REPORT OF SAGAL AND AKUZIKE

FYS-STK4155 Project 3

ABSTRACT

We explored a number of machine learning models: Linear regression, decision trees, neural networks and random forests on a bike sharing dataset. The dataset is a regression dataset containing bike sharing data from cycling in London from 2015 to 2016. We fitted the data to each of the models to obtain the optimal parameters for each model. For linear regression: simple linear regression (ordinary least squares), Ridge and Lasso models in scikit learn were used. A neural network built from scratch was used for the training and testing. Random forest regressor and decision tree regressor from scikit learn were used. All linear regression models showed 2 as the optimal number of degrees, higher degrees led to overfitting. Ridge, in addition, had *alpha=1* as the optimal while Lasso had *alpha=0.0003* as the optimal hyper parameter with train MSE of 0.08520218169206484 and train R2 score of 0.3343406502924487. Random Forest had the optimal depth as 8, with optimal number of trees as 100. Out of all 19 features, features 2 and 3 were the most important followed by 1 and 0 in that order.

INNHOLDSFORTEGNELSE

REPORT OF SAGAL AND AKUZIKE **1**

ABSTRACT **1**

INTRODUCTION **2**

THEORY 3
Linear regression *3*
Neural Network *4*
Decision Tree *6*
Random forest *8*
DATA DESCRIPTION 10

IMPLEMENTATION **11**

IMPLEMENTATION OF DATA 11
IMPLEMENTATION OF LINEAR REGRESSION 11
OLS *11*
Ridge *12*
Lasso *12*
IMPLEMENTATION OF NEURAL NETWORK 12
IMPLEMENTATION OF DECISION TREES 12
IMPELENTATION OF RANDOM FOREST 13

RESULT **14**

LINEAR REGRESSION 14
ols *14*
ridge *14*
LASSO *15*
ARTIFICIAL NEURAL NETWORK 15
DECISION TREES 17

INTRODUCTION

Machine learning is a scientific study of algorithms and statistical models, and is applied to various kinds of data to make predictions after learning from a training set. In this project we have studied a regression data set containing bike sharing(London 2015 to 2016) obtained from Kaggle[1]. We applied some supervised methods, and measured their performance using the MSE, and R2-score. The algorithms we used were; Linear Regression(OLS, Ridge and Lasso), Decision trees, Neural network and Random forest. We used the SKLEARN package to do various calculations, and made some codes to help us with the analysis.

For Linear Regression we used OLS, Ridge and Lasso. We looked over various polynomial degrees, and noted down the MSE, and R2score for every degree. We then used the obtained information to look at the bias variance tradeoff. For Lasso, and Ridge we tried to find optimal regularization parameter so that we gained best possible score for the MSE, and R2score.

Neural Network was a bit cumbersome to do analysis on. The computation was time consuming especially when looking over many layers(with many nodes as well). We calculated at R2score and MSE for networks containing 1,2 and 3 hidden layers with different numbers of nodes, and chose the layers we thought would give best MSE, and R2score. For this layers we then iterated over a given range and looked at the train, and test error and observed overfit and underfit.

The next Algorithm was Regression Trees. The method was not as time consuming as the Neural Network, and was easy to compute. Here we did pre-pruning, and tried to find optimal number of leave nodes together with the ideal minimum number of sample split. When we did that, we changed the depth of the tree and studied both the training, and test error. We also looked at how the R2score evolved when the complexity of the tree increased.

Finally we did analysis using Random Forests. We looked over different number of Trees, and looked at MSE, and R2score as we did for the above methods.

This report consist of an introduction, implementation, discussion, conclusion and a reference part. We will continue with introduction by going through necessary theory.

THEORY

LINEAR REGRESSION

If we have an input vector $X^T = (X_1, X_2, X_3, \ldots . X_p)$ and want to predict an output Y we can describe it with the linear regression model as follows:-

$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j B_j \qquad (1)$$

The $\beta_j`s$ are unknown coeficients, $f(X)$ represents the outcome Y and X_j the design matrix. The design matrix contains variables that the output is dependent on, and contains p number of predictors. For example, if you have an experiment where you measure 5 different quantities, we say number of predictors are 5. If you make n = 20 measurments, we say that the number of data points are 20. In total, we can represent this in a design matrix with dimension $p \times n$. The variables of the design matrix can have different sources such as quantitative inputs, where you have different measurements that together give an output. We can also have an output that can be described using a polynomial representation, and that is what we will be using in our report. But, no matter the source of the design matrix, the model is linear in the parameters.

Equation(1) above, can be rewritten as follows:-

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\epsilon} \qquad (2)$$

$\boldsymbol{y}$ is a known quantity and represents as $f(X)$ in (1) the output. $\boldsymbol{X}$ is the design matrix, $\boldsymbol{\beta}$ is the coeficients while $\boldsymbol{\epsilon}$ is a vector containing the error in each measurement. In the above expression $\boldsymbol{\beta}$ and $\boldsymbol{\epsilon}$ are unknowns. And the question that is often asked is how to get the optimal value for the coefficients $\boldsymbol{\beta}$.

OLS

One very popular method of finding the coefficients is using ordinary least square error(OLS). Here we first predict a model using expression (3) below:

$$\tilde{\boldsymbol{y}} = \boldsymbol{X\beta}. \qquad (3)$$

Then we try find the spread between the true values $\boldsymbol{y}$ and $\tilde{\boldsymbol{y}}$ with help of our defined cost function in equation (4):

$$C(\boldsymbol{\beta}) = \frac{1}{n}\{(\boldsymbol{y} - \tilde{\boldsymbol{y}})^{\boldsymbol{T}}(\boldsymbol{y} - \tilde{\boldsymbol{y}})\}. \qquad (4)$$

By minimizing the cost function in (4) with respect to $\boldsymbol{\beta}$ we get the following solution:

$$\frac{\delta C(\boldsymbol{\beta})}{\delta \boldsymbol{\beta}} = 0 = \boldsymbol{X^T}(\boldsymbol{y} - \boldsymbol{X\beta}),$$

and juggling around with the expression above we get the following solution for $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} = (\boldsymbol{X^T X})^{-1}\boldsymbol{X^T y} \qquad (5)$$

RIDGE

Equation (5) above shows that finding optimal value for $\boldsymbol{\beta}$ involves taking the inverse of $(\boldsymbol{X^T X})^{-1}$. From linear algebra we know we can`t calculate the inverse of a singular matrix, and thus obtaining the coefficients in (5) becomes challenging. Ridge regression is a method that solves this problem by adding a hyper parameter λ and an identity matrix to equation (4) so that we get:

$$C(\boldsymbol{\beta}) = \frac{1}{n}\{(\boldsymbol{y} - \tilde{\boldsymbol{y}})^{\boldsymbol{T}}(\boldsymbol{y} - \tilde{\boldsymbol{y}})\} + \lambda \boldsymbol{\beta^T \beta}. \qquad (6)$$

Minimizing the cost function again with respect to $\boldsymbol{\beta}$ gives us:

$$\boldsymbol{\beta} = (\boldsymbol{X^T X} + \lambda \boldsymbol{I})^{-1}\boldsymbol{X^T y}. \qquad (7)$$

For Ridge regression it is important choosing a λ value that gives the lowest possible error. As we will see later, having optimal value of lambda is essential for the ridge regression method.

LASSO

The Ideal scenario when analyzing a data is having many data points so that the model can be predicted well. With less data points a challenge arises for producing a good model. To understand this, you can think of training a kid to learn 100 names. If you teach him all the names, then he would be able to answered if he is asked to tell the names. If you else only know 20 names and teach him that, then the kid would not be able to answer all questions. The same is for the data points. The more data we have, the better the model we can make since the model has been trained well. With less data points, then the opposite is true.

So, how do we solve this problem? One way is doing what we call variable selection. We take a look at the predictors, and find out which are contributing less. Doing that, we get a less complex model that may give a good approximation. Lasso regression is a method that uses that logic. Just like Ridge it uses a penalty parameter, but instead of $\boldsymbol{\beta}_j^2$, $\left|\boldsymbol{\beta}_j\right|$ is used. The total equation is as shown below:

$$\boldsymbol{f(X) = \beta_0 + \sum_{j=1}^{p} X_j B_j + \lambda \sum_{j=1}^{p} \left|\beta_j\right|} \qquad (8)$$

So $\left|\boldsymbol{\beta}_j\right|$ just like $\boldsymbol{\beta}_j^2$ shrinks the coefficients. It also in addition set some of the $\boldsymbol{B_j}$ values equal to zero when $\boldsymbol{\lambda}$ is big enough. That is why it is said that lasso performs a variable selection. The model that is yield is *sparse*, and only a subset of variables are used for prediction.

Just like ridge, choosing optimal value for λ is essential.

For linear regression we used the notes of Morten to get necessary theory.[5]

NEURAL NETWORK

Artificial Neural Network(ANN) is a computational method that is inspired by the way biological neural network(BNN) in the brain processes information. The simplest unit of the BNN is the neuron and it functions by receiving signals from the dendrites as shown in figure (1) below. The signal is processed at the cell body, and sent through the axon of the neuron. In the same way, as can be been seen at the lower part of picture(1), the artificial neuron receives input from other neurons. The input is processed(by being transformed by mathematical expressions), and sent out again. In that very simply explained illustration in (1), artificial neurons mimics the biological neurons, but are much more simpler. The biological neurons are living things, and function in a much more complex way. Our goal in this part, is to explain how an ANN functions, and since the node(aritificial neuron) is the most simple unit of the ANN, we will start with it.

