Report from a machine learning study: **Software bugs prediction based on selected object-oriented code metrics**

Authors:

Volodymyr Havryliuk
Marcin Rabiza

Poznan, Jan 2020

Table of contents

Objective **2**

Introduction **2**

Data inspection **3**

Correlation Analysis **4**

Introduction to Prediction **7**

K-Nearest Neighbours Regression ***8***

Random Forest Regression **11**

Extreme Gradient Boosting Regression **14**

Caret ANN Regression **20**

Tensorflow/Keras ANN Regression **24**

Results Discussion and Conclusions **28**

Validity Threats **30**

References **32**

Objective

The aim of this study is to determine whether there is a statistical correlation between various code properties such as the number of lines of code or the number of class methods and the number of bugs (post-released defects) found. Furthermore, we try to investigate how significant impact on the number of bugs those properties have. The analysis is based on a purely statistical and analytical approach with numerous visualizations. Results are obtained by training and testing several regression models.

Introduction

In order to conduct the research we have used already prepared dataset presented in [1]. This bug prediction dataset is a collection of models and metrics of software systems and their histories. The dataset is designed to perform bug prediction at the class level. However package or subsystem information can be derived by aggregating class data, since per each class it is specified the package that contains it.

We have decided to use data gathered for Eclipse JDT Core software system, which is the Java infrastructure of the Java IDE. In particular, the "CK" (Chidamber-Kemerer metrics suite, see [2]) along with 11 other object-oriented metrics were downloaded[1]. Among them, we have decided to investigate the following metrics:

CBO	Coupling Between Objects - indicates the dependency degree of a class by another one
DIT	Depth of Inheritance Tree - indicates the depth (i.e., the length of the maximal path from the node representing the class to the root of the tree) of the class in the inheritance tree. Deeper trees constitute greater design complexity, since more methods and classes are involved.
FanIn	The number of other classes that reference a class.
FanOut	The number of other classes referenced by a class.
LCOM	Lack of Cohesion in Methods - indicates the level of cohesion between methods and attributes of a class.
NOC	Number of Classes - computes the number of classes within the app package.
RFC	Response For Class - measures the complexity of the class in terms of method calls.
WMC	Weighted Methods per Class - complexity metric introduced by Chidamber and Kemerer. The WMC metric is the sum of the complexities of all class methods.
LOC	The number of Lines Of Code.
Attributes	Number of attributes.

[1] Direct link to the dataset in CSV format: http://bug.inf.usi.ch/data/eclipse/single-version-ck-oo.csv

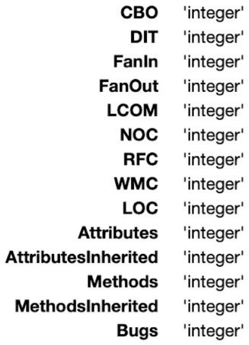
AttributesInherited	Number of attributes inherited.
Methods	Number of methods.
MethodsInherited	Number of methods inherited.
Bugs	Number of bugs (post-release defects).

For further explanations see: [3] and [4].

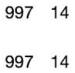
For data manipulation and regression purposes we have used mostly R language with particular ML libraries as well as Python technologies such as Tensorflow and Keras.

Data inspection

After initialization of the environment (see *initialization.r* script) the data was acquired from *bugs.csv* file. We can check the type of the columns and dimensions of the dataset.



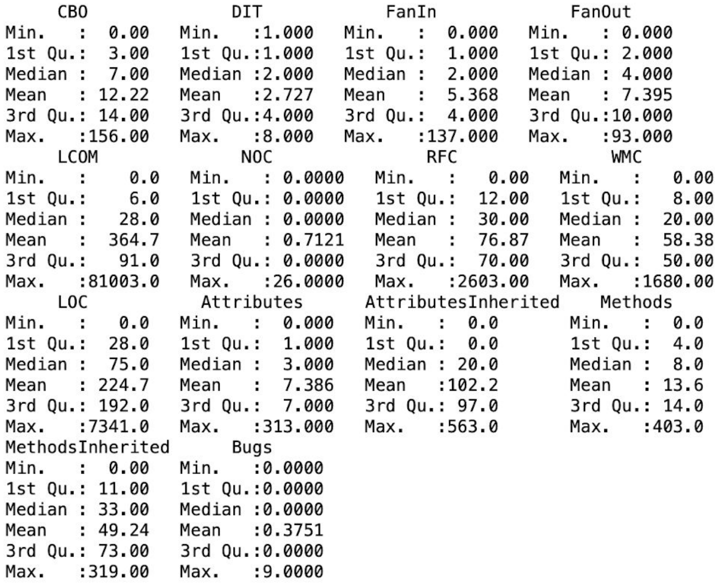
```
                CBO  'integer'
                DIT  'integer'
              FanIn  'integer'
             FanOut  'integer'
               LCOM  'integer'
                NOC  'integer'
                RFC  'integer'
                WMC  'integer'
                LOC  'integer'
         Attributes  'integer'
AttributesInherited  'integer'
            Methods  'integer'
   MethodsInherited  'integer'
               Bugs  'integer'
```



```
997  14

997  14
```

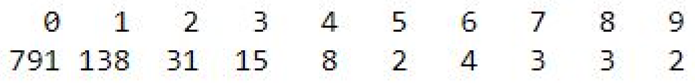
As one may observe, all of the attributes are integers and there were no missing values found, so no further preprocessing at this stage of the study was needed.

The one can inspect basic information about the data with *summary*. We can see that there are 0-9 bugs found per class/package.



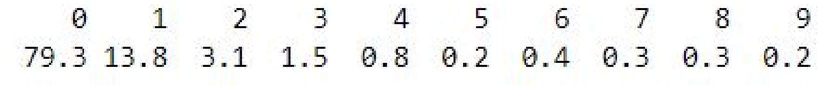
```
      CBO              DIT             FanIn             FanOut     
 Min.   :  0.00   Min.   :1.000   Min.   :  0.000   Min.   : 0.000  
 1st Qu.:  3.00   1st Qu.:1.000   1st Qu.:  1.000   1st Qu.: 2.000  
 Median :  7.00   Median :2.000   Median :  2.000   Median : 4.000  
 Mean   : 12.22   Mean   :2.727   Mean   :  5.368   Mean   : 7.395  
 3rd Qu.: 14.00   3rd Qu.:4.000   3rd Qu.:  4.000   3rd Qu.:10.000  
 Max.   :156.00   Max.   :8.000   Max.   :137.000   Max.   :93.000  
      LCOM              NOC               RFC               WMC         
 Min.   :    0.0   Min.   : 0.0000   Min.   :   0.00   Min.   :   0.00  
 1st Qu.:    6.0   1st Qu.: 0.0000   1st Qu.:  12.00   1st Qu.:   8.00  
 Median :   28.0   Median : 0.0000   Median :  30.00   Median :  20.00  
 Mean   :  364.7   Mean   : 0.7121   Mean   :  76.87   Mean   :  58.38  
 3rd Qu.:   91.0   3rd Qu.: 0.0000   3rd Qu.:  70.00   3rd Qu.:  50.00  
 Max.   :81003.0   Max.   :26.0000   Max.   :2603.00   Max.   :1680.00  
      LOC            Attributes      AttributesInherited    Methods     
 Min.   :   0.0   Min.   :  0.000   Min.   :  0.0       Min.   :  0.0  
 1st Qu.:  28.0   1st Qu.:  1.000   1st Qu.:  0.0       1st Qu.:  4.0  
 Median :  75.0   Median :  3.000   Median : 20.0       Median :  8.0  
 Mean   : 224.7   Mean   :  7.386   Mean   :102.2       Mean   : 13.6  
 3rd Qu.: 192.0   3rd Qu.:  7.000   3rd Qu.: 97.0       3rd Qu.: 14.0  
 Max.   :7341.0   Max.   :313.000   Max.   :563.0       Max.   :403.0  
 MethodsInherited      Bugs       
 Min.   :  0.00   Min.   :0.0000  
 1st Qu.: 11.00   1st Qu.:0.0000  
 Median : 33.00   Median :0.0000  
 Mean   : 49.24   Mean   :0.3751  
 3rd Qu.: 73.00   3rd Qu.:0.0000  
 Max.   :319.00   Max.   :9.0000  
```

It is possible to observe that there are 791 classes without bugs and two classes with nine bugs.



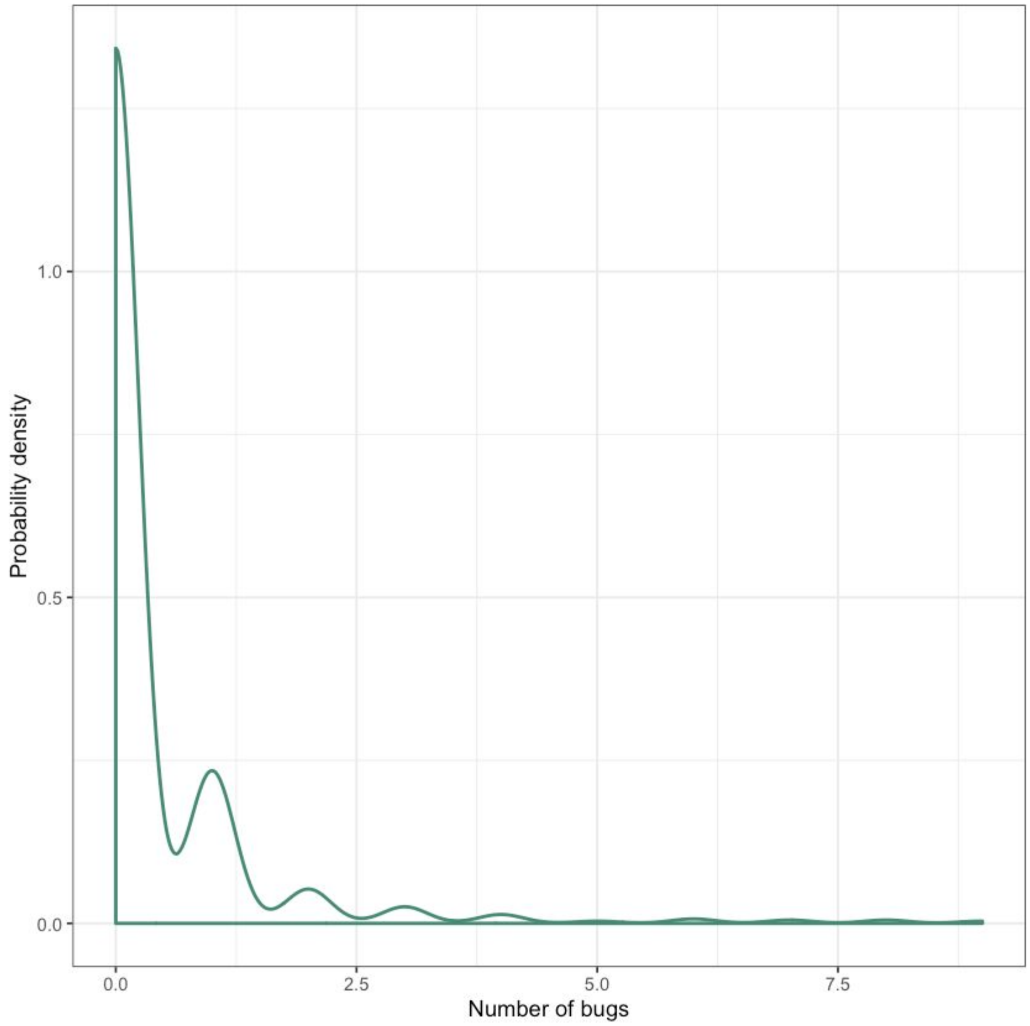
```
  0   1   2   3   4   5   6   7   8   9 
791 138  31  15   8   2   4   3   3   2 
```

This information can be represented by percentage. The one can see that 79.3 percent of examples have no bugs, while 0.2 percent of examples have 9 bugs.



```
   0    1    2    3    4    5    6    7    8    9 
79.3 13.8  3.1  1.5  0.8  0.2  0.4  0.3  0.3  0.2 
```

As probability density chart implies, the most probable number of bugs is 0.

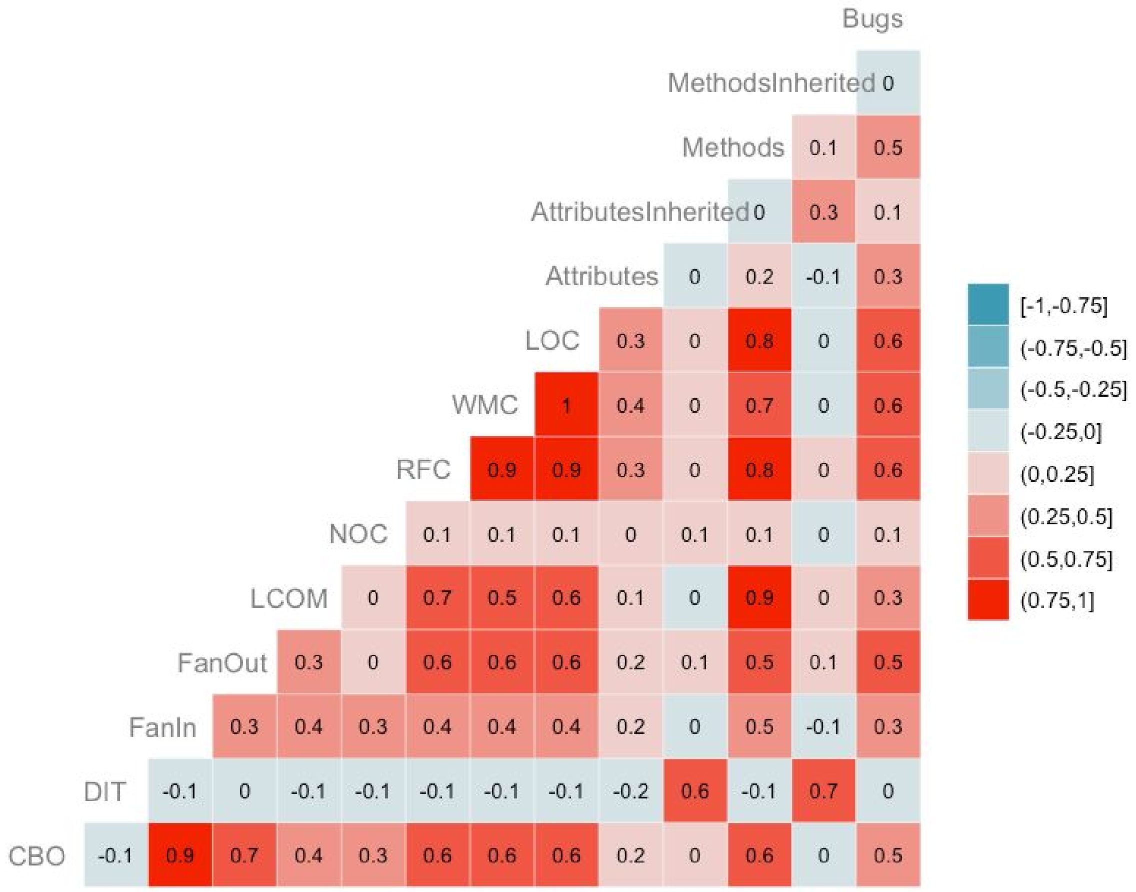


Correlation Analysis

Correlation is about the linear relationship between two variables. Usually, both are continuous (or nearly so). Due to the fact that our data are entirely quantitative, we can perform such a correlation analysis with Pearson coefficient. There is no need for performing chi-square test of independence, which is suited more for categorical variables[2].

