Université Libre de Bruxelles - Specialized Master in Data Science, Big Data
Academic year 2020-2021

Internship Report

All-relevant-feature-selection methods and the arfs package

Student: Dimitrios Kottoras
Coordinators: Thomas Bury and Thomas Verdebout

Contents

Abstract

This report is focused on the implemented All-relevant-feature-selection methods found in the *arfs* Python package. The package includes three methods, two of which are forks of pre-existing ones and a third one that is completely novel. The purpose of my internship consisted of studying and performing tests on the later, in order to assess its performance under certain conditions that were thought to be potentially harmful. This report is organized as follows: Initially, we present a brief overview of some basic Machine Learning concepts. These include the purpose of ML tasks and their classification, the concepts of fitting, over and under, as well as some models like the linear and the tree-based ones. Then we explain briefly the ideas behind feature selection and All-relevant methods before diving in to the tests performed. Overall, the effects that were studied were correlation effects and noise as well as the scaling of the method and its performance in a high dimensional setting.

This report was performed in the context of my internship in the Machine Learning Group of **Allianz** under the guidance of **Thomas Bury**.

1 Basics of Machine Learning

A common introductory example to Machine Learning is the spam/no spam email problem [1]. The situation is the following: We imagine having to write a program that detects whether or not an email is a spam. To this end, one could study a lot of emails, check which ones are tagged as spam, study the patterns and create the rules that define the "spam" according to the patterns. However, such a program could potentially be very difficult to maintain given the fact that the patterns/rules may evolve over time or simply because the program itself is too complex (for instance, it could have a huge number of rules). A solution to this complexity is to write an algorithm that takes some input emails and spams, and studies the patterns itself in order to produce a "model" that, given a new email can give you an **estimation** of whether it is spam or not. Given that the algorithm does not depend on any data, it can be used on any collection of emails and spams! This would be a Machine Learning algorithm. The process of feeding data to the algorithm is usually referred to as "learning" or "fitting" the model. The data that is used to make the fit is called a "training set". One then has to assess the quality of the model by making predictions with another set of data, which is preferably statistically uncorrelated to the training set. The new data set is called a "test set".

Machine Learning is basically equivalent to Statistical Learning, although it has perhaps a more "coding" related connotation. In mathematical terms, we find ourselves studying a phenomenon that is described by a **response** variable or Y, and some "explanatory" variables or **predictors** $X_1, \ldots, X_p$. Furthermore, we assume that the relationship between the two is of the form [2]:

$$Y = f(X) + \epsilon \tag{1}$$

where ϵ is known as the **error** or **noise** term and is considered as independent from the predictors. In some cases we may know the form of the relationship f but in general we do not know anything about it. In essence, the task of Statistical/Machine Learning is to estimate this relationship. To this end, there are many methods/models/estimators but in general there is no one method that is to be preferred over all possible situations. **There is no free lunch is statistics.**

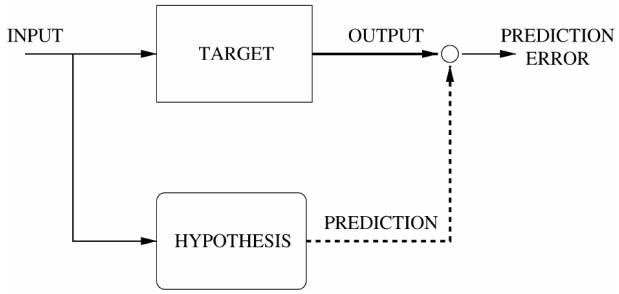


Figure 1: The Machine Learning procedure. The input X is passed through a target operation which gives us the output Y. On the basis of the same input an hypothesis (model) is constructed to give a prediction of the result of the target operation.

In this context, there are two main estimation tasks[1]: **regression** and **classification**. In regression the response variable is continuous and the model is usually assessed with the so called Mean Error Rate:

$$\text{Ave}(\hat{f}(x_0) - f(x_0))^2 \tag{2}$$

where the hat denotes our estimation of the output and the average is taken over the test set. In classification the response can only take a small set of values, also known as levels. The accuracy measure is similar to the one above:

$$\text{Ave}(I(y_0 \neq \hat{y_0})) \tag{3}$$

where I denotes the indicator function.[2]

As already stated, the model is not assessed on the same training set because this would overestimate its accuracy. In general, a big issue in ML is that of **overfitting**, where an estimator performs exceptionally well on the training set (and thus has a low prediction error when applied on the training data) but behaves poorly with new sets (and thus the error defined above is large). This is usually due to the model being very complex/flexible. On the other end, we can also have the so called **underfitting**, where the model is so simple that is inadequate to describe the response variable. The problem is therefore that of finding a good balance in terms of this complexity and the corresponding performance. This is usually done through trial and error.

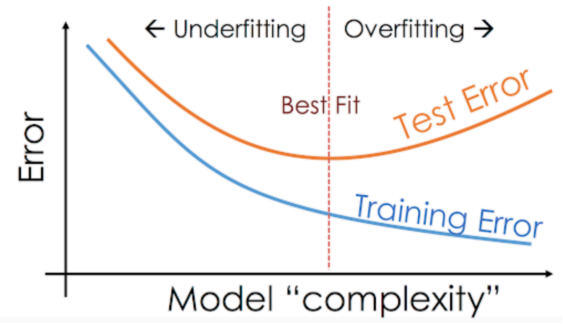


Figure 2: Overfitting and undefitting. The higher the complexity of the model, the better it fits the data. However, crossing a threshold of complexity leads to an increase in the test error.

Another important issue in ML applications arises when the number of features is too large[3] and/or when some of the features are irrelevant. This can have a negative impact on the accuracy and interpretability of the model[3].[4] In most of these applications a pre-processing step is therefore required which entails a feature selection procedure. This topic motivates this work, which is basically an investigation of some feature-selection methods found in the **arfs** package.

[1] See [2] for more details.
[2] The indicator function here is such that $I(TRUE) = 1$, otherwise it is nul.
[3] This is known as the *curse of dimensionality*.
[4] This relates to our ability to obtain an intuitive understanding of the model and its predictions.

2 Some models/estimators

Among the large number of models found in Machine Learning there are some that are used/referenced in the two last chapters, and so a brief overview seems appropriate. We begin with the most famous and oldest model in Statistical Learning: the linear model.

2.1 Linear model

The linear model makes the simple assumption that the dependence between the target variable and the regressors is linear. In particular, the simplest case is the uni-dimensional one:

$$f(X) = \beta_0 + \beta_1 X$$

One can easily generalize the above with p predictors:

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

In both cases, the task of learning reduces to estimating the β coefficients on the basis of the available data set. In practice this is done with the well known *least-squares* procedure that basically returns the coefficients that minimize the error between the observed output and the corresponding regression function.

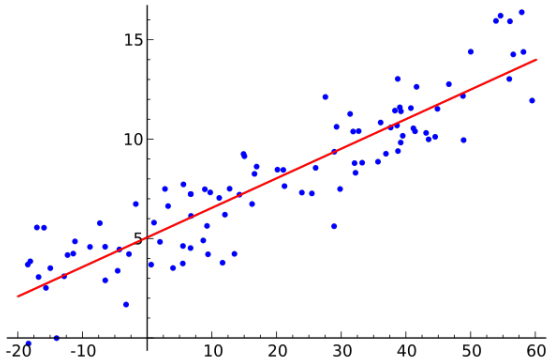


Figure 3: A simple linear regression model. One can easily see that the dependency between the x variable and the y axis is linear. However the points are distributed in a uniform fashion around the line defined by the real coefficients due to the presence of the noise.

If the assumptions hold, the linear model has very nice properties. One can show for instance, that the least squares estimates are unbiased, ot that for a normally distributed error term, the coefficients have also a normal distribution. However, in real life situations the linearity assumption is inadequate and the model may fail to provide us with accurate predictions.

2.2 Trees and Forests

Some of the feature selection methods discussed in the next chapter make use of **random forests**[5] which are basically collections of **classification or regression trees**.

[5] See [4].

Trees are very important models in Machine Learning. One can think of them as machines that divide the input space into separate regions, and then make predictions on new input based on the position of the input in the input space.

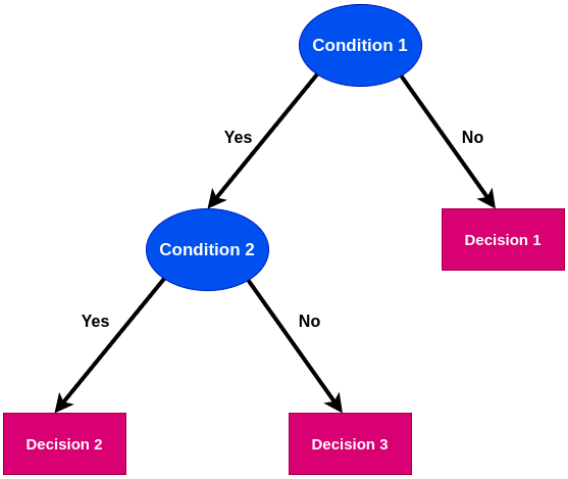


Figure 4: A simple tree.

The trees are made up of *nodes* and *leaves*. The node is a unit where a decision is made that leads to a child node. The leaf is the "last" child in a series of nodes and it is associated with a certain prediction/estimation.

Since trees are models, they need to be trained before they can provide us with predictions. This basically entails the partitioning of the input space and the determination of the structure of the tree itself. There are several algorithms that do this.

Trees are quite flexible but can have a high variance. This can very easily lead to overfitting. However, there are several techniques that remedy this, such as pruning the trees after growing them. This basically uses brute force to restrict the tree from following the noise patterns and thus generalize better to new data.

Another way of reducing variance is to consider collections of unpruned trees like random forests. These models are based on the idea that the combination of uncorrelated and unbiased estimators of the same quantity has a lower variance than the estimators themselves. An illustration of this effect can be seen in the following way: Consider two estimators $\hat{\theta}_1$ and $\hat{\theta}_2$ such that:

$$E[\hat{\theta}_1] = E[\hat{\theta}_2] = \theta, \tag{4}$$

$$var(\hat{\theta}_1) = \sigma^2, \tag{5}$$

$$var(\hat{\theta}_2) = \sigma^2 \tag{6}$$

Now consider the combined estimator: $\hat{\theta} = \frac{\hat{\theta}_1 + \hat{\theta}_2}{2}$. This estimator is unbiased:

$$E[\hat{\theta}] = E[\hat{\theta}_1]/2 + E[\hat{\theta}_2]/2 = \theta \tag{7}$$

and its variance is:

$$var(\hat{\theta}) = var(\frac{\hat{\theta}_1 + \hat{\theta}_2}{2}) = \frac{1}{4}(var(\hat{\theta}_1) + var(\hat{\theta}_2)) = \frac{\sigma^2}{2} \tag{8}$$

The same principle applies to trees, which are themselves statistical estimators.

Consequently, the forest is trained by growing trees with as little correlation as possible. This is achieved in a two step procedure where:

1. We generate "different" data sets from the original training set using resampling with replacement. Each of this sets is used to train the individual trees.
2. During the tree growing, we consider a random selection of features at each step, from which a node can be created (split).

Finally, to make a prediction one averages over the results of the trees (regression) or uses a majority vote (classification).

It is worth noting that the arfs package relies mostly on **Gradient Boosting Machines** (GBM) and more precisely on the *lightGBM* flavor. Like random forests, these machines are ensemble methods: they combine the results of individual trees. Specifically, this model combines weak learners in an iterative fashion in order to achieve a strong learner.[5]

3 Feature selection

The basic assumption one makes when using a feature selection method is that the data contains variables that are either *redundant* or *irrelevant* and whose removal from the learning process has a small loss of information. In addition, it also has a few benefits:

1. The simplest advantage is simply the reduction of training time since this is usually directly related to the size of the data set.
2. As we have already stated, by limiting the number of predictors, we essentially limit the complexity of the model and therefore we improve its interpretability, or in other words, **the degree to which a human being can understand the causes of the model's decisions**.
3. A lot of traditional methods like the linear model that we discussed in the previous chapter have been designed for tasks where the input dimension is very small compared to the size of the data set and most variables are informative. Therefore, in the presence of a huge number of unnecessary predictors the performance of these methods can be greatly degraded. Inversely, the removal of these variables can improve the model's prediction accuracy.
4. These methods also try to avoid the so called *curse of dimensionality*. In learning tasks, this refers to the fact that the sparsity of the data increases exponentially with its dimension. This can have very negative results on local learning algorithms like *k-means*.

In general, feature-selection algorithms fall into two main categories: **Minimal optimal** and **All-relevant** methods[6]. The first category seeks a small subset of features that gives us the best possible predictive power. On the other hand, *All-relevant* select features that individually can have any predictive power at all. Basically what this means is that these methods select the so-called relevant features but they also return redundant features: predictors that are either highly correlated with the relevant ones or they can be obtained through a linear combination of these.

Of particular interest to this report are the *All relevant feature selection methods*, some of which are implemented in the python package *arfs*. These are relatively novel procedures with a premature theoretical backing. Hence the purpose of this work, namely to make mostly a qualitative study of some possibly harmful conditions to these methods.

As stated above, the ARFS procedures attempt to perform an inclusion of all the relevant variables present in the problem. Here, by relevant we mean any variable that may carry some useful information about the target. Furthermore, the methods are model agnostic, in the sense that they are utilized as a pre-processing step without taking into account the model that will be used in the learning/prediction step. We note that concrete definitions and categorizations of relevancy exist in the literature, but an intuitive understanding of this notion will be sufficient for our purposes. In practice, the *arfs* methods use an **importance score**[7] that estimates the contribution of a given variable to the target. By imposing a

[6] See [6].

"cutoff" in the score, the method then classifies variables as relevant or irrelevant. In that sense, the All-relevant problem is a classification problem.

One of the most famous ARFS algorithms is the **Boruta**[7], developed in the ICM, University of Warsaw, back in 2010. One of its advantages is that the cutoff described earlier is not set by the user but it is embedded in the method itself. The algorithm works in the following way:

1. Permuting the given predictors, it generates new variables, called **shadow features** which are added in the data set as extra but 'false' predictors. This essentially breaks any connection of the initial variables with the target.

2. A random forest is fitted on this new data set in order to assess the importance of each of the variables using a pre-defined criteria. Among the shadow features, the maximum importance is set as the cutoff.

3. The importance of the original features is then compared with the cutoff. If it is bigger it is tagged as relevant. Otherwise it is tagged as irrelevant.

4. Finally, the procedure is repeated so that a confidence interval of 'relevancy' can be constructed using a binomial distribution.

One more all-relevant algorithm/implementation related to the arfs is the **BoostARoota** method[9], which is very similar to the Boruta. It is different in the following ways:

1. It involves a pre-processing step of encoding categorical variables using the One-Hot method.

2. Like Boruta, it creates shuffled copies of the predictors and adds them to the data set. However, instead of using a random forest to assess the importance, it uses the XGBoost estimation procedure which is run a certain number of times on the set. The final importance of the variables is then the average over the importance scores obtained in each iteration. Note, that this assumes that averaging the scores gives a better approximation of the real importance.

3. The cutoff is defined as the average of these scores among the shadow features divided by a parameter set by the user (ususally 4). This is in contrast to the Boruta where the threshold does not use any input from the user.

4. The original features with a score lower than the cutoff are rejected.

5. Finally, the process is repeated until no more than 10 percent of the features have been removed.

3.1 SHAP

As we've already mentioned, the all relevant methods need an importance measure in order to assess the relevancy of a variable. There are many measures that are applicable to the

[7]For a visual explanation, see [8].

machine learning task, among which the so called **permutation importance** and **SHAP importance**[10]. Both are available in the *arfs* package and the user can choose the one that fits best to their needs. The main focus of this report is the novel method **GrootCV** whose default importance measure is the SHAP.