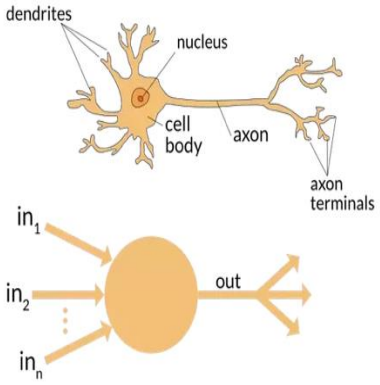


FIGURE 1: THE ABOVE IMAGE SHOWS ILLUSTRATION OF A BIOLOGICAL NEURON, WHILE THE LOWER SHOWS AN ARTIFICIAL NEURON.[1]

SINGLE NEURON

The single *neuron* is the simples possible ANN, as illustrated in figure(1) above. An arbitrary node takes in inputs n number of inputs: x_1, x_2 …. x_n(+1 intercept is also included) and produces an output $h_{w,b}(x)$;

$$\boldsymbol{h_{w,b}(x) = f(W^T x) = f\left(\sum_{i=1}^{3} W_i x_i + b\right)} \qquad (9)$$

$\boldsymbol{W}$ in the equation presents the weights connecting different layers, and a bias $\boldsymbol{b}$. The function $\boldsymbol{f}$ is an activation function, and can have different forms. Below we list some of them;

a) Sigmoid Function

Sigmoid function also called the logistic function is expressed in equation (10) below. As shown in figure(2) the logistic function maps its values between 0, and 1. So with large numbers(either positive or negative) we face the problem of saturation.

$$\boldsymbol{\sigma(x) = \frac{1}{1 + e^{-z}}} \qquad (10)$$

b) Hyperbolic Tangent Function

The hyperbolic tangent(tanh) function has output values in the range(-1,1), and thus makes it more robust when dealing with negative values. It is expressed as follows;

$$\boldsymbol{tanh(x) = \frac{sinh(x)}{cosh(x)}} \qquad (11)$$

c) ReLU

Rectified Linear Unit(RELU) is a non linear function that has outputs ranging from [0,∞]. It has the advantage of not getting saturated for very high positive values, but produces what is called dead neurons if the inputs are large and negative(outputs only zero). It is expressed by the formula below;

$$\boldsymbol{ReLU(x) = max(0, x)} \qquad (12)$$

There are other activation functions like the identity, and ELU but we won't be dealing with them now. Figure(2) below shows the sigmoid, ReLU and tanh functions.

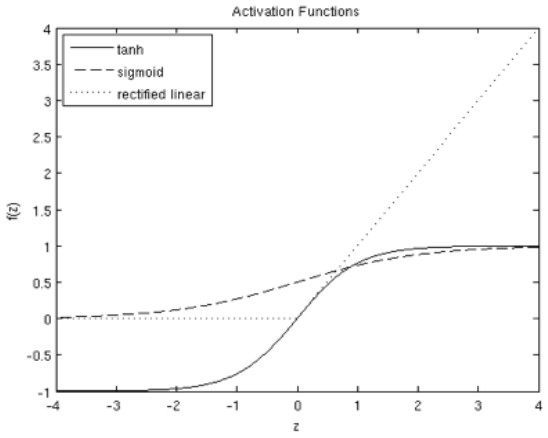


FIGURE 2: SHOWS RELU, SIGMOID AND THE HYPERBOLIC FUNCTIONS. THE FIGURE IS LABELD WITH FUNCTION TYPE.[3]

MULTI-LAYER PERCEPTRON

By hooking together many "single neurons" so that output of a neuron can be the input of another, we can end up with what is shown figure (3) below.

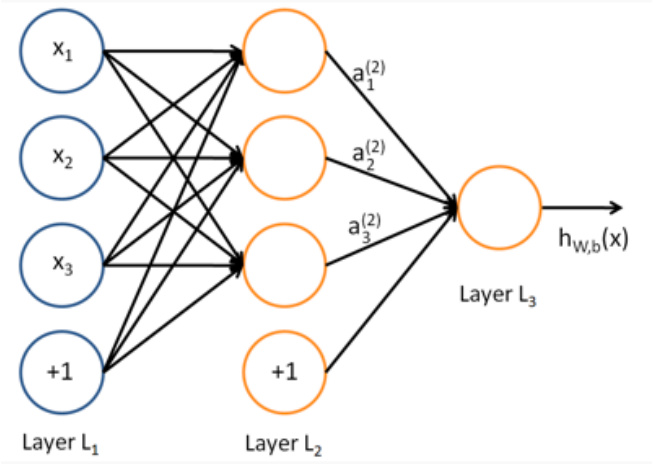


FIGURE 3: SHOWS AN ILLUSTRATION OF AN INPUT LAYER, HIDDEN LAYER AND AN OUPUT. [3]

The sketch above shows a simple illustration of a Multi-Layer Perceptron(MLP). It consist of an input layer with different inputs, a hidden layer where the inputs are transformed, and an output neuron(also a layer) where we get a result. The +1 in the figure just illustrates the bias units, and corresponds to the intercept term.

The MLP has shown to be very effective to solve both Regression, and classification task. In this report we will assign it to solve a Regression problem.

So, how are the values transformed in between the different layers? As we described in equation(9) there are weights and biases involved at every transitions. The weights and biases are summed together with the inputs, and then transformed with a given activation.

If we assign l to the number of layers present, then the weights and biases can be presented in matrix form as follows;

$$\boldsymbol{Weight}: \boldsymbol{W_{ij}^{(l)}}, \boldsymbol{Bias}: \boldsymbol{b_i^{(l)}}$$

Here the indices *i* represents the neuron in layer $l+1$ whereas j presents the layer l. As we can see the weights makes a connection between the different layers. If we take for example the weights that connect layer 1, and 2 in figure(3) we get that;

$$\boldsymbol{W_{33}^{(1)}} = \begin{matrix} \boldsymbol{w_{11}^1} & \boldsymbol{w_{12}^2} & \boldsymbol{w_{13}^3} \\ \boldsymbol{w_{21}^1} & \boldsymbol{w_{22}^2} & \boldsymbol{w_{13}^3}, \\ \boldsymbol{w_{31}^1} & \boldsymbol{w_{32}^2} & \boldsymbol{w_{13}^3} \end{matrix} \qquad \boldsymbol{b_3^{(1)}} = \begin{matrix} \boldsymbol{b_1^1} \\ \boldsymbol{b_2^1} \\ \boldsymbol{b_3^1} \end{matrix}$$

And in that way, the input values that get in and are passed through the network. Depending on what kind of a problem we are dealing with, the output is processed differently.

At the beginning of the iteration the weights and biases are randomly chosen(can also be zeros), so they most probably don't predict correct. So training is important, and for the MLP a back propagation algorithm is used. Generally this works by calculating the error between the output and the data, going backwards and updating the biases and weights for every iteration. For this optimization methods like stochastic gradient, and gradient descent can be used. We won't be going into detail of this, but it is well explained in our previous project[4]. But generally this is what is done when an MLP is learning:-

a) Values are passed in from the input(data), and passed through different layers of the network. At every node a transformation using (9) occurs.
b) When ouput is gotten, error is calculated and a backpropagation algorithm is used to update the weights at every iteration.

DECISION TREE

Decision Trees(DT) are widely used supervised methods that do both classification and regression tasks. They operate by making if and else statements, and splitting a specific data till we end up with a decision. Figure (4) below shows an example of a Classification Decision Tree.

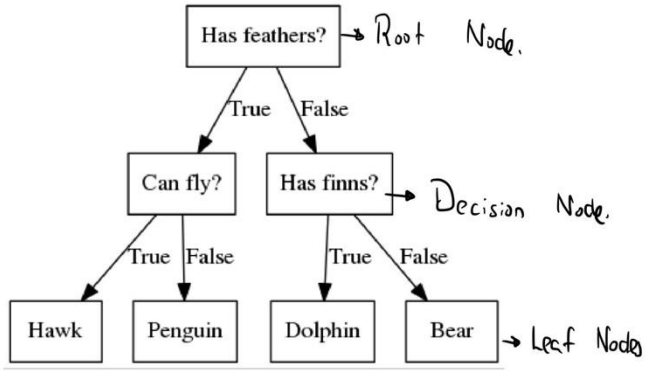


FIGURE 4: SHOWS AN ILLUSTRATION A DECISION TREE FOR CLASSIFICATION OF SPECIFIC ANIMALS GIVEN SOME PREDICTORS.

In the above we assume distinguishing between 4 animals; Hawk, Penguin, Dolphin and Bear. To do so, we use the following predictors; *Has feathers*, *can fly* and *has fins*. By asking the questions, we end up with the animal we are looking for. This is how Decision Trees in Machine Learning generally functions. They are generally simple to explain, and present.

The *Has feather* question in figure (4) is called the root node, while the other two following questions are called children nodes or decision nodes. The bottom node is called the leaf node and presents a particular result. For classification it shows which unit class we end up as shown above, while for Regression we get a value.

One can ask the following questions; *When do we stop dividing the tree, and how is the data split?* Generally one stops increasing depth of Decision Tree when end result is gotten. This means for example for a classification problem when a given sample is classified, and the data proportion is pure. Like in the example in figure (4) splitting is stopped when animal type is named. For regression it is stopped when a certain value is reached. The number of samples at the leaf node can then be as low as 5 samples.

