Comparing Random Forest, XGBoost and Neural Networks With Hyperparameter Optimization by Nested Cross-Validation

Roskilde University

6th semester, Spring 2019

Supervised by: Finn Gustafsson

Team:

Casper Bøgeskov Hansen	61092	casphan@ruc.dk
Ahmed Magdy Mahmoud Ali	66924	ammali@ruc.dk
Mads Zeuch Ethelberg	62608	mze@ruc.dk
Martin Moltke Wozniak	55518	mmoltke@ruc.dk

Abstract

It can be challenging to obtain an objective house price when selling your house since real-estate agents could be biased by many factors. This paper attempts to predict the sales price of houses in Ames, Iowa, by applying different machine learning models, which each have different strengths and weaknesses. In this project, a generic pipeline to inputting different machine learning models has been proposed. The proposed pipeline includes hyperparameter optimization, implemented with a nested cross-validation procedure. From the pipeline, the paper compares the algorithms and contrasts the average scores with a root mean squared logarithmic error (RMSLE) scoring for each models. XGBoost scored 0.123, Random Forest scored 0.139 and Neural Networks scored 0.414, where a lower score is better.

From the hyperparameter optimization across the different sets of hyperparameters generated from nested CV, it was found for random forest, that the range of values was not wide enough to get a proper optimization. It was learned that hyperparameters are important, and one of the most important areas to consider because it relates very directly to how well any model performs. It was concluded that XGBoost produced a lower score than the rest of the models in the proposed pipeline, thus XGBoost was the best model. In further studies, more focus should be given to other imputation methods, model stacking, and deployment.

Table of Contents

Abstract 1

1 Introduction 6

1.1 What Is Machine Learning? 6

1.1.1 Supervised and Unsupervised Learning 7

1.1.2 Case: House Prices 8

1.2 Research Question 8

1.3 Limitations Of The Bachelor Project 9

1.4 Target Audience 9

2 General Machine Learning 10

2.1 Bias-Variance Trade-off 11

2.1.1 Bias 11

2.1.2 Variance 11

2.1.3 Combining Bias and Variance 12

2.2 K-Fold Cross-Validation 13

2.2.1 Nested Cross-Validation 14

2.2.2 Hyperparameter Optimization Methods 17

2.2.3 Pseudo Code - Algorithm 1 18

2.3 Encoding data 19

2.3.1 Label Encoding 19

2.3.2 One-Hot encoding 20

2.3.3 One-Hot Encoding with Dummy variables 20

2.4 Cost Function 21

2.5 Gradient Descent 22

2.6 Feature Selection 23
2.6.1 Feature Selection vs Dimensionality Reduction 23
2.6.2 Types of Feature Selection 23
2.7 Evaluating Models 24
2.7.1 Metrics 24

3 Machine Learning Algorithms 26
3.1 Multivariable Linear Regression 27
3.1.1 Predicting Values with Linear Regression 27
3.2 Ensemble Methods 29
3.2.1 Bagging 29
3.2.2 Boosting 30
3.3 Decision Trees 31
3.3.1 Shortcomings of Decision Trees 34
3.3.2 Random Forest — Regression 34
3.4 Regularization 36
3.5 XGBoost 37
3.5.1 Gradient Boosting Machine — Regression 37
3.6 Neural Networks 40
3.6.1 The Neural Network 40
3.6.2 Feedforward Neural Network 40
3.6.3 The Node 41
3.6.4 Dropout 42

4 Implementation 44
4.1 Data Exploration and Preparation 45
4.2 Algorithm 1 - Nested CV 49
4.2.1 Fit Function 51
4.3 Pipeline: Inputting Algorithms and Hyperparameters 55

5 Results and Analysis 59
5.1 Result 60

5.1.1 Results of RF 60
5.1.2 Results of XGBoost 61
5.1.3 Results of Neural network 62
5.2 Analysis 63
5.2.1 Analysis of RF 63
5.2.2 Analysis of XGBoost 66
5.2.3 Analysis of Neural network 67

6 Discussion 68

6.1 Reflection Upon Results 68
6.1.1 Hyperparameters 68
6.1.2 Pipeline trials 69
6.1.3 Time vs Hyperparameters 69
6.2 Neural Networks 71
6.2.1 Model setup 71
6.2.2 Neural network — Regularization 71
6.2.3 Cross Validation 72
6.3 Overfitting and Underfitting 73
6.3.1 Evaluating Nested Cross-Validation 73
6.3.2 Regularization 74
6.3.3 Feature Selection 75
6.4 Imputation 75
6.4.1 How Could the Imputation be Improved 76

7 Conclusion 77

8 Perspective 78

8.1 Nested CV — Further Experiments 78
8.2 Model Stacking 78
8.3 Imputation Suggestions 79
8.4 Custom Data Handling 80
8.5 Deployment 80

Sources **81**

Appendix **85**

Code copy-paste 85

Raw results — all models 96

Raw parameters — all models 98

1 Introduction

1.1 What Is Machine Learning?

Perhaps the most important question, to understand machine learning, is *what is machine learning (ML)?* The motivation for doing ML is the fact that some jobs can be more efficiently done, compared to human beings. Perhaps one of the earliest discoveries of ML was Arthur Samuel, which defined the ML as a *field of study that gives computers the ability to learn without being explicitly programmed.* Samuel programmed a checkers-playing program, that played 10000 games of checkers against itself. In a live game, he could then figure out which board position was good or bad, depending on the wins and losses of that move [Samuel, 1959].

We work with data and call accumulated data a *dataset*. In general the larger the dataset is, the more accurate the learning algorithm will be, when it has more data. A dataset has M features (interchangeably referred to as variables) and n observations (also interchangeably called data points and samples). You could visualize features as columns and observations as rows in a table. This table or matrix is referred to as x. For each row in the training data set there is also a target value. This vector is generally referred to as y. When applying a learning algorithm, we work with the terms *training* and *testing* dataset. These refer to two types of processes, when applying a learning algorithm. Firstly, we *train* a learning algorithm with the training dataset, and secondly, we *test* what the algorithm has learned, from the training dataset, with the testing dataset. When referring to learning algorithms, we refer to them as a *model*. You could then say, that we train and test different models, i.e. we train and test different learning algorithms. We often say that a model learns a prediction rule, and therefore that the model predicts some type of value for us. When testing a model, it should never be tested on the training dataset.

1.1.1 Supervised and Unsupervised Learning

There are many different types of models in ML. What often happens, is that we categorize a learning problem into supervised or unsupervised learning, and we also define if the learning problem is a classification or regression problem. In figure 1.1, the 4 regions of machine learning can be seen. A problem can either be supervised or unsupervised and a classification or regression problem.

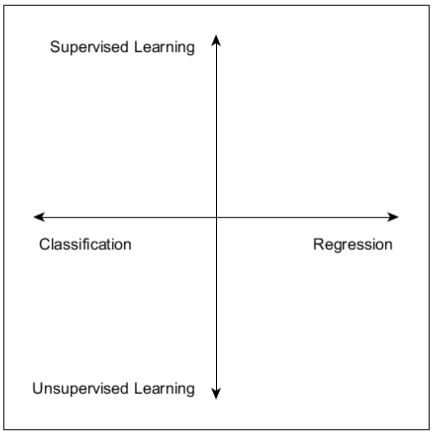

Figure 1.1: The 4 regions of Machine Learning

Supervised learning refers to a learning problem, where you have a dataset and know what you are trying to predict, e.g. whether the images in a dataset is dogs or cats. Though in unsupervised learning, you have a dataset, but you don't know what you are trying to predict. It could be that your dataset consists of images, but you would not know what animal you are trying to predict.

Classification refers to learning how to label data with classes, e.g. the example of predicting whether it is a dog or a cat in a given image. In many examples you would have only 2 classes, but it is certainly possible that there exists n classes. Regression refers to predicting or estimating values for a new data point. All supervised machine learning models can be used to make a model, which is trained using the training dataset and tested with testing dataset.

1.1.2 Case: House Prices

The problem we are tackling is predicting house prices. To do this, we have retrieved data from the *Kaggle House Prices competition*. The data consists of a training and testing dataset. The training dataset has 81 features ($M = 81$) and 1460 observations ($n = 1460$). The testing dataset has 80 features and 1459 observations, one feature less because the output feature (SalePrice) is not in the testing dataset. All of the features in the dataset are something that describes the house, e.g. OverallQual, GarageArea, YearSold etc.

What exactly we are predicting is the *SalePrice* feature in the training dataset. This is the output variable, this is also called y-hat ($\hat{y}$) in most equations for different algorithms. Our data is labeled, meaning that we know what the values of each feature means.

The goal of this house prices case, is to predict the house prices using the datasets provided by Kaggle. Since the problem is to predict a price, it is categorized as a regression problem. Since the output variable is provided in the dataset, it would be ideal to make use of supervised learning, which further categorize the problem as a supervised regression problem. The authors of the dataset left a note in *their article*, that says there are outliers which should be removed. It was identified that hyperparameter optimization is important in ML, and it was chosen to spent the time on researching ways of doing hyperparameter optimization. It was found that nested cross-validation is important when doing hyperparameter optimization, which is why we are researching the area of nested cross-validation.

1.2 Research Question

How can we use nested cross-validation to compare the models Random Forest, XGBoost and Neural Networks, and contrast between the different models by scores, using a house prices dataset from the city Ames in Iowa?

- What are *some* of the important areas of ML to take into account, when applying ML models?
- How to deal with missing data in a dataset?

1.3 Limitations Of The Bachelor Project

When trying to solve the research question with regards to ML, there is a certain mathematical background knowledge, which is needed for the underlying algorithms. The basis for the limitations of the project report is the following question: *To what extent do we need to know the math behind ML models and what do we need to know to complete the project?*

While there are many different areas of math, we put it as outside the scope of this project. Though, we still need to know the process behind the math, and in which way these different models work. The way which we go about this, is learning by doing and learning on a basis of need to know. This means we pick up knowledge of models when we need it to apply ML, or understand how the ML model works. Of course when you don't know what is ahead, you don't know what you need, and trying to plan or even read about how to cross a river, when you are not there, might sometimes be a waste of time. However the theory covered in the coming chapters, is something that we, to an extent of this approach, finds suitable.

The project is therefore not focused on understanding every detail in all the algorithms used. And therefore some of the libraries are partly used, and some aspects of them will be a black box. This project therefore also relies on other computer science experts opinions, found on different forums such as StackOverflow and Quora. While imputation is a big topic, it has been decided by the project team, that a generic approach for imputation is needed. While taking this approach, the theory of imputation was excluded, to limit the scope of the project.

1.4 Target Audience

The target audience of this report are a student that are studying a bachelor's degree with the topic Computer Science as their main subject. The reader should at least be at the final year of their study, or be able to understand a higher level of programming.

2 General Machine Learning

While you were roughly introduced you to what ML is, now let us take a swing at some of the terms we use and care about. In this chapter, you will come across different general ML issues or need-to-know. Firstly, we will start off by describing the bias-variance trade-off, then a way to get the optimal trade-off called nested cross-validation. Then how to prepare the dataset, e.g. by encoding the data. At last, a general cost function is introduced, which is used to evaluate the different models.

2.1 Bias-Variance Trade-off

Bias-Variance Trade-off is the balance between bias and variance to predict a good model. The struggle of getting a model that has low variance and low bias is generally one of the hardest problems in ML [James et al., 2013]. It is known as the *bias-variance trade-off*, which means trying to perfectly balance the right amount of variance and bias, to get the best model that it can be reached. The bias-variance trade-off is perhaps one of the most important things to consider when applying ML.

2.1.1 Bias

Bias is the inability of a ML algorithm to predict the right relationship between the items of the dataset, because a straight line will only capture linear tendencies. Bias causes the model to underfit the provided data and this causes under-fitting the data. In simple terms, it is called under-fitted or high biased model, too simple of a model. High Bias means that the algorithm is not able to use the right features of the dataset, neither understanding the relationship between the features of the dataset. Under-fitting is when both the training data and the testing data produce a poor result.

For many algorithms, it is not possible to calculate the bias. Taking linear regression as an example; it is not possible to find the difference between the actual true underlying values in the data and the expected calculation for the underlying values. This means that bias is more conceptual, rather than something that can be calculated.

2.1.2 Variance

Variance is the difference between fitting the train and test data which cause the machine learning model to give good results on the train data and inaccurate results for the test data. This inaccuracy is caused by over-fitting and learning random information from the features of the dataset that is noise and not relevant to the relationship between the data features. Overfitting is when the training dataset performs good, but the testing dataset performs poorly [Patel, 2018].

2.1.3 Combining Bias and Variance

Here it can be seen what the differences between low and high bias and variance look like. It is preferable to have low bias and low variance since that will have a minimal amount of error in the model. Having low bias and variance is hard to impossible, in some cases, to reach. High bias is the amount the trained model is systematical off the "true" value. High variance describes the "randomness" of the model's predictions around the target.

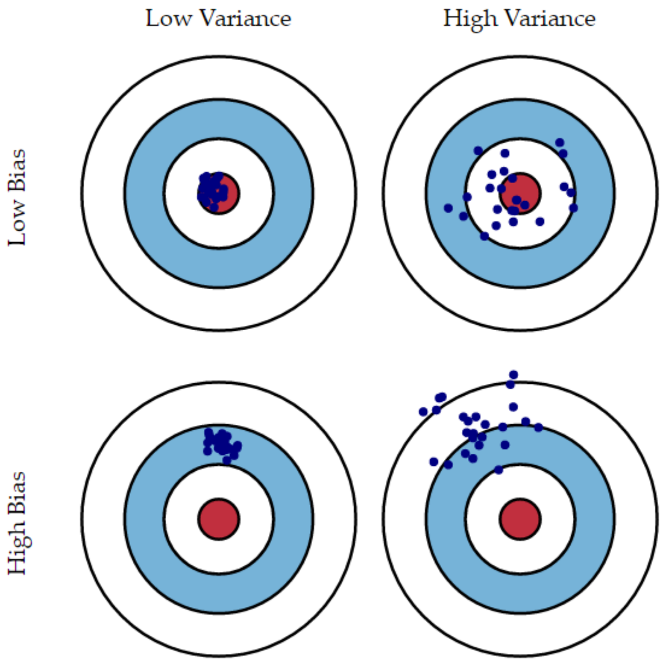

Figure 2.1: An illustration that indicate how low and high bias and variances affect a prediction. [Singh, 2018]

2.2 K-Fold Cross-Validation

Cross-Validation (CV) is a tool that provides a way to figure out, how well a model generally performs. CV works by splitting the data into partitions, denoted by K. The recommended partitions to split into is 5 or 10, since it has been shown that these values give the best results [James et al., 2013]. CV works by dividing the full data, training, and testing combined, into k partitions. One partition is used as a testing dataset and the k-1 partitions are combined into the training dataset. If a K-fold of 5 is used, it will produce 1 partition for testing and 4 for training, which is a 20% / 80% spread between the testing and training dataset. The training of a model is then repeated 'K' times, so all partitions have been the testing dataset once, see figure 2.2. For each iteration an MSE is produced were in the case of K-fold = 5, five different MSEs is produced. To summaries the K-fold CV, the mean of the MSEs are calculated and presented as the final MSE, also referred to as generalization error.

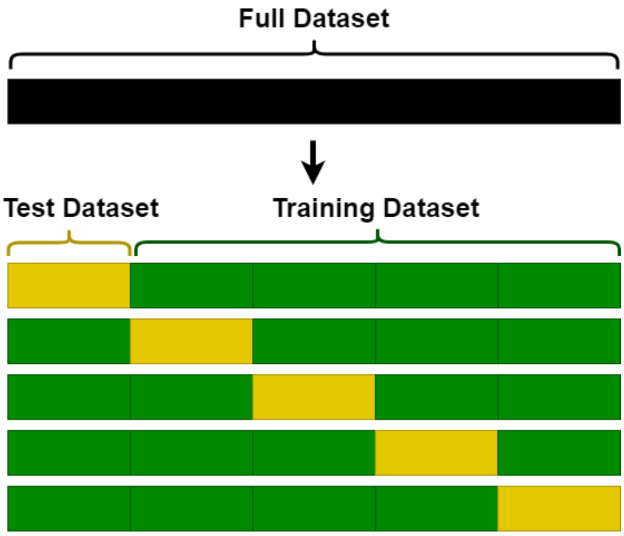

Figure 2.2: Example of cross-validation with $K = 5$

Another way to expand on CV is by doing CV while trying to find the optimal hyperparameters. The way one distinguishes between hyperparameters and parameters is by defining parameters as input to a function, that is not optimized, but hyperparameters as parameters that are optimized. The hyperparameters are used to train different models, where the model with the lowest score will have the best hyperparameters, which would be called optimal hyperparameters. This means one can input a list of hyperparameters with different values, get the best hyperparameters for a model, then train and test the model all in one process.

The model picked by CV is referred to as the best model. This model tends to be too optimistic [Varma and Simon, 2006], meaning that the result in most of the instances has a lower MSE, which is equivalent to overfitting. This means that our model has high variance and low bias, so the model will generally reflect the true relationship between the predicted values and the features in our dataset, but it won't generalize well to new data. Obviously, this is not desirable.

2.2.1 Nested Cross-Validation

One of the more modern approaches, that is not yet too well described and does not have a standard approach in practice, is nested cross-validation. There are many approaches, but 'developer' approach here is suggested. It is called that since it caters more to the developer of a ML algorithm, who would want to find the optimal parameters to run an algorithm.

It is reasonable to ask, *why nested CV is needed?* As illustrated earlier, the reasonable model here is needed. To get a reasonable model, there should be a guarantee to ensure that the model is not over-fitted or under-fitted. To ensure that the model is not over-fitted or under-fitted, the right bias and variance trade-off should be ensured. Though what *is* the right trade-off? This question is hard to answer and has been the subject of many books and articles. As stated earlier, the textbooks and articles generally say, when you have low variance, you have high bias and the reverse. But in practice, that is not something can be generalized to every dataset and modeling situation [James et al., 2013].
Thus, there is uncertainty about what exactly the way to go about estimating what the right trade-off is. But there is a tool called nested CV, that supposedly does the best job of getting the right trade-off; noted by many articles, most notably [Cawley and Talbot, 2010], but also others have demonstrated why this tool is needed [Krstajic et al., 2014][Varoquaux et al., 2017].

