Which model is a good fit for a particular problem?

In this case study I will try to find good ML model using different models and compare their result plotting the prediction into the real data. We will compare several model's RMSE and visually inspect their results(predictions).

Problem Introduction.

A tunnel building company is using X-rays to find out rock density. They are using an automated boring machine to drill (you can watch an example video of this technique in this link). The machine boring heads on the equipment are switched based on rock density.

The company have performed some lab test. And found signal strength returned in nHZ to their sensors for various rock density types tested. We will plot a scatter plot of the result and perform some exploratory analysis of the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("rock_density_xray.csv")

df.head()
```

These are 5 rows from the data frame.

	Rebound Signal Strength nHz	Rock Density kg/m3
0	72.945124	2.456548
1	14.229877	2.601719
2	36.597334	1.967004
3	9.578899	2.300439
4	21.765897	2.452374

Data Exploration.

Getting some basic information about the data.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300 entries, 0 to 299
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   Signal   300 non-null    float64
 1   Density  300 non-null    float64
dtypes: float64(2)
memory usage: 4.8 KB
```

The data is clean. Their is no need for data cleaning or preparation. It consist of 300 rows. We can go strait into building regression models.

df.describe()

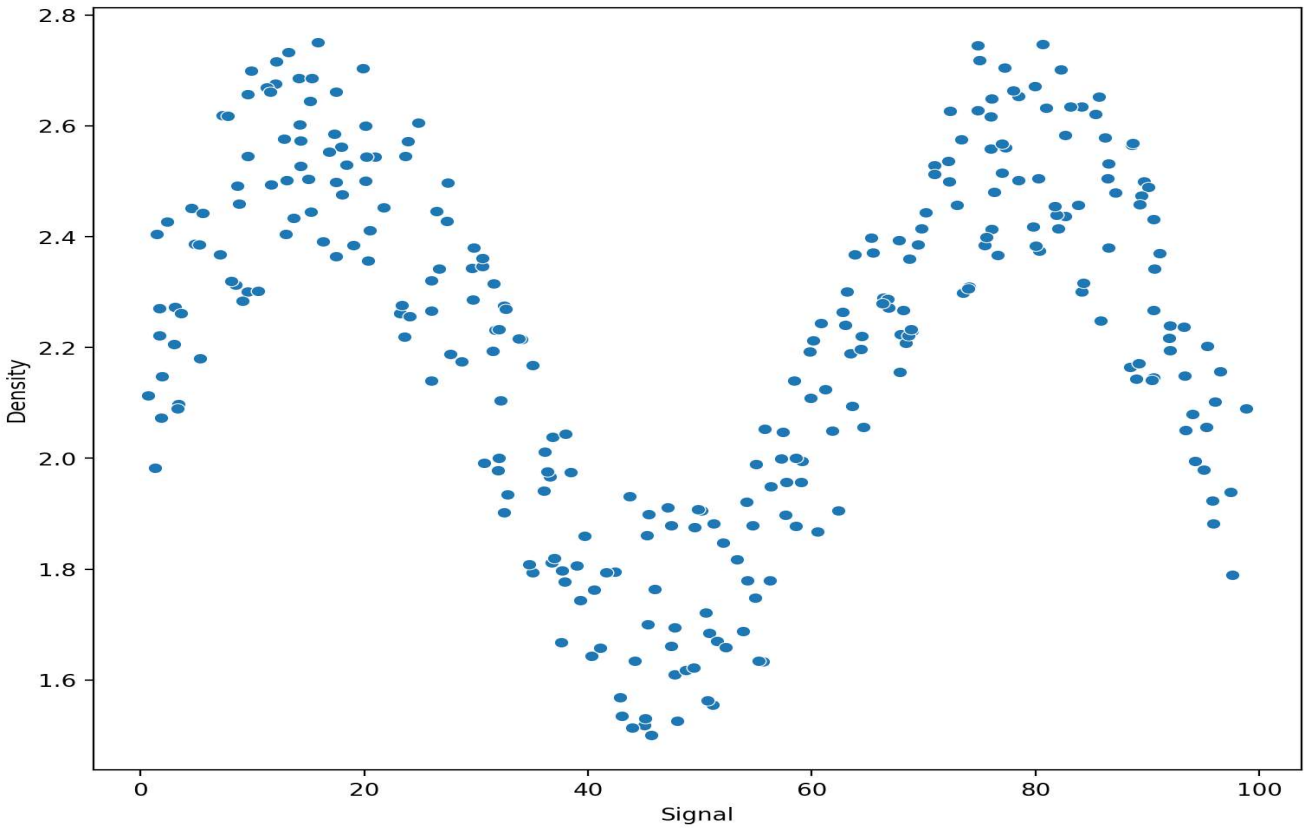
Getting some statistical information about the data.