The other question is, how the data is split? This is done as we saw in figure (2) using the features. The feature to split by, is again chosen by using the predictor that gives the lowest error. One, at when you split, the data is divided at minimum entropy .Different parameters are used to do this, and we will look closer into it. But first, we will distinguish between a Regression Tree, and a Classification Tree.

REGRESSION TREES

Simply put, one can say that for a Regression Tree(RT) we have continuous outputs, whereas for

classification the output are categorical. Due to this, the way that error in a particular attribute is calculated is different.

But generally, when building a Decision Tree, there are mainly two steps:-

1. Dividing the predictor space into J different and non overlapping regions R_1, R_2,…., R_J.
2. For every sample that is in region R_J the same prediction is made. That means taking the mean average of all the samples.

Assuming we divide the regions into high dimensional rectangles, the goal is to find boxes R_1, R_2,…., R_J that minimize the following equation;

$$\sum_{j=1}^{J} \sum_{i \in R_J} (y_i - \widehat{y_{R_j}})^2. \quad (13)$$

$\widehat{y_{R_J}}$ in (13) is the mean of all the observations in the box R_J.

When calculating the possible partitions of the features into J boxes, it is computationally infeasible to consider every partition. Because of that, a *top-down, greedy* approach known as *recursive binary splitting* is taken. It is *top-down* because the method begins at the top of the tree and considers all samples belonging to **one** single region. Then it successively splits the predictor space, creating two new regions. *Greedy* because it determines the best split at every step, rather than choosing a step that would eventually lead to a better tree in the future.

So, to perform the *recursive binary splitting* a feature X_J, and a cut point s is chosen so that we get the smallest reduction in equation (13) above. This leads to us forming following regions; $\{X | X_J < \mathrm{s}\}$ and $\{X | X_J \geq \mathrm{s}\}$.

To find the feature to split, and the value to choose as cut point, we have to look through all features, and the possible split values in each particular feature. Explaining it in greater detail, for every s and j, we define the following half planes;

$$\boldsymbol{R_1(j,s) = \{X | X_J < \mathrm{s}\}} \textbf{ and } \boldsymbol{R_2(j,s) = \{X | X_J \geq \mathrm{s}\}},$$

and seek the value of s and j that minimize the equation;

$$\sum_{i:\, x_i \in R_1(j,s)} (y_i - \widehat{y_{R_1}})^2 + \sum_{i:\, x_i \in R_2(j,s)} (y_i - \widehat{y_{R_2}})^2 \quad (14)$$

$\widehat{y_{R_1}}$ and $\widehat{y_{R_2}}$ are respectively the mean response for the training observations in $R_1(j,s)$ and $R_2(j,s)$. When the new regions are created, the process of splitting, and using (14) to determine best feature and split value is done again. We continue till we reach a stopping criterion which can be low as 5 samples.

OVERFITTING

Doing as explained above will create a tree that eventually fits the training data well, but performs bad on the test data. It is even possible to get 100% fit on the training data if the leave-nodes get pure. To overcome this problem something called pruning can be done and there are two variants of them:-

a) Pre-Pruning

Also called *early stopping* and includes stop of growing of the tree if it does not prove to be statistically significant. Ways to do it are to put a limit on depth of the tree, limiting maximum number of leaves and requiring a minimum number of points in a node to keep splitting.

b) Post-Pruning

Even though the Pre-Pruning I explained above might increase the performance of the method, it could happen by stopping the splitting of the tree we miss on gaining a bigger score. With this I mean at a particular step it might prove not so smart to split, but after splitting for two times it can happen we get a greater drop in error. For this reason Pre-pruning can be a quick fix that is too short sighted.

What can instead be done is to grow a very large Tree T_0, and stop the splitting only when a minimum node size(e.g 5) is reached. Then we can prune the Tree backwards using the *cost-complexity pruning* that is described by (15) below:-

$$\sum_{m=1}^{|T|} \sum_{i:\, x_i \in R_2(j,s)} (y_i - \widehat{y_{R_m}})^2 + \alpha |T|. \quad (15)$$

In the formula above $|T|$ is the number of terminal nodes(Decision nodes) in the tree, R_m is the rectangle corresponding to the *m*th terminal node, and $\widehat{y_{R_m}}$ is the predicted response in R_m. The constant α is the tuning parameter and controls the tradeoff between the subtree`s complexity and the fit to the training data. With α = 0, we get the whole tree with no pruning, and as we increase α we get a smaller with less decision nodes. The α parameter is reminiscent of

the lasso penalty parameter that decreased the influence of the beta parameters as λ increased in equation (8).

CLASSIFICATION TREE

As I named at the beginning about Regression Trees, the main difference between Classification Trees(CT) and RT is the output. The output of the Regression Trees are quantitative whereas that of classification are qualitative. An example of a Classification Tree can be seen in figure(4).

Also, the way the response in every region is considered is different between the two methods. For RT the mean response in every region R_m is calculated. Classification Trees the classes occurring in every region is considered, and when making a leave node, the region is classified with respect to the most occurring sample. Due to that, the way the error is calculated is different. Following equations are commonly used when considering the purity of a certain node:-

a) Gini index

$$\boldsymbol{G} = \sum_{\boldsymbol{k=1}}^{\boldsymbol{K}} \hat{\boldsymbol{p}}_{\boldsymbol{mk}}(\boldsymbol{1} - \hat{\boldsymbol{p}}_{\boldsymbol{mk}}) \qquad (16)$$

$\hat{p}_{mk}$ above represents proportion of training observation in the *m*th region for the *k*th class in. The equation as a whole measures total variance across the *K* classes. When $\hat{p}_{mk}$ is close to zero, or 1 then the Gini index takes on small values. It gives thus an indication of the node impurity.

b) Entropy

This is an alternative to the Gini index above. It is calculated by equation below:-

$$\boldsymbol{D} = -\sum_{\boldsymbol{k=1}}^{\boldsymbol{K}} \hat{\boldsymbol{p}}_{\boldsymbol{mk}} \boldsymbol{log}(\hat{\boldsymbol{p}}_{\boldsymbol{mk}}) \qquad (17)$$

Since $\hat{p}_{mk}$ is between zero and 1, (17)has values equal zero or above. The equation just like the Gini Index takes on small values when a node is very pure.

Theory under Decision Trees we have used the book of Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani: An Introduction to Statistical Learning chapter 8.1.

RANDOM FOREST

One model that has been widely used in many applications and grown in popularity over the few last years in Random forests (Murphy, 2012). A random forest is an ensemble of decision trees generally trained via bagging method (Geron, 2017) hence improving bagged trees. Random forest algorithm introduces extra randomness when growing trees on bootstrapped training samples. It searches for the best feature among a random subset of features,m rather than considering set of all features,p. At each split in a tree, a fresh sample of m features is taken, and typically we chose $m \approx \sqrt{p}$ resulting in greater tree diversity and higher training speed. The great diversity further aids in yielding a better model by trading high bias for a lower variance. The need to prune the trees is eliminated as a result of the feature selection. A new parameter is introduced, number of featured to be considered (Marsland, 2014).

The basic random forest training algorithm can be applied to both classification and regression problems. If we consider growing a forest of N trees, then for each tree n :

- create a new bootstrap sample of the training set organized in our design matrix X
- use this bootstrap sample to train a random-forest T_n decision tree
- at each node of the decision tree, randomly select m features, and compute the information gain (or Gini impurity) only on that set of features, selecting the optimal one
- repeat until the tree is complete
- Output the ensemble of trees $\{T_n\}_1^N$

The output of the forest is the majority vote for classification and the mean response for regression. To make a prediction at a new point x, for regression:

$$\boldsymbol{f}_{\boldsymbol{rf}}^{\boldsymbol{N}}(\boldsymbol{x}) = \frac{\boldsymbol{1}}{\boldsymbol{N}} \sum_{\boldsymbol{n=1}}^{\boldsymbol{N}} \boldsymbol{T_n}(\boldsymbol{x}) \qquad (18)$$

And for classification, Let $C_n(x)$ be the class of prediction of the n^{th} random-forest tree. Then $C_{rf}^N(x) = majority\ vote\ \{C_n(x)\}_1^N$ (Hastie, Tibshirani, & Friedman, 2009).

The expectation of an average of N such trees is the same as the expectation of any one of them since each tree generated by bagging is identically

distributed (i.d.). An average of N i.i.d random variables, each with variance σ^2 has variance $\frac{1}{N}\sigma^2$. If the variables are simply identically distributed, but not necessarily independent) with positive pairwise correlation ρ, the variance of the average is

$$\boldsymbol{D} = -\sum_{k=1}^{K} \hat{\boldsymbol{p}}_{mk} \boldsymbol{log}(\hat{\boldsymbol{p}}_{mk}) \quad (19)$$

RANDOM FOREST FEATURES

OUT OF BAG (OOB) SAMPLES

For each observation $z_i = (x_i, y_i)$ its random forest predictor is constructed by averaging only those trees corresponding to bootstrap samples in which z_i did not appear (Hastie et al., 2009). An OOB error estimate is almost identical to that obtained by K-fold cross-validation. For this reason, random forests can fit in one sequence, with cross-validation being performed along the way, unlike other nonlinear estimators.

