MACHINE LEARNING NANODEGREE - CAPSTONE PROJECT

by Sven Görres

DEFINITION

DOMAIN BACKGROUND

For my capstone project I want to use the data provided by Mercedes Benz on kaggle[1] ("Mercedes-Benz Greener Manufacturing"). The original idea of this kaggle project is to identify potential for using test stations more efficiently thus reducing emissions (since in this case it's car test stations) and subsequently costs by increasing capacity. This is a general issue in many manufacturing industrial fields since every test station has a relatively high initial cost and even being able to test 5 more parts a day on a single test station can lead to more than one thousand parts tested more per year thus the total amount of deliverable product does increase.

PROBLEM STATEMENT

Based on testing data of a product (in this case a car) predict the assumed testing time for this specific feature combination.

My idea is to use a decision tree based model for this. Since the features are in mostly binary (1 or 0) it should be possible to create a (though very deep and wide) tree that should be able to predict the overall time rather well. Also the non binary features should be easily includable.

METRICS

As a measuring instrument I want to use the coefficient of determination (R^2[2]) as intended by Mercedes-Benz in the first place. This is also applicable to the benchmark model.

ANALYSIS

DATA EXPLORATION

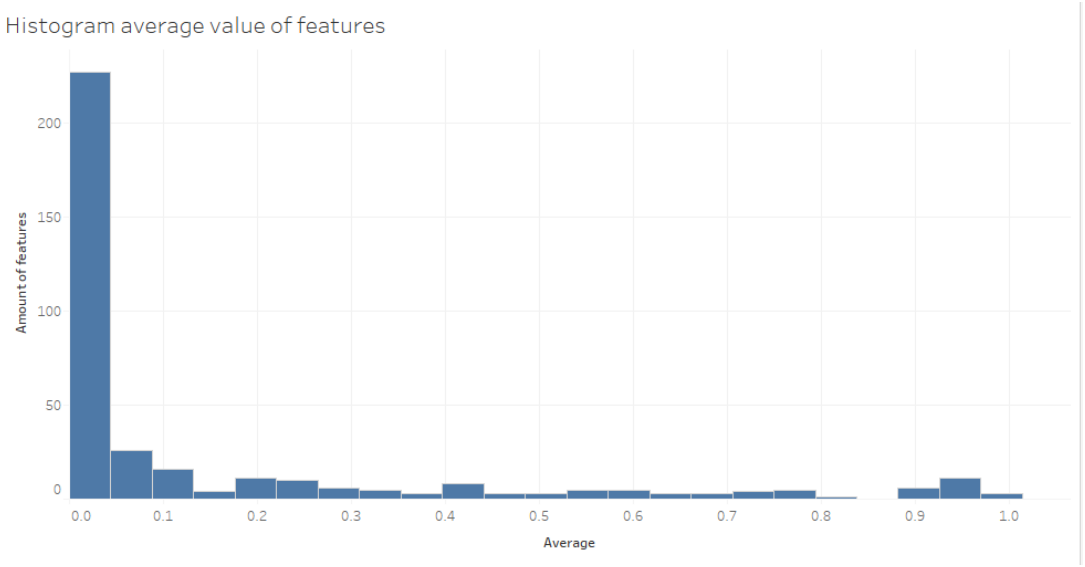
The dataset used for this poject is obtained from kaggle[3]. It consists of a training data set with 4209 entries including resulting time and a testing data set with again 4209 entries but without the resulting time.
The data itselft contains of a unique test ID (incrementally count), the time consumed for the test (unit unknown) and 382 features unique to this test ID. Every feature could stand for e.g. colors or

[1] https://www.kaggle.com/c/mercedes-benz-greener-manufacturing
[2] https://en.wikipedia.org/wiki/Coefficient_of_determination
[3] https://www.kaggle.com/c/mercedes-benz-greener-manufacturing