	Signal	Density
count	300.000000	300.000000
mean	49.725766	2.225063
std	28.341792	0.314512
min	0.700227	1.500000
25%	25.685798	1.987830
50%	50.602886	2.268597
75%	74.854294	2.476944
max	98.831658	2.750000

Data visualization.

df.columns=['Signal',"Density"]

plt.figure(figsize=(12,8),dpi=200)
sns.scatterplot(x='Signal',y='Density',data=df)



From the scatter plot it can be seen that it looks like sine wave relationship.

We will begin by Splitting the data so that have a test set for performance metric evaluation. When it has only one factor the data should be reshaped.

X = df['Signal'].values.reshape(-1,1)
y = df['Density']

The data should be split into training set. Once we found a model we test it on a data that the model has not been trained yet.

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

Linear Regression Model

from sklearn.linear_model import LinearRegression
lr_model = LinearRegression()

lr_model.fit(X_train,y_train)
lr_preds = lr_model.predict(X_test)

from sklearn.metrics import mean_squared_error

The Mean Squared Error (RMSE) will be our metric to evaluate model performance.
np.sqrt(mean_squared_error(y_test,lr_preds))

This models RMSE is 0.2570051996584629

Model evaluation with visualization

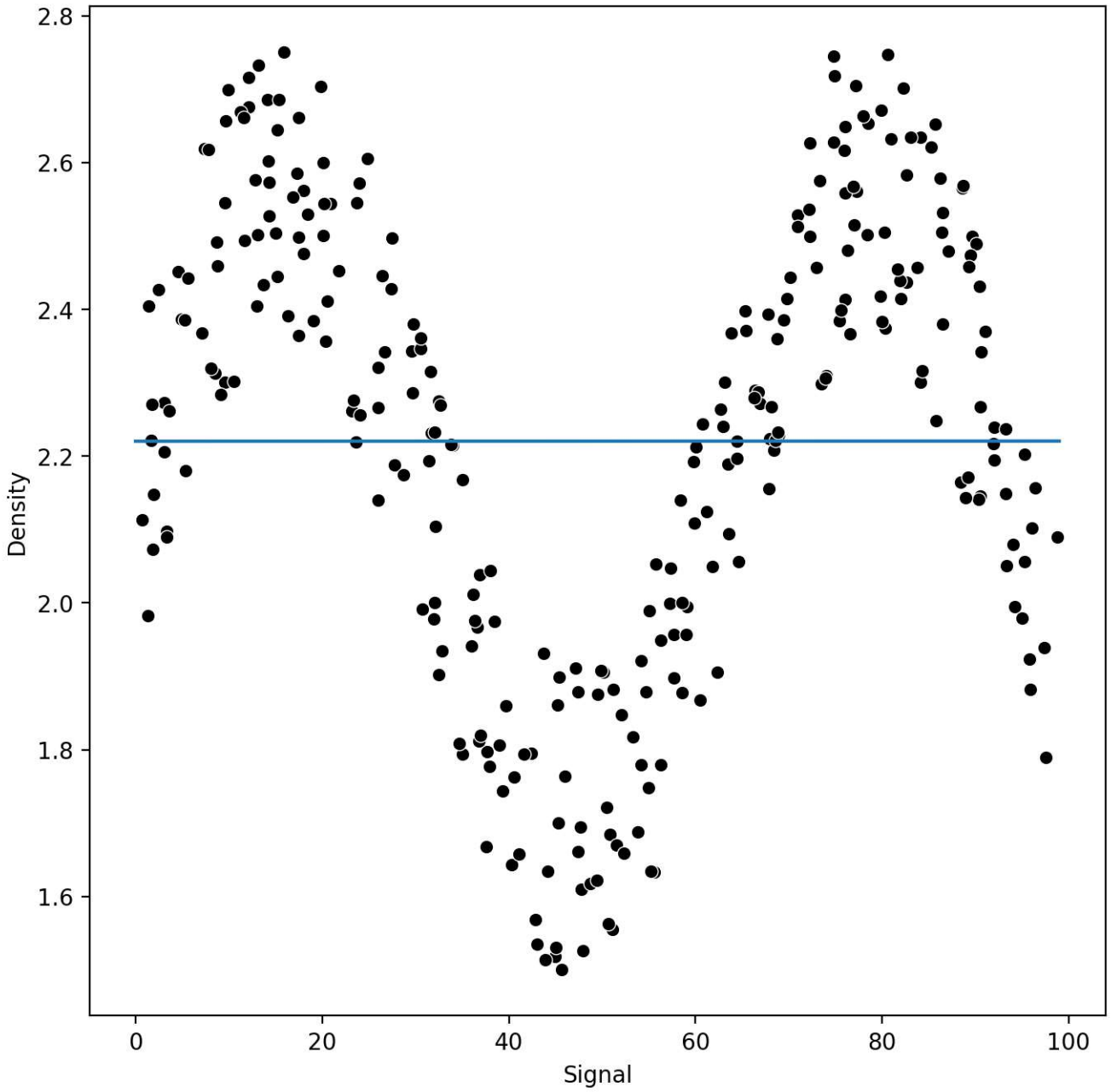
Let’s pass some input data and see how well the model fits.