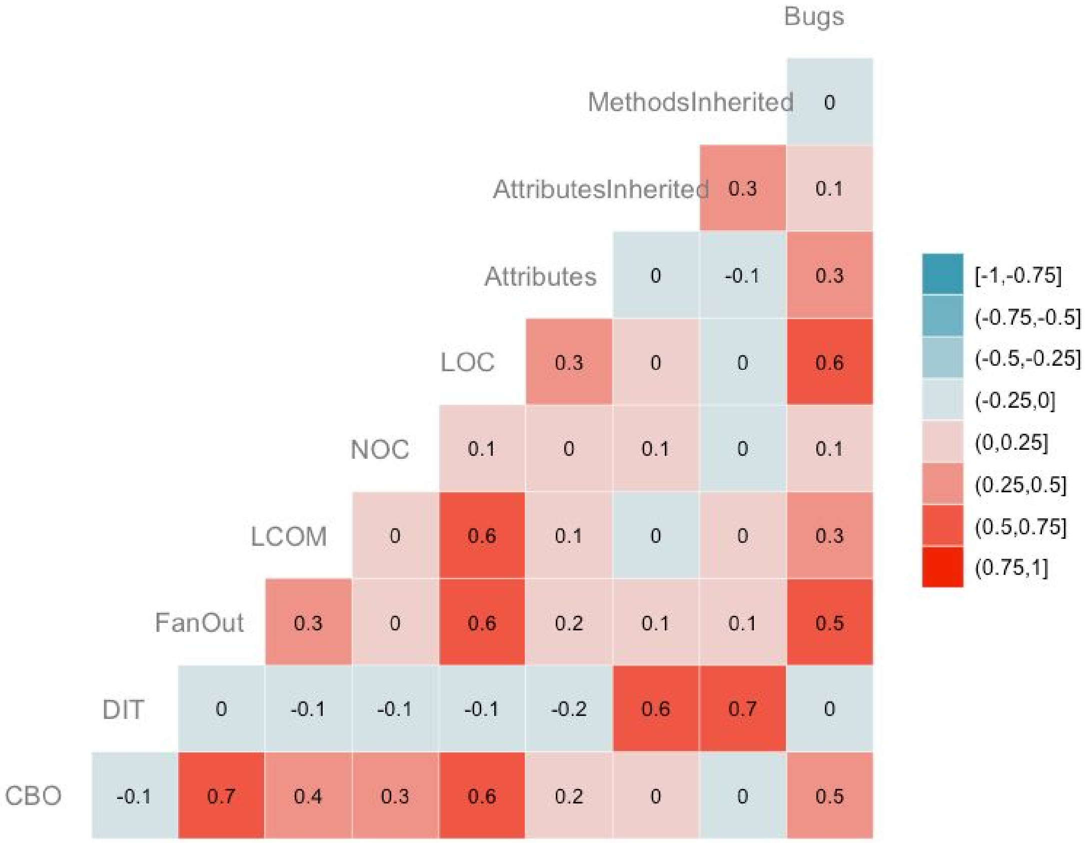
[2] It should be noted that two variables can be correlated and have not the slightest causal relationship and correlation does not measure all relationships, just linear ones (either on the quantities themselves (Pearson) or their ranks (Spearman)).

For performing correlation analysis for all of attributes *ggcor* function was used.

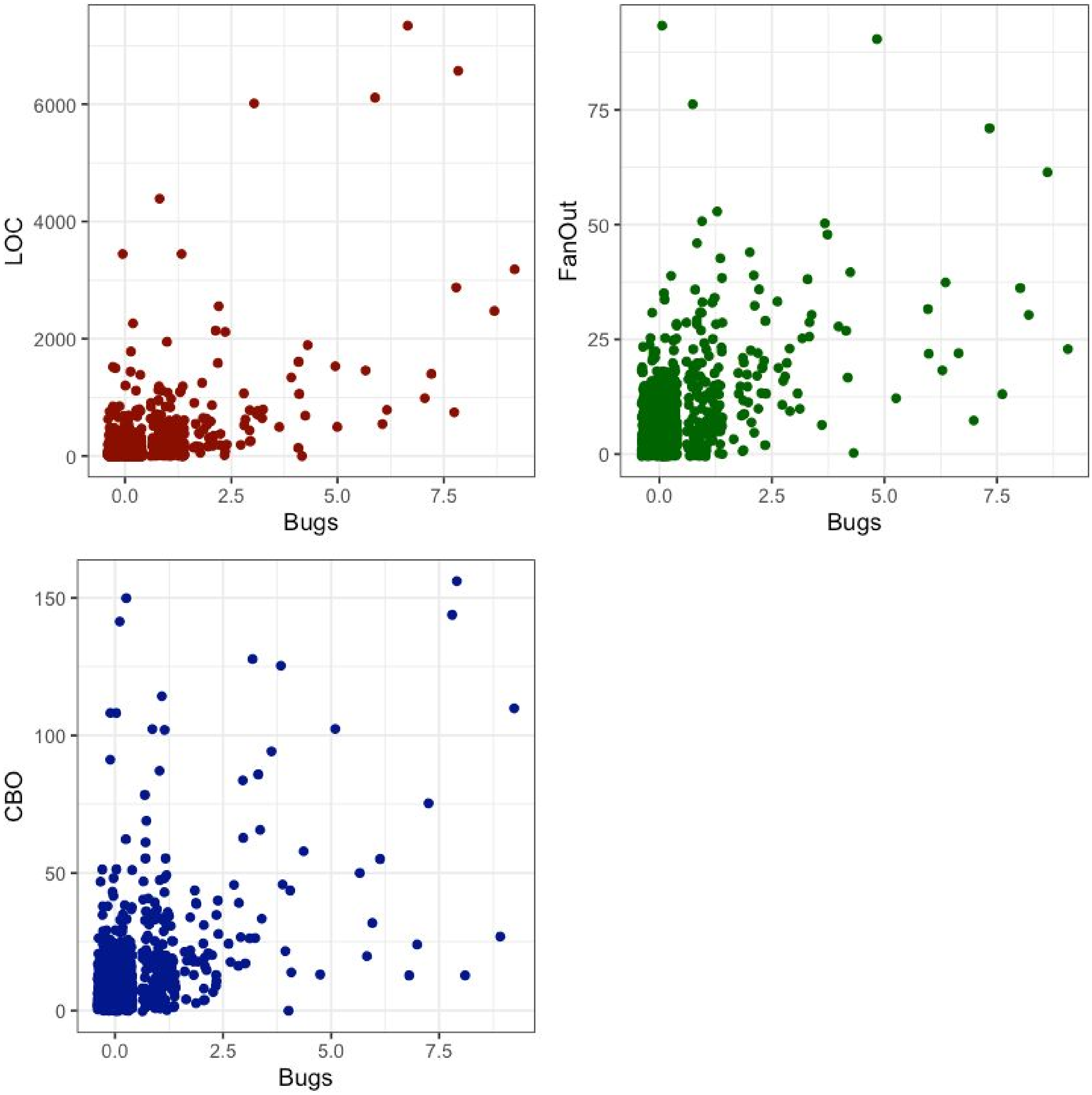


Inside colored boxes we can see Pearson's *r* coefficient value. It can be seen that the number of bugs is in moderate uphill (positive) linear relationship mostly with **LOC** (number of Lines of Code), **WMC** (Weighted Methods per Class) and **RFC** (Response For Class) attributes, even though the correlation is not that strong (ex. 0.70+). What is more the number of bugs is in weaker positive linear relationship with number of class methods, **FanOut** (the number of other classes referenced by a class) and **CBO** (Coupling Between Objects). There seems not to be any linear relationship between the number of bugs and number of methods inherited as well as DIT(Depth of Inheritance Tree). These variables seem to be redundant in next stages of the analysis.

Interestingly, there is very strong linear correlation between LOC and WMC and also between WMC and RFC, LOC and RFC, CBO and FanIn, LCOM and the number of Methods. In order to achive possible better results we have decided to delete highly correlated attributes such as WMC and RFC (LOC correlation coeff. >= 0.9), as well as Methods (LOC and LCOM correlation coeff. >= 0.8) and FanIn (CBO correlation coeff. >= 0.9). Attributes were picked in such a manner to operate on most basic, important or easy to gather metrics – taking LOC as an example. Correlation matrix without strongly correlated attributes is shown below.



Let's see how the relationship between the most promising variables and the number of bugs.



However, it should be noted that FanOut and CBO variables are still significantly correlated with correlation coefficient of about 0.7.

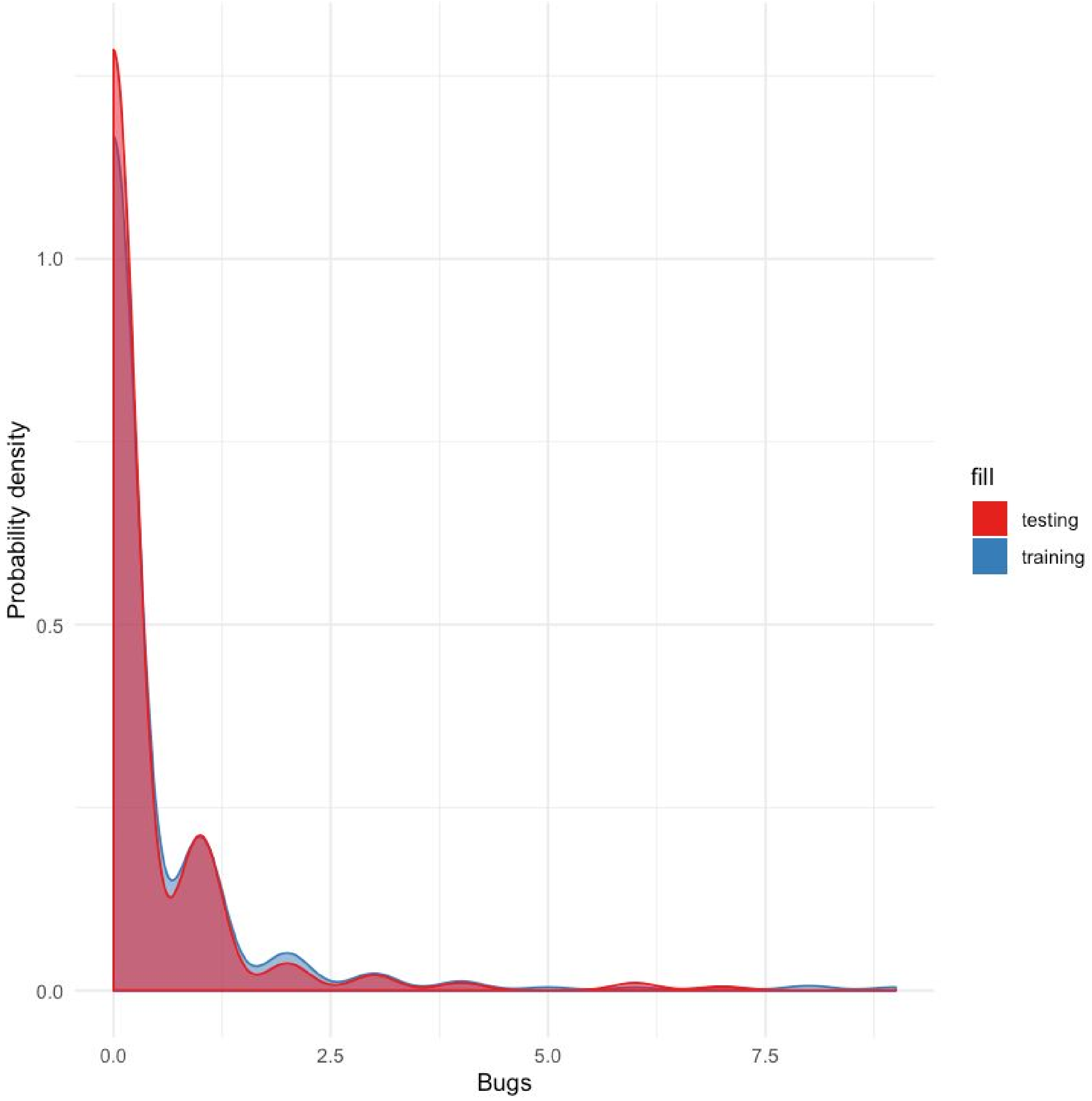
Introduction to Prediction

For prediction purposes, we can use R language *caret* library[3]. Let's first divide the data set into training and testing subsets in 7:3 ratio. The one can observe that subsets overlap more or less well.

In the following part, various predictions will be made by means of several regressors.

[3] *Caret* is a R package which provides general interface to nearly 150 ML algorithms. It also provides some methods to sample the data (for training and testing), preprocessing, evaluating the model etc. To get familiar with *caret* package, check following URLs:

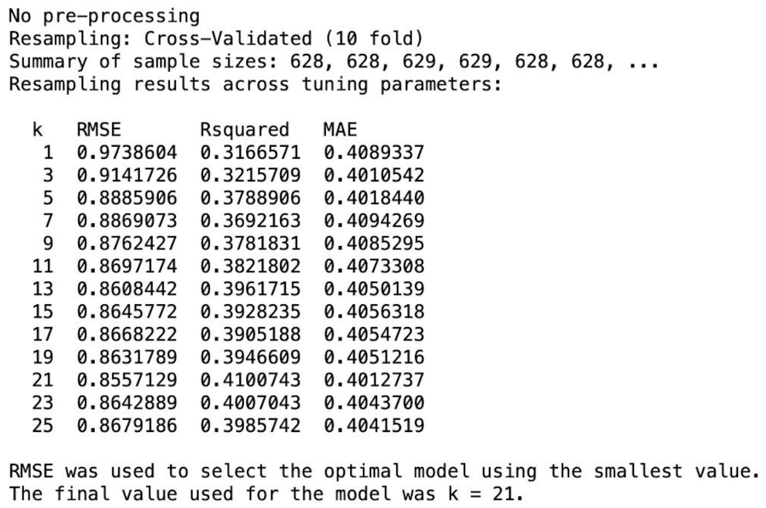
- https://www.youtube.com/watch?v=7Jbb2ItbTC4
- http://caret.r-forge.r-project.org
- http://cran.r-project.org/web/packages/caret/vignettes/caret.pdf



K-Nearest Neighbours Regression

K-nearest neighbors algorithm (*k*-NN) is a non-parametric method used for classification as well as regression. Since we are using *k*-NN algorithm for the latter, we expect the output to be the object property value, which is the average of the values of *k* nearest neighbors.

K-NN model was trained using a grid of 1 to 25 (only odd numbers) neighbours and 10-fold cross-validation. The model summary can be seen below.



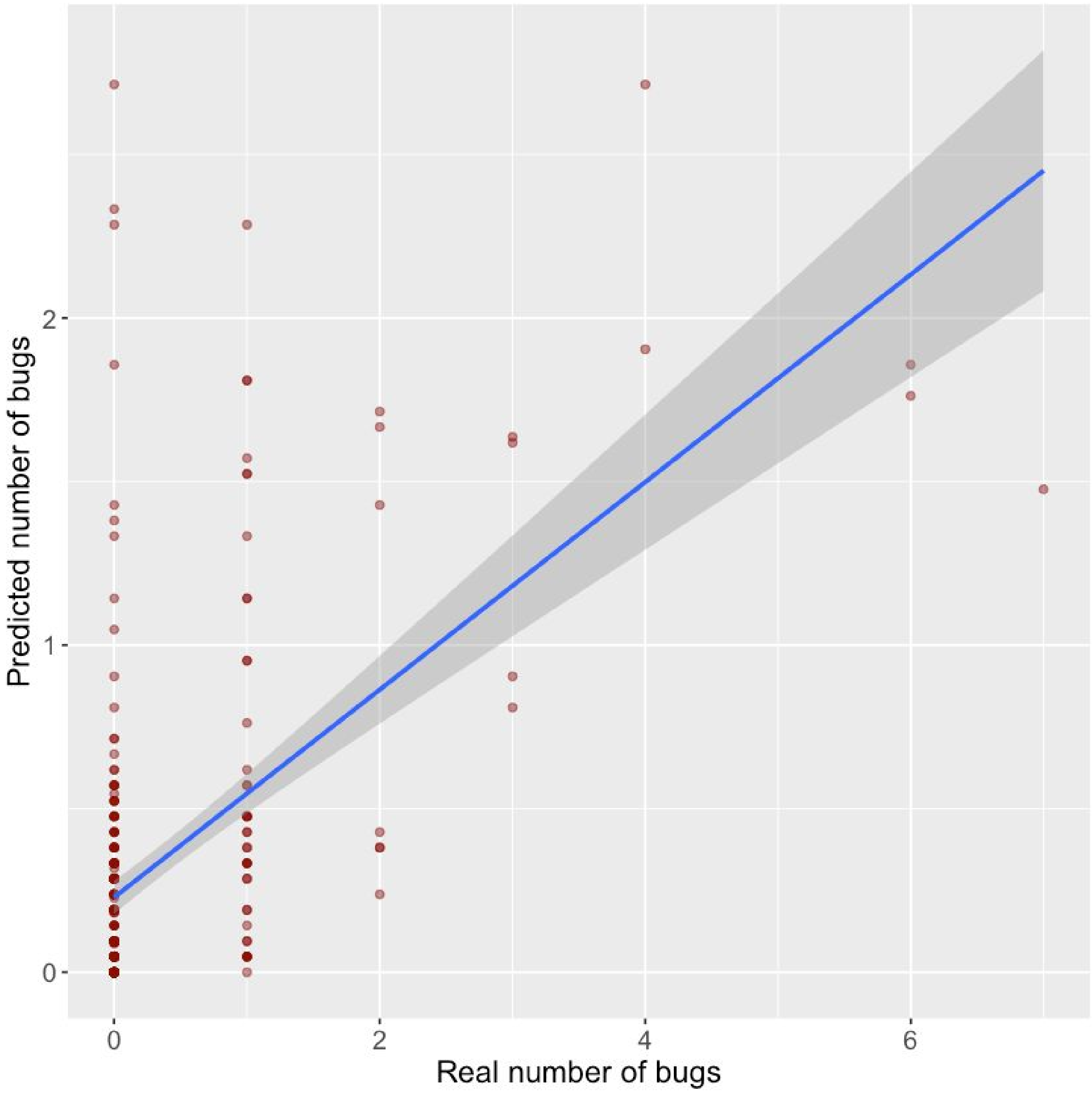
```
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 628, 628, 629, 629, 628, 628, ...
Resampling results across tuning parameters:

  k   RMSE       Rsquared   MAE
   1  0.9738604  0.3166571  0.4089337
   3  0.9141726  0.3215709  0.4010542
   5  0.8885906  0.3788906  0.4018440
   7  0.8869073  0.3692163  0.4094269
   9  0.8762427  0.3781831  0.4085295
  11  0.8697174  0.3821802  0.4073308
  13  0.8608442  0.3961715  0.4050139
  15  0.8645772  0.3928235  0.4056318
  17  0.8668222  0.3905188  0.4054723
  19  0.8631789  0.3946609  0.4051216
  21  0.8557129  0.4100743  0.4012737
  23  0.8642889  0.4007043  0.4043700
  25  0.8679186  0.3985742  0.4041519

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was k = 21.
```

As we can see, an optimal value for the model k=21 was found using smallest value of RMSE at the 11th iteration. This calculation was performed for the training data. We can evaluate our model by calculating RMSE (root mean squared error) and R^2 (coefficient of determination)[4] for the testing data. The results can be seen in the table below.