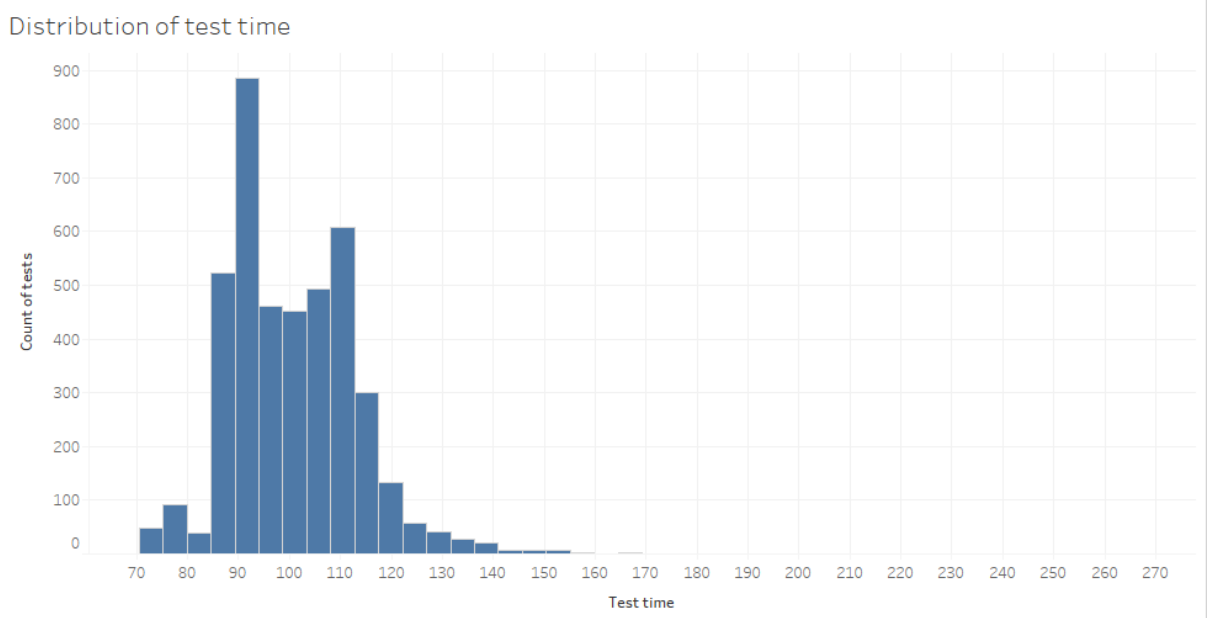
different sales components of the car (e.g. does it have bluetooth connectivity?). The data appears to be complete without any feature missing for any entry.

Every test contains more attributized features X0 to X8 which appear to be string codings. This will be addressed during data cleanup.

Basic statistics for every feature shows that the distribution of values (0 and 1) is not always an evenly distribution. There are columns with constant 0, columns with constant 1 and almost everything in between.

