MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION

FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION

NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

«MISIS»

Informatics and Computer Technologies

Major code 09.04.01

Coursework

In Artificial neural network in data science

Feedforward ANN for predicting linearly dependent data

Student: Asmerom Fessehaye

Group: МИВТ-19-7-12A

Supervisor: Prof. Igor Temkin

Moscow

2020

CONTENTS

Multiple Linear Regression (MLR)

Multiple Linear Regression is one of the important regression algorithms which models the linear relationship between a single dependent continuous variable and more than one independent variable.

The multiple linear regression equation is as follows:

$$\hat{Y} = b_0 + b_1 X_1 + b_2 X_2 + \dots + b_p X_p$$

where $\hat{Y}$ is the predicted or expected value of the dependent variable, X_1 through X_p are p distinct independent or predictor variables, b_0 is the value of Y when all of the independent variables (X_1 through X_p) are equal to zero, and b_1 through b_p are the estimated regression coefficients. Each regression coefficient represents the change in Y relative to a one unit change in the respective independent variable. In the multiple regression situation, b_1, for example, is the change in Y relative to a one unit change in X_1, holding all other independent variables constant (i.e., when the remaining independent variables are held at the same value or are fixed).

Problem Description: We have a dataset of **dump trucks**. This dataset contains 22 parameters:

control id - system ID of the agent;
Vehicle_type - agent type (dump truck or excavator);
vehicle_id - ID of the physical agent (tail number of the dump truck);
lat - GPS latitude value at a time;
lon - GPS longitude value at a time;
height - GPS altitude value at a time;
Course - the value of direction (azimuth) from GPS;
speed - speed of movement;
X - latitude value in an unknown coordinate grid;
Y - longitude value in an unknown coordinate grid;
distance - distance covered by the dump truck since the previous time;
fuel - amount of fuel in the tank;
weight - weight of cargo in the body at a time;
num_satellite - number of visible satellites at a time;
gps_quality - quality (accuracy) of GPS coordinate values;
sender - unknown value;
link_quality - quality (accuracy?) communication;
mes_number - empty unknown value;
type - unknown value;
received - unknown value;
fix_status - number of status changes.

Our goal is to create a model that can easily forecast value of the dump **truck speed** in response to a vector of known or interesting parameters (coordinates, direction of movement, weight of the load), which, as we think, have a linear relationship.

Multiple Linear Regression model using Python(Keras)

The Keras library is a high-level API for building deep learning models that has gained favor for its ease of use and simplicity facilitating fast development. Here we will use the Keras library to build regression model.

Regression is a type of supervised machine learning algorithm used to predict a continuous label. The goal is to produce a model that represents the 'best fit' to some observed data, according to an evaluation criterion. The basic architecture of the deep learning neural network, which we will be following, consists of three main components.

1. Input Layer: This is where the training observations are fed. The number of predictor variables is also specified here through the neurons.
2. Hidden Layers: These are the intermediate layers between the input and output layers. The deep neural network learns about the relationships involved in data in this component.
3. Output Layer: This is the layer where the final output is extracted from what's happening in the previous two layers. In case of regression problems, the output later will have one neuron

Steps of implimentation

Step 1 - Loading the required libraries and modules.

Step 2 –Loading Data and performing pre-processing (outlier removal)

Step 3 –Performing feature extraction.

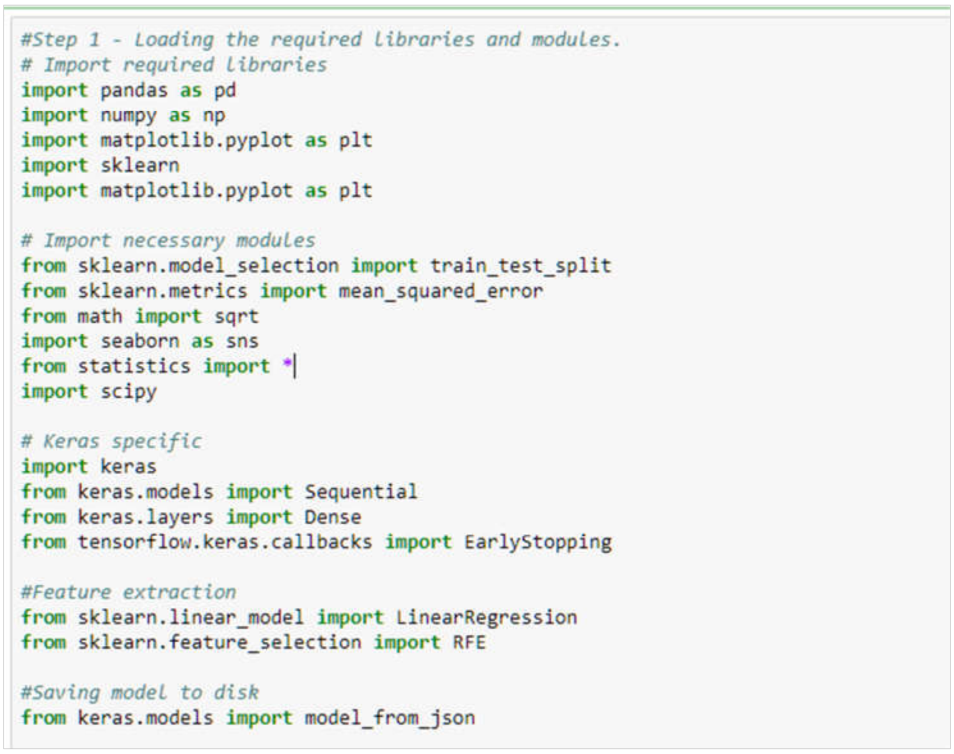
Step 4 - Creating arrays for the features and the response variable.

Step 5 - Creating the training and test datasets.

Step 6 - Define, compile, and fit the Keras regression model.

Step 7- Predict on the test data and compute evaluation metrics.

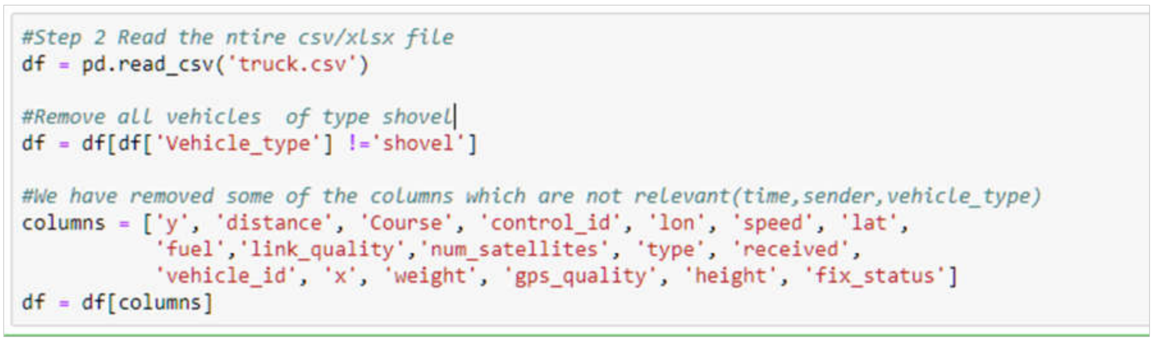
Step 1 - Loading the required libraries and modules.



```
#Step 1 - Loading the required libraries and modules.
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import matplotlib.pyplot as plt

# Import necessary modules
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from math import sqrt
import seaborn as sns
from statistics import *
import scipy

# Keras specific
import keras
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping

#Feature extraction
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFE

#Saving model to disk
from keras.models import model_from_json
```

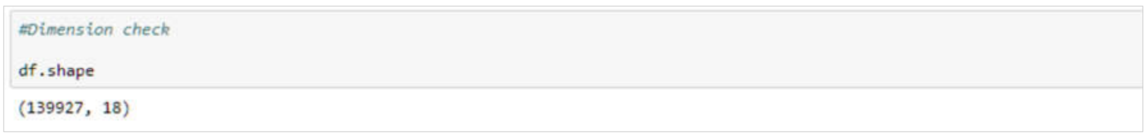
Step 2 –Loading Data and performing pre-processing (outlier removal)

At the initial stage we remove the parameters which are not relevant to us, like vehicle_type(truc k),time ,sender(MessageDetector) and mes_number(which has no value). Then we will use diffe rent methods of feature extraction in **step 3**



```
#Step 2 Read the ntire csv/xlsx file
df = pd.read_csv('truck.csv')

#Remove all vehicles  of type shovel
df = df[df['Vehicle_type'] !='shovel']

#We have removed some of the columns which are not relevant(time,sender,vehicle_type)
columns = ['y', 'distance', 'Course', 'control_id', 'lon', 'speed', 'lat',
           'fuel','link_quality','num_satellites', 'type', 'received',
           'vehicle_id', 'x', 'weight', 'gps_quality', 'height', 'fix_status']
df = df[columns]
```

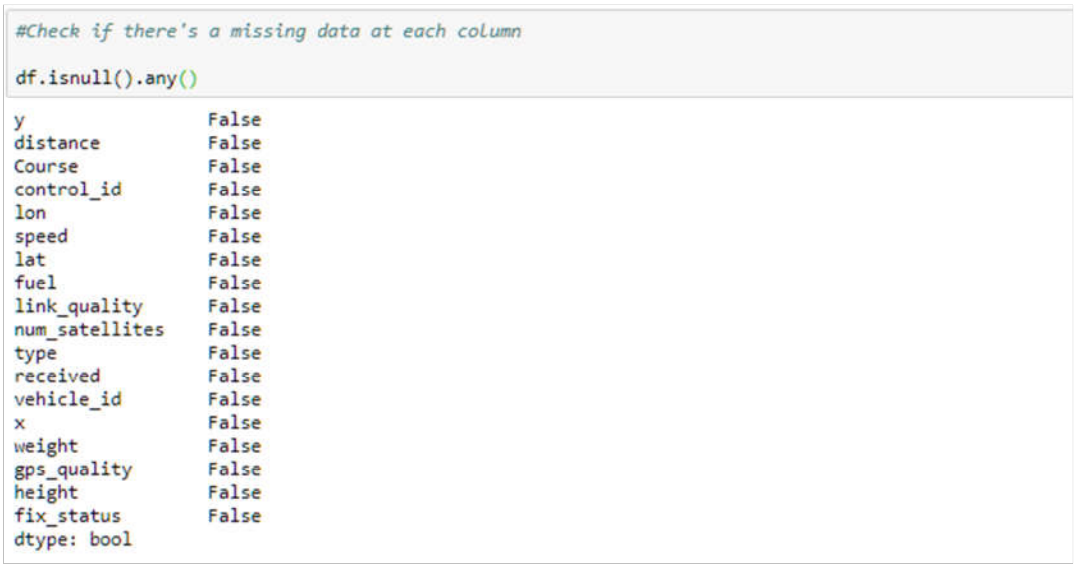
Dimension Check



```
#Dimension check

df.shape
```

```
(139927, 18)
```

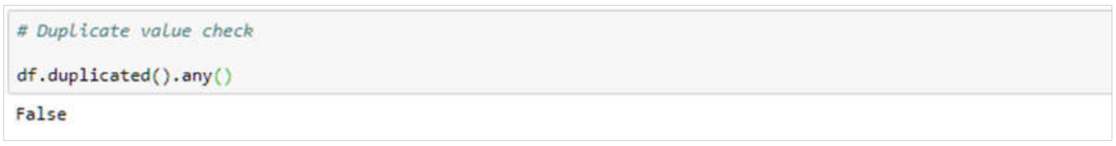
Check for missing data



```
#Check if there's a missing data at each column

df.isnull().any()
```

```
y                  False
distance           False
Course             False
control_id         False
lon                False
speed              False
lat                False
fuel               False
link_quality       False
num_satellites     False
type               False
received           False
vehicle_id         False
x                  False
weight             False
gps_quality        False
height             False
fix_status         False
dtype: bool
```

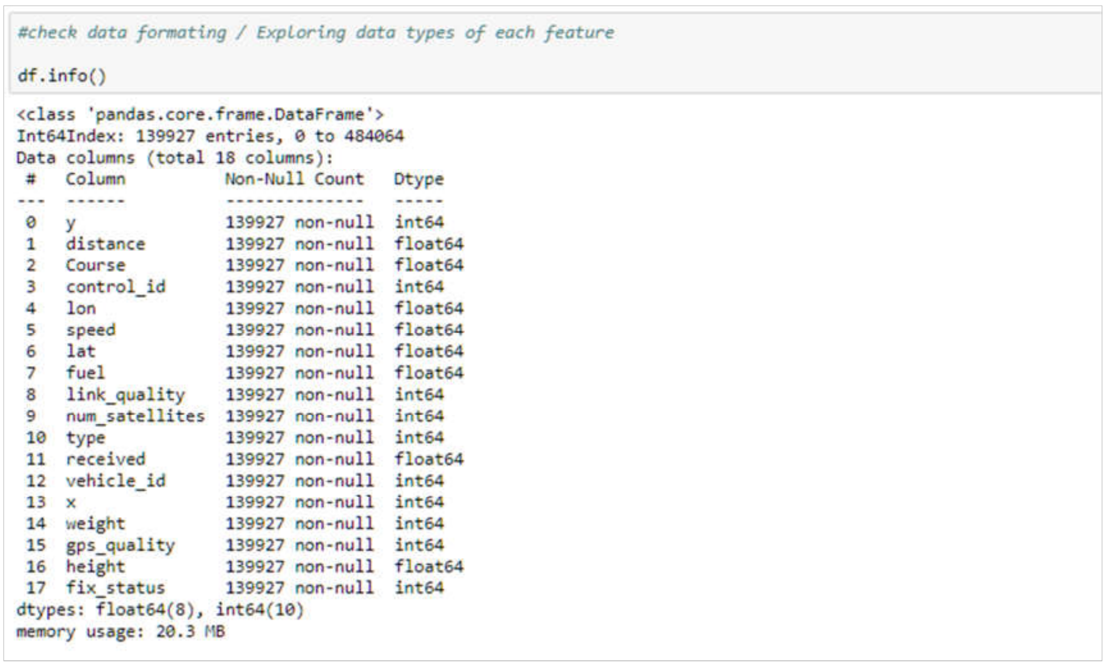
Duplicate Check



```
# Duplicate value check

df.duplicated().any()
```

```
False
```