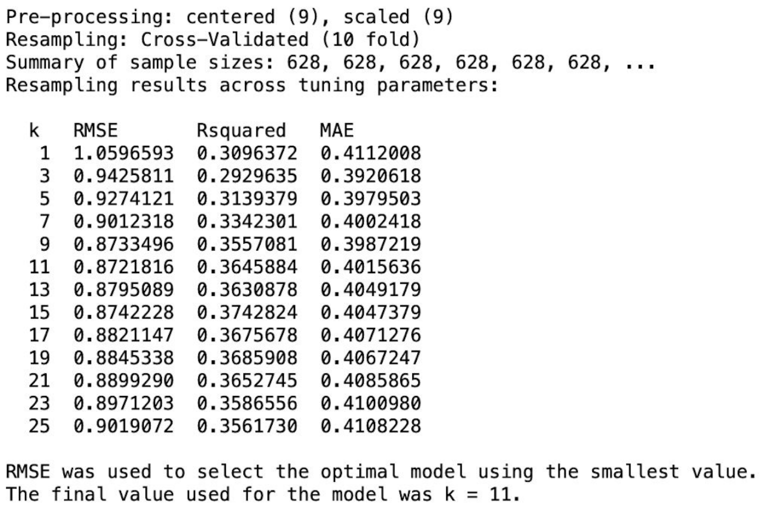
RMSE	**R^2**
0.728	0.306

Prediction chart is presented below.

[4] R-squared (R2) is a statistical measure that represents the proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model. Whereas correlation explains the strength of the relationship between an independent and dependent variable, R-squared explains to what extent the variance of one variable explains the variance of the second variable. So, if the R2 of a model is 0.50, then approximately half of the observed variation can be explained by the model's inputs [5].



K-NN predictor works best on variables that are normalized or scaled. Since *caret* package provides suitable facility to preprocess data, we will repeat model training procedure with basic data preprocessing: **centering** and **scaling**. The model summary can be seen below.

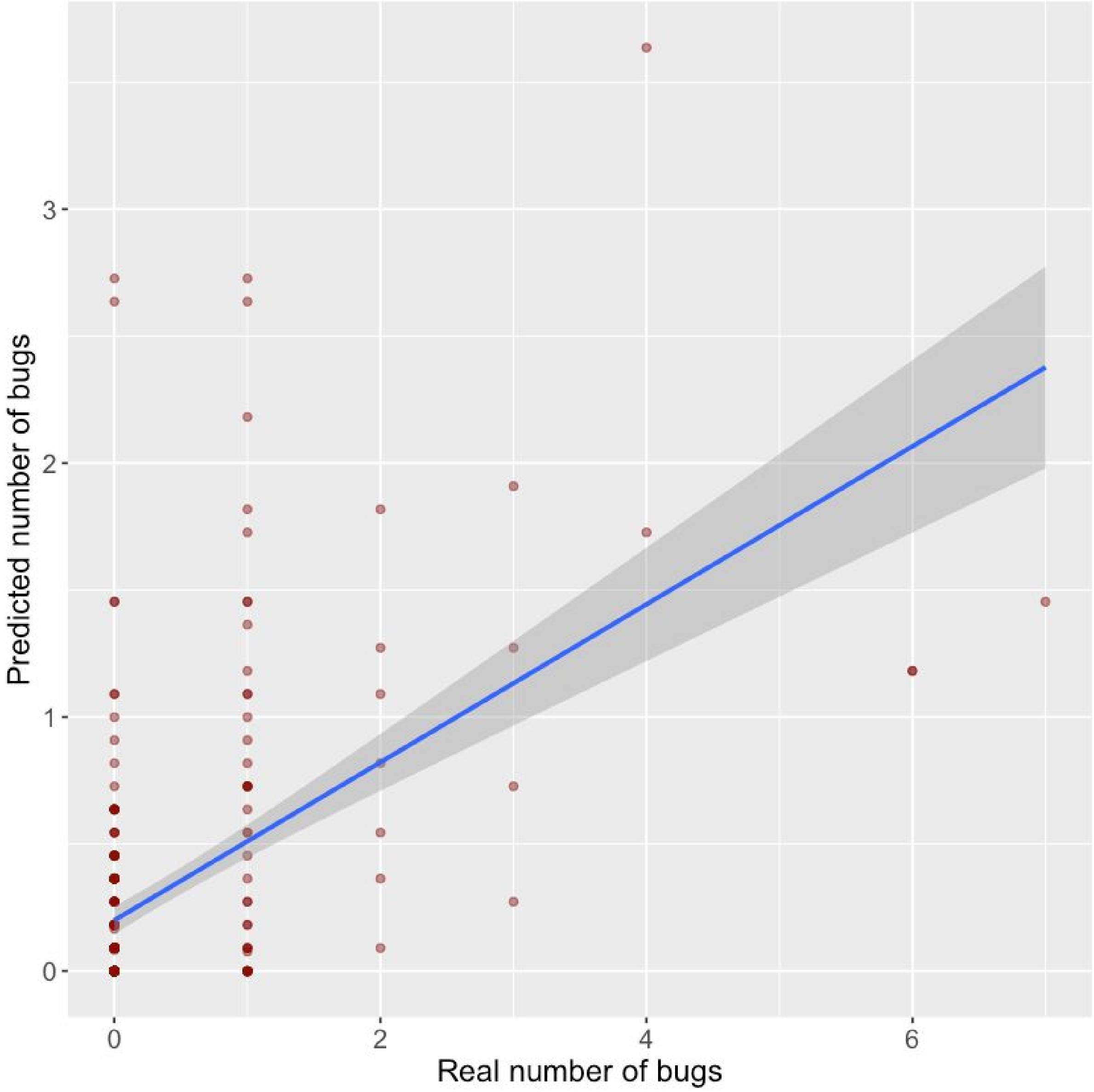


```
Pre-processing: centered (9), scaled (9)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 628, 628, 628, 628, 628, 628, ...
Resampling results across tuning parameters:

  k   RMSE       Rsquared   MAE
   1  1.0596593  0.3096372  0.4112008
   3  0.9425811  0.2929635  0.3920618
   5  0.9274121  0.3139379  0.3979503
   7  0.9012318  0.3342301  0.4002418
   9  0.8733496  0.3557081  0.3987219
  11  0.8721816  0.3645884  0.4015636
  13  0.8795089  0.3630878  0.4049179
  15  0.8742228  0.3742824  0.4047379
  17  0.8821147  0.3675678  0.4071276
  19  0.8845338  0.3685908  0.4067247
  21  0.8899290  0.3652745  0.4085865
  23  0.8971203  0.3586556  0.4100980
  25  0.9019072  0.3561730  0.4108228

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was k = 11.
```

Interestingly, with preprocessing performed optimal neighbours number resulted in k=11. Again, we can evaluate our model by calculating RMSE (root mean squared error) and R^2 (coefficient of determination) for the testing data. The results can be seen in the table below.

RMSE	**R^2**
0.752	0.258

It can be seen that with pre-processing performed we achieved bigger RMSE value and smaller R^2 this time. Prediction chart is presented below.



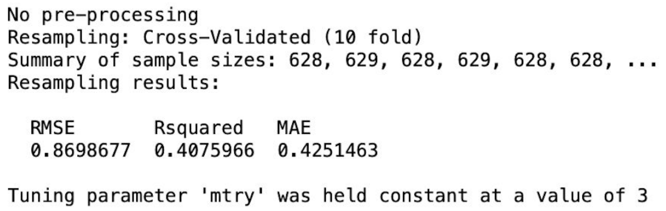
Random Forest Regression

We can now discuss machine learning technique using raining data to learn how to make predictions. Random forest basically is an ensemble of decision trees. One of the drawbacks of learning with a single tree is the problem of overfitting. Single trees tend to learn the training data too well, resulting in poor prediction performance on unseen data.

In case of random forests the algorithm works in two parts. The first step involves Bootstrapping technique for training and testing and second part involves decision tree for the prediction purpose. Now like every other predictive modelling technique we have the goal to minimize the generalization of error. To decrease that we make the balance between the bias and variance called as Bias-variance tradeoff. To have the lowest generalization of error we need to find the best tradeoff of bias and variance. If our decision tree are shallow then we have high bias and low variance and if our decision tree is too deep then it has low bias but high variance. Due to that, in random forest algorithm the idea of Bagging is very important. Bagging means Bootstrap aggregation. Bootstrap means generating random samples from the dataset with the replacement. The main reason of bagging is to reduce the variance of the model class. So it’s obvious that if we are using bagging then we are basically going for deep trees as they have the low variance. But the bias got increased. So we are not taking care of the bias.

As we pointed out, random forests use a variation of bagging whereby many independent trees are learned from the same training data. Such a forest typically contains several hundred trees. This way random forests fit many classification or regression tree models to random subsets of the input data and uses the combined result (the forest) for prediction, what is expected to correct decision trees' habit of overfitting to their training set.

We have started with creating a grid that contains the number of variables available for splitting at each tree node. In the random forests literature, this is referred to as the mtry parameter. According to the literature, we have chosen number of predictor variables divided by 3[5]. The training process of the model was analogous to the case of the *k*-NN regression models. The model summary can be seen below.



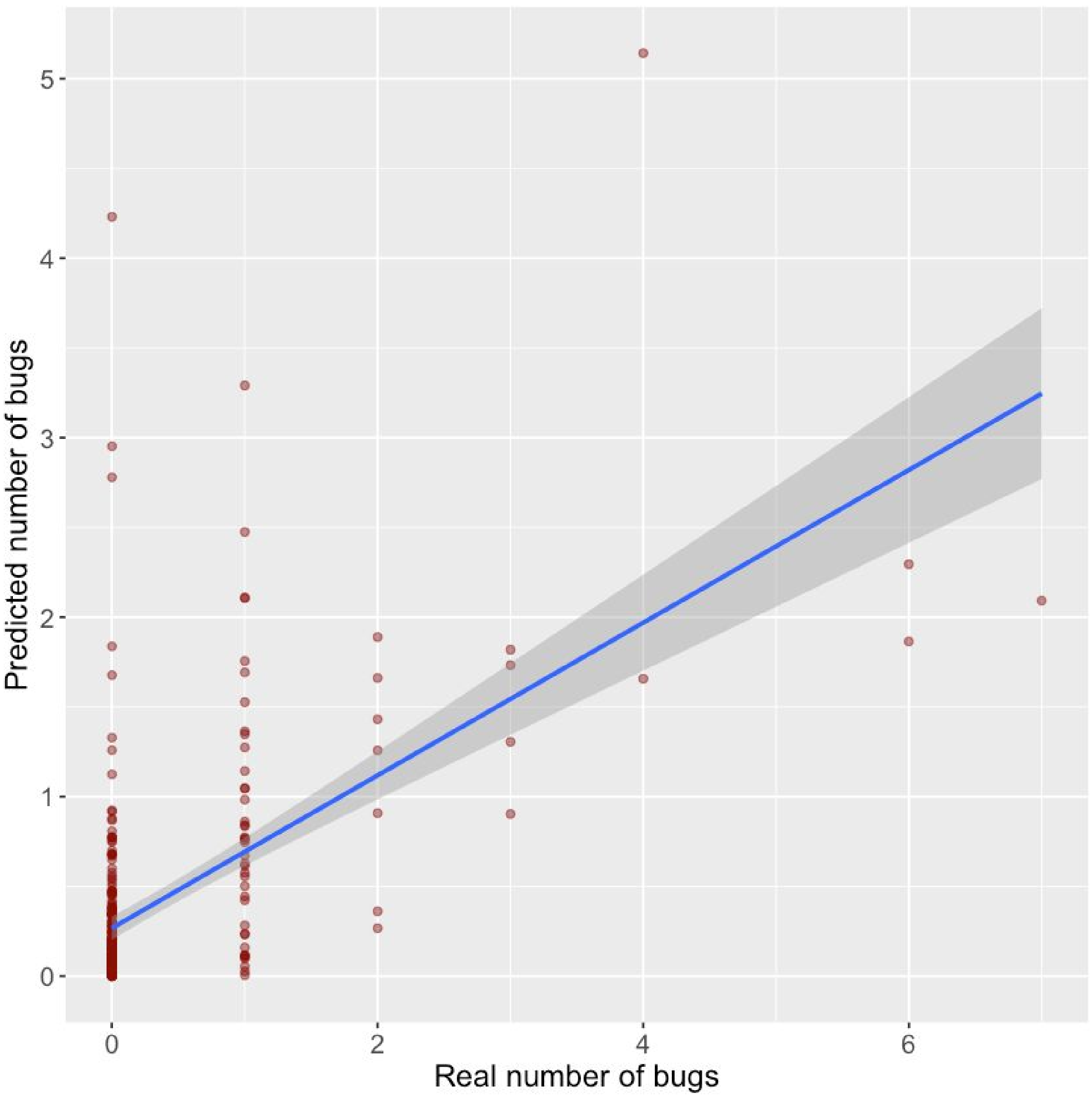
```
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 628, 629, 628, 629, 628, 628, ...
Resampling results:

  RMSE       Rsquared   MAE
  0.8698677  0.4075966  0.4251463

Tuning parameter 'mtry' was held constant at a value of 3
```

Such as in previous cases, we can evaluate our model by calculating RMSE (root mean squared error) and R^2 (coefficient of determination) for the testing data. The results can be seen in the table below.

RMSE	**R^2**
0.742	0.277

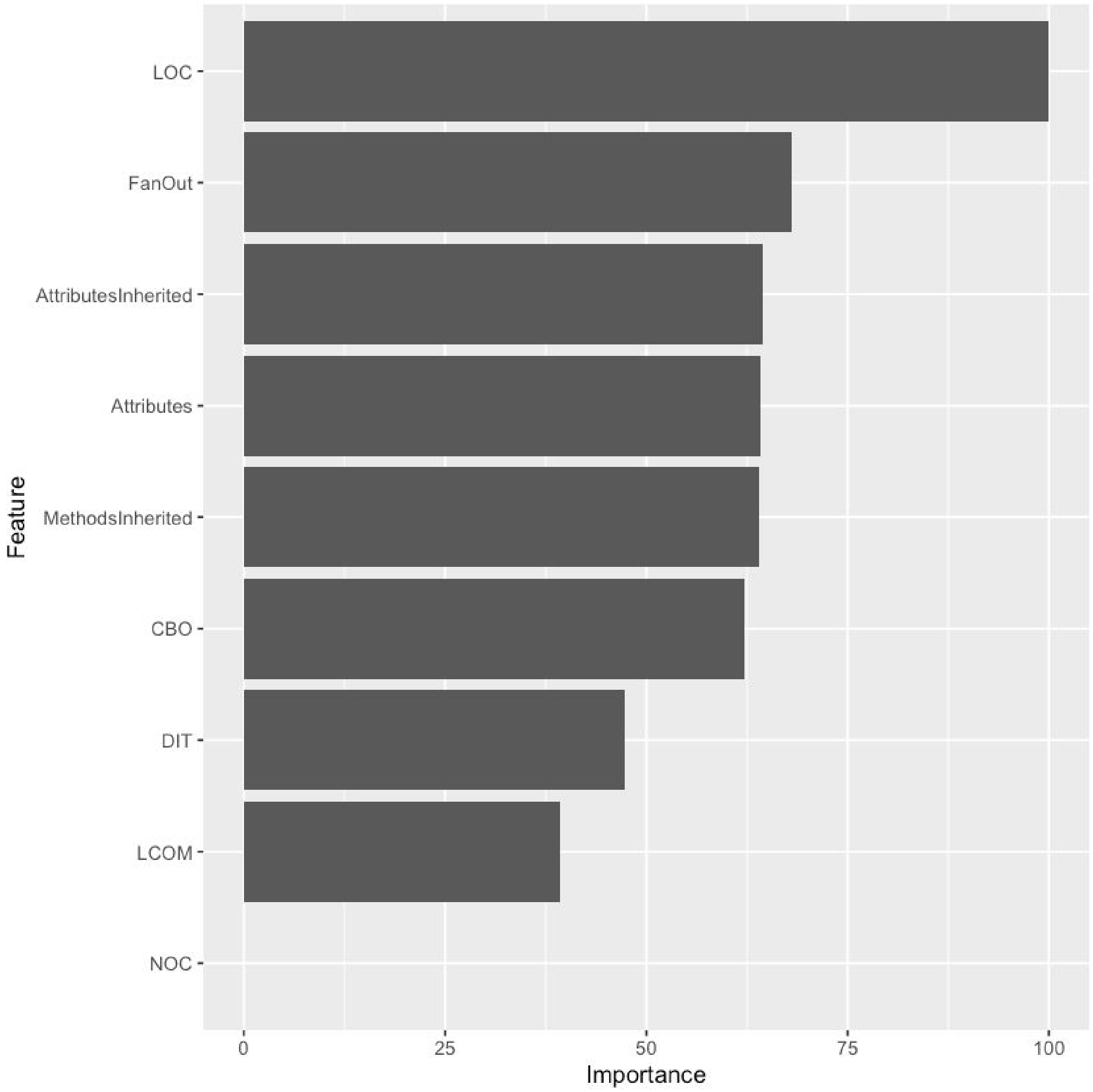
Prediction chart is presented below.