SHAP importance is based on the idea of **Shapley values** whose origin lies in the mathematical field of game theory. In the ML context the Shapley values gives us the contribution of the value of a feature to the prediction associated with a particular instance. In practice this is calculated with the help of *coalitions*. More specifically, in order to calculate the Shapley value of feature j the procedure is as follows: First of all, a model is trained using the available data set. Then for a specific instance in the set, we form a coalition of values that contains the actual value of some features (including j) and values of the other features that are drawn randomly from within the data set. Then a prediction is made with the coalition values. After this we make another prediction using the same coalition but with a randomly chosen value for j. This prediction is subtracted from the initial one. We perform the same procedure for all possible coalitions and average the results. We note this value by ϕ_j. One can show, that these values are the *linear attributions* of each feature to the target value. In other words, for an instance i we have:

$$y_i = \sum_{j=1}^{n} \phi_j^{(i)} x_j \tag{9}$$

The SHAP importance is defined as the sum of absolute Shapley values per feature across the data:

$$I_i = \sum_{j=1}^{n} |\phi_j^{(i)}| \tag{10}$$

This simply means that features with large absolute Shapley values are important to the target.

3.2 arfs and problems

The *arfs* is a python package built by my coordinator Thomas Bury. It can be installed using the *pip* package manager and it is *sklearn* compatible. It provides three all relevant methods, two of which are forks of pre-existing ones. The third one is a novel method and it is the one that we focus on in this report.

The forks are the following:

1. Leshy, evolved from Boruta. It performs automatically an encoding of categorical features. It uses the lightGBM as an estimator/model which is much faster than the random forest. It supports native, SHAP and permutation feature importance.
2. BoostAGroota, evolved from BoostARoota. It uses lightGBM instead of XGBoost and it replaces the native feature importance with the SHAP feature importance

The novel method is given the name **GrootCV**. It has the following characteristics:

1. It calculates the feature importance using cross-validation. This basically means that the model is trained on different subsets (folds) of the training set, thus allowing for multiple values of importance.
2. This importance is by default the SHAP importance.
3. The cutoff is taken to be the max among shadow variables of the median importance taken over folds.
4. It is not based on a given percentage of cols needed to be deleted.
5. It provides a *Plot* method of the variable importance.

In what follows, we perform some simple tests on the method above. Since there is not yet enough theoretical backing to the method, we approach the subject in an experimental fashion. Namely, we create simulated data sets, that consist of predictors and a target, of whose relationship we can control. Thus we can generate relevant and irrelevant variables, and thus observe whenever the method fails to return all the relevant ones.

The goal is to make some qualitative observations on the possible harmful effects that certain conditions may cause. In particular, these may be **correlation** effects, noise terms and other conditions. Simply put, noise terms should degrade the performance of the method when important, since the all-relevant methods train a model in order to assess relevancy. When there is much noise, the relationship between predictor and target becomes unclear and the model that is trained is also inadequate to explain the true form of the regression function. Therefore we expect the relevancy to be assessed poorly when the noise is too great. On the other hand, correlation effects may be more subtle. Since SHAP is basically a linear attribution, the method could potentially split in equal amounts the importance of two collinear variables that contribute in different ways to the target. If this importance is great then there is no problem with this scenario. The problem arises when the importance of the relevant variables is lowered to such an extent that it falls below the desired threshold. Thus the relevant are assessed as irrelevant. When the number of collinear variables is large, this scenario may be easily realized.

In summary, the tests that we perform follow this structure:

1. First we test the scaling of the method.
2. We make some correlation tests. We check for the stability of the method and its performance for a range of correlation values. These tests are performed on a few different cases. This will be explained in detail in the following section.
3. We perform tests with noise terms. The performance is assessed for larger and larger standard deviation of noise.

Since the method is experimental, we can not consider the whole set of possible causes and effects, neither the whole set of possible relationships (between target and predictor). Hence, any attempt to make a quantitative statement that applies to all situations is not possible. The settings that we used are explained in the next section.

4 tests and results

As we've already mentioned, the tests were performed explicitly on the GrootCV method that uses by default the SHAP importance. We can split the tests into three categories: Scaling, Correlation and Noise. We note that the tests were performed in the python language and can be found in detail in the appendix along with the appropriate explanations and output. The machine used to make the tests has a 6-core processor with a base clock speed of 2.6 GHz, an 8Gb RAM and a 250 Gb SSD.

4.1 Scaling

The scaling of the method is very important when we are interested in large data sets. If the computation time is too large for huge sets an algorithm can be useless in practice. To get an idea of the scaling of GrootCV we applied it on an artificial data set with 11 predictors and a uni-dimensional target. Six of the predictors were taken to be relevant and 6 irrelevant. The relevant variables have a linear relationship with the target and noise terms are absent. We iterated over sets with size from 100 to 10000 with a step of 1000 and estimated the time needed to complete the computation. The result is summarized in this plot:

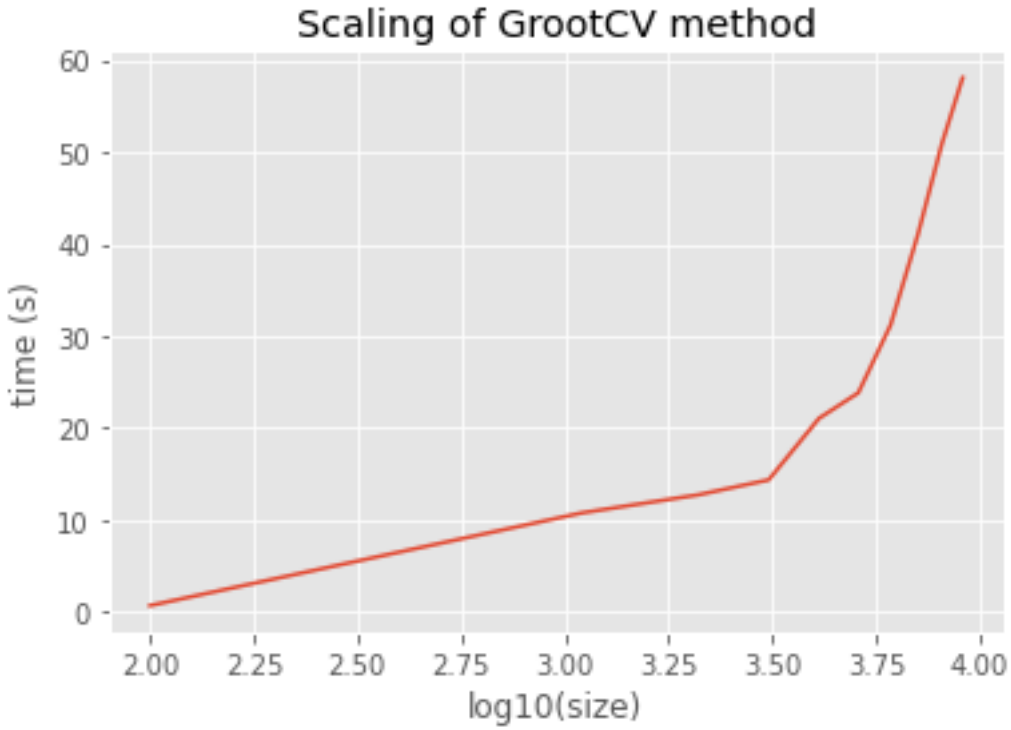


Figure 5: The time is shown in seconds while the size (number of rows) is given in log base 10.

The method seems to have an almost exponential growth. In simple terms, for very big data sets, this will lead to a large computation time. Above, it takes almost minute to compute for a data set with 10000 rows. This effect was also observed in practice during this report.

Some of the tests took nearly 6 hours to complete.[8]

4.2 Correlation

We have performed tests to check whether correlation can actually be harmful to arfs. These can be split into the fixed number of variables tests and the non-fixed ones. In the first category we have a data frame with 11 variables, six of which are relevant. The data frame was generated in a few different cases of correlation and target dependence. On the other hand, in the second category, we have one case (of correlation and dependence) but we have varied the number of correlated variables.

4.2.1 Fixed number of variables

We fixed the size of the set to 10000 rows. We applied the method in a few different cases of correlation between the relevant variables. These are the following:

1. A correlated pair. The rest are independent and uncorrelated.
2. Two correlated pairs.
3. Three correlated pairs.
4. Three correlated variables.
5. Three and three correlated.
6. Four correlated.
7. Five correlated.
8. Six correlated.

We also consider three relationships between target and predictors. These are:

1. Linear (the one used above):

$$y = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + (1 - w_5) x_6 \tag{11}$$

2. Weakly non-linear:

$$y = w_1 abs(\sqrt{x_1}) + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + (1 - w_5) x_6 \tag{12}$$

3. Strongly non-linear:

$$y = w_1 sin(x_1) + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + (1 - w_5) x_6 \tag{13}$$

The variables are generated by sampling from a known distribution (mostly a normal one) with a unit variance. This unity was chosen in order to avoid any biased assessment of

[8] Of course, hardware specifications are important.

predictors with large standard deviation. The correlation between them was achieved by sampling from a multidimensional normal distribution with a known covariance matrix. The correlation strength parameter enters as the non diagonal element of the covariance. This matrix is constructed in the following way: First we define:

$$m = \begin{pmatrix} 1 & cor \dots & cor \\ \vdots & \ddots & \\ cor & \dots & 1 \end{pmatrix} \tag{14}$$

Then we define the covariance matrix:

$$Cov = m^T * m \tag{15}$$

This ensures that we get a semi-positive definite symmetric matrix for all possible values of the cor parameter.[9]

First of all, we have chosen the values of the coefficients to be:

$$w_1 = 0.4 \tag{16}$$
$$w_2 = 0.2 \tag{17}$$
$$w_3 = 0.1 \tag{18}$$
$$w_4 = 0.1 \tag{19}$$
$$w_5 = 0.05 \tag{20}$$
$$w_6 = 0.05 \tag{21}$$

They add up to 1 for convenience.

We applied the method once across all the cases and models outlined above. We have obtained the output for these values of the cor parameter: 0, 0.2, 0.5, 0.8 and 1. This is presented in the form of boxplots like the one below (case 1, linear model, cor = 0):

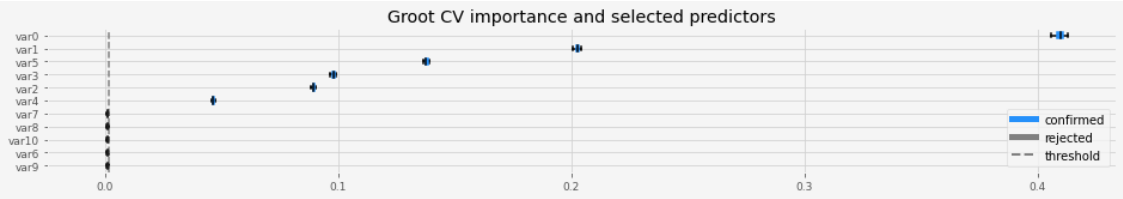


Figure 6: Importance score across variables. Case 1, linear model.

The plot is rather straightforward. The different importance scores are plotted for all the variables. The threshold is found in the zero point of the x-axis. As we can see the relevant variables lie above the cutoff and they are marked with blue colors. The irrelevant ones have zero importance. Note also that the median of these scores is near the coefficients defined above. The rest of the boxplots can be found in the appendix. However the results are quite similar: First we can see that the method works pretty well. The irrelevant variables

[9]One can check that this resulting non diagonal elements are equivalent and between -1 and 1.

are completely discarded and all the relevant ones are returned. Second, the median of the importances is very close to the actual weight of the variable in the linear model. This is no coincidence, since the SHAP values are basically linear contributions of the predictors to the target and hence, for a linear model they should correspond to the coefficients. Of course this should not apply in the other models that we use. However one can check that even in non linear cases, the method does a very good job at ranking the variables correctly when the correlation strength is not very close to 1. Furthermore, the split of the importance discussed above is also visible here, where the median of most important variable decreases while the median of the correlated variable increases.

In this particular scenario, this does not harm the method, but given the right circumstances it could lead to problems. Finally, as one can see in the boxplots, the spread of the importance among the correlated pairs increases with the correlation strength. For example, under the same conditions but with a higher correlation parameter, the resulting boxplot is:

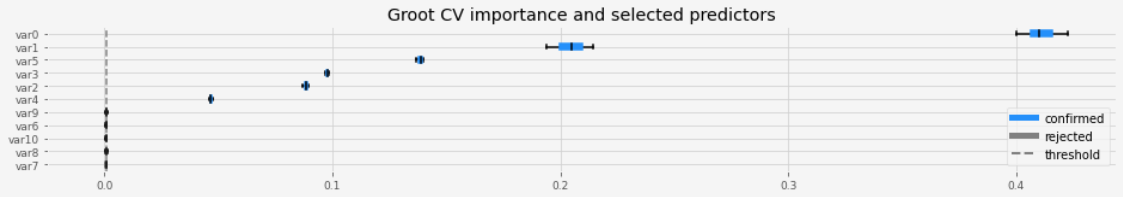
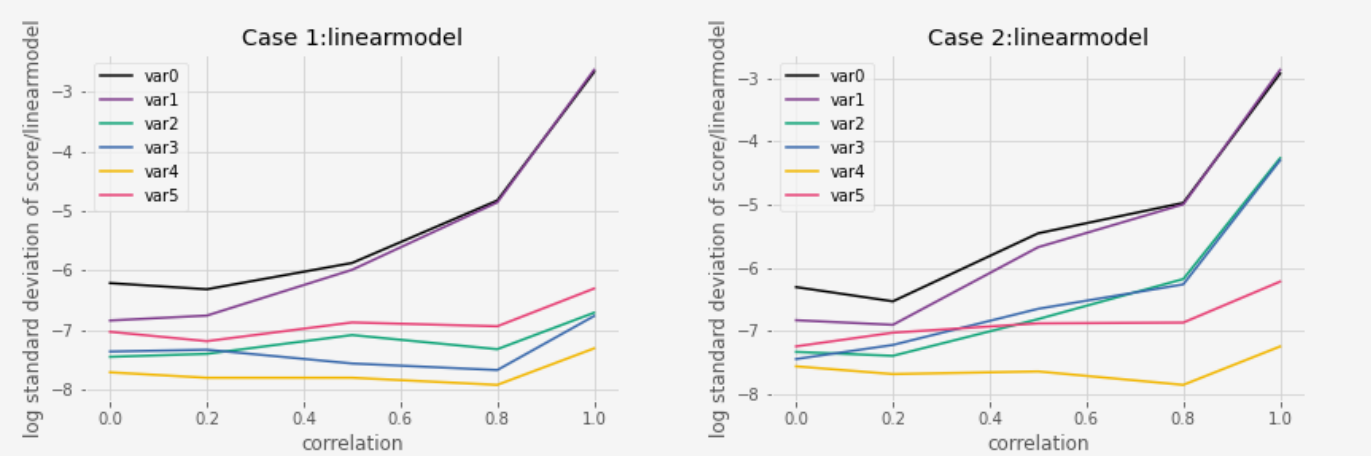


Figure 7: Higher correlation.

