Howdy!

Today we will make a function that will imitate the pipeline of the standard regression problem.

This function will allow us to quickly investigate our regression results and give insight into the performance of different models.

We will use classical libraries: Numpy, Pandas, Matlplotlib, Seaborn, and Sklearn.

In this post, we will be focused on performing each step one by one, so we can see how basic functions work, and we will focus on steps that can lead to memory leakage between train and test data; which can result in models with less quality.

But the first thing we need to answer is: What is the regression problem?

From Wikipedia:

In [statistical modeling](https://en.wikipedia.org/wiki/Statistical_model), **regression analysis** is a set of statistical processes for [estimating](https://en.wikipedia.org/wiki/Estimation_theory) the relationships between a [dependent variable](https://en.wikipedia.org/wiki/Dependent_variable) (often called the ''outcome'' or ''response'' variable) and one or more [independent variables](https://en.wikipedia.org/wiki/Independent_variable) (often called ''predictors'', ''covariates'', ''explanatory variables'' or ''features'').

In easy words: We need to predict the value of a variable based on other variables.

Let's dive in and import required libraries:

NumPy NumPy NumPy as np

import pandas as pd

import csv

import time

from sklearn.preprocessing import OneHotEncoder, PolynomialFeatures, StandardScaler

from sklearn.model\_selection import cross\_validate, KFold, train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.linear\_model import Ridge, Lasso

import seaborn as sns

import matplotlib.pyplot as plt

1. First step:

Define the dependent and independent variables:

We will work with DataFrame called – Data and will predict the price of the house based on other parameters.

# Define our independent variables:

dependent = “price”

X = Data.drop(dependent, axis = 1)

y = Data[dependent]

1. Second step:

Let’s initialize transformers that will help us to achieve it:

kf = KFold(5)

encoder = OneHotEncoder(handle\_unknown='ignore')

scaler = StandardScaler()

poly = PolynomialFeatures(degree = n)

OneHotEncoder – will encode categorical variables into binary form. For example:

Note! : Sometimes, people use the standard Pandas method pd.to\_dummies, which is easier to use. However, this method has one disadvantage – we can't give instructions what to do in case of an unknown class in the testing dataset. It can happen when the training dataset doesn't have some value for variables during the making dummies process. For example: imagine in your dataset you encoded the type of the house, but there was only one ""seaside villa"" and it got to your testing set.

In this case, your model trained on a dataset without it will not know what to do with it and will throw you an error.

OneHotEncoder from the other side allows us to define strategy, what to do with unknown variables, and we can specify it in parameters during initialization - OneHotEncoder(handle\_unknown='ignore")

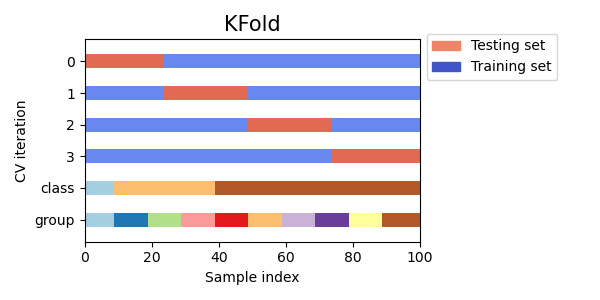
StandardScaler – will standardize features by removing the mean and scaling to unit variance. So it will put every numerical feature in the scales approximately -3 to 3.

In our data, we can have one variable between 100 000 and 1 000 000(price, for example), another one in the range 500-5000(sq footage). So we need to treat them equally, and StandardScaler will allow us to achieve it.

PolynomialFeatures – will engineer new features by raising to the power, and it will also add their products.

KFold will equally split our data into several folds and will allow us to divide between training and testing.

More about it from the SKlearn website: <https://scikit-learn.org/stable/modules/cross_validation.html#k-fold>



1. We need to divide our data into two groups:

Numerical and Categorical so that we can apply methods based on a class.

Fortunately, Pandas have nice functions to achieve it. pd.select\_dtypes: this function will grab only columns of appropriate types.