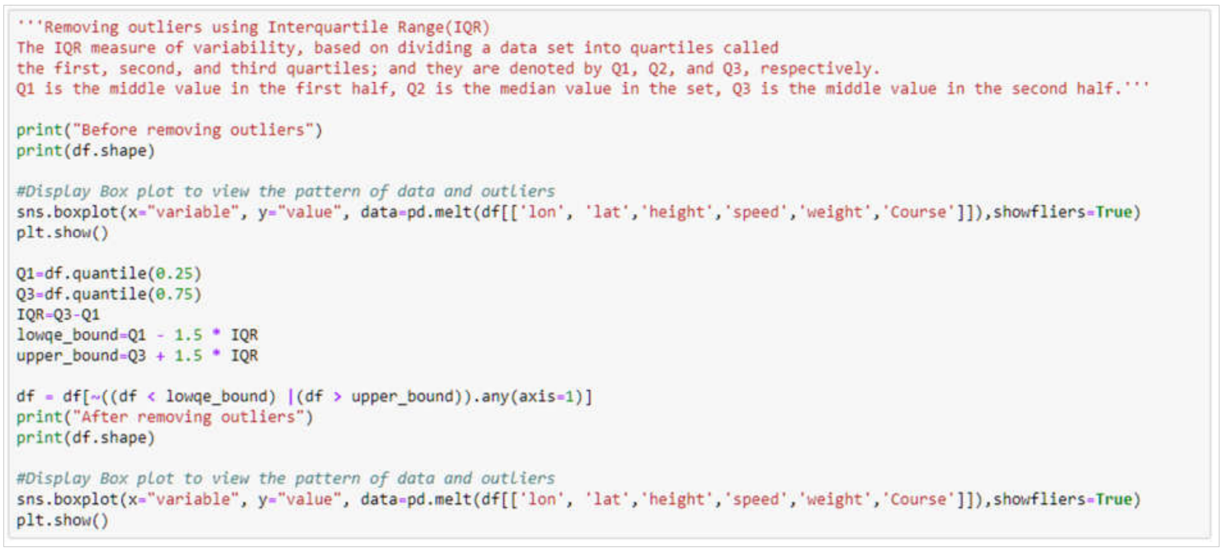
Check Data Formatting



```
#check data formating / Exploring data types of each feature

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 139927 entries, 0 to 484064
Data columns (total 18 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   y               139927 non-null  int64
 1   distance        139927 non-null  float64
 2   Course          139927 non-null  float64
 3   control_id      139927 non-null  int64
 4   lon             139927 non-null  float64
 5   speed           139927 non-null  float64
 6   lat             139927 non-null  float64
 7   fuel            139927 non-null  float64
 8   link_quality    139927 non-null  int64
 9   num_satellites  139927 non-null  int64
 10  type            139927 non-null  int64
 11  received        139927 non-null  float64
 12  vehicle_id      139927 non-null  int64
 13  x               139927 non-null  int64
 14  weight          139927 non-null  int64
 15  gps_quality     139927 non-null  int64
 16  height          139927 non-null  float64
 17  fix_status      139927 non-null  int64
dtypes: float64(8), int64(10)
memory usage: 20.3 MB
```

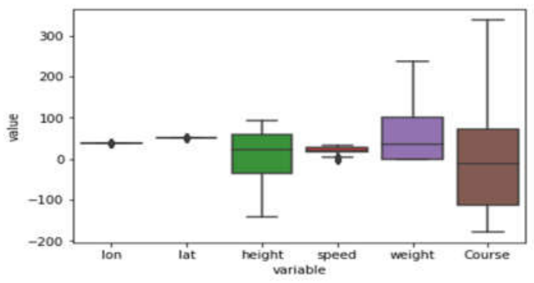
For removing outlier we use Interquartile Range(IQR) method. The rule of thumb is that anythin g not in the range of *(Q1 - 1.5* IQR) and (Q3 + 1.5 *IQR)* is an outlier, and can be removed.



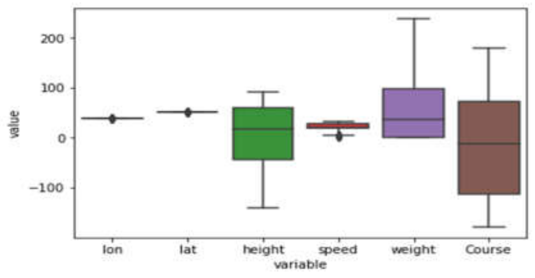
```
'''Removing outliers using Interquartile Range(IQR)
The IQR measure of variability, based on dividing a data set into quartiles called
the first, second, and third quartiles; and they are denoted by Q1, Q2, and Q3, respectively.
Q1 is the middle value in the first half, Q2 is the median value in the set, Q3 is the middle value in the second half.'''

print("Before removing outliers")
print(df.shape)

#Display Box plot to view the pattern of data and outliers
sns.boxplot(x="variable", y="value", data=pd.melt(df[['lon', 'lat','height','speed','weight','Course']]),showfliers=True)
plt.show()

Q1=df.quantile(0.25)
Q3=df.quantile(0.75)
IQR=Q3-Q1
lowqe_bound=Q1 - 1.5 * IQR
upper_bound=Q3 + 1.5 * IQR

df = df[~((df < lowqe_bound) |(df > upper_bound)).any(axis=1)]
print("After removing outliers")
print(df.shape)

#Display Box plot to view the pattern of data and outliers
sns.boxplot(x="variable", y="value", data=pd.melt(df[['lon', 'lat','height','speed','weight','Course']]),showfliers=True)
plt.show()
```

Here since we have many columns I just displayed the box plot of the columns of interest to have clear boxplot. We could have also displayed all the columns but the box plot will not be clear because of not enough space to show all columns clearly.

```
Before removing outliers
(139927, 18)
```



```
After removing outliers
(107600, 18)
```

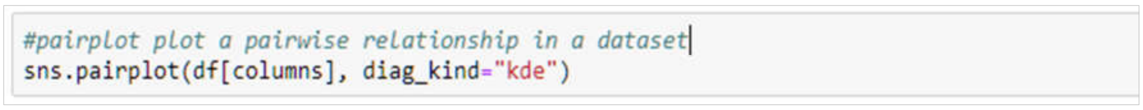


So as we can see from the box plots 32327 rows(139927-107600) with outliers are removed.

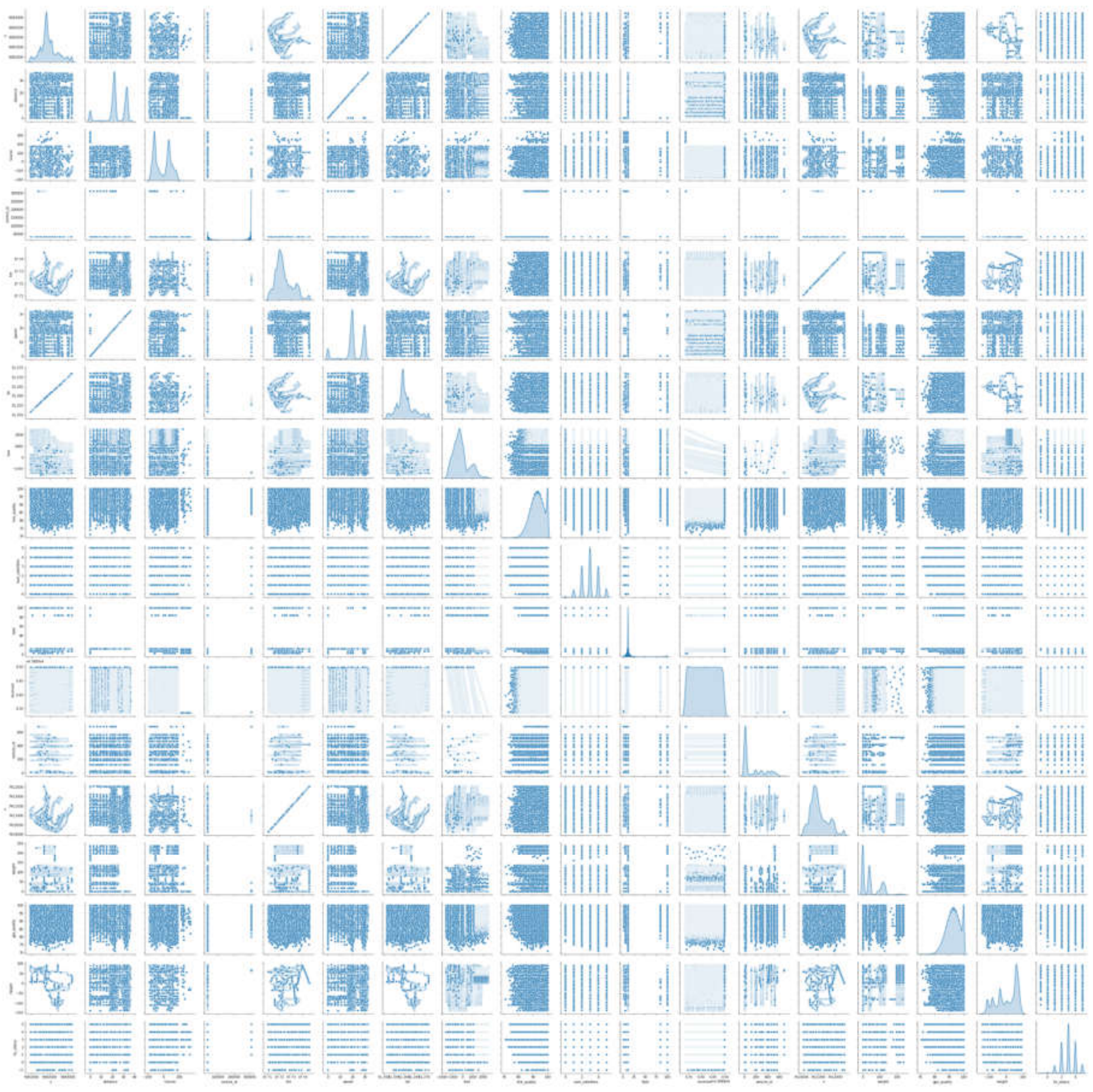
Step 3 –Performing feature extraction.

Pairplot plot a pairwise relationship in a dataset

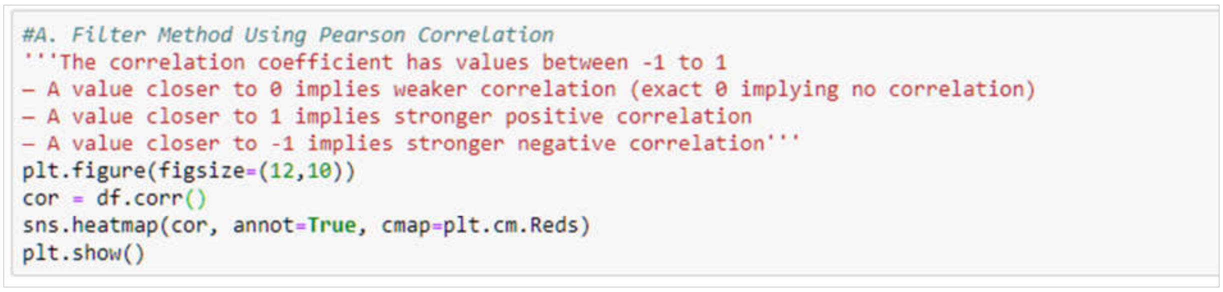
A pairplot plot a pairwise relationship in a dataset. The pairplot function creates a grid of Axes such that each variable in data will by shared in the y-axis across a single row and in the x-axis across a single column. That creates plots as shown below.