VARIABLE IMPORTANCE

Just as with gradient-boosted models, variable importance plots can be constructed for random forests. When we bag a tree in a random forest, it is no longer clear which variables are most important to the procedure. However, when growing the tree the OOB samples are passed down the tree, and the prediction accuracy is recorded. Then the values for the *j*th variable are randomly permuted in the OOB samples, and the accuracy is again computed. The decrease in accuracy as a result of this permuting is averaged over all trees, and is used as a measure of the importance of variable *j* in the random forest (Hastie et al., 2009). For regression trees, the importance of each predictor is measured using the (Residual Sum of Squares) RSS while for classification trees the Gini index is used (James, Witten, Hastie, & Tibshirani, 2013).

ASSESSING ACCURANCY OF MODEL

The linear regression methods I have talked about need to tested, and seen how well the models that are predicted fit the data. To do so, two quantities are assessed:-

a) MSE
b) R^2 statistics

MSE

The mean squared error estimates the spread between the predicted model and the measurement as equation (4) does. It tells us the average amount the response will deviate from the true regression line. The equation can be expressed as:-

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}(x_i)\right)^2. \quad (20)$$

$y_i$is the measurement value and $\hat{y}(x_i)$ represents the predicted model. Later if we divide the data into a test and train, and predict the model using the train data, then we can see how good the the predicted model is by testing it with the test data.

Equation (20) is the expectation value of

$$\left(y_i - \hat{y}(x_i)\right)^2$$

Writing the above as vectors, we have that:

$$y_i \rightarrow \boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\epsilon} = \boldsymbol{f} + \boldsymbol{\epsilon}$$

$$\left(y_i - \hat{y}(x_i)\right)^2 \rightarrow \tilde{\boldsymbol{y}} = \boldsymbol{X\beta}.$$

The expectation value can then be written as:

$$E|(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2| = E|(\boldsymbol{f} + \boldsymbol{\epsilon} - \tilde{\boldsymbol{y}})^2|$$

By adding and subtracting the expectation value of $\tilde{\boldsymbol{y}}$, and doing a bit computation the following result is gotten:

$$E|(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2| = E\left|\left(\boldsymbol{y} - E(\tilde{\boldsymbol{y}})\right)^2\right| + Var(\tilde{\boldsymbol{y}}) + \sigma^2 \quad (21)$$

The first term is what we call the squared bias term, and gives an indication of how far the predicted model is from the measurement values. The second term is the variance , and tells us how the variance of our predicted model is. The last term is a measure of the variance in $\boldsymbol{\epsilon}$.

In the prediction of our model, there is not much we can due about the variance in the $\boldsymbol{\epsilon}$. But, we can modulate the bias and variance term according to the complexity we are fitting the model, and also the

linear regression method we are using. We will se more of this later in the report.

R^2 STATISTICS

R^2 statistics is an accurancy model used to see how much proportion of the variance of the data that is explained by the regression method. It takes values between 1, and 0. So, a R^2-score equal to 0.6 says that 60% of the data is explained by the regression method. The R^2- can be calculated using the equation below:-

$$R^2 = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \hat{y}(x_i))^2}{\sum_{i=0}^{n-1}(y_i - \bar{y}(x_i))^2}. \quad (22)$$

$\bar{y}$ in equation(22) is the mean value of the true values y_i.

RESAMPLING TECHNIQUES

Resampling methods involve repeatedly drawing samples from a training set and refitting a model of interest on each sample to obtain additional information about the model (James). Two most commonly used resampling techniques are:

a) Cross-validation
b) Bootstrap.

CROSS-VALIDATION

Cross-validation (CV) is a method used to estimate test (prediction) error by holding out a part of the training observations from the fitting process and then applying statistical learning methods to the observations held out (James). This method estimates the average generalization error when the method f(x) is applied to an independent test samples from the joint distribution of X and Y (Hastie).

One strategy used in Cross-validation is the Validation set approach where the set of observations id randomly divided into two parts: training and validation sets. The model is fit on the training set. The fitted model is used to predict responses for observations in the validation set (James). K-fold cross-validation randomly divides the set of observations into k equal groups, referred to as folds. It is mostly performed with k=5 or k=10 for computational advantages. The procedure is repeated K times, while changing the fold used for validation each time.. For each iteration, the model is fit on a training set made up of k-1 folds. The remaining last fold is regarded as the validation set. K Mean Squared Errors MSEs are calculated. The k-fold CV estimate is computed by averaging these values (James).

DATA DESCRIPTION

A regression dataset containing bike sharing data from cycling in London from 2015 to 2016 was used obtained from Kaggle https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset . The data has the following attributes with their respective description:

TABLE 1: SHOWS INFORMATION ABOUT THE DIFFERENT ATTRIBUTES.

Attribute name	Description
TimeStamp	Gives date the data was obtained.
Cnt	Counts of sharing
Temperature	The real temperature measured
Temperature_feels	The temperature it felt like
Humidity	Humidity in percentage
Wind_speed	The measured wind speed in km/h
Weather_code	Category of weathet, that were as follows:- 1 = Clear 2=Scattered clouds/few clouds 3=Broken Clouds 4=Cloudy 7 = Rainy/light Rain shower/Light rain 10 = rain with thunderstorm 26 = snowfall 94 = Freezing Fog
Is_holiday	Gives 1 if yes, and 0 else

Is_weekend	1 if yes, 0 else
Season	0 = Spring 1=summer 2=fall 3=winter

IMPLEMENTATION

Here we will explain implementation of the data(processing, and different changes done) and the four supervised methods we used.

IMPLEMENTATION OF DATA

Using pandas, we got the below view of our data. As can be seen, there are 9 columns where the column called “cnt” is the target of our data. Also, when doing analysis we removed the “TimeStamp” which just contained the information of the date.

	cnt	t1	t2	hum	wind_speed	weather_code	is_holiday	is_weekend	season
0	cnt	t1	t2	hum	wind_speed	weather_code	is_holiday	is_weekend	season
1	182	3.0	2.0	93.0	6.0	3.0	0.0	1.0	3.0
2	138	3.0	2.5	93.0	5.0	1.0	0.0	1.0	3.0
3	134	2.5	2.5	96.5	0.0	1.0	0.0	1.0	3.0
4	72	2.0	2.0	100.0	0.0	1.0	0.0	1.0	3.0

FIGURE 5: FIGURURE SHOWS OUTLAY OF DATA VISUALIZED USING PANDAS FRAME

Then we checked if our data contained any “holes”, or missing data and found out that there were none. Also we looked at the correlation of our data, and saw that the columns t1, and t2 were highly correlated as can be seen in figure(6) below. We decided thus to remove t2. The handling of the codes can be seen in our

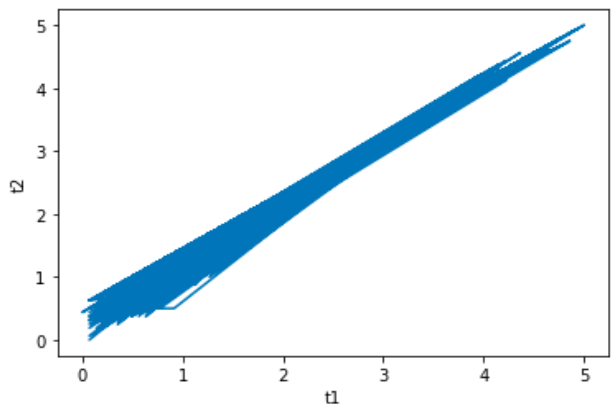


FIGURE 6: FIGURE SHOWING CORRELATION BETWEEN TEMPERATURE FEEL(T2), AND REAL TEMPERATURE(T1)

Next, we started the preprocessing of our data. To increase the performance of our methods, we use MaxMin scaling from sklearn. Below the code used is shown.

```
# Features and targets
x = data.loc[:, data.columns != 'cnt'].values # Feature
Y = data.loc[:, data.columns == 'cnt'].values #Target
#sc = preprocessing.StandardScaler(with_mean = True). I
# To avoid my prediction to be close to zero
sc = MinMaxScaler(feature_range=(0, 5))
#Y=np.log1p(Y)
# Scaling of the output
Y = sc.fit_transform((Y[1:,]))
# Transofrming the output
Y=np.log1p(Y)
```

FIGURE 7: SHOWS A CODE SNUT FOR DATA PROCESSING.

We chose to scale between 0 and 5 so that we could fix a problem we realized when the scaling was between 0, and 1. Ridge and Lasso were giving results that the bigger the regularization parameter, the better the fit. This may be understood that since the regularization parameter is shrinking values towards zero, that the zero line might be close to values between 1, and zero. We will explain more when we get to that part.

Also, we OneHotEncoded the categorical features so that we ended up with 20 columns instead of only 8.

IMPLEMENTATION OF LINEAR REGRESSION

We fit our data bike sharing data to linear regression models with different parameters to obtain the best parameters for each of the models. The data was fit to a simple linear regression (Ordinary Least Squares) model, Ridge and Lasso models.

OLS

The data was fit to a simple linear regression model with different polynomial degrees to find the most suitable polynomial degree for our dataset. The model *linearRegression* from scikit learn was used for the fitting. Only degrees 1 to 5 were used. Higher degrees were left out due to the higher execution time that they required.. The figure below shows a snippet of the source code that was used.

```
polyDegree = [1,2,3,4,5]
for deg in polyDegree:
    poly = PolynomialFeatures(degree = deg)
    X_train = poly.fit_transform(X_train)
    X_test = poly.fit_transform(X_test)
    model = LinearRegression()
    model.fit(X_train, y_train)

    y_tilde= model.predict(X_train)
    y_pred = model.predict(X_test)
    mse.append(mean_squared_error(y_test,y_pred))
    r2.append(r2_score(y_test,y_pred))
```

FIGURE 8: SHOWS A SNIPPET OF OLS CODE WE USED.