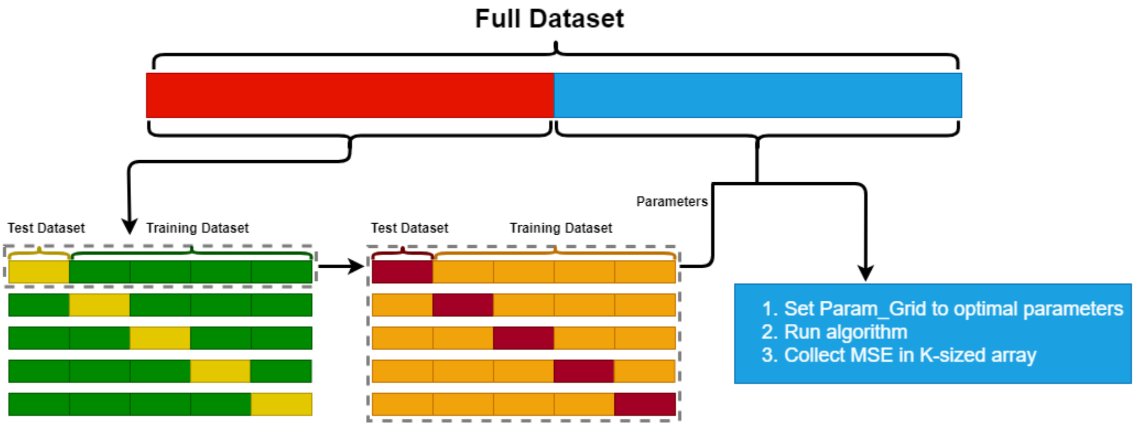

Figure 2.3: Example of nested cross-validation with $K = 5$. Split the full dataset into two splits. Take one of two splits and do CV on that (outer CV). For each K-partition in the outer CV, do CV (inner CV) with K-partitions of each outer K-partition. Optimal parameters for each outer partition are retrieved from inner CV. Run algorithm with optimal parameters and another half of full dataset for the K-th split.

The problem is that bias is not easily estimated in complex algorithms, although there have been attempts to make a general equation [Pedro, 2000]. There is no industry standard for this at the moment, generally saying that when there is underfitting, then there is a high bias. A high bias also usually means that both the training and testing error is high and that the model is too simple. Arguably, it is easy to make the parameters of a model more complex to decrease underfitting, but when stop increasing the complexity? At some point, overfit will be accrued, which means that the parameters of the model need to be more simple.

In the paper by [Cawley and Talbot, 2010], they answer the question of *Is Over-fitting in Model Selection Really a Genuine Concern in Practice?*:

> *"this is a clear demonstration that over-fitting in model selection can be a significant problem in practical applications, especially where there are many hyper-parameters or where only a limited supply of data is available"* [Cawley and Talbot, 2010]

Thus, the argument for using nested CV is to get a reasonable, unbiased view of the performance of models, and to prevent overfitting. Nested CV is the procedure that one should wish to use, given that it is computationally feasible, rather than using CV, which demonstrably produces an over optimistic result.

As demonstrated by Scikit-Learn in the figure 2.2.1, a lower test error should be expected when doing nested CV, than with CV. Since it tends to be too optimistic when doing CV, this should hold true in most instances.

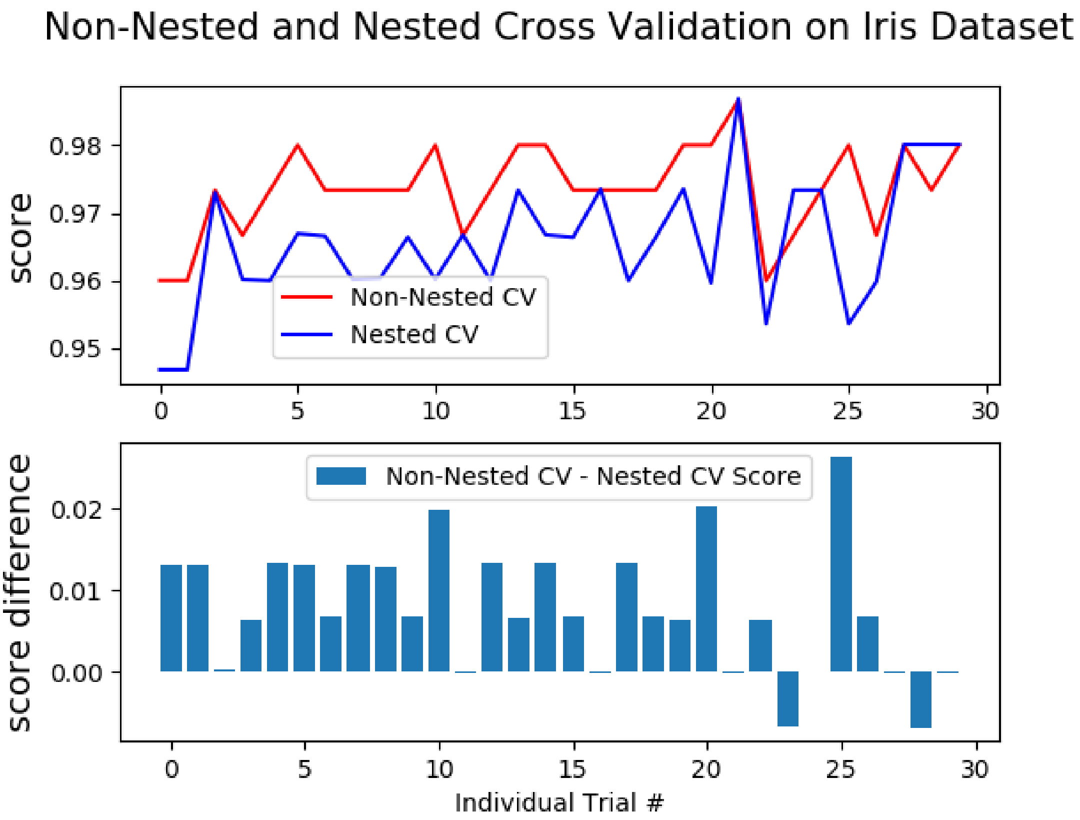

Figure 2.4: CV vs Nested CV plotted with accuracy score against number of trials the model ran for. Image retrieved from Scikit-Learn [Scikit-Learn, 2018]

Although counter-intuitive, if the score is higher, it might not be a better model, because overfitting might have happened. The difference in scores is demonstrated in the figure above. Thus, and as demonstrated by many articles, the best bias-variance should be selected effectively trade-off, by doing nested cross-validation [Cawley and Talbot, 2010] [Krstajic et al., 2014] [Varoquaux et al., 2017].

2.2.2 Hyperparameter Optimization Methods

ML models are not a fixed thing were the behavior can be adjusted by different sets of parameters. There are most likely not a best set of parameter or standard fixed that will give us the best results. At this point in time, it is a try and catch processes to find the most accurate model, by trying different sets of parameters. Hyperparameter optimization or Hyperparameter tuning is the ability of cross-validation to pick the right hyperparameters that will give the model the best accuracy.

2.2.2.1 Grid Search Parameter Tuning

Grid Search is a technique that will try different combinations of parameters and evaluate the resulting model. Grid search will build a model, evaluate it with every combination possible and by the end choose the most accurate one. This technique has some disadvantages, like it takes a lot of time because it consider all of the possible combinations, so the run time depends directly on the amount of parameters and the amount of values a give parameter has.

2.2.2.2 Random Search Parameter Tuning

Random Search is a technique used to generate random combinations of the hyperparameters to find the best fit for the model. These combinations of hyperparameters will be tested and evaluated on a model and take the most accurate one. It will not cover all the possibilities of the hyperparameters because it runs only for a fixed number of iterations but there is a high chance that it will find the most optimized hyperparameter because of the random search pattern. It's useful when there is a wide range of parameters to be tested, because it will take less time to find the best random combination. After all it is a random search and therefor there is no guarantee that it will be the best parameter combination of the provided range. The problem with random search is that it can produce high variance because the parameters are totally random, and it's used for a specific trained model that may not fit the test data.

2.2.3 Pseudo Code - Algorithm 1

The importance of nested CV should be clear now. It is worth providing a higher level view of how the algorithm operates, in pseudo code.

Algorithm 1: K-Fold Nested Cross-Validation with Random Search

```
Require: $K_1, K_2$, where $K_1$ is number of outer folds and $K_2$ inner folds
Require: $\mathcal{D}$, dataset containing input features $X$ and output feature $y$
Require: $P_{sets}$, set of hyperparameters with different values
Require: $\mathcal{M}$, a single estimator, model.
for $i = 1$ to $K_1$ splits do
    Split $\mathcal{D}$ into $\mathcal{D}_i^{train}$,$\mathcal{D}_i^{test}$ for the i'th split
    for $j = 1$ to $K_2$ splits do
        Split $\mathcal{D}_i^{train}$ into $\mathcal{D}_j^{train}$,$\mathcal{D}_j^{test}$ for the j'th split
        foreach $p$ in RandomSample$(P_{sets})$ do
            Train $\mathcal{M}$ on $\mathcal{D}_j^{train}$ with hyperparameter set $p$
            Compute test error $E_j^{test}$ for $\mathcal{M}$ with $\mathcal{D}_j^{test}$
    Select optimal hyperparameter set $p^*$ from $P_{sets}$, where $E_j^{test}$ is best
    Train $\mathcal{M}$ with $\mathcal{D}_i^{train}$, using $p^*$
    Compute test error $E_i^{test}$ for $\mathcal{M}$ with $\mathcal{D}_i^{test}$
```

2.3 Encoding data

Many machine learning models can only deal with numbers and it can't deal with a strings, so in House Prices dataset there are a couple of fields that have an empty Values (NaN) and they also have the string type, so this string types need to be converted to a format that the algorithm can understand and take to give us the results that we're looking for, after searching for the available ways of changing the strings to numbers, there are a couple of ways has been found:

- One Hot encoding
- Label Encoding.
- One Hot Encoding with (Dummy variables)

2.3.1 Label Encoding

Label encoding is a mechanism to convert the string data to a format that the computer can understand and the way that label encoding works are by replacing each of the values in the dataset with a number that represents that value. An example would be a column called *"HouseStyle"* which contains the style of dwelling which has values such as: 1Story, 2Story, SLvl, 1.5Fin and more. The label encoding will give each one of these values a number to represent it and in this case:

House Style	**House Style value**
1Story	1
2Story	2
SLvl	3
1.5Fin	4

And with that, label encoding will treat them as a number so the higher the number is, the more important will be. This is not right in this case it's just a values and it's different from house to house and from area to another, so an another methods that suits the case would have to be found.

2.3.2 One-Hot encoding

One hot encoding is different from label encoding in a way that it does not convert the values into numbers, but instead converts the string values into dataset's columns. If a entry has the given feature it is assigned 1, if not it is assigned 0 in the new column. An example of this would look like this:

1Story	2Story	SLvl	1.5Fin
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

By using One-hot encoding it will be more clear, because it is converted to boolean values, to the ML algorithm to recognize this value as a different choice rather than a higher priority.

2.3.3 One-Hot Encoding with Dummy variables

One-Hot Encoding with Dummy variables follows the same way as One Hot encoding and the only difference is it drops one dimension. The first dummy variable from the converted columns to avoid dependency among variables, a categorical variable of K categories, or levels, usually enters a regression as a sequence of K-1 dummy variables. This amounts to a linear hypothesis on the level means [Mortazavi, 2018].

> *"A categorical variable of K categories, or levels, usually enters a regression as a sequence of K-1 dummy variables. This amounts to a linear hypothesis on the level means. So it will be different from ML model to another model, different types accept different formats."*
>
> [Perktold et al., 2018].

2.4 Cost Function

The cost function or Mean squared error (MSE) will give us an idea of how well a line, from linear regression or the other model, fit some data points. The lower the value the better the fit. If MSE equals zero the predictor function $\hat{y}$ goes through all data points. This is normally a sign of overfitting. Commonly MSE can be used together with a regularization term to reduce overfitting, more on that in the regularization chapter.
However a common equation for MSE [Miller, 2018] looks like this :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2 \tag{2.1}$$

to understand this equation recall that a data set D consists of $\{X_i, y_i\}$ and with i being the number of rows. X is a matrix of rows(n) and features(m) without the target(y) vector in our case y is all the sales prices in the dataset.
The equations 2.1 say that one takes the difference between each predicted price ($\hat{y_i}$) and each actual price (y). This difference is called a residual. See the figure: 2.5. Each residual is then squared. all the squared residuals are then summed. and finally divided by the number of residuals the same as the number of points(n).

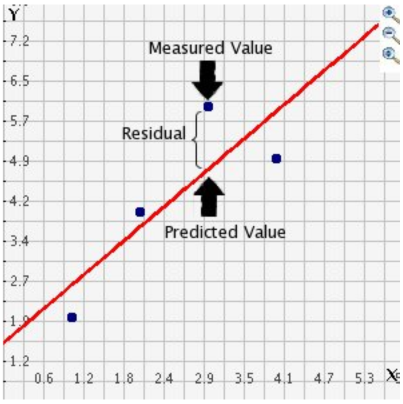

Figure 2.5: an example of residuals. The red line is an arbitrary predicted linear model($\hat{y}$), the vertical difference between the line and the points(y) are called the residual (y-$\hat{y}$), each of the residuals is squared then summed and then divided by the number of residuals

More examples of calculating the cost function can be found in the appendix.

2.5 Gradient Descent

Gradient descent (GD), is a way to minimize a cost function, which is common when applying machine learning to a problem.
The method is especially suited when the number of variables is larger than 10.000 [andrew NG, 2019]. This turns out to quite use full for neural networks since there are many weights and biases that can be used as parameters. The method is arranged in such a way so it takes decreasingly small steps in the steepest direction until a local minimum is found. see example in the figure below 2.6

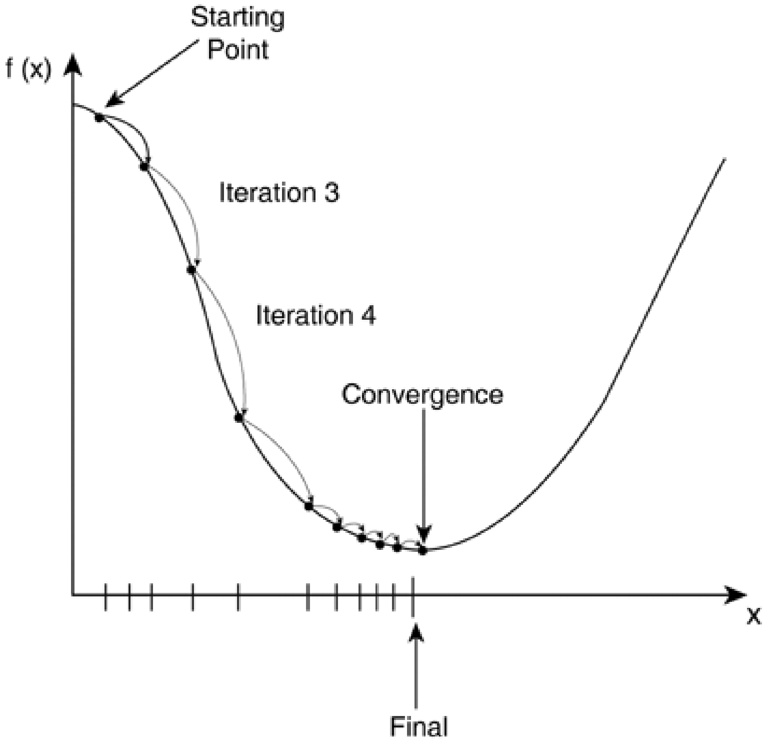

Figure 2.6: The figure illustrates the reduction in step size as the slope is getting less steep. at the end it converges, when the slope is flat. credit[McDonald, 2017]

(There are more information about the gradient and gradient descent in the appendix.)

2.6 Feature Selection

Is one of the important concepts in ML because it affects the performance and the accuracy of the model. Feature Selection is the mechanism that selects the features or attributes manually or automatically, which have the most contribution to the predicted values or the output of the model. The benefits of using feature selection is to reduce training time, since the problem will get reduced as fewer columns have to be processed. This will lead to a quicker processing time and a less complex model. At the same time feature selection will reduce overfitting because redundant or unrelated data affect the model's decision less. This reduce the amount of noise the model will have to base its decision on, which will improve the accurate and the amount of errors in the predictions of the model.

2.6.1 Feature Selection vs Dimensionality Reduction

Both concepts are trying to reduce the amount of the features that are use in a given model. The core difference is, feature selection is including or excluding the columns of the data with changing them, where dimensionality reduction is combining multiple attributes together.

2.6.2 Types of Feature Selection

Univariate Selection is a selection method can be used to select the features which have the strongest relationship with the predicted values by using statistical tests. One of the most famous tests is called Chi-squared, which is used for non-negative features. This type is called Filter Methods.

Recursive Feature Elimination (RFE) is selecting different combinations of data attributes and building a model with these attributes recursively. This is done to find the best combination of the different attributes by using model accuracy. This type is called Wrapper Methods.

Choosing the right way of using feature selection is different and does not guarantee the best results. The best way to know is by trying until the best score is reached. To do this some different APIs are available in Python for different ML Models which give the related attributes and how strong relation they have to compare to the predicted model.

2.7 Evaluating Models

In the following section evaluation of models will be presented.

2.7.1 Metrics

Metrics are used to evaluate, measure the performance and specify how accurate is a specified model and how it can be compared to another model, there are different types of metrics algorithms to evaluate the accuracy of the algorithm, there are specific ones for classification and some other for regression, in this part the focus will be on the regression algorithms since it's related to our problem.

2.7.1.1 Mean Squared Error (MSE)

While there are many evaluation metrics, what is usually referred to is a metric called mean squared error. This refers to when you have trained a model using the training dataset, which has data points, that will be referred to as the true estimated class or value:

1. Subtract the predicted value from the true estimated class or value
2. Square the result
3. Sum over all predicted values for n observations.
4. Find the average (or mean) of all predicted values

This process is referring to an equation, where lower is better:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2.2}$$

This gives us some way of evaluating how well each model predicted classes or values for us. The result of the equation is entirely dependent on the values of the dataset, meaning that the MSE cannot be generalized from one learning problem to another. Of course, there are many ways to evaluate models, but this equation is the one that will be used for this report.

2.7.1.2 Root Mean Squared Error (RMSE)

Root mean squared error, sometimes called root-mean-square deviation, gives an idea of how much the residuals deviates from the model on average. The closer the RMSE is to zero the better the fit.

This could be used when you have a scatter plot that seems to fit with a line. If the points in one end of the scatter plot are much further away from the rest the method aren't that use full since you get the average.

Since the way its calculated is by squaring each residual and then average them and taking the square root. When you then take the root, you get a number in the same scale as the residuals. The equation looks like this:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y_i})^2} \tag{2.3}$$

Its important to note that there can be a problem with using these methods since there might be a big difference in size of residuals in one end of the spectre compared to the other. For instance houses prices may be of orders of magnitudes different. In that case an other method is needed one way is RMSLE.