This can also be seen in the figures below (linear model, all cases):



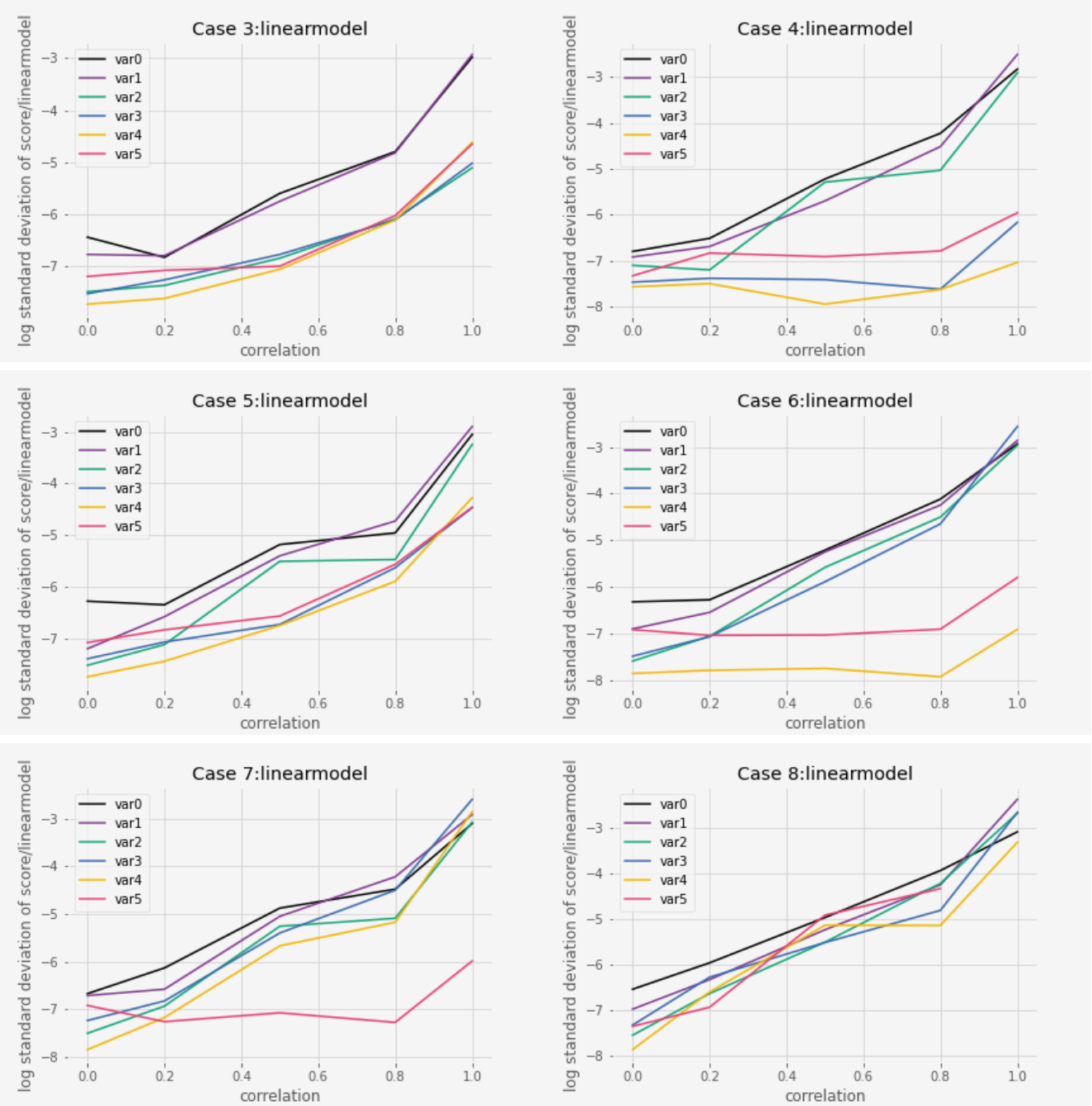


Figure 8: the correlation parameter is plotted against the logarithm of the standard deviation of the score for each variable.

As we can see, it is only for correlated variables that there is an increase in the spread. The rate of the increase is also pretty much equivalent between them.

In the same context, we have also performed some stability tests. Namely, we have ran the method a couple of times just to obtain the percentage of times where the method fails. To illustrate in pseudocode, let S be the set of relevant variables and O be the output of the

GrootCV method (variables chosen):

Algorithm 1: Stability test

$c \leftarrow 0$;
for $i \leftarrow 1$ **to** N **do**
  generate data set X,y;
  apply GrootCV on X,y;
  if $S \in O$ **then**
    $c \leftarrow c + 1$
  end
end
return $1 - c/N$

In this way, we count the number of times that the method discards at least one relevant variable. We repeat this procedure for different values of the cor.

It is worth noting that this test is the longest to run. The raw output of the corresponding code can be found in my Github page[11]. However the results are not very interesting. Overall the method is pretty stable with 0 error rate in all these cases except for the last one (6 correlated) with cor set to 1. Under these conditions the error rate jumps to 0.8.

Similar tests were also performed with different values for the coefficients with equivalent results.

4.2.2 More variables

As stated above, the method is very stable under the previous conditions. But one can imagine that for a sufficient number of correlated columns, the importance of some variables may decrease and fall below the cutoff. These variables will then be discredited.

In order to achieve this in practice we wrote a function that generates data sets with a varying number of columns, half of which are relevant and correlated predictors. These are linearly related to the target with equal coefficients (adding up to 1). We chose a high correlation strength ($cor = 0.8$) and we set the number of rows to 1000 (for a large number of columns and rows the computation time became too large in the machine used). Once more, we have tested the method by running it a couple of times ($n = 5$) for different column numbers. We have extracted the error rate and the number of relevant variables chosen on average.

```
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.42it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.43it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.45it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.45it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.44it/s]

---------- number of columns =  200 -------------
number of times that all relevant features were chosen:  5 out of  5
error rate =  0.0
```

```
average number of relavant variables:
100.0

Repeated k-fold: 100%|      | 25/25 [00:38<00:00,  1.56s/it]
Repeated k-fold: 100%|      | 25/25 [00:39<00:00,  1.56s/it]
Repeated k-fold: 100%|      | 25/25 [00:38<00:00,  1.55s/it]
Repeated k-fold: 100%|      | 25/25 [00:39<00:00,  1.57s/it]
Repeated k-fold: 100%|      | 25/25 [00:39<00:00,  1.57s/it]

----------- number of columns =  400 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
average number of relavant variables:
120.0

Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.94s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.95s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.93s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.94s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.94s/it]

----------- number of columns =  800 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
average number of relavant variables:
74.0

Repeated k-fold: 100%|      | 25/25 [02:24<00:00,  5.76s/it]
Repeated k-fold: 100%|      | 25/25 [02:24<00:00,  5.79s/it]
Repeated k-fold: 100%|      | 25/25 [02:23<00:00,  5.76s/it]
Repeated k-fold: 100%|      | 25/25 [02:24<00:00,  5.78s/it]
Repeated k-fold: 100%|      | 25/25 [02:23<00:00,  5.75s/it]

----------- number of columns =  1000 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
average number of relavant variables:
61.0

Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.32s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.31s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.32s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.31s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.31s/it]

----------- number of columns =  1200 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
```

```
average number of relavant variables:
59.0
```

As we can see, when the number of variables is more than half the number of rows the error rate jumps to 1. More specifically, the percentage of average relevant variables decreases as we increase the number of columns in our data set. This could be due to the importance split that we mentioned earlier, but it could also be due to a degrading of the model's (trained during the method) performance in a high dimensional setting.

4.3 Noise

Until now, we have isolated our models from any noise terms. We wanted to test the effects of correlation without additional influences. However, noise terms exist in every realistic scenario and its influence can have a huge impact on our models. To test this influence, we added a noise term to the linear model (with 6 relevant + 5 irrelevant) and kept all the relevant variables uncorrelated. The coefficients are the ones used in the previous section. The noise term follows a normal distribution and has a varying magnitude of standard deviation.

We observed the performance of the method for different values of the variance of the noise. We repeat that we have set the variance of all simulated variables to 1 to make the comparison easier and to avoid any bias of the method towards variables with a high variance.

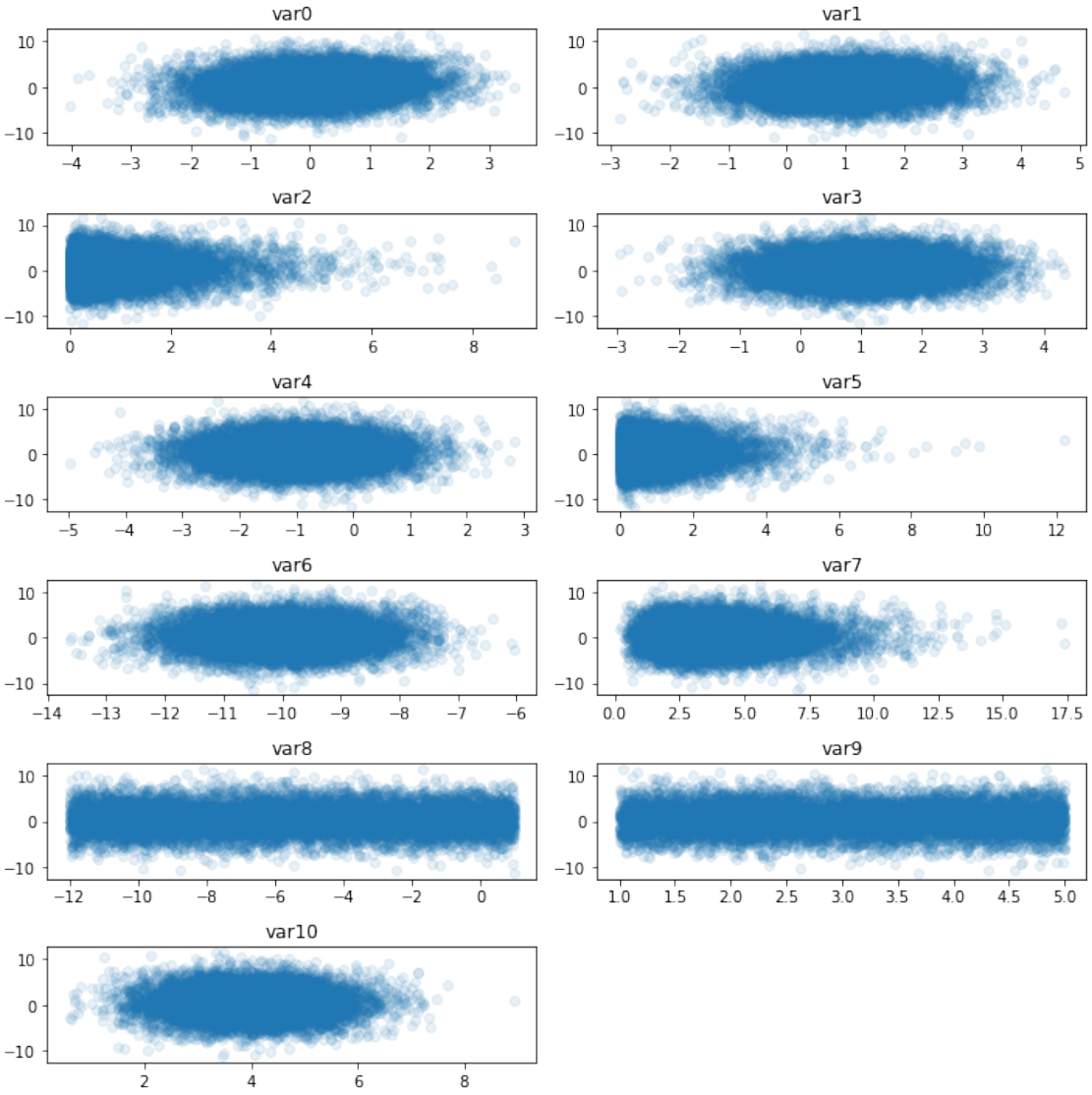


Figure 9: Target (y axis) against predictors (x axis) in the presense of noise.

Above we have plotted the predictors against the target. We can see that the noise disturbs the true relationship between them. Since a model needs to be trained in order to assess the importance of each variable, this effect can be detrimental. To test this we have made some stability checks. Here, instead of iterating over the correlation strength, we iterate over some values of the standard deviation (of noise).

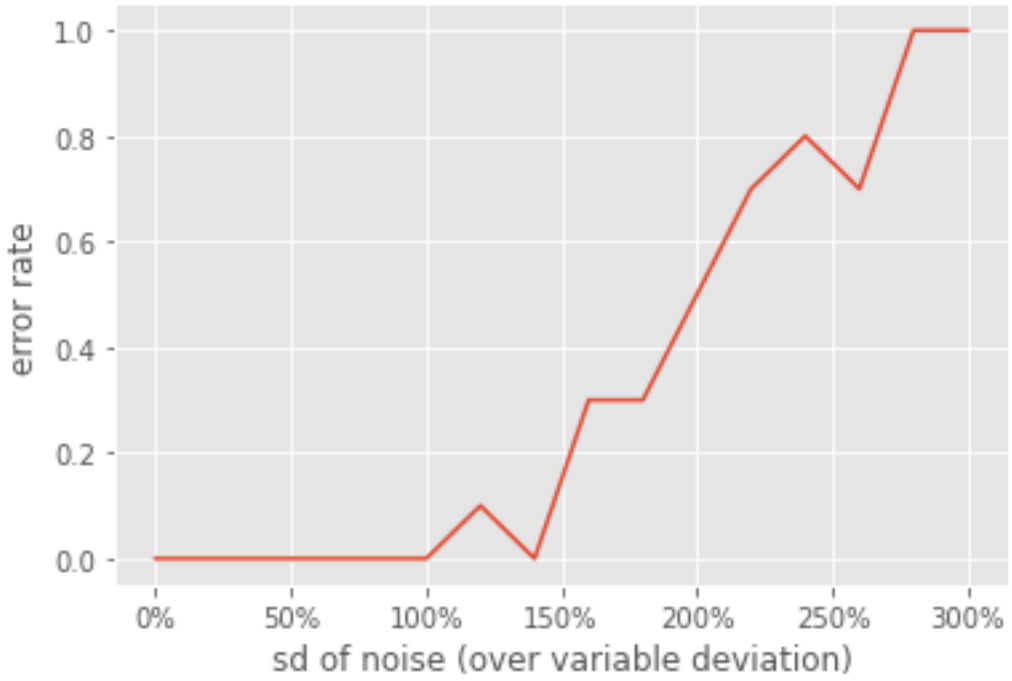


Figure 10: Error rate against standard deviation of noise.

Above we have plotted the error rate of the method for different values of standard deviation. This is given as a percentage (i.e 100% is 1 or the variance of the predictors).

It is clear that the method behaves poorly when the magnitude of the noise variance exceeds the variance of the variables. The method becomes rather unstable (> 0.5) after the 200% threshold and reaches 1 at 300%.

5 Conclusion and Remarks

In this report we have focused on the *GrootCV* method found in the *arfs* package. This is an all-relevant-feature-selection method that uses cross-validation and SHAP importance in order to assess the relevant variables. The purpose of this work consisted in performing certain tests and checks over the possibly harmful conditions of correlation and noise terms. The tests were realized in the python programming language, over a set of different settings and regression functions in order to make some qualitative statements about the method. Overall the statements that we can make are the following:

1. When the number of columns in our data set is sufficiently small compared to the size of the set, the method seems to be rather stable, even when the relevant columns are correlated. However, the stength of correlation has an effect over the spread of the importance score of the correlated variables. These scores increase with similar rates as the stength increases. The problems arise when the number of columns increases as well (compared to the data set size). For sufficiently large column numbers the error rate (defined in the previous section) jumps to 1 and the number of relevant variables chosen by the method on average decreases. This can lead to poor selection in high dimensional settings.

2. Adding noise terms to the model can also have a detrimental effect on the method's performance. In particular, we have observed that the magnitude of the standard deviation plays an important role: When it exceeds the deviation of the predictors the method becomes unstable rather fast. For very large error rates the method is completely unreliable. This is simply due to the nature of the all-relevant method: training a model to assess importance.

These observations have just reinforced our initial expectations. However it is worth noting that more tests can be made in order to make more accurate statements and draw better conclusions. We note that due to lack of time and the necessary computing power, we have limit ourselves in rather small data sets with a number of rows reaching a maximum size of 10.000. In general, the method should work better with larger sets and in realistic cases these sets can have millions of rows. In addition, it is worth testing out the rest of the methods as well, and making a systematic comparison with the GrootCV.

A code

A.1 Imports

We begin by importing all the necessary functions and classes. For convenience, I have put all of them in the tools.py script which can be found in the appendix with much detail.