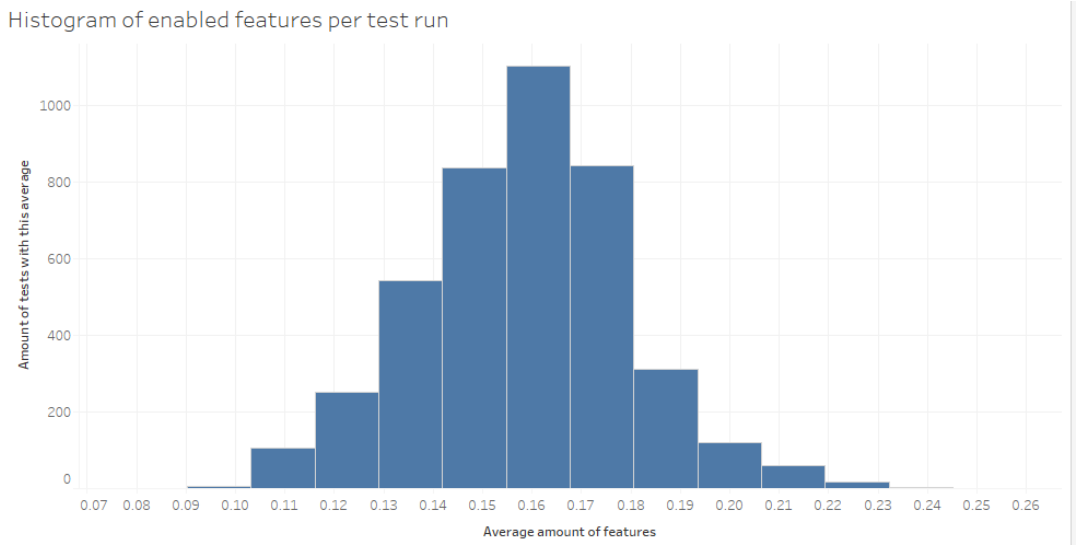


Even the final result (y) was not resembling any normal distribution:



The standard deviation in this case was ~12 where the mean was ~100 meaning the typical deviation is already in a range >10% of the mean. However since this time is feature dependand there should not be given to much weight to this point.

The average amount of "enabled" features (feature equals 1) is appearantly following some kind of normal distribution:



The data itself contains several duplicates where all features are the same (however test time must not be the same). However since the test time can be different only tests with also the same test time should be excluded which resulted in removing 1 row.

Also, some features seem to appear always together (e.g. feature X and feature Y). To reduce the overall complexity only one of these features will be sufficient for further analysis.

This reduced the amount of features by 45.

In addition I checked for features that are always 0 if one other features is 1 und thus found even more binary correlations that can be eliminated for complexity reduction.

This reduced the amount of features by another 8.

Benchmark

As a benchmark I will look at a publicly available solution to the original problem at kaggle based on deep learning[4]. Since decision trees and deep learning are techniques from different fields of machine learning it will be interesting to observe how decision trees will perform in this situation.

The model appears to have performed with an R2 score of ~0.57 which will be my benchmark.

[4] https://www.kaggle.com/umbertogriffo/deep-learning/code

METHODOLOGY

DATA PREPROCESSING

The overall data preprocessing is taken care of in the file DataCleanup.py. Basic data statistic is done in DataStatistic.py.

In a first step I try to identify and remove columns where the value never changes thus are constant and have no impact on the overall result.

The second step consists of checking for duplicate rows. First the columns that are unique for every test have to removed (ID and y) to only reduce the data to the basic features. Following this every duplicate row is removed from the original dataset.

Checking for duplicate columns is a little bit more complex since pandas only supllies a duplicated function on a row basis[5]. Thus the data is transposed first then checked for duplicates and then the original data is cleaned up based on the column id.

Checking for a relation for inverted columns turned out to be a little more complex though.
The idea of this cleanup step is to identify all columns that always appear in a negated combination (e.g. feature X is always 1 if feature Y is 0 and vice versa). I first had to invert all columns containing binary data, then iterate over all columns and find duplicates of the now inverted column in the original dataset. These columns where then marked for deletion.

As the last step I performed one-hot-encoding on the features that had no numeric value assigned to them.

IMPLEMENTATION AND REFINEMENT

My implementation started with the data cleanup as described above. To me personally this produced some obstacles since I have not been that firm in using pandas or numpy since Python is, at the moment, not part of my daily business.

During this phase, after every cleanup step, I created an output file of the newly generated data set to crosscheck my implementation. Also, single outputs using the print command where a huge part in the implementation phase.

Following the data cleanup phase was the actual model implementation.

I started by performing a basic train_test_split on the data with a predefined random_state in order to being able to rep

roduce the outputs.

[5] https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.duplicated.html

Out of curiosity the next step actually was trying to fit a heavily overfitted AdaBoostRegressor (with DecisionTreeRegressor as base estimator) by setting max_depth of the DecisionTreeRegressor to the total number of remaining features as well as the maximum number of estimators for the AdaBoost Regressor. Interestingly enoguh the R2 score after fitting this was ~0.19 on the test data.