2.7.1.3 RMSLE

RMSLE (root mean square logarithm error) can deal with problems of large difference in residuals. However RMSLE penalizes underpredicting estimates greater than over predicted.[Drakos, 2018] this is the equation:

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(Y_i+1) - log(\hat{Y_i}+1))^2} \tag{2.4}$$

3 Machine Learning Algorithms

In this chapter the different models are presented and described. The description will be limited to give a general understanding of the models and how the algorithms work. The algorithms will primarily be described by a step in the iterative process the algorithms runs. In total 4 models will be covered in this project. First a brief introduction to multivariable linear regression, which is used as a baseline model. The second model is random forest and its underlying concepts of bagging and decision trees. The third model is XGBoost and the corresponding concepts of boosting and gradient boosting machine. The last model is a neural network which will have a brief introduction of the neural network and the concept of dropout.

3.1 Multivariable Linear Regression

Linear regression is a supervised machine learning algorithm. The algorithm predicts a range of continuous ranged values, like a sale price. As should be obvious, linear regression is only used for regression, and not classification.
A multivariable linear regression equation might look like this:

$$f(x,y,z) = w_0 + w_1 x + w_2 y + w_3 z \tag{3.1}$$

The function variables represent the dataset features (x,y,z) and it could be any feature from our dataset, e.g. HouseStyle.

$$HouseSalePrice = w0 + w1 \times Utilities + w2 \times Neighborhood + w3 \times HouseStyle \tag{3.2}$$

these features can be taken, then assign a weight to them, also called a coefficient. And since the number of features available is reaching above 70, the complexity of the data is increasing, and so it the intuition of how to visualize all of the data at the same time. Therefore, some features can be picked and visualize some of them together, while exploring how they impact the house prices.

3.1.1 Predicting Values with Linear Regression

The linear regression function will try to estimate the house price given the current coefficients and the dataset features. The best will be the one with the smallest residual error. The data points form a cloud where different coefficients are tested to get the best fitting linear regression model. For each of the different models, a line through the data points is tested. For each new testing data point from the test dataset, the distance is calculated to the plotted line. The difference between the true value y (from the training dataset) and the predicted value $\hat{y}$ are summed together for all observations and squared. This is also referred to as the total sum of squares

$$SS_{tot} = \sum_{i=1}^{n}(y_i - \hat{y_i})^2 \tag{3.3}$$

3.2 Ensemble Methods

One of the first concepts to explain is bagging and boosting. Later we need them for understanding random forest and gradient boosting. Ensemble methods are where we use multiple models to predict output values.

3.2.1 Bagging

Bagging is an ensemble method, where randomly taken subsets of the original dataset is used, to form a new dataset. We are randomly sub-sampling, and therefore we say that we perform bagging with replacement, meaning that the same points might occur in the same dataset and some points might be omitted. With bagging, we train a model on each of these subsets. When doing regression, we could train many different models of the same kind, i.e. training a 100 least squares regression models, then averaging the result. A short model for this is summing from 1 to M regression models, using the same regression model for each subset of the data D_r.

$$y = \frac{1}{M} \sum_{m=1}^{M} f(D_r) \tag{3.4}$$

With this equation, it will be sum over the result, $f(D_r)$, of all our regression models, using *r* different subsets of our original dataset ($\sum$) from m to M, starting from $m = 1$. A term often used for bagging is saying that there is *bagged M models*. Then it can be said that *y* is the result for each model.

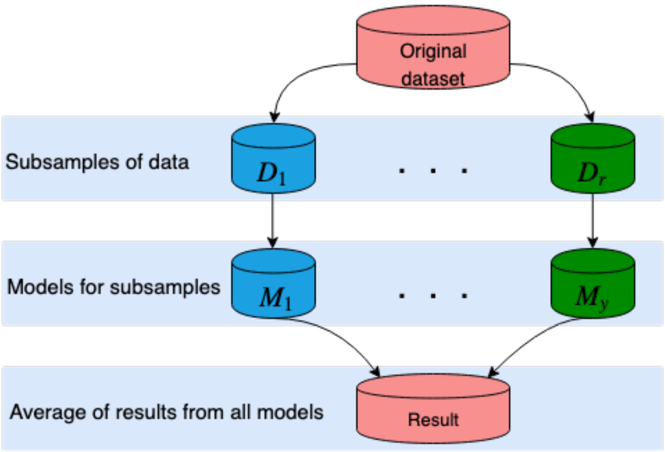

Figure 3.1: Bagging with regression

3.2.2 Boosting

Boosting is an algorithm that helps ML models give us better predictions. Boosting sub-samples the data, like bagging also does. So we create a sub-sample multiple times and make a model for it. The general idea behind boosting is assigning a weight to each observation. This is done because some observations are harder to classify or calculate a value for, and want to specify which are easy and which are hard. At first the weight is the same for every observation. Then some predictions for the observations are calculated, and the weights updated for each observation. The weight for an observation will be low when it was easy to classify or calculate the value for it, and high when it was hard. After having calculated the weights for each observation in the first sub-sample, a new sub-sample is created. The new model for the second sub-sample would then try to predict the first sub-sample even better. In this sense, the observations that were hard to predict should now be easier to predict. The observations that were hard to predict had a weak learner, and which are made stronger by creating new sub-samples with new models, that predict them better. This process continues, which most of the time results in a very good algorithm. Later on, we will see an example of an algorithm that uses boosting, called gradient boosting machine.

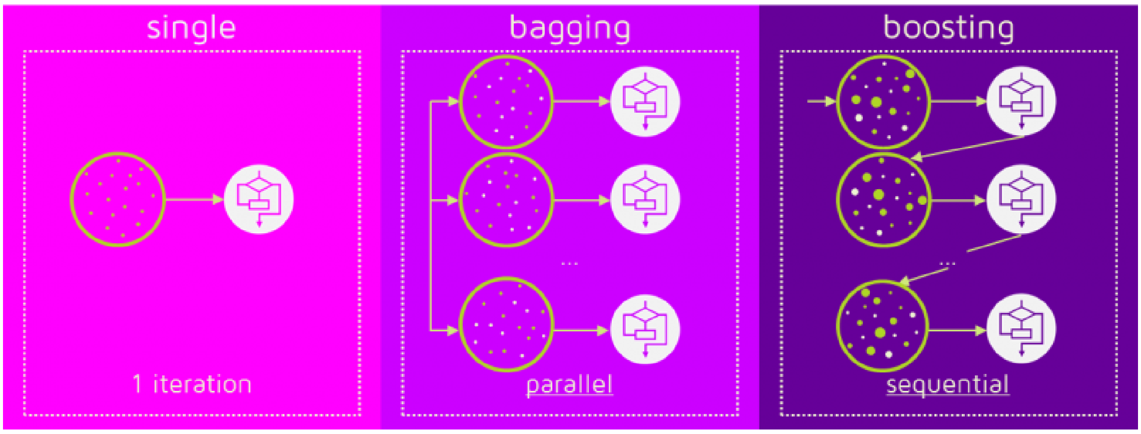

Figure 3.2: Single, Bagging and Boosting pipeline [D'Souza, 2018]

3.3 Decision Trees

Decision trees can be summed to asking a question, where the following answer will be a branch in the decision tree. questions keep asked, and from there, many branches emerges from each question. Naturally, you would follow the correct answer, e.g. with the question *is it raining today?* The answer is binary, and if the answer to that question is yes, it is raining today, then we follow that branch.
To build a decision tree, we need some problem or main question that we want to answer. In other words, we make decision trees to try and answer another question. Then in the decision tree, you ask relevant questions that influence the answer to the main question. The question of *is it raining today* could be in a decision tree, where the main question is *should we go to the beach today?* Luckily, this turns out to be easy to visualize, as seen by the figure 3.3.

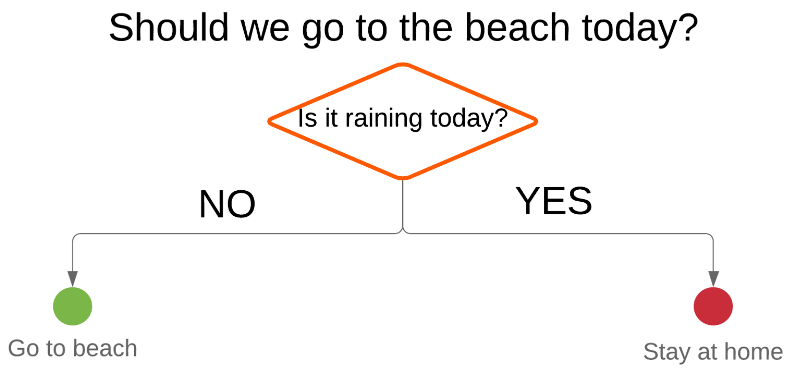

Figure 3.3: Decision tree figure.

We have yet to introduce the three types of nodes in such a decision tree. There are three types of nodes; Root, Internal and Leaf Nodes. If we are to consider the below figure, we can introduce the three types of nodes. Firstly, the root node, which is the first node. In this example it is the diamond with the question *Cold blooded?* Internal nodes are the questions after the root node, which has one or more answers to them. The answers to an internal node (or sometimes root node) is a leaf node. In this case, we can say that non-mammal, where we placed snake, is a leaf node, whereas the question of if the animal lays eggs is an internal node. Then the answers to that question is also leaf nodes. What has been described here is referred to as Hunt's algorithm [Hunt and Kubler, 1984].

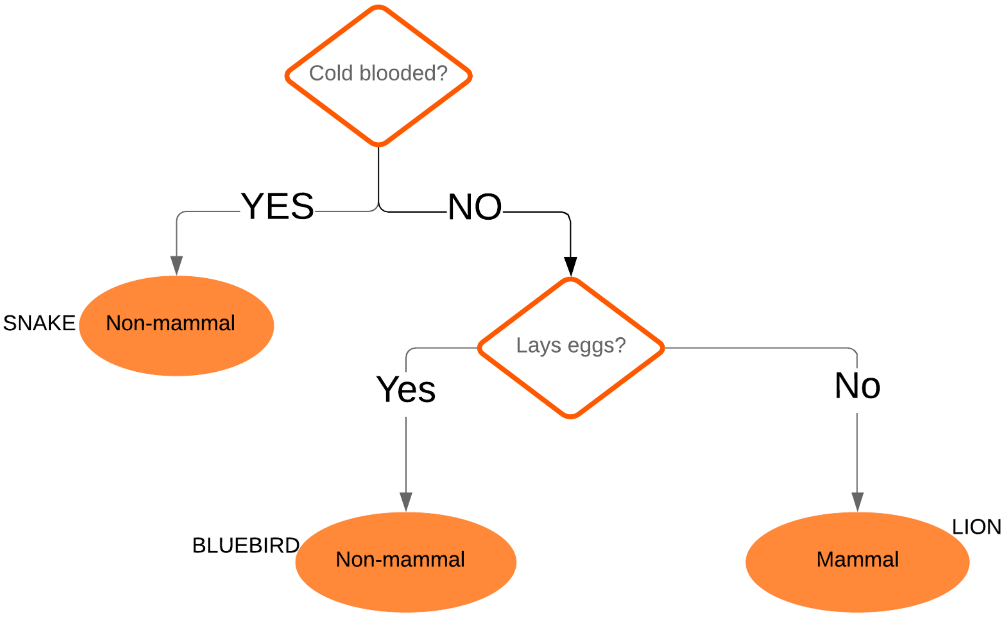

Figure 3.4: We consider three animals. A snake, a bluebird and a lion. First we ask, is the animal cold-blooded? If the answer is yes, then it must be a non-mammal, and we place the snake there. If the answer is no, we ask another question, does it lay eggs? If the answer is yes, the answer then it is a non-mammal, which is where we place the bluebird. If the answer is no, then it must be a mammal, which is where we place the lion.

Decision trees don't just choose which question to ask (i.e. what feature to pick). There is a process that goes on, when choosing what the next question might be. The decision trees consider every possible value for all variables, making it quite a greedy algorithm. This is referred to as impurity and purity gain, where we calculate impurity at the root node and for each potential internal node. Afterwards, the purity gain is calculated by subtracting the impurity at the root node by the impurity from the questions we asked, at the different internal nodes.

This is usually denoted as Δ. At last, after this process, we would have the decision tree, that is the most *pure* at each step, i.e. where the algorithm picked the best question and values for the question every time we had to make a new internal node with leaf nodes.

3.3.1 Shortcomings of Decision Trees

Decision trees have some shortcomings. Since the algorithm only calculates the best variable to pick for the next internal node, it is only given a picture of what the best step for the next internal node, but not what the best step overall is. It could be explained such as that the decision trees only consider local optimums, where it generally is preferred to consider global optimums, that is the best decision tree overall. Another shortcoming is how much decision trees overfit. While a decision tree might pick the most relevant variables for the next internal node, it does not mean this is a good pick. Suppose the dataset consists of 1000 observations with 80 features. One might imagine that if just 10 of these features are categorical, with 5 different values for each feature, the resulting decision tree would make decisions based on very little data. This is not optimal for learning some prediction rule, because it is making decisions based on little to no evidence. Thus, it might be found that the more questions that is asked, the less relevant these questions are in the overall scope of how well the decision tree performs.

Some ways to combat these shortcomings is by:

- Pruning, setting a limit for how many questions we ask.
- Stop asking questions when a new internal node contains less than some number of observations.
- Setting a limit on when we have enough purity gain, and then when we reach that threshold, we stop asking questions.

3.3.2 Random Forest — Regression

Random Forest is a ML model to predict values and it supports classification and regression. It is built on top of decision trees and it uses bagging ensemble method to build its predictions, the way it works is by building a forest of totally independent random decision trees to predict the best results. With bagging, the original dataset is split into sub-samples of the dataset. The way the dataset is split is by replacement, meaning that the same observations can occur multiple times in a single sub-sample, and there might be observations that are totally excluded.

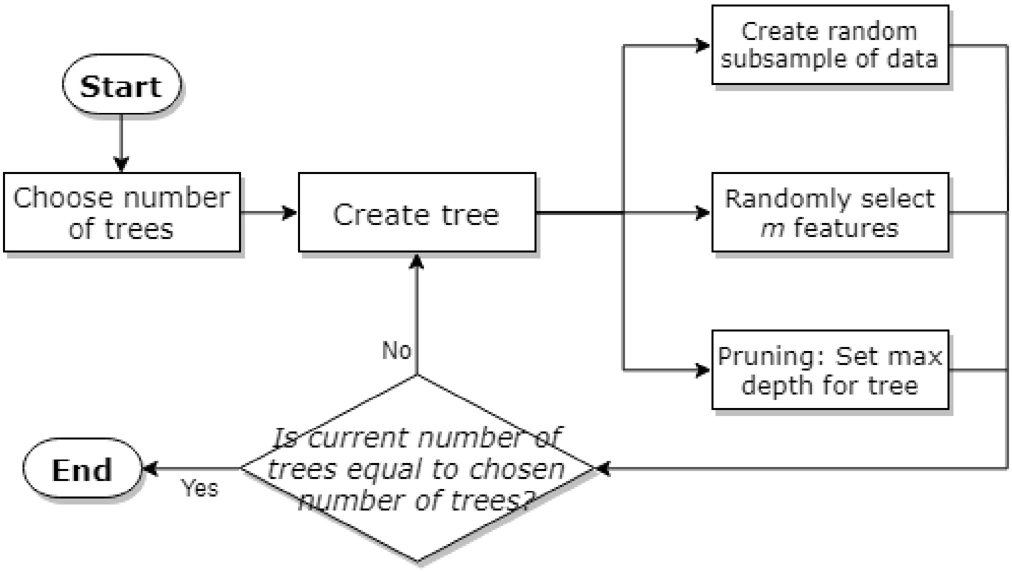
Start
Choose number of trees
Create tree
Create random subsample of data
Randomly select *m* features
Pruning: Set max depth for tree
No
Is current number of trees equal to chosen number of trees?
Yes
End

Figure 3.5: A simple flowchart of how a random forest is created.

When making a tree in RF, a decision tree with some special rules are made and randomness added to them. Opposite to decision trees, not all features are considered in RF and only a finite number of features are considered. The way the finite features are selected are by random, hence the term ’random’ forest. In practice, the maximum number of features is chosen, and the actual features themselves are randomly chosen by the algorithm. Pruning is introduced, as described in 3.3.1, where the depth of the tree is limited. After all these steps, a tree in the RF is complete.

In essence and to combine decision trees and bagging, an arbitrary number of trees in the forest is made. In practice, it’s just a hyperparameter, that is set to some specific value, e.g. 1000 to generate 1000 trees in the RF. Each tree will produce some prediction when given data, and to get a final prediction all the resulting predictions are averaged. Now a RF is generated and should be able to make predictions based on testing data.

3.4 Regularization

Regularization is a technique that can be applied to any machine learning models to avoid overfitting. The problem with overfitting is that it will give inaccurate results when used on a new test dataset. Overfitting happens when a model is adjusted for a specific dataset and it will not give accurate results for a different dataset whether it's generalized or from a different source.

3.5 XGBoost

To explain XGBoost, at first gradient boosting machine have to be explained because XGBoost is an optimized version of that. Then in a subsection, we explain where XGBoost 'wins' with regularization.

3.5.1 Gradient Boosting Machine — Regression

Knowledge of the ML algorithm Gradient Boosting Machine (GBM) is necessary for understanding some underlying aspects of the popular machine learning XGBoost. GBM for regression problems can predict values by summing up a sequence of decision trees, called boosting. Each tree contributes to a narrowing process and is gradually closing in on the prediction. See figure below for an illustration giving a brief overview (fig: 3.6)

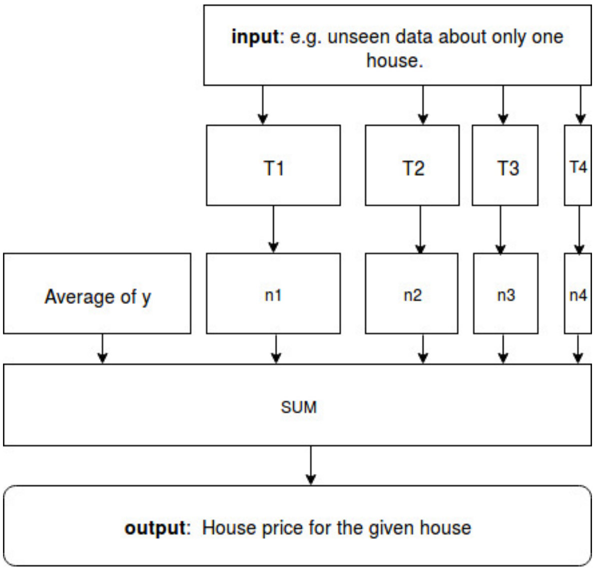
input: e.g. unseen data about only one house.
T1
T2
T3
T4
Average of y
n1
n2
n3
n4
SUM
output: House price for the given house

Figure 3.6: Flow chart of Gradient boosting regression: Each decision tree is represented with T, and n is the output of T. Each value from n is in different sizes to illustrate, that the output of some trees gives bigger and smaller output values n — the significance. The *average of y* is the average of the output feature y, and it is supposed to be the largest. In other words in the series of the tree, each one contributes less and less to the final sum or prediction. This is a simplified example, in reality, there are hundreds of decision trees. The output value of each decision tree is also multiplied by a common fixed learning rate (ν) that apply to all T's (ν is between 0 and 1); this is to reduce overfitting.