[1]:
```
from tools import *
```

A.2 Scaling

Now that we have our tools, we want to investigate the scaling of the *GrootCV* method. By scaling we mean the computation time over different sizes of data sets (number of rows). To this end, we first generate the data set using the *generate_corr_regr* function. This returns a data frame X that contains 6 relevant and 5 irrelevant variables with a size controlled by the *size* parameter. The *cor* parameter controls the strength of the correlation between relevant variables, here set to 0. The *target* here is the relationship between predictors and target. For simplicity we are using a linear relationship. The weights are the coefficients of this relationship and they add up to 1. Note that we are not adding any noise terms yet.

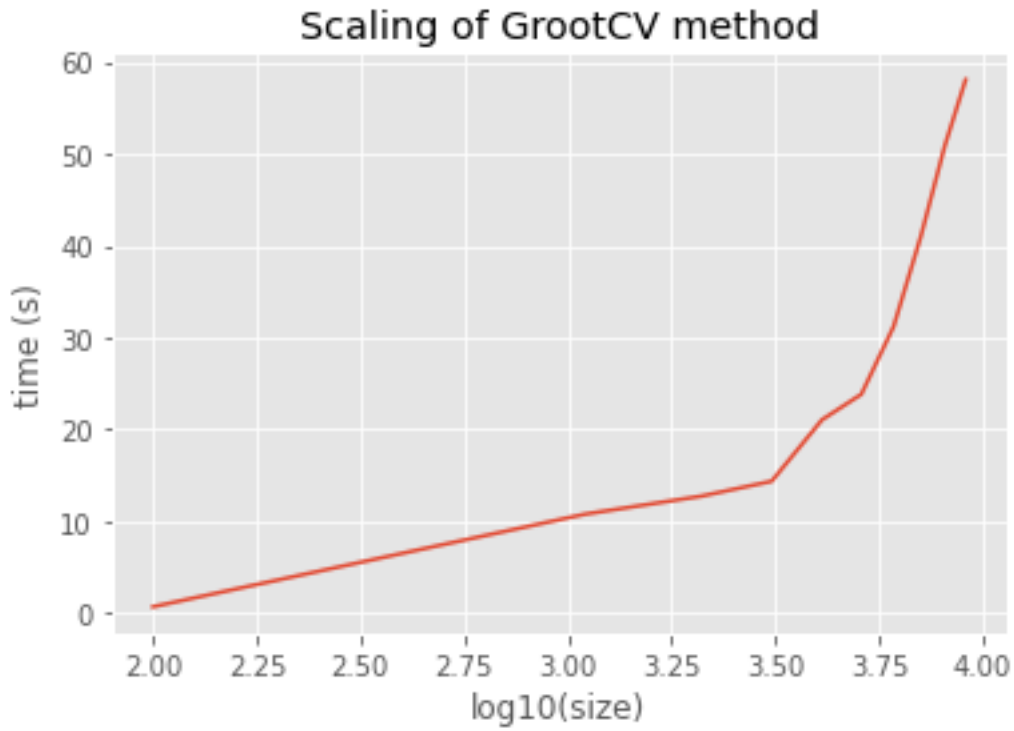
[2]:
```
%%capture
#weights
w1 = 0.4
w2 = 0.2
w3 = 0.1
w4 = 0.1
w5 = 0.05
# initialize object
feat_selector = arfsgroot.GrootCV(objective='rmse', cutoff = 1,␣
 ↪n_folds=5, n_iter=5)
cor = 0
# storage of times
times = []
# loop over sizes
for size in range(100,10000,1000):
    # generate data set
    X,y =␣
 ↪generate_corr_dataset_regr(size=size,cor=cor,case=case,target='linear',w1=w1,w2=w2,w3
    # number of iternal loops
    loop = 10
    # time taken to fit with Groot CV
    result = timeit.timeit('feat_selector.fit(X, y,␣
 ↪sample_weight=None)',globals=globals(), number=loop)
    #store it in times
    times.append(result/loop)
```

The arfs method is called as *arfs.GrootCV* and it is applied on the data set with the *object.fit* method. The API is very similar to the one used by Sklearn.

We plot the results using a logarithmic scale for the sizes.

[3]:
```
x = [math.log10(i) for i in range(100,10000,1000)]
plt.style.use(style='ggplot')
plt.plot(x,times)
plt.title("Scaling of GrootCV method")
plt.xlabel("log10(size)")
plt.ylabel("time (s)")
```

[3]:
```
Text(0, 0.5, 'time (s)')
```



A.3 Correlation

We would like now to turn our attention to those harmful effects mentioned in the beginning. One expects one of these effects to be the correlation between relevant variables. In broad terms, this is due to the fact that SHAP importance can be split between correlated variables, and thus potentially lowered to such a degree that the variables are discarded by the method.

[4]:
```
size = 10_000
C = [0,0.2,0.5,0.8,1]
```

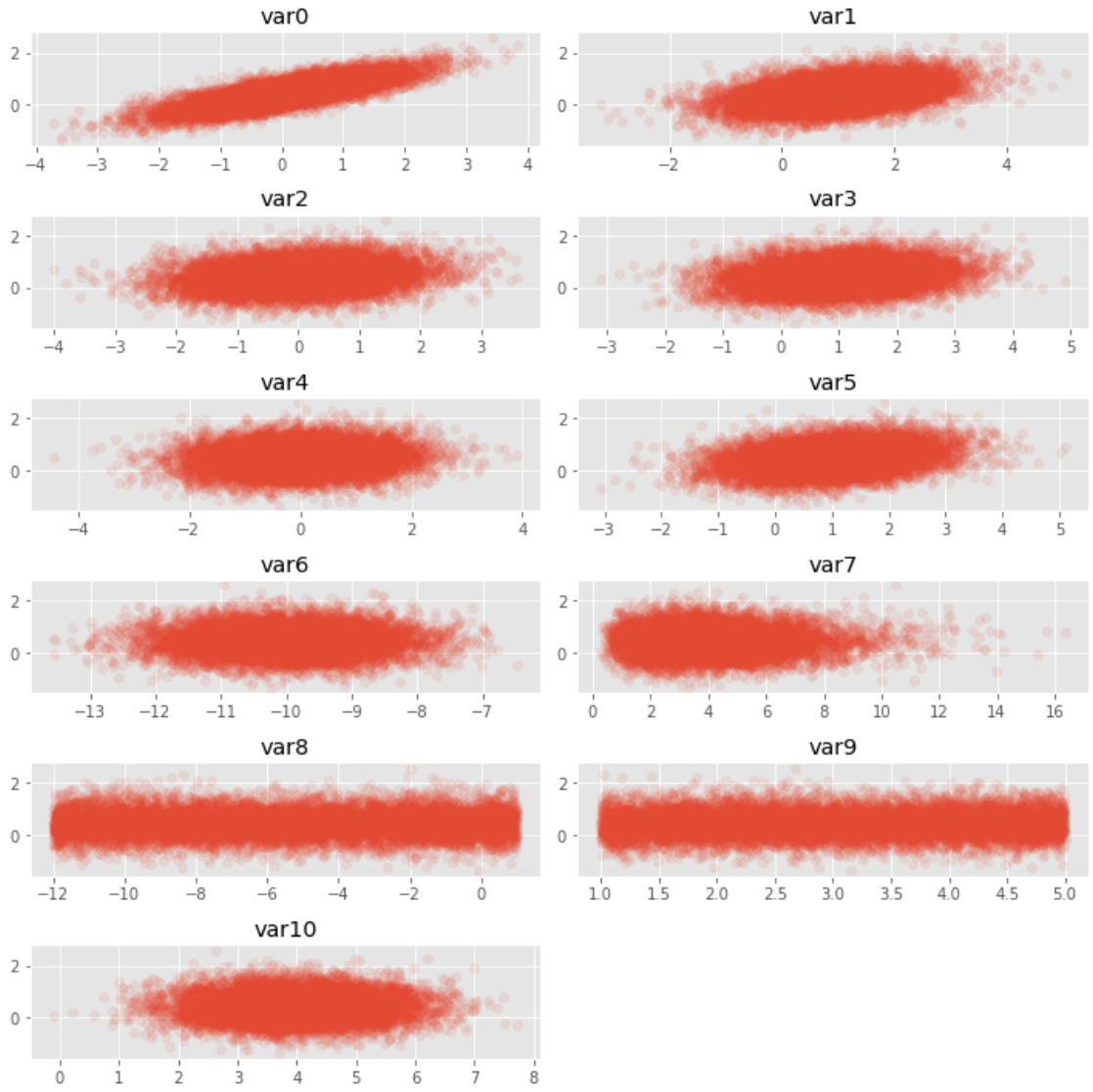
The list C above contains different values for the correlation strength between the relevant variables, defined in the appendix. For convenience, it takes values between -1 and 1, but we consider values between 0 and 1. We also consider different cases of correlation and model dependence.

A.3.1 Importance across cases and correlation.

The plots below illustrate the target relationship in the simulated data set:

[5]:
```
#weights
w1 = 0.4
w2 = 0.2
w3 = 0.1
w4 = 0.1
w5 = 0.05

X, y = generate_corr_dataset_regr(case=3,cor=0,size=size,target="linear",
                                  w1=w1,w2=w2,w3=w3,w4=w4,w5=w5)
```

[6]:
```
f = plot_y_vs_X(X=X,y=y)
```



In the cell below, we iterate over models, cases and the C list. We use the *groothelper* class which allows us to obtain the standard deviation of the variable importance across folds and it also has a Plot method that gives us some boxplots. Here we only show the output of the linear model in case 1 to save space. The results are very similar in the rest of the cases and can be found in my Github page.

[7]:
```
models = ['linear', 'weak non-linear', 'strong non linear']

for target in models:
    ␣
  ↪print("**********************************************************************")
```

```
   print("*                    Target = " +  target +"                    ␣
↪          *")
  ␣
↪print("**********************************************************************")
   print("\n")
   print("\n")
   for case in range(1,9):
       spread0 = []
       spread1 = []
       spread2 = []
       spread3 = []
       spread4 = []
       spread5 = []
       print("************************ CASE␣
↪",case,"*******************************")
      ␣
↪print("**************************************************************************")
       for cor in C:
           X,y =␣
↪generate_corr_dataset_regr(size=size,cor=cor,case=case,target=target,
                                      w1=w1,w2=w2,w3=w3,w4=w4,w5=w5)
           print("--------------correlation parameter set to ",␣
↪cor,"-------------------")
           helper = groothelper(X,y)
           helper.Plot()
           spread0.append(helper.s0)
           spread1.append(helper.s1)
           spread2.append(helper.s2)
           spread3.append(helper.s3)
           spread4.append(helper.s4)
           spread5.append(helper.s5)
       print("\n")
       fig=plt.figure()
       plt.plot(C,np.log(spread0))
       plt.plot(C,np.log(spread1))
       plt.plot(C,np.log(spread2))
       plt.plot(C,np.log(spread3))
       plt.plot(C,np.log(spread4))
       plt.plot(C,np.log(spread5))
       plt.legend(["var0","var1","var2","var3","var4","var5"])
       plt.xlabel("correlation")
       plt.ylabel("log standard deviation of score/" + target + "model")
       plt.title("Case " +  str(case)+ ":" + target + "model")
       plt.savefig("case" + str(case) + ":" + target + "model" + ".png")
```

```
        plt.close(fig)
    
↪print("**********************************************************************")
```

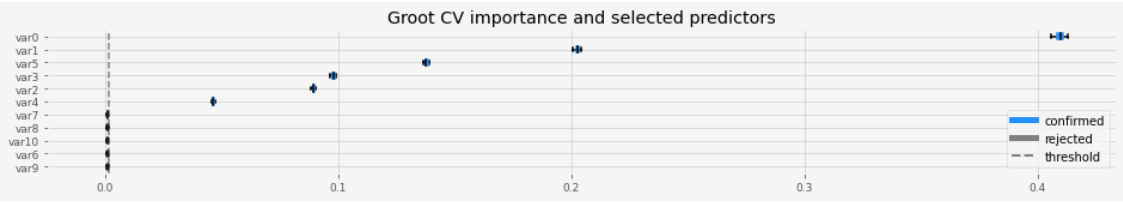
```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
```

```
**********************************************************************
*                    Target = linear                                 *
**********************************************************************
```

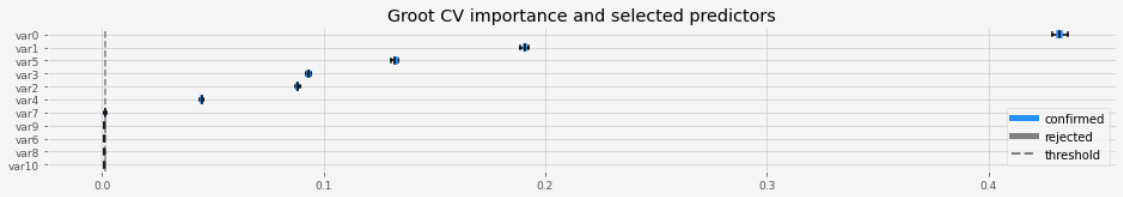
```
************************ CASE  1 *********************************
**********************************************************************
--------------correlation parameter set to  0 --------------------
```

```
Repeated k-fold: 100%|      | 25/25 [01:02<00:00,  2.51s/it]
```

```
Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.409294
1    var1  0.202485
2    var2  0.089286
3    var3  0.097897
4    var4  0.046227
5    var5  0.137881
```



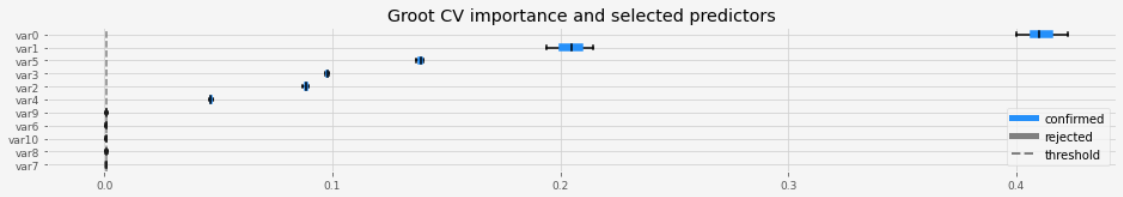
```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
--------------correlation parameter set to  0.2 -------------------
Repeated k-fold: 100%|     | 25/25 [01:13<00:00,  2.95s/it]
Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
7    var7
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.432193
1    var1  0.191029
2    var2  0.088196
3    var3  0.093105
4    var4  0.045106
5    var5  0.132423
```



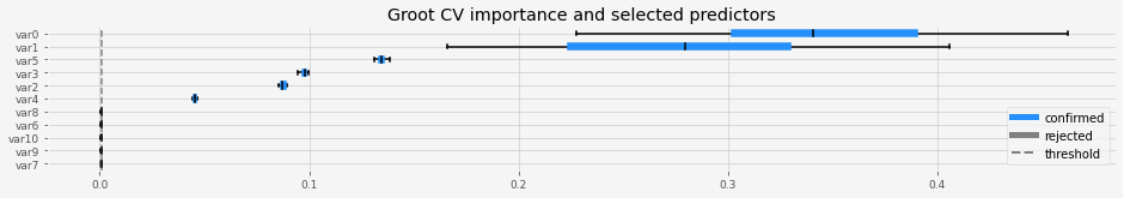

```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
--------------correlation parameter set to  0.5 -------------------
Repeated k-fold: 100%|     | 25/25 [01:11<00:00,  2.85s/it]
Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object
```

```
Median Values:
  feature       Med
0    var0  0.456001
1    var1  0.163602
2    var2  0.088782
3    var3  0.094706
4    var4  0.046300
5    var5  0.134509
```


```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
--------------correlation parameter set to  0.8 -------------------
Repeated k-fold: 100%|      | 25/25 [01:03<00:00,  2.54s/it]
Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.411629
1    var1  0.203407
2    var2  0.088408
3    var3  0.097795
4    var4  0.046787
5    var5  0.138523
```



```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

--------------correlation parameter set to  1 -------------------

Repeated k-fold: 100%|      | 25/25 [00:52<00:00,  2.11s/it]

Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.342561
1    var1  0.280635
2    var2  0.087503
3    var3  0.097548
4    var4  0.045382
5    var5  0.134407
```




```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
```

A.3.2 Different coefficients

Below we repeat the same experiment (only linear) with different coefficients. The idea here was to create some conditions that could potentially lead to an instability in the method (similar weights, some very small). We show only case 3.