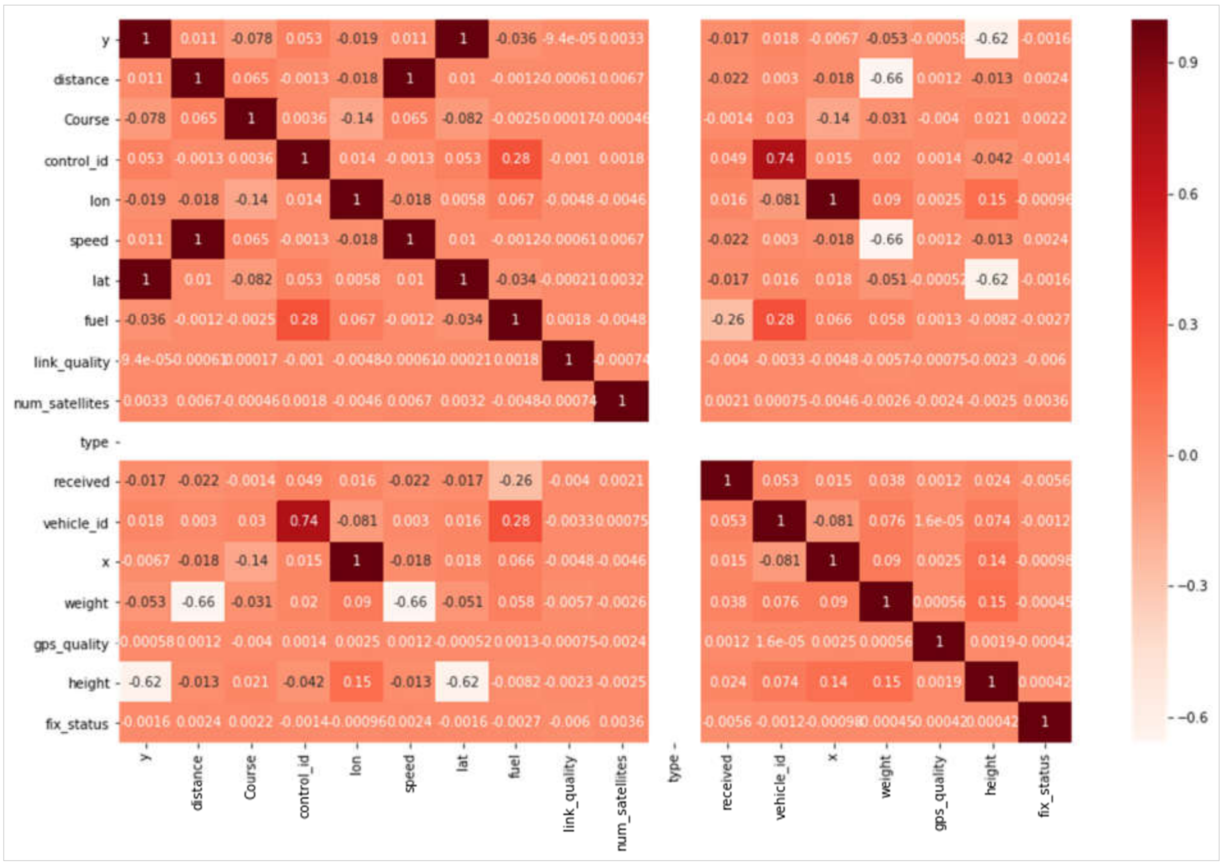
```
#pairplot plot a pairwise relationship in a dataset
sns.pairplot(df[columns], diag_kind="kde")
```



Filter Method Using Pearson Correlation



```
#A. Filter Method Using Pearson Correlation
'''The correlation coefficient has values between -1 to 1
- A value closer to 0 implies weaker correlation (exact 0 implying no correlation)
- A value closer to 1 implies stronger positive correlation
- A value closer to -1 implies stronger negative correlation'''
plt.figure(figsize=(12,10))
cor = df.corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.Reds)
plt.show()
```

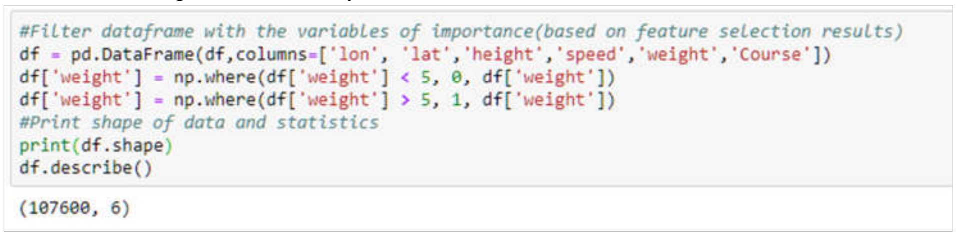


Based on the above, we can conclude that to solve the problem of predicting the speed of a dump truck at a given time on a given data set, it is necessary to choose the most informative parameters characterized by a linear relationship. Thus, the parameters **lat**, **long**, **height**, **Course**, **speed**, **weight** are of the greatest practical interest.

And the most important thing is that we need to concentrate with features which are related to speed.

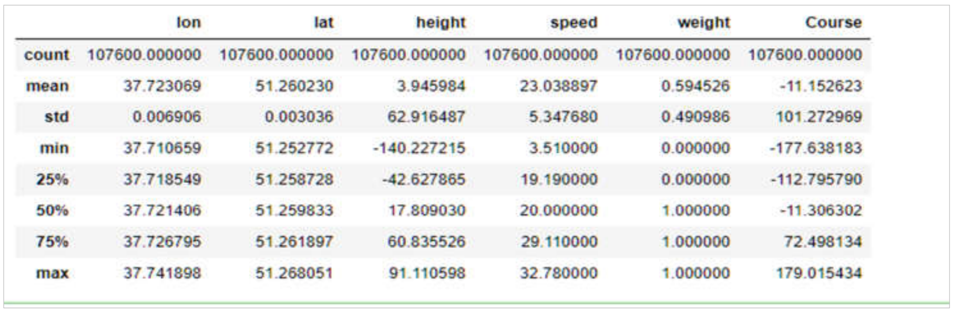
Step 2 – Create a data frame of selected features and perform Basic Data Checks

The first line of code reads in the data as pandas dataframe, the second and third line changes the weight data in to two categories(Loaded=1 where weight<5 and Unloaded=0 where weight>5), while the fourth line of code prints the shape - 107600 observations of 6 variables. The fifth line gives summary statistics of the numerical variables.



```
#Filter dataframe with the variables of importance(based on feature selection results)
df = pd.DataFrame(df,columns=['lon', 'lat','height','speed','weight','Course'])
df['weight'] = np.where(df['weight'] < 5, 0, df['weight'])
df['weight'] = np.where(df['weight'] > 5, 1, df['weight'])
#Print shape of data and statistics
print(df.shape)
df.describe()
```

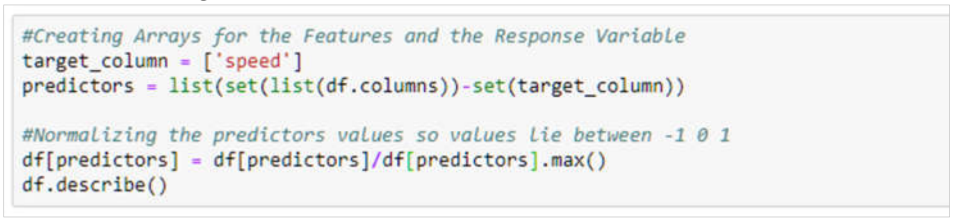
```
(107600, 6)
```



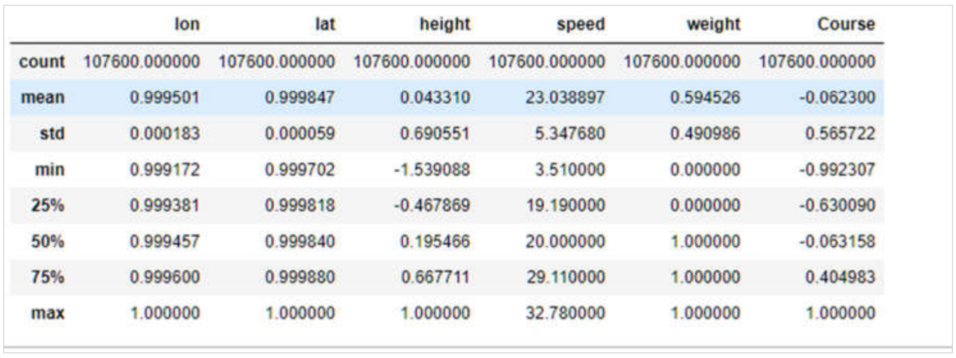
	lon	lat	height	speed	weight	Course
count	107600.000000	107600.000000	107600.000000	107600.000000	107600.000000	107600.000000
mean	37.723069	51.260230	3.945984	23.038897	0.594526	-11.152623
std	0.006906	0.003036	62.916487	5.347680	0.490986	101.272969
min	37.710659	51.252772	-140.227215	3.510000	0.000000	-177.638183
25%	37.718549	51.258728	-42.627865	19.190000	0.000000	-112.795790
50%	37.721406	51.259833	17.809030	20.000000	1.000000	-11.306302
75%	37.726795	51.261897	60.835526	29.110000	1.000000	72.498134
max	37.741898	51.268051	91.110598	32.780000	1.000000	179.015434

Step 4 - Creating Arrays for the Features and the Response Variable

The first line of code creates an object of the target variable, while the second line of code gives us the list of all the features, excluding the target variable speed. The third line normalizes the predictors. This is important because the units of the variables differ significantly and may influence the modeling process. To prevent this, we will do normalization via scaling of the predictors between 0 and 1.The fourth line displays the summary of the normalized data. We can see that all the independent variables have now been scaled between 0 and 1. The target variable remains unchanged.



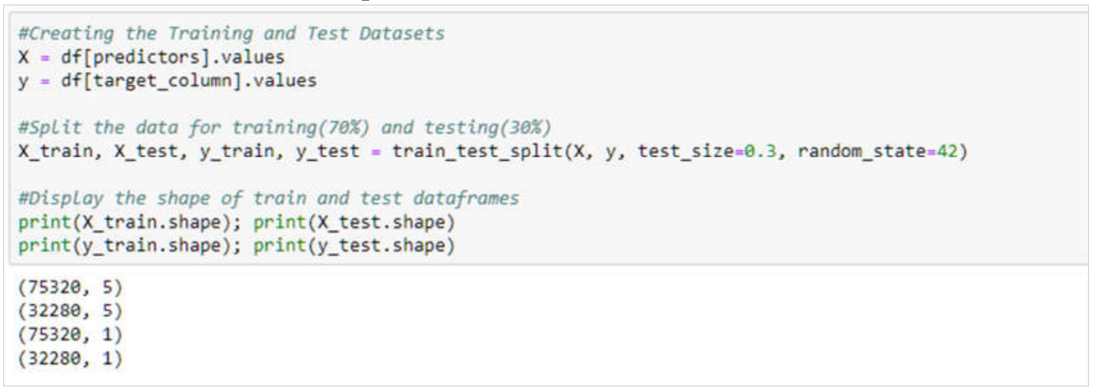
```
#Creating Arrays for the Features and the Response Variable
target_column = ['speed']
predictors = list(set(list(df.columns))-set(target_column))

#Normalizing the predictors values so values lie between -1 0 1
df[predictors] = df[predictors]/df[predictors].max()
df.describe()
```



	lon	lat	height	speed	weight	Course
count	107600.000000	107600.000000	107600.000000	107600.000000	107600.000000	107600.000000
mean	0.999501	0.999847	0.043310	23.038897	0.594526	-0.062300
std	0.000183	0.000059	0.690551	5.347680	0.490986	0.565722
min	0.999172	0.999702	-1.539088	3.510000	0.000000	-0.992307
25%	0.999381	0.999818	-0.467869	19.190000	0.000000	-0.630090
50%	0.999457	0.999840	0.195466	20.000000	1.000000	-0.063158
75%	0.999600	0.999880	0.667711	29.110000	1.000000	0.404983
max	1.000000	1.000000	1.000000	32.780000	1.000000	1.000000

Step 5 - Creating the Training and Test Datasets

The first couple of lines create arrays of independent (X) and dependent (y) variables, respectively. The third line splits the data into training and test dataset, while the fourth line prints the shape of the new data. Here 70% of the data is for training and the rest is for testing. Further the test data will be split to 15% for validation dataset.