[5] “The default value of this parameter depends on which R package is used to fit the model. In terms of randomForest package for classification models, the default is the square root of the number of predictor variables (rounded down). For regression models, it is the number of predictor variables divided by 3 (rounded down)” [6].



After training a random forest, it is natural to ask which variables have the most predictive power. Variables with high importance are drivers of the outcome and their values have a significant impact on the outcome values. By contrast, variables with low importance might be omitted from a model, making it simpler and faster to fit and predict[6]. The importance plot generated using *varImp* generic method for calculating variable importance for objects produced by *train* and method-specific methods can be seen below.

[6] There are two measures of importance given for each variable in the random forest. The first measure is based on how much the accuracy decreases when the variable is excluded. This is further broken down by outcome class. The second measure is based on the decrease of Gini impurity (a measurement of the likelihood of an incorrect classification of a new instance of a random variable, if that new instance were randomly classified according to the distribution of class labels from the data set) when a variable is chosen to split a node. For more information see [7].



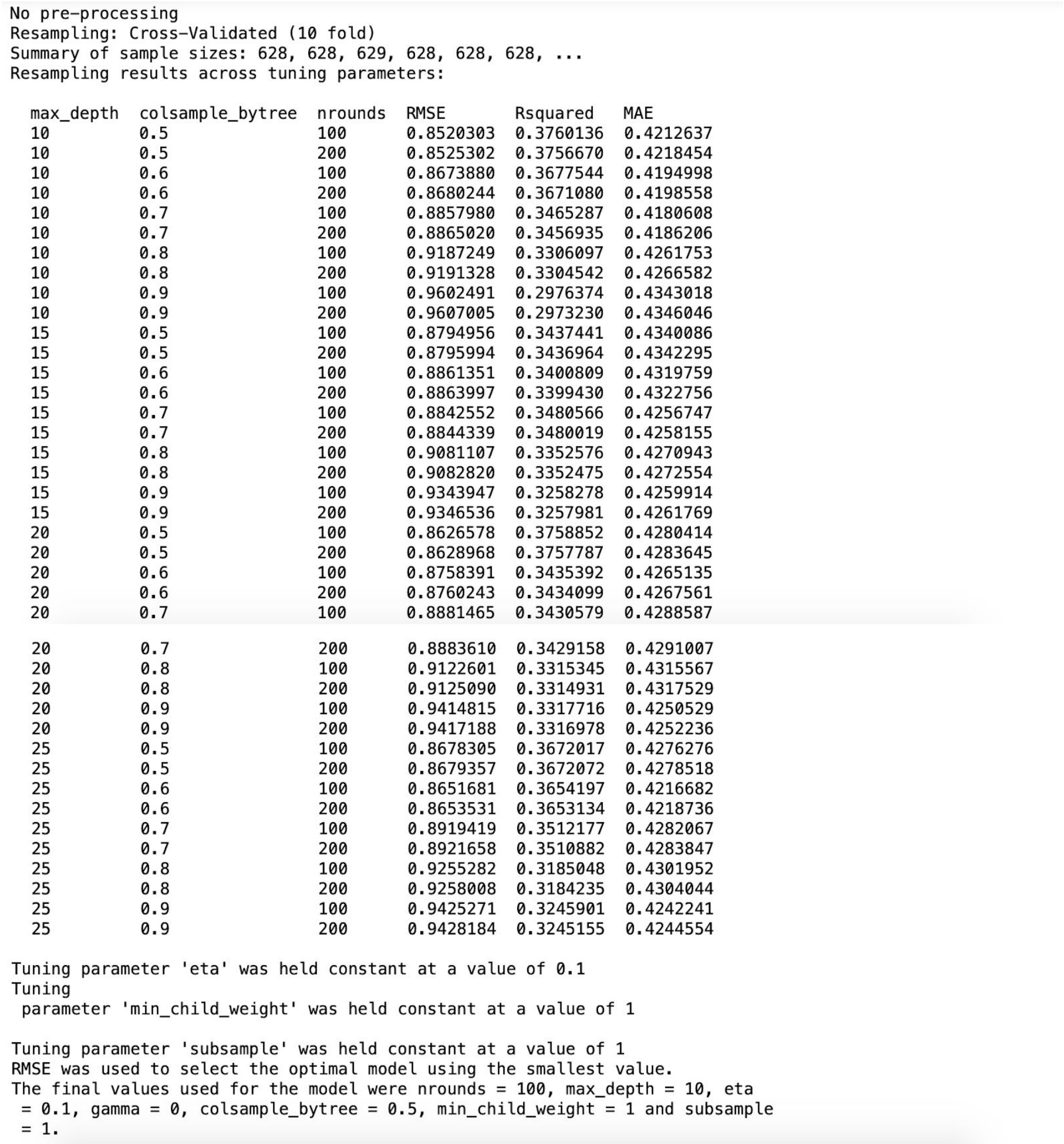
As the one may conclude, the plot confirms the accuracy of correlation analysis to some degree, since it points out our most promising attribute, which in this case is **LOC**, and the second one (but much less important) is FanOut. NOC attribute seems to be redundant in this analysis.

Extreme Gradient Boosting Regression

Gradient boosting is another popular machine learning algorithm which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. Due to that, the one may say it is a variation of a random forest algorithm, in which case one of the issues was associated with increasing bias. Now to take care of minimizing the bias we incorporate the idea of Boosting. Boosting itself nullifies the overfitting issue and it takes care of minimizing the bias. It helps to find a predictor which is a weighted average of all the models used. It manipulates the training set to work on the area where we find high errors. It adds new trees to the original trees and helps to achieve the maximum accuracy.

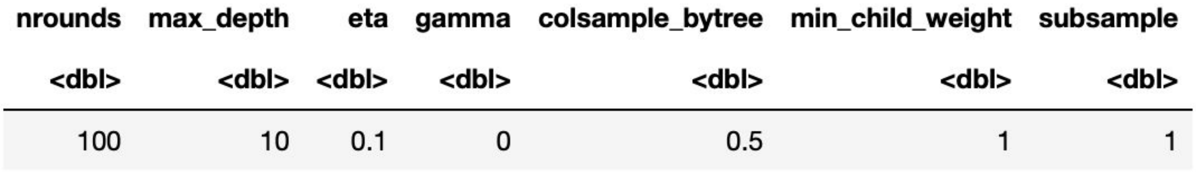
The difference between Gradient Boosting and Random Forest is that non only the boosting strategy for training takes care of the minimization of bias which the random forest lacks, but also it uses regression trees for prediction purpose where a random forest use decision tree. Whereas Random forests overfit a sample of the training data and then reduces the overfit by simple averaging the predictors, Gradient boosting repeatedly train trees on the residuals of the previous predictors.

For the sake of this study we use popular Gradient Boosting implementation called *XGBoost* (*Extreme Gradient Boosting*), which, according to some data scientist, has become the "state-of-the-art” machine learning algorithm to deal with structured data. The grid used in this case can be much more tailor, but we will stick to default values of the parameters (ex. *eta, gamma, min_child_weight*, *subsample*). The training process of the model was analogous to the case of previous regression models. The model summary can be seen below.



```
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 628, 628, 629, 628, 628, 628, ...
Resampling results across tuning parameters:

  max_depth  colsample_bytree  nrounds  RMSE       Rsquared   MAE
  10         0.5               100      0.8520303  0.3760136  0.4212637
  10         0.5               200      0.8525302  0.3756670  0.4218454
  10         0.6               100      0.8673880  0.3677544  0.4194998
  10         0.6               200      0.8680244  0.3671080  0.4198558
  10         0.7               100      0.8857980  0.3465287  0.4180608
  10         0.7               200      0.8865020  0.3456935  0.4186206
  10         0.8               100      0.9187249  0.3306097  0.4261753
  10         0.8               200      0.9191328  0.3304542  0.4266582
  10         0.9               100      0.9602491  0.2976374  0.4343018
  10         0.9               200      0.9607005  0.2973230  0.4346046
  15         0.5               100      0.8794956  0.3437441  0.4340086
  15         0.5               200      0.8795994  0.3436964  0.4342295
  15         0.6               100      0.8861351  0.3400809  0.4319759
  15         0.6               200      0.8863997  0.3399430  0.4322756
  15         0.7               100      0.8842552  0.3480566  0.4256747
  15         0.7               200      0.8844339  0.3480019  0.4258155
  15         0.8               100      0.9081107  0.3352576  0.4270943
  15         0.8               200      0.9082820  0.3352475  0.4272554
  15         0.9               100      0.9343947  0.3258278  0.4259914
  15         0.9               200      0.9346536  0.3257981  0.4261769
  20         0.5               100      0.8626578  0.3758852  0.4280414
  20         0.5               200      0.8628968  0.3757787  0.4283645
  20         0.6               100      0.8758391  0.3435392  0.4265135
  20         0.6               200      0.8760243  0.3434099  0.4267561
  20         0.7               100      0.8881465  0.3430579  0.4288587

  20         0.7               200      0.8883610  0.3429158  0.4291007
  20         0.8               100      0.9122601  0.3315345  0.4315567
  20         0.8               200      0.9125090  0.3314931  0.4317529
  20         0.9               100      0.9414815  0.3317716  0.4250529
  20         0.9               200      0.9417188  0.3316978  0.4252236
  25         0.5               100      0.8678305  0.3672017  0.4276276
  25         0.5               200      0.8679357  0.3672072  0.4278518
  25         0.6               100      0.8651681  0.3654197  0.4216682
  25         0.6               200      0.8653531  0.3653134  0.4218736
  25         0.7               100      0.8919419  0.3512177  0.4282067
  25         0.7               200      0.8921658  0.3510882  0.4283847
  25         0.8               100      0.9255282  0.3185048  0.4301952
  25         0.8               200      0.9258008  0.3184235  0.4304044
  25         0.9               100      0.9425271  0.3245901  0.4242241
  25         0.9               200      0.9428184  0.3245155  0.4244554

Tuning parameter 'eta' was held constant at a value of 0.1
Tuning
 parameter 'min_child_weight' was held constant at a value of 1

Tuning parameter 'subsample' was held constant at a value of 1
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were nrounds = 100, max_depth = 10, eta
 = 0.1, gamma = 0, colsample_bytree = 0.5, min_child_weight = 1 and subsample
 = 1.
```

We can retrieve best tune parameters:

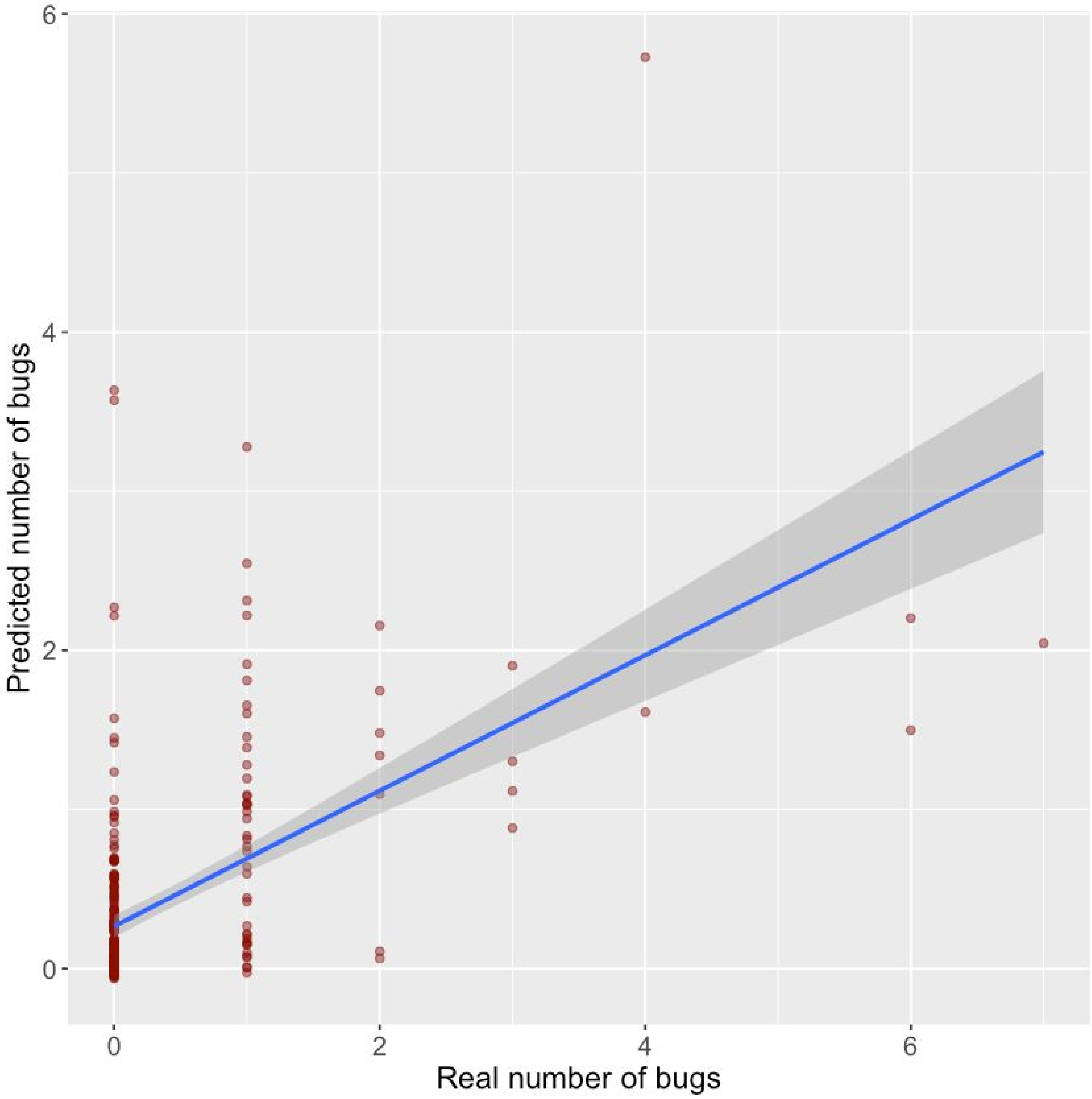


nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
100	10	0.1	0	0.5	1	1

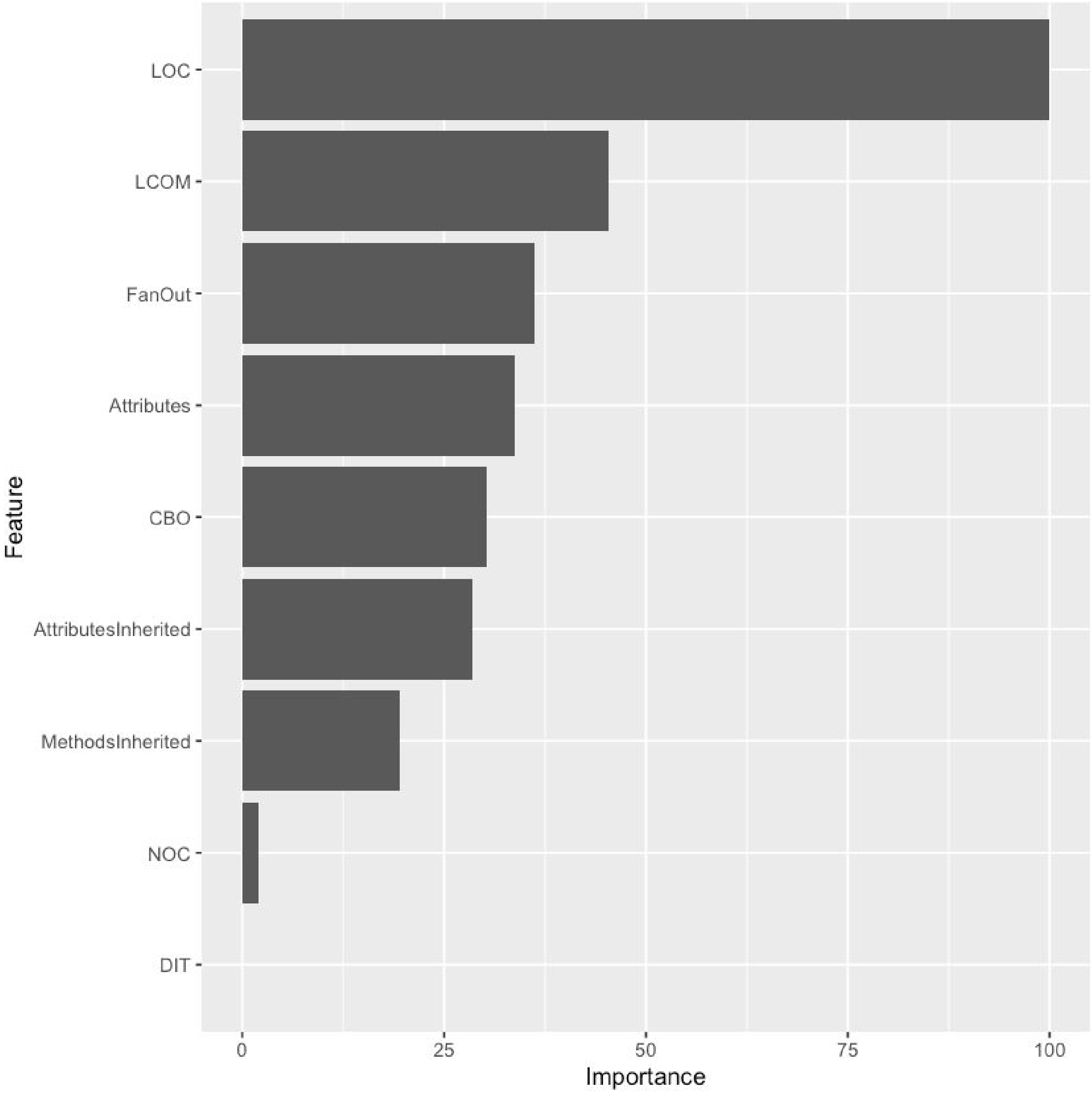
Such as we did in previous cases, we can evaluate our model by calculating RMSE (root mean squared error) and R^2 (coefficient of determination) for the testing data. The results can be seen in the table below.

RMSE	R^2
0.770	0.222

Prediction chart is presented below.

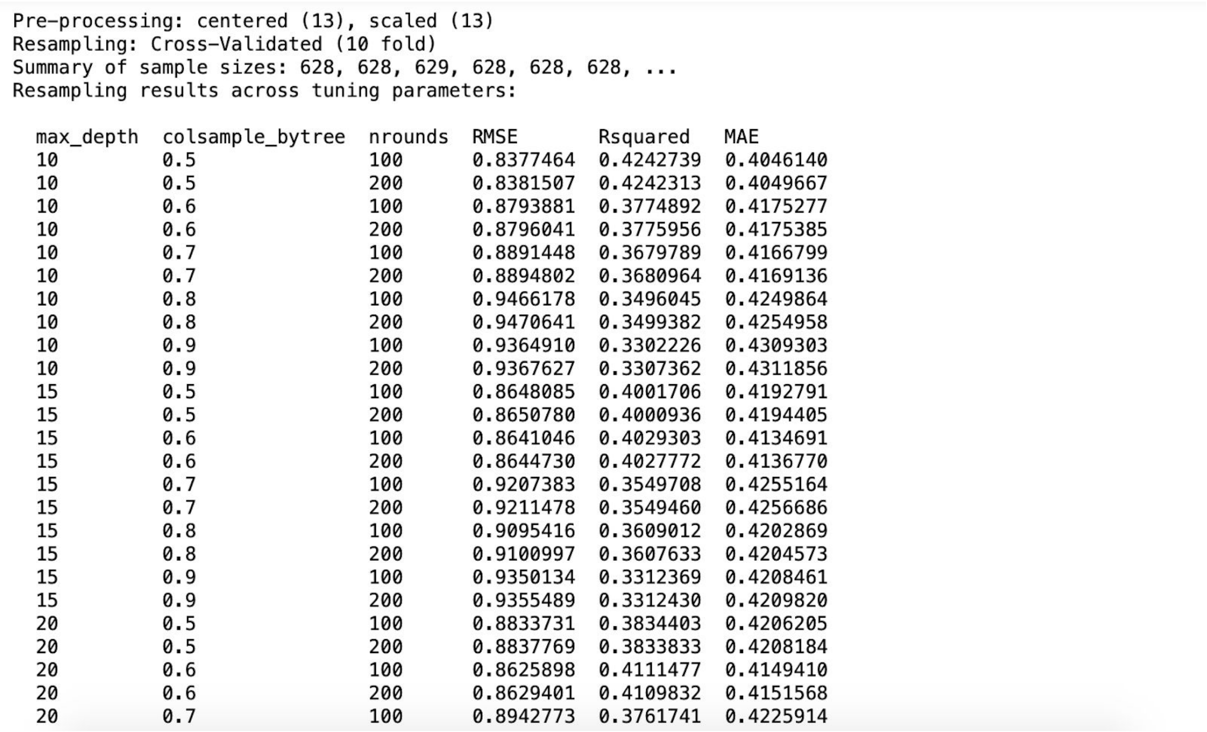


As it was in the case of training a random forest, we can ask which variables have the most predictive power. Variables with high importance are drivers of the outcome and their values have a significant impact on the outcome values. The importance plot generated using *varImp* generic method for calculating variable importance for objects produced by *train* and method-specific methods can be seen below.

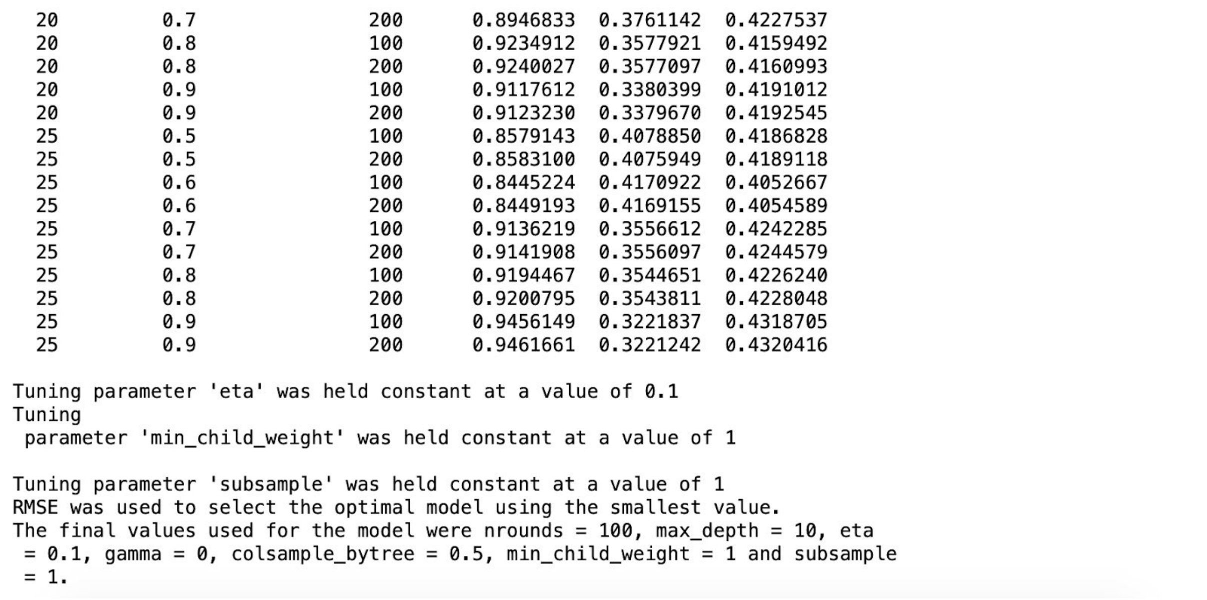


The plot again indicates that **LOC** attributes is the most important predictor. The rest of attributes have much smaller impact on the final number of bugs. DIT attribute seems to be redundant in this analysis.

If eXtreme Gradient Boosting is concerned, we can again perform similar training with preprocessing (**scaling** and **centering**), as it was in case of *k*-NN regression models. The summary of model trained after data preprocessing can be seen below.

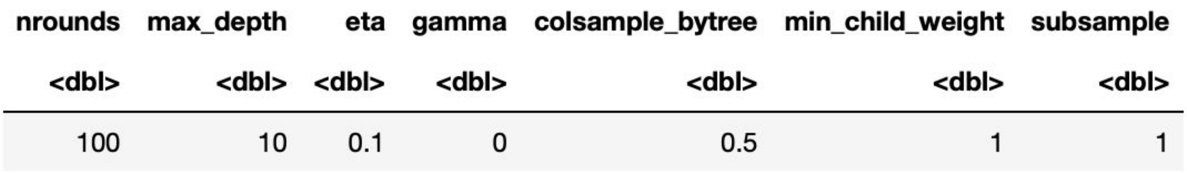


```
Pre-processing: centered (13), scaled (13)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 628, 628, 629, 628, 628, 628, ...
Resampling results across tuning parameters:

  max_depth  colsample_bytree  nrounds  RMSE       Rsquared   MAE
  10         0.5               100      0.8377464  0.4242739  0.4046140
  10         0.5               200      0.8381507  0.4242313  0.4049667
  10         0.6               100      0.8793881  0.3774892  0.4175277
  10         0.6               200      0.8796041  0.3775956  0.4175385
  10         0.7               100      0.8891448  0.3679789  0.4166799
  10         0.7               200      0.8894802  0.3680964  0.4169136
  10         0.8               100      0.9466178  0.3496045  0.4249864
  10         0.8               200      0.9470641  0.3499382  0.4254958
  10         0.9               100      0.9364910  0.3302226  0.4309303
  10         0.9               200      0.9367627  0.3307362  0.4311856
  15         0.5               100      0.8648085  0.4001706  0.4192791
  15         0.5               200      0.8650780  0.4000936  0.4194405
  15         0.6               100      0.8641046  0.4029303  0.4134691
  15         0.6               200      0.8644730  0.4027772  0.4136770
  15         0.7               100      0.9207383  0.3549708  0.4255164
  15         0.7               200      0.9211478  0.3549460  0.4256686
  15         0.8               100      0.9095416  0.3609012  0.4202869
  15         0.8               200      0.9100997  0.3607633  0.4204573
  15         0.9               100      0.9350134  0.3312369  0.4208461
  15         0.9               200      0.9355489  0.3312430  0.4209820
  20         0.5               100      0.8833731  0.3834403  0.4206205
  20         0.5               200      0.8837769  0.3833833  0.4208184
  20         0.6               100      0.8625898  0.4111477  0.4149410
  20         0.6               200      0.8629401  0.4109832  0.4151568
  20         0.7               100      0.8942773  0.3761741  0.4225914
```



```
  20         0.7               200      0.8946833  0.3761142  0.4227537
  20         0.8               100      0.9234912  0.3577921  0.4159492
  20         0.8               200      0.9240027  0.3577097  0.4160993
  20         0.9               100      0.9117612  0.3380399  0.4191012
  20         0.9               200      0.9123230  0.3379670  0.4192545
  25         0.5               100      0.8579143  0.4078850  0.4186828
  25         0.5               200      0.8583100  0.4075949  0.4189118
  25         0.6               100      0.8445224  0.4170922  0.4052667
  25         0.6               200      0.8449193  0.4169155  0.4054589
  25         0.7               100      0.9136219  0.3556612  0.4242285
  25         0.7               200      0.9141908  0.3556097  0.4244579
  25         0.8               100      0.9194467  0.3544651  0.4226240
  25         0.8               200      0.9200795  0.3543811  0.4228048
  25         0.9               100      0.9456149  0.3221837  0.4318705
  25         0.9               200      0.9461661  0.3221242  0.4320416

Tuning parameter 'eta' was held constant at a value of 0.1
Tuning
 parameter 'min_child_weight' was held constant at a value of 1

Tuning parameter 'subsample' was held constant at a value of 1
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were nrounds = 100, max_depth = 10, eta
 = 0.1, gamma = 0, colsample_bytree = 0.5, min_child_weight = 1 and subsample
 = 1.
```

We can retrieve best tune parameters:

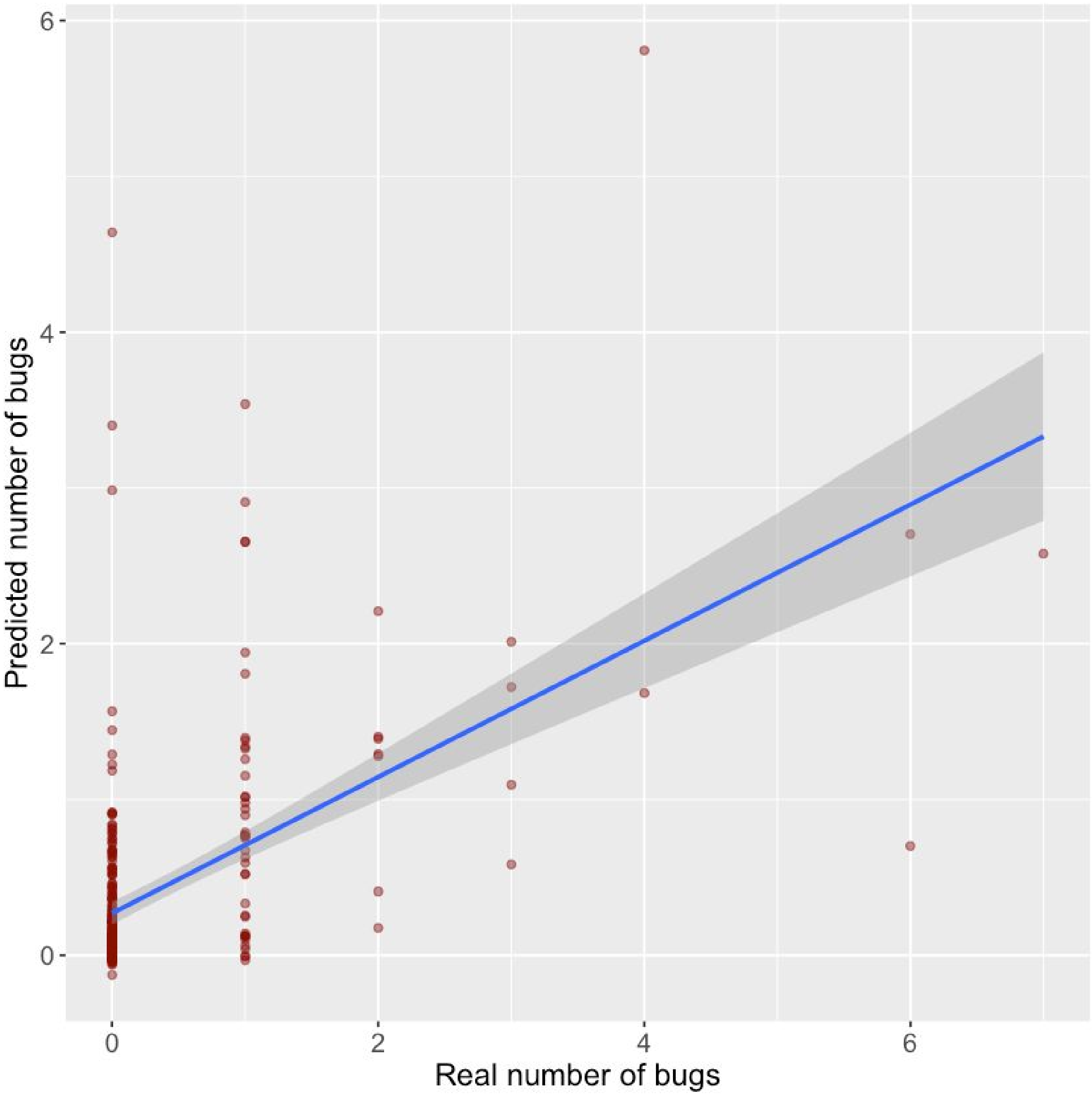


nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
100	10	0.1	0	0.5	1	1

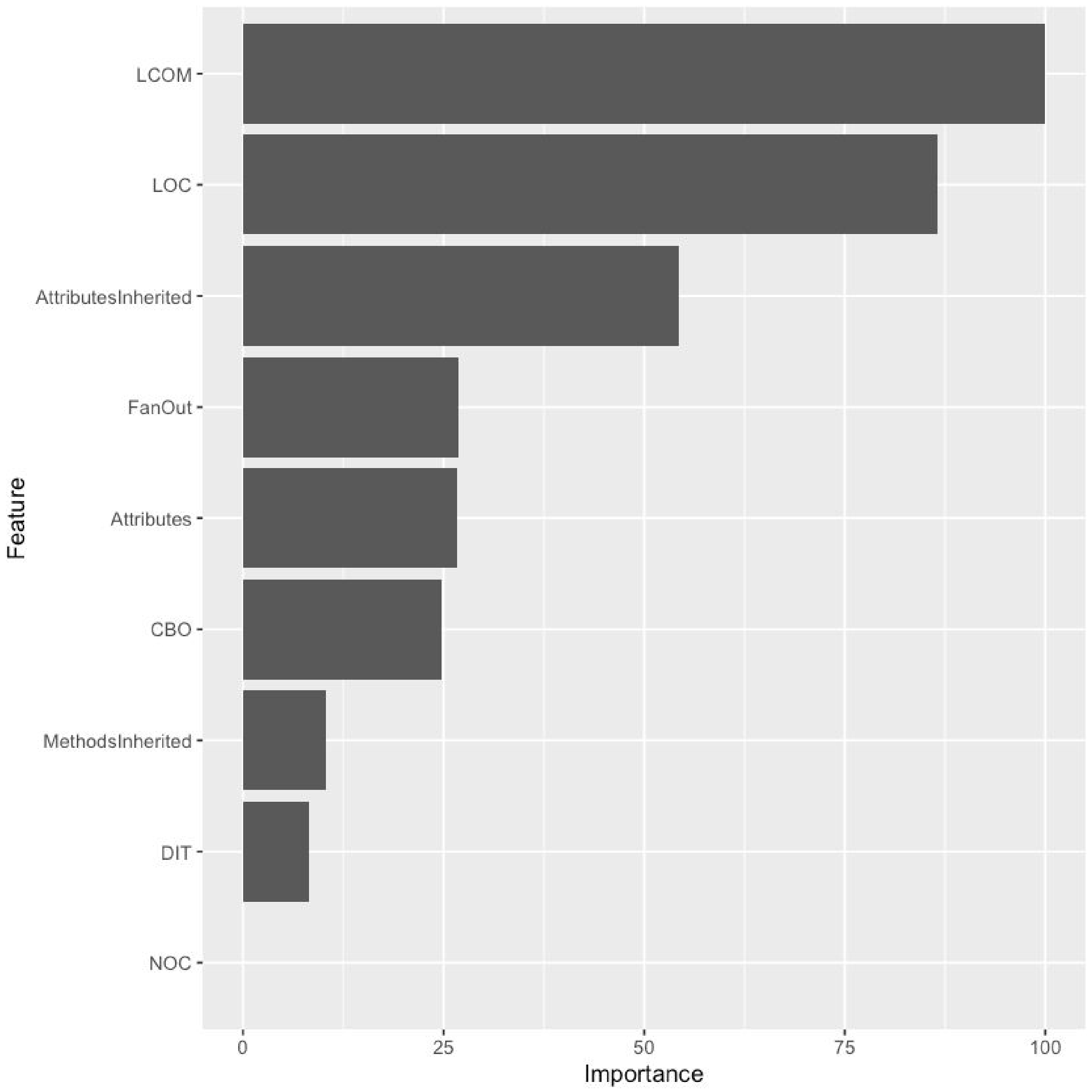
Such as we did in previous cases, we can evaluate our model by calculating RMSE (root mean squared error) and R^2 (coefficient of determination) for the testing data. The results can be seen in the table below.

RMSE	R^2
0.791	0.179

Prediction chart is presented below.



We can again inspect variables that have the most predictive power. Variables with high importance are drivers of the outcome and their values have a significant impact on the outcome values. The importance plot generated using *varImp* generic method for calculating variable importance for objects produced by *train* and method-specific methods can be seen below.



Surprisingly, the most important predictor in this analysis turned out to be **LCOM** attribute with LOC on the second place. However, there is not much difference in Importance between LCOM and LOC.

Caret ANN Regression

Artificial Neural Networks are a subset of ML algorithms, modeled loosely after the human brain, that are designed to recognize patterns. As a subdomain of machine learning focused on using deep multi-layered neural networks is called *deep learning*.

Neural networks help us with problems such as clustering, regression and classification. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on.

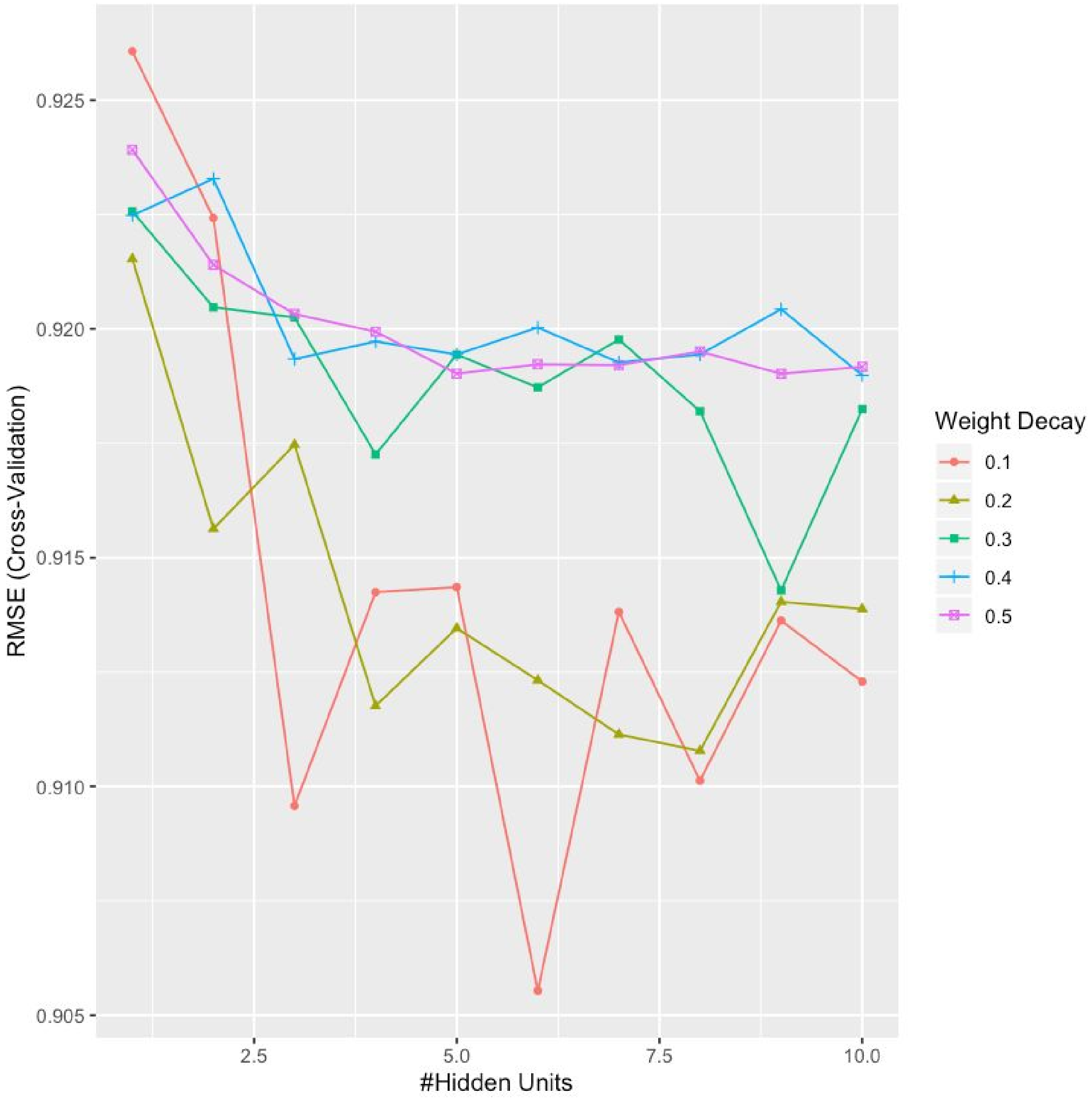
An ANN architecture is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it. In ANN implementations, the "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

For regression purposes we have used ANN implementation offered by R *Caret* library. *Caret* neural network (nnet) model architecture consists of just one hidden layer. Training a neural network using Caret we need to specify two hyper-parameters[7]: size and decay. **Size** is the number of units in the hidden layer (nnet fit a single hidden layer neural network) and **decay** is the regularization parameter to avoid over-fitting.

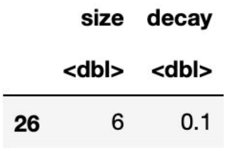
There are two different ways to tune the hyper-parameters using Caret: Grid Search and Random Search. Using Grid Search (basically Brute Force approach) we need to define the grid for every parameter according to your prior knowledge or we can fix some parameters and iterate on the remaining ones. We have decided to iterate through size grid established as a sequence of numbers from 1 to 10 every 1 unit and decay grid established as a sequence from 0.1 to 0.5 every 0.1 unit. Once the Grid Search has finished or the Tune Length is completed *Caret* uses the performance metrics to select the best model according to the criteria previously defined.

Model training plot can be seen below.

[7] A hyperparameter is a parameter whose value is set before the learning process begins. The values of parameters are derived via learning. Examples of hyperparameters include learning rate, the number of hidden layers and batch size.



We can retrieve best tune parameters:

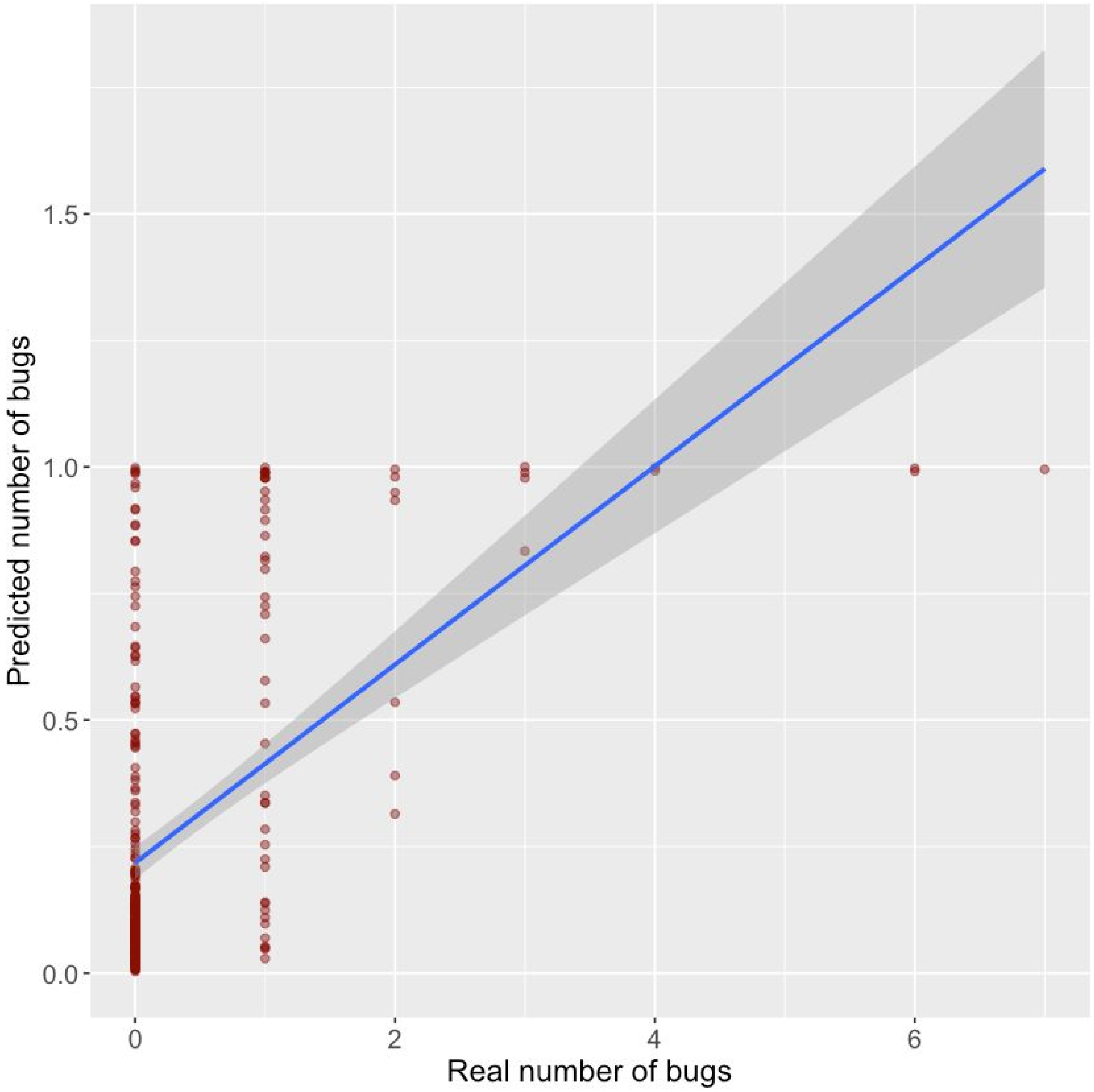


	size	decay
	<dbl>	<dbl>
26	6	0.1

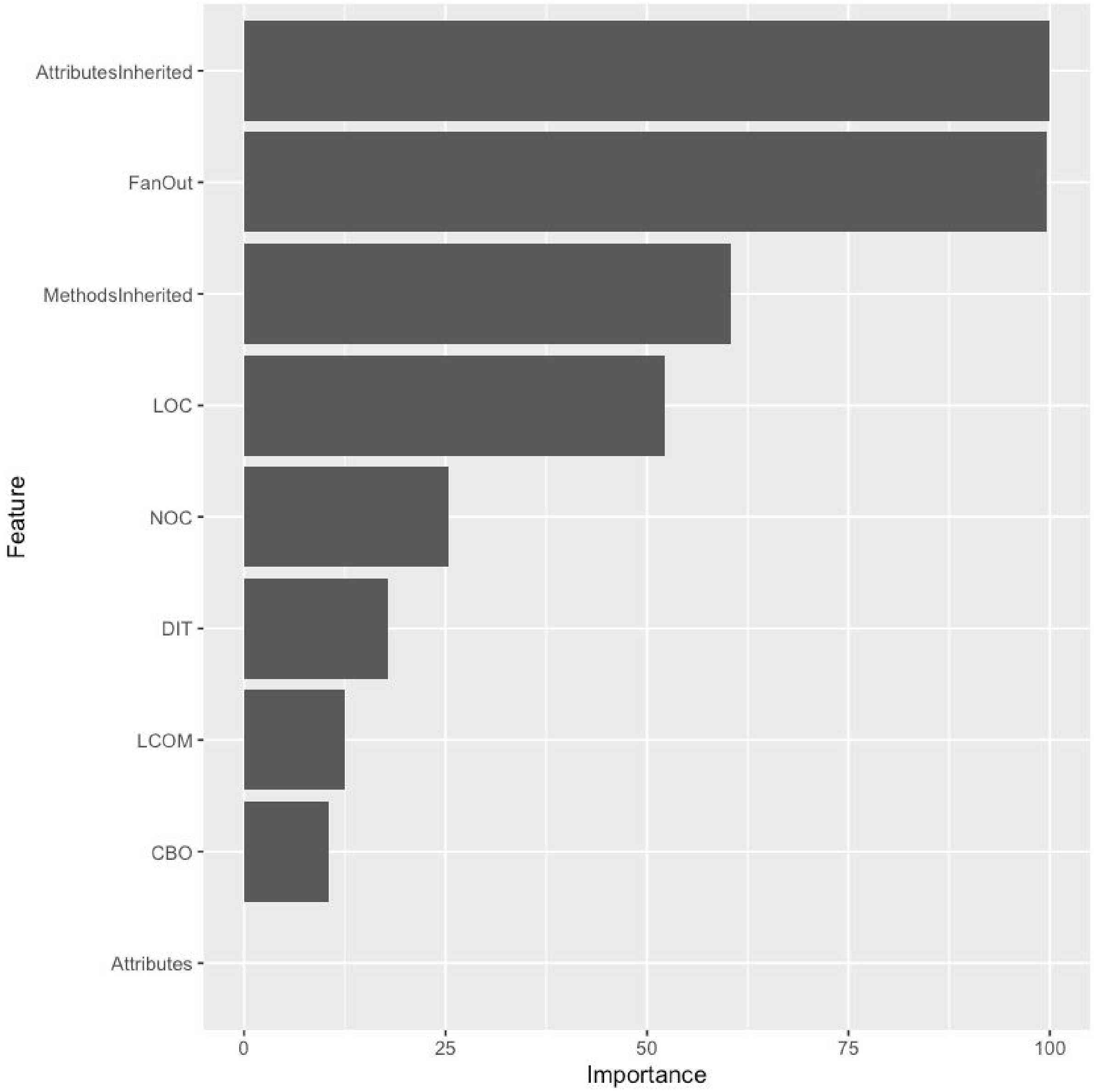
We can evaluate our model by calculating RMSE (root mean squared error) and R^2 (coefficient of determination) for the testing data, in the same way as in previous cases. The results can be seen in the table below.

