In [1]:

```
%load_ext autoreload
%autoreload 2
```

In [2]:

```
import copy
import git
import json
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn
import subprocess
import sys

from git import Repo
from unidiff import PatchSet

from sklearn import tree
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

from src import loading
from src import predictor
from src import preprocessing
from src import visualization
from src.reorderer import *
from src.reordering_analyzer import ReorderingAnalyzer
```

Structure

This notebook is separated into three main sections

1. Visualization

Making the mutation testing data gathered visible and exploring the test suites visually.

2. Prediction

Preparing the data for predicting failing tests and exploring features and different models.

3. Reordering

Using the prepared data for a dive into Test Case Reordering, using both binary classifiers from before, as other methods and comparing there results.

Look into the Data

Our starting point is mutation testing data generated by Mutester, a tool that was developed for exactly this task. Also have a look at the midterm presentation slides for more information about how and why we gathered data.

In [3]:

```
# It is possible to use the sparsify flag, if you want to test if models throw errors / f
or a quick check.
# USE WITH CAUTION: This is by no means designed to give an appropriate / random sample.
# For long running cells, I also tried to split the visualization with the resource inten
se part, so the notebook can run overnight, and then the output will be generated when yo
u reconnect.
datasets = loading.load_datasets({
    'Flask': 'data/flask_full.pkl',
    'Jinja': 'data/jinja_full.pkl',
    'Httpie': 'data/httpie_full.pkl',
    'Docopt': 'data/docopt_full.pkl'
}, sparsify=False)
```

The mutants_and_tests data structure

The main structure of such datasets is that for each mutant, with its respective `mutant_id`, and every test case, with its respective `test_id`, there is a row in the dataset, recording the outcome of the test, and additional information, like how long the test took, what files where touched, etc. The dataframe index we will use in the whole notebook will always be the same. That is how we can find back datapoints later if we want to.

In [20]:

```
# Have a quick look here:
datasets['Flask'].head()
```

Out[20]:

	mutant_id	contains_branch_mutant	contains_equality_comparison_mutant	contains_loop_mutant	contains_math_operands_m
0	1977.0	1	0	0	
1	1977.0	1	0	0	
2	1977.0	1	0	0	
3	1977.0	1	0	0	
4	1977.0	1	0	0	

5 rows × 29 columns

Visualization

Visualizing data generated by Mutation testing

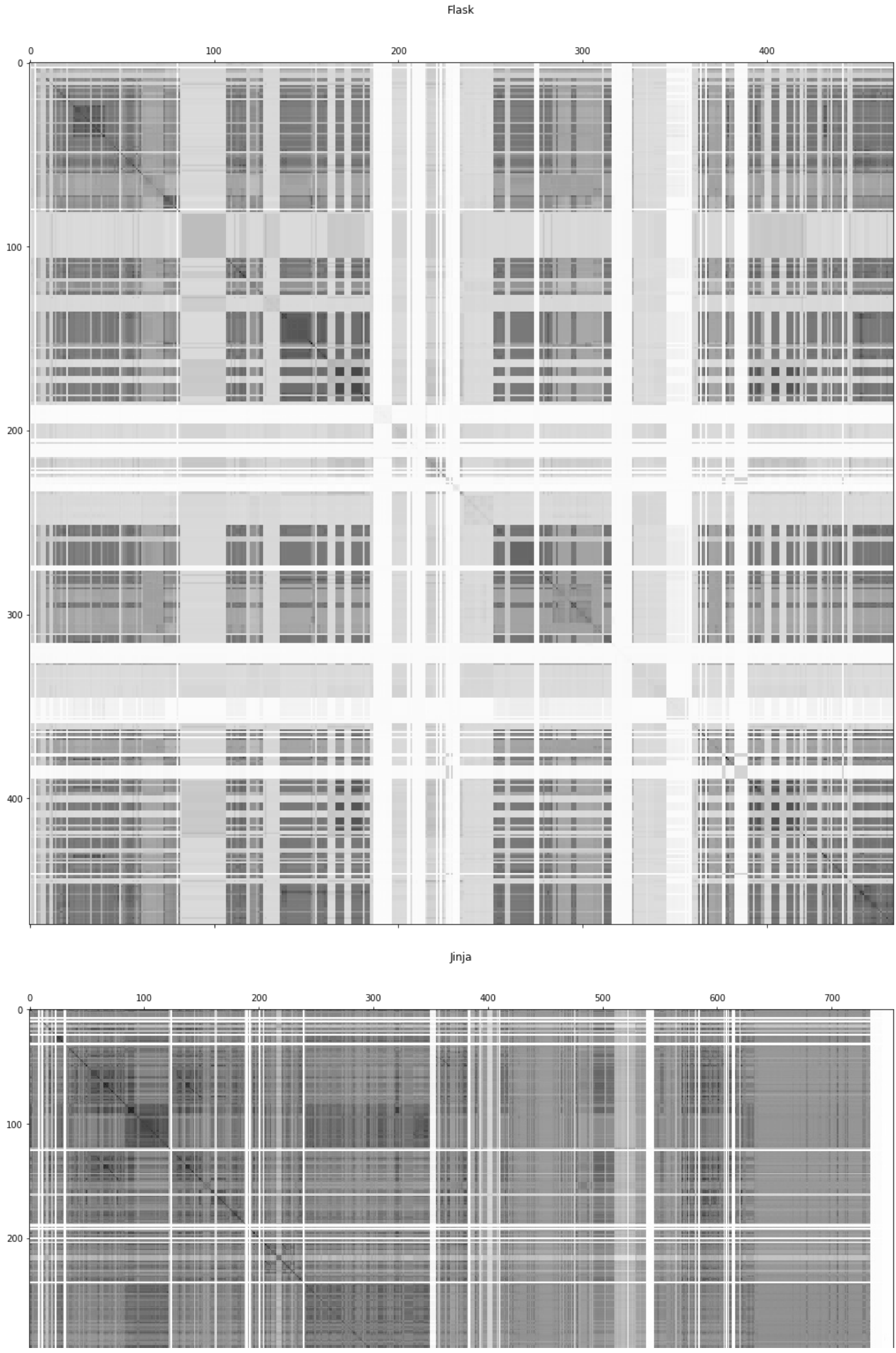
At the beginning of the project, I tried to get a glimpse of the generated data by trying out different visualizations, the most basic beeing the covariance matrixs.

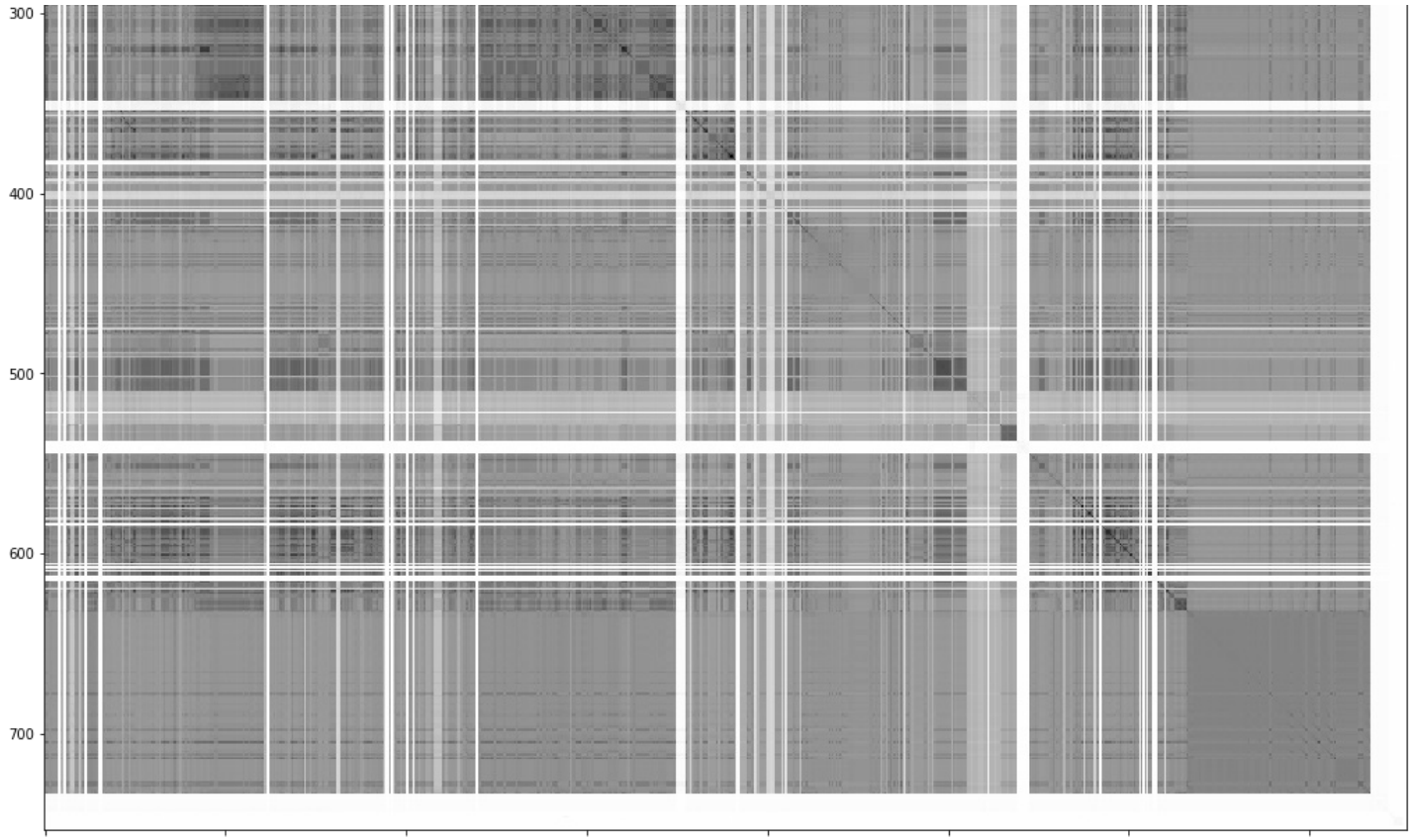
Correlation Matrixes

In [4]:

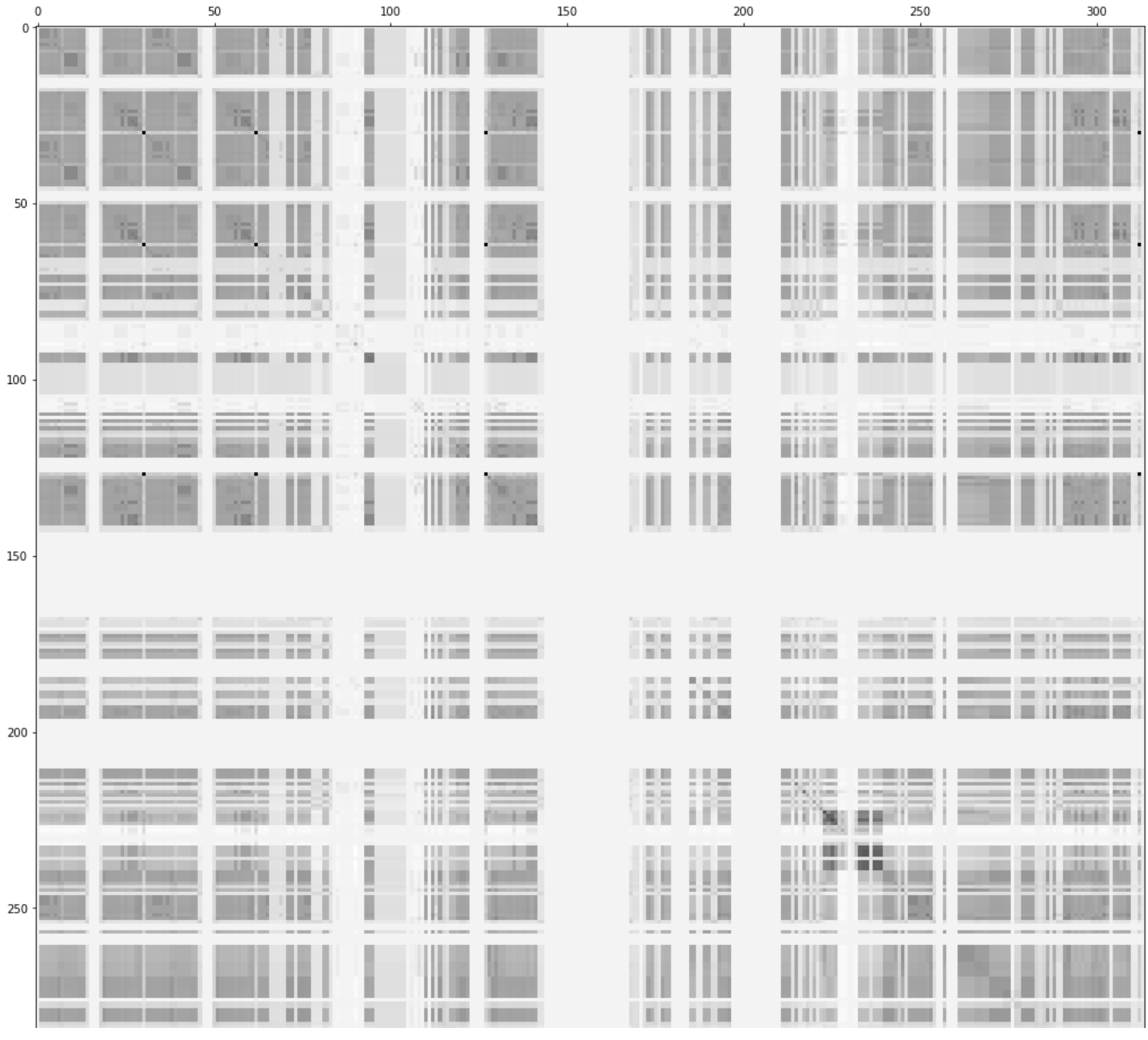
```
for name, mutants_and_tests in datasets.items():
```