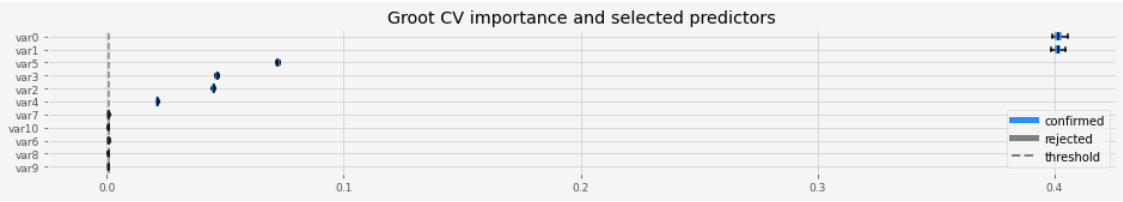
[8]:
```
#weights
w1 = 0.4
w2 = 0.4
w3 = 0.05
```

```
w4 = 0.05
w5 = 0.025
```

[9]:
```
target='linear'
print("*****************************************************************")
print("*                    Target = " +  target +"                                  ␣
 ↪      *")
print("*****************************************************************")
print("\n")
print("\n")
for case in range(1,9):
    spread0 = []
    spread1 = []
    spread2 = []
    spread3 = []
    spread4 = []
    spread5 = []
    print("************************ CASE␣
 ↪",case,"*********************************")
   ␣
 ↪print("*************************************************************************")
    for cor in C:
        X,y =␣
 ↪generate_corr_dataset_regr(size=size,cor=cor,case=case,target=target,
                                        w1=w1,w2=w2,w3=w3,w4=w4,w5=w5)
        print("--------------correlation parameter set to ",␣
 ↪cor,"--------------------")
        helper = groothelper(X,y)
        helper.Plot()
        spread0.append(helper.s0)
        spread1.append(helper.s1)
        spread2.append(helper.s2)
        spread3.append(helper.s3)
        spread4.append(helper.s4)
        spread5.append(helper.s5)
    print("\n")
    fig=plt.figure()
    plt.plot(C,np.log(spread0))
    plt.plot(C,np.log(spread1))
    plt.plot(C,np.log(spread2))
    plt.plot(C,np.log(spread3))
    plt.plot(C,np.log(spread4))
    plt.plot(C,np.log(spread5))
    plt.legend(["var0","var1","var2","var3","var4","var5"])
```

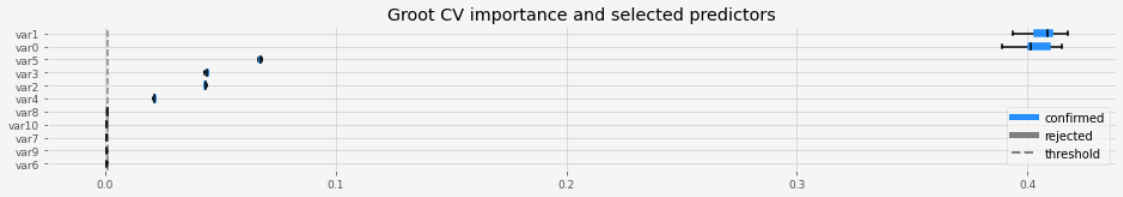
```
    plt.xlabel("correlation")
    plt.ylabel("standard deviation of score/" + target + "model")
    plt.title("Case " +  str(case)+ ":" + target + "model")
    plt.savefig("case" + str(case) + ":" + target + "model" +"V2" + ".
↪png")
    plt.close(fig)
  ␣
↪print("**************************************************************************")
```

```
**************************************************************************
************************ CASE  3 ********************************
**************************************************************************
--------------correlation parameter set to  0 -------------------
Repeated k-fold: 100%|      | 25/25 [00:55<00:00,  2.21s/it]
Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.401726
1    var1  0.401141
2    var2  0.045024
3    var3  0.046725
4    var4  0.021394
5    var5  0.072098
```



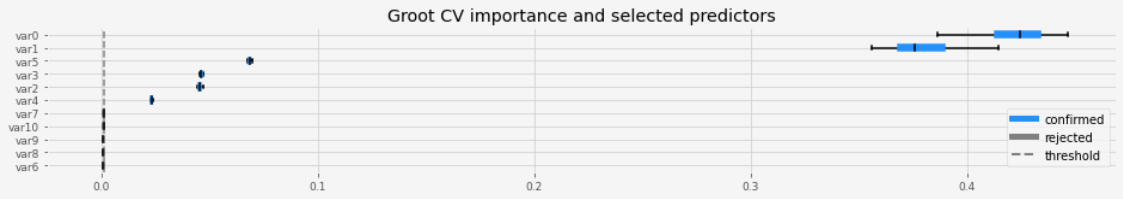

```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
```

```
--------------correlation parameter set to  0.2 -------------------

Repeated k-fold: 100%|      | 25/25 [00:58<00:00,  2.34s/it]

Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.402914
1    var1  0.407320
2    var2  0.043543
3    var3  0.044214
4    var4  0.021552
5    var5  0.067321
```

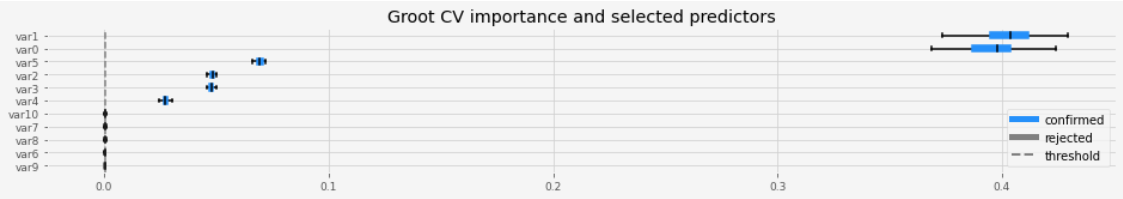



```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

--------------correlation parameter set to  0.5 -------------------

Repeated k-fold: 100%|      | 25/25 [01:07<00:00,  2.70s/it]

Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
```

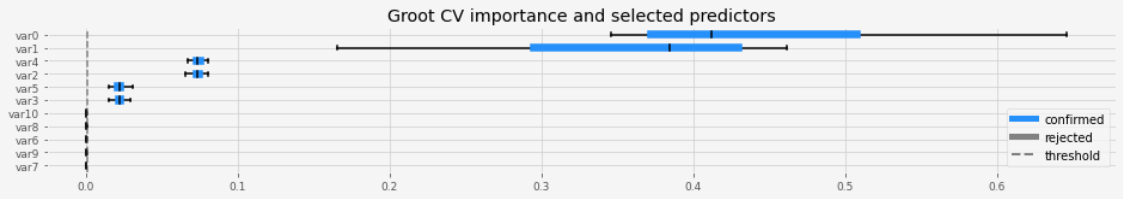
```
0    var0  0.421214
1    var1  0.381010
2    var2  0.045360
3    var3  0.046228
4    var4  0.023207
5    var5  0.068469
```




```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
--------------correlation parameter set to  0.8 --------------------
Repeated k-fold: 100%|     | 25/25 [00:36<00:00,  1.48s/it]
Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.396538
1    var1  0.403489
2    var2  0.048141
3    var3  0.047936
4    var4  0.027305
5    var5  0.069268
```

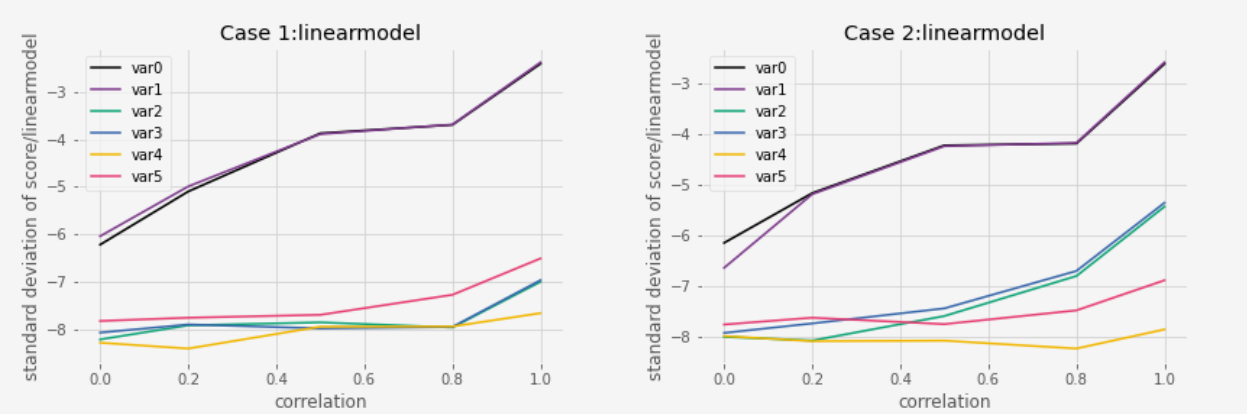


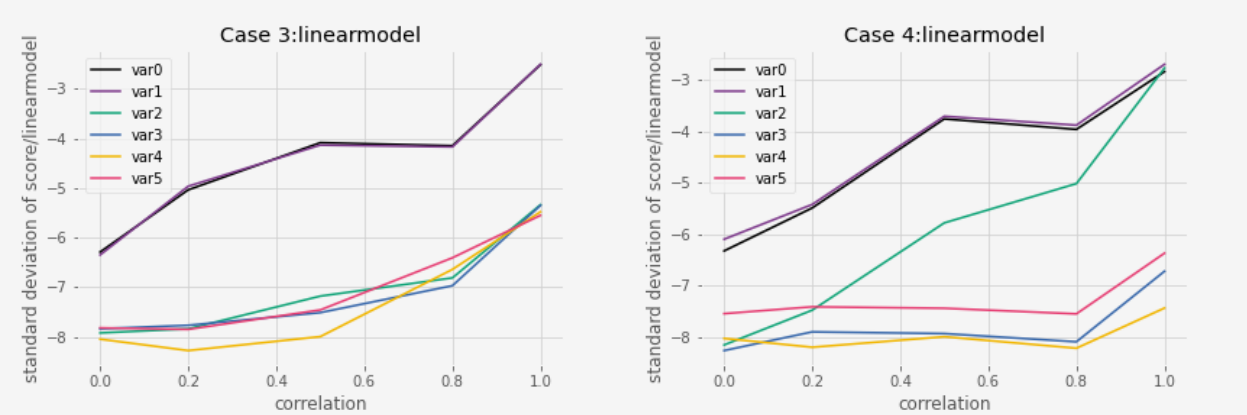
```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
--------------correlation parameter set to  1 -------------------
Repeated k-fold: 100%|     | 25/25 [00:19<00:00,  1.28it/s]
Relevant:
0    var0
1    var1
2    var2
3    var3
4    var4
5    var5
Name: feature, dtype: object

Median Values:
  feature       Med
0    var0  0.441593
1    var1  0.362088
2    var2  0.073882
3    var3  0.021409
4    var4  0.074054
5    var5  0.022178
```

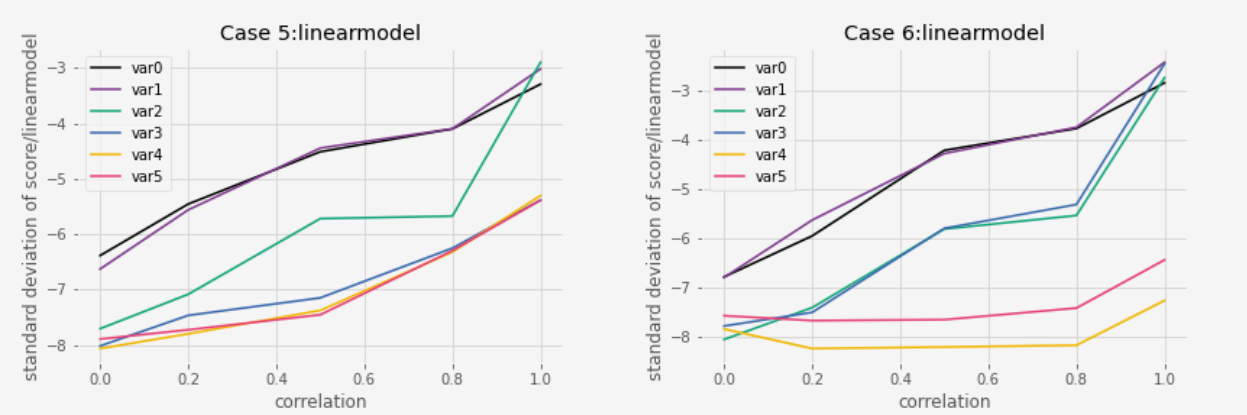


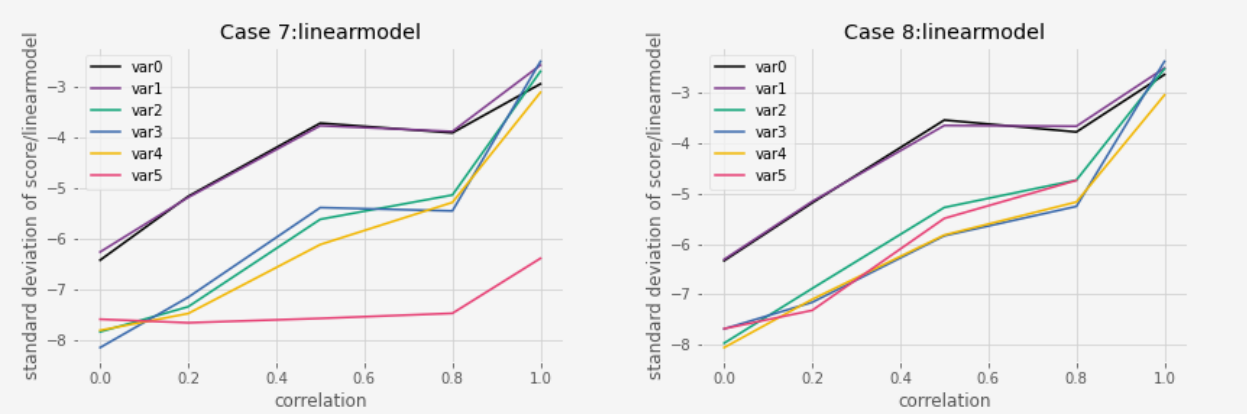

```
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]
```

The results are summarized in the figures below. Again, we observe the same behavior.


Case 1:linearmodel
Case 2:linearmodel
standard deviation of score/linearmodel
correlation
var0
var1
var2
var3
var4
var5


Case 3:linearmodel
Case 4:linearmodel
standard deviation of score/linearmodel
correlation
var0
var1
var2
var3
var4
var5


Case 5:linearmodel
Case 6:linearmodel
standard deviation of score/linearmodel
correlation
var0
var1
var2
var3
var4
var5


Case 7:linearmodel
Case 8:linearmodel
standard deviation of score/linearmodel
correlation
var0
var1
var2
var3
var4
var5

A.3.3 Stability

Above, the figures and boxplots are obtained by using the method only once on the training set. However, applying the method multiple times one can get different results. It is important therefore to test the stability of the method using multiple runs. This is attempted in the cells below. We are using the function *grootRepeatCor* that implements the method N times over a certain range of correlation values. We consider it as an error whenever the method does not return the whole set of relevant features. The error rate is therefore calculated as: *number of errors/N*

[4]:
```
#weights
w1 = 0.4
w2 = 0.2
w3 = 0.1
w4 = 0.1
w5 = 0.05
models = ['linear', 'weak non-linear', 'strong non linear']

for target in models:
    ␣
 ↪print("*****************************************************************")
    print("*                    Target = " +  target +"                    ␣
 ↪          *")
    ␣
 ↪print("*****************************************************************")
    print("\n")
    print("\n")
    for case in range(1,9):
        print("************************* CASE␣
 ↪",case,"*******************************")
        ␣
 ↪print("*****************************************************************************")
        grootRepeatCor(size=size,w1=w1,w2=w2,w3=w3,w4=w4,w5=w5,
                       target=target,case=case,C=C)
        ␣
 ↪print("*****************************************************************************")
        print("\n")
```

Once more, we have suppressed the output of the cell but one can check that the method is very stable. Across models and correlation values the error rate is 0. However, one can check that when the *cor* is set to 1 and all the variables are correlated the error rate jumps to 0.8.