signal_range = np.arange(0,100)
lr_output = lr_model.predict(signal_range.reshape(-1,1))

Plotting the predicted result over the real data.

plt.figure(figsize=(12,8),dpi=200)

sns.scatterplot(x='Signal',y='Density',data=df,color='black')
plt.plot(signal_range,lr_output)



In [ ]:

This does not look like any good fit. Even thought the RMSE of 0.26 is not bad.

We can always remind ourselves that the mean, standard deviation and RMSE are not the best measures for model fitness. They can be misleading sometimes. It is always good idea to plot and check results. This is another examples of the Anscombe's quartet which comprises four data sets that have nearly

identical simple descriptive statistics, yet have very different distributions and appear very different when graphed. More info about Anscombe's quartet can be found in this `Wikipedia article`.

Polynomial model

from sklearn.preprocessing import PolynomialFeatures

For simplicity, first we will build are model with degree=2. Later on, test it for models with different dgrees.

polynomial_converter = PolynomialFeatures(degree=2, include_bias=False)

poly_features = polynomial_converter.fit_transform(X_train)

poly_features.shape

(300, 2)

from sklearn.linear_model import LinearRegression
model = LinearRegression()

model.fit(poly_features,y_train)

Creating a help function so that we can test various models quickly for the same data.

```
def run_model(model,X_train,y_train,X_test,y_test):

    # Fit Model
    model.fit(X_train,y_train)
     # Get Metrics
    preds = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test,preds))
    print(f'RMSE : {rmse}')
    # Plot results
    signal_range = np.arange(0,100)
    output = model.predict(signal_range.reshape(-1,1))


    plt.figure(figsize=(12,6),dpi=150)
    sns.scatterplot(x='Signal',y='Density',data=df,color='black')
    plt.plot(signal_range,output)
```

If we make a pipeline, then it can be even faster to compare different polynomial models.