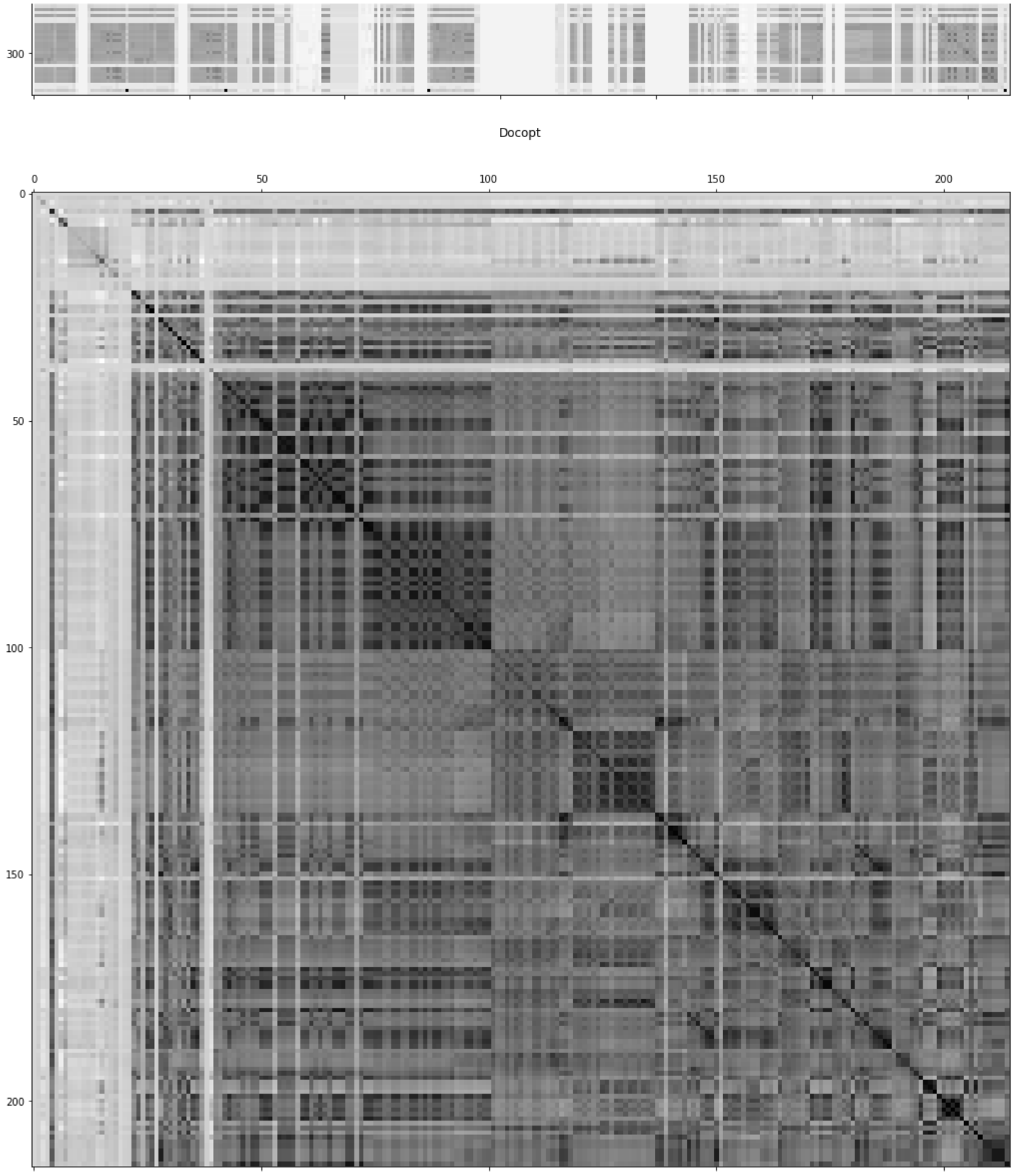
```
for name, mutants_and_tests in datasets.items():
    visualization.plot_covariance_matrix(name, mutants_and_tests)
```





Httpie



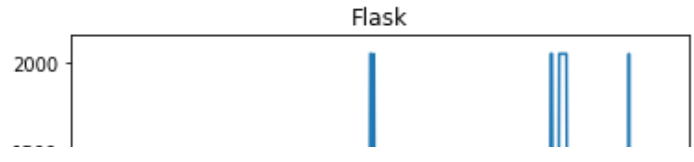


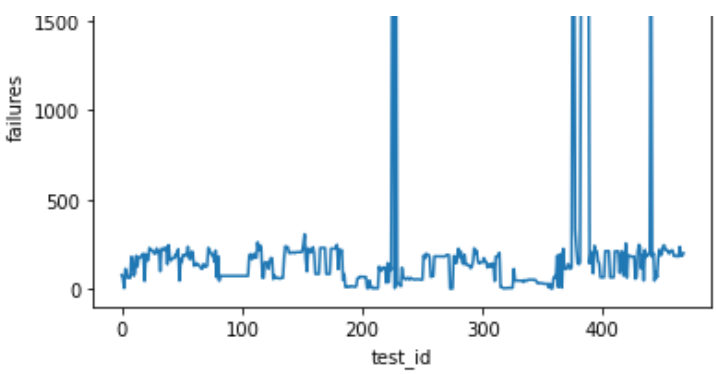
An interesting aspect is also how often tests fail in general, and how the distribution is through the datasets:

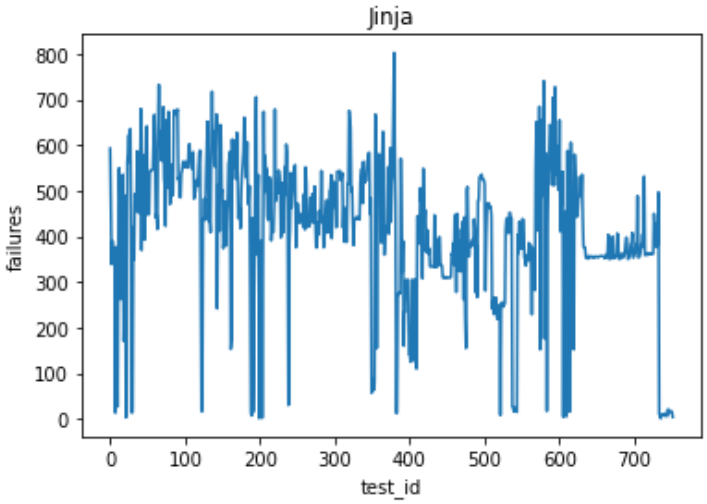
What we can observe is that while `Flask` and `Httpie` have a very low number of often failing tests, `Jinja` and `Docopt` have a wider variety of tests failing. We can see the behaviour also in the covariance matrixes, that are a lot lighter for `Flask` and `Httpie`, due to less covariance in tests often failing together.

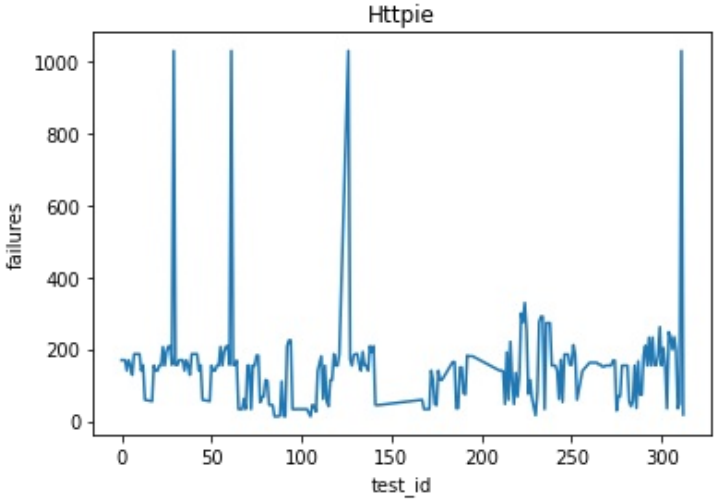
In [5]:

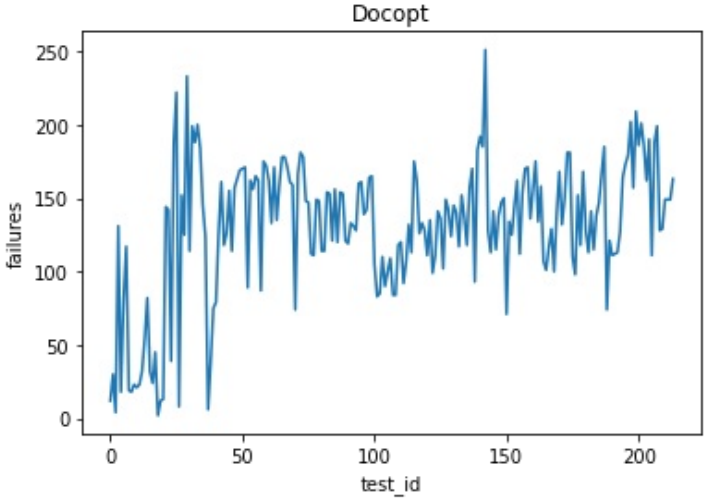
```
for name, data in datasets.items():
    visualization.plot_failure_histogram(name,data)
```









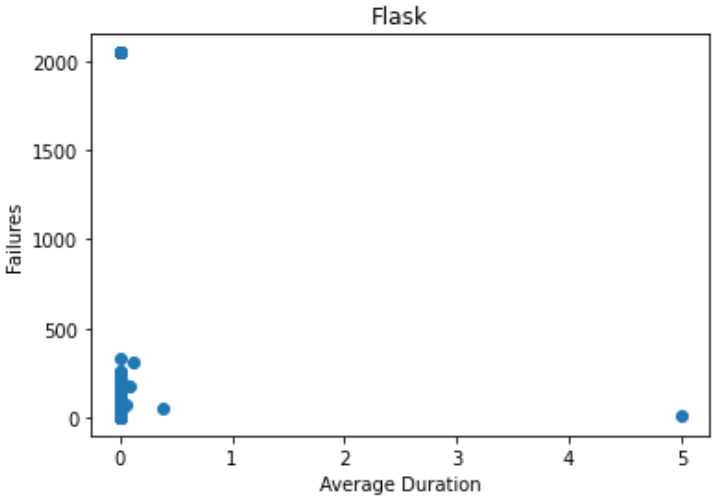


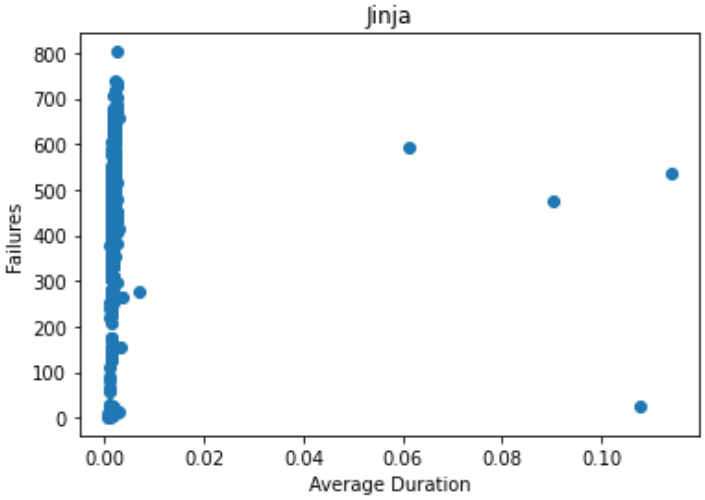
Later, it will also be interesting how the test failures correlates to the average duration of the test in the datasets. You can see this in these scatter plots:

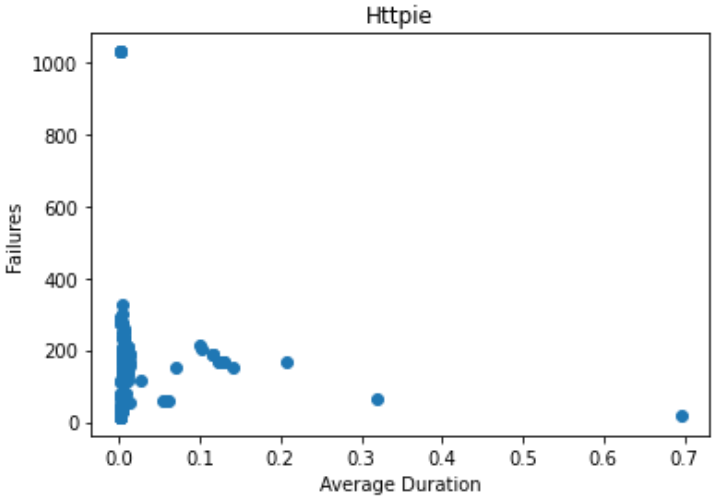
In [6]:

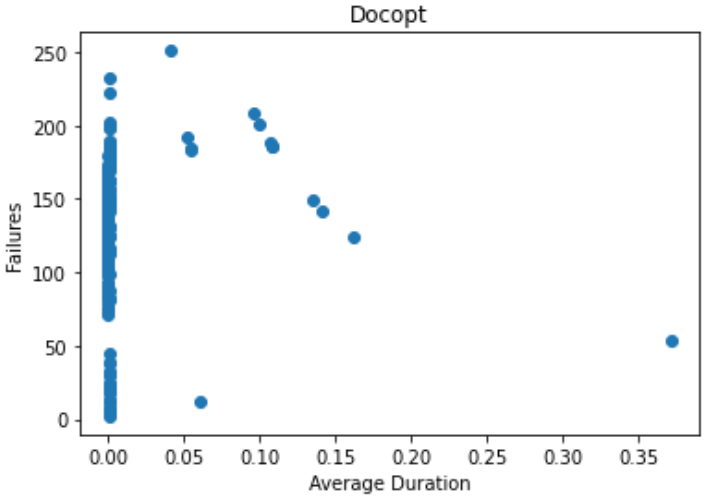
```
for name, data in datasets.items():
```

```
visualization.plot_failures_vs_duration(name,data)
```









Hierarchical Plot of tests

To try to get even more insight into the different datasets, I tried to come up with my own domain specific visualization, that builds on top of the failure histogram.

We print one dot for every test, with the following properties: x-axis = average of ids of a changed line (mutant_ids) that make this test fail y-axis = number of mutants that make this test fail

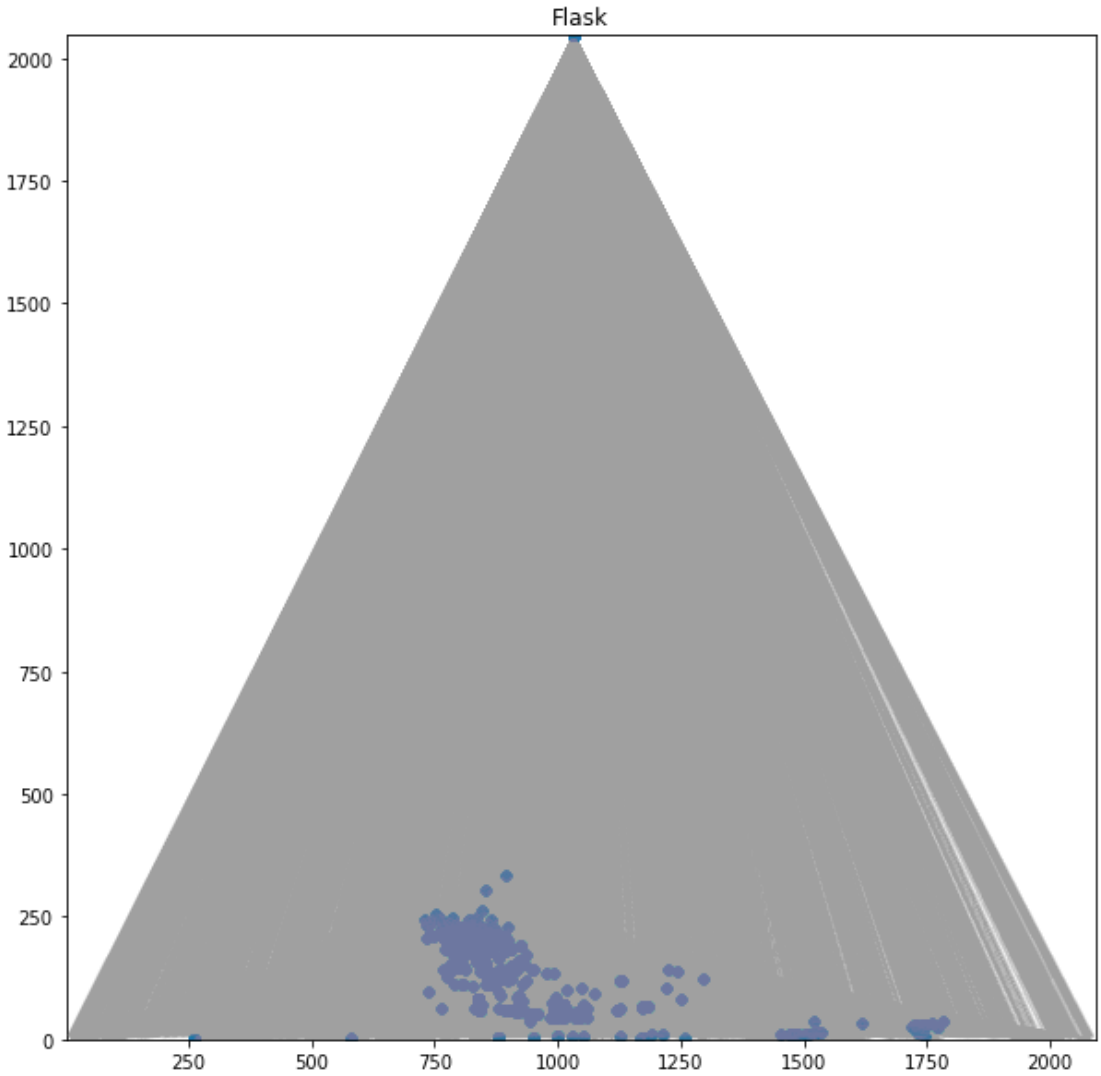
For example a unit test should be relatively wide down, since it should not fail on too many mutants.

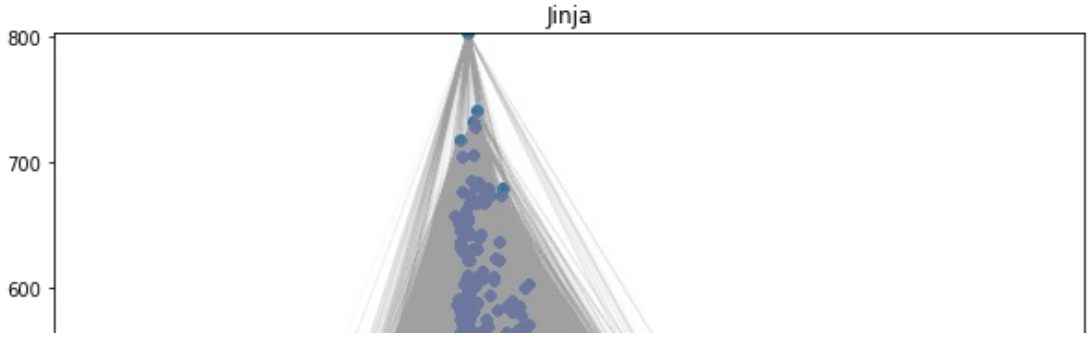
Integration Tests would more often fail, and should have a higher y-coordinate.

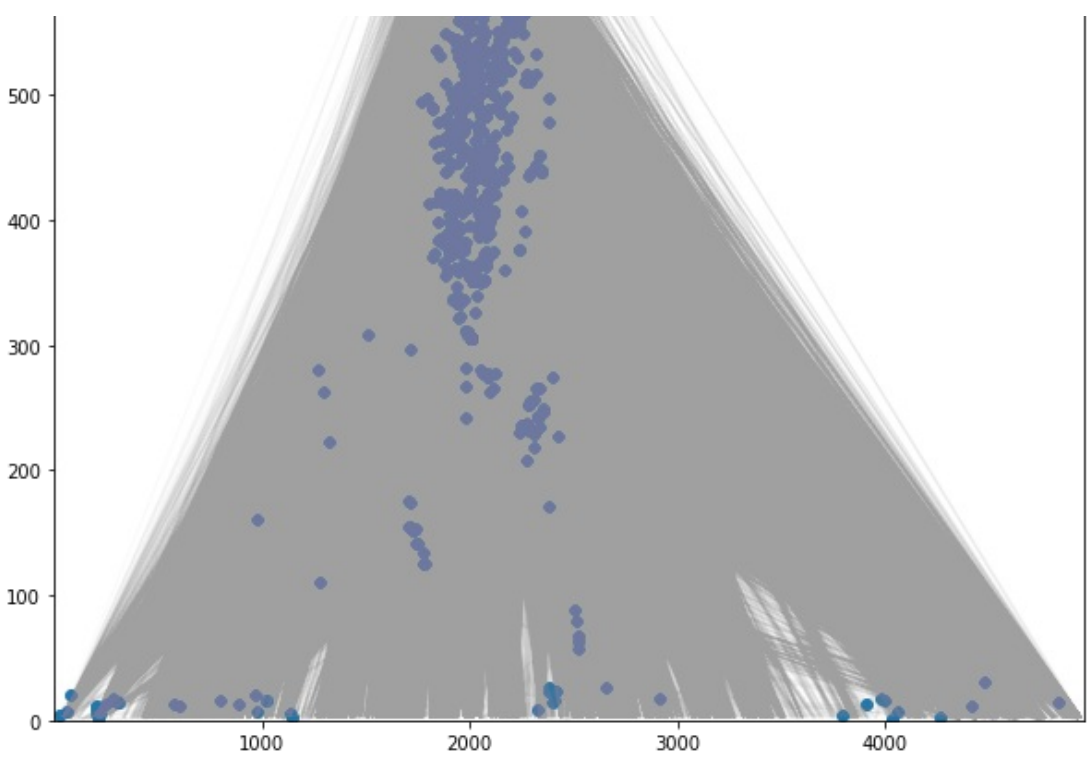
To show the 'spread' of which mutant_ids made which tests fail, we draw an arrow from (mutant_id, 0) to (x,y) of the test, iff mutant_id makes the test fail. For the upper integration tests, we can here see if they reach different mutants, or only some areas.