The next step included preparations to perform GridSearchCV.
This meant to choose parameters that I wanted to correlate using GridSearch.
My personal pick went for max_depth of the DecisionTreeRegressor base estimator as well the n_estimators of the AdaBoostRegressor.

I picked these features because, in my opinion and experience from previous projects during the nanodegree these appear to have a large impact on the overall outcome.
The parameters themselves I chose in a range of:

- DecisionTreeRegressor (base estimator)
    - Max_depth:
      1 to 15 % of number of features
      The basic idea behind AdaBoost is using many weak learners to become better and faster in the combination of all of them.[6] Thus to me it made no sense to allow full trees of all features in this phase.
- AdaBoostRegressor
    - N_estimators:
      1 to number of available features
      worst case the combination yields a result where all weak learners have depth 1 and represent a single feature. The potential positive outcome of having such a huge range of estimators outweight the potential increase in calculation time for me.

After starting this GridSearch on my local PC (Pentium i5-4300U) it became quickly appearant that the calculation time of this GridSearch would exceed almost everything I would be willing to wait for (potentially days of calculation) so I went for an AWS t3.2xlarge instance (8 vCPUs) for this calculation since GridSearch mainly puts a huge load on the CPU. Because of a bug in joblib used for handling the parallel execution[7] I had to adjust the main execution of my project to be able to use all available cores.

Running GridSearch for these parameters took in total more than 490 minutes before I cancelled it because the total runtime would have exceeded days. (Output of GridSearch can be found in GridSearchBigApproach.txt)

However, while looking through the logfile (GridSearchBigApproach.txt) I found out that even for far fewer max_depth of the DecisionTreeRegressor the r2 score exceded 0.4 easily.

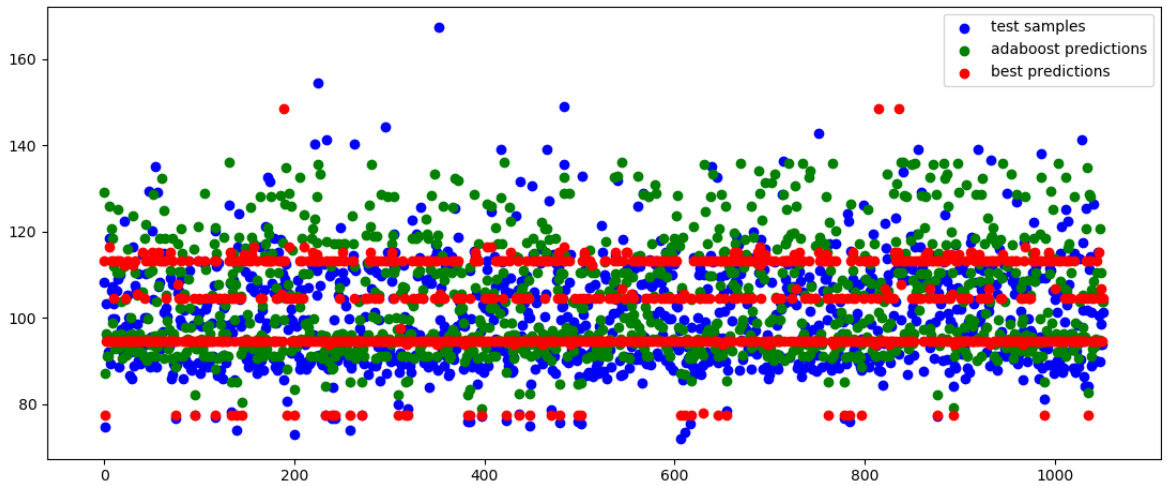
Thus I reduced the GridSearch parameter for base estimator max_depth to 1 to 25 and started the GridSearch again and it finished after only ~398 minutes. This lead to a best estimator with R2 score of ~0.56 on the training set with only max_depth of 4 for the base estimator and 4 estimators in total (see GridSearch25MaxDepth.txt).

[6] https://en.wikipedia.org/wiki/AdaBoost
[7] https://joblib.readthedocs.io/en/latest/parallel.html#old-multiprocessing-backend

Comparing the outcome of the predictions of the fully overfitted AdaBoost from the first step with the best estimator from the GridSearch to the intended outcome we can obtain the R2 scores[8]:

Model	R2 score on test set
Adaboost full	0.195
Best estimator	0.472

Comparing the prediction results of both full adaboost ("adaboost" in the plot) and the best estimator found by GridSearch ("best" in the plot) it becomes appearant that indeed the full depth adaboost heavily overfitted whereas the best estimator appears to more or less resemble several linear regressions:



During the implementation phase I heavily worked with console outputs and storing of intermediate outputs to files. During several executions two things became appearant:

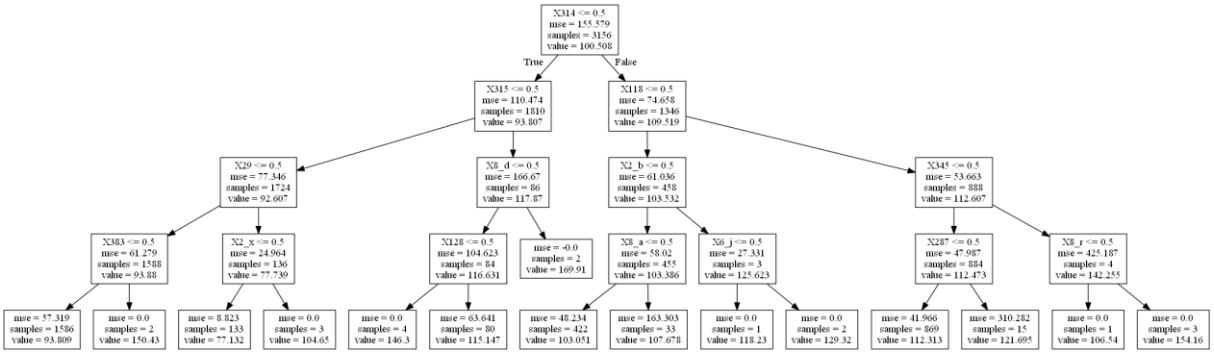
1. I do not always need all the outputs and exports
2. I do not know how long all this actually takes

Thus I implemented a small stopwatch solution to get an idea of the overall complexity of all these calculations. I output this for all relevant steps (e.g. data cleanup, fitting AdaBoost or GridSearch) in order to get an idea of the overall runtime.

In addition I created two boolean flags "verbose" and "export" set globally in the project. "verbose" enables or disables the detailled output of all intermediate steps (like e.g. which columns got exluded in cleanup or how the data looks like after cleanup). "export" enables or disables the storage of intermediate outputs to the filesystem. This includes any plots, graphs etc.. This was also necessary because (again out of curiosity) I was eager to find out how the DecisionTrees looked like. Therefore I found a nice solution online[9] that enabled me to plot the underlying trees using GraphViz dot.

[8] see FullVerboseOutput.txt for all results
[9] https://stackoverflow.com/a/39772170

One example taken from the best fitting estimator can be seen here:



However it was far more interesting to see that every estimator of the best one chosen by GridSearch was mainly focussing on the first non boolean features from the original dataset (X0 to X8) and only in some cases on few other features. (see Result/best0.png to Result/best3.png)

CONCLUSION

In the end the steps performed did not have to differ much from the original proposal.

The first intial try on an overfitting AdaBoost was impressive to me based on the R2 score it achieved in the end since I actually expected this potentially overfitted AdaBoost to not do well at all.

As expected in the proposal already the overall score did not nearly reach the score of the benchmark. However, also as expected, the calculation time for this model was extemely high.

Personally I was still surprised on how long the GridSearch took in the end (even on a very beefy machine on AWS) as well as how good the score of the final result was compared to the overfitted AdaBoost or even the benchmark model, especially because the resulting trees are not very complex.