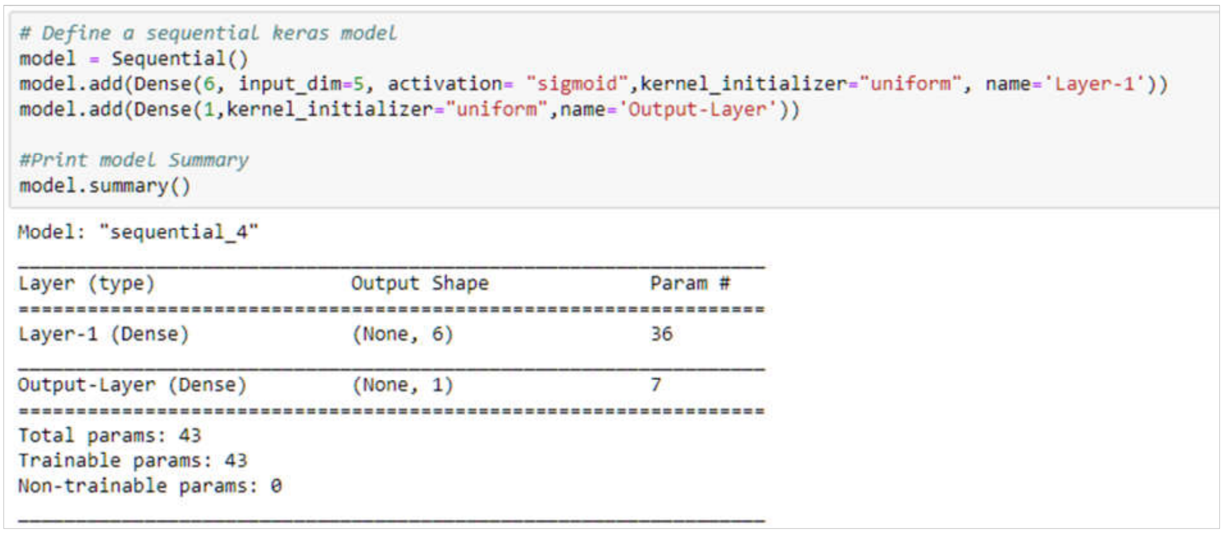
```
#Creating the Training and Test Datasets
X = df[predictors].values
y = df[target_column].values

#Split the data for training(70%) and testing(30%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

#Display the shape of train and test dataframes
print(X_train.shape); print(X_test.shape)
print(y_train.shape); print(y_test.shape)
```

```
(75320, 5)
(32280, 5)
(75320, 1)
(32280, 1)
```

Step 6 - Building the Deep Learning Regression Model

We will build a regression model using deep learning in Keras. To begin with, we will define the model. The first line of code below calls for the Sequential constructor. Note that we would be using the Sequential model because our network consists of a linear stack of layers. The second line of code represents the first layer which specifies the activation function and the number of input dimensions, which in our case is 5 predictors. Here we have 6 number of neurons which is number of features (5) plus 1.

The activation function used in the hidden layers is sigmoid. The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

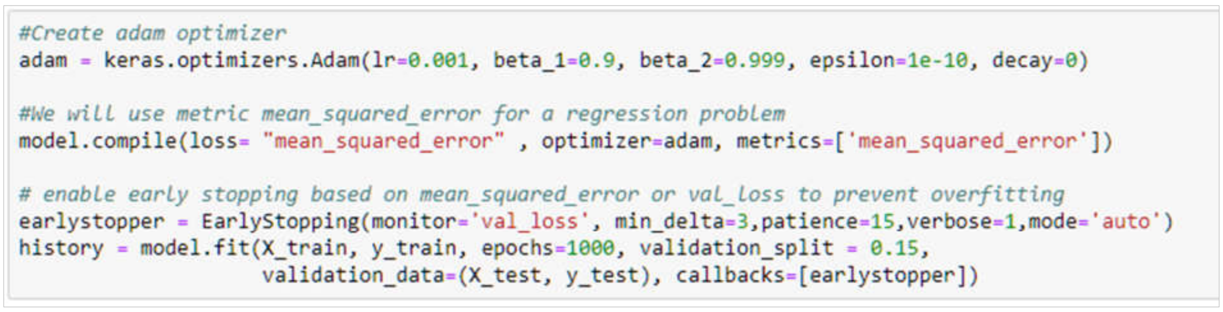


```
# Define a sequential keras model
model = Sequential()
model.add(Dense(6, input_dim=5, activation= "sigmoid",kernel_initializer="uniform", name='Layer-1'))
model.add(Dense(1,kernel_initializer="uniform",name='Output-Layer'))

#Print model Summary
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Layer-1 (Dense)              (None, 6)                 36
_________________________________________________________________
Output-Layer (Dense)         (None, 1)                 7
=================================================================
Total params: 43
Trainable params: 43
Non-trainable params: 0
_________________________________________________________________
```

The next step is to define an optimizer and the loss measure for training. The mean squared error is our loss measure and the "adam" optimizer is our minimization algorithm. And also we are using mean squared error as a metric for training.

The third line of code fits the model on the training dataset. We also provide the argument, epochs, which represents the number of training iterations. We have taken 1000 epochs.



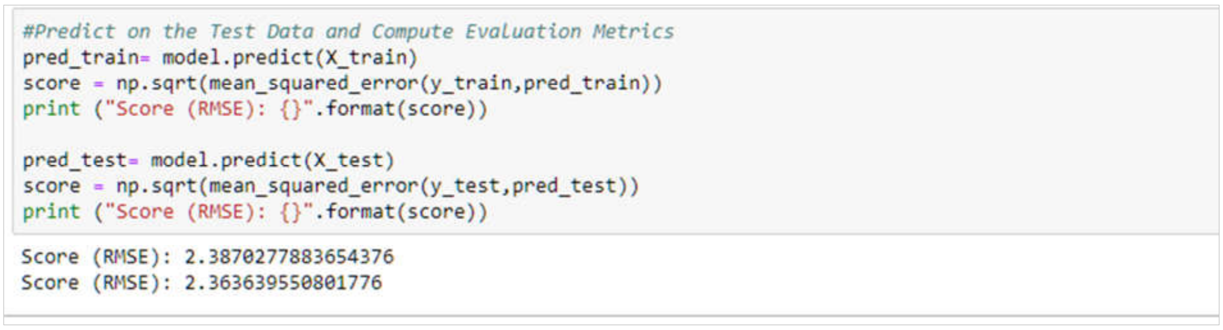
```
#Create adam optimizer
adam = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-10, decay=0)

#We will use metric mean_squared_error for a regression problem
model.compile(loss= "mean_squared_error" , optimizer=adam, metrics=['mean_squared_error'])

# enable early stopping based on mean_squared_error or val_loss to prevent overfitting
earlystopper = EarlyStopping(monitor='val_loss', min_delta=3,patience=15,verbose=1,mode='auto')
history = model.fit(X_train, y_train, epochs=1000, validation_split = 0.15,
                    validation_data=(X_test, y_test), callbacks=[earlystopper])
```

A problem with training neural networks is in the choice of the number of training epochs to use. Too many epochs can lead to **overfitting** of the training dataset, whereas too few may result in an **underfit** model. To avoid this situation we use here Early stooping mechanism.

Early stopping is a method that allows you to specify an arbitrarily large number of training epochs and stop training once the model performance stops improving on the validation dataset.

Step 7 - Predict on the Test Data and Compute Evaluation Metrics

The first line of code predicts on the train data, while the second line prints the RMSE value on the train data. The same is repeated in the third and fourth lines of code which predicts and prints the RMSE value on test data.



```
#Predict on the Test Data and Compute Evaluation Metrics
pred_train= model.predict(X_train)
score = np.sqrt(mean_squared_error(y_train,pred_train))
print ("Score (RMSE): {}".format(score))

pred_test= model.predict(X_test)
score = np.sqrt(mean_squared_error(y_test,pred_test))
print ("Score (RMSE): {}".format(score))
```

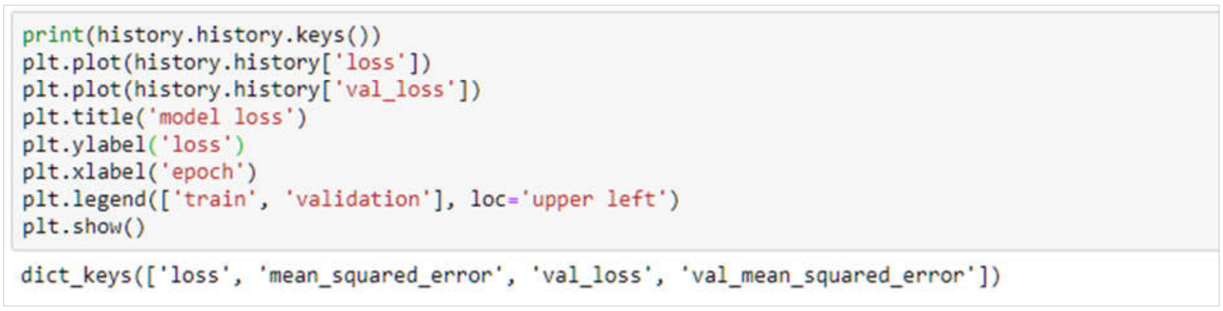
```
Score (RMSE): 2.3870277883654376
Score (RMSE): 2.363639550801776
```

RMSE. The most commonly used metric for regression tasks is RMSE (root-mean-square error). This is defined as the square root of the average squared distance between the actual score and the predicted score.

Evaluation of the Model Performance

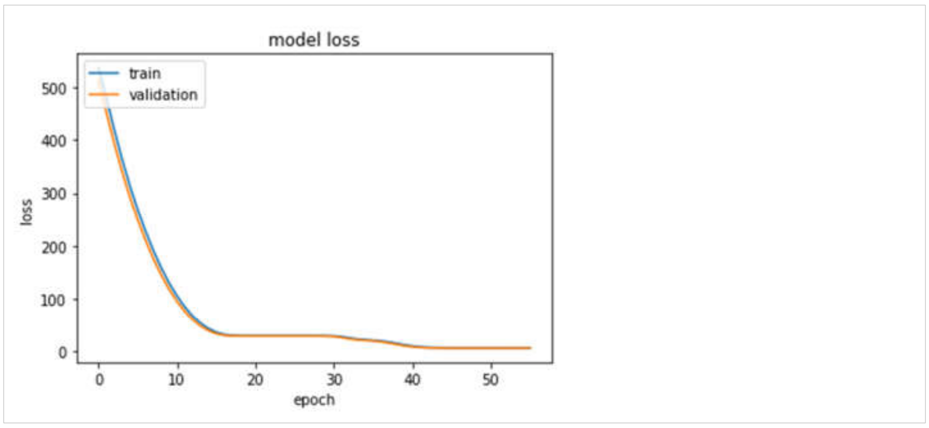
The output above shows that the RMSE, 2.38 for train data and 2.36 for test data. Ideally, the lower the RMSE value, the better the model performance. However, in contrast to accuracy, it is not straightforward to interpret RMSE as we would have to look at the unit which in our case is in thousands.

Plotting Model loss graph



```
print(history.history.keys())
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'mean_squared_error', 'val_loss', 'val_mean_squared_error'])
```

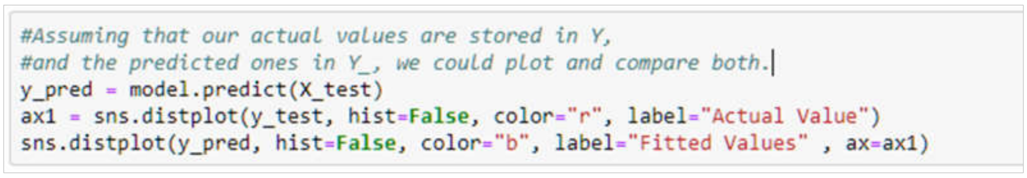


A good fit is the goal of the learning algorithm and exists between an overfit and underfit model. A plot of learning curves shows a good fit if:

- The plot of training loss decreases to a point of stability.
- The plot of validation loss decreases to a point of stability and has a small gap with the training loss.

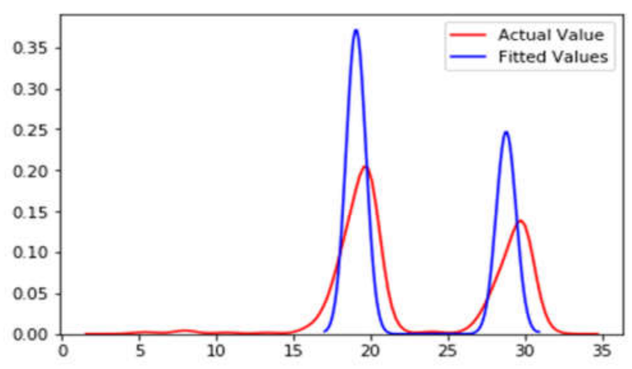
Continued training of a good fit will likely lead to an overfit.