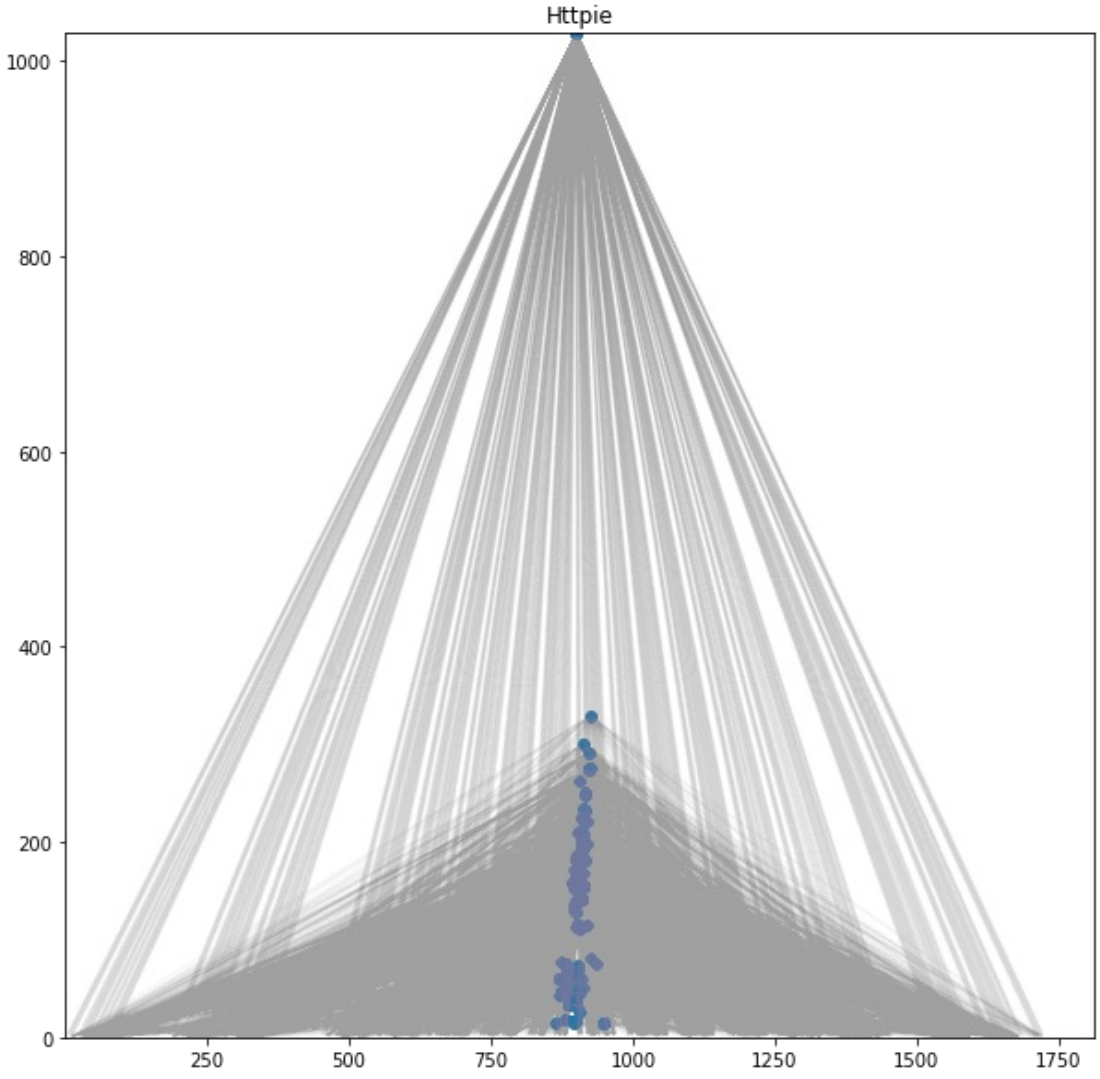
In [7]:

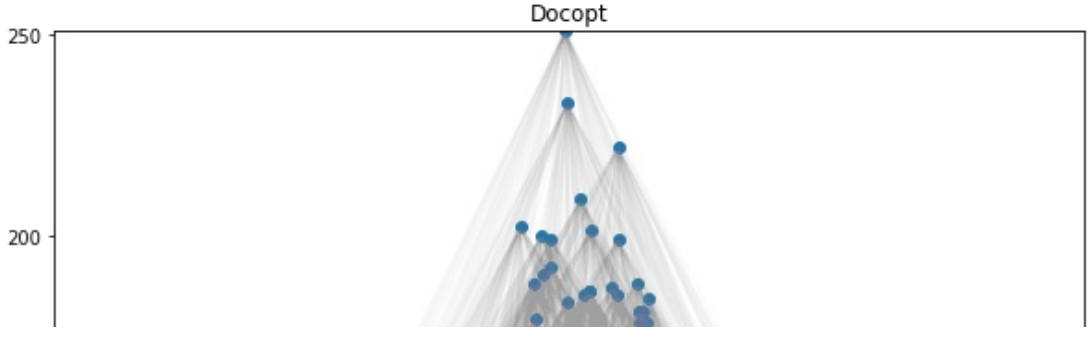
```
# For larger datasets, this may take a while (~5minutes). You can disable the arrows, the
n it becomes seconds
for name, mutants_and_tests in datasets.items():
    visualization.plot_hierarchical_failures(name, mutants_and_tests, arrows=True)
```

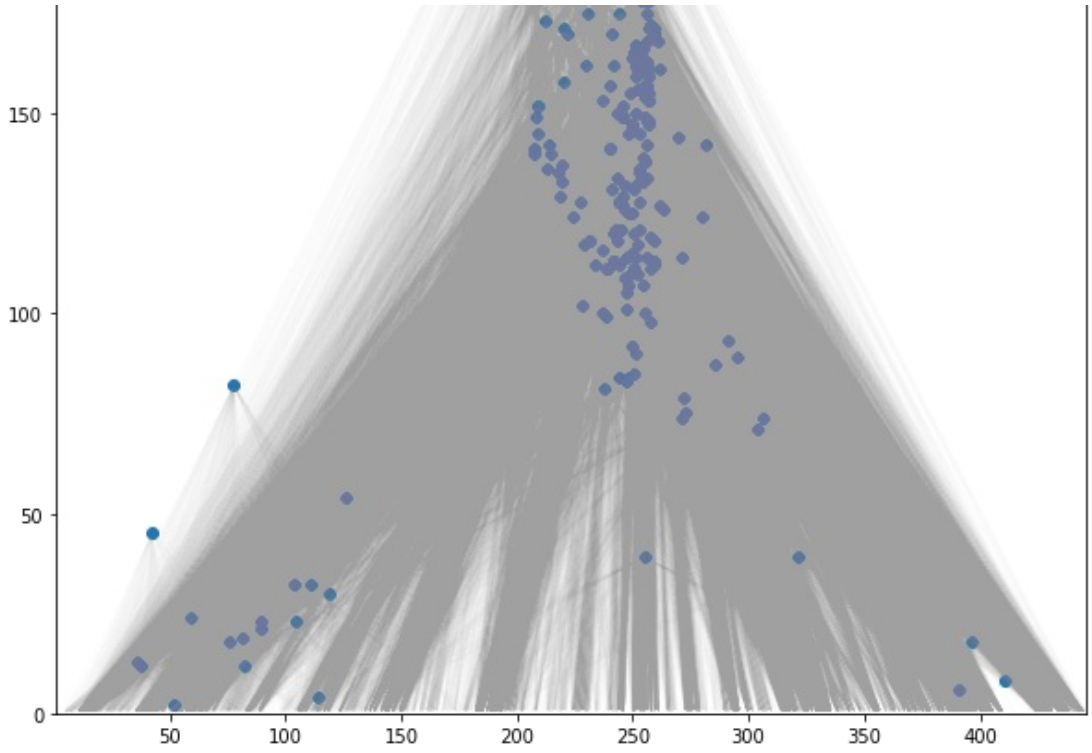





500
400
300
200
100
0
1000
2000
3000
4000


Httpie
1000
800
600
400
200
0
250
500
750
1000
1250
1500
1750


Docopt
250
200



Prediction

Predicting failing tests

After having a first visualization of the data (and we will continue to do that), the question is if we can train machine learning models to predict failing tests. While this has obvious applications in test selection and reordering, we might also be able to find hidden dependencies in the dataset.

Data Preparation

Cleansing and Feature Generation

In [8]:

```
encoded_column_names = ["modified_method", "modified_file_path", "name", "filepath", "cu
rrent_line", "previous_line"]

encoded_datasets = {}
for name, mutants_and_tests in datasets.items():
    preprocessing.cleanse_data(mutants_and_tests)
    preprocessing.add_edit_distance_feature(mutants_and_tests)
    mutants_and_tests = preprocessing.filter_NaN_values(name, mutants_and_tests)
    encoded_datasets[name] = preprocessing.encode_columns(mutants_and_tests, encoded_col
umn_names)
```

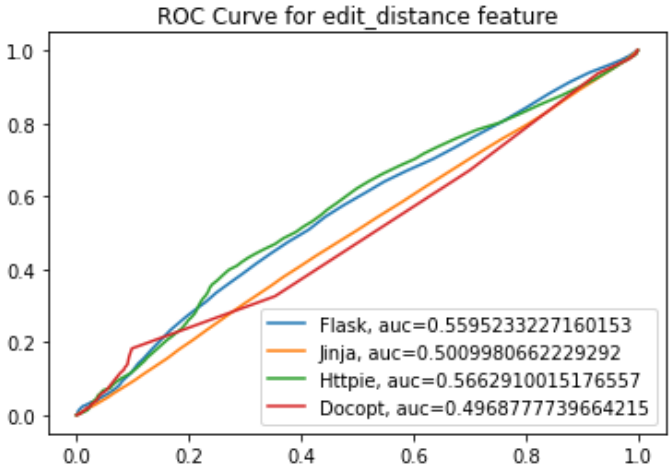
```
Flask: Kicked out 45 values from 958041 total values. (0.004697084989058187%) (NaN value
filter)
Jinja: Kicked out 145 values from 3624070 total values. (0.004001026470235036%) (NaN valu
e filter)
Httpie: Kicked out 605 values from 322369 total values. (0.18767313234212546%) (NaN value
filter)
Docopt: Kicked out 2 values from 95018 total values. (0.0021048643414900425%) (NaN value
filter)
```

The method_name feature seems to make sense from a human perspective: If code is changed in the same method, similar tests are expected to fail. But to go even further, I tried to add a numerical value, the edit distance between the test name and the method name. The expectation here was that when both are similar, the test is more likely to fail. Examples for that are in the presentation slides, page 40

In the ROC Curve, we can see that if we use the edit_distance as a standalone feature, Httpie and Flask behave better than Jinja and Docopt. Remembering that we have previously seen that both of these projects also had a far greater number of 'Unit Tests', that failed only for a small range of mutants and therefore test a specific part of the code, this might be an indicator for a good overall test suite, that is predictable.

In [9]:

```
visualization.plot_edit_distance_roc_curve(datasets)
```



Feature Selection

We can now select the features we want to use for our predictions.

To be on the save side, we previously drop

- **features that could spoiler the outcome of the test (** `dangerous_features` **)**
- **features that are unencoded (** `unencoded_features` **)**

The features are grouped into three categories:

- **Basic features, which are generated with the diff of the mutant, and also test properties**
- **Semantic features, which are generated from the semantics in the mutant**
- **Context features, which are gathered in the code around the test and the mutant See also the [final_presentation_slides, page 37pp]**

In [10]:

```
dangerous_features = ['duration', 'setup_outcome', 'setup_duration', 'call_outcome', 'cal
l_duration', 'teardown_outcome', 'teardown_duration']
unencoded_features = ['repo_path', 'full_name']

selected_features = [
    # Basic features:
    'current_line', 'line_number_changed',
    'modified_file_path', 'previous_line', 'test_id',
    'name', 'filepath', 'mutant_id', 'outcome',
    # Semantic features:
    'modified_method', 'edit_distance',
    # Context features:
    'contains_branch_mutant',
    'contains_equality_comparison_mutant', 'contains_loop_mutant',
    'contains_math_operands_mutant', 'contains_loop_execution', 'contains_math_operands_e
xecution',
    'contains_equality_comparison_execution', 'contains_branch_execution'
]


for name, mutants_and_tests in encoded_datasets.items():
    encoded_datasets[name] = mutants_and_tests.drop(dangerous_features, axis=1).drop(une
ncoded_features, axis=1)[selected_features].copy()
```

In [11]:

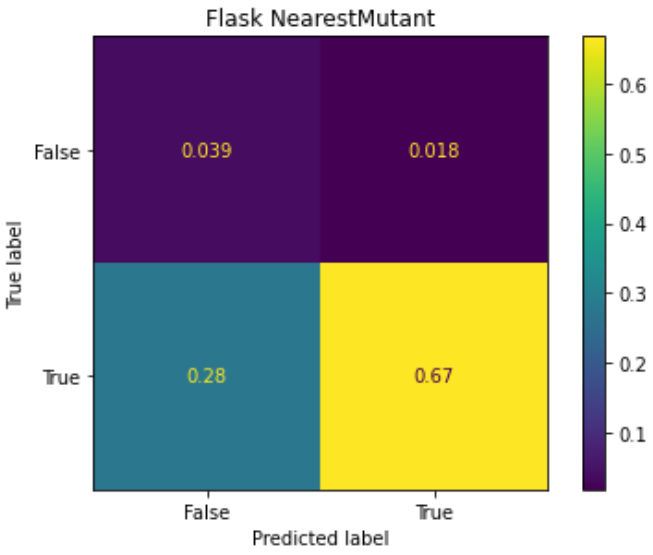
```
test_train_data = {}

for name, mutants_and_tests in encoded_datasets.items():
    test_train_data[name] = preprocessing.train_test_split(mutants_and_tests)
```