X\_train\_num = X\_train.select\_dtypes(include = np.number)

X\_test\_num = X\_test.select\_dtypes(include = np.number)

X\_train\_cat = X\_train.select\_dtypes(include = [object, 'category'])

X\_test\_cat = X\_test.select\_dtypes(include = [object, 'category'])

1. Now, let's create a function that will make all these transformations.

Important! We train our supporting transformers based only on training data and apply them to train and test data to avoid data leakage between test and train sets.

Test data should always be ""invisible"" during training.

This code contains additional comments to understand what happens during each step.

1. Cross-Validation.

Cross-validation is an essential step in estimating the performance of our model. It allows us to divide our dataset into several train-test splits and check model performance with different numbers. As we will see later, the model performance will be other in each of these splits. One of the objectives of a data scientist is not just to make the best performing model, but the model also **should show stable results on new unseen data.**

Sklearn has 2 nice functions that allows to perform cross validation, such as:

sklearn.model\_selection.cross\_val\_score

sklearn.model\_selection.cross\_validate

However, in this tutorial, we will recreate the process on our own for two reasons:

1. These are automatic methods, and we can't integrate our data transformations(step 4) inside. So there will be some data leakage during cross-validation. This data leakage isn't critical; however, we want to exclude it.
2. For educational purposes, we want to imitate cross\_validation with KFold.

We will do 5 KFold splits; we will perform transformations, modeling, and prediction in each of them.

As a final score, we will use the mean value of both five models, but with information about standard deviation in results values.

# Perform 5 - kfold splits with

for train\_index, test\_index in kf.split(X): #We define kf before.

X\_train, X\_test = X.iloc[train\_index,:], X.iloc[test\_index,:]

y\_train, y\_test = y[train\_index], y[test\_index]

# Data transormation based on previous function

X\_train, X\_test = transformation(X\_train, X\_test)

1. Modeling

Next step, we will make a model, train it and make predictions.

We will make predictions for training and test datasets. It will allow us to see when the model starts to overfit.

#Modeling

model = model\_input # Define our model based on function inputs

model.fit(X\_train, y\_train) # Train the model based on train data.

y\_train\_hat = model.predict(X\_train) # Predict Y train

y\_test\_hat = model.predict(X\_test) # Preduct Y test

1. Calculate performance metrics.

We will use 2 metrics: R^2 and RMSE (Root mean squared error)

#Calculate scores

RMSE\_train = mean\_squared\_error(y\_train, y\_train\_hat, squared = False)

RMSE\_test = mean\_squared\_error(y\_test, y\_test\_hat, squared = False)

r2\_train = r2\_score(y\_train, y\_train\_hat)

r2\_test = r2\_score(y\_test, y\_test\_hat)

#Add scores for each split into overall list of scores

r2\_train\_list.append(r2\_train)

r2\_test\_list.append(r2\_test)

RMSE\_train\_list.append(RMSE\_train)

RMSE\_test\_list.append(RMSE\_test)

# Find mean values and standard deviation of test scores.

# This step goes after all cross validation is finished.

r2\_train\_mean = np.mean(r2\_train\_list)

r2\_test\_mean = np.mean(r2\_test\_list)

r2\_test\_std = np.std(r2\_test\_list)

RMSE\_train\_mean = np.mean(RMSE\_train\_list)

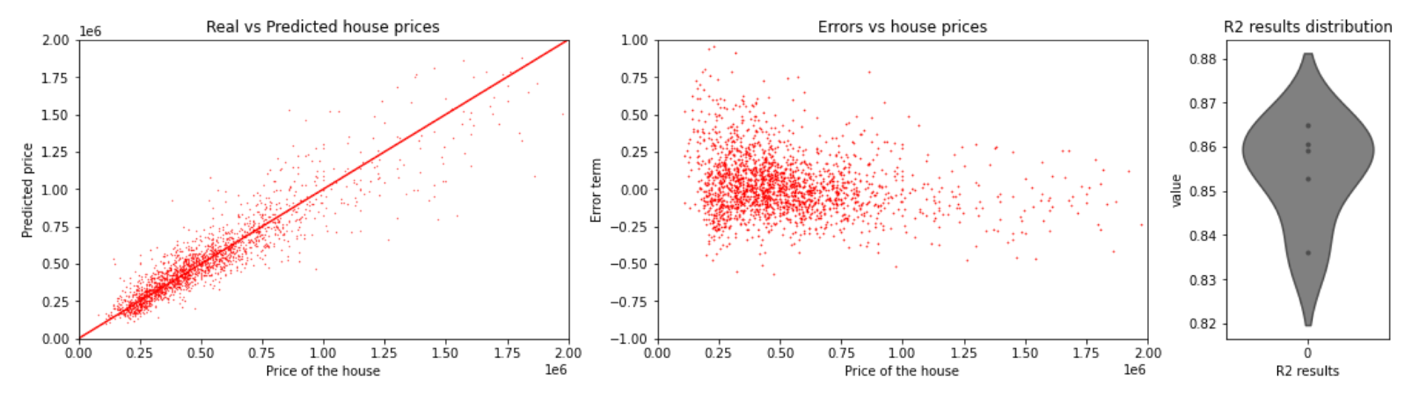
RMSE\_test\_mean = np.mean(RMSE\_test\_list)