The first *tree* is the average of the *target* column. The target column in the *house prices* dataset is all the sales prices in the training set. taking the average is will give a constant value. The average "sales price" can be thought of as a rough starting point. Then as each trees, small prediction value is added gradually, GBM *focuses* in on its prediction through a series of small adjustments. This focusing process is a descending process. Like gradient descent. Each step is smaller than the last. Hence the name Gradient in GBM. Though gradient decent and GBM is not the same. Its just that they both have a gradient.

It should be mentioned that the values given by a decision tree can be both positive and negative. This is important since this allows for predictions below the average starting point.

3.5.1.1 How GBM is trained

To examine the workings of the GBM algorithm one can study the article "Greedy Function Approximation: A Gradient Boosting Machine" [Friedman, 1999]. A simple overview of the training of the algorithm is review in the following. To train the GBM regression one needs a loss l function a data set D with features and a target y vector and a number of trees M to iterate over.

Step 1:

- Make an initial value. This is the average value of y. This will be called f_0 and this is a constant.

Step 2:

- Make a loop for m=1 to M.
- Each loop produces a function that consists of a new decision tree plus the previous decision tree(s). The first "tree " is the constant value of f_0 from step 1.
- F_1 will then consist of a new tree and the constant value. F_2 is then the next tree plus F_1.

Final step:

- Output the current model F_M.

3.5.1.2 Step 2 — Closer look

Here a closer look can be taken at what happens in the loop in step 2.

1. Find the pseudo residuals (r) i.e. calculate the value of $f_mi - y_i$, where i is a row in D in the first round this is the current function $f_0i - y$ this is done for each of the rows(i) in the data set$D$.

2. Make a decision tree from the data and fit to r.

3. Compute multiplier γ. There are different versions of the algorithm where γ is the same for all trees or different for each tree.

4. Update model: Now there is multiply the new tree by the parameter γ and add this to our previous model.

3.5.1.3 XGBoost — Regularization

It should be mentioned that XGBoost is GBM with regularization, XGBoost has many other advantages of speed and parallelization though no to be covered here. However, XGBoost consists of an objective function that is needed to be minimized as much as possible the objection function consists of two added terms a cost function and a regularization term. The claim is that XGBoost performs better since it has this regularization term. The differences between GBM and XGBoost is this regularization term. [Chen and Guestrin, 2016]

3.6 Neural Networks

Neural networks are a group of models that are able to process information through the help of neurons. The idea behind neural networks has its basis in the human brain, where neurons are connected in many different ways. When a neuron is activated, the activation will propagate to the connected neurons by synapses. Likewise a node in a neural network acts like a neuron in the brain, when it is activated the signal will propagate to the connected nodes. [Tue Herlau and Mørup, 2019].

3.6.1 The Neural Network

There are a few different kinds of neural networks which is classified accordingly as, to how the information propagates, how the nodes are linked in the network and if the information travels in one or two directions in the network. In this project we will only talk about the feedforward neural network, where in information only flows in one direction. [Tue Herlau and Mørup, 2019].

3.6.2 Feedforward Neural Network

A neural network consists of one or more input nodes, zero to many hidden nodes and finally one or more output nodes. For a feedforward neural network, these nodes are typically arranged in layers, where all input nodes are in an input layer, the hidden nodes in one or more hidden layers and the output nodes in an output layer. For a neural network to be a feedforward neural network, the information has to flow from the input layer to the output layer in one direction. In figure 3.7 a graphical representation is showed, where an input layer with 3 nodes are connected to all of the nodes in the first of the hidden layers. The hidden layers propagate the information from the input layer, towards the output layer by passing the information through the first, second and third of the hidden layers. [Tue Herlau and Mørup, 2019].

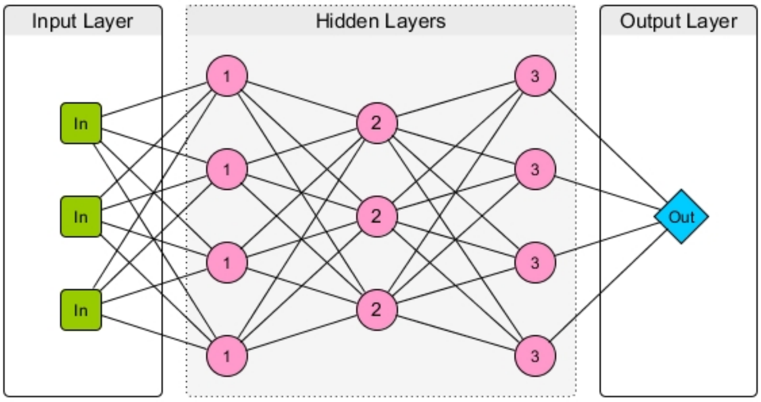

Figure 3.7: Feedforward neural network.

In this feedforward neural network, a given node in a layer is connected to all nodes in the layers behind and in front of it. In figure 3.8, a node from a feedforward neural network is showed that is connected to two nodes in the layer behind it and two nodes in the layer in front of the node. The arrows show how the information only is able to flow from the layer behind the node to the layer in front of the node. [Tue Herlau and Mørup, 2019].

3.6.3 The Node

When a node receives information, it happens by the signals from all the nodes in the previous layer is added together. The value of this sum of input signals is then passed into an activation function which defines what this node's output value will be. The output value of the node is then passed along to all nodes in the layer in front, where the output value is uniquely modified by a weight specific to a given connection between the two nodes.

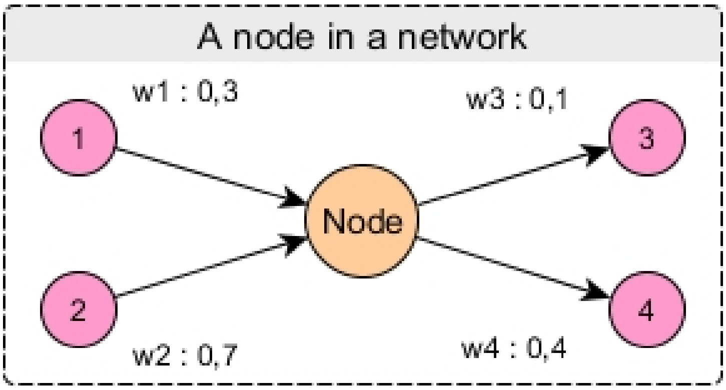

Figure 3.8: A node in a network.

In figure 3.8 it can be seen that the received information was modified by node “1” and “2” with the corresponding weights, and the output of node to “3” and “4” is modified by weight “w3” and “w4”. The neural network will be able to “learn” by adjusting these weights between the different nodes. The ability of the neural network to automatically adjust the weights between different nodes is called backpropagation [Schmidhuber, 2015]. Backpropagation is an important topic when diving deeper into neural networks but backpropagation is outside the scope of this project and will not be discussed further.

3.6.4 Dropout

One way to deal with overfitting in neural networks is a method called dropout. The idea behind dropout is to randomly disable some of the nodes in the neural network. By disabling some of the nodes the neural network is forced to find other pathways for a feature to become relevant in the given prediction. By having more pathways for a given feature to be relevant, the neural network as a whole will become better at generalizing instead of remembering the training set.

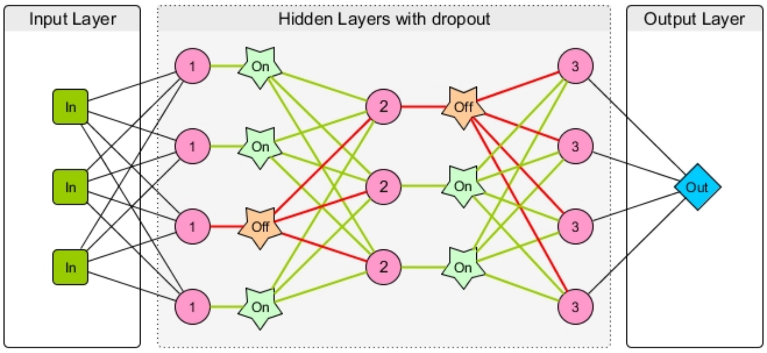

Figure 3.9: A network with dropout layers.

At figure 3.9 a neural network with included dropout layers are showed. Compared to the neural network showed in figure 3.7 the only differences are the inclusion of the 2 dropout layers. The nodes in the dropout layers are illustrated by a red or green star where the color depends on if the dropout node is turning the signal "on" or "off" for the node in the layer before. At the top node in layer 1, the corresponding dropout node is turned on and the signal is passed on to all nodes in layer 2. The top node in layer 2’s corresponding dropout node is turned off, which stop the signal from getting to the nodes in layer 3 [Jimmy Ba, 2013].

4 Implementation

In this chapter, we will present both how each part of the program work and provide our general pipeline.

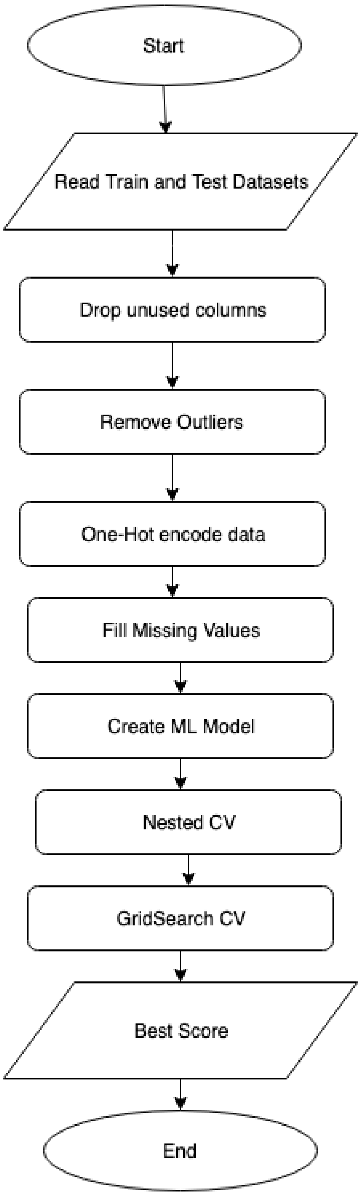

Figure 4.1: The end-to-end flow of for each algorithm.

4.1 Data Exploration and Preparation

First, we will start by analyzing some of the data, manipulating it, and then move on to ML modeling.

This snippet tells about the size of the training and testing set. `.shape` gives number of rows and columns. The different data types that are in the training data is given by `.dtypes` from the library `pandas.DataFrame.dtypes`. `.describe()` is a way to see the whole data set and relevant statistics.

```python
print(train.shape)
print(test.shape)
print(train.dtypes)
print(train.describe())
```

In the following code snippet, a simple test is done to see if there is correlation between *LotArea* and *SalePrice*. Doing this shows that the correlation is not obvious. Here using the library *seaborn* is used as *sns*.

```python
# We can plot with LotArea vs SalePrice
sns.scatterplot(x='LotArea', y='SalePrice', data=train)
```

To prepare the data we need to see what values are missing if any. This can be illustrated in a matrix using the library *missingno*. where the missing numbers are blank. and the values filled are black. this gives a quick overview of the data. We can see if there are columns filled with NaN (not a number) and missing values.

```
missingno.matrix(train, figsize = (30,10))
```

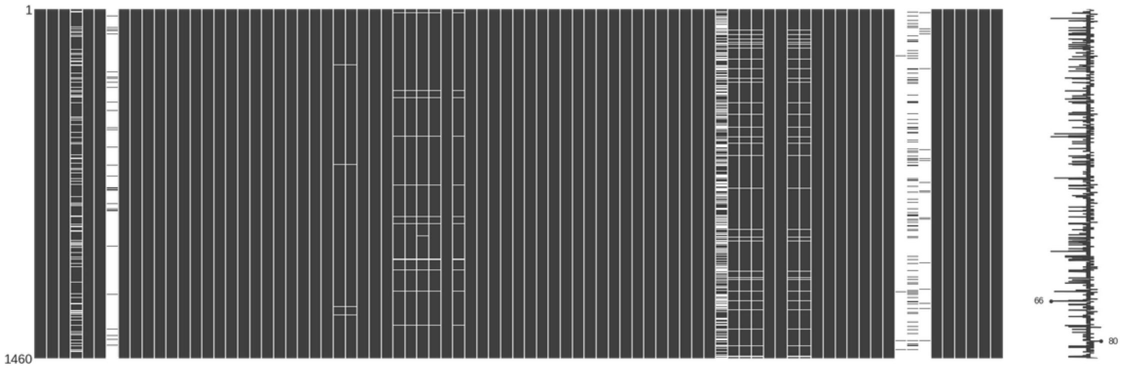

Figure 4.2: The NaN values are illustrated as white vertical lines, whereas the black lines indicates that there are values present.

This snippet is basically showing a function that returns the missing values of a data frame which is another name for dataset, but with labels on each feature. To get an overview of the features with the missing values listed in an array. We are here finding all the missing value for each of the features and returning it in a "dictionary" to get a quick overview.

In the loop *column* is the iterator, this is not a number, but the actual name of the column in the dataset, e.g. 'Alley' in `train['Alley'].value_counts().sum()`. Printing this would result in the number 91, which is the number of values that are not missing in the Alley feature.

In the rest of the loop the number of real features is then subtracted form the length of the data set this is assigned to the `missing_values[column]`.The whole block starts when `all_missing_values_by_feature` is called.

This method is to find the missing values of the dataset — it will loop through all the columns and get the value counts of each column then subtract it from an actual number of rows. By the end, the method itself will return a dictionary of with feature names and the associated number of missing values. `value_counts` is a function from Pandas and it will not consider NaN values or missing values.

Finally, this function will return features and number of observations where there is missing values.

```python
def find_missing_values(df, columns):
    missing_values = {}
    print("Number of missing or NaN values for each column:")
    df_length = len(df)
    for column in columns:
        total_column_values = df[column].value_counts().sum()
        missing_values[column] = df_length - total_column_values
    return missing_values

all_missing_values_by_feature = find_missing_values(train, columns=train.columns)
all_missing_values_by_feature
```

Here we *join* or concatenate the training and testing datasets. This makes it easier to handle and saves some lines of code. In order to change the categorical features, we use one-hot encoding by the function `pd.get_dummies`.

```
19 # Concatenate all of training and testing data
20 cc_data = pd.concat([train, test], sort=True)
21
22 # One-Hot encoding of the whole dataset
23 cc_data = pd.get_dummies(cc_data, prefix_sep='_')
```

Next is another function, where `fill_ii` is used to impute missing data. From the documentation, this will *"model each feature with missing values as a function of other features, and use that estimate for imputation"*, which is the process of filling in missing values in a dataset.

On line 26, the function fills in the missing values (imputation). The way it works behind the scenes is by using a form of an iterative process to loop over each of the features that have NaN values and then replace them. It calculates a correlation from the remaining features, from which the function takes an educated guess of what the imputed value for each NaN value should be. Still, on line 26, the `fit_transform` is where the function actually imputes (fit) and then transforms (transform) the data that was used as input.

```
24 def fill_ii(df):
25     # Make a dataframe of the newly calculated values from the IterativeImputer
26     df_filled_ii = pd.DataFrame(IterativeImputer().fit_transform(df.values))
27
28     # Assign columns and indexes as they originally were
29     df_filled_ii.columns = df.columns
30     df_filled_ii.index = df.index
31
32     return df_filled_ii
33
34 # Fill the missing data
35 cc_data = fill_ii(cc_data)
```

4.2 Algorithm 1 - Nested CV

This is the pipeline that we input everything into. Here we have implemented algorithm 1 from 2.2.3, and we use it to put every algorithm into it. It was found that this algorithm has not been released as a general working solution in an open-source repository, so the code has been released into a package, *available here as an open-source package**.