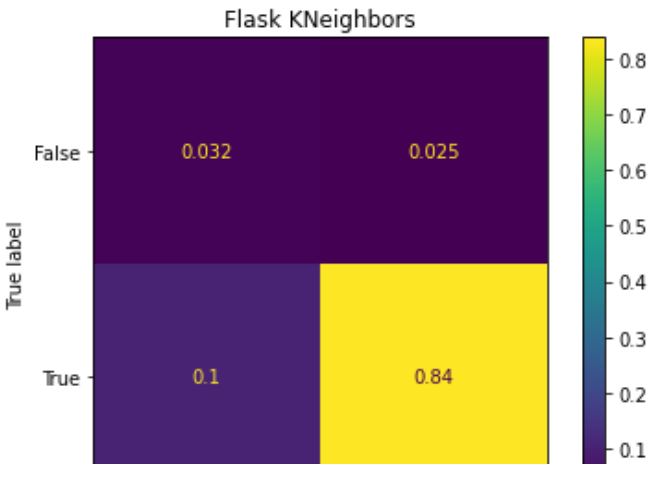
In [12]:

```
# The training here takes a long time, depending on the model it can go over hours
predictors = {
    'NearestMutant': predictor.NearestMutantPredictor(),
    'KNeighbors': KNeighborsClassifier(n_neighbors=1),
    'DecisionTree': tree.DecisionTreeClassifier(),
    'RandomForest': RandomForestClassifier(random_state=420),
    'GradientBoosing': GradientBoostingClassifier(random_state=420),
}

fitted_predictors = {}

for dataset_name, mutants_and_tests in encoded_datasets.items():
    X_train, y_train, X_test, y_test = test_train_data[dataset_name]
    fitted_predictors[dataset_name] = {}
    for predictor_name, predictor_instance in predictors.items():
        predictor_instance = copy.deepcopy(predictor_instance)
        predictor_instance.fit(X_train, y_train)
        fitted_predictors[dataset_name][predictor_name] = predictor_instance
```

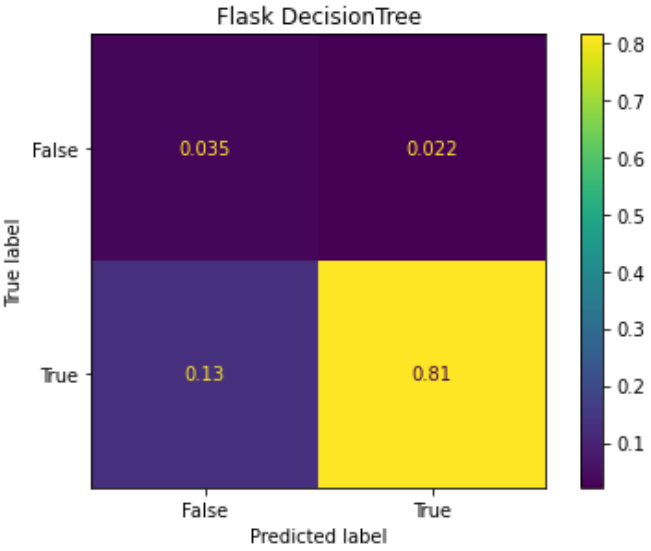
In [13]:

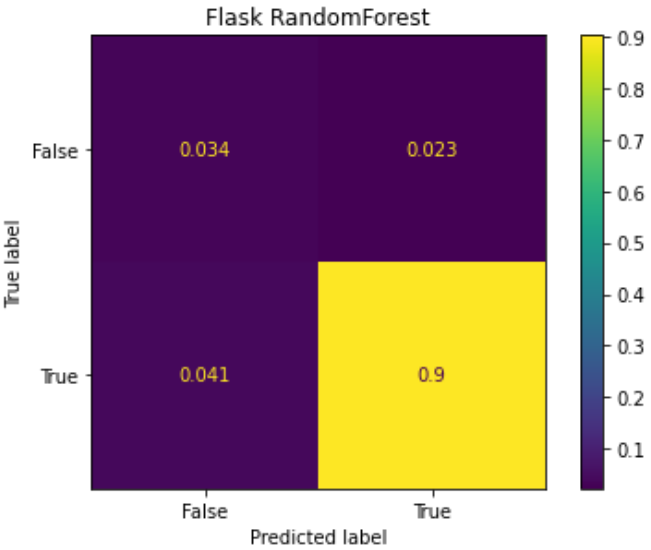
```
# Splitted visualization, so if reconnected to the notebook, one can get the visualizatio
n without having to train again
for dataset_name, mutants_and_tests in encoded_datasets.items():
    X_train, y_train, X_test, y_test = test_train_data[dataset_name]
    for predictor_name, predictor_instance in fitted_predictors[dataset_name].items():
        visualization.plot_confusion_matrix(dataset_name + ' ' + predictor_name, predict
or_instance, X_test, y_test)
```

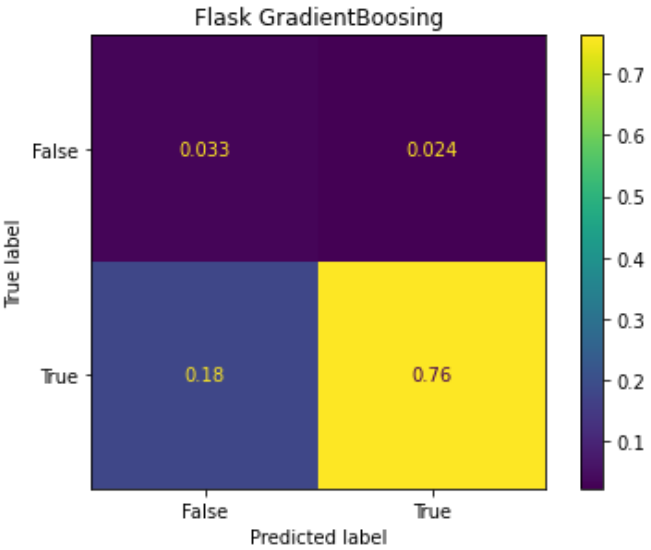


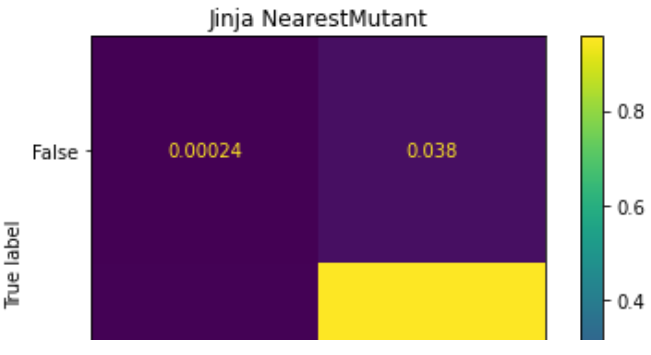


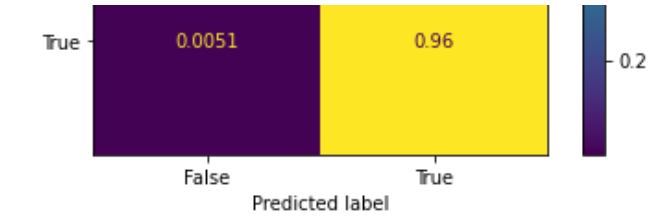

False
True
Predicted label

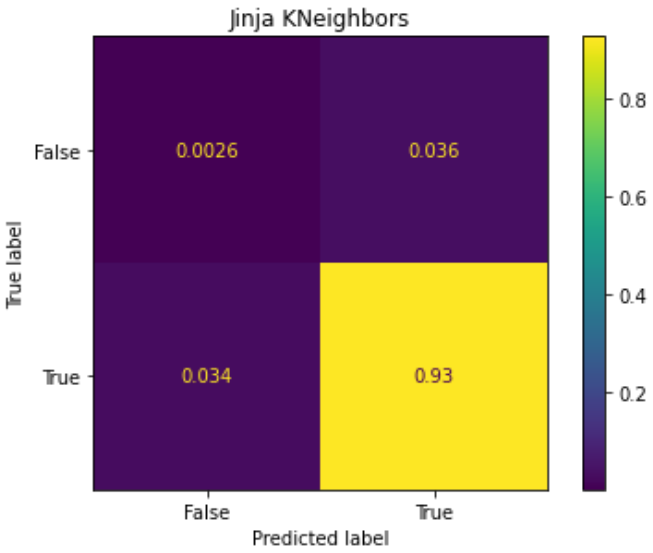

Flask DecisionTree
False
True
True label
0.035
0.022
0.13
0.81
False
True
Predicted label
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

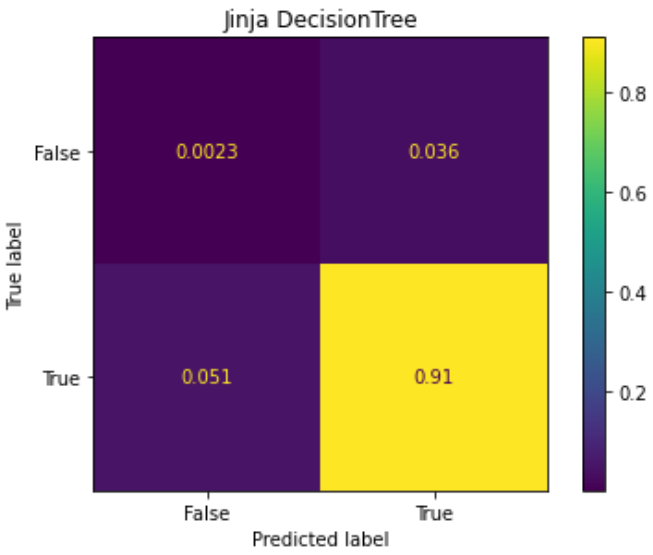

Flask RandomForest
False
True
True label
0.034
0.023
0.041
0.9
False
True
Predicted label
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

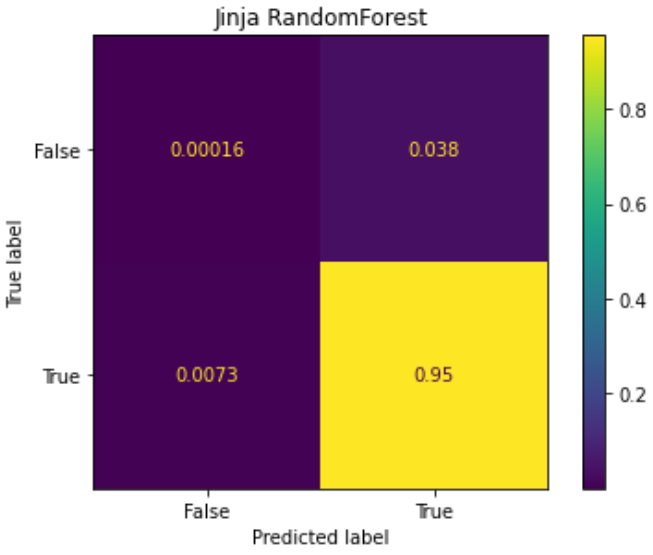

Flask GradientBoosing
False
True
True label
0.033
0.024
0.18
0.76
False
True
Predicted label
0.7
0.6
0.5
0.4
0.3
0.2
0.1

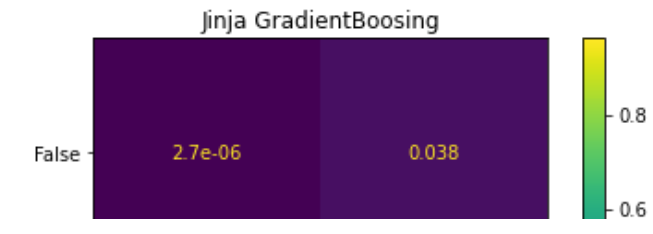

Jinja NearestMutant
False
True label
0.00024
0.038
0.8
0.6
0.4

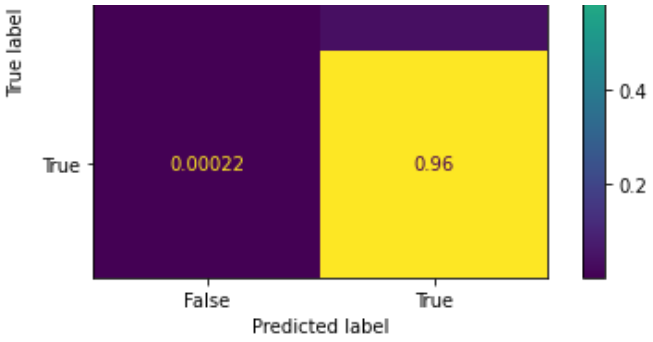

True
0.0051
0.96
False
True
Predicted label
0.2


Jinja KNeighbors
False
0.0026
0.036
True
0.034
0.93
True label
False
True
Predicted label
0.8
0.6
0.4
0.2