RMSE\_test\_std = np.std(RMSE\_test\_list)

1. Visualize results.

We will make 3 graphs, as well as DataFrame with results.

Our graphical results will look like this:



Left graph: Scatterplot of predicted vs. real house prices. The Redline represents the best fit results. We are interested in making our scatterplot closer to the red line.

Central graph: The representation of prediction errors – the X-axis is our real price, and the Y-axis is an error term in prediction. +1 means that we predicted the price with a 100% error (expected price two times higher than the real price).

As we can see from the graph – there is a higher error in cheap houses(less than 500 000$), which probably requires new features engineering.

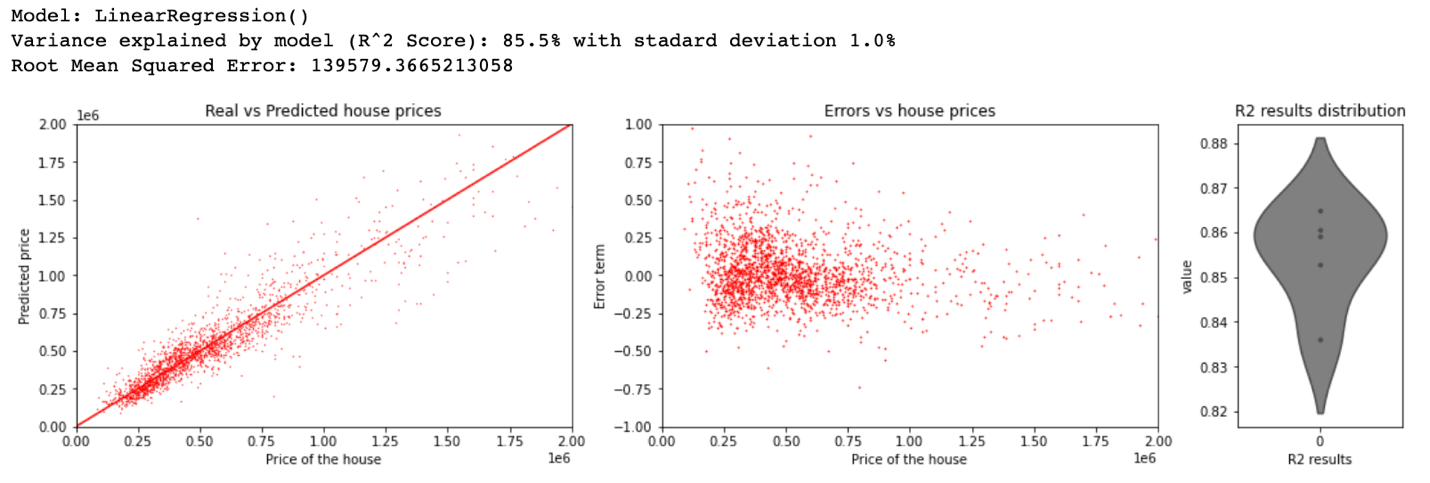
Right graph – Distribution of R^2 scores after cross-validation. We can see how stable our models are.

Our code to make this visualization:

1. Output results:

As an output, we will make a Dataframe that consists of our results; also, we will print our significant parameters.