A.3.4 Different coefficients

We repeat the above experiment with different coefficients. The results are equivalent.

[5]:
```
#weights
w1 = 0.4
w2 = 0.4
w3 = 0.05
w4 = 0.05
w5 = 0.025
target = 'linear'
print("**********************************************************************")
print("*                    Target = " +  target +"                          ␣
 ↪      *")
print("**********************************************************************")
print("\n")
print("\n")
for case in range(1,9):
    print("************************ CASE␣
 ↪",case,"*******************************")
   ␣
 ↪print("**********************************************************************")
    grootRepeatCor(size=size,w1=w1,w2=w2,w3=w3,w4=w4,w5=w5,
                   target=target,case=case,C=C)
   ␣
 ↪print("**********************************************************************")
    print("\n")
```

Overall, the method behaves very well under the given circumstances.

A.4 More variables

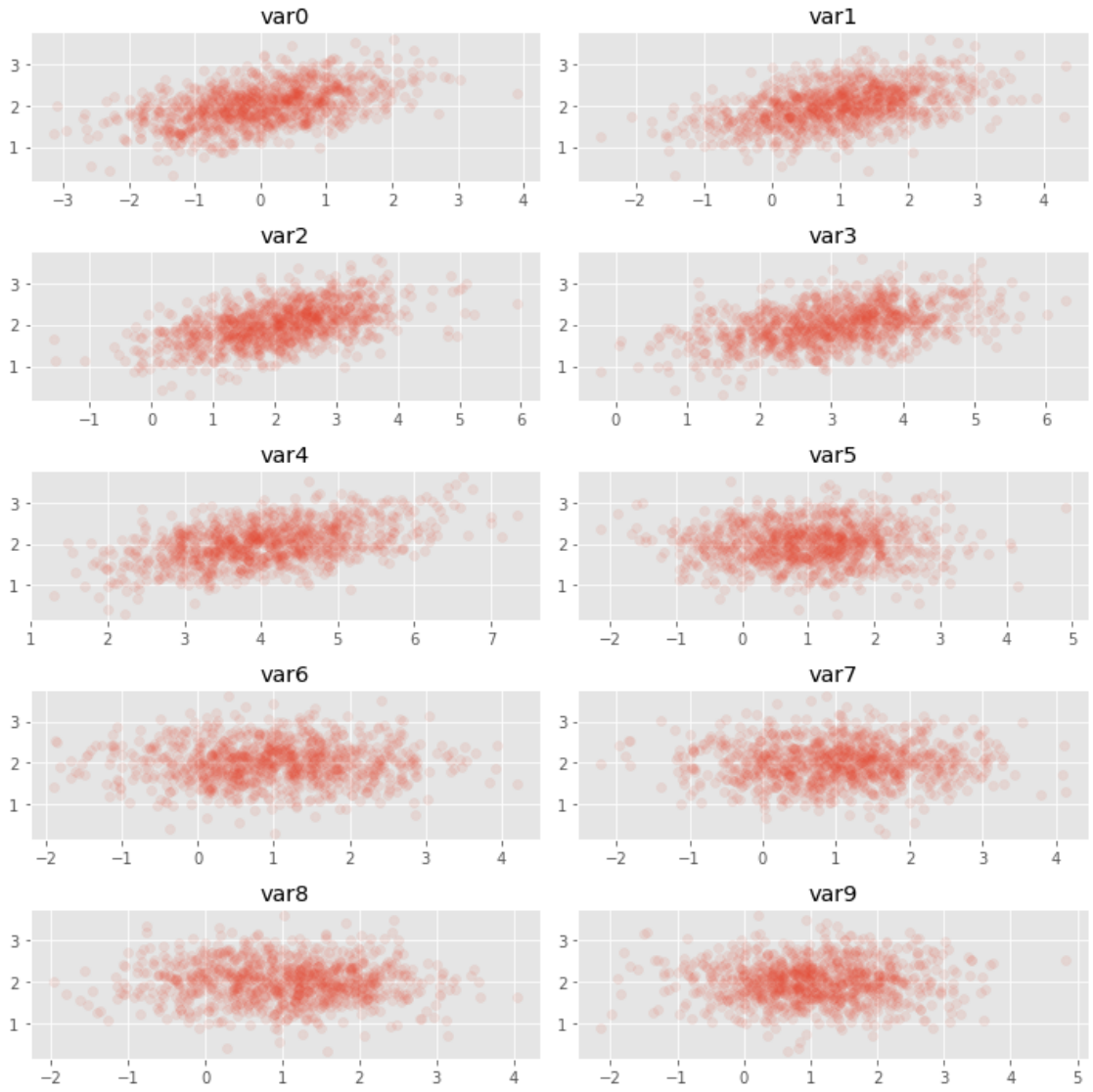
In what follows we will make use of the *generate_more_var* function that generates a data set with a given number of relevant columns (and the same number of irrelevant variables). The target relationship is linear and all the coefficients are equal and add up to 1. We will make use of this function to check how the number of correlated columns affects the performance of the method.

A.4.1 Illustration of code

Just to illustrate the output of the function, we plot the data set relations first for 10 columns and then for 20.

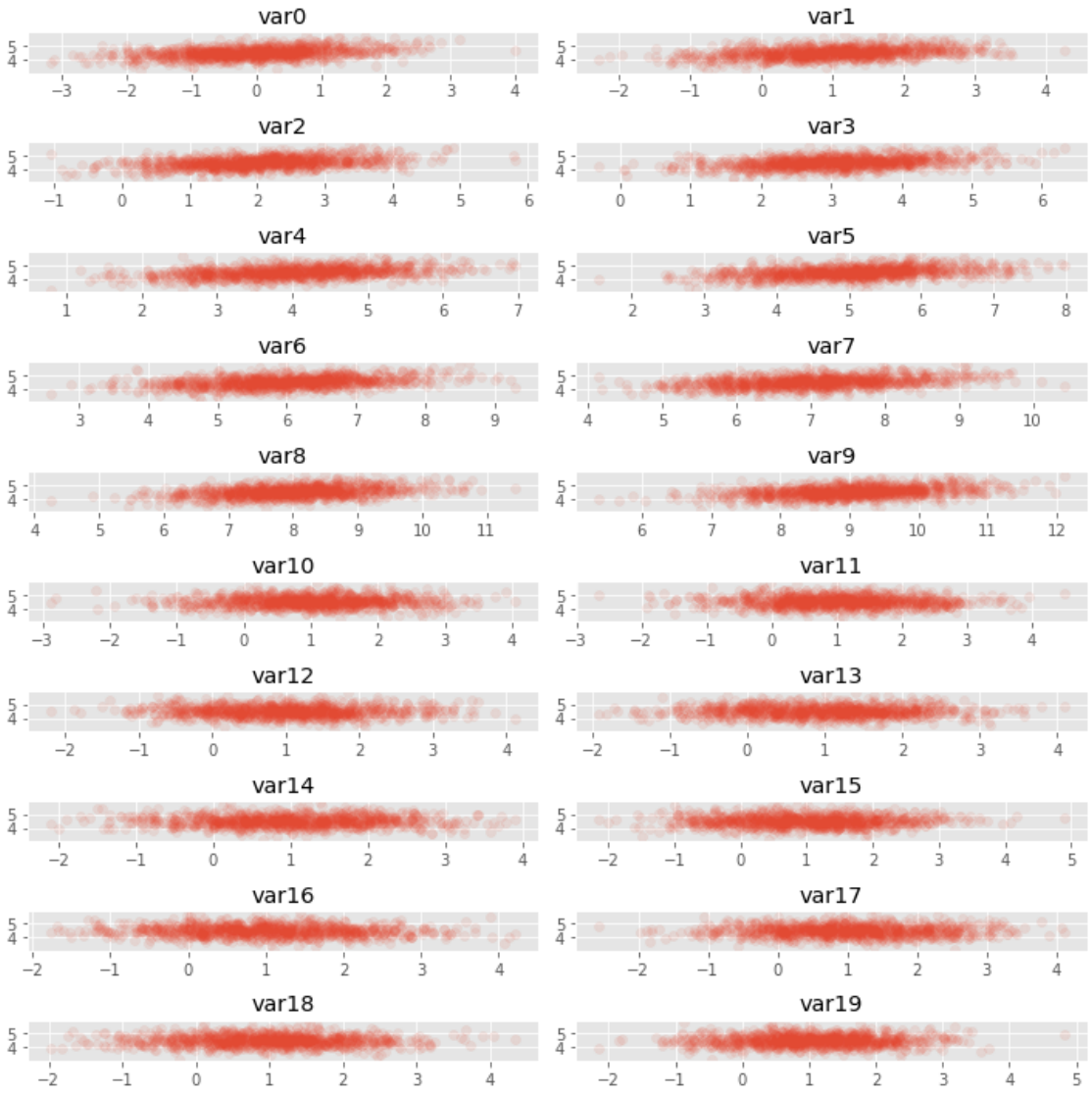
[14]:
```
X,y = generate_more_var(size=1000,n=5,cor=0.02)
```

[15]:
```
f = plot_y_vs_X(X=X,y=y)
```



[16]:
```
X,y = generate_more_var(size=1000,n=10,cor=0.02)
```

[17]:
```
f = plot_y_vs_X(X=X,y=y)
```



A.4.2 Stability across cases

Let us now check the stability by iterating over different numbers of columns. In the cell below, we test how many relevant variables the method picks out for 200,400,800,1000 and 1200 columns. All the variables are correlated with a correlation coefficient set to 0.8. Note that we have kept the number of rows to 1000. Otherwise the computation time becomes too large. The stability is tested by running the experiment N (here 5) times and printing the error rate and also the average number of relevant variables.

[21]:
```
size = 1000
cor = 0.8
N = 5
```

```
np.random.seed(42)
A = [100,200,400,500,600]

for n in A:
    c = 0
    relVar = set(['var' + str(i) for i in range(n)])
    # number of relevant var in each run
    nbV = []
    for i in range(N):
        # create data set
        X,y = generate_more_var(size=size,n = n,cor=cor)
        #groot
        feat_selector = arfsgroot.GrootCV(objective='rmse', cutoff = 1,␣
↪n_folds=5, n_iter=5)
        feat_selector.fit(X, y, sample_weight=None)
        if relVar.intersection(set(feat_selector.support_names_)) ==␣
↪relVar:
            c += 1

        nbV.append(len(relVar.intersection(set(feat_selector.
↪support_names_))))
    print("----------- number of columns = ", 2*n,"-------------")
    print("number of times that all relevant features were chosen: ",c,␣
↪"out of ",N)
    print("error rate = ", 1-c/N)
    print("average number of relavant variables: ")
    print(sum(nbV)/len(nbV))
```

```
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.42it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.43it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.45it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.45it/s]
Repeated k-fold: 100%|      | 25/25 [00:17<00:00,  1.44it/s]
```

```
----------- number of columns =  200 -------------
number of times that all relevant features were chosen:  5 out of  5
error rate =  0.0
average number of relavant variables:
100.0
```

```
Repeated k-fold: 100%|      | 25/25 [00:38<00:00,  1.56s/it]
Repeated k-fold: 100%|      | 25/25 [00:39<00:00,  1.56s/it]
Repeated k-fold: 100%|      | 25/25 [00:38<00:00,  1.55s/it]
Repeated k-fold: 100%|      | 25/25 [00:39<00:00,  1.57s/it]
Repeated k-fold: 100%|      | 25/25 [00:39<00:00,  1.57s/it]
```

```
---------- number of columns =  400 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
average number of relavant variables:
120.0

Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.94s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.95s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.93s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.94s/it]
Repeated k-fold: 100%|      | 25/25 [01:38<00:00,  3.94s/it]

---------- number of columns =  800 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
average number of relavant variables:
74.0

Repeated k-fold: 100%|      | 25/25 [02:24<00:00,  5.76s/it]
Repeated k-fold: 100%|      | 25/25 [02:24<00:00,  5.79s/it]
Repeated k-fold: 100%|      | 25/25 [02:23<00:00,  5.76s/it]
Repeated k-fold: 100%|      | 25/25 [02:24<00:00,  5.78s/it]
Repeated k-fold: 100%|      | 25/25 [02:23<00:00,  5.75s/it]

---------- number of columns =  1000 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
average number of relavant variables:
61.0

Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.32s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.31s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.32s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.31s/it]
Repeated k-fold: 100%|      | 25/25 [02:37<00:00,  6.31s/it]

---------- number of columns =  1200 -------------
number of times that all relevant features were chosen:  0 out of  5
error rate =  1.0
average number of relavant variables:
59.0
```

A.5 Noise

Let us now test the influence of noise terms (in the linear model). The relevant variables are kept uncorrelated.

We begin by defining our parameters:

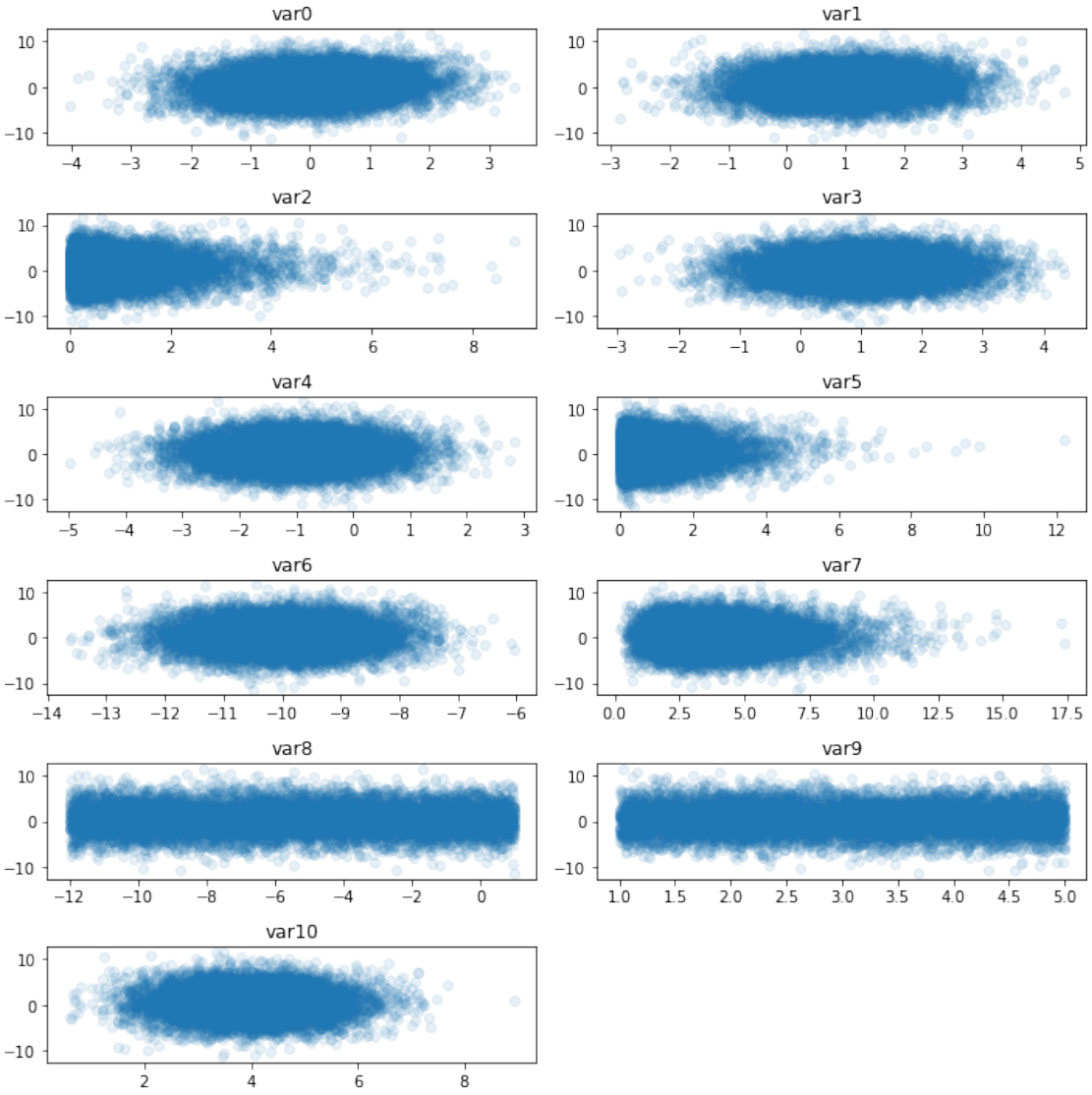
[3]:
```
#weights
w1 = 0.4
w2 = 0.2
w3 = 0.1
w4 = 0.1
w5 = 0.05

case = 1
cor = 0
target = 'linear'

size = 10_000
```

We can observe the effect of the noise by plotting the predictors against the target:

[4]:
```
X, y =␣
↪generate_corr_dataset_regr(case=case,cor=cor,size=size,target=target,
                                                    ␣
↪w1=w1,w2=w2,w3=w3,w4=w4,w5=w5,noise=True, sd = 3)
```

[5]:
```
f = plot_y_vs_X(X=X,y=y)
```

As we can see below the relationships are greatly distorted by the error term.