RMSE	**R^2**
0.752	0.259

v



We can again inspect variables that have the most predictive power. Variables with high importance are drivers of the outcome and their values have a significant impact on the outcome values. The importance plot generated using *varImp* generic method for calculating variable importance for objects produced by *train* and method-specific methods can be seen below.



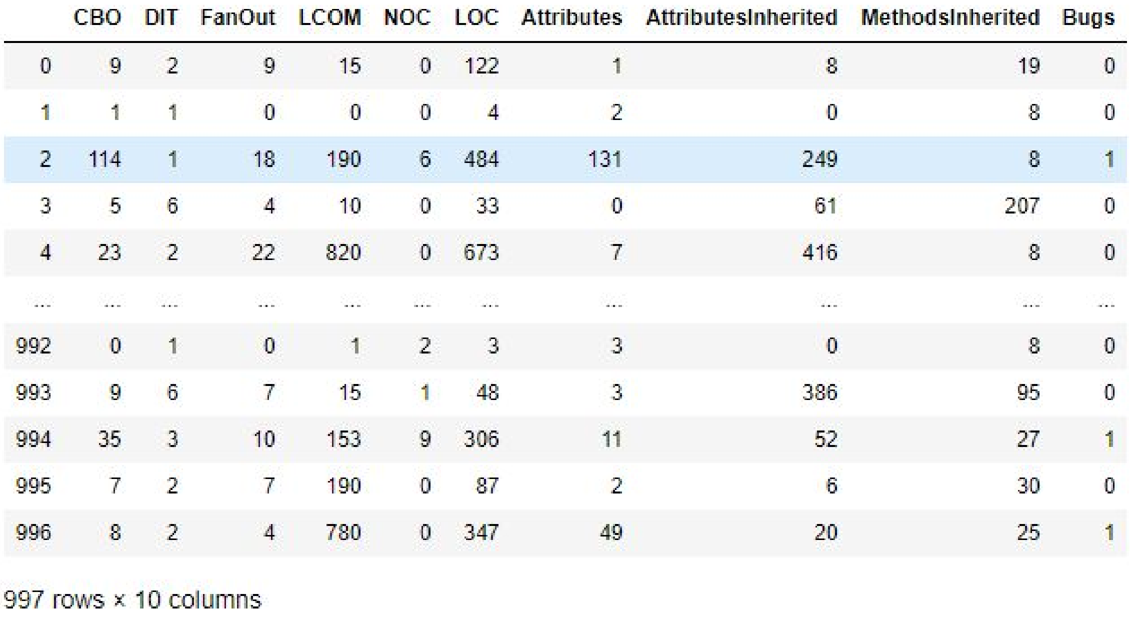
Unexpectedly, the most important predictor in this analysis turned out to be **AttributesInherited** (number of inherited attributes) as well as **FanOut** with almost the same score in terms of Importance. Interestingly, general number of attributes seem to be unimportant in this prediction. Results gathered are absolutely unlike any other regression model we have discussed so far.

Tensorflow/Keras ANN Regression

In order to try to achieve better prediction results using artificial neural network, we took another try with designing more sophisticated regression model using Python Tensorflow and Keras libraries, which are considered to be the most popular deep learning tools in data science. All operations performed and the dataset can be inspected via NN_Python notebook.

We have started with some basic **data preprocessing**. When numeric input data features have values with different ranges, each feature should be scaled independently to the same range. One can use StandardScaler to standardize features by removing the mean and scaling to unit variance.

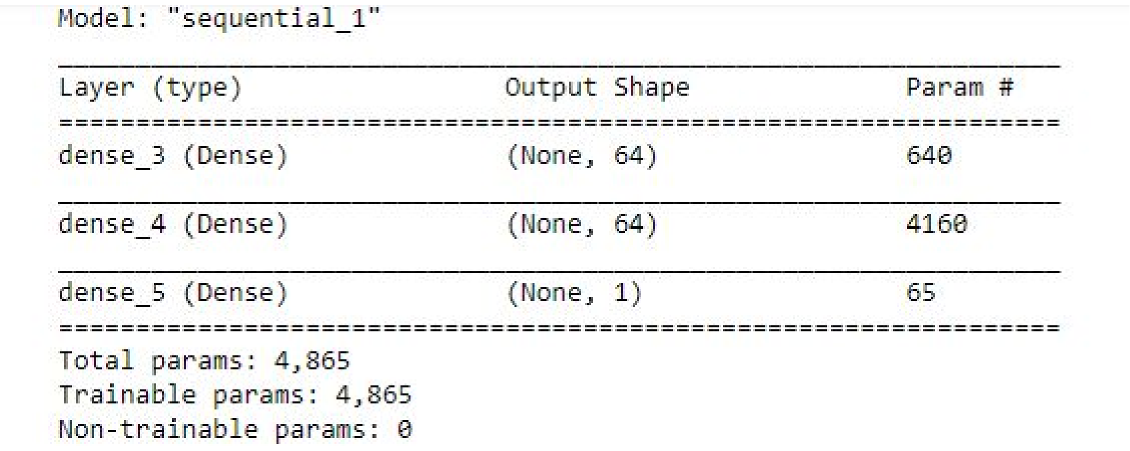
For reducing training time and improving results one should remove highly correlated features. After doing so our dataset looks like on the screenshot below.



	CBO	DIT	FanOut	LCOM	NOC	LOC	Attributes	AttributesInherited	MethodsInherited	Bugs
0	9	2	9	15	0	122	1	8	19	0
1	1	1	0	0	0	4	2	0	8	0
2	114	1	18	190	6	484	131	249	8	1
3	5	6	4	10	0	33	0	61	207	0
4	23	2	22	820	0	673	7	416	8	0
...	...	...	...	...	...	...	...	...	...	...
992	0	1	0	1	2	3	3	0	8	0
993	9	6	7	15	1	48	3	386	95	0
994	35	3	10	153	9	306	11	52	27	1
995	7	2	7	190	0	87	2	6	30	0
996	8	2	4	780	0	347	49	20	25	1

997 rows × 10 columns

In the next stage of the study, we aimed at creating a regression model. We have used a sequential model with two densely connected hidden layers, and an output layer that returns a single, continuous value. One can use *summary* method to print a simple description of the model.



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_3 (Dense)              (None, 64)                640       
_________________________________________________________________
dense_4 (Dense)              (None, 64)                4160      
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 65        
=================================================================
Total params: 4,865
Trainable params: 4,865
Non-trainable params: 0
```

In order to automatically stop training when the validation score doesn't improve, one can use an EarlyStopping callback. It tests a training condition for every epoch. If a set amount of epochs elapses without showing improvement, then it automatically stops the training. Early stopping is a useful technique to prevent overfitting.

One can see the values of Mean Absolute Error and Mean Squared Error in the model training summary. Mean Absolute Error (MAE) is a common regression metric.

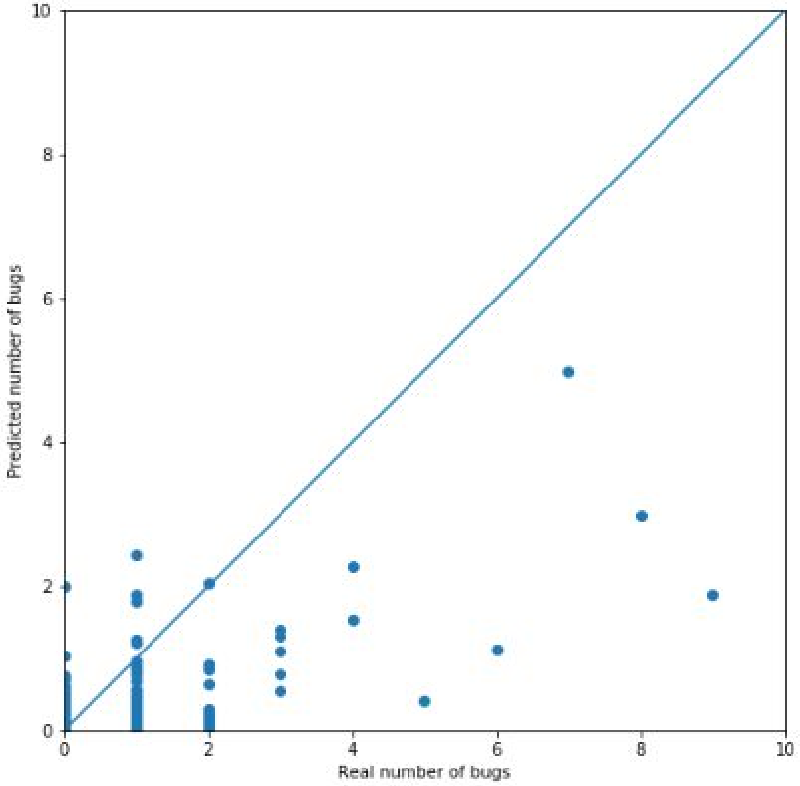
```
Epoch 1/1000
667/667 - 0s - loss: 1.1850 - mean_absolute_error: 0.4915 - mean_squared_error: 1.1850 - val_loss: 5195.2504 - val_mean_absolute_error: 21.2039 - val_mean_squared_error: 5195.2505
Epoch 2/1000
667/667 - 0s - loss: 0.8461 - mean_absolute_error: 0.4335 - mean_squared_error: 0.8461 - val_loss: 17401.9664 - val_mean_absolute_error: 32.9716 - val_mean_squared_error: 17401.9668
Epoch 3/1000
```

