ForestFire Documentation

Release 1.0.0

Marlon Weinert

Nov 10, 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 the most promising feature sets are selected and computed. 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.

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 a nearly indifinite number of possible selections (= feature sets) emerges. 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 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 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 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, the way this data should be split (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 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 (mean and variance) are selected, computed by the original *MLA* and 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 the list of the top 5 feature sets changes.

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 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

Todo

write a nice Overview do it in the end when all references are complete

2.1.1 Abbreviations

DT *Decision Tree*

RF *Random Forest*

MLA *Machine Learning Algorithm*

2.1.2 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

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

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

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

2.1.3 References

2.1.4 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
from PIL import Image, ImageDraw
```

2.1.5 About the author

Information about author.

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 = 10  # default = 30
# 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 = 0.4
# minimum percentage of Datasets that is used in RF generation; default = 0.2
min_data = 0.2
# 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 = 25
```

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 = 4 ? make variable?
seen_forests = 'default'
# the chosen feature sets default = 4 ? make variable?

# 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 = 'variance'
# set random seed for repeatabilit; comment out if no repeatability is required;␣
↪default = 1
```

```
np.random.seed(1)
```

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*.

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]
```

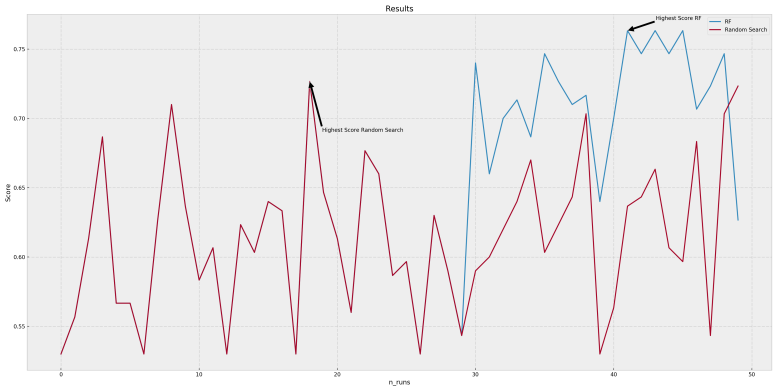
Interpretation:

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 a plot is produced:

Todo

no green highlighting in source code



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).