Plotting graph of actual and predicted values



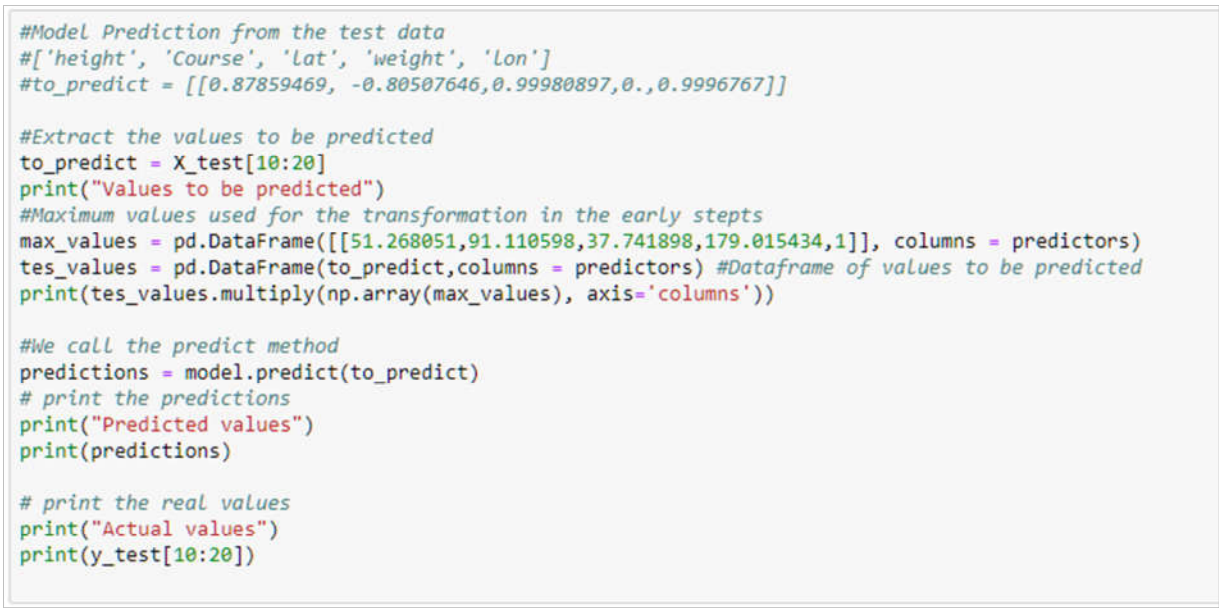
```
#Assuming that our actual values are stored in Y,
#and the predicted ones in Y_, we could plot and compare both.
y_pred = model.predict(X_test)
ax1 = sns.distplot(y_test, hist=False, color="r", label="Actual Value")
sns.distplot(y_pred, hist=False, color="b", label="Fitted Values" , ax=ax1)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1ca64d13470>
```

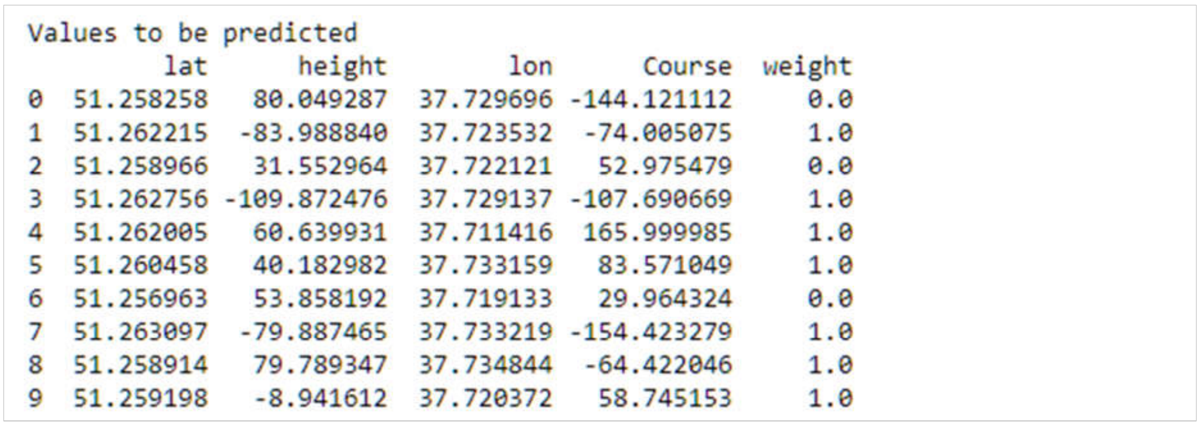


Performing Model prediction on the test data

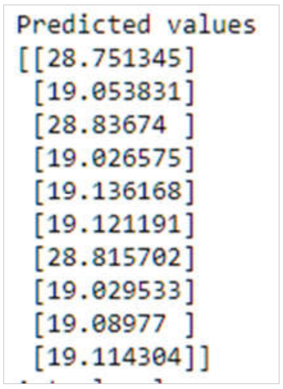
Here since the data was transformed we multiply the values to be predicted with the maximum values used for transformation so we can get the actual values of the features.



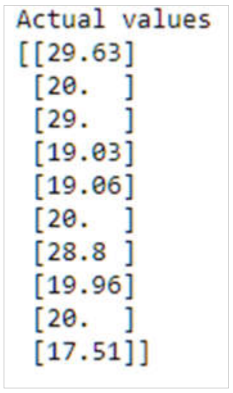
```
#Model Prediction from the test data
#['height', 'Course', 'lat', 'weight', 'lon']
#to_predict = [[0.87859469, -0.80507646,0.99980897,0.,0.9996767]]

#Extract the values to be predicted
to_predict = X_test[10:20]
print("Values to be predicted")
#Maximum values used for the transformation in the early stepts
max_values = pd.DataFrame([[51.268051,91.110598,37.741898,179.015434,1]], columns = predictors)
tes_values = pd.DataFrame(to_predict,columns = predictors) #Dataframe of values to be predicted
print(tes_values.multiply(np.array(max_values), axis='columns'))

#We call the predict method
predictions = model.predict(to_predict)
# print the predictions
print("Predicted values")
print(predictions)

# print the real values
print("Actual values")
print(y_test[10:20])
```



```
Values to be predicted
         lat       height        lon       Course  weight
0  51.258258    80.049287  37.729696  -144.121112     0.0
1  51.262215   -83.988840  37.723532   -74.005075     1.0
2  51.258966    31.552964  37.722121    52.975479     0.0
3  51.262756  -109.872476  37.729137  -107.690669     1.0
4  51.262005    60.639931  37.711416   165.999985     1.0
5  51.260458    40.182982  37.733159    83.571049     1.0
6  51.256963    53.858192  37.719133    29.964324     0.0
7  51.263097   -79.887465  37.733219  -154.423279     1.0
8  51.258914    79.789347  37.734844   -64.422046     1.0
9  51.259198    -8.941612  37.720372    58.745153     1.0
```



```
Predicted values
[[28.751345]
 [19.053831]
 [28.83674 ]
 [19.026575]
 [19.136168]
 [19.121191]
 [28.815702]
 [19.029533]
 [19.08977 ]
 [19.114304]]
```

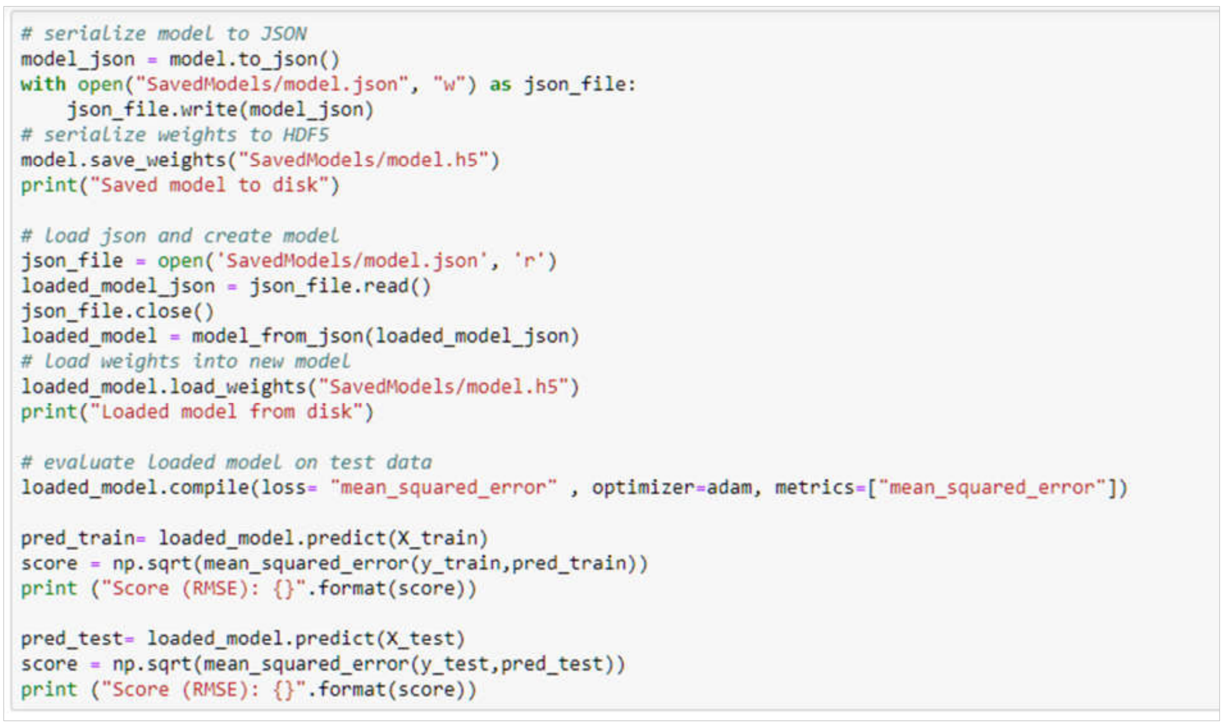


```
Actual values
[[29.63]
 [20.  ]
 [29.  ]
 [19.03]
 [19.06]
 [20.  ]
 [28.8 ]
 [19.96]
 [20.  ]
 [17.51]]
```

As we can see from the predicted and actual values they are very similar so this indicates that our model is performing well.

Save the trained model to disk

Saving the trained model allows us to use the trained model directly. We will not need to retrain the model again and again this saves us time.



```
# serialize model to JSON
model_json = model.to_json()
with open("SavedModels/model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("SavedModels/model.h5")
print("Saved model to disk")

# load json and create model
json_file = open('SavedModels/model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("SavedModels/model.h5")
print("Loaded model from disk")

# evaluate loaded model on test data
loaded_model.compile(loss= "mean_squared_error" , optimizer=adam, metrics=["mean_squared_error"])

pred_train= loaded_model.predict(X_train)
score = np.sqrt(mean_squared_error(y_train,pred_train))
print ("Score (RMSE): {}".format(score))

pred_test= loaded_model.predict(X_test)
score = np.sqrt(mean_squared_error(y_test,pred_test))
print ("Score (RMSE): {}".format(score))
```

Conclusion

We have built Regression models using the deep learning framework, Keras. This model is achieving a stable performance with not much variance in the train and test set RMSE. The most ideal result would be an RMSE value of zero, but that's almost impossible.

There are other iterations such as changing the number of neurons, adding more hidden layers, or increasing the number of epochs, which can be tried out to see the impact on model performance. This regression problem could also be modeled using other algorithms such as Decision Tree, Random Forest, Gradient Boosting or Support Vector Machines.

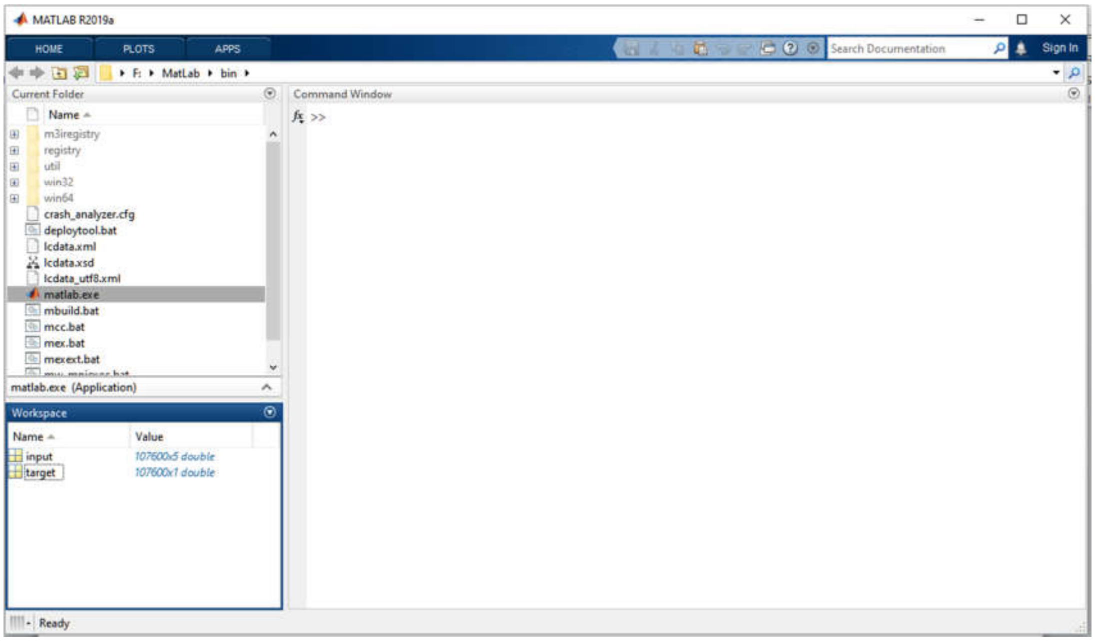
References

- https://www.pluralsight.com/guides/regression-keras?clickid=3-pw7KzT5xyLU24wUx0Mo3EHUkEwLGT5S3swyM0&irgwc=1&mpid=29332&aid=7010a000001xAKZAA2&utm_medium=digital_affiliate&utm_campaign=29332&utm_source=impactradius
- https://github.com/keras-team/keras/issues/7947
- https://stackabuse.com/tensorflow-2-0-solving-classification-and-regression-problems/
- https://machinelearningmastery.com/save-load-keras-deep-learning-models/
- https://machinelearningmastery.com/custom-metrics-deep-learning-keras-python/
- https://towardsdatascience.com/ways-to-detect-and-remove-the-outliers-404d16608dba
- https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