Because we are focused on creating a recursive function that can be used many times, we will also have our output dataframe as input.



1. Let's sum up all our steps into one function that can be used many times for further investigations.

We will also add the possibility to turn off visualization.

There are also additional smaller steps in our code, and you can find explanations there.

Input parameters will be:

Lets test it.

For testing we will use House Sales in King County, USA dataset from Kaggle: (<https://www.kaggle.com/harlfoxem/housesalesprediction?select=kc_house_data.csv> )

Data preparation are out of the scope of this post, but we will try to keep them simple.

# Download file and load into the notebook.

# Original file can be found here: https://www.kaggle.com/harlfoxem/housesalesprediction?select=kc\_house\_data.csv

df = pd.read\_csv("./data/kc\_house\_data.csv")

# General data preparation - Out of scope of this post.

# We should ensure that:

# All numberical columns have type: - int or float

# All categorical: - object or category

df['age'] = df['date'].map(lambda x: int(x[:4])) - df['yr\_built']

df["renovation"] = df["yr\_renovated"].map(lambda x : 1 if x>0 else 0).astype("object")

df["basement"] = df["sqft\_basement"].map(lambda x : 1 if x>0 else 0).astype("object")

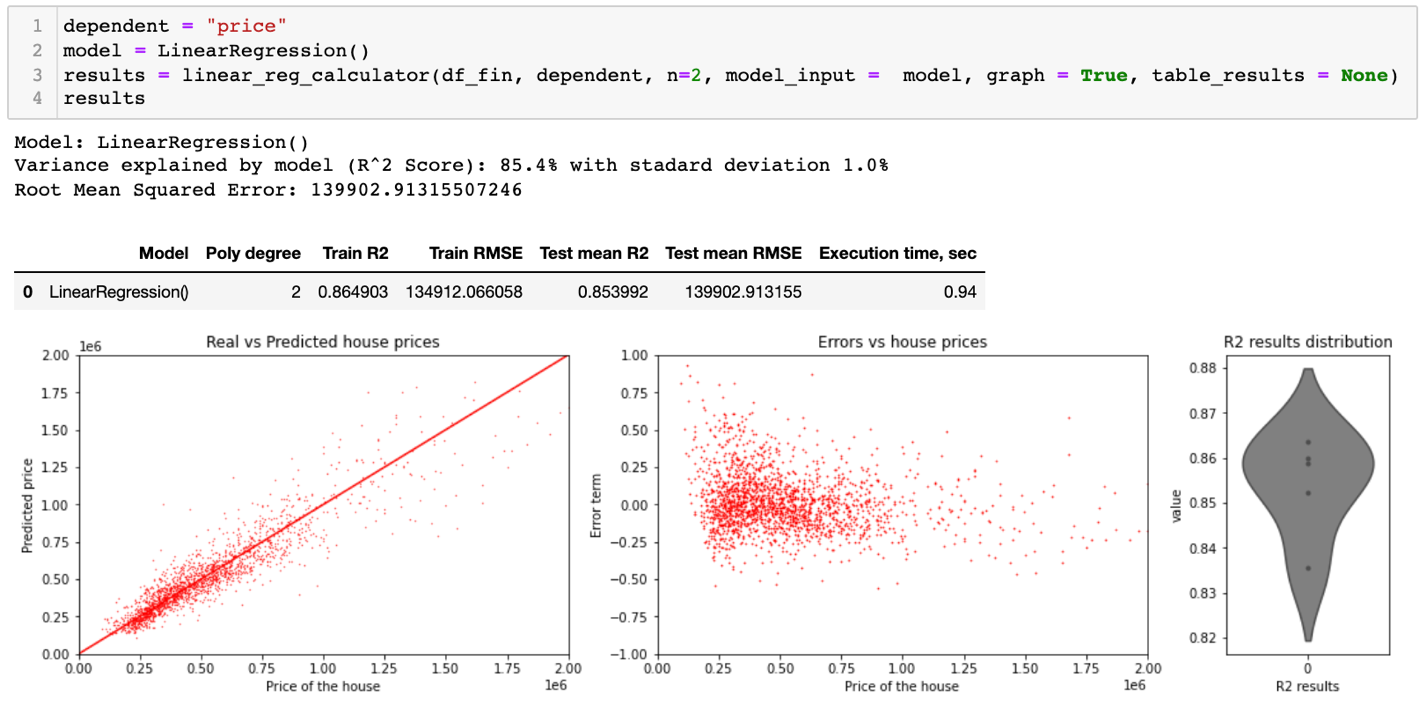
df["zipcode"] = df["zipcode"].astype("object")