First of is the `__init__` function. This function is a reserved function in Python, and in object-oriented programming terminology, we would call it our constructor. Basically, when you call the NestedCV class, the init function requires you to enter some parameters. `self` is another reserved word, which refers to the instance of the class. In the following lines of code, we just initialize some variables to the NestedCV instance.

```python
class NestedCV():
    def __init__(self, model, params_grid, outer_kfolds, inner_kfolds,
     ↪  cv_options={}):
        self.model = model
        self.params_grid = params_grid
        self.outer_kfolds = outer_kfolds
        self.inner_kfolds = inner_kfolds
        self.metric = cv_options.get('metric', mean_squared_error)
        self.metric_score_indicator_lower = cv_options.get(
            'metric_score_indicator_lower', True)
        self.sqrt_of_score = cv_options.get('sqrt_of_score', False)
        self.randomized_search = cv_options.get('randomized_search', True)
        self.randomized_search_iter = cv_options.get(
            'randomized_search_iter', 10)
        self.recursive_feature_elimination = cv_options.get(
            'recursive_feature_elimination', False)
        self.outer_scores = []
        self.best_params = {}
        self.best_inner_score_list = []
        self.variance = []
```

*https://github.com/casperbh96/Nested-Cross-Validation

The next function is very simple. If the NestedCV instance has a `self.sqrt_of_score=True`, then we return the square root of *scoreValue*, else we just return the scoreValue again. It's a simple way to output the score in the format you want, specifically also for outputting the RMSLE score, that we want.

```
    def _transform_score_format(self, scoreValue):
        if self.sqrt_of_score:
            return np.sqrt(scoreValue)
        return scoreValue
```

This next function will take an array of dictionaries, called `best_inner_params_list`, and turn it into one dictionary, but without any duplicate values in each key.
Here is an example of how it removes repeated values and cumulates the rest of the array of dictionaries into one dictionary:

```
[{'max_depth': [None], 'n_estimators': [1000]},
{'max_depth': [None], 'n_estimators': [600]},
{'max_depth': [None], 'n_estimators': [800]},
{'max_depth': [None], 'n_estimators': [1000]},
{'max_depth': [None], 'n_estimators': [50]}]
```

becomes

```
{'max_depth': [None], 'n_estimators': [1000,600,800,50]}
```

```
    def _score_to_best_params(self, best_inner_params_list):
        params_dict = {}
        for best_inner_params in best_inner_params_list:
            for key, value in best_inner_params.items():
                if key in params_dict:
                    if value not in params_dict[key]:
                        params_dict[key].append(value)
                else:
                    params_dict[key] = [value]
        return params_dict
```

4.2.1 Fit Function

This next function is the one you call after having initialized the class NestedCV. So you could initialize by `search = NestedCV()`, obviously with some parameters for the constructor. Then you could do `search.fit()` afterwards, and also with some parameters. The fit function is split up into many parts and will be explained accordingly, with the explanation above the code piece being explained. You input a X and y to this function, where X is the input features and y is the output feature. The first thing we do here is to print the algorithm being fitted to the data. After that, we initialize the outer and inner K-Fold partitions, with K equal to the class instance of `self.outer_kfolds` and `self.inner_kfolds`. At last, we make arrays to store values in.

```
    def fit(self, X, y):
        print('\n{0} <-- Running this model
         ↪  now'.format(type(self.model).__name__))
        outer_cv = KFold(n_splits=self.outer_kfolds, shuffle=True)
        inner_cv = KFold(n_splits=self.inner_kfolds, shuffle=True)
        model = self.model

        outer_scores = []
        variance = []
        best_inner_params_list = []
        best_inner_score_list = []
```

Next, we split X and y into some number of K-partitions. The KFold function gives us some random training and testing indexes, which is used to split the data repeatedly. If K is 5, then we get 80% of the observations as training index and 20% as a testing index. This is the outer loop running. But there is more code later when we exit the inner loop.

```
        for (i, (train_index, test_index)) in enumerate(outer_cv.split(X, y)):
            print('\n{0}/{1} <-- Current outer fold'.format(i+1,
             ↪  self.outer_kfolds))
            X_train_outer, X_test_outer = X.iloc[train_index],
             ↪  X.iloc[test_index]
            y_train_outer, y_test_outer = y.iloc[train_index],
             ↪  y.iloc[test_index]
            best_inner_params = {}
```

```
49              best_inner_score = None
```

The same thing happens here, as in the last loop. This is simply the inner loop. Notice that we only `split(X_train_outer, y_train_outer)` and preserve the `X_test_outer` and `y_test_outer` for later.

```
50              for (j, (train_index_inner, test_index_inner)) in
                 ↪  enumerate(inner_cv.split(X_train_outer, y_train_outer)):
51                  print('\n\t{0}/{1} <-- Current inner fold'.format(j+1,
                     ↪  self.inner_kfolds))
52                  X_train_inner, X_test_inner = X_train_outer.iloc[
53                      train_index_inner], X_train_outer.iloc[test_index_inner]
54                  y_train_inner, y_test_inner = y_train_outer.iloc[
55                      train_index_inner], y_train_outer.iloc[test_index_inner]
```

We received some dictionary with hyperparameters and arrays of values for them. Here, we specify whether to use a grid search or random search, which is a boolean for the instance of the class. So `if self.randomized_search`, i.e. if the randomized search boolean is *true*, then we execute the code to the left/above the if-statement, which is a random search. If it is not true, we execute the code to the right/below, which is grid search.

The rest, lines 58 to 62, is setting the hyperparameters of the model equal to the generated sample of hyperparameters from the random search or grid search. This is done in a loop until neither of one of the chosen search functions delivers any more grids to try out. For each grid, we fit the current algorithm, i.e. we fit the data to the model. After that, we make predictions.

```
56                  for param_dict in
                     ↪  ParameterSampler(param_distributions=self.params_grid,
57                  n_iter=self.randomized_search_iter) if (self.randomized_search)
                     ↪  else ( ParameterGrid(param_grid=self.params_grid)):
58                      model.set_params(**param_dict)
59                      model.fit(X_train_inner, y_train_inner)
60                      inner_pred = model.predict(X_test_inner)
61                      inner_grid_score = self.metric(y_test_inner, inner_pred)
62                      current_inner_score_value = best_inner_score
```

This part is about finding the hyperparameters which yielded the best test error. All we do here is assign the best test error and hyperparameters for each dictionary delivered by the search algorithm. In the end, lines 76 to 77, the inner loop ends afterward. We append the best hyperparameters and test errors from each inner loop, to an array of best hyperparameters and test errors.

```
63                      if(best_inner_score is not None):
64                          if(self.metric_score_indicator_lower and
                             ↪  best_inner_score > inner_grid_score):
65                              best_inner_score =
                                 ↪  self._transform_score_format(inner_grid_score)
66
67                          elif (not self.metric_score_indicator_lower and
                             ↪  best_inner_score < inner_grid_score):
68                              best_inner_score =
                                 ↪  self._transform_score_format(inner_grid_score)
69                      else:
70                          best_inner_score =
                             ↪  self._transform_score_format(inner_grid_score)
71                          current_inner_score_value = best_inner_score+1
72
73                      if(current_inner_score_value is not None and
                         ↪  current_inner_score_value != best_inner_score):
74                          best_inner_params = param_dict
75
76              best_inner_params_list.append(best_inner_params)
77              best_inner_score_list.append(best_inner_score)
```

Now we are in the outer loop again. This short piece of code will either run feature elimination and/or train and test the model with the testing data from the outer loop. In the last line, we make predictions on the outer testing data.

```
78              if self.recursive_feature_elimination:
79                  pred = self._fit_recursive_feature_elimination(
80                      best_inner_params, X_train_outer, y_train_outer,
                         ↪  X_test_outer)
81              else:
82                  model.set_params(**best_inner_params)
83                  model.fit(X_train_outer, y_train_outer)
84                  pred = model.predict(X_test_outer)
```

In this next part, we use the metric of choice (from the input to the class instance). The metric will score how well our outer loops performed. At last, we print out the best inner score, best inner hyperparameters, and the outer score. Afterward, we just set the variables of the instance of the class, equal to what we figured out for the current model. At line 95, we step out of the outer loop.

```
85              outer_scores.append(self._transform_score_format(
86                  self.metric(y_test_outer, pred)))
87
88              variance.append(np.var(pred, ddof=1))
89
90              print('\nResults for outer fold:\nBest inner parameters was:
                 ↪  {0}'.format(
91                  best_inner_params_list[i]))
92              print('Outer score: {0}'.format(outer_scores[i]))
93              print('Inner score: {0}'.format(best_inner_score_list[i]))
94
95          self.variance = variance
96          self.outer_scores = outer_scores
97          self.best_inner_score_list = best_inner_score_list
98          self.best_params = self._score_to_best_params(best_inner_params_list)
99          self.best_inner_params_list = best_inner_params_list
```

The last function is for plotting variance versus the score. We used it for a little while, but we do not use it anymore. It just makes a simple plot, that one could look at, and you could then see the variance and score fluctuations for each outer loop.

```
    def score_vs_variance_plot(self):
        plt.figure()
        plt.subplot(211)

        variance_plot, = plt.plot(self.variance, color='b')
        score_plot, = plt.plot(self.outer_scores, color='r')

        plt.legend([variance_plot, score_plot],
                   ["Variance", "Score"],
                   bbox_to_anchor=(0, .4, .5, 0))

        plt.title("{0}: Score VS Variance".format(type(self.model).__name__),
                  x=.5, y=1.1, fontsize="15")
```

4.3 Pipeline: Inputting Algorithms and Hyperparameters

The next few code blocks have the purpose of preparing the neural network by a function than defining which parameters that need to be included and which values the parameters can take. Afterwards, the models are used in a loop that finds the optimal parameters by nested cross-validation and grid search. Lastly, the results are plotted and printed in the text.

In the code below is to define a function that will be able to construct a model of a neural network, where the given layers are added to the model and in the end returned to the caller.

```python
def create_neural_network_model(first_neuron=64,
                                activation='relu',
                                optimizer='Adam',
                                dropout_rate=0.1):
    model = Sequential()
    columns = X.shape[1]

    model.add(Dense(64, activation=activation, input_shape=(columns,)))
    model.add(Dense(128, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(64, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(32, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(16, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(8, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, activation='linear'))

    model.compile(
        loss='mean_squared_error',
        optimizer = 'adam',
        metrics=['mean_squared_error']
    )

    return model
```

The next part has the purpose to initialize the variables that define how the coming trials should be run. First, an array of models is produced, where values for different parameters are given for each "type" of models. Lastly, an array for each type of models is instantiated to hold the scores for the given model and the number of trial runs are defined.

```
models_to_run = [KerasRegressor(build_fn=create_neural_network_model,verbose=0),
 ↪  RandomForestRegressor(), xgb.XGBRegressor()]

models_param_grid = [
                    { # 1st param grid, corresponding to KerasRegressor
                            'epochs' :                [50,100,150,200],
                            'batch_size' :            [16,32],
                            'optimizer' :             ['Adam', 'Nadam', 'Adamax'],
                            'dropout_rate' :          [0.1, 0.3, 0.5],
                            'activation' :            ['relu', 'elu'],
                            'first_neuron' :          [64, 128, 256]
                    },
                    { # 2nd param grid, corresponding to RandomForestRegressor
                            'max_depth': [3, None],
                            'n_estimators':
                             ↪  [100,200,300,400,500,600,700,800,900,1000],
                            'max_features' : [50,100,150,200]
                    },
                    { # 3rd param grid, corresponding to XGBRegressor
                            'learning_rate': [0.05],
                            'colsample_bytree': np.linspace(0.3, 0.5),
                            'n_estimators':
                             ↪  [100,200,300,400,500,600,700,800,900,1000],
                            'reg_alpha' : (1,1.2),
                            'reg_lambda' : (1,1.2,1.4)
                    }
                    ]
NUM_TRIALS = 50

RF_scores = []
XGB_scores = []
NN_scores = []
```

The part below is where the computation happens. The interesting part of the code is nested inside two for-loops which take care of running the code on each model the specified amount of times. The code inside the for-loops will than instantiate the nestedCV class and run the two functions “fit” and “best_params”. The fit function runs the nestedCV algorithm and best_params returns the parameters found to produce the best results for the given iteration. Afterwards, the results are passed to GridSearchCV from the sklearn package, where the results from GridSearchCV is formatted and stored in the array matching the model type that produced the result.

```python
for trial in range(NUM_TRIALS):
    for i,model in enumerate(models_to_run):
        nested_CV_search = NestedCV(model=model, params_grid=models_param_grid[i], outer_kfolds=5, inner_kfolds=5,
                          cv_options={'sqrt_of_score':True, 'randomized_search_iter':30})
        nested_CV_search.fit(X=X,y=y)
        model_param_grid = nested_CV_search.best_params
        print('\nCumulated best parameter grids was:\n{0}'.format(model_param_grid))

        gscv = GridSearchCV(estimator=model, param_grid=model_param_grid, scoring='neg_mean_squared_error', cv=5)
        gscv.fit(X,y)

        print('\nFitting with optimal parameters:\n{0}'.format(gscv.best_params_))
        gscv.predict(X_test)
        score = np.sqrt(-gscv.best_score_)

        if(type(model).__name__ == 'KerasRegressor'):
            NN_scores.append(score)
        elif(type(model).__name__ == 'RandomForestRegressor'):
            RF_scores.append(score)
        elif(type(model).__name__ == 'XGBRegressor'):
            XGB_scores.append(score)

        print('\nFinal score for {0} was {1}'.format(type(model).__name__,score))
```

The last block of code handles the representation of the output by producing a graph. The graph plots the scores on the y-axis and the iteration that produced the score on the x-axis.

```
plt.figure()

rf, = plt.plot(RF_scores, color='b')
xgb, = plt.plot(XGB_scores, color='r')
nn, = plt.plot(NN_scores, color='g')

plt.legend([rf, xgb, nn],
           ["Random Forest", "XGBoost", "Neural Networks"],
           bbox_to_anchor=(0, .4, .5, 0))

plt.title('Test scores as RMSLE with hyperparameter optimization',
          x=.5, y=1.1, fontsize="15")
```

5 Results and Analysis

In this chapter, the results from the 3 models will be presented. The results are presented by graphs that show the output from each model in an easily readable way. The 3 models RF, XGBoost and neural network are first presented on each own and afterward presented in a combined graph for easy comparison. Each model has been through the nested cross-validation algorithm 50 times, except for the neural network. To see the raw results look in appendix 8.5

5.1 Result

In the 3 following sub-chapters presenting the results for the 3 models, RF, XGBoost and neural network. Note that the x-axis is given in the iteration in chronological order, and the y-axis is the resulting score as RMSLE. One would wish to estimate the performance of a given model, because when deploying a model, you would want to select the model with the best possible performance, in terms of predicting new data.

5.1.1 Results of RF

The result for RF is summaries in figure 5.1, where it can be seen, that RF scores are lower than linear regression which acts as a baseline. The results are consistent, except a single iteration which produced a result far worse than the average iteration with a score of 0,187357. If this iteration is excluded the lowest score is 0,137077 and the highest score is 0,138527 across the 50 iterations.

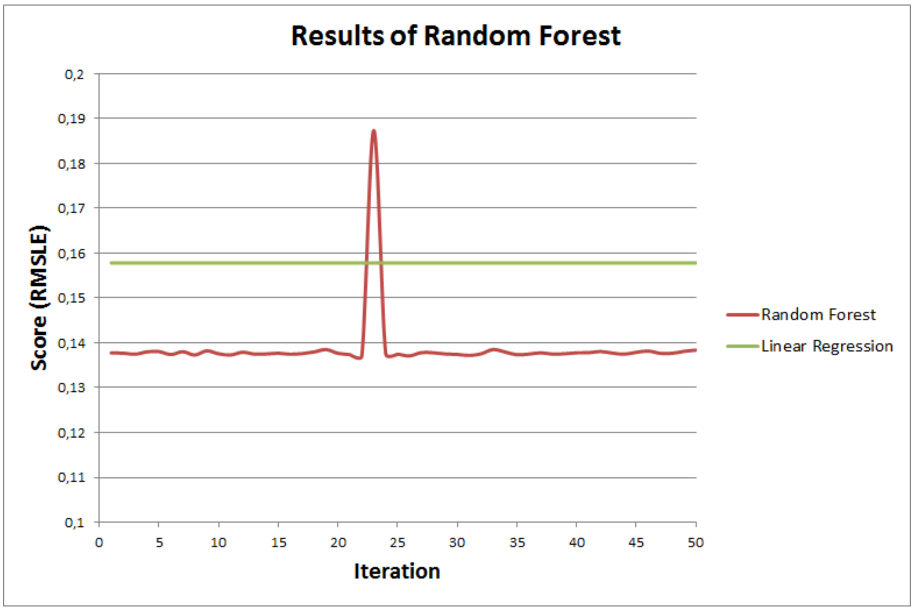

Figure 5.1: The results of the random forest in a graphical representation

The general estimate of the performance of the presented random forest models is 0,138706, but without the outlier it is 0.137714.

5.1.2 Results of XGBoost

The result for XGBoost is summaries in figure 5.2, where it can be seen, that XGBoost scores were lower than linear regression which acts as a baseline. The results are consistent, across all 50 iterations. The lowest produced score is 0,122859 and the highest produced score is 0,122883.

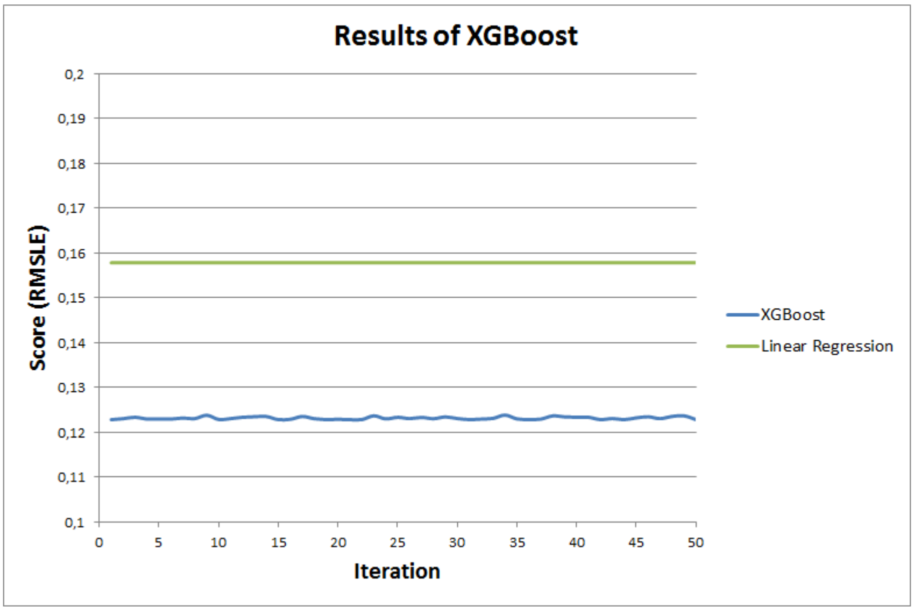

Figure 5.2: The results of the XGBoost in a graphical representation

The general estimate of the performance of the presented XGBoost models is 0,12320576116.

5.1.3 Results of Neural network

The result for the neural network is summaries in figure 5.3, where it can be seen, that the neural network has not completed all 50 iterations. The general score of the neural network is far above the linear regression which acts as a baseline. It shall be noted that the y-axis is more than 2 times the scale compared to the other model's graphs. The lowest score is 0,407714 and the highest score is 0,424597 across the 4 iterations. The results are inconsistent and too few, to really say more about the results of this model.

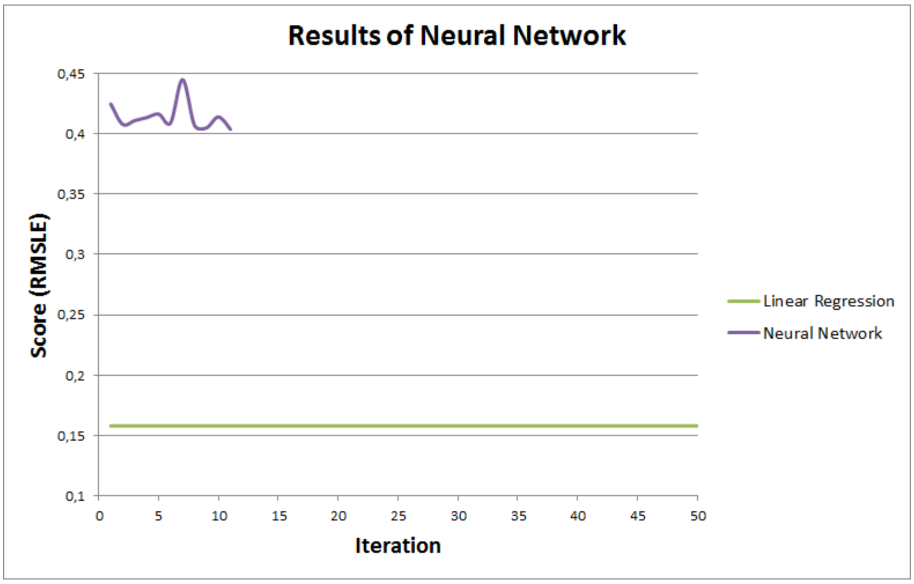

Figure 5.3: The results of the Neural Network in a graphical representation

5.2 Analysis

In this section, the results from each model will be analyzed, except the baseline. We will look at distributions and hyperparameters for the models, and analyze the scores.