Multiple Regression using MATLAB

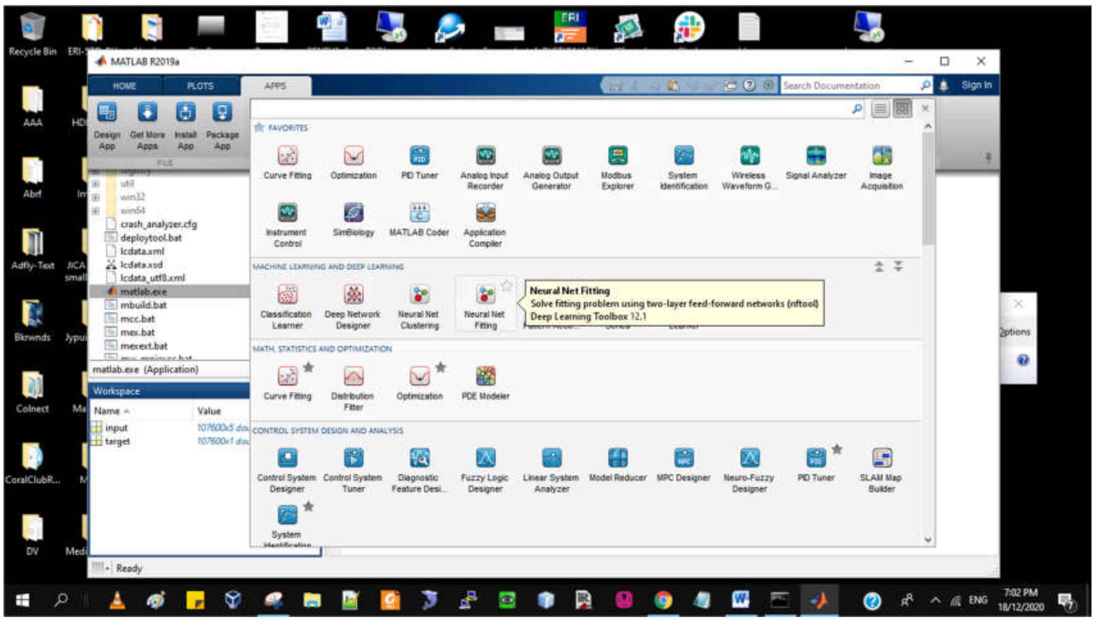
With the same model structure (number of neurons 6 and activation function sigmoid) we get similar results in MATLAB.

Step 1: Split the data set in to target (speed) and input (lat,height,lon ,Course,weight) files and import it to MATLAB as shown below.



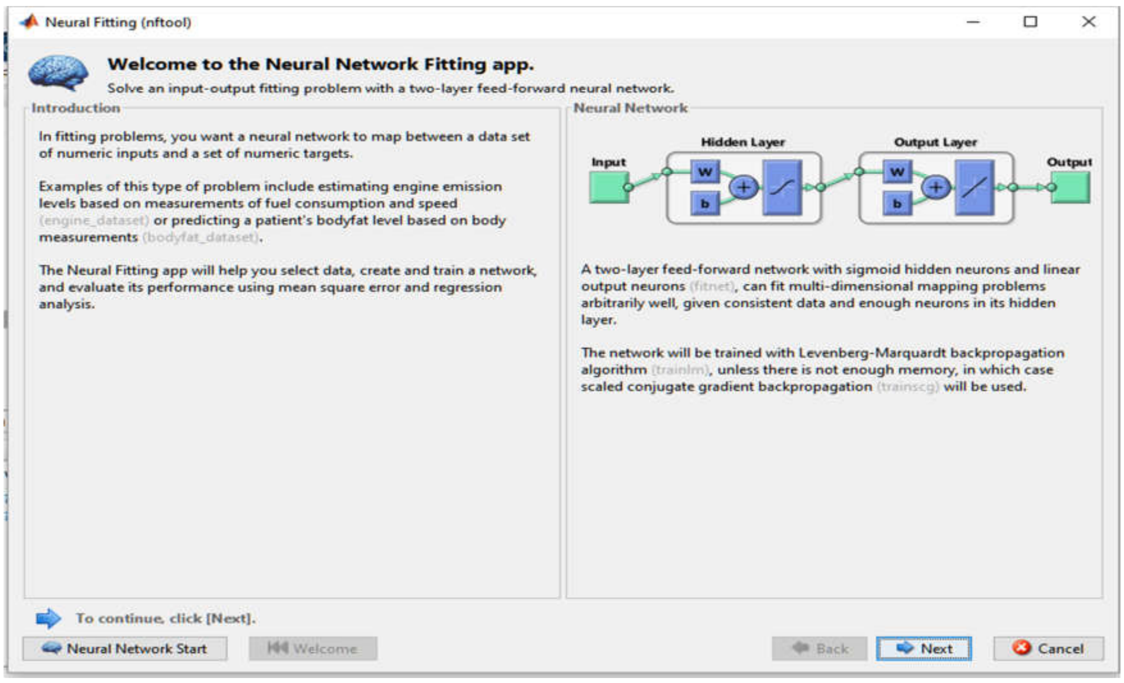
```
Name      Value
input     107600x5 double
target    107600x1 double
```

Step 2: Go to APPS menu and select Neural Net Fitting as shown in screenshot.



Neural Net Fitting
Solve fitting problem using two-layer feed-forward networks (nftool)
Deep Learning Toolbox 12.1

Then click Next in the following screenshot.



Welcome to the Neural Network Fitting app.

Solve an input-output fitting problem with a two-layer feed-forward neural network.

Introduction

In fitting problems, you want a neural network to map between a data set of numeric inputs and a set of numeric targets.

Examples of this type of problem include estimating engine emission levels based on measurements of fuel consumption and speed (engine_dataset) or predicting a patient's bodyfat level based on body measurements (bodyfat_dataset).

The Neural Fitting app will help you select data, create and train a network, and evaluate its performance using mean square error and regression analysis.

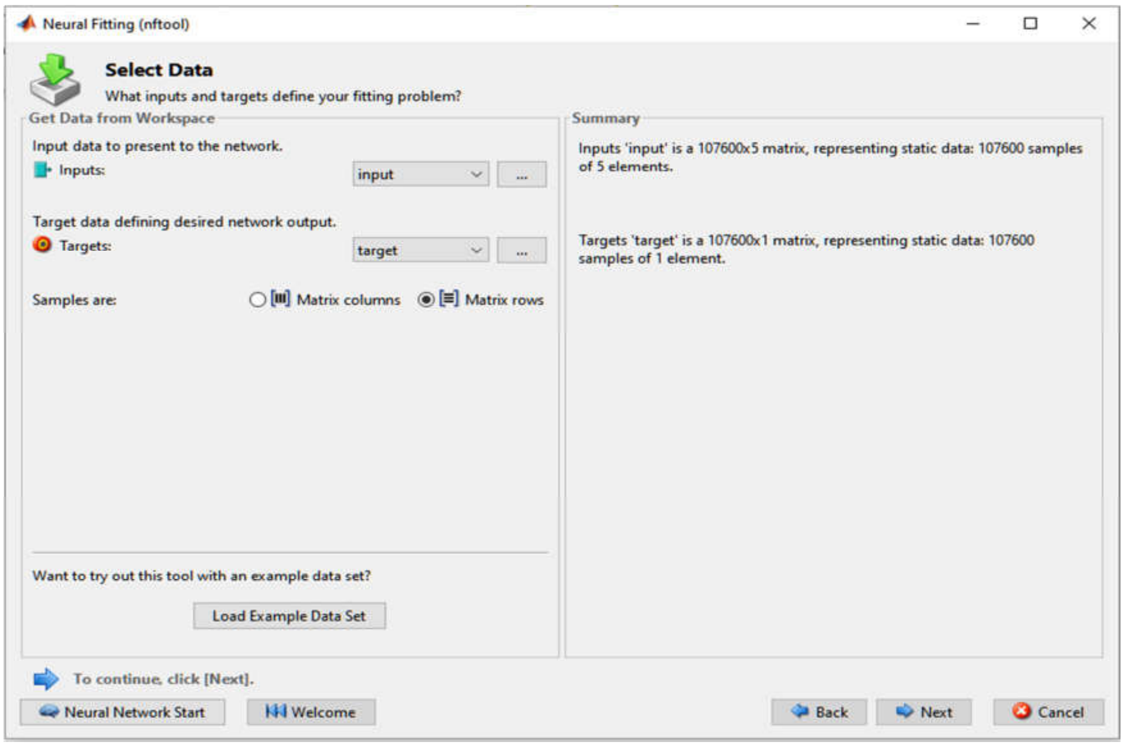
Neural Network

A two-layer feed-forward network with sigmoid hidden neurons and linear output neurons (fitnet), can fit multi-dimensional mapping problems arbitrarily well, given consistent data and enough neurons in its hidden layer.

The network will be trained with Levenberg-Marquardt backpropagation algorithm (trainlm), unless there is not enough memory, in which case scaled conjugate gradient backpropagation (trainscg) will be used.

To continue, click [Next].

Step 3: Then select the target and input files for training as shown below. And also select the Matrix rows option. Most importantly make sure that the input and target files have the same number of Rows.



Select Data

What inputs and targets define your fitting problem?

Get Data from Workspace

Input data to present to the network.

Inputs: input

Target data defining desired network output.

Targets: target

Samples are: ○ Matrix columns ● Matrix rows

Want to try out this tool with an example data set?

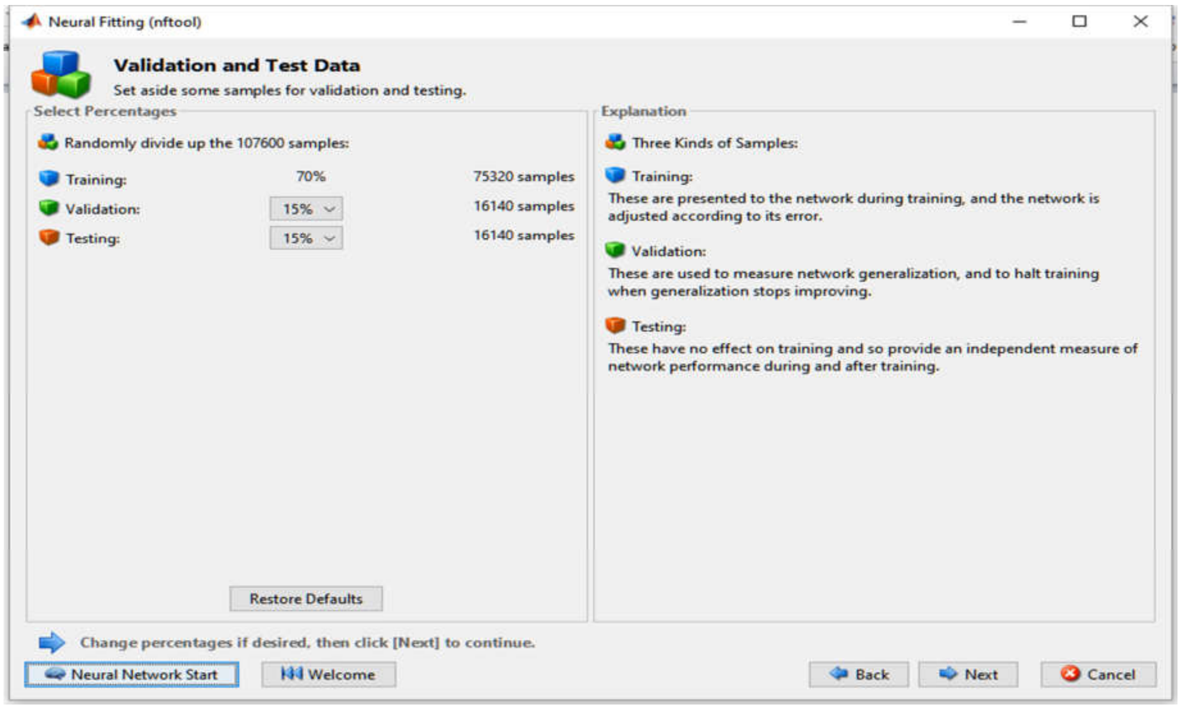
Summary

Inputs 'input' is a 107600x5 matrix, representing static data: 107600 samples of 5 elements.

Targets 'target' is a 107600x1 matrix, representing static data: 107600 samples of 1 element.

To continue, click [Next].

Then Click Next in the following form


Neural Fitting (nftool)

Validation and Test Data

Set aside some samples for validation and testing.