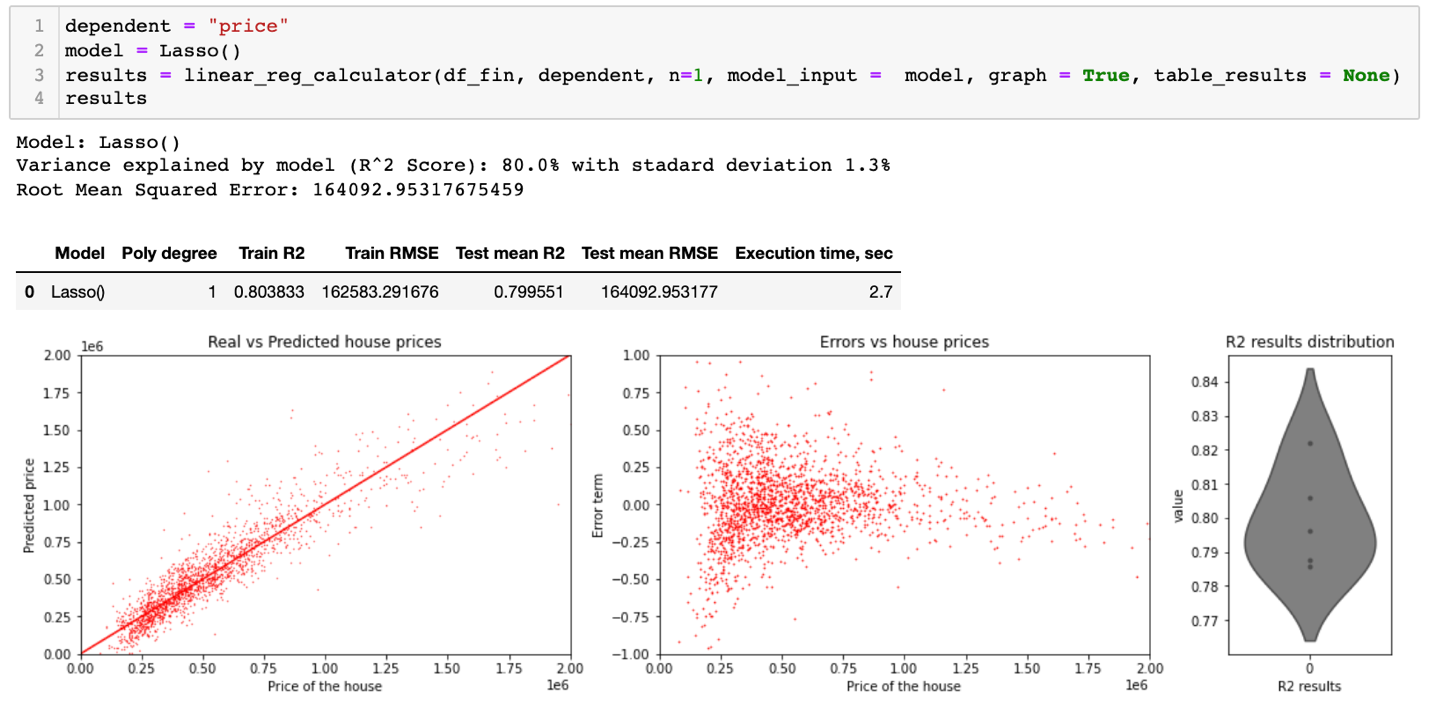
columns\_drop\_prepare = ["id", "date", "bedrooms", "bathrooms", "floors", "sqft\_above", "sqft\_basement", "yr\_built", "yr\_renovated", "lat", "long"]

df\_fin = df.drop(columns\_drop\_prepare, axis = 1)

df\_fin.info()