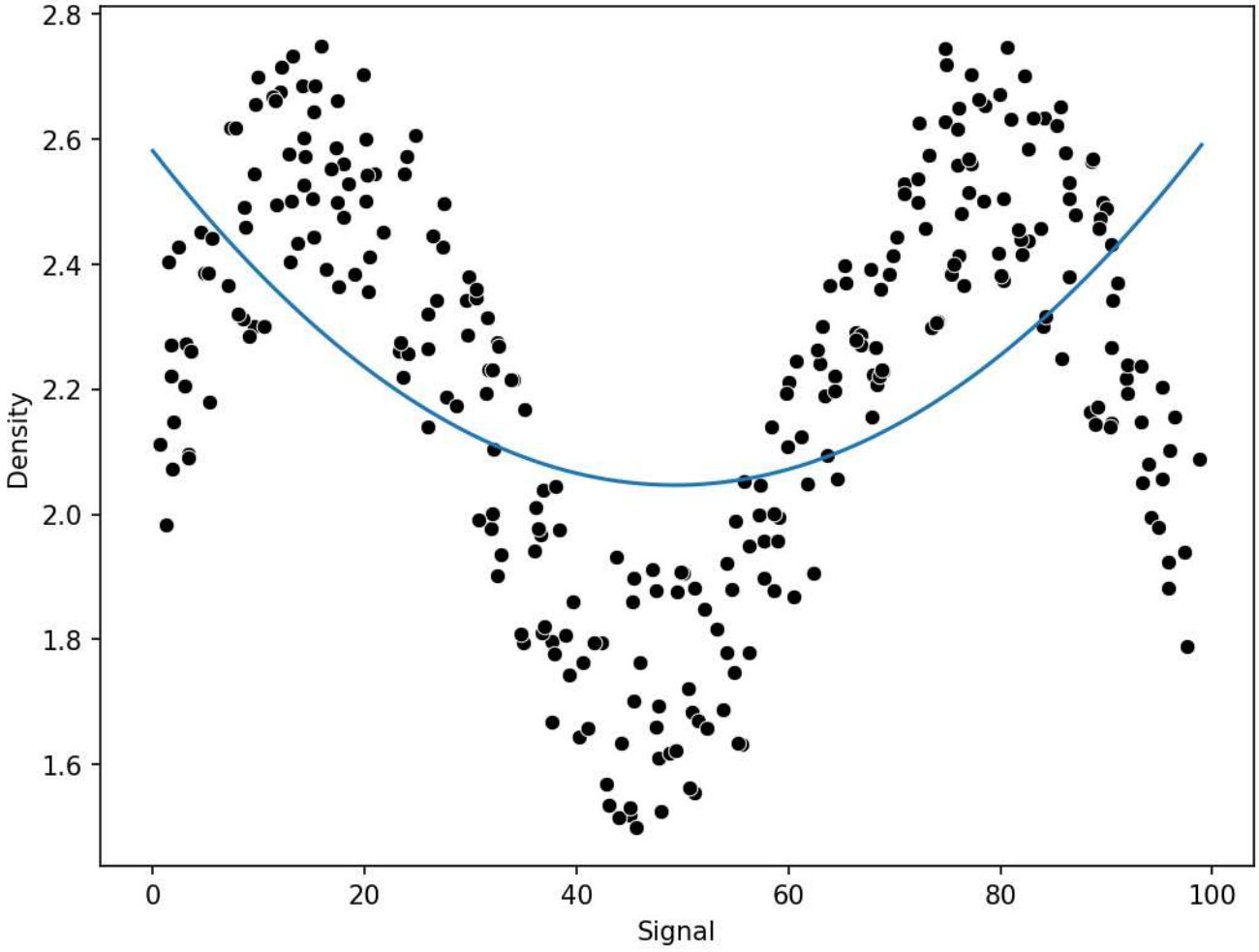
from sklearn.pipeline import make_pipeline

from sklearn.preprocessing import PolynomialFeatures

pipe = make_pipeline(PolynomialFeatures(2),LinearRegression())

run_model(pipe,X_train,y_train,X_test,y_test)

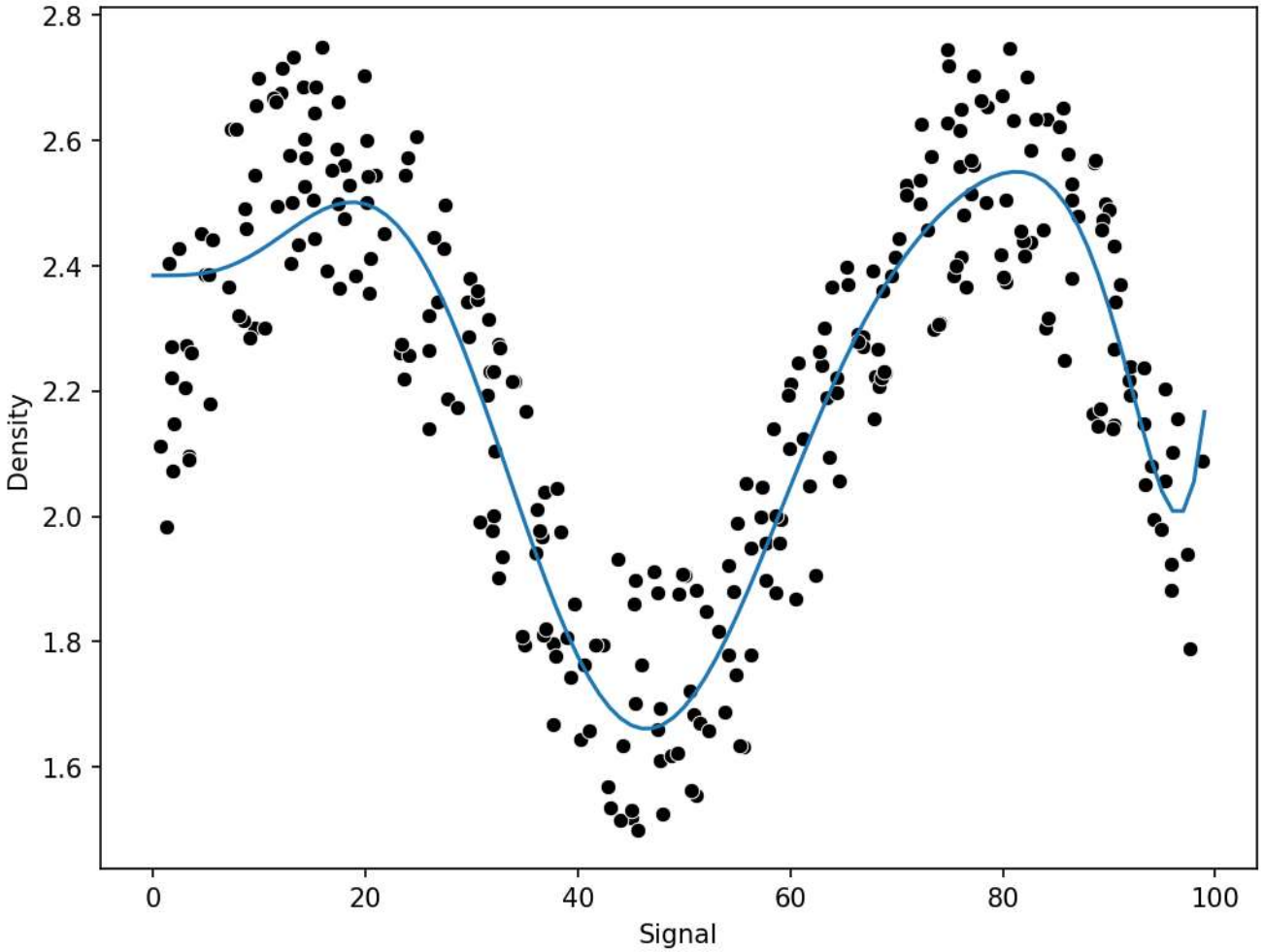
RMSE: 0.28173095637255835



Not quite a good fit even for Polynomial Method with degree=2. Let's try higher order. Maybe 10.

pipe = make_pipeline(PolynomialFeatures(10),LinearRegression())
run_model(pipe,X_train,y_train,X_test,y_test)

RMSE : 0.14035228495977833



Polynomial order of 10 degrees performs way better.

The magic of machine learning is that we can try to solve a given problem building and testing many different models. Some models perform well at certain type data, others data are good in other areas. Now let's try some other methods that we have not seen in the previous examples.

KNN Regression