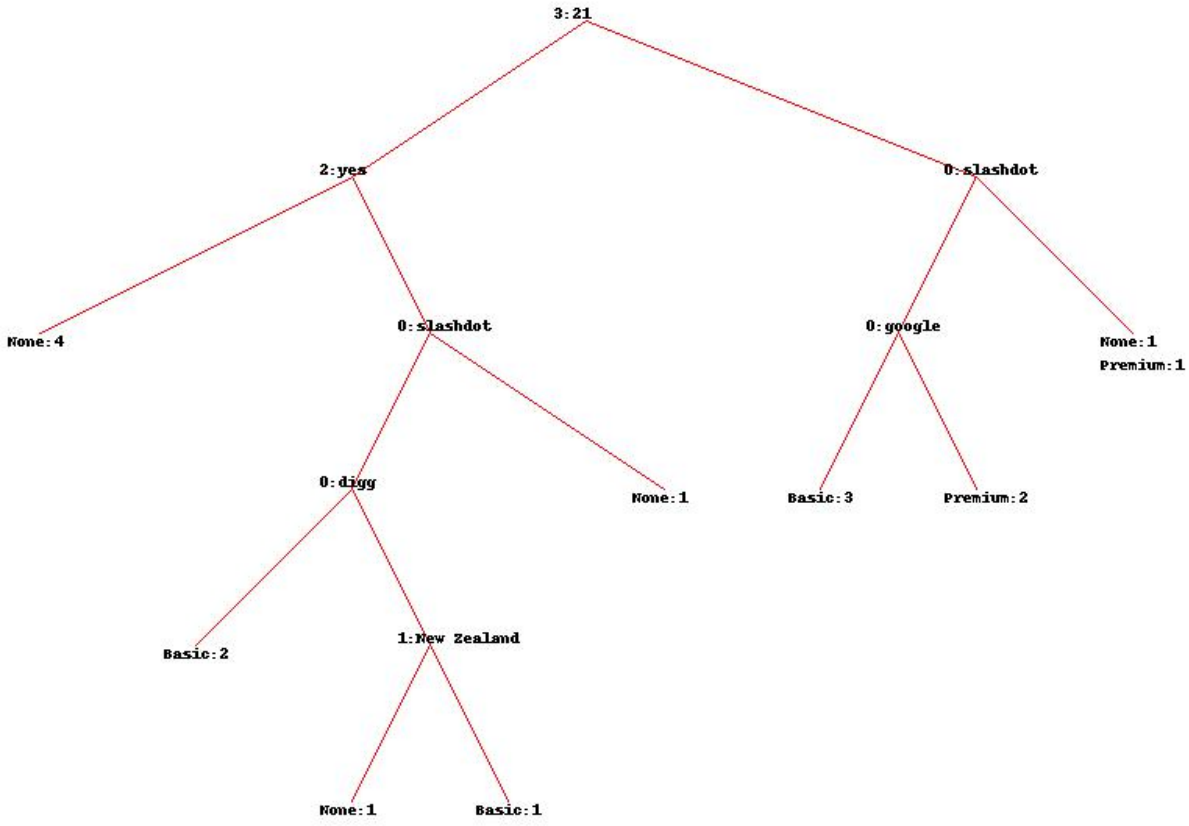


Fig. 3.1: Treeview.jpg

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

3.2 Random Forest

corresponding file: Main.py

3.2.1 Why is a single Decision Tree 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 subset 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.

3.2.3 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
- entirely randomly chosen feature sets

The two *hyperparameters* *n_configs_biased* and *n_configs_unbiased* determine the amount of feature sets that get tested. Since predicting takes not much computing capacity this number can safely be set fairly high.

Of all predicted feature sets two are chosen for the next computing runs with the *MLA*. One with a high average (mean) and one with a high variance.

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 – importances of single features in the forest
- Prob_current – importance of the features in the forest
- trees – the structure of the single trees the forest consists of

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 reward)

RF 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.**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.

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)

3.3 Update Database

Important: Functions used in this chapter

CHAPTER

FOUR

EVALUATION MODE

4.1 Evaluation

Important: Functions used in this chapter

4.2 Plot

Important: Functions used in this chapter

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

Remaining Todos:

Todo

write a nice Overview do it in the end when all references are complete

(The original entry is located in /Users/Dandelo/CloudStation_Marlon/Energietechnik/Studienarbeit/Sphinx/ForestFire/source/Overview line 6.)

Todo

no green highlighting in source code

(The original entry is located in /Users/Dandelo/CloudStation_Marlon/Energietechnik/Studienarbeit/Sphinx/ForestFire/source/execution line 195.)

Todo

so so much more...

(The original entry is located in /Users/Dandelo/CloudStation_Marlon/Energietechnik/Studienarbeit/Sphinx/ForestFire/source/index.rst, line 70.)

CHAPTER

SIX

KINDERSPIELPLATZ

es folgt Quatsch mit Sose

Eine Referenz zu getting started: *Using ForestFire*

Note: Eine Notiz! Ha!

Warning: ohhh eine Warnung!

Todo

so so much more...

- eintrag 1
- eintrag 2
- eintrag 3
- eintrag 4

Wir bauen einen *DT*

Listing 6.1: oha

```
a = 5
```

```
import matplotlib.pyplot as plt

from sklearn import preprocessing
from PIL import Image, ImageDraw
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')

# 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 X_sub
```

```
10
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
```

BIBLIOGRAPHY

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

PYTHON MODULE INDEX

f

INDEX