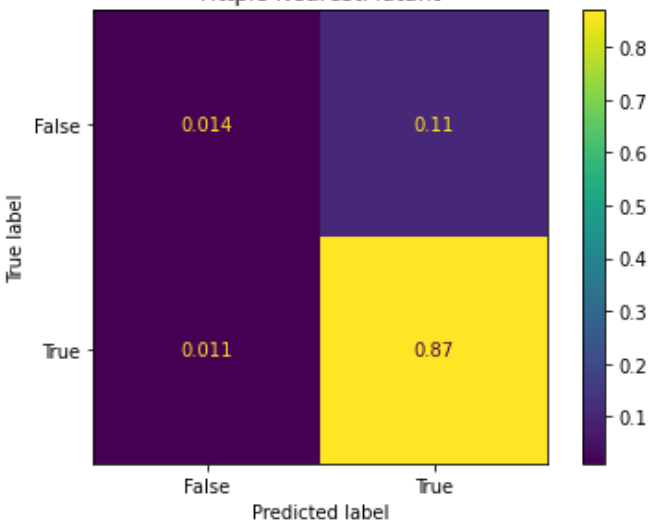

Jinja DecisionTree
False
0.0023
0.036
True
0.051
0.91
True label
False
True
Predicted label
0.8
0.6
0.4
0.2


Jinja RandomForest
False
0.00016
0.038
True
0.0073
0.95
True label
False
True
Predicted label
0.8
0.6
0.4
0.2

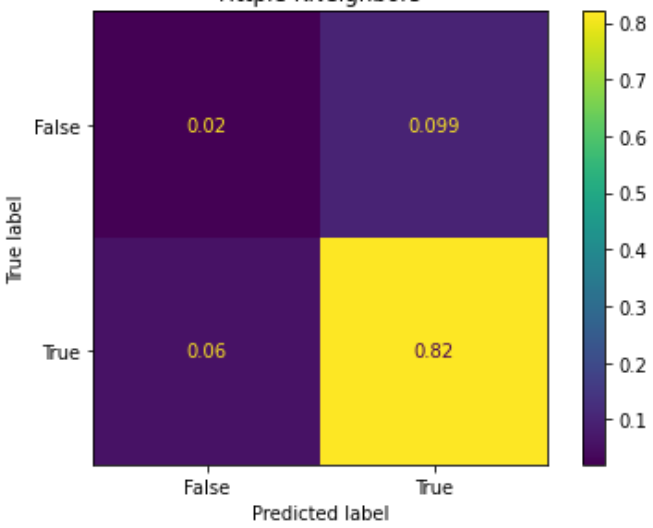

Jinja GradientBoosing
False
2.7e-06
0.038
0.8
0.6



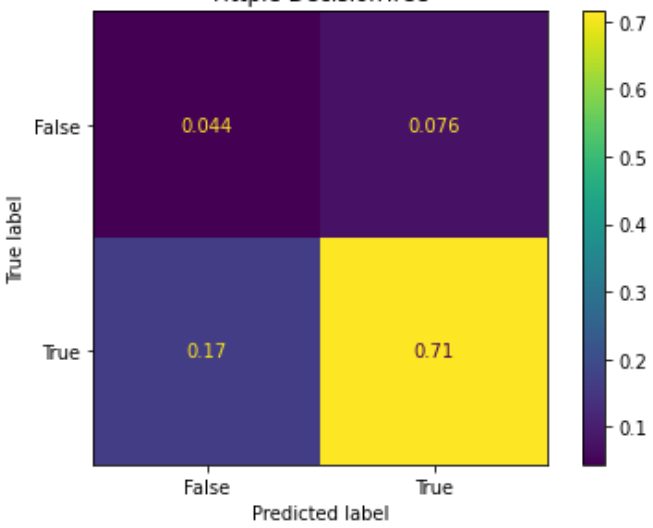
Httpie NearestMutant



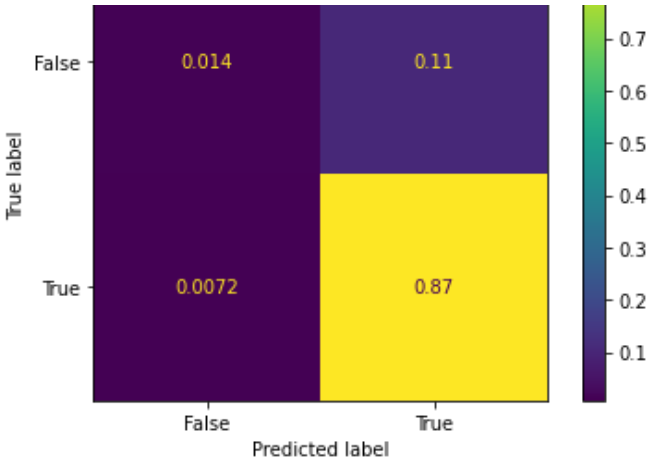
Httpie KNeighbors

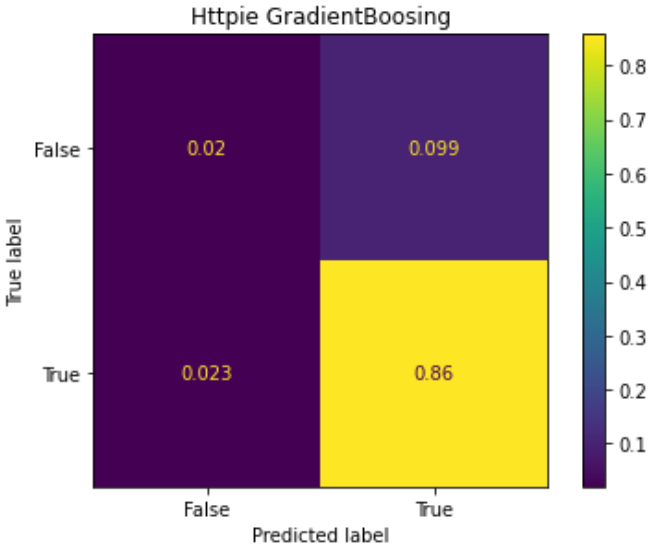


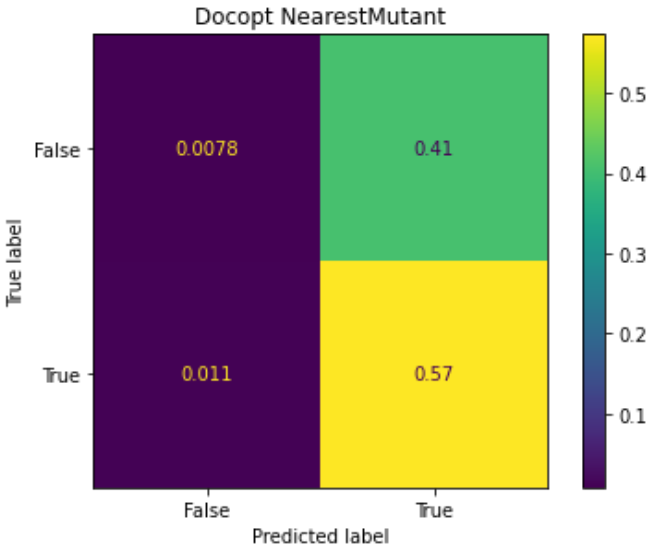
Httpie DecisionTree

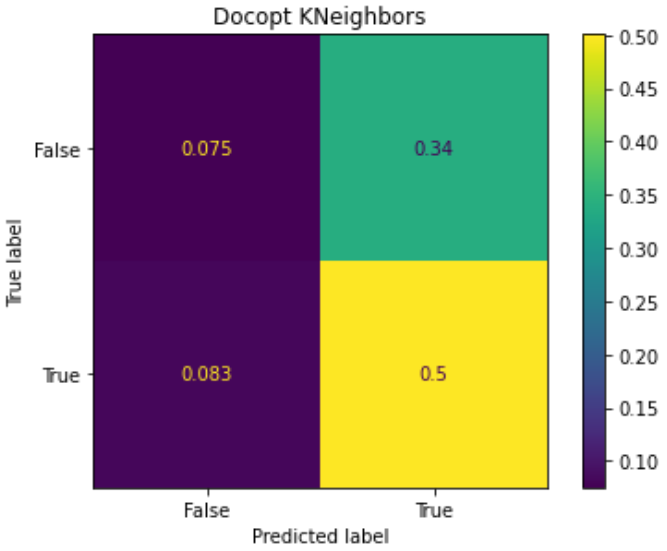


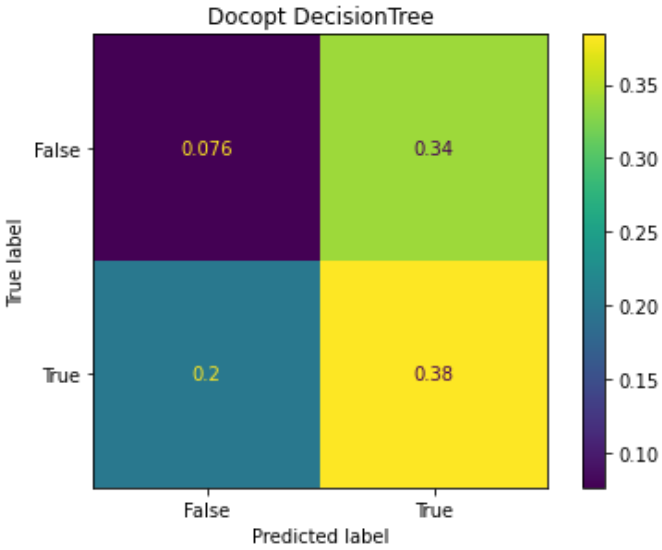
Httpie RandomForest

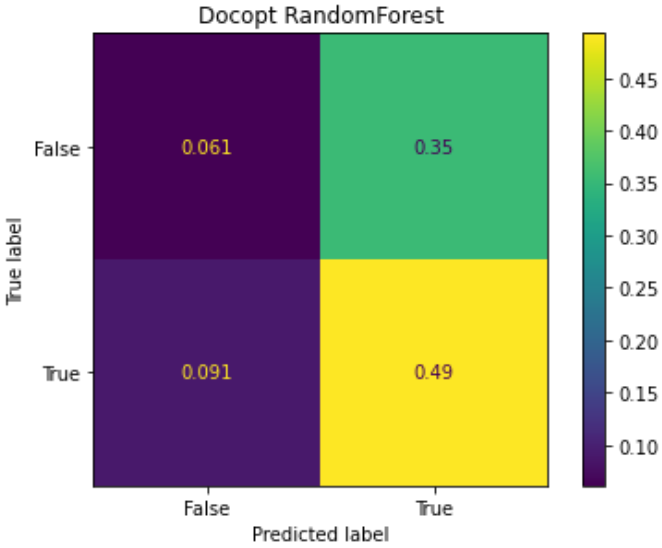

False
True
True label
0.014
0.11
0.0072
0.87
False
True
Predicted label
0.7
0.6
0.5
0.4
0.3
0.2
0.1

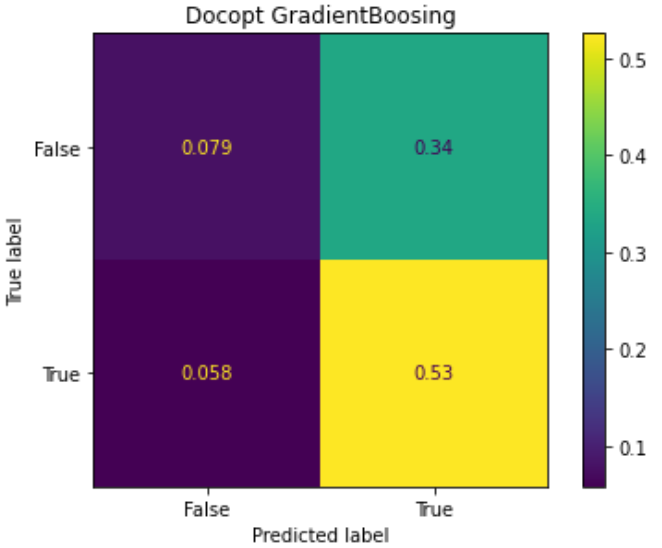

Httpie GradientBoosing
False
True
True label
0.02
0.099
0.023
0.86
False
True
Predicted label
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