5.2.1 Analysis of RF

When taking a look at figure 5.1 what is immediately drawing attention is the spike at iteration 23. This spike will in the first pass be handled as an outlier and excluded from the analysis of RF. After the analysis of RF, a more in-depth look at the outlier at iteration 23 will be taken to explain why it is so different compared to the 49 other iterations. When comparing RF to linear regression it can be seen of figure 5.1 that RF produces a lower score than linear regression. This is expected since RF should be able to take into account more details than linear regression as explained in chapter 3.3.2.

Fitting the results of RF to a normal distribution curve, the figure 5.4 is produced. The figure 5.4 shows the gathered data is distributed in a normal way. This could indicate that the randomness from RF is coming forth to the data, and not obfuscated by some unknown influence.

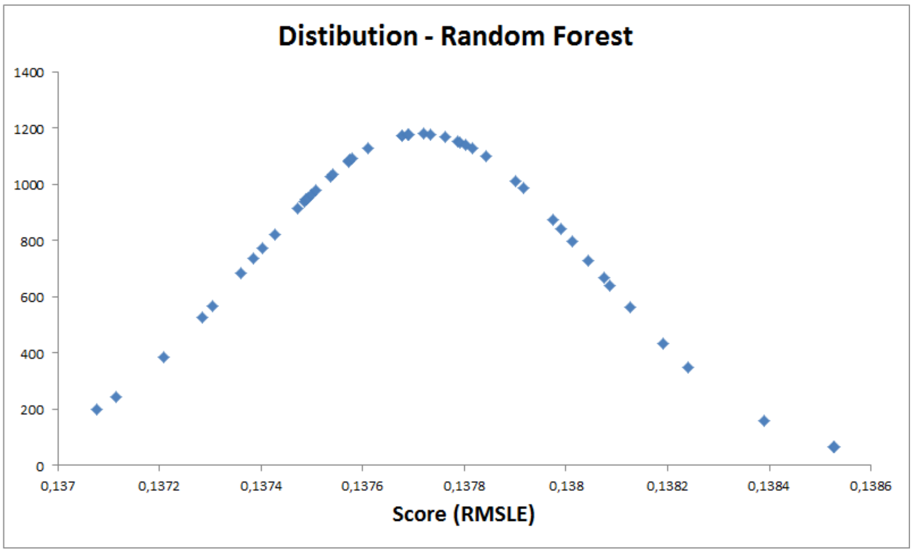

Figure 5.4: The RF results fitted to a normal distribution curve.

Taking a closer look at the logs produced by running RF, shows that some values of the parameters consistently produce a good score, where others consistently do not. Taking a look at the histogram at figure 5.5 shows that the n_estimators ranges from 100 to 1000 but properly could be extended with higher values. This is because there still is a lot of score in the upper range of this hyperparameter. In the lower range it can be seen that as the hyperparameter decrease, fewer iterations produce good scores by using values like 100 and 200. By having nearly non using values around 100, it can be argued that there is no need to extend the range lower.

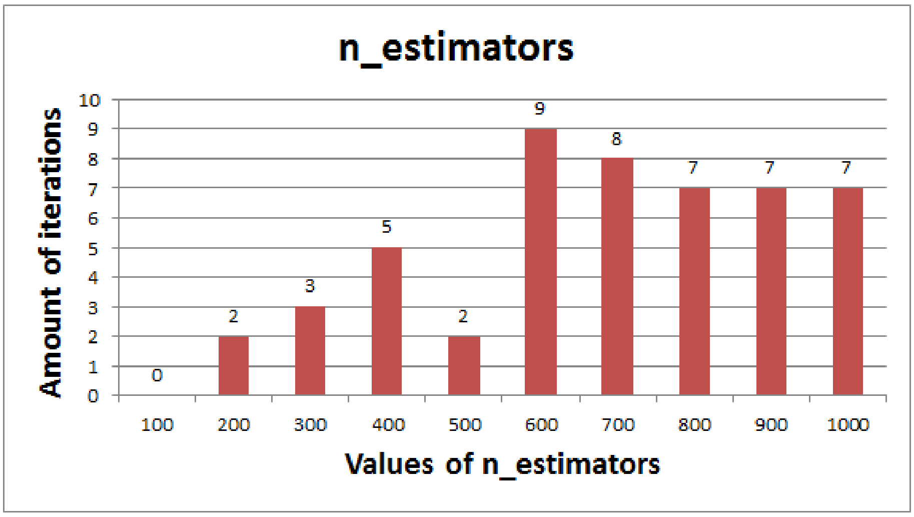

Figure 5.5: The n_estimators (number of trees) hyper parameter presented in a histogram.

While taking a lot at the hyperparameter max_feature, it can be argued that a finer division of possible values could improve the accuracy. This is because most of the iterations are grouped at a value of 100 with a few in the neighboring values of 50 and 150, accordingly to figure 5.6.

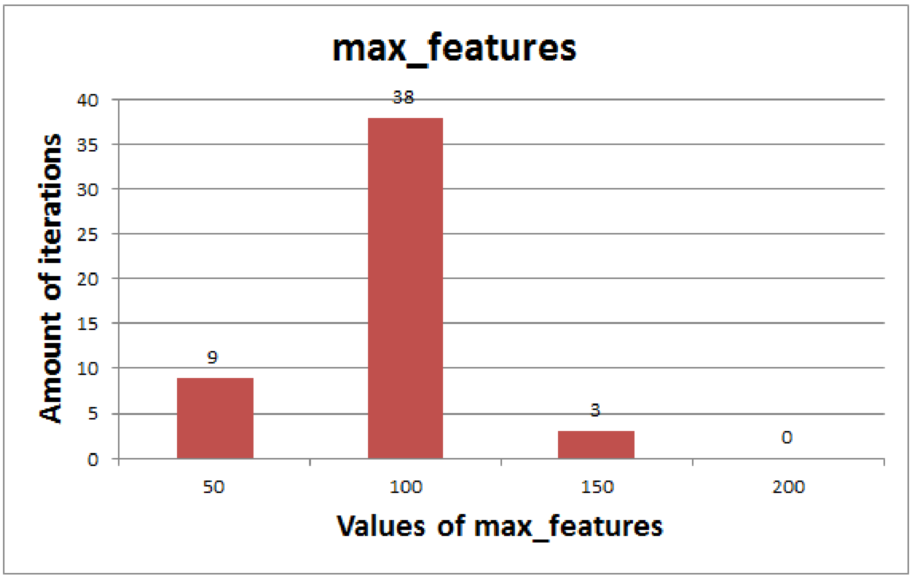

Figure 5.6: The max_features (max number of features when looking for the best node split) hyperparameter presented in a histogram.

Lastly taking a look at the max_depth hyperparameter at figure 5.7, it is clear that the iterations are grouping in the “none” value. This is here the reason why a single outlier is found. When taking a look at the raw data it can be found that the single iteration that produced a score of 0,187356 is the same iteration that has used the hyperparameter max_depth with the value of 3. The raw data can be found in the appendix 8.5.

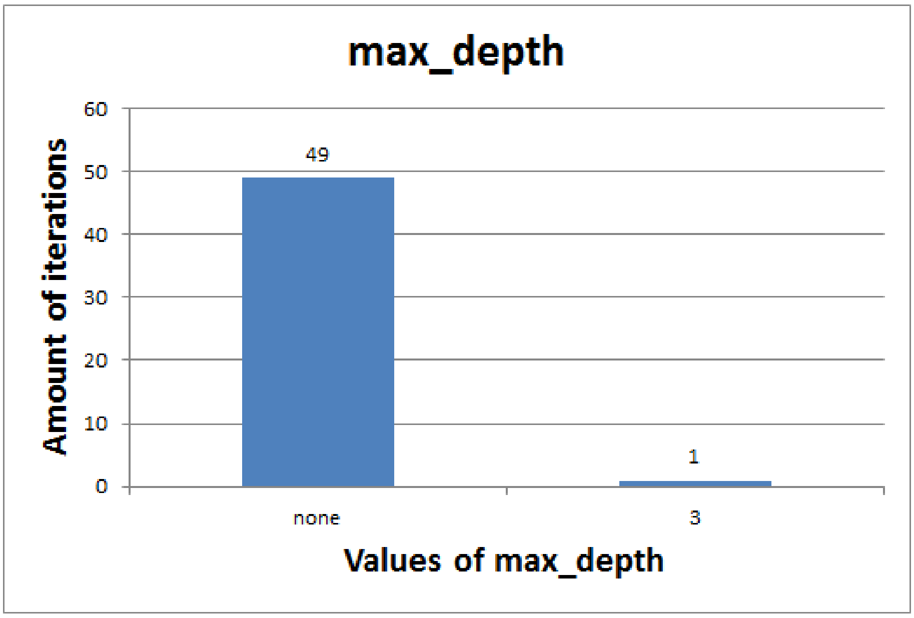

Figure 5.7: The max_depth(maximum depth the tree) hyper parameter presented in a histogram.

5.2.2 Analysis of XGBoost

On first look, the results presented in figure 5.2 looks stable and consistent. Compared to Linear Regression, XGBoost produces a much lower score than Linear Regression, which is expected.

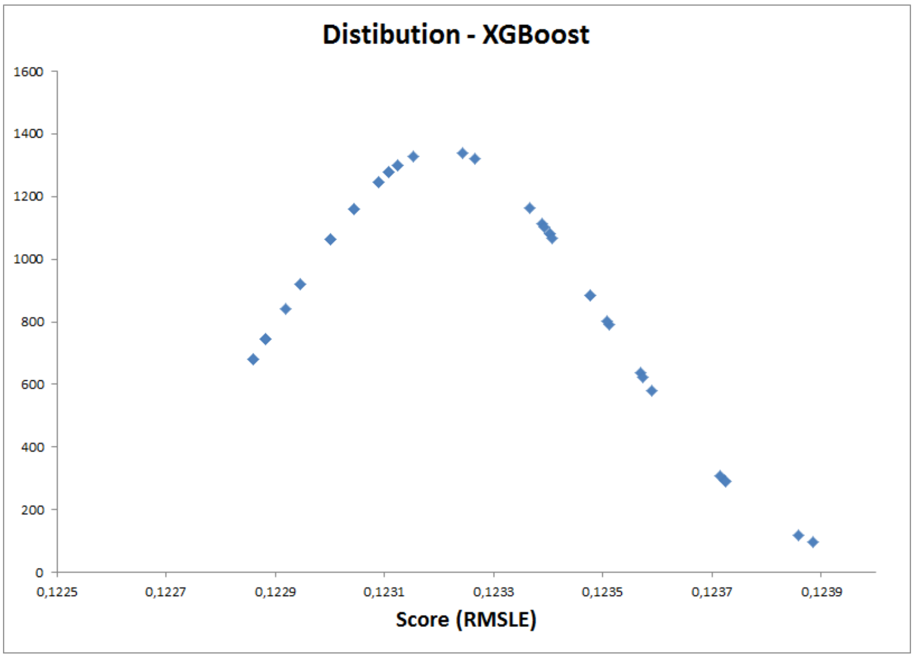

Figure 5.8: The XGBoost results fitted to a normal distribution curve.

When the results from XGBoost is fitted to a normal distribution curve, it can be seen in figure 5.8, some of the left sides of the curve is missing. There can be a few reasons for this; it could be that there is some lower boundary which makes it harder to obtain a score below 0,122859. It could the number of iterations, where more iterations could fill out the left part.

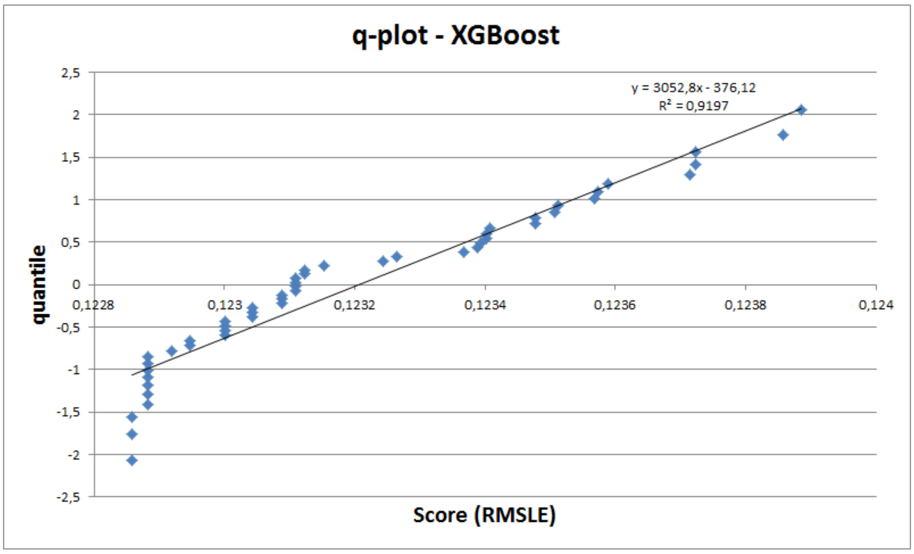

Figure 5.9: The XGBoost results presented in a q-plot.

When making a q-plot of the XGBoost data, see figure 5.9. It can be seen that the different iterations are spread nicely across the full range, except at the far left side. At score 0,122859 and 0,122883, which are the two best scores, about 10 different iterations are clumping up. When comparing to figure 5.8, it could be argued that when getting to a score around 0,122859 it will become harder to lower it. This statement is based on the missing left side of the curve on figure 5.8, and the clumping of iterations to the left on figure 5.9.

5.2.3 Analysis of Neural network

As said in chapter 5.3, the neural network has not completed the 50 iterations and only 11. By having so few data points not much can be said. Trying to extract some information of the results would not be correct since the foundation is not good enough and the information would be incorrect. The best possible use of the results of the neural network would be for improving the model and to troubleshoot why it is scoring as it is doing.

6 Discussion

6.1 Reflection Upon Results

In the coming chapter, you will read about reflections and critique on the different relevant topics. Among those are data analysis, neural network, overfitting and underfitting, and imputation.

6.1.1 Hyperparameters

Comparing the Linear Regression, Random Forest and XGBoost results, some differences can be noticed. Linear Regression and XGBoost are more stable than Random Forest because Random Forest has a spike in score in one of the 50 trials. This spike can be related to the hyperparameter range as seen in the data analysis. Fair criticism is that more hyperparameters and values should have been included, which could lead to a better model. After all, we are looking for the best hyperparameters and values, so we should consider a large enough spectrum so that the returned hyperparameters and values are the optimal ones.

The results of the pipeline could be improved by adding a wider range and more values of hyperparameters to nested CV for all models. In Random Forest, there are only two hyperparameters, with relatively few values, meaning that it is easy to add more hyperparameters and values. In Linear Regression there is no way to improve the model since there are no hyperparameters to optimize. In XGBoost, the results were better than the other two models but there is room for improvements, and what can be done to optimize it, is by trying a wider range and more values of hyperparameters for the model.

6.1.2 Pipeline trials

We have run XGBoost, Random Forest and Linear Regression a total of 50 independent cycles through the pipeline. We chose 50, because it's enough iterations to monitor and observe the different performances of the model, yet not too computationally expensive. If we were to run the pipeline more times or provide more hyperparameters, we would have increased the computing cost, but in total, it will most likely be enough with 50 trials to give a generalized estimate of how well our model performs.

Neural Networks was a special case though because it took anywhere between 6-7 hours to run a full iteration while using the CPU and GPU on a powerful computer. As also seen in the results, we only ran neural networks a total of 11 independent cycles, since it proved to give us worse scores than our baseline, and was 4 to 5 times as computationally expensive compared to the other models.

6.1.3 Time vs Hyperparameters

Running such a pipeline as seen in chapter 4 is quite computationally expensive. While BigO is outside the scope of our project, we can at least argue about the number of hyperparameters and values for the models. As the values and numbers of possible hyperparameters increase the corresponding grid will expand accordingly in size, which hugely increases the possible combinations. Increasing the combinations will result in an increase in running a random search to test the increased combinations. In reverse, fewer hyperparameters and values will equal a shorter runtime. Hyperparameters could be denoted as hp, and having values for each hyperparameter from $= hp_1^{values}$ to $= hp_n^{values}$. To give the KFolds a different name, the KFolds would be called the number of outer folds O_{kf} and inner folds I_{kf}. The number of models that were running can be calculated by this equation:

$$models_n = O_{kf} \times (I_{kf} \times (hp_1^{values} \times, ..., \times hp_n^{values}) + 1) \tag{6.1}$$

When a random search is done, the best set of parameters from the inner loop is run through one extra time and that is where the (+1) comes in. Suppose we had *max_depth* with 2 values, *n_estimators* with 10 values, and max_features with 4 values. Then plugging into the equation we would get

$$models_n = 5 \times (5 \times (2 \times 10 \times 4) + 1) = 2005\, models \tag{6.2}$$

Then, as seen in the results, nested CV was running 50 times, which results in 100250 models trained. That is many models, and in runtime, it took days to finish training all those models. Keep in mind, this is with a relatively small grid of hyperparameters. If more hyperparameters and values were chosen, the number of models would only grow. The runtime of such a script will take even longer. With neural networks, it was calculated that 108050 models will need to be trained, though only 11 iterations where completed, since there were many hyperparameters. Another important runtime problem would be if the data set was really large, like with millions of samples.

6.2 Neural Networks

We started later on a side project, neural networks because it belongs to deep learning which is a promising field in machine learning. Since Neural Networks was a side project, not as much time was possible to allocate to it compared to the other models. The neural network did not get a fair amount of time to get optimized to a level where it could compete at equal terms with the other models.

6.2.1 Model setup

The is simple to build a neural network model, but there are many options to choose from, and each option small or big will affect the performance of the neural network. There have been used two types of layers in the current model, "Dense" and "Dropout". "Dropout" will be discussed later, and "Dense" is part of building the neural networks.

The reason for using this structure of layers for the neural network is that we have been inspired by different implementations methods of how to build a neural network. The current structure has reached a point where it is more beneficial than the first implementation of the neural network since the newer version was better performing.

Considering the structure of the model a more simple structure would have been easier to optimize, using a model with a lower number of layers or nodes would have reduced the complexity of the model.

6.2.2 Neural network — Regularization

Dropout is used for regularization. By experience with trial an error of adjusting of the number of Dropout layers in the model, it has been noticed that it fewer layers will increase the RMSLE score, which will probably be different in other neural network models.

On the other hand, the lowest of the 3 possible dropout rates was chosen because it gave a better performance in the nested CV. Note that the dropout rate is the same for each dropout layers for the current implementation of the neural network.