Select Percentages

Randomly divide up the 107600 samples:

Training:	70%	75320 samples
Validation:	15%	16140 samples
Testing:	15%	16140 samples

Explanation

Three Kinds of Samples:

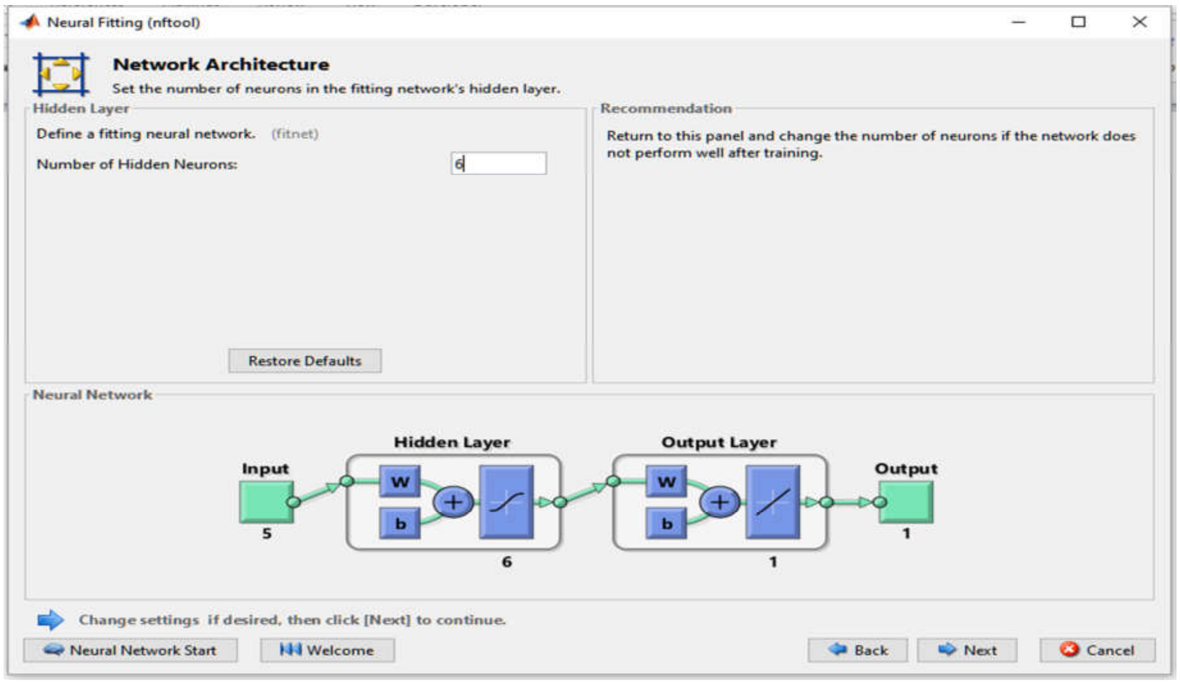
Training:
These are presented to the network during training, and the network is adjusted according to its error.

Validation:
These are used to measure network generalization, and to halt training when generalization stops improving.

Testing:
These have no effect on training and so provide an independent measure of network performance during and after training.

Change percentages if desired, then click [Next] to continue.

Step 4: Specify the number of neurons. Here we use 6 which is number of features plus 1. Same as we used in python. Then Click Next.


Neural Fitting (nftool)

Network Architecture

Set the number of neurons in the fitting network's hidden layer.

Hidden Layer

Define a fitting neural network. (fitnet)

Number of Hidden Neurons: 6

Recommendation

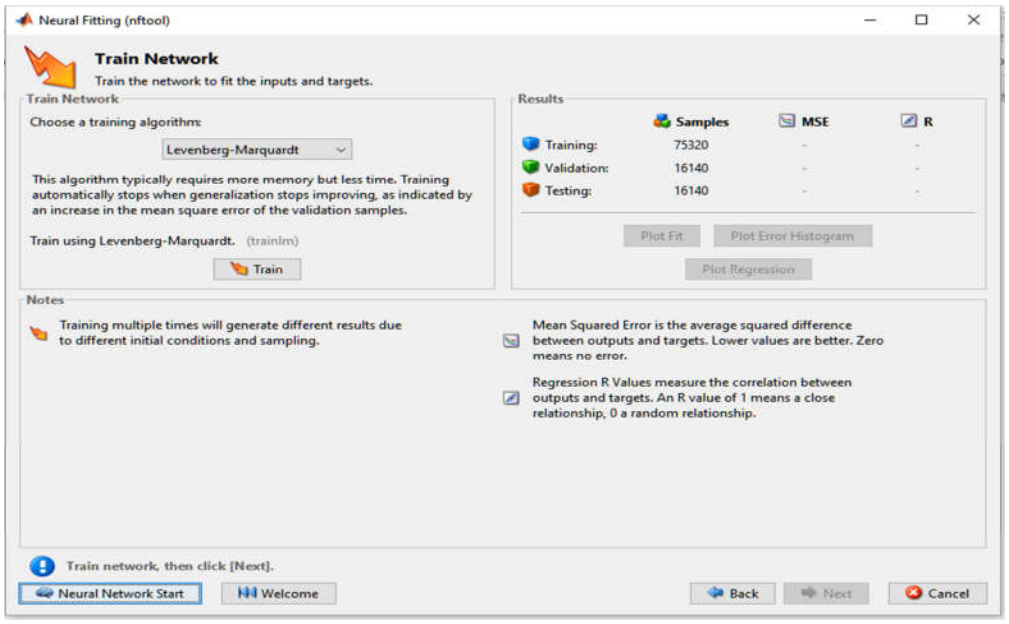
Return to this panel and change the number of neurons if the network does not perform well after training.

Neural Network

Change settings if desired, then click [Next] to continue.

Step 5: Start training.

Click the Train Button.



Neural Fitting (nftool)

Train Network

Train the network to fit the inputs and targets.

Train Network

Choose a training algorithm: Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt. (trainlm)

Train

Results

	Samples	MSE	R
Training:	75320	-	-
Validation:	16140	-	-
Testing:	16140	-	-

Plot Fit | Plot Error Histogram | Plot Regression

Notes

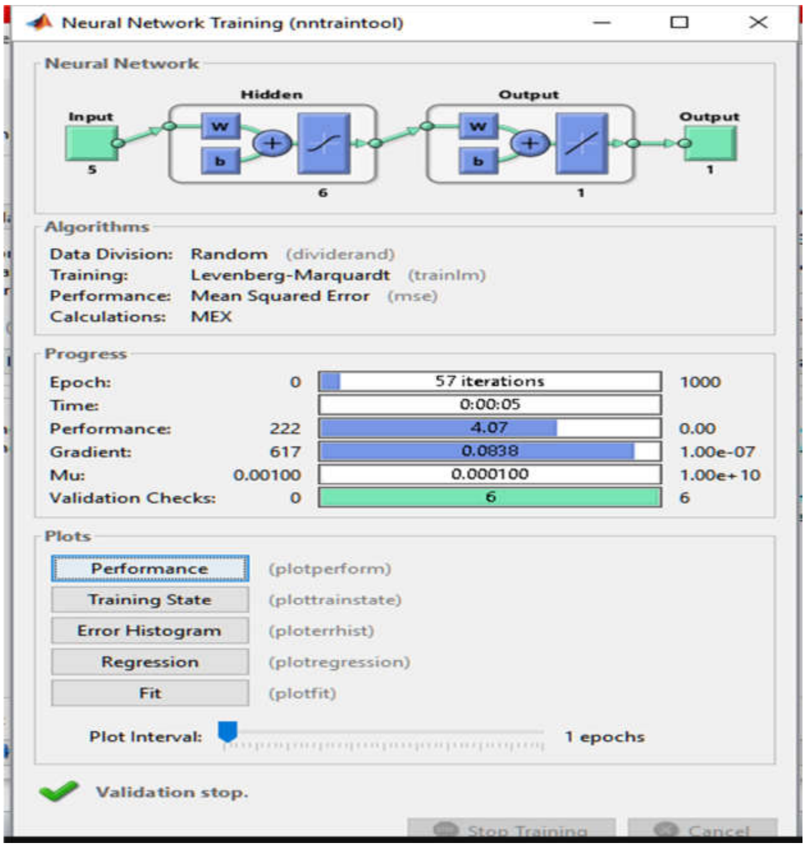
Training multiple times will generate different results due to different initial conditions and sampling.

Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Train network, then click [Next].

Neural Network Start | Welcome | Back | Next | Cancel



Neural Network Training (nntraintool)

Neural Network

Algorithms

Data Division:	Random (dividerand)
Training:	Levenberg-Marquardt (trainlm)
Performance:	Mean Squared Error (mse)
Calculations:	MEX

Progress

Epoch:	0	57 iterations	1000
Time:		0:00:05	
Performance:	222	4.07	0.00
Gradient:	617	0.0838	1.00e-07
Mu:	0.00100	0.000100	1.00e+10
Validation Checks:	0	6	6

Plots

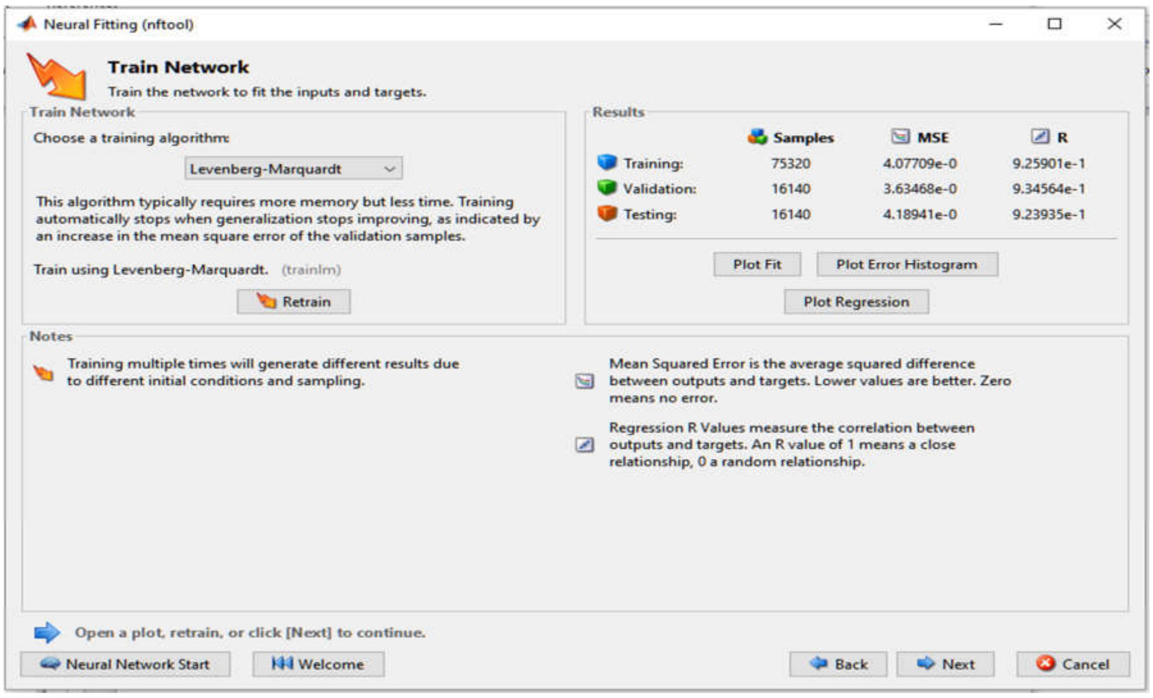
Performance	(plotperform)
Training State	(plottrainstate)
Error Histogram	(ploterrhist)
Regression	(plotregression)
Fit	(plotfit)

Plot Interval: 1 epochs

Validation stop.

Stop Training | Cancel

Here we can see the results of the training. We got RMSE 4 which is very good result. But RMSE was a little better in the Model in Python. Any way this is also a good result. Click Next.



Neural Fitting (nftool)

Train Network

Train the network to fit the inputs and targets.

Train Network

Choose a training algorithm: Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt. (trainlm)

Results

	Samples	MSE	R
Training:	75320	4.07709e-0	9.25901e-1
Validation:	16140	3.63468e-0	9.34564e-1
Testing:	16140	4.18941e-0	9.23935e-1

Notes

Training multiple times will generate different results due to different initial conditions and sampling.

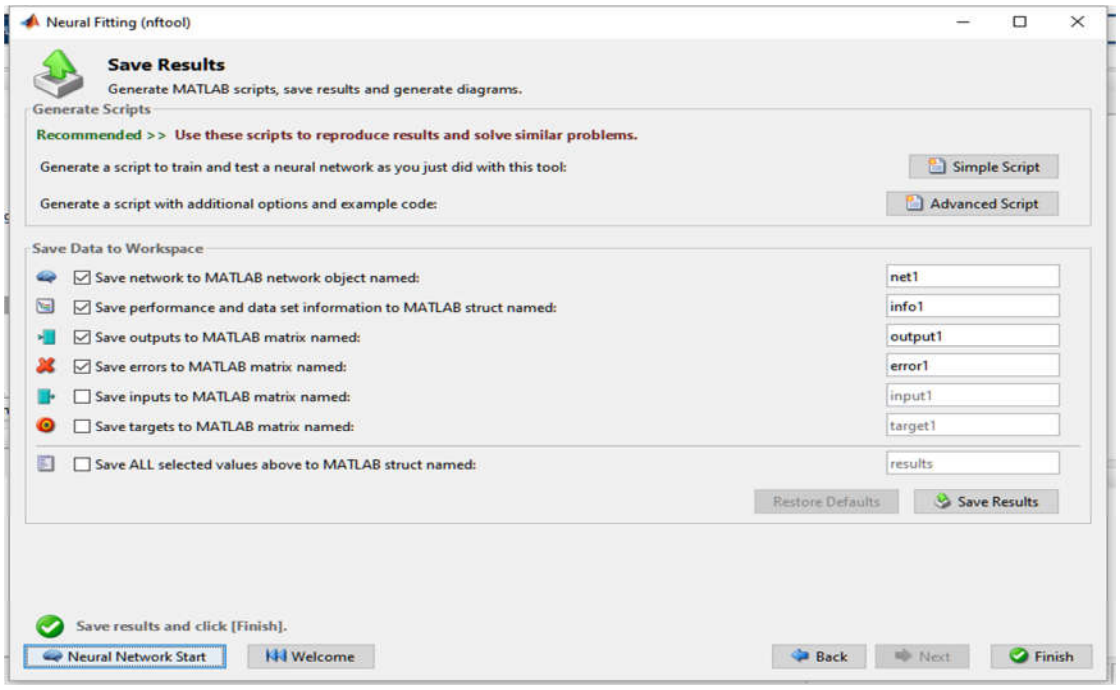
Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Open a plot, retrain, or click [Next] to continue.