Docopt NearestMutant
False
True
True label
0.0078
0.41
0.011
0.57
False
True
Predicted label
0.5
0.4
0.3
0.2
0.1


Docopt KNeighbors
False
True
True label
0.075
0.34
0.083
0.5
False
True
Predicted label
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10







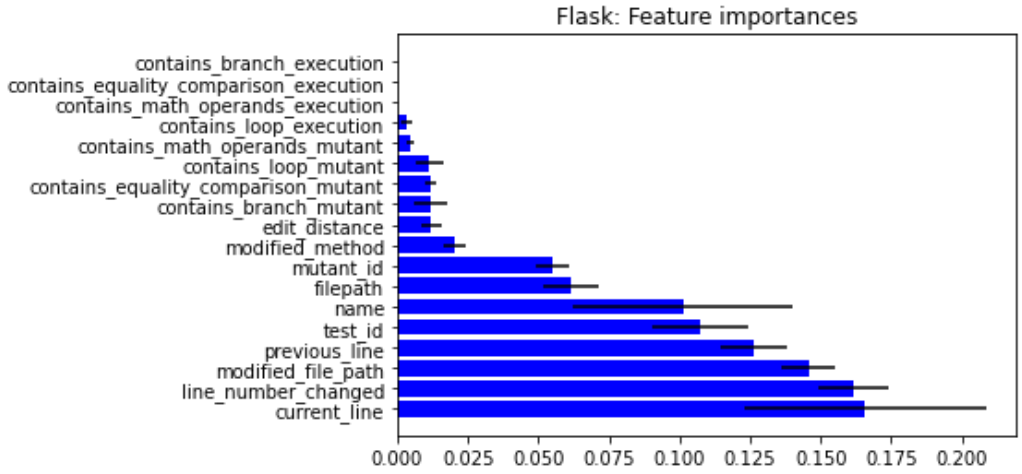
We are especially interested in the "True False Predicted False" score, as we can use that to make a useful reordering for example. False positives are not too bad in this scenario, since we might order them first, but that does not hurt too much.

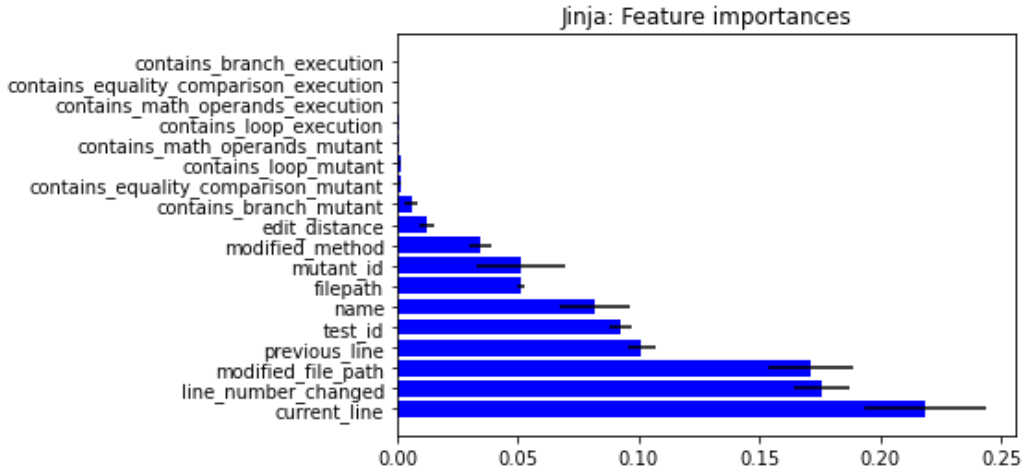
However the true negative rate is bad overall, only differing slightly between the datasets and predictors. This is bad if we want to find the tests that fail.

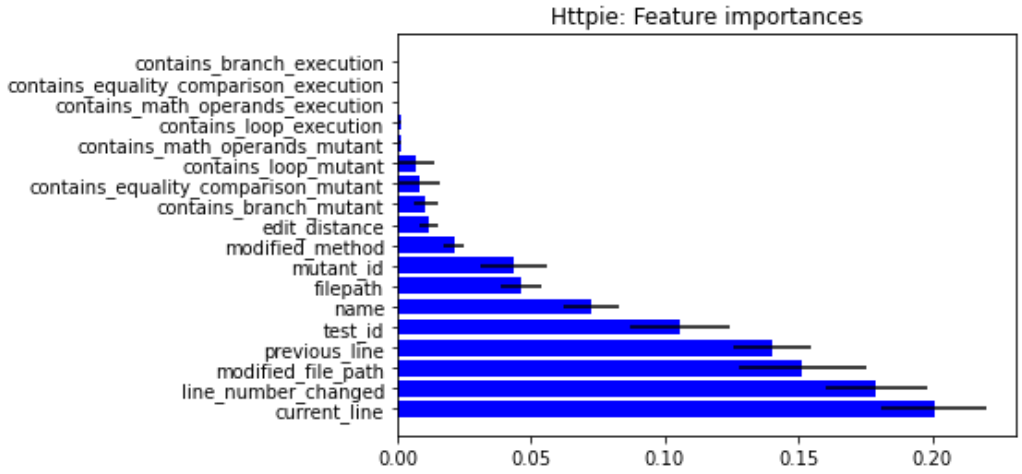
To get a glimpse on what the classifiers learn in the dataset, we have a look at the RandomForestClassifier. One nice thing about this classifier is that we can try to debug it using its feature importances. These can hint on what the forest uses to decide between a false and true outcome.

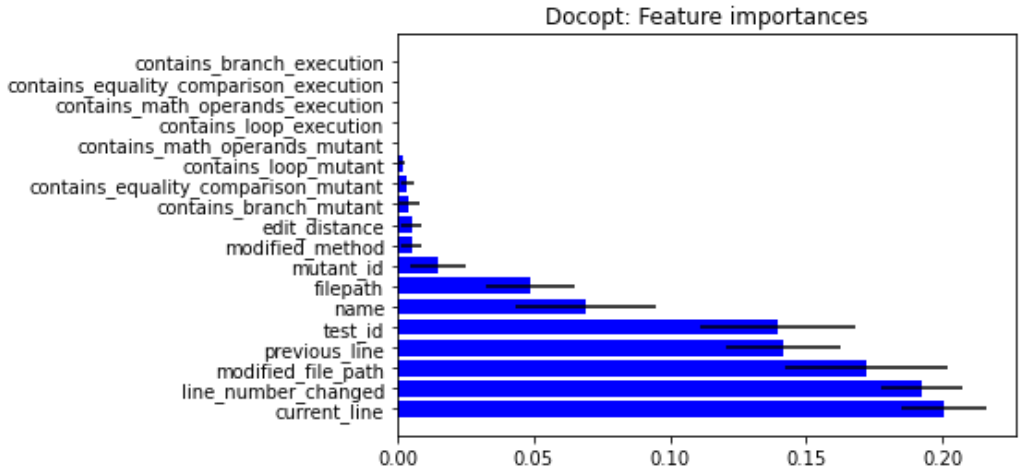
In [14]:

```
for name, fitted_predictor in fitted_predictors.items():
    visualization.plot_feature_importances(name, fitted_predictor['RandomForest'], test_t
rain_data)
```









Here we can see that for the Flask and Httpie dataset, the random forest makes more usage of the 'semantic'

Here we can see that for the Flask and Httpie dataset, the random forest makes more usage of the 'semantic' and 'context' features in the dataset, which might indicate that the tests in that project are especially well organized. It would be an interesting point of research to check whether this can also be perceived by humans as a 'predictable' test suite.

Reordering

Reordering the test cases to improve the developer experience

While it is nice to be able to predict which tests fail and which not, it is easier to decide if a classification is good based on a concrete use case, since more specific metrics can be used. Therefore, the predictors are tested on various metrics in the test case reordering domain.

- APFD, which is a measure on how good a order of tests is in general, but which is duration agnostic
- APFDc, which is APFD weighted after the average durations of the mentioned tests
- First Failing, which is the time that emerges till a test is found that fails
- Last Failing, which is the time that emerges till the last test that fails is found

There are multiple ways to derive an order for test case execution. To bridge the gap between prediction and reordering, two main helper classes exist:

- `BinaryPredictionReorderer`, which takes a binary predictor, and moves the (predicted) failing tests to the front
- `OrdinaryPredictionReorderer`, which sorts the tests ascending after there (predicted) chance of failing

For a baseline we use Mutant agnostic Reorderers. That means that they will output the same order for every mutant:

- `NaiveReorderer`, which just outputs the tests after ascending test_ids
- `AverageReorderer`, which orders the tests after there overall a priori probability to fail
- `QTF`, which is sorting the tests ascending after there duration. It is a heuristic mentioned in this paper.

The `ReorderingAnalyzer` class can be used to automatically train and evaluate the reorderes, and print boxplots:

In [15]:

```
# This step takes not only long to train, but also long to evaluate, with a lot of predic
tors it will take hours. Every 50 orders processed there will be a dot printed,
# so you can get a feeling on how long it will take.

evaluations = {}
for name, mutants_and_tests in encoded_datasets.items():
    evaluation = ReorderingAnalyzer([
        NaiveReorderer(),
        AverageReorderer(),
        QTF(datasets[name][['test_id', 'duration']]),
        BinaryPredictionReorderer(tree.DecisionTreeClassifier()),
        BinaryPredictionReorderer(KNeighborsClassifier()),
        BinaryPredictionReorderer(RandomForestClassifier(random_state=420)),
        OrdinalPredictionReorderer(RandomForestClassifier(random_state=420)),
        BinaryPredictionReorderer(GradientBoostingClassifier(random_state=420)),
        OrdinalPredictionReorderer(GradientBoostingClassifier(random_state=420))
    ])
    X_train, y_train, X_test, y_test = preprocessing.train_test_split(mutants_and_tests)
    evaluation.fit(X_train, y_train)
    evaluation.predict(X_test)
    evaluation_data = evaluation.evaluate(datasets[name])
    evaluations[name] = evaluation
```

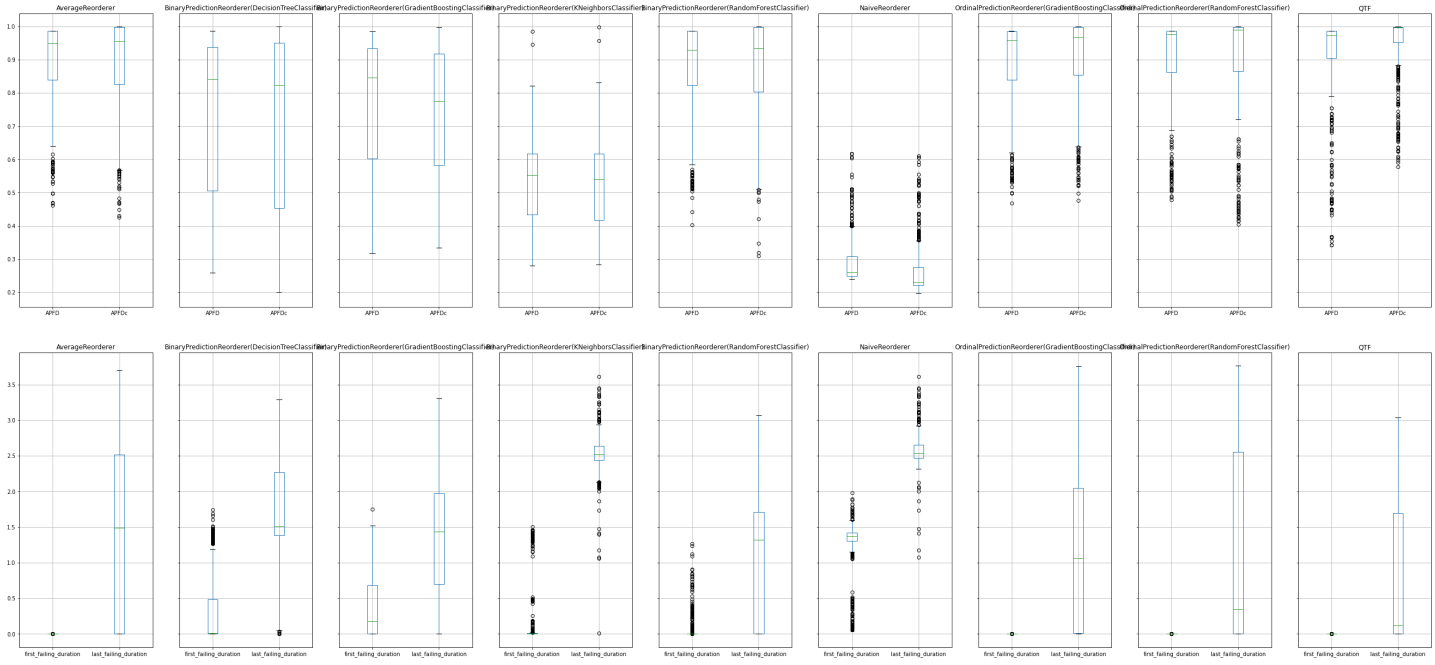
```
Starting evaluation .......... finished.
Starting evaluation .......... finished.
Starting evaluation .......... finished.
Starting evaluation .......... finished.
Starting evaluation .......... finished.
Starting evaluation .......... finished.
```

```
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ............................. finished.
Starting evaluation ............................. finished.
Starting evaluation ............................. finished.
Starting evaluation ............................. finished.
Starting evaluation ............................. finished.
Starting evaluation ............................. finished.
Starting evaluation ............................. finished.
Starting evaluation ............................. finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation ..... finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
Starting evaluation .. finished.
```