As stated increasing the number of dropout layers improved the performance of the neural network, and since the lower dropout rate where preferred. It could indicate that the optimal dropout rate would be between 0.0 and 0.1.

Another way to do regularization for the neural network is to use early-stopping which is when the score stops decreasing for a given amount of training iterations (epochs), then training stops. This is another way to do regularization since it will stop overfitting models.

We have tried to use it in an early stage but we didn't manage to make it work with nested-cv and because of this, it is not included in our results. Trying to do it will most likely give different results, but we can not tell exactly how big the change is or the direction that it will lead to.

6.2.3 Cross Validation

Running a neural network takes a huge amount of computational calculations, which will lead to a lot of time spent and also will need a powerful computer to be able to run these calculations quickly. In addition to that, running nested-cv with a neural network would further increase the amount of calculation that it needs to complete. A faster method than nested-cv could decrease the number of calculations needed like leave-one-out cross-validation which should need fewer calculations. Another option is not to use cross-validation at all.

6.3 Overfitting and Underfitting

To investigate and discuss overfitting and underfitting for the pipeline, we would like to start by evaluating nested cross-validation, since it gives us an unbiased estimate of the test error for a model.
After that, regularization is important, since it helps reduce overfitting — and arguably, it's much easier to make complex models, than it is making a reasonable model. Some of the last relevant points are feature selection and how other participants in the Kaggle competition dealt with overfitting and underfitting.

6.3.1 Evaluating Nested Cross-Validation

As talked about in section 2.2.1, nested CV is the way in which one should choose to evaluate the performance of models. Though, what did the pipeline evaluated in this report gain or lose from using this procedure?

6.3.1.1 Bias and Variance dependence

Usually, when one thinks of bias and variance, there is an indication that a high variance means low bias, and high bias means low variance. While this can be true, one of the authors of [Cawley and Talbot, 2010] argues, rather than the suggested indication, that bias and variance depend on the model selection criterion [Cawley, 2015]. This means that, if a model has variance by some selection criterion, then it likely also has a high bias, if evaluated by normal CV.
By using nested CV, we would be more certain of achieving a lower bias, lower variance, and therefore a lower scoring. The dataset for this report consists of 1460 observations, meaning that it is a relatively small dataset. This means that we should be careful with model selection and the resulting bias. While looking at these presented facts, the right way to approach the presented pipeline with nested CV *is* the right way to go about estimating a model's performance. Though we cannot just say that nested CV is enough because it does not guarantee to control the complexity of a model.

6.3.2 Regularization

Most commonly used to solve the problem of overfitting is regularization, as presented in section 3.4. Regularization is just penalizing the more complex models, than the simple models, to avoid overly optimistic performance of a model. The question is, did we utilize regularization to a good enough extent, where we actually got something from it?

Part of the reason why XGBoost is so successful at predicting new unseen data is because of the regularization term. Thus a good question to ask, to either criticize or justify how we implemented regularization is: *did we use regularization to somewhat of its best capability, to limit overfitting?*

6.3.2.1 XGBoost Regularization

A lot of knowledge is needed to understand the hyperparameters of XGBoost. In this report, a generic approach has been chosen to deal with the hyperparameters. It could be argued that the hyperparameters grid could be expanded in search of better combinations of hyperparameters. particular the range of hyperparameter gamma could be investigated. Gamma is in charge of "complexity control"[Leoni, 2019]. In the current state, gamma is set as default to 0. A range from 0.1 to 2 with a number of steps could be tested. [Leoni, 2019]

I general it could be interesting to change the hyperparameters grid. If there was included a large range to choose from for each hyperparameter it might change something in the results. Since this would allow for more combinations. So far the number of trees (n_estimators), has much longer array than the rest of the hyperparameters. That would be interesting to investigate further.

It is really hard to know beforehand which of the parameters should have a large span like "n_estimators" has. It might prove productive to give the other parameters an equal array size.

The hyperparameters were found through a random grid search. This method was chosen because it runs faster. But to be sure that all the combinations are tried one would need to do a grid search.

6.3.3 Feature Selection

Unfortunately, we did not get to include an important part of machine learning in our pipeline. To include this in our pipeline, we would have had to wrap the for-loop responsible for creating each parameter grid in our nested CV function. The wrapping would then perform a normal CV for each *j*'th inner split in algorithm 1 (2.2.3). In this feature selection procedure, we would fit each model again, as we do in the innermost for-loop in algorithm 1, resulting in a very computationally expensive setup — even for a smaller dataset. Considering we had about 250 features after encoding, we would have to calculate the importance of each feature by fitting to a model, which is computationally expensive, and then remove the least important feature, reducing the dataset to 249 features. And so it would continue on until one feature is left and then pick the best number of features.

6.4 Imputation

There are many ideas and opinions on imputation. It seems to some extent like art on its own. It has been chosen to limit the feature imputation to a generic approach which would be address. First, the method has been used and can roughly be set up with the following steps 4

1. Concatenate train and test data.

2. Drop features. Here the 5 features are dropped that have more than 50% missing values. 'FireplaceQu' has 47% NaN's The rest has more than 80%. NAN's (also SalePrice and Id are dropped since Id is irrelevant and Sales prices doesn't occur the Test set) 3 One hot encoding.

4. Impute values using Iterative imputer over the concatenated dataset.

5. Split the data set into train and test again.

6.4.1 How Could the Imputation be Improved

There are overwhelming many aspects to imputation and seen in retrospect there are a lot of different ways it could be carried out. Before thinking of that its reasonable to ask to the method that has been used so far. Is it, for instance, correct to concatenate the data beforehand? One could think that the test data and the training should be kept apart as much as possible. Therefore it could be argued that, the more one interferes with the test data the more bias one introduces.

Since it has not been made a big deal of understanding how "ItertiveImputer()" from fancyimpute works it could be thought of as a black box. Fancyimpute will "Model each feature with missing values as a function of other features, and use that estimate for imputation." - [fancyimpute, 2019]

Knowing that the NaN is calculated "as a function of other features" doesn't really give the full view. The test and the training data could be intertwined after this process. This is not keeping it objective. Therefore it would be more reasonable to keep the data sets separate as much as possible. This is relatively easy to change in the pipeline.

On stack exchange, it is commented that there exists a way to first train an "imputer model" on the training data. This "imputer model" can then be used on the test data. In that sense, the test data gets imputed in a similar fashion. The test and the train data are then separated.

-"The imputer will apply the same imputations that it learned from the training set." [Scratch'N'Purr, 2018]

This technique might be worth a try.

7 Conclusion

A generic pipeline was made, where nested cross-validation was implemented within. The different models from the research question were interchangeably used as input to the pipeline to produce an RMSLE score. Nested CV was used to compare the different models with different scores, based on the optimal hyperparameters produced in the iterative inner loop in the nested CV. Given the found optimal hyperparameters for each model found by nested CV, it was found that XGBoost produced the lowest RMSLE score, in terms of predicting house prices on new, unseen data. The generalized RMSLE scores for XGBoost was 0.123, Random Forest 0.139 and Neural Networks 0.414.

Hyperparameter optimization is clearly one of the most important things to consider when wanting a better score. In this report, it was found that it is worth providing a wide range of hyperparameters for a nested CV procedure. In random forest, an outlier score was produced due to a misplaced range and/or the number of values for the hyperparameters. In both random forest and XGBoost, it was found that the values for the hyperparameter n_estimators were appropriately wide — it consistently chose a wide range of values, meaning that the values provided was a good range. Although it was a good range, it was found that it also is important to experiment with how wide the range should be.

Overfitting, underfitting, and regularization are an important area in ML. Assuming nested CV finds the optimal hyperparameters from a set of a range of hyperparameters, it was found through investigation, that regularization hyperparameters are crucial to reducing overfitting during hyperparameter selection in nested CV.

Imputation is also an important area in ML, which was learned during the project progression. The generic approach to imputation was used in this project, but whether it was the right approach is unanswered.

8 Perspective

8.1 Nested CV — Further Experiments

Nested CV definitely needs more experimenting, and it needs to be improved to the point, where it has many more options for the function. One big question is, what do you do after nested CV? What was done in this report was to use the best hyperparameters from each outer fold in a grid search, but there is no indication of what is right and wrong at this moment. Though one paper suggests that nested CV is pedantic. The writers suggest that for classification, it is *probably* not needed when there are relatively few hyperparameters. Though we used regression and not classification, they even suggest that it applies to regression algorithms, like random forest and gradient boosting [Wainer and Cawley, 2018]. Thus, it would be an interesting experiment to evaluate many regression techniques to find out if nested CV *is* is needed. Though in relation to this, they also use the words *probably* and *strong evidence*, but they are not totally sure. Therefore in extension, a concrete experiment to find *when* nested CV is not pedantic, and when it is clear that you should use it.

8.2 Model Stacking

One of the most prominent ways to do model stacking and getting the best possible score, especially in Kaggle competitions is something called model stacking. The idea behind model stacking is making a pipeline, where you hold back a testing dataset, and on the training dataset, you choose some models to train. For each of the models, you would take the prediction and add it as a new feature, so that you would get as many new features as you choose to run models. At last, you would have a final model that you would train and

test, preferably the one with the best predictive power (the one able to score the lowest). The new training dataset with the added features and the testing dataset that was not used is used in the final model.
In particular, it was found that an open-source package called *pystacknet*, and also *an example* for another way of doing it. This proposed way of doing machine learning supposedly does give a better score, which is exactly what one wants.

8.3 Imputation Suggestions

It would be worth to look into other generic ways to impute the dataset. There are 4 popular strategies that should be considered.[Badr, 2019] Here the pros and cons are considered as well in the scope of this project.
1. Do nothing. Leave the data as it is, Pro: XGBoost should be able to do this. Con: However this would not work for the other models.
2. Imputation Using K nearest neighbors(KNN) can be much more accurate "frequent imputation" [Badr, 2019] methods.
"The assumption behind using KNN for missing values is that a point value can be approximated by the values of the points that are closest to it, based on other variables." [Obadia, 2017]
pros: Depending on the data set can KNN be much more accurate than "most frequent" imputation methods. [Badr, 2019] Cons: Quite sensitive to outliers and "Computationally expensive" methods[Badr, 2019]
3. Imputation Using Multivariate Imputation by Chained Equation (MICE)
"Works by filling the missing data multiple times"[Badr, 2019] and is "much better than a single imputation ". The method assumes the data is missing at random(MAR). [Akinfaderin, 2017] Can handle different variables of different data types such as binaries and continues also skip-patterns. [Badr, 2019]
A skip pattern reflects something about the way people answers in questionnaires eg."If you marked 'a', skip to question 4; if you marked 'b', continue to the next question."
How MICE really works it quite advanced. It involves the Monte Carlo method which is out of the scope of this report. MICE can be found within the fancyimpute library:

"fancyimpute.MICE().complete(data matrix)"
Seems like a good choice for the pipeline.
4. Imputation Using Neural network (Datawig) Neural network for imputation is outside of the scope of this project. However, the method can be applied from a library called Datawig. pros: -Quite accurate compared to other methods.[Badr, 2019] -Can handle categorical data. Cons: -Can be quite slow with large datasets.
Each of these methods should be tested since there is not a fit all method.

8.4 Custom Data Handling

It seems like a generic approach to data treatment might not win Kaggle competitions. The founder of Kaggle was interviewed about how the different competitions were won. His reply was clear, that "..If you have lots of structured data, the handcrafted approach is your best bet.." [Fogg, 2016].
Given that information, it should be considered to take a non-generic approach to data handling. Since the dataset, in this case, is structured.

8.5 Deployment

When a house pricing machine learning model has been trained and tested is should be "shipped". It would be desired to use it in the real estate industry. This process is called Deployment. Deployment is one of the crucial parts of ML development. Using the model is referred to as "loading". the model can be loaded This is because it allows saving and loading the model and take advantage of it so it can be integrated with a different set of technologies and use it in real-life applications. It can be integrated into a web or mobile application, where it can be loaded and used to predict the house prices date without taking all the time to evaluate the model itself. In addition to that, it can be loaded again to python and reused in a different context as a model so the state will be saved.

References

[Akinfaderin, 2017] Akinfaderin, W. (2017). Missing data conundrum: Exploration and imputation techniques. https://medium.com/ibm-data-science-experience/missing-data-conundrum-exploration-and-imputation-techniques-9f40abe0fd87.

[andrew NG, 2019] andrew NG (2019). Ml:linear regression with multiple variables. https://www.coursera.org/learn/machine-learning/resources/QQx8l.

[Badr, 2019] Badr, W. (2019). 6 different ways to compensate for missing values in a dataset (data imputation with examples). https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779.

[Cawley, 2015] Cawley, D. G. (2015). When is nested cross-validation really needed and can make a practical difference? https://stats.stackexchange.com/a/178347/246891.

[Cawley and Talbot, 2010] Cawley, G. C. and Talbot, N. L. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. http://jmlr.csail.mit.edu/papers/volume11/cawley10a/cawley10a.pdf.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. https://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf.

[Drakos, 2018] Drakos, G. (2018). How to select the right evaluation metric for machine learning models: Part 2 regression metrics. https://towardsdatascience.com/

how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-2-regression-metrics-d4a1a9ba3d74.

[D'Souza, 2018] D'Souza, J. (2018). A quick guide to boosting in ml. https://medium.com/greyatom/a-quick-guide-to-boosting-in-ml-acf7c1585cb5.

[fancyimpute, 2019] fancyimpute (2019). fancyimpute. https://pypi.org/project/fancyimpute/.

[Fogg, 2016] Fogg, A. (2016). Anthony goldbloom gives you the secret to winning kaggle competitions. https://www.import.io/post/how-to-win-a-kaggle-competition/.

[Friedman, 1999] Friedman, J. H. (February 1999). Greedy function approximation: A gradient boosting machine. https://statweb.stanford.edu/~jhf/ftp/trebst.pdf.

[Hunt and Kubler, 1984] Hunt, B. and Kubler, O. (1984). Karhunen-loeve multispectral image restoration, part i: Theory. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1164363.

[James et al., 2013] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). An introduction to statistical learning. https://www-bcf.usc.edu/~gareth/ISL/ISLR%20First%20Printing.pdf.

[Jimmy Ba, 2013] Jimmy Ba, B. F. (2013). Adaptive dropout for training deep neural networks. http://papers.nips.cc/paper/5032-adaptive-dropout-for-training-deep-neural-networks.pdf.

[Krstajic et al., 2014] Krstajic, D., Buturovic, L. J., Leahy, D. E., and Thomas, S. (2014). Cross-validation pitfalls when selecting and assessing regression and classification models. https://jcheminf.biomedcentral.com/track/pdf/10.1186/1758-2946-6-10.

[Leoni, 2019] Leoni, F. (2019). From zero to hero in xgboost tuning. https://towardsdatascience.com/from-zero-to-hero-in-xgboost-tuning-e48b59bfaf58.

[McDonald, 2017] McDonald, C. (2017). Machine learning fundamentals (i): Cost functions and gradient descent. https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220.

[Miller, 2018] Miller, L. (2018). Machine learning week 1: Cost function, gradient descent and univariate linear regression. https://medium.com/@lachlanmiller_52885/machine-learning-week-1-cost-function-gradient-descent-and-univariate-linear-regression-8f5fe69815fd.

[Mortazavi, 2018] Mortazavi, M. (2018). Why do we need to discard one dummy variable? https://datascience.stackexchange.com/questions/27957/why-do-we-need-to-discard-one-dummy-variable/27993.

[Obadia, 2017] Obadia, Y. (2017). The use of knn for missing values. https://towardsdatascience.com/the-use-of-knn-for-missing-values-cf33d935c637.

[Patel, 2018] Patel, A. (2018). Chapter-3 bias and variance trade-off in machine learning. https://medium.com/ml-research-lab/chapter-3-bias-and-variance-trade-off-in-machine-learning-a449fa1e2729.

[Pedro, 2000] Pedro, D. (2000). A unified bias-variance decomposition and its applications. https://homes.cs.washington.edu/~pedrod/papers/mlc00a.pdf.

[Perktold et al., 2018] Perktold, J., Seabold, S., and Taylor, J. (2018). Patsy: Contrast coding systems for categorical variables. http://www.statsmodels.org/dev/contrasts.html.

[Samuel, 1959] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.368.2254&rep=rep1&type=pdf.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. https://linkinghub.elsevier.com/retrieve/pii/S0893608014002135.

[Scikit-Learn, 2018] Scikit-Learn (2018). Nested versus non-nested cross-validation. https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html.

[Scratch'N'Purr, 2018] Scratch'N'Purr (2018). Imputation on the test set with fancyimpute. https://stackoverflow.com/questions/53322182/imputation-on-the-test-set-with-fancyimpute.

[Singh, 2018] Singh, V. (2018). What is bias and variance? https://www.quora.com/What-is-bias-and-variance.

[Tue Herlau and Mørup, 2019] Tue Herlau, M. N. S. and Mørup, M. (2019). Introduction to machine learning and data mining.

[Varma and Simon, 2006] Varma, S. and Simon, R. (2006). Bias in error estimation when using cross-validation for model selection. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1397873/pdf/1471-2105-7-91.pdf.

[Varoquaux et al., 2017] Varoquaux, G., Raamana, P. R., Engemann, D. A., Hoyos-Idrobo, A., Schwartz, Y., and Thirion, B. (2017). Assessing and tuning brain decoders: cross-validation, caveats, and guidelines. https://arxiv.org/pdf/1606.05201.pdf.

[Wainer and Cawley, 2018] Wainer, J. and Cawley, G. (2018). Nested cross-validation when selecting classifiers is overzealous for most practical applications. https://arxiv.org/pdf/1809.09446.pdf.

