"
```

```
    np.savetxt('results.txt', best_featuresets_sorted)
    np.save('results', best_featuresets_sorted)

    print " "
    print "ForestFire finished"
    print " "

    if demo_mode:
        # Generate additional data set to compare performance of RF to random␣
↪selection of feature sets
        print "Generating more randomly selected feature sets for comparison"
        data_compare = np.append(data_start, gen_database(2 * n_forests, X, y, X_test,
↪ y_test), axis=0)
        # print "len(data_compare): " + str(len(data_compare))

        # sort according to lowest MSE
        best_featuresets_sorted_compare = data_compare[np.argsort(-data_compare[:, -
↪1])]

        # print out some of the results
        print "best 5 feature sets of random selection: " + str(best_featuresets_
↪sorted_compare[:5])
        print " "
        print "Best result after " + str(n_start + 2 * n_forests) + " random SVM␣
↪runs: " + str(best_featuresets_sorted_compare[0, -1])
        print "Best result of ForestFire after " + str(n_start) + " initial random␣
↪runs and " + str(2 * n_forests) + " guided runs: " + str(best_featuresets_sorted[0,␣
↪-1])
        if best_featuresets_sorted[0, -1] > best_featuresets_sorted_compare[0, -1]:
            print "Performance with ForestFire improved by " + str(-100 * (1 - np.
↪divide(best_featuresets_sorted[0, -1], best_featuresets_sorted_compare[0, -1]))) + "
↪%"
        if best_featuresets_sorted[0, -1] == best_featuresets_sorted_compare[0, -1]:
            print "Performance could not be improved (same MSE as in random selection)
↪"
        if best_featuresets_sorted[0, -1] < best_featuresets_sorted_compare[0, -1]:
            print "Performance deteriorated, ForestFire is not suitable :("
        print "Execution finished"

        # Compare Random Search VS Random Forest Search
        print " "
        print "Found best value for ForestFire Search after " + str(n_start) + "␣
↪initial runs and " + str(np.argmax(data[:, -1] + 1) - n_start) + "/" +␣
↪str(len(data) - n_start) + " smart runs"
        print "Best value with ForestFire: " + str(np.max(data[:, -1]))
        print " "
        print "Found best value for Random Search after " + str(np.argmax(data_
↪compare[:, -1])) + " random runs"
        print "Best value with Random Search: " + str(np.max(data_compare[:, -1]))

        print " "
        print "Creating Plots"

        # plots
        if plot_enable and demo_mode:
            # first plot
            plt.figure(1, figsize=(25, 12))
            plt.plot(np.array(range(len(data[:, -1]))), data[:, -1], label='ForestFire
↪')
```

```
            plt.plot(np.array(range(len(data_compare[:, -1])))), data_compare[:, -1],␣
↪label='Random Search')

            plt.xlabel('No. tested Feature Sets')
            plt.ylabel('Score')
            plt.title('Results current best score')
            plt.legend(loc=2)
            plt.annotate('Highest Score ForestFire', xycoords='data',
                         xy=(np.argmax(data[:, -1]), np.max(data[:, -1])),
                         xytext=(np.argmax(data[:, -1]) * 1.05, np.max(data[:, -1]) *␣
↪1.01),
                         arrowprops=dict(facecolor='black', shrink=1),
                         )
            plt.annotate('Highest Score Random Search', xycoords='data',
                         xy=(np.argmax(data_compare[:, -1]), np.max(data_compare[:, -
↪1])),
                         xytext=(np.argmax(data_compare[:, -1]) * 1.05, np.max(data_
↪compare[:, -1]) * 0.95),
                         arrowprops=dict(facecolor='black', shrink=1),
                         )

            # second plot
            data_high = data
            for x in range(len(data_high) - 1):
                if data_high[x, -1] > data_high[x + 1, -1]:
                    data_high[x + 1, -1] = data_high[x, -1]

            data_compare_high = data_compare
            for x in range(len(data_compare_high) - 1):
                if data_compare_high[x, -1] > data_compare_high[x + 1, -1]:
                    data_compare_high[x + 1, -1] = data_compare_high[x, -1]

            plt.figure(2, figsize=(25, 12))
            plt.plot(np.array(range(len(data[:, -1])))), data_high[:, -1], label=
↪'ForestFire')
            plt.plot(np.array(range(len(data_compare[:, -1])))), data_compare[:, -1],␣
↪label='Random Search')

            plt.xlabel('No. tested Feature Sets')
            plt.ylabel('Score')
            plt.title('Results all time best score')
            plt.legend(loc=2)

            plt.show()
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Collective_Intelligence] Collective Intelligence, O'Reilly, ISBN: 978-0-596-52932-1

PYTHON MODULE INDEX

f

INDEX