RIDGE

The linear model, Ridge from scikit learn was used to fit the data to a ridge regression model. The fitting was carried out for polynomial degrees 1 to 5 and for the following range of alpha values: [0,1e-20,1e-15,1e-10,1e-5,1e-4,1e-3,1e-2,1e-1,1,10,100,1000,10000]. The code snippet in the figure below shows part of the implementation that was used.

```
alph= [0,1e-20,1e-15,1e-10,1e-5,1e-4,1e-3,1e-2,1e-1,1,10,100,1000,10000]
alphaLength = len(alph)
degreeLength=len(degree)
mse=np.empty((degreeLength, alphaLength))
r2 = np.empty((degreeLength, alphaLength))
test_error = np.empty((degreeLength, alphaLength))
train_error = np.empty((degreeLength, alphaLength))

for d, deg in enumerate(degree):
    poly = PolynomialFeatures(degree = deg)
    X_train = poly.fit_transform(X_train)
    X_test = poly.fit_transform(X_test)
        # for each degree, fit model into train set with correct number of predictor

    for a, afa in enumerate(alph):
        model = Ridge(alpha = afa,random_state = None)
        model.fit(X_train, y_train)
        y_tilde= model.predict(X_train)
        y_pred = model.predict(X_test)
        train_error[d,a]=mean_squared_error(y_train,y_tilde)
        test_error[d,a]=mean_squared_error(y_test,y_pred)
```

FIGURE 9: SHOWS SNIPPET OF RIDGE CODE

From the r2 scores and MSEs for different alpha values, *alpha=1* was chosen to compare the polynomial degree. Degree orders 1 through 5 were fitted to the scikit learn ridge regression linear model to find the optimal degree.

LASSO

The same calculations were made to the dataset with Lasso as with ride to find optimal values for the model. Polynomial Degree and the hyperparameter Alpha were iterated to determine optimal values.

IMPLEMENTATION OF NEURAL NETWORK

For the Neural Network we used the SKLEARN package and imported MLP Regressor. We had many layers to look through, and used a self-made function called best_score to find optimal regularization, and learning parameter. Below is a picture of an output of the code. Full code can be found in our Jupiter notebook.

```
r2Score: 0.2850429512269034 layer: [8, 8] learning_rate: 0.1 alpha: 0.001
MSE: -0.08191308384501 85 layer: [8, 8] learning_rate: 0.1 alpha: 0.001
r2Score: 0.3442334710901562 layer: [8, 8] learning_rate: 0.01 alpha: 0.001
MSE: -0.08458978389543717 layer: [8, 8] learning_rate: 0.01 alpha: 0.001
Highest_r2Score 0.3664053743504557 Best_Layer_r2 [8, 8] Best_Alpha_r2: 0.01 Best_Learning_r2 0.1
Lowest_MSE: -0.08191308384501 85 Best_Layer_MSE: [8, 8] Best_Alpha_MSE: 0.001 Best_Learning_MSE: 0.1
```

FIGURE 10: SHOWS THE OUTPUT OF ONE OF OUR FUNCTIONS.

Then, we used a function called BESTBATCHSIZE(also own code) to pick up the best batch size. Below is also an output from that code.

```
Highest_r2Score 0.38620338791722897 Best_Batch_r2: 200 Best_NumberOfIteration_r2: 100
Lowest_MSE: -0.07915699997621442 Best_Batch_MSE: 100 Best_NumberOfIteration_MSE: 300
```

FIGURE 11: SHOWS OUTPUT FROM FUNCTION BESTBATCHSIZE

Finally we computed the MSE, and R2score while changing the number of iterations from 2, to above 1000. We used the python code shown below, and we used the output data to compute for train, and Test error.

```
def gradient_stochastic_cal(BatchSize,Layers,epochs,OptimizationMethod,Xdata,Ydata):
    #Create empty list for storing R2-score, and MSE for stochastic gradient descent
    r2ScoreSGD = []
    MSEScoreSGD = []
    #Create empty list for storing R2-score, and MSE for gradient descent
    r2ScoreGD = []
    MSEScoreGD = []
    for i in (epochs):
        for optimazor in OptimizationMethod:
            mlp = MLPRegressor(hidden_layer_sizes=[Layers], max_iter=i, alpha =0.0001,learning_rate_init=0.001,early_st
            score = cross_val_score(mlp, Xdata, Ydata.ravel(), cv=5, scoring='r2')
            MSEscore = cross_val_score(mlp, Xdata, Ydata.ravel(), cv=5, scoring='neg_mean_squared_error')
            if optimazor == 'sgd':
                r2score = np.mean(score)
                r2ScoreSGD.append(r2score)
                MSEScore = np.mean(MSEscore)
                MSEScoreSGD.append(MSEScore)
            else:
                r2score = np.mean(score)
                r2ScoreGD.append(r2score)
                MSEScore = np.mean(MSEscore)
                MSEScoreGD.append(MSEScore)
    return(r2ScoreSGD,MSEScoreSGD,r2ScoreGD,MSEScoreGD)
```

FIGURE 12: SHOWS PICTURE OF THE CODE USED TO COMPUTE FOR TEST, AND TRAINING ERROR.

IMPLEMENTATION OF DECISION TREES

For the Decision Tree we did pre-pruning, and not post-pruning. The reason was because the SKLEARN did not include postpruning. Our first part, was to find optimal number of leave nodes, and also the minimum split sample we should use. We used a function we made called DecisionTreeAnalyzor. This can be found easily in the jupyter notebook. The next part, we iterated over different depths, and studied the bias, and variance tradeoff. Below is picture of the code we used to change the depth of out tree:-

```
def DecisionDepth(DataType,depth):
    mseDepth = []
    r2Depth = []
    for i in range(1,depth):
        model = DecisionTreeRegressor(max_depth = i,min_samples_split = 100,max_features=19,max_
        model.fit(XTrain,yTrain)
        if DataType == 'Train':
            score = cross_val_score(model,XTrain,yTrain,cv=5,scoring='r2')
        else:
            score = cross_val_score(model,XTest,yTest,cv=5,scoring='r2')
        r2Score = np.mean(score)
        r2Depth.append(np.mean(score))
        # Also here, we calculate the MSE. Differentiate between training or Test data
        if DataType == 'Train':
            mseScore=cross_val_score(model,XTrain,yTrain,cv=5,scoring='neg_mean_squared_error')
        elif DataType =='Test':
            mseScore=cross_val_score(model,XTest,yTest,cv=5,scoring='neg_mean_squared_error')
        MSE = np.mean(mseScore)
        mseDepth.append(np.mean(mseScore)) # Make a list
    return mseDepth,r2Depth
f = DecisionDepth(('Train'),20)
g = DecisionDepth(('Test'),20)
```

FIGURE 13: DECIONDEPTH FUNCTION IS SHOWN IN PICTURE ABOVE.

IMPELENTATION OF RANDOM FOREST

The model *sklearn.ensemble.RandomForestRegressor* from scikit learn was used to fit our data.The model is specifically designed for fitting regression data. Using the *RandomForestRegressor*, a number of classifying decision trees are fit on different batches of the training data. Means are used to improve the predictive accuracy and control overfitting. Out of the parameters of the model, these were focused on in our analysis: n_estimators,max_depth,max_iterations,OOB_samples and max_

The parameter *n_estimators* is the number of trees to be included in our tree. *Max_depth* is the maximum depth of the tree.

The first analysis that was carried out for Random Forests involved training the data on different number of trees to find the optimal *n_estimators*. The RandomForestRegressor was fit with the default parameters made available by scikit learn. Of all parameters, *n_estimators* was iterated through by the model. Resampling was applied to the model using cross-validation with 5 folds. We then calculated the r2 score and MSE for the training using each of the *n_estimators.* The Cross validation score *neg_mean_squared_error* was used to calculate the MSE. Figure below shows part of this implementation.

```
numberOfTrees = [1,5,10, 50, 100,150,200,250,300]
for num in numberOfTrees:
    rf = RandomForestRegressor(n_estimators=num, random_state=0,)
    rf.fit(X_train, y_train)
    sco.append(rf.score)
    y_pred = rf.predict(X_test)
    score = cross_val_score(rf,X_train,y_train,cv=5,scoring='r2')
    r2.append(np.mean(score))
    mseScore=cross_val_score(rf,X_train,y_train,cv=5,
                             scoring='neg_mean_squared_error')
    mse.append(np.mean(mseScore))
```

FIGURE 14:SHOWS CODE SNIPP USED FOR RANDOM FOREST

The data being used initially had eight features only but with preprocessing and one-hot-encoding, the features were increased to 19. As expected, these features would have different percentages of significance in our quest to predict on the test data. The features were arranged based on their significance that was returned by a *fetaure_importance*, an attribute of the regressor. A higher value entails that the particular feature is more important. Figure 2 below shows a snippet of the code that calculated the feature importance.

```
rf = RandomForestRegressor( n_estimators=100,max_features=19,
                random_state=0)
rf.fit(X_train, y_train)
feature_importances = rf.feature_importances_
std = np.std([tree.feature_importances_ for tree in rf.estimators_],
             axis=0)
indices = np.argsort(feature_importances)[::-1]
# Print the feature ranking
print("Feature ranking:")
for f in range(X_train.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f],
                                   feature_importances[indices[f]]))

y_ticks = np.arange(0, len(feature_importances))
fig, ax = plt.subplots()
ax.barh(y_ticks, feature_importances[indices])
ax.set_yticklabels(indices)
ax.set_yticks(y_ticks)
ax.set_title("Random Forest Feature Importances ")
fig.tight_layout()
plt.show()
```