Step 6: Save the trained model.

First Click Save Results and then Click Simple Script Buttons.



Neural Fitting (nftool)

Save Results

Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

Recommended >> Use these scripts to reproduce results and solve similar problems.

Generate a script to train and test a neural network as you just did with this tool:

Generate a script with additional options and example code:

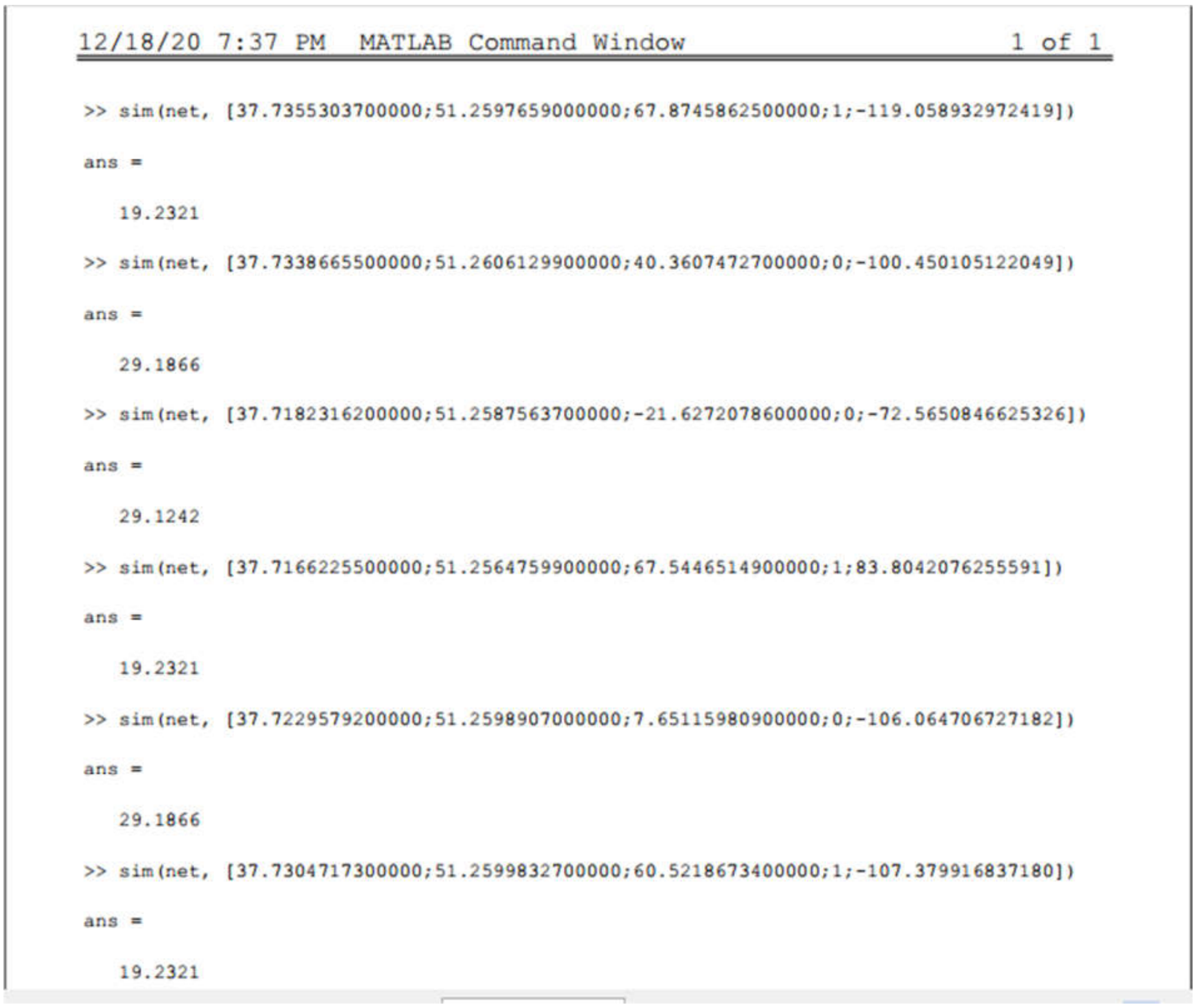
Save Data to Workspace

☑ Save network to MATLAB network object named:	net1
☑ Save performance and data set information to MATLAB struct named:	info1
☑ Save outputs to MATLAB matrix named:	output1
☑ Save errors to MATLAB matrix named:	error1
☐ Save inputs to MATLAB matrix named:	input1
☐ Save targets to MATLAB matrix named:	target1
☐ Save ALL selected values above to MATLAB struct named:	results

Save results and click [Finish].

Step 7: Prepare the data for prediction. Here we have 6 sets of data.

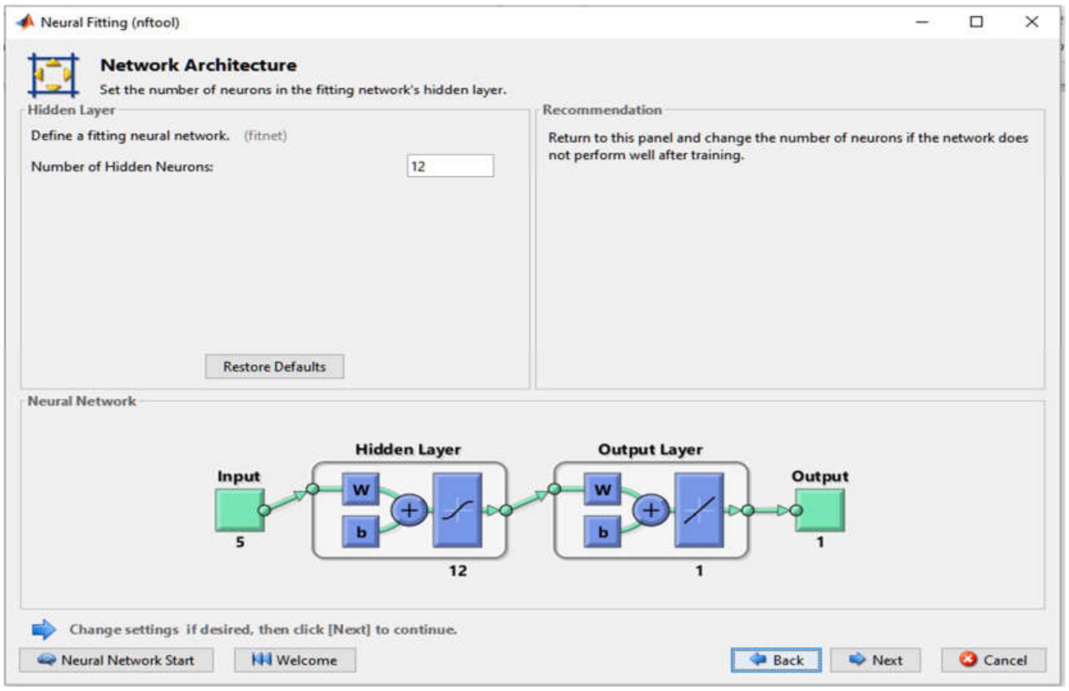
lon	lat	height	weight	Course	Actual speed	Predicted speed
37.73553	51.25977	67.87459	1	-119.059	18.27	19.23
37.73387	51.26061	40.36075	0	-100.45	27.77	29.19
37.71823	51.25876	-21.6272	0	-72.5651	27.13	29.12
37.71662	51.25648	67.54465	1	83.80421	19.41	19.23
37.72296	51.25989	7.65116	0	-106.065	27.44	29.19
37.73047	51.25998	60.52187	1	-107.38	20.54	19.23



```
12/18/20 7:37 PM   MATLAB Command Window                                1 of 1

>> sim(net, [37.7355303700000;51.2597659000000;67.8745862500000;1;-119.058932972419])

ans =

   19.2321

>> sim(net, [37.7338665500000;51.2606129900000;40.3607472700000;0;-100.450105122049])

ans =

   29.1866

>> sim(net, [37.7182316200000;51.2587563700000;-21.6272078600000;0;-72.5650846625326])

ans =

   29.1242

>> sim(net, [37.7166225500000;51.2564759900000;67.5446514900000;1;83.8042076255591])

ans =

   19.2321

>> sim(net, [37.7229579200000;51.2598907000000;7.65115980900000;0;-106.064706727182])

ans =

   29.1866

>> sim(net, [37.7304717300000;51.2599832700000;60.5218673400000;1;-107.379916837180])

ans =

   19.2321
```

As we can see from the predictions the actual value and the predicted values are almost the same. So we can conclude that the model is performing well.

As we can see the created model is predicting well. To improve the accuracy of prediction we can increase the number of neurons. Here it is 12, so RMSE is decreased a little which is a good sign.



Network Architecture

Set the number of neurons in the fitting network's hidden layer.

Hidden Layer

Define a fitting neural network. (fitnet)

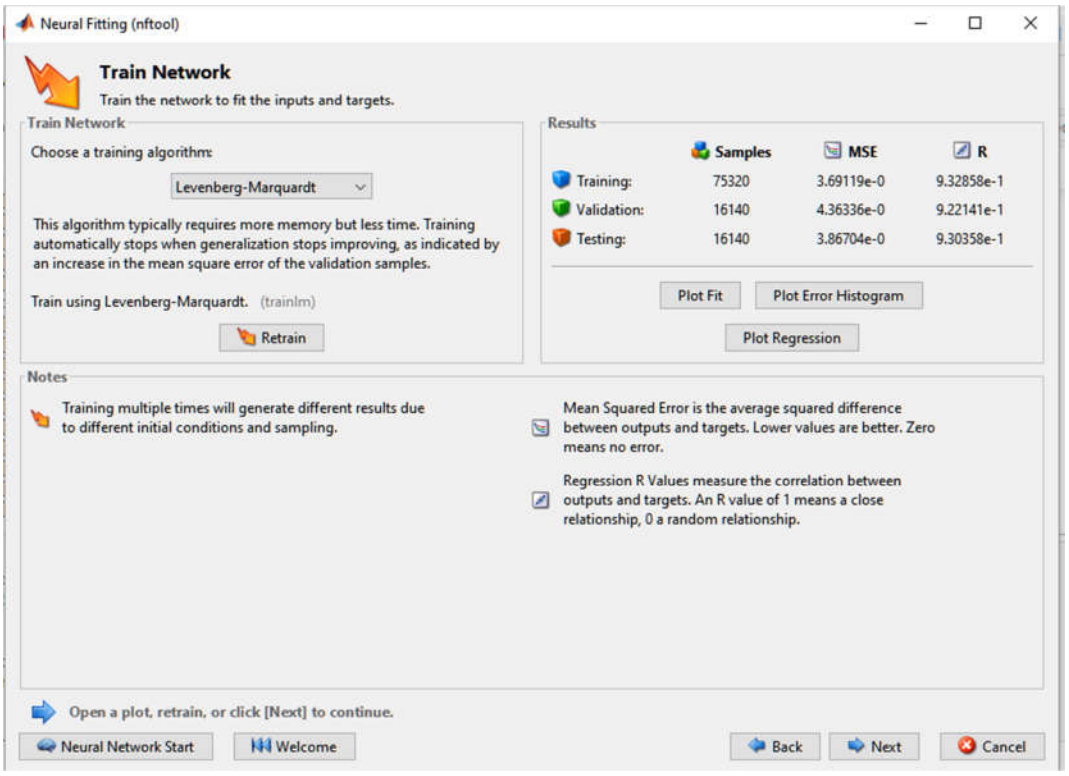
Number of Hidden Neurons: 12

Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

Neural Network

Change settings if desired, then click [Next] to continue.



Train Network

Train the network to fit the inputs and targets.

Train Network

Choose a training algorithm: Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt. (trainlm)

Results

	Samples	MSE	R
Training:	75320	3.69119e-0	9.32858e-1
Validation:	16140	4.36336e-0	9.22141e-1
Testing:	16140	3.86704e-0	9.30358e-1

Notes

Training multiple times will generate different results due to different initial conditions and sampling.

Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Open a plot, retrain, or click [Next] to continue.