```
    667/667 - 0s - loss: 0.7604 - mean_absolute_error: 0.4149 - mean_squared_error: 0.7604 -
val_loss: 33150.4439 - val_mean_absolute_error: 49.9120 - val_mean_squared_error: 33150.4414
    Epoch 4/1000
    667/667 - 0s - loss: 0.7105 - mean_absolute_error: 0.4012 - mean_squared_error: 0.7105 -
val_loss: 37764.7386 - val_mean_absolute_error: 57.1804 - val_mean_squared_error: 37764.7383
    Epoch 5/1000
    667/667 - 0s - loss: 0.6896 - mean_absolute_error: 0.4113 - mean_squared_error: 0.6896 -
val_loss: 40967.9779 - val_mean_absolute_error: 61.3195 - val_mean_squared_error: 40967.9805
    Epoch 6/1000
    667/667 - 0s - loss: 0.6797 - mean_absolute_error: 0.3955 - mean_squared_error: 0.6797 -
val_loss: 39886.2372 - val_mean_absolute_error: 64.1172 - val_mean_squared_error: 39886.2383
    Epoch 7/1000
    667/667 - 0s - loss: 0.6610 - mean_absolute_error: 0.3993 - mean_squared_error: 0.6610 -
val_loss: 45959.8151 - val_mean_absolute_error: 66.9820 - val_mean_squared_error: 45959.8164
    Epoch 8/1000
    667/667 - 0s - loss: 0.6502 - mean_absolute_error: 0.3957 - mean_squared_error: 0.6502 -
val_loss: 19855.6518 - val_mean_absolute_error: 43.1761 - val_mean_squared_error: 19855.6523
    Epoch 9/1000
    667/667 - 0s - loss: 0.6480 - mean_absolute_error: 0.3814 - mean_squared_error: 0.6480 -
val_loss: 30729.6461 - val_mean_absolute_error: 56.8368 - val_mean_squared_error: 30729.6445
    Epoch 10/1000
    667/667 - 0s - loss: 0.6314 - mean_absolute_error: 0.3947 - mean_squared_error: 0.6314 -
val_loss: 31988.0159 - val_mean_absolute_error: 57.5216 - val_mean_squared_error: 31988.0176
    Epoch 11/1000
    667/667 - 0s - loss: 0.6263 - mean_absolute_error: 0.3842 - mean_squared_error: 0.6263 -
val_loss: 22191.6377 - val_mean_absolute_error: 48.2574 - val_mean_squared_error: 22191.6387
```

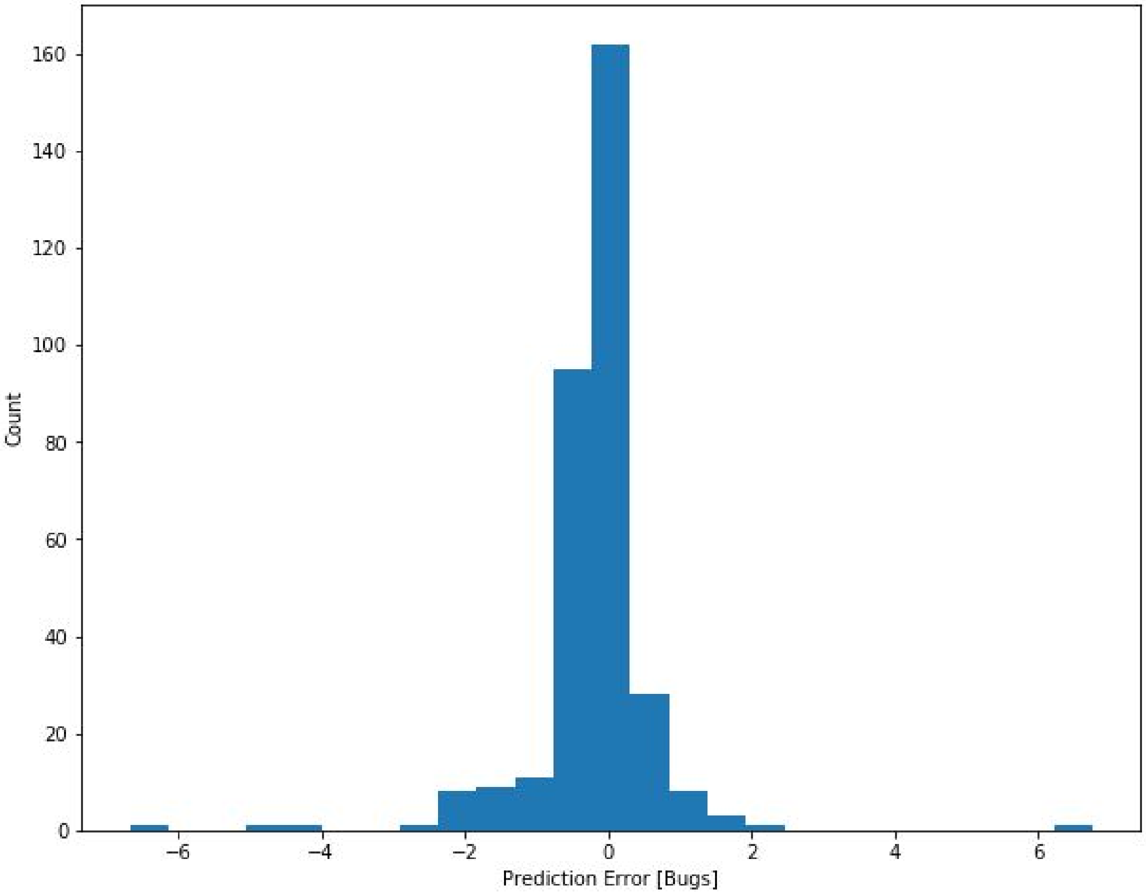
It is possible to measure different error metrics using *sklearn.metrics* package. The one may inspect the metrics gathered in the table below.

Variance	**Max Error**	**Mean Abs. Error**	**Median Abs. Error**	**MSE**	**RMSE**	**R^2**
0.361	6.754	0.458	0.239	0.755	0.869	0.329

Prediction chart for this regression model can be seen below.



The error distribution also can be seen. It's not quite gaussian, but we might expect that because the number of samples is very small.



Results Discussion and Conclusions

We have gathered all prediction results in order to find the best regression models for the bugs prediction task.

We have decided to compare two of the most popular statistical measures: Root Mean Square Error (RMSE) and R-Squared (R2). RMSE is the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are. In other words, it tells us how concentrated the data is around the line of best fit. On the other hand, R-squared is a statistical measure that represents the proportion of the variance for a dependent variable that is explained by an independent variable or variables in a regression model. Basically it means that R-squared can be seen as a measure of how close the data are to the fitted regression line by showing the percentage of the response variable variation that is explained by a linear model. In this study we have R2 values always between 0 – 1 (ex. 0 – 100%).

- 0 indicates that the model explains none of the variability of the response data around its mean.
- 1 indicates that the model explains all the variability of the response data around its mean.

All RMSE and R^2 results for different regression models are shown in the table below.

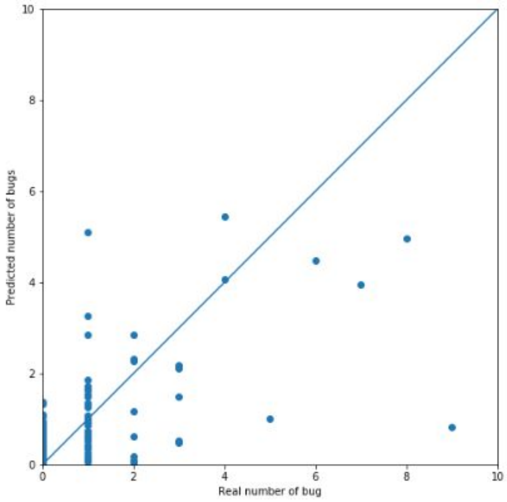
Regression Model	**RMSE**	**R^2**
k-NN	**0.728**	0.306
k-NN + Preprocessing	0.752	0.258
Random Forest	0.742	0.277
XGBoost	0.770	0.222
XGBoost + Preprocessing	0.791	**0.179**
Caret ANN	0.758	0.246
Tensorflow/Keras ANN	**0.869**	**0.329**

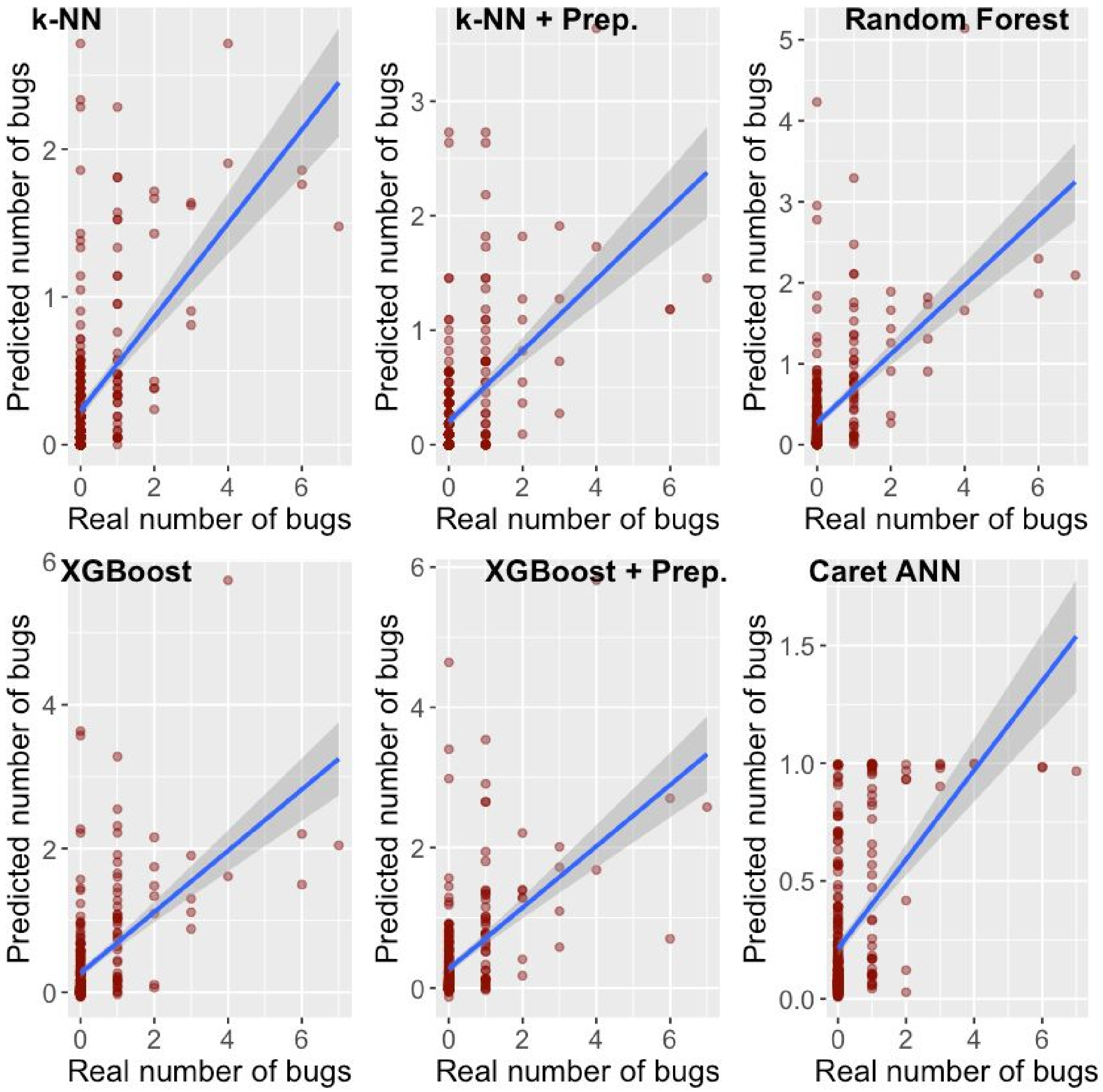
The one can observe the lowest value of RMSE metric for ***k*-Nearest Neighbours algorithm without data preprocessing**, particularly centering and scaling. The biggest RMS error was produced while using Tensorflow/Keras artificial neural network regression model. On the other hand, Tensorflow ANN model scored the highest value of R^2 coefficient of determination, although it is still not very high – about 33%.

To our mind, the best model for this bugs prediction task are ***k*-NN** without data preprocessing due to its low RMSE and high R^2 values, having **Random Forest** model on a second place. However, it should be noted that training Random Forest model takes significantly more time then developing *k*-NN one.

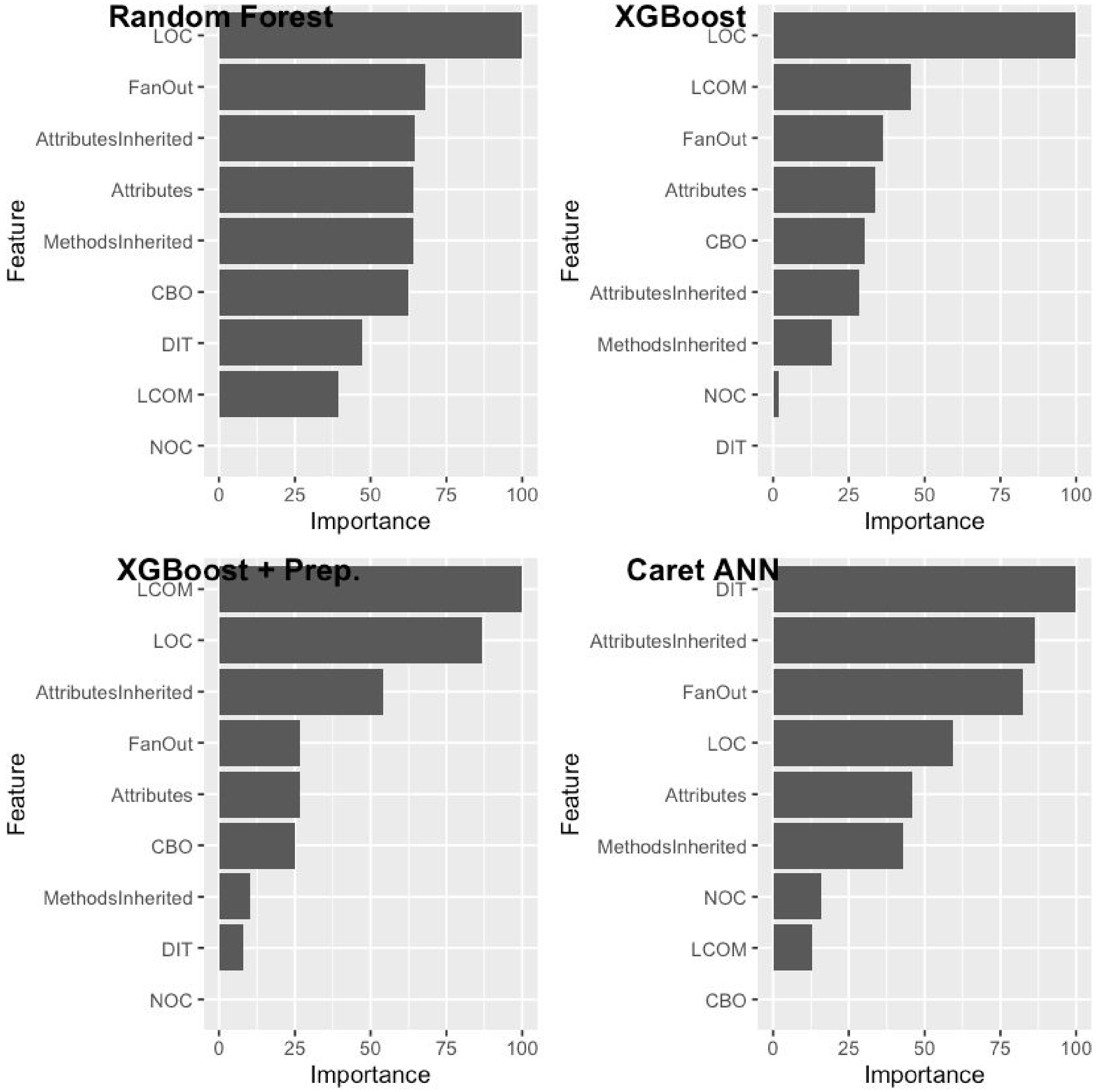
On the prediction charts below, the one can see that quite predicted to real number of bugs ratio (best fit line) was observed in Random Forest, Extreme Gradient Boosting and

Extreme Gradient Boosting with data preprocessing models. However, the highest ratio was observed in case of Tensorflow ANN prediction that can be inspected below.





We were able to retrieve variables of the highest prediction impact in case of Random Forest, XGBoost, XGBoost with Preprocessing and Caret ANN, which can be inspected below. We can observe than in most cases LOC attribute held the first place in terms of Importance metric. What is interesting, Importance metric plot has totally different form in case of Caret ANN prediction model than in others.



To sum up, we would like to conclude that the most promising bug predictor is LOC attribute (number of lines of code). Other object-oriented metrics that could be studied as possible bug predictors are LCOM (lack of cohesion in methods) nd FanOut (number of classes referenced by other class). The most promising machine learning regression algorithms for bug prediction turned out to be *k*-Nearest Neighbours and Random Forest ones, although Tensorflow ANN shows the best variable explanation in a linear model.

What is more, the one should notice that R-square values for all the models obtained are low (< 40%). This fact may indicate that the data contain an inherently high amount of unexplainable variability. Small R-square values may inform us that that bugs appearances are highly unpredictable.

In the next paragraph we discuss threats to validity of our results and conclusions.

Validity Threats

Conducting the study, we have identified several validity threats. Only some of them have been successfully overcome.

1. One of the common threats to validity of data science results is having **missing values** in data. The data was inspected to make sure no missing values were found.

2. Correlation coefficients for all of attributes were calculated. Highly **correlated variables** (correlation coef. > 0.7) were removed from further inquiries. Removing correlated attributes indeed improved the results of the study having impact on a smaller RMSE values in case of all regression models.

3. The main problem with ***k*-NN** algorithm is that it is a lazy learner, i.e. it does not learn anything from the training data and simply uses the training data itself for classification. To predict the label of a new instance the KNN algorithm will find the K closest neighbors to the new instance from the training data, the predicted class label will then be set as the most common label among the K closest neighboring points. The main disadvantage of this approach is that the algorithm must compute the distance and sort all the training data at each prediction, which can be slow in certain cases.

   Secondly, *k*-NN prediction can suffer from skewed class distributions. For example, if a certain class is very frequent in the training set, it will tend to dominate the majority voting of the new example.

   What is more, the accuracy of *k*-NN can be severely degraded with high-dimension data because there is little difference between the nearest and farthest neighbor.

   Another threat of this approach is that the algorithm does not learn anything from the training data, which can result in the algorithm not generalizing well and also not being robust to noisy data.

   The one can handle skewed class distributions is by implementing weighted voting. With this solution, the class of each of the k neighbors is multiplied by a weight proportional to the inverse of the distance from that point to the given test point. This ensures that nearer neighbors contribute more to the final vote than the more distant ones. Accuracy of the algorithm may be improved by implementing data preprocessing techniques such as normalization, rescaling, centering etc. In the study conducted preprocessing was used indeed, although it had not improved the results. Other means to improved *k*-NN accuracy may involve trying different distance metrics.

4. **Random forest** models are known for no need for feature normalization/scaling, they are easy to measure the relative importance of each feature and can handle categorical and numerical features. However, in terms of regression, Random Forest model may not predict beyond the range in the training data, and that they may overfit data sets that are particularly noisy.

   Secondly, it can take a long time to train a model with a large number of trees. What is more, Random Forest can feel like a black box approach for a statistical modelers we have very little control on what the model does. They are less intuitive (when you have a large collection of decision trees it is hard to have an intuitive grasp of the relationship existing in the input data) and leave not much room for interpretation. Random forests are then used when high performance with less need for interpretation is a priority.

5. **Gradient Boosting** model build trees one at a time, where each new tree helps to correct errors made by previously trained tree. With each tree added, the model becomes even more expressive. Due to that, Gradient Boosting model training generally takes longer than Random Forest ones because of the fact that trees are built sequentially. Apart from that, Gradient Boosting models are also prone to overfitting. Caret library however use some strategies to overcome same and build more generalized trees using a combination of parameters like learning rate (shrinkage) and depth of tree.

6. One of a threats to validity of **neural networks** models results in terms of using Caret package might be associated with single-layer architecture of Caret neural network model. One-layer architecture may result in too simple model, which is not able to predict new observations efficiently. What is more, neural networks in general may result with overfitting the data. However, using pre-built packages like Caret allows for using built-in optimization tools such us automatic parameter tuning.

   Tensorflow/Keras tools enables training multi-layered neural network models. As for the threats to model validity in this case, we have used EarlyStopping feature, which is a useful technique to prevent overfitting. EarlyStopping callback tests a training condition for every epoch. If a set amount of epochs elapses without showing improvement, then it automatically stops the training.

7. One of a threats to validity of **neural networks** models results in terms of using Caret package might be associated with single-layer architecture of Caret neural network model. One-layer architecture may result in too simple model, which is not able to predict new observations efficiently. What is more, neural networks in general may result with overfitting the data. However, using pre-built packages like Caret allows for using built-in optimization tools such us automatic parameter tuning.

References

[1] D'Ambros M., Michele Lanza M., Robbes R., *An Extensive Comparison of Bug Prediction Approaches,* Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories), p. 31-41

[2] Caret library documentation

[2] Chidamber S., Kemerer C., *A Metrics Suites for Object-Oriented Design*, IEEE Transactions on Software Engineering 20(6), 1994, p. 476-493

[3] Grano G., *Code Quality Metrics* (online), URL: https://github.com/sealuzh/user_quality/wiki/Code-Quality-Metrics [extracted: 02.12.2019]

[4] *Object-Oriented Software Metrics - Class Level Metrics* (online), Virtual Machinery, URL: http://www.virtualmachinery.com/jhawkmetricsclass.htm [extracted: 02.12.2019]

[5] Hayes. A, *R-Squared Definition* (online), Investopedia, URL: https://www.investopedia.com/terms/r/r-squared.asp [extracted: 02.12.2019]

[6] *Fit Random Forest Model* (online), URL: http://code.env.duke.edu/projects/mget/export/HEAD/MGET/Trunk/PythonPackage/dist/TracOnlineDocumentation/Documentation/ArcGISReference/RandomForestModel.FitToArcGISTable.html [extracted: 02.12.2019]

[7] Hoare J., How is Variable Importance Calculated for a Random Forest? (online), URL: https://www.displayr.com/how-is-variable-importance-calculated-for-a-random-forest/ [extracted: 02.12.2019]

[8] Tensorflow and Keras documentation

[9] Stackoverflow forum

[10] How to Interpret a Regression Model with Low R-squared and Low P values (online), URL: https://blog.minitab.com/blog/adventures-in-statistics-2/how-to-interpret-a-regression-model-with-low-r-squared-and-low-p-values [extracted: 15.01.2020]