1. Test on one mode:

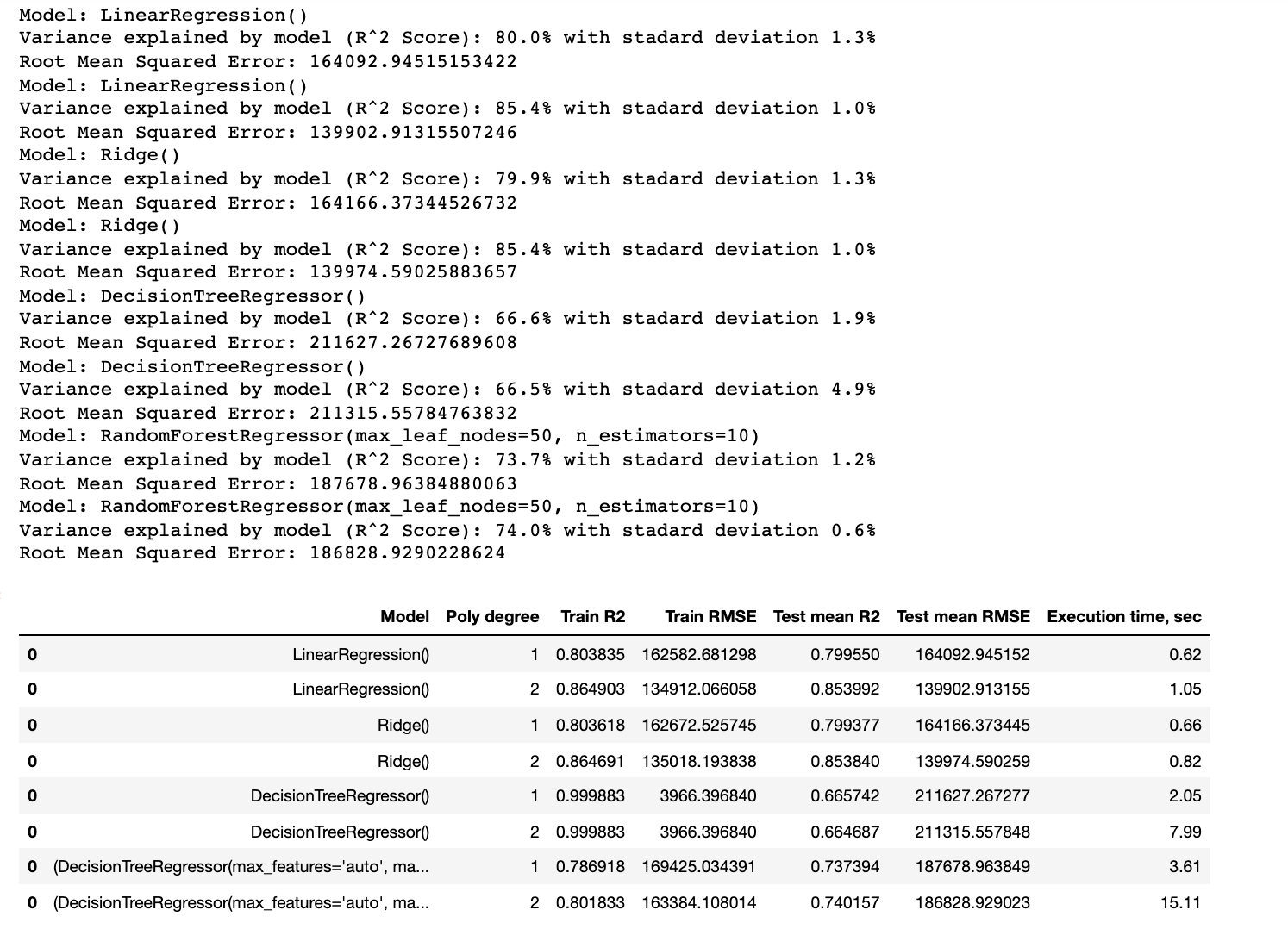




1. Lets make a loop and check performance using different models and parameters. We will turn off visualization:

We will use one of the model with changed hyperparameters.

All these insight is possible to get using a explore functions and making just a few lines of code.



As we can see from the results:

Decision tree with default parameters tends to overfit.

Poly degrees generally improves our model performance.