Once more we will perform a stability test. We run the experiment N times (here 10) and iterate over different values of the standard deviation of noise placed in the list SD. We basically want to calculate the error rate for each of these values. This is achieved with *grootRepeatNoise* function that can be found in the appendix. The output is given below.

[10]:
```
# plot GrootCV results for sd from 0 to 2
SD = [i*0.2 for i in range(0,16)]
```

[11]:
```
p = grootRepeatNoise(size=size,w1=w1,w2=w2,w3=w3,w4=w4,w5=w5,target=target,case=case,
                     cor=cor,N=10,SD=SD)
```

```
Repeated k-fold: 100%|      | 25/25 [00:59<00:00,  2.39s/it]
```

```
Repeated k-fold: 100%|      | 25/25 [01:06<00:00,  2.66s/it]
Repeated k-fold: 100%|      | 25/25 [01:05<00:00,  2.63s/it]
Repeated k-fold: 100%|      | 25/25 [01:02<00:00,  2.50s/it]
Repeated k-fold: 100%|      | 25/25 [01:10<00:00,  2.83s/it]
Repeated k-fold: 100%|      | 25/25 [01:07<00:00,  2.72s/it]
Repeated k-fold: 100%|      | 25/25 [01:09<00:00,  2.77s/it]
Repeated k-fold: 100%|      | 25/25 [01:02<00:00,  2.50s/it]
Repeated k-fold: 100%|      | 25/25 [01:01<00:00,  2.47s/it]
Repeated k-fold: 100%|      | 25/25 [00:55<00:00,  2.23s/it]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  0.0 -------------
number of times that all relevant features were chosen:  10 out of 10
error rate =  0.0

Repeated k-fold: 100%|      | 25/25 [00:15<00:00,  1.66it/s]
Repeated k-fold: 100%|      | 25/25 [00:14<00:00,  1.68it/s]
Repeated k-fold: 100%|      | 25/25 [00:16<00:00,  1.55it/s]
Repeated k-fold: 100%|      | 25/25 [00:15<00:00,  1.61it/s]
Repeated k-fold: 100%|      | 25/25 [00:16<00:00,  1.55it/s]
Repeated k-fold: 100%|      | 25/25 [00:15<00:00,  1.64it/s]
Repeated k-fold: 100%|      | 25/25 [00:15<00:00,  1.59it/s]
Repeated k-fold: 100%|      | 25/25 [00:15<00:00,  1.63it/s]
Repeated k-fold: 100%|      | 25/25 [00:16<00:00,  1.47it/s]
Repeated k-fold: 100%|      | 25/25 [00:15<00:00,  1.66it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  0.2 -------------
number of times that all relevant features were chosen:  10 out of 10
error rate =  0.0

Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.13it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.23it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.19it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.17it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.20it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.11it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.23it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.09it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.17it/s]
Repeated k-fold: 100%|      | 25/25 [00:11<00:00,  2.12it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  0.4 -------------
number of times that all relevant features were chosen:  10 out of 10
error rate =  0.0

Repeated k-fold: 100%|      | 25/25 [00:09<00:00,  2.77it/s]
```

```
Repeated k-fold: 100%|      | 25/25 [00:09<00:00,  2.65it/s]
Repeated k-fold: 100%|      | 25/25 [00:08<00:00,  2.78it/s]
Repeated k-fold: 100%|      | 25/25 [00:09<00:00,  2.71it/s]
Repeated k-fold: 100%|      | 25/25 [00:08<00:00,  2.89it/s]
Repeated k-fold: 100%|      | 25/25 [00:08<00:00,  2.87it/s]
Repeated k-fold: 100%|      | 25/25 [00:09<00:00,  2.65it/s]
Repeated k-fold: 100%|      | 25/25 [00:08<00:00,  2.83it/s]
Repeated k-fold: 100%|      | 25/25 [00:09<00:00,  2.72it/s]
Repeated k-fold: 100%|      | 25/25 [00:09<00:00,  2.67it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  0.6000000000000001 -------------
number of times that all relevant features were chosen:  10 out of 10
error rate =  0.0

Repeated k-fold: 100%|      | 25/25 [00:08<00:00,  2.94it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.14it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.15it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.23it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.37it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.39it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.52it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.27it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.16it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.22it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  0.8 -------------
number of times that all relevant features were chosen:  10 out of 10
error rate =  0.0

Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.46it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.71it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.69it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.60it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.49it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.65it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.34it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.29it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.53it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.82it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  1.0 -------------
number of times that all relevant features were chosen:  10 out of 10
error rate =  0.0

Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.95it/s]
```

```
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.84it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.34it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  4.04it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.63it/s]
Repeated k-fold: 100%|      | 25/25 [00:07<00:00,  3.37it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.18it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.71it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.74it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  4.04it/s]
Repeated k-fold:   4%|          | 1/25 [00:00<00:04,  5.31it/s]

---------- standard deviation of noise =  1.2000000000000002 -------------
number of times that all relevant features were chosen:  9 out of 10
error rate =  0.09999999999999998

Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.34it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.89it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  4.09it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  4.16it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.83it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.85it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  4.17it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.41it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.58it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.90it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  1.4000000000000001 -------------
number of times that all relevant features were chosen:  10 out of 10
error rate =  0.0

Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.68it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.51it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.69it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.42it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.30it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.81it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.65it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.30it/s]
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  3.75it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.43it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  1.6 -------------
number of times that all relevant features were chosen:  7 out of 10
error rate =  0.30000000000000004

Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.13it/s]
```

```
Repeated k-fold: 100%|      | 25/25 [00:06<00:00,  4.08it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.93it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.48it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.62it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.78it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.76it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.82it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.13it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.05it/s]
Repeated k-fold:   0%|          | 0/25 [00:00<?, ?it/s]

---------- standard deviation of noise =  1.8 -------------
number of times that all relevant features were chosen:  7 out of 10
error rate =  0.30000000000000004

Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.95it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.34it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.07it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.57it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.80it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  5.00it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.83it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.70it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.80it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.59it/s]
Repeated k-fold:   4%|          | 1/25 [00:00<00:03,  7.25it/s]

---------- standard deviation of noise =  2.0 -------------
number of times that all relevant features were chosen:  5 out of 10
error rate =  0.5

Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  6.05it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.16it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.46it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.61it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.27it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.32it/s]
Repeated k-fold: 100%|      | 25/25 [00:05<00:00,  4.62it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.59it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.14it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  6.20it/s]
Repeated k-fold:   4%|          | 1/25 [00:00<00:04,  5.90it/s]

---------- standard deviation of noise =  2.2 -------------
number of times that all relevant features were chosen:  3 out of 10
error rate =  0.7

Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.80it/s]
```

```
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.86it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.77it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.66it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.88it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.81it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.74it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.74it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.28it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.29it/s]
Repeated k-fold:   4%|          | 1/25 [00:00<00:03,  6.44it/s]

---------- standard deviation of noise =  2.4000000000000004 -------------
number of times that all relevant features were chosen:  2 out of 10
error rate =  0.8

Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.26it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.46it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.56it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  6.07it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.98it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.54it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.49it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.13it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.64it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.38it/s]
Repeated k-fold:   4%|          | 1/25 [00:00<00:02,  8.37it/s]

---------- standard deviation of noise =  2.6 -------------
number of times that all relevant features were chosen:  3 out of 10
error rate =  0.7

Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.41it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.60it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.70it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.87it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.55it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.30it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.29it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.24it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.39it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.83it/s]
Repeated k-fold:   4%|          | 1/25 [00:00<00:04,  5.69it/s]

---------- standard deviation of noise =  2.8000000000000003 -------------
number of times that all relevant features were chosen:  0 out of 10
error rate =  1.0

Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.35it/s]
```

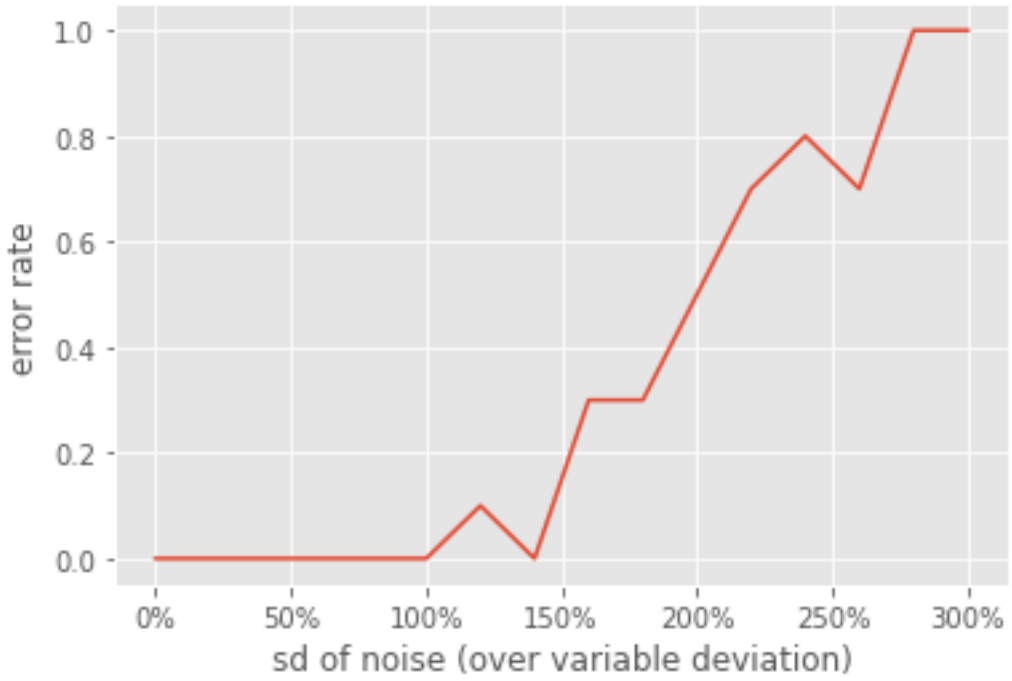
```
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.53it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.86it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.70it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.52it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  6.09it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.69it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.47it/s]
Repeated k-fold: 100%|      | 25/25 [00:03<00:00,  6.72it/s]
Repeated k-fold: 100%|      | 25/25 [00:04<00:00,  5.23it/s]

---------- standard deviation of noise =  3.0 -------------
number of times that all relevant features were chosen:  0 out of 10
error rate =  1.0
```

We can now plot the error rate against the standard deviation of noise over the variable deviation (set to unity) as a percentage:

[12]:
```
plt.style.use(style='ggplot')
plt.plot(SD,p)
plt.xlabel("sd of noise (over variable deviation)")
plt.ylabel("error rate")
plt.gca().set_xticklabels(['{:.0f}%'.format(x*100) for x in plt.gca().
 ↪get_xticks()])
```

```
FixedFormatter should only be used together with FixedLocator
```

[12]:
```
[Text(-0.5, 0, '-50%'),
 Text(0.0, 0, '0%'),
 Text(0.5, 0, '50%'),
 Text(1.0, 0, '100%'),
 Text(1.5, 0, '150%'),
 Text(2.0, 0, '200%'),
 Text(2.5, 0, '250%'),
 Text(3.0, 0, '300%'),
 Text(3.5, 0, '350%')]
```


1.0
0.8
0.6
0.4
0.2
0.0
error rate
0%
50%
100%
150%
200%
250%
300%
sd of noise (over variable deviation)

B tools

B.1 Imports

The tools.py script contains the functions discussed in the main body of this report. In order for the functions (and the arfs package) to work, some modules must be imported beforehand. These are the following:

```
import numpy as np
from sklearn.inspection import permutation_importance
from sklearn.base import clone
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
import shap
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from lightgbm import LGBMRegressor, LGBMClassifier
from xgboost import XGBRegressor, XGBClassifier
import arfs
import pandas as pd
import arfs.featselect as arfsfs
import arfs.allrelevant as arfsgroot
from matplotlib import pyplot as plt
from arfs.utils import highlight_tick
import timeit
import math
```

B.2 Plots

The function below can be found in the arfs repository in the utils.py script. It plots the target y against each one of the predictors in the X data frame:

```
def plot_y_vs_X(X, y, ncols=2, figsize=(10, 10)):
    """
    Plot target vs relevant and non-relevant predictors

    :param X: pd.DataFrame
        the pd DF of the predictors
    :param y: np.array
        the target
    :param ncols: int, default=2
        the number of columns in the facet plot
    :param figsize: 2-uple of float, default=(10, 10)
        the figure size
```

```
    :return:f, matplotlib objects
        the univariate plots y vs pred_i
    """

    X = pd.DataFrame(X)
    ncols_to_plot = X.shape[1]
    n_rows = int(np.ceil(ncols_to_plot / ncols))

    # Create figure and axes (this time it's 9, arranged 3 by 3)
    f, axs = plt.subplots(nrows=n_rows, ncols=ncols, figsize=figsize)

    # delete non-used axes
    n_charts = ncols_to_plot
    n_subplots = n_rows * ncols
    cols_to_enum = X.columns

    # Make the axes accessible with single indexing
    if n_charts > 1:
        axs = axs.flatten()

    for i, col in enumerate(cols_to_enum):
        # select the axis where the map will go
        if n_charts > 1:
            ax = axs[i]
        else:
            ax = axs

        ax.scatter(X[col], y, alpha=0.1)
        ax.set_title(col)

    if n_subplots > n_charts > 1:
        for i in range(n_charts, n_subplots):
            ax = axs[i]
            ax.set_axis_off()

    # Display the figure
    plt.tight_layout()
    return f
```

B.3 Correlated variables Generation

The next function is used a lot in the report. Its purpose is to generate a data set of predictors and a target with a pre-defined relationship. The *case* parameter defines the number of correlated variables. This correlation is achieved using a normal multivariate distribution from which we sample the correlated variables. The correlation strength parameter enters

as the non diagonal element in the covariance matrix. This matrix is constructed in the following way: First we define:

$$m = \begin{pmatrix} 1 & cor \dots & cor \\ \vdots & \ddots & \\ cor & \dots & 1 \end{pmatrix} \tag{22}$$

Then we define the covariance matrix:

$$Cov = m^T * m \tag{23}$$

This ensures that we get a semi-positive definite symmetric matrix.