FIGURE 15:USED TO PLOT FEATURE IMPORTANCE

Two parameters of the estimator, *max_features* and *max_depth* were iterated in a fitting to find their effects on the error and score calculated from the models. *Max_depth* is the maximum depth of the tree while *max_features* is the number to consider when looking for the best split. Figure below indicates the different values that were used for the parameters. For *max_features,* values predefined in scikit learn were iterated. The default value of *Max_depth* was included (None), together with some other integer values.

```
#max features list has alll options available for mac features
max_featureL=['auto','sqrt','log2',19]
max_feat_names=['auto=n_features','sqrt[n_features]',
                'log2[n_features]','n_features']
max_depthL = [None,1,2,3,4,5,6,7,8,9,10]
i = len(max_depthL)
testerror = np.empty((4, i))
trainerror = np.empty((4, i))
mse = np.empty((4, i))
r2 = np.empty((4, i))
for f,feature in enumerate(max_featureL):
    for s,dep in enumerate(max_depthL):
        rf = RandomForestRegressor( n_estimators=100,
                    max_features=feature,
                    max_depth=dep,random_state=0)
        rf.fit(X_train, y_train)
        score = cross_val_score(rf,X_train,y_train,
                                cv=5,scoring='r2')
        r2[f,s] = np.mean(score)
        mseScore=cross_val_score(rf,X_train,y_train,
                        cv=5,scoring='neg_mean_squared_error')
        mse[f,s] = np.mean(mseScore)
```

FIGURE 16:SHOWS THE DEPTH VARIATOR FUNCTION

RESAMPLING TECHNIQUES

To calculate the MSE, and R2 score we have used k-fold cross validation implemented in sklearn package we were importing cross_val. How kfold crossvalidation function we have already discussed that in theory part.

RESULT

LINEAR REGRESSION

OLS

Test and train mean squared errors (MSEs) were plotted for degrees 1 to 5. The figure below shows the plot gotten.

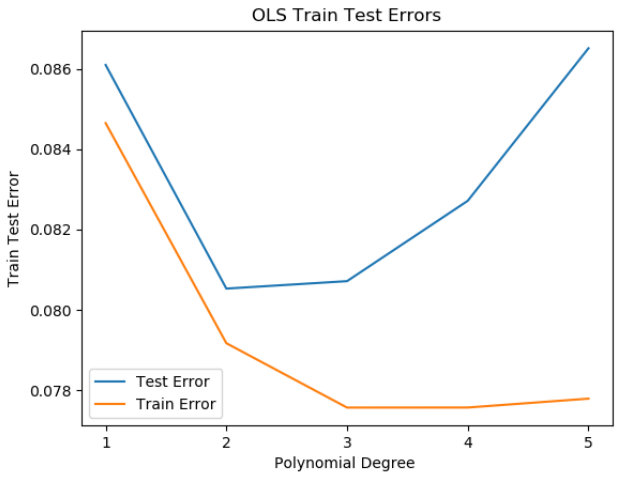


FIGURE 17:SHOWS TRAIN AND TEST ERROR FOR OLS.

The R2score for the Test, and Train data is shown in figure (18) below.

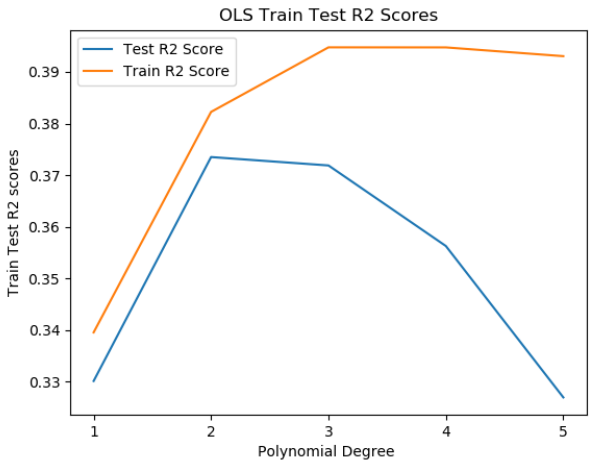


FIGURE 18: R2 SCORE FOR TRAIN, AND TEST DATA.

RIDGE

To find optimal alpha value, we obtained MSE, and R2score for degree 1. Numbers can be seen in table(2) below.

TABLE 2: SHOWS MSE, AND R2SCORE AT DEGREE 1 FOR DIFFERENT REGULARIZATION VALUES.

Alpha	Train MSE	Train R2 Score
0	0.09	0.34
1e-20	0.09	0.34
1e-15	0.09	0.34
1e-10	0.09	0.34
1e-5	0.09	0.34

1e-4	0.09	0.34
1e-3	0.09	0.34
1e-2	0.09	0.34
1e-1	0.09	0.34
10	0.09	0.34
100	0.09	0.33
1000	0.09	0.33
10000	0.1	0.26

From the above we see choosing alpha = 1 looks safe. We then use that value, and iterate over 5 degrees polynomial. We get data shown in table (3) below;

TABLE 3: SHOWS R2SCORE FOR TRAIN, AND TEST DATA OVER DIFFERENT DEGREES.

Degree	MSE		R2SCORE	
	Train	*Test*	*Train*	*Test*
1	0.10	0.10	0.34	0.33
2	0.10	0.10	0.37	0.37
3	0.10	0.10	0.39	0.36
4	0.10	0.10	0.38	0.30
5	0.10	0.10	0.34	-0.57

LASSO

For Lasso, we did as we did for Ridge. First we tried to find best Alpha value. Numbers gotten can be seen in table(4) below.

TABLE 4: SHOWS MSE, AND R2SCORE AT DEGREE 1 FOR DIFFERENT REGULARIZATION VALUES.

Alpha	Train MSE	Train R2 Score
0	0.09	0.336
1e-20	0.09	0.336
1e-15	0.09	0.356
1e-10	0.09	0.356
1e-5	0.09	0.356
1e-4	0.09	0.334
1e-3	0.09	0.31
1e-2	0.1	0.15
1e-1	0.13	0
10	0.13	0
100	0.13	0
1000	0.13	0
10000	0.13	0

Then we iterated over different degrees after choosing 0.00001 as our optimal alpha value. Results gotten for train, and test MSE and R2score can be seen in table (5).

TABLE 5: SHOWS R2SCORE FOR TRAIN, AND TEST DATA OVER DIFFERENT DEGREES.

Degree	MSE		R2SCORE	
	Train	*Test*	*Train*	*Test*
1	0.10	0.10	0.34	0.33
2	0.10	0.10	0.37	0.37
3	0.10	0.10	0.39	0.36
4	0.10	0.10	0.39	0.30

ARTIFICIAL NEURAL NETWORK

For the Neural network we did major analyze using one and two layers. Since the computation time was very long, we only managed only to include; [8,8,8],[32,32,32] and [64,64,64] for the tree layers.

The first thing we did was to test around and find, for each layer which Alpha value, and learning rate give the highest R2-score, and lowest MSE-values. The results can be seen in the table(6), table(7),table(8),table(9),table(10)and table(11) below;

1. One Layer

TABLE 6: SHOWING WHICH REGULARIZATION AND LEARNING PARAMETER EACH LAYER HAD FOR HIGHEST R2-SCORE.

Best R2-score	Layer	Alpha	Learning Rate
0.356	8	1.0	0.01
0.362	16	1.0	0.01
0.372	32	0.01	0.1
0.379	64	0.1	0.01
0.386	128	0.01	0.1
0.383	256	0.1	0.01
0.391	512	0.01	0.1

TABLE 7: SHOWING WHICH REGULARIZATION AND LEARNING PARAMETER EACH LAYER HAD FOR LOWEST MSE.

Best MSE	Layer	Alpha	Learning Rate
0.083	8	1.0	0.01
0.082	16	1.0	0.01
0.082	32	0.01	0.1
0.08	64	0.1	0.01
0.0790	128	0.1	0.01
0.0792	256	0.01	0.1
0.077	512	0.01	0.1

2. Two Layers

TABLE 8: SHOWING WHICH REGULARIZATION AND LEARNING PARAMETER EACH LAYER HAD FOR HIGHEST R2-SCORE.

Best R2-score	Layer	Alpha	Learning Rate
0.356	[8,8]	0.1	0.01
0.38	[32,32]	0.1	0.01
0.386	[64,64]	0.1	0.01
0.386	[64,128]	0.1	0.01
0.39	[128,128]	0.1	0.01

TABLE 9: SHOWING WHICH REGULARIZATION AND LEARNING PARAMETER EACH LAYER HAD FOR LOWEST MSE.

Best MSE	Layer	Alpha	Learning Rate
0.083	[8,8]	0.1	0.01
0.08	[32,32]	0.1	0.01
0.079	[64,64]	0.1	0.01
0.078	[64,128]	0.1	0.01
0.078	[128,128]	0.1	0.01

3. Three Layers

TABLE 10: SHOWING WHICH REGULARIZATION AND LEARNING PARAMETER EACH LAYER HAD FOR HIGHEST R2-SCORE.