Appendix

Code copy-paste

File: nested_cv

```
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import KFold, ParameterGrid, ParameterSampler
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import RFE, RFECV

class NestedCV():
    '''A general class to handle nested cross-validation for any estimator that
    implements the scikit-learn estimator interface.
    Parameters
    ----------
    model : estimator
        The estimator implements scikit-learn estimator interface.
    params_grid : dict
        The dict contains hyperparameters for model.
    outer_kfolds : int
        Number of outer K-partitions in KFold
    inner_kfolds : int
        Number of inner K-partitions in KFold
    cv_options: dict, default = {}
        Nested Cross-Validation Options, check docs for details.
        metric : callable from sklearn.metrics, default = mean_squared_error
            A scoring metric used to score each model
        metric_score_indicator_lower : boolean, default = True
            Choose whether lower score is better for the metric calculation or higher score is better,
            `True` means lower score is better.
        sqrt_of_score : boolean, default = False
            Whether or not the square root should be taken of score
```

```python
        randomized_search : boolean, default = True
            Whether to use gridsearch or randomizedsearch from sklearn
        randomized_search_iter : int, default = 10
            Number of iterations for randomized search
        recursive_feature_elimination : boolean, default = False
            Whether to do feature elimination
    '''

    def __init__(self, model, params_grid, outer_kfolds, inner_kfolds, cv_options={}):
        self.model = model
        self.params_grid = params_grid
        self.outer_kfolds = outer_kfolds
        self.inner_kfolds = inner_kfolds
        self.metric = cv_options.get('metric', mean_squared_error)
        self.metric_score_indicator_lower = cv_options.get(
            'metric_score_indicator_lower', True)
        self.sqrt_of_score = cv_options.get('sqrt_of_score', False)
        self.randomized_search = cv_options.get('randomized_search', True)
        self.randomized_search_iter = cv_options.get(
            'randomized_search_iter', 10)
        self.recursive_feature_elimination = cv_options.get(
            'recursive_feature_elimination', False)
        self.outer_scores = []
        self.best_params = {}
        self.best_inner_score_list = []
        self.variance = []

    # to check if use sqrt_of_score and handle the different cases
    def _transform_score_format(self, scoreValue):
        if self.sqrt_of_score:
            return np.sqrt(scoreValue)
        return scoreValue

    # to convert array of dict to dict with array values, so it can be used as params for parameter tuning
    def _score_to_best_params(self, best_inner_params_list):
        params_dict = {}
        for best_inner_params in best_inner_params_list:
            for key, value in best_inner_params.items():
                if key in params_dict:
                    if value not in params_dict[key]:
                        params_dict[key].append(value)
                else:
                    params_dict[key] = [value]
        return params_dict

    # a method to handle  recursive feature elimination
    def _fit_recursive_feature_elimination(self, best_inner_params, X_train_outer, y_train_outer, X_test_outer):
        print('\nRunning recursive feature elimination for outer loop... (SLOW)')
        # K-fold (inner_kfolds) recursive feature elimination
        rfe = RFECV(estimator=self.model, min_features_to_select=20,
                    scoring='neg_mean_squared_error', cv=self.inner_kfolds, n_jobs=-1)
        rfe.fit(X_train_outer, y_train_outer)

        # Assign selected features to data
        print('Best number of features was: {0}'.format(rfe.n_features_))
        X_train_outer_rfe = rfe.transform(X_train_outer)
        X_test_outer_rfe = rfe.transform(X_test_outer)

        # Train model with best inner parameters on the outer split
        self.model.set_params(**best_inner_params)
        self.model.fit(X_train_outer_rfe, y_train_outer)
        return self.model.predict(X_test_outer_rfe)

    def fit(self, X, y):
        '''A method to fit nested cross-validation
        Parameters
        ----------
        X : pandas dataframe (rows, columns)
            Training dataframe, where rows is total number of observations and columns is total number of features

        y : pandas dataframe
            Output dataframe, also called output variable. y is what you want to predict.

        Returns
        -------
        It will not return directly the values, but it's accessable from the class object it self.
        You should be able to access:

        variance
            Model variance by numpy.var()

        outer_scores
            Outer score List.

        best_inner_score_list
            Best inner scores for each outer loop

        best_params
            All best params from each inner loop cumulated in a dict

        best_inner_params_list
            Best inner params for each outer loop as an array of dictionaries
        '''

        print('\n{0} <-- Running this model now'.format(type(self.model).__name__))
        outer_cv = KFold(n_splits=self.outer_kfolds, shuffle=True)
        inner_cv = KFold(n_splits=self.inner_kfolds, shuffle=True)
        model = self.model

        outer_scores = []
        variance = []
        best_inner_params_list = []  # Change both to by one thing out of key-value pair
        best_inner_score_list = []

        # Split X and y into K-partitions to Outer CV
        for (i, (train_index, test_index)) in enumerate(outer_cv.split(X, y)):
            print('\n{0}/{1} <-- Current outer fold'.format(i+1, self.outer_kfolds))
            X_train_outer, X_test_outer = X.iloc[train_index], X.iloc[test_index]
            y_train_outer, y_test_outer = y.iloc[train_index], y.iloc[test_index]
            best_inner_params = {}
            best_inner_score = None

            # Split X_train_outer and y_train_outer into K-partitions to be inner CV
            for (j, (train_index_inner, test_index_inner)) in enumerate(inner_cv.split(X_train_outer, y_train_outer)):
                print('\n\t{0}/{1} <-- Current inner fold'.format(j+1, self.inner_kfolds))
                X_train_inner, X_test_inner = X_train_outer.iloc[
                    train_index_inner], X_train_outer.iloc[test_index_inner]
                y_train_inner, y_test_inner = y_train_outer.iloc[
                    train_index_inner], y_train_outer.iloc[test_index_inner]

                # Run either RandomizedSearch or GridSearch for input parameters
                for param_dict in ParameterSampler(param_distributions=self.params_grid,
                n_iter=self.randomized_search_iter) if (self.randomized_search) else (
                ParameterGrid(param_grid=self.params_grid)):
                    # Set parameters, train model on inner split, predict results.
                    model.set_params(**param_dict)
                    model.fit(X_train_inner, y_train_inner)
                    inner_pred = model.predict(X_test_inner)
                    inner_grid_score = self.metric(y_test_inner, inner_pred)
                    current_inner_score_value = best_inner_score

                    # Find best score and corresponding best grid
                    if(best_inner_score is not None):
                        if(self.metric_score_indicator_lower and best_inner_score > inner_grid_score):
                            best_inner_score = self._transform_score_format(inner_grid_score)

                        elif (not self.metric_score_indicator_lower and best_inner_score < inner_grid_score):
                            best_inner_score = self._transform_score_format(inner_grid_score)
                    else:
                        best_inner_score = self._transform_score_format(inner_grid_score)
                        current_inner_score_value = best_inner_score+1  # first time random thing

                    # Update best_inner_grid once rather than calling it under each if statement
                    if(current_inner_score_value is not None and current_inner_score_value != best_inner_score):
                        best_inner_params = param_dict

            best_inner_params_list.append(best_inner_params)
            best_inner_score_list.append(best_inner_score)

            if self.recursive_feature_elimination:
                pred = self._fit_recursive_feature_elimination(
                    best_inner_params, X_train_outer, y_train_outer, X_test_outer)
            else:
                # Train model with best inner parameters on the outer split
                model.set_params(**best_inner_params)
                model.fit(X_train_outer, y_train_outer)
                pred = model.predict(X_test_outer)

            outer_scores.append(self._transform_score_format(
                self.metric(y_test_outer, pred)))

            # Append variance
            variance.append(np.var(pred, ddof=1))

            print('\nResults for outer fold:\nBest inner parameters was: {0}'.format(
                best_inner_params_list[i]))
            print('Outer score: {0}'.format(outer_scores[i]))
            print('Inner score: {0}'.format(best_inner_score_list[i]))

        self.variance = variance
        self.outer_scores = outer_scores
        self.best_inner_score_list = best_inner_score_list
        self.best_params = self._score_to_best_params(best_inner_params_list)
        self.best_inner_params_list = best_inner_params_list

    # Method to show score vs variance chart. You can run it only after fitting the model.
    def score_vs_variance_plot(self):
        # Plot score vs variance
        plt.figure()
        plt.subplot(211)

        variance_plot, = plt.plot(self.variance, color='b')
        score_plot, = plt.plot(self.outer_scores, color='r')

        plt.legend([variance_plot, score_plot],
                   ["Variance", "Score"],
                   bbox_to_anchor=(0, .4, .5, 0))

        plt.title("{0}: Score VS Variance".format(type(self.model).__name__),
                  x=.5, y=1.1, fontsize="15")
```

File: Final.ipynb

```python
# Start Python Imports
import math, time, random, datetime

# Data Manipulation
import numpy as np
import pandas as pd

# Visualization
import matplotlib.pyplot as plt
import missingno
import seaborn as sns
plt.style.use('seaborn-whitegrid')

# Preprocessing
from sklearn.model_selection import train_test_split
from fancyimpute import IterativeImputer

# Evaluation
from sklearn.metrics import mean_squared_error
from nested_cv import NestedCV

# Modelling
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV, KFold
import xgboost as xgb
from keras.wrappers.scikit_learn import KerasRegressor
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)

# Set the data equal to some variables so we can use them later
train = pd.read_csv('experimental/data/train.csv')
test = pd.read_csv('experimental/data/test.csv')

print(train.shape)
print(test.shape)

# Correlation plot

```

```python
correlations = train.corr(method='pearson')
correlations

sns.heatmap(correlations,
            xticklabels=correlations.columns,
            yticklabels=correlations.columns,
            cmap=sns.diverging_palette(220, 10, as_cmap=True))

# Look for missing values by plotting
# Black columns are for each feature,
# And the horizontal white lines in those black columns are missing or NaN values
missingno.matrix(train, figsize = (30,10))

# Ok, many values are missing for some specific features
# Let's look at those
def find_missing_values(df, columns):
    """
    Finds number of rows where certain columns are missing values.
    ::param_df:: = target dataframe
    ::param_columns:: = list of columns
    """
    missing_values = {}
    print("Number of missing or NaN values for each column:")
    df_length = len(df)
    for column in columns:
        total_column_values = df[column].value_counts().sum()
        missing_values[column] = df_length - total_column_values
    return missing_values

all_missing_values_by_feature = find_missing_values(train, columns=train.columns)
all_missing_values_by_feature

def fill_ii(df):
    df_filled_ii = pd.DataFrame(IterativeImputer().fit_transform(df.values))
    df_filled_ii.columns = df.columns
    df_filled_ii.index = df.index

    return df_filled_ii

def data_engineering(train, test):
    # We have an odd number of observations.
    train = train.drop(train.index[0])

    # Concatenate all of training and testing data
    # Drop columns with too many missing values
    cc_data = pd.concat([train, test], sort=True)
    cc_data = cc_data.drop(['Id', 'SalePrice','Alley', 'FireplaceQu', 'PoolQC', 'Fence', 'MiscFeature'], axis=1)

    # We can make the output variable logarithmic, if we want RMSLE scoring
    train["SalePrice"] = np.log1p(train["SalePrice"])
    y = train['SalePrice']

    # One-Hot encoding of the whole dataset
    cc_data = pd.get_dummies(cc_data, prefix_sep='_')

    # Fill the missing data
    cc_data = fill_ii(cc_data)

    # Slice our data back into training and testing data
    X_train = cc_data[:train.shape[0]]
    X_test = cc_data[train.shape[0]:]

    return X_train,X_test,y

X,X_test,y = data_engineering(train,test)

# Check if values have been filled
# NOTE: It has been checked, that the values are correctly filled
missingno.matrix(X, figsize = (30,10))

def create_neural_network_model(first_neuron=64,
                                activation='relu',
                                optimizer='Adam',
                                dropout_rate=0.1):
    model = Sequential()
    columns = X.shape[1]

    model.add(Dense(64, activation=activation, input_shape=(columns,)))
    model.add(Dense(128, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(64, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(32, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(16, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(8, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, activation='linear'))

    model.compile(
        loss='mean_squared_error',
        optimizer = 'adam',
        metrics=['mean_squared_error']
    )

    return model

models_to_run = [KerasRegressor(build_fn=create_neural_network_model,verbose=0),
 ↪  RandomForestRegressor(), xgb.XGBRegressor()]

models_param_grid = [
                    { # 1st param grid, corresponding to KerasRegressor
                            'epochs' :                [50,100,150,200],
                            'batch_size' :            [16,32],
                            'optimizer' :             ['Adam', 'Nadam', 'Adamax'],
                            'dropout_rate' :          [0.1, 0.3, 0.5],
                            'activation' :            ['relu', 'elu'],
                            'first_neuron' :          [64, 128, 256]
                    },
                    { # 2nd param grid, corresponding to RandomForestRegressor
                            'max_depth': [3, None],
                            'n_estimators':
                             ↪  [100,200,300,400,500,600,700,800,900,1000],
                            'max_features' : [50,100,150,200]
                    },
                    { # 3rd param grid, corresponding to XGBRegressor
                            'learning_rate': [0.05],
                            'colsample_bytree': np.linspace(0.3, 0.5),
                            'n_estimators':
                             ↪  [100,200,300,400,500,600,700,800,900,1000],
                            'reg_alpha' : (1,1.2),
                            'reg_lambda' : (1,1.2,1.4)
                    }
                    ]
NUM_TRIALS = 50

RF_scores = []
XGB_scores = []
NN_scores = []

for trial in range(NUM_TRIALS):
    for i,model in enumerate(models_to_run):
        nested_CV_search = NestedCV(model=model,
         ↪  params_grid=models_param_grid[i], outer_kfolds=5, inner_kfolds=5,
                          cv_options={'sqrt_of_score':True,
                           ↪  'randomized_search_iter':30})
        nested_CV_search.fit(X=X,y=y)
        model_param_grid = nested_CV_search.best_params
        print('\nCumulated best parameter grids
         ↪  was:\n{0}'.format(model_param_grid))

        gscv = GridSearchCV(estimator=model, param_grid=model_param_grid,
         ↪  scoring='neg_mean_squared_error', cv=5)
        gscv.fit(X,y)

        print('\nFitting with optimal
         ↪  parameters:\n{0}'.format(gscv.best_params_))
        gscv.predict(X_test)
        score = np.sqrt(-gscv.best_score_)

        if(type(model).__name__ == 'KerasRegressor'):
            NN_scores.append(score)
        elif(type(model).__name__ == 'RandomForestRegressor'):
            RF_scores.append(score)
        elif(type(model).__name__ == 'XGBRegressor'):
            XGB_scores.append(score)

        print('\nFinal score for {0} was
         ↪  {1}'.format(type(model).__name__,score))

plt.figure()

rf, = plt.plot(RF_scores, color='b')
xgb, = plt.plot(XGB_scores, color='r')
nn, = plt.plot(NN_scores, color='g')

plt.legend([rf, xgb, nn],
           ["Random Forest", "XGBoost", "Neural Networks"],
           bbox_to_anchor=(0, .4, .5, 0))

plt.title('Test scores as RMSLE with hyperparameter optimization',
          x=.5, y=1.1, fontsize="15")
```

Raw results — all models

See the tables below for all results produced by the 3 models.

Iteration	XG Boost	Random Forest	Linear Regression	Neural Network
1	0,122859188	0,137762574	0,157838643	0,424597417
2	0,123088483	0,137719369	0,157838643	0,407714262
3	0,123402901	0,1374845	0,157838643	0,410855197
4	0,123001486	0,138013033	0,157838643	0,413298971
5	0,123043634	0,138073989	0,157838643	0,41628646
6	0,123001486	0,137403262	0,157838643	0,408994162
7	0,123243336	0,13804476	0,157838643	0,444788927
8	0,123108735	0,137285446	0,157838643	0,406931472
9	0,123858206	0,138240788	0,157838643	0,40478335
10	0,122918684	0,137571936	0,157838643	0,413898421
11	0,123108735	0,137304946	0,157838643	0,403653728
12	0,123402901	0,137916891	0,157838643	
13	0,123511505	0,137497828	0,157838643	
14	0,123568713	0,137540221	0,157838643	
15	0,122882801	0,137691387	0,157838643	
16	0,122946589	0,137471288	0,157838643	
17	0,123572897	0,137611241	0,157838643	
18	0,123088483	0,137990794	0,157838643	
19	0,122859188	0,138527384	0,157838643	
20	0,122946589	0,137677828	0,157838643	
21	0,122859188	0,13738611	0,157838643	
22	0,122882801	0,13707683	0,157838643	
23	0,123723335	0,18735659	0,157838643	
24	0,123043634	0,137573092	0,157838643	
25	0,123389034	0,13748971	0,157838643	

Iteration	XG Boost	Random Forest	Linear Regression	Neural Network
26	0,123108735	0,137115031	0,157838643	
27	0,123366768	0,137842322	0,157838643	
28	0,123088483	0,137791004	0,157838643	
29	0,12347769	0,13753602	0,157838643	
30	0,123124073	0,137426822	0,157838643	
31	0,122882801	0,137208043	0,157838643	
32	0,123001486	0,137575655	0,157838643	
33	0,123152871	0,138525026	0,157838643	
34	0,12388486	0,137975446	0,157838643	
35	0,123043634	0,137360331	0,157838643	
36	0,122882801	0,137507702	0,157838643	
37	0,123001486	0,137802523	0,157838643	
38	0,123713915	0,137491112	0,157838643	
39	0,12347769	0,137580108	0,157838643	
40	0,123393146	0,137787678	0,157838643	
41	0,12340808	0,13781492	0,157838643	
42	0,122882801	0,138086873	0,157838643	
43	0,123124073	0,137734089	0,157838643	
44	0,122882801	0,13748859	0,157838643	
45	0,123264978	0,137900782	0,157838643	
46	0,123507002	0,138190656	0,157838643	
47	0,123108735	0,137687897	0,157838643	
48	0,123589682	0,137676908	0,157838643	
49	0,123724134	0,138125492	0,157838643	
50	0,122882801	0,138388861	0,157838643	

Raw parameters — Random Forest

See the tables below for the hyper parameters of each iteration of the random forest.

Iteration	RMSLE score	max_depth	max_features	n_estimators
1	0,137762574	0	50	600
2	0,137719369	0	100	900
3	0,1374845	0	100	300
4	0,138013033	0	100	1000
5	0,138073989	0	150	800
6	0,137403262	0	100	900
7	0,13804476	0	50	1000
8	0,137285446	0	100	1000
9	0,138240788	0	50	600
10	0,137571936	0	100	900
11	0,137304946	0	100	700
12	0,137916891	0	50	400
13	0,137497828	0	100	1000
14	0,137540221	0	100	600
15	0,137691387	0	100	900
16	0,137471288	0	100	900
17	0,137611241	0	100	900
18	0,137990794	0	100	400
19	0,138527384	0	150	800
20	0,137677828	0	100	700
21	0,13738611	0	100	700
22	0,13707683	0	100	700
23	0,18735659	3	100	600
24	0,137573092	0	100	800
25	0,13748971	0	100	200

Iteration	RMSLE score	max_depth	max_features	n_estimators
26	0,137115031	0	100	400
27	0,137842322	0	100	600
28	0,137791004	0	100	800
29	0,13753602	0	100	500
30	0,137426822	0	100	700
31	0,137208043	0	100	700
32	0,137575655	0	100	1000
33	0,138525026	0	50	600
34	0,137975446	0	100	1000
35	0,137360331	0	100	600
36	0,137507702	0	100	600
37	0,137802523	0	100	700
38	0,137491112	0	100	1000
39	0,137580108	0	100	600
40	0,137787678	0	100	400
41	0,13781492	0	100	300
42	0,138086873	0	50	800
43	0,137734089	0	100	400
44	0,13748859	0	100	200
45	0,137900782	0	50	900
46	0,138190656	0	50	500
47	0,137687897	0	100	800
48	0,137676908	0	100	800
49	0,138125492	0	50	300
50	0,138388861	0	150	700