Note also that the function has the *noise* parameter that can be used to add the noise term in the target.

```
"""CORRELATED AND STRONGLY RELEVANT: LINEAR + WEAK NON LINEARITY
MODEL + Strong NON linearity  AND (NO) NOISE"""

# This function generates different data sets. Specifically it generates 8
# different cases:
# case1 : 1 pair of correlated and strongly relevant
# case2 : 2 pairs of cor. and s. r.
# case3 : 3 pairs of c. and s.r.
# case4 : 3 c. and s.r.
# case5: 3 and 3 ..
# case6 : 4 c. and s.r.
# case7 : 5 ...
# case8 : 6

def generate_corr_dataset_regr(size, cor, w1, w2, w3, w4, w5, case=1,␣
 ↪target='linear', noise=False, sd=0):


    """ This function returns a data set X of predictors in data frame␣
 ↪format and a numpy
    array that contains the target y.
    Parameters:
    1. The relationship between the target and the features is set with
    the 'target' parameter: 'linear', 'weak non-linear' and 'strong non-
 ↪linear'.
    2. The 'cor' parameter defines the correlation strength between␣
 ↪variables.
    3. The weights 'w1', 'w2', 'w3', 'w4', 'w5' are the coefficients of␣
 ↪the features in
    the input-output relation.
```

```
   4. The 'size' is the number of rows of the data sets.
   5. The  'case' parameter distinguishes between the different cases of␣
↪correlation.
   6. The 'noise' is a Boolean that determines the presense/absense of␣
↪noise
   7. 'sd' is the standard deviation of noise. """

   # error term
   w = np.random.normal(loc=0,size=size,scale=sd)
   # GENERATION ----------------------------------------------------
   if case <= 3:
       # define mean
       mean = [0, 1]
       # covariance matrix
       m = np.array([[1, cor], [cor, 1]])
       cov = (1/(1+cor**2))*np.dot(m, m.transpose())
       # first pair
       x0, x1 = np.random.multivariate_normal(mean, cov, size=size).T
       #x0 = np.random.normal(0,1,size)
       #x1 = np.random.normal(1,1,size)
       if case == 1:
           # no correlation
           x2 = np.random.gamma(1, 1, size)
           x3 = np.random.normal(1, 1, size)
           x4 = np.random.normal(-1, 1, size)
           x5 = np.random.gamma(1, 1, size)
       else:
           # second pair
           x2, x3 = np.random.multivariate_normal(mean, cov, size=size).T
           if case == 2:
               x4 = np.random.normal(-1, 1, size)
               x5 = np.random.gamma(1, 1, size)
           else:
               # third pair
               x4, x5 = np.random.multivariate_normal(mean, cov,␣
↪size=size).T

   if case == 4 or case == 5:
       # mean
       mean = [0, 1, 2]
       # covariance
       m = np.array([[1, cor, cor], [cor, 1, cor], [cor, cor, 1]])
       cov = (1/(1+2*cor**2))*np.dot(m, m.transpose())
       # three correlated
```

```
        x0, x1, x2 = np.random.multivariate_normal(mean, cov, size=size).T
        if case == 4:
            x3 = np.random.normal(1, 1, size)
            x4 = np.random.normal(-1, 1, size)
            x5 = np.random.gamma(1, 1, size)
        else:
            # three more correlated
            x3, x4, x5 = np.random.multivariate_normal(mean, cov,␣
↪size=size).T

    if case == 6:
        # mean
        mean = [0,1,2,3]
        # covariance matrix
        m = np.array([[1, cor, cor, cor], [cor, 1, cor, cor],
                      [cor, cor, 1, cor], [cor, cor, cor, 1]])
        cov =(1/(1+3*cor**2))*np.dot(m, m.transpose())
        # four correlated
        x0, x1, x2, x3 = np.random.multivariate_normal(mean, cov,␣
↪size=size).T
        x4 = np.random.normal(-1, 1, size)
        x5 = np.random.gamma(1, 1, size)

    if case == 7:
        # mean
        mean = [0,1,2,3,4]
        # covariance
        m = np.
↪array([[1,cor,cor,cor,cor],[cor,1,cor,cor,cor],[cor,cor,1,cor,cor],␣
↪[cor,cor,cor,1,cor],[cor,cor,cor,cor,1]])
        cov = (1/(1+4*cor**2))*np.dot(m, m.transpose())
        # 5 correlated
        x0, x1, x2, x3, x4 = np.random.multivariate_normal(mean, cov,␣
↪size=size).T
        x5 = np.random.gamma(1,1,size)

    if case == 8:
        # mean
        mean = [0,1,2,3,4,5]
        # covariance
        m = np.array([[1,cor,cor,cor,cor,cor],
                      [cor,1,cor,cor,cor,cor],[cor,cor,1,cor,cor,cor],
                      [cor,cor,cor,1,cor,cor],[cor,cor,cor,cor,1,cor],
                      [cor,cor,cor,cor,cor,1]])
```

```
        cov = (1/(1+5*cor**2))*np.dot(m, m.transpose())
        # 6 correlated
        x0, x1, x2, x3, x4, x5 = np.random.multivariate_normal(mean, cov,␣
↪size=size).T


    # DATA FRAME CONSTRUCTION ------------------------------------------
↪---------------
    # initiate empty matrix
    X = np.zeros((size,11))


    # 6 columns = relevant features
    X[:, 0] = x0
    X[:,1] = x1
    X[:,2] = x2
    X[:,3] = x3
    X[:,4] = x4
    X[:,5] = x5

    # 5 columns =  irrelevant ones
    X[:,6] = np.random.normal(-10,1,size)
    X[:,7] = np.random.gamma(4,1,size)
    X[:,8] = np.random.uniform(-12,1,size)
    X[:,9] = np.random.uniform(5,1,size)
    X[:,10] = np.random.normal(4,1,size)


    # make  it a pandas DF
    column_names = ['var' + str(i) for i in range(11)]
    X = pd.DataFrame(X)
    X.columns = column_names



    # CONSTRUCTION OF TARGET -------------------------------------------
↪-------------------------
    #
    if target == 'linear':
        # target lin model - no noise
        y = w1*x0 + w2*x1 + w3*x2 + w4*x3 + w5*x4 + (1-w1-w2-w3-w4-w5)*x5
    elif target == 'weak non-linear':
        # weak linearity
```

```
        y = w1*np.sqrt(abs(x0)) + w2*x1 + w3*x2 + w4*x3 + w5*x4 + (1-w1-w2-
↪w3-w4-w5)*x5
    elif target == 'strong non-linear':
        # strong non linearity
        y = w1*np.sin(x0) + w2*x1 + w3*x2 + w4*x3 + w5*x4 + (1-w1-w2-w3-w4-
↪w5)*x5
    else:
        # just return zero
        y = np.zeros(size)

    # add noise if true
    if noise == True:
        y += w


    return X, y
```

B.4 Stability and Correlation

This is a simple function that makes use of the one above. It basically runs the same experiment N times and prints the number of times that all relevant features were chosen.

```
def grootRepeatCor(size,w1,w2,w3,w4,w5,target,case,N=5,C=[0]):
    c = 0
    relVar = set(['var0','var1','var2','var3','var4','var5'])
    np.random.seed(42)
    N = N
    p = []
    C = C
    for cor in C:
        c = 0
        for i in range(N):
            # create data set
            X,y = generate_corr_dataset_
↪regr(w1=w1,w2=w2,w3=w3,w4=w4,w5=w5,target=target,
                                                size=size,cor=cor,case=case)
            #groot
            feat_selector = arfsgroot.GrootCV(objective='rmse', cutoff = 1,␣
↪n_folds=5, n_iter=5)
            feat_selector.fit(X, y, sample_weight=None)
            if relVar.intersection(set(feat_selector.support_names_)) ==␣
↪relVar:
                c += 1
        print("---------- correlation = ", cor,"-------------")
```

```
        print("number of times that all relevant features were chosen: ",c,
↪"out of", N)
        print("error rate = ", 1-c/N)
        p.append(1-c/N)
    return(p)
```

B.5 Helper Class

This class was developed in order to access the standard deviation of variable importance across folds in a concise way. The plot method automatically runs the GrootCV on the data set, prints the boxplots and calculates the standard deviations.(this is achieved with the *object.cv_df* attribute which consists of a data frame of scores.)

```
class groothelper:

    def __init__(self, X, y):

        # data sets
        self.X = X
        self.y = y

        # spread of scores
        self.s0 = None
        self.s1 = None
        self.s2 = None
        self.s3 = None
        self.s4 = None
        self.s5 = None

    def Plot(self):

        #groot fit
        feat_selector = arfsgroot.GrootCV(objective='rmse', cutoff = 1, n_
↪folds=5, n_iter=5)
        feat_selector.fit(self.X, self.y, sample_weight=None)

        print("Relevant:")
        print(feat_selector.support_names_)
        fig = feat_selector.plot_importance(n_feat_per_inch=5)
        print("")
        # print median
        print("Median Values:")
        print(feat_selector.cv_df[['feature','Med']][:6])
        # get standard deviation of scores
        self.s0 = np.std(np.array(feat_selector.cv_df.iloc[:1,2:26]))
```

```
        self.s1 = np.std(np.array(feat_selector.cv_df.iloc[1:2,2:26]))
        self.s2 = np.std(np.array(feat_selector.cv_df.iloc[2:3,2:26]))
        self.s3 = np.std(np.array(feat_selector.cv_df.iloc[3:4,2:26]))
        self.s4 = np.std(np.array(feat_selector.cv_df.iloc[4:5,2:26]))
        self.s5 = np.std(np.array(feat_selector.cv_df.iloc[5:6,2:26]))


        # highlight synthetic random variable
        fig = highlight_tick(figure=fig, str_match='random')
        fig = highlight_tick(figure=fig, str_match='genuine', color='green')
        plt.show()
```

B.6 Stability and Noise

This is equivalent to the *grootRepeatCor* but it iterates over the different values of standard deviation of the noise term.

```
def grootRepeatNoise(size,w1,w2,w3,w4,w5,target,case,cor,N=5,
                             SD=[i*0.4 for i in range(0,24)]):
    c = 0
    relVar = set(['var0','var1','var2','var3','var4','var5'])
    np.random.seed(42)
    N = N
    p = []
    SD = SD
    for sd in SD:
        c = 0
        for i in range(N):
            # create data set
            X,y = generate_corr_dataset_
↪regr(w1=w1,w2=w2,w3=w3,w4=w4,w5=w5,target=target,noise=True,
                                                    ␣
↪size=size,cor=cor,case=case,sd=sd)
            #groot
            feat_selector = arfsgroot.GrootCV(objective='rmse', cutoff = 1,␣
↪n_folds=5, n_iter=5)
            feat_selector.fit(X, y, sample_weight=None)
            if relVar.intersection(set(feat_selector.support_names_)) ==␣
↪relVar:
                c += 1
        print("----------- standard deviation of noise = ", sd,"------------
↪-")
        print("number of times that all relevant features were chosen: ",c,
↪"out of 10")
        print("error rate = ", 1-c/N)
```

```
        p.append(1-c/N)
    return(p)
```

B.7 Generate multiple variables

The purpose of the function below was to gain some control over the number of features of our data set. To this end, we generate n relevant correlated variables and n irrelevant ones. The correlation is achieved with the multivariate normal distribution and the covariance matrix that we defined in an earlier section. The linear model is used as the target-predictor relationship in which the coefficients are equal and add up to 1.

```
def generate_more_var(size, n, cor):

    """This function generates a data set with n relevant and
    n irrelevant variables. All the relevant variables have a
    correlation. Parameters:
    size : number of rows
    n : number of columns/2
    cor : correlation strength"""

    # first generate mean
    mean1 = [i for i in range(n)]
    # covariance matrix
    a = np.ones(n) - cor
    m = np.diag(a,0) + np.ones((n,n))*cor
    cov1 = (1/(1+(n-1)*cor**2))*np.dot(m, m.transpose())

    # initiate data matrix
    X1 = np.zeros((size,n))
    # fill its contents:
    y = 0
    #relevant and correlated
    for i in range(n):
        # set seed so that the columns are the same
        np.random.seed(2)
        # fill matrix
        X1[:,i] =  np.random.multivariate_normal(mean = mean1,
            cov = cov1, size=size).T[i]

        y += (1/n)*X1[:,i]

    # make  it a pandas DF
    column_names = ['var' + str(i) for i in range(n)]
    X = pd.DataFrame(X1)
    X.columns = column_names
```

```
    #
    # irrelevant variables
    X2 = np.zeros((size,n))
    mean2 = np.ones(n)
    cov2 = np.diag(a,0)
    for i in range(n):
        # set seed so that the columns are the same
        np.random.seed(42)
        # fill matrix
        X2[:,i] =  np.random.multivariate_normal(mean = mean2,
        cov = cov2, size=size).T[i]

    # make  it a pandas DF
    column_names = ['var' + str(i) for i in range(n,2*n)]
    X2 = pd.DataFrame(X2)
    X2.columns = column_names

    X = pd.concat([X, X2], axis = 1)

    return X, y
```

B.8 Outliers

```
def generate_outliers(size, out=True):

    # relevant variables
    mean = [0, 1]
    m = np.array([[1, 0], [0, 1]])
    #cov = np.array([[1,cor],[cor,1]])
    x0, x1 = np.random.multivariate_normal(mean, cov, size=size).T
    x2 = np.random.gamma(1, 1, size)
    x3 = np.random.normal(1,1,size)
    x4 = np.random.normal(-1,1,size)
    x5 = np.random.gamma(1,1,size)
    X = np.zeros((size,11))


    # 6 relevant features
    X[:, 0] = x0
    X[:,1] = x1
    X[:,2] = x2
    X[:,3] = x3
    X[:,4] = x4
    X[:,5] = x5
```

```
    # 5 irrelevant ones
    X[:,6] = np.random.normal(-10,1,size)
    X[:,7] = np.random.gamma(4,1,size)
    X[:,8] = np.random.uniform(-12,1,size)
    X[:,9] = np.random.uniform(5,1,size)
    X[:,10] = np.random.normal(4,1,size)


    # make  it a pandas DF
    column_names = ['var' + str(i) for i in range(11)]
    X = pd.DataFrame(X)
    X.columns = column_names

    if out==True:
        # outliers
        y = 0.4*np.sqrt(abs(x0)) + 0.2*np.sin(x1) + 0.1*x2 + 0.1*np.cos(x3)␣
↪+ 0.05*x4 + 0.05*np.tan(x5)
    else:
        y = 0.4*np.sqrt(abs(x0)) + 0.2*np.sin(x1) + 0.1*x2 + 0.1*np.cos(x3)␣
↪+ 0.05*x4 + 0.05*x5


    return X, y
```

References

[1] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[2] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics. Springer New York, 2014.

[3] Gianluca Bontempi. *"Statistical foundations of machine learning" (2nd edition) handbook*. 02 2021.

[4] z_ai. Random forest explained. https://towardsdatascience.com/random-forest-explained-7eae084f3ebe.

[5] Cheng Li. A gentle introduction to gradient boosting. *URL: http://www. ccs. neu. edu/home/vip/teach/MLcourse/4_ boosting/slides/gradient_boosting. pdf*, 2016.

[6] Samuele Mazzanti. Mrmr explained exactly how you wished someone explained it to you. https://towardsdatascience.com/mrmr-explained-exactly-how-you-wished-someone-explained-to-you-9cf4ed27458b.

[7] Witold Rudnicki, Mariusz Wrzesień, and Wieslaw Paja. All relevant feature selection methods and applications. *Studies in Computational Intelligence*, 584:11–28, 12 2015.

[8] Samuele Mazzanti. Boruta explained the way i wish someone explained it to me. https://towardsdatascience.com/boruta-explained-the-way-i-wish-someone-explained-it-to-me-4489d70e154a.

[9] Thomas Bury. All relevant feature selection. https://pypi.org/project/arfs/.

[10] C. Molnar. *Interpretable Machine Learning*. Lulu.com, 2020.

[11] Dimitrios Kottoras. arfs-tests. https://github.com/kottorov/arfs-tests.