4. Best R2-score	Layer	Alpha	Learning Rate
0.37	[8,8,8]	0.1	0.1
0.381	[32,32,32]	0.1	0.01
0.384	[64,64,64]	0.1	0.01

TABLE 11: SHOWING WHICH REGULARIZATION AND LEARNING PARAMETER EACH LAYER HAD FOR LOWEST MSE.

Best MSE	Layer	Alpha	Learning Rate
0.081	[8,8,8]	0.1	0.1
0.079	[64,64,64]	0.1	0.01

To do the computation we used SKLEARN function MLPRegressor, and a code that could pick up the best

alpha, and learning rate. It can be found in our Jupyter Notebook under Neural Network with function name: *def best_score*

Following alpha, and learning rate values were used;

TABLE 12: ALPHA, AND LEARNING RATE VALUES ARE PRESENTED.

Alpha	Learning Rate
10	
1	
0.1	0.1
0.01	0.01
0.001	0.001
0.0001	0.0001
0.00001	0.00001

The next thing we did was to look over all layers presented in the tables above, and pick layers that looked to perform the best,. We chose the following; Layer [512], and Layer [128,128]. The R2Score values they have is 0.391 as shown in table(7), and (9). The MSE is 0.77 and 0.78. The network seems to be slightly better for the one with one hidden layer, but we will explore more.

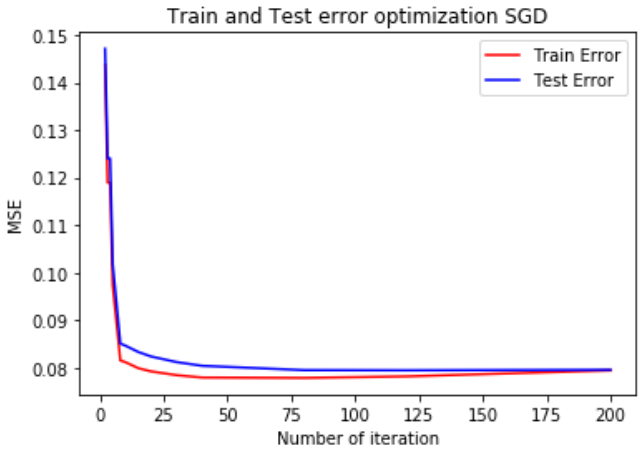
Then we used a self-made function called *BestBatchSGDNumOfIteration* to find optimum batch size. The function can as well be found in our jupyter notebook.

We opted therefor to go for stochastic gradient descent. Running the function *BestBatchSGDNumOfIteration* with alpha values, and learning values gotten from the tables(7) and (9) presented earlier, we got the following optimal batch size:

TABLE 13: SHOWING R2 SCORE WE GOT WITH BEST BATCH SIZE CHOSEN FROM GIVEN BATCH SIZES.

Best R2-score	Layer	Batch-Size	Number of iterations.
0.401	512	300	1300
0.394	128,128	1.0	0.01

Since both Layers look to perform as well, we opt to go for layer 512. We got the



DECISION TREES

As we explained in our implementation, in this part we did pre-pruning as described under theory part, and not post-pruning. We played around with the *maximum number leaf nodes* at every depth, together with minimum number of splits and the depth of the Tree. At first, we let the minimum number of splits be constantly equal 100(Not coincidental), and changed the polynomial Degree and maximum Number og nodes. We got the plots shown in (20) below for training data;

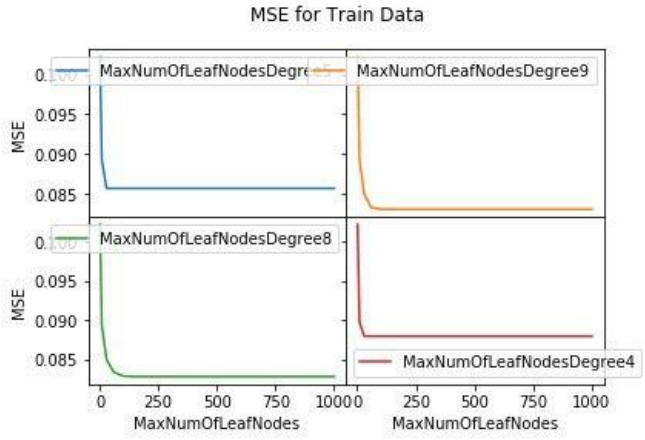


FIGURE 19: SHOWS MSE AS FUNCTION OF MAXIMUM LEAF NODES FOR DIFFERENT DEGREES. THE PLOT WAS GENERATED USING TRAIN DATA.

and (22),

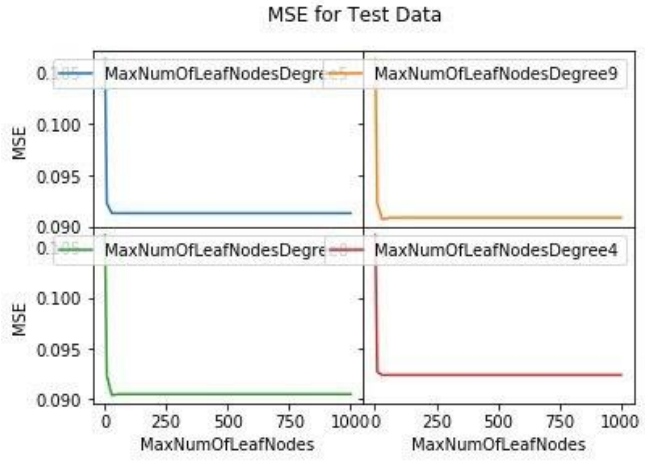


FIGURE 20: SHOWS MSE AS FUNCTION OF MAXIMUM LEAF NODES FOR DIFFERENT DEGREES. THE PLOT WAS GENERATED USING TEST DATA.

for test data. As we see in the plots, the MSE value is high at the start, and falls drastically as the maximum number of leaf nodes are increased. Not very surprising for the training as we would expect that by increasing the number of leaf nodes, we also increase the fitting to the train data. But, the test data does not show much change in the MSE when maximum number of leaf nodes is increased. This may be an indication showing that the maximum number of leaf nodes is not a very sensitive hyper parameter, and does not show clearly the overfitting.

We did the same for *minimum number of samples*. We set the maximum number of leaves equal 100, and as can be seen from figure(23) and (24) it seems to be an okay number. By doing so we got the illustrations (25), and (23) below for the training, and test data respectively.

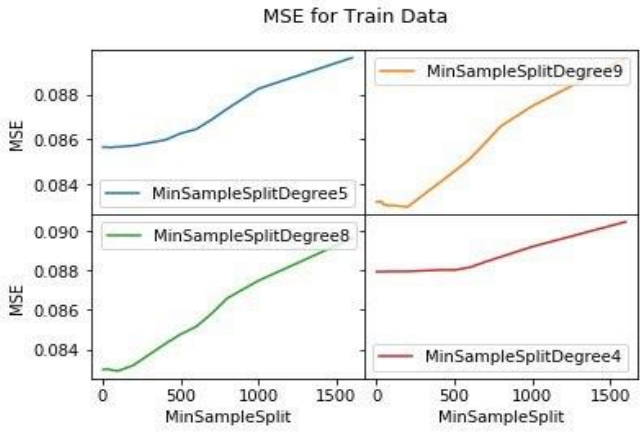


FIGURE 21: SHOWS MSE AS FUNCTION OF MINIMUM NUMBER OF NODES FOR DIFFERENT DEGREES. THE PLOT WAS GENERATED USING TRAIN DATA.

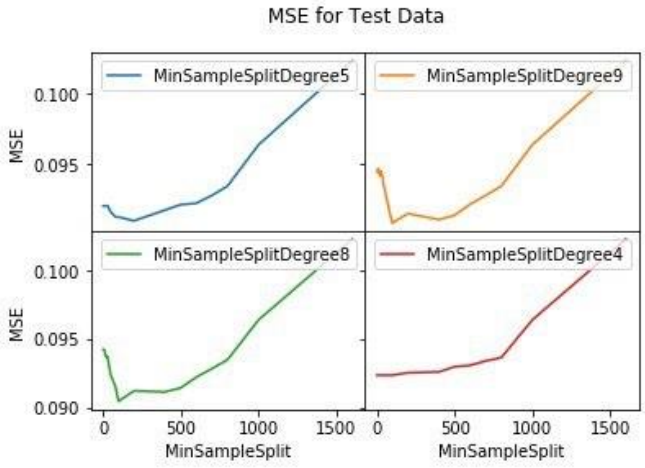


FIGURE 22: SHOWS MSE AS FUNCTION OF MINIMUM NUMBER OF NODES FOR DIFFERENT DEGREES. THE PLOT WAS GENERATED USING TEST DATA.

The plots above are telling us that as we increase the *minimum number of samples* the model underfits more and more. But, the test data is showing a dip indicating there is an optimal number of samples. In our case we have chosen to use 100, and looking at the plots above it seems to be a reasonable number generally.

The next thing we did, was to look how the MSE, and R2 score changed when we changed the depth of Tree. Our analyse showed us that the depth of the tree is the most sensitive hyper parameter, and we thus used it to check out over, and underfitting.

Figure(23) shows the train, and test error. As we can see from the plot, it seems like after depth = 6 the test error start to increase. Thus, a depth = 6 gives the best choice.