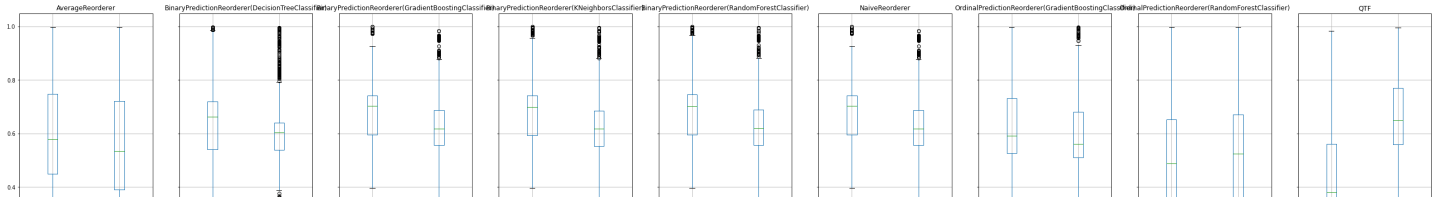
In [17]:

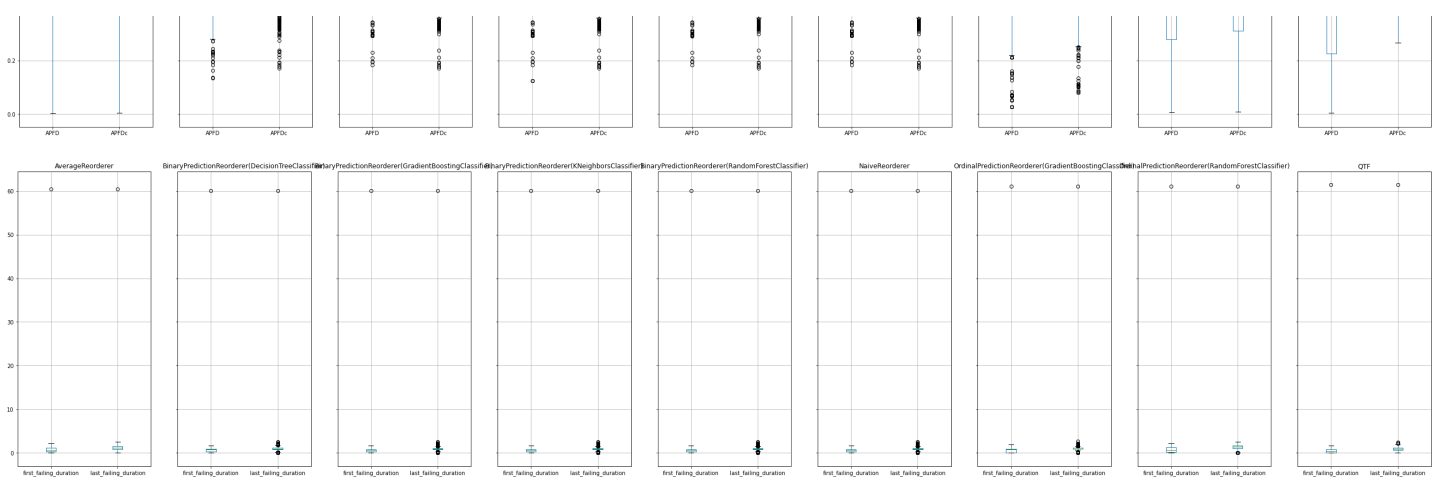
```
# Splitted visualization, so if reconnected to the notebook, one can get the visualizatio
n without having to train again
for name, evaluation in evaluations.items():
    print('Dataset: ' + name)
    evaluation.boxplot()
    plt.show()
```

```
Dataset: Flask
```

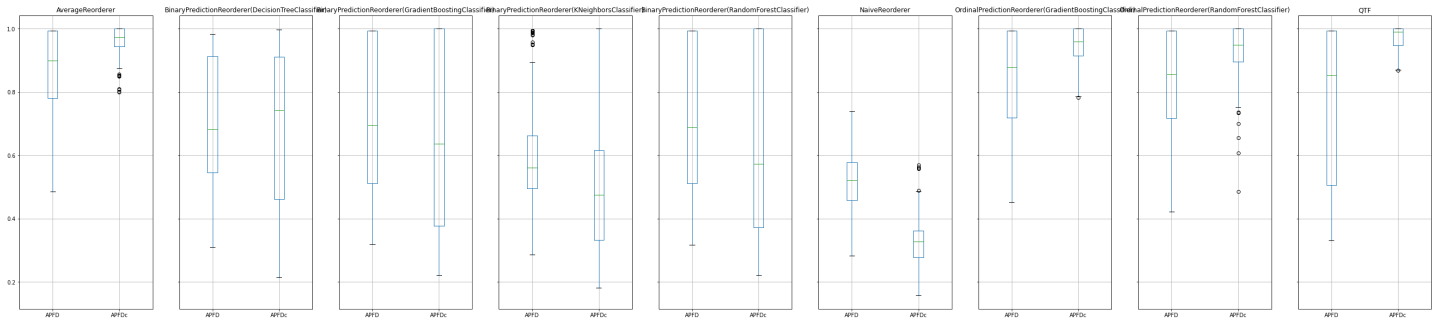


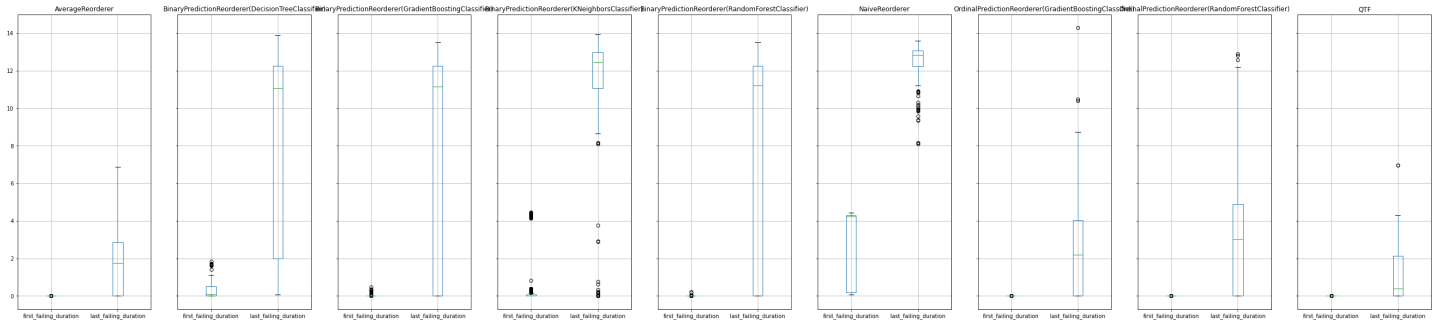

```
Dataset: Jinja
```



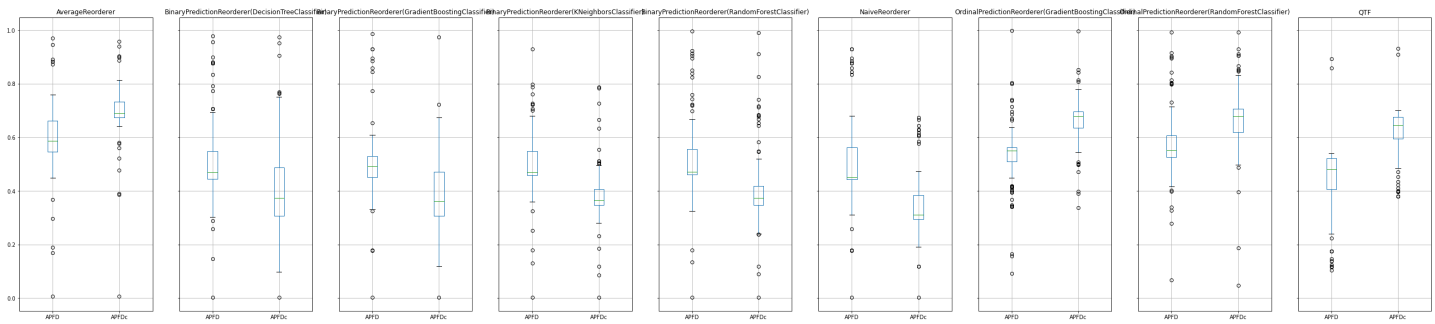


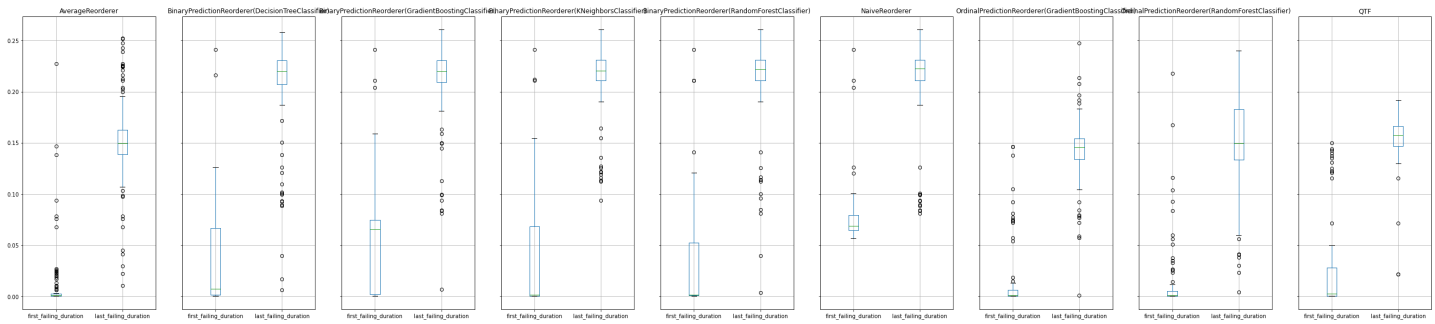
Dataset: Httpie





Dataset: Docopt





A first interesting observation is that indeed `QTF` does not perform much worse on any of the datasets, and that even in the duration agnostic APFD measure. Remember that this is a simple heuristic, that does not need mutation testing data beforehand to train, but only the average durations of test case runs.

We also see again the phenomena that Flask and Httpie seem to be better predictable, while Jinja and Docopt have overall worse metric scores.

One could use some Hyperparameter-Tuning to improve some of the values seen in the boxplots. Even training based on a reordering instead of a prediction of individual test cases would be possible. This single test prediction approach has however, the potential to be also applied in other use cases. For example test case selection, where only a subset of the tests is run.

An interesting aspect could also be the analysis of the outliers for APFD and APFDc seen with all of the reorderers. Do these mutants maybe uncover hidden dependencies in the tests, as they are really unpredictable? Are the mutants that are hard to reorder the same for the datasets?

Challenges

The externel validity of mutation testing failure data is extremly hard to validate. Since the GH-Torrent-Dataset does not contain any python build logs, I tried to scrape some python projects, including `Flask` and `Jinja` with a server script build on the BuildScour project. You can find the used script here. But there are two main problems with gathered build logs on this way: A lot of them don't fail, and if they fail, it is often not from testing. From the 525, only 63 failed, and from them only ~10 had failures in the test suite, which makes it impossible to make statistically proven arguments about the data found. If however one downloads these files regularly over a longer period of time, the data set might become useful.