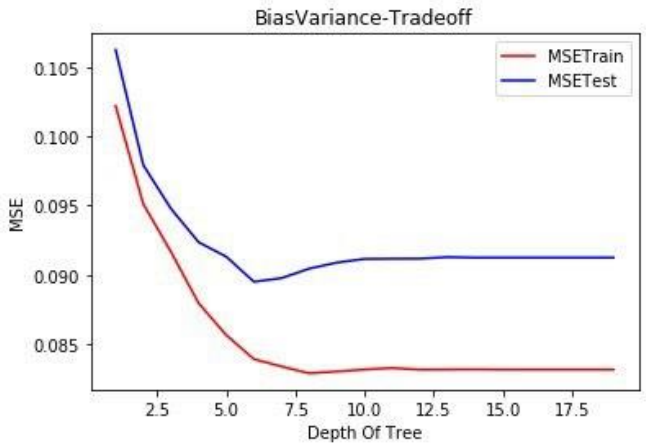


FIGURE 23:SHOWS THE TRAIN AND TEST ERROR AS COMPLEXITY INCREASES.

Also, for the R2 score in figure)24) for the train, and test data shows that indeed a depth = 5 is optimal.

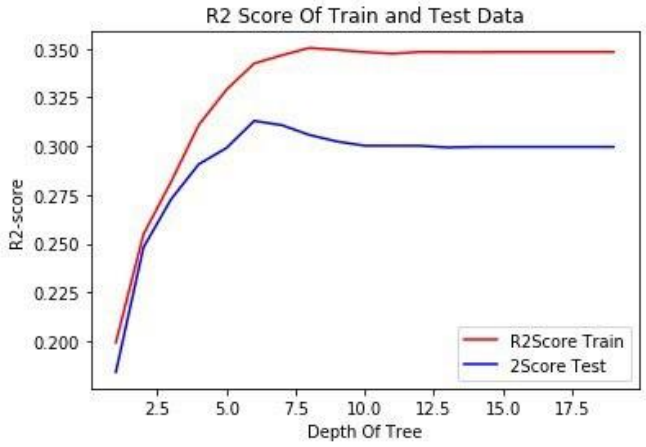


FIGURE 24:SHOWS THE R2-SCORE FOR THE TRAIN AND TEST DATA.

Noting down the gotten values, we have the following MSE, and R2-score for a decision tree with depth = 6, minimum number of samples before split = 100 and a maximum number of nodes = 100;

TABLE 14: SHOWS BEST MSE, AND R2 SCORE GOTTEN FOR DECISION TREES.

	MSE	R2SCORE	
TEST	TRAIN	TEST	TRAIN
0.09	0.08	0.32	0.35

The last thing we looked at was the feature importance. We did this with help of a SKLEARN function called feature_importance, and plotted the below graph. Here the features the wind speed(corresponding the third predictor) is the most important, followed by the humidity. Then comes the temperature. We have many columns, and that is due to the onehot encoding we did to the data set.

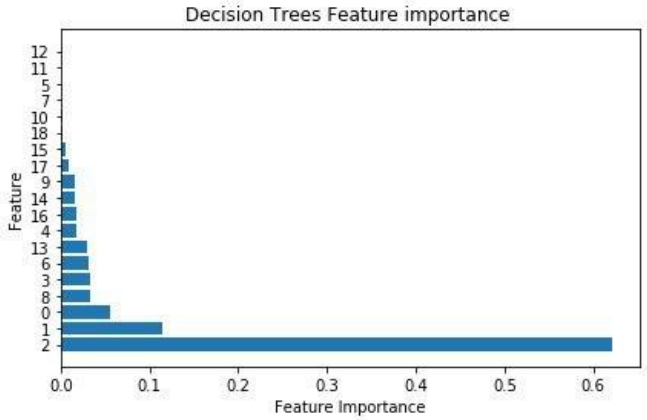


FIGURE 25:SHOWS THE FEATUREIMPORTANCE OF THE CYCLE DATA SET.

RANDOM FOREST

The plot in figure(26) below shows r2 scores and MSE for different n_estimators. Since the MSE was calculated using cross validation's predefined *neg_mean_squared_error,* the values obtained were negative of MSE. For the other models we were able to fix this problem, but not for random ForestThe best negative MSE calculated was 0.168 for 300 number of trees. The model trained with 300 trees also had the best r2 score at 0.3635.

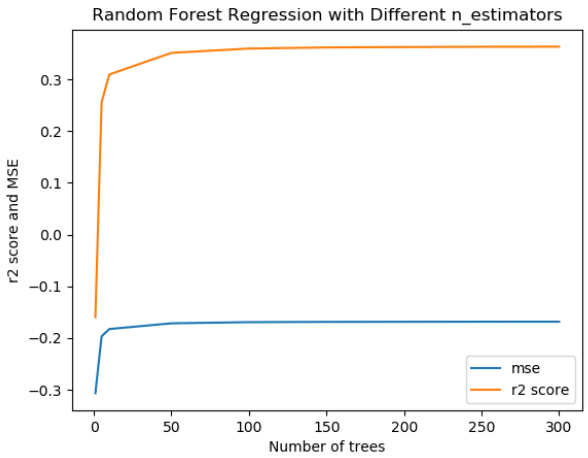


FIGURE 26:R2 SCORE AND MSE.

The calculation of feature importance yielded the plot in figure below. The features included were transformed through one-hot-encoding for the categorical predictors and scaling for the continuous predictors. Feature 2 had the highest importance followed by feature 3 then 1. Feature 5 had the lowest

score, indicating that it did not play an essential role in our prediction.

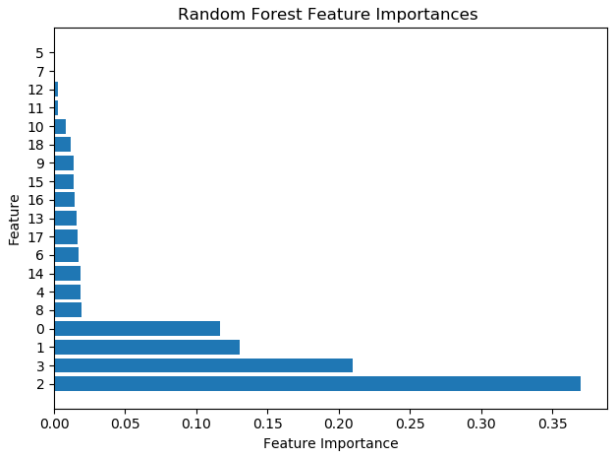


FIGURE 27: SHOWS THE FEATUREIMPORTANCE OF THE CYCLE DATA SET.

DISCUSSION

LINEAR REGRESSION

Among all linear regression models, OLS model had the best performance with regards to time taken to execute. For OLS, polynomial degree 2 had the best combination of R2 scores equal 0.38(figure(24)) and MSE = 0.8(figure(23)) for train data. That of test data R2score and MSE was respectively 0.37, and 0.9 . After polynomial degree of order 3, the test error blows up and the test MSE remains very low(figure(23) shows this). This is as a result of the overfitting caused by increasing the polynomial degree. Degree 3 has the lowest train MSE = 0.76 but since the test MSE is notably higher, considering degree 3 as the best for our data would be slightly overfitting our dataset.

Studying the trends demonstrated when fitting the ridge model, it was observed that the MSE remained constant whether it was the alpha values or the polynomial degree being iterated. It constantly stayed equal 0.1, and MSE value did not play a decisive role in choosing optimal parameters for fitting our model. The best alpha values were the ones close to zero. However, the difference in our case was minimal or not available for alpha values in the range *(1e-20,10(table(2)).* Getting very small alpha values also had the effect of increasing the execution time. For these reasons, *alpha=1* was chosen as the optimal value. Degree 3 had the best train R2 score of 0.39 while degree 2 had the best test R2 score at 0.37 as shown in table(3). But, since our goal is to choose best combination of R2score, for Train and Test data we think degree 2 with R2score values equal 0.37 for both Test, and Train data is the best.

For Lasso regression, bigger alpha values led to negative R2 scores. As it was the case with Ridge, the train and test MSE values for different polynomial degrees are the same, only the R2 score shows variation with respect to polynomial degree. The best alpha values were those closer to zero. The optimal alpha value was recognized as *alpha=1e-5* as shown in table(4). Lower alpha values had better MSE and R2 score values but the difference was minimal. For *alpha = 0.00001,* degree 2 was selected as the optimal polynomial degree. Polynomial 2 had a low MSE of 0.08520218 and a high R2 score of 0.33434.

RANDOM FORESTS

Even though 300 had the best scores, it took longer for the fitting to be done. As the number of trees got higher than 100, here was gradual change in the scores and this change got smaller and smaller. This led to the decision that for our regressor, the optimal value to use for the n_estimators parameter would be 100. This is also the default value for the parameter that was defined in the latest version of Scikit learn.

Features 3 and 2, followed by 1 then 0 are the most important features. Our model relies on these features to come up with the best prediction. If any of these features was to be left out in training, it would impact on the accuracy and prediction of our model. The features with the lowest importance may be left out from training without having any ill effects on the training. Feature_importance attribute has the drawback that the importances can be high even for features that are not predictive of the target variable since it is computed based on statistics derived from the training set.

REFERENCE

https://www.quora.com/What-is-the-differences-between-artificial-neural-network-computer-science-and-biological-neural-network [1]

https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/ [2]

http://deeplearning.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/ [3]

https://compphysics.github.io/MachineLearning/doc/pub/Regression/html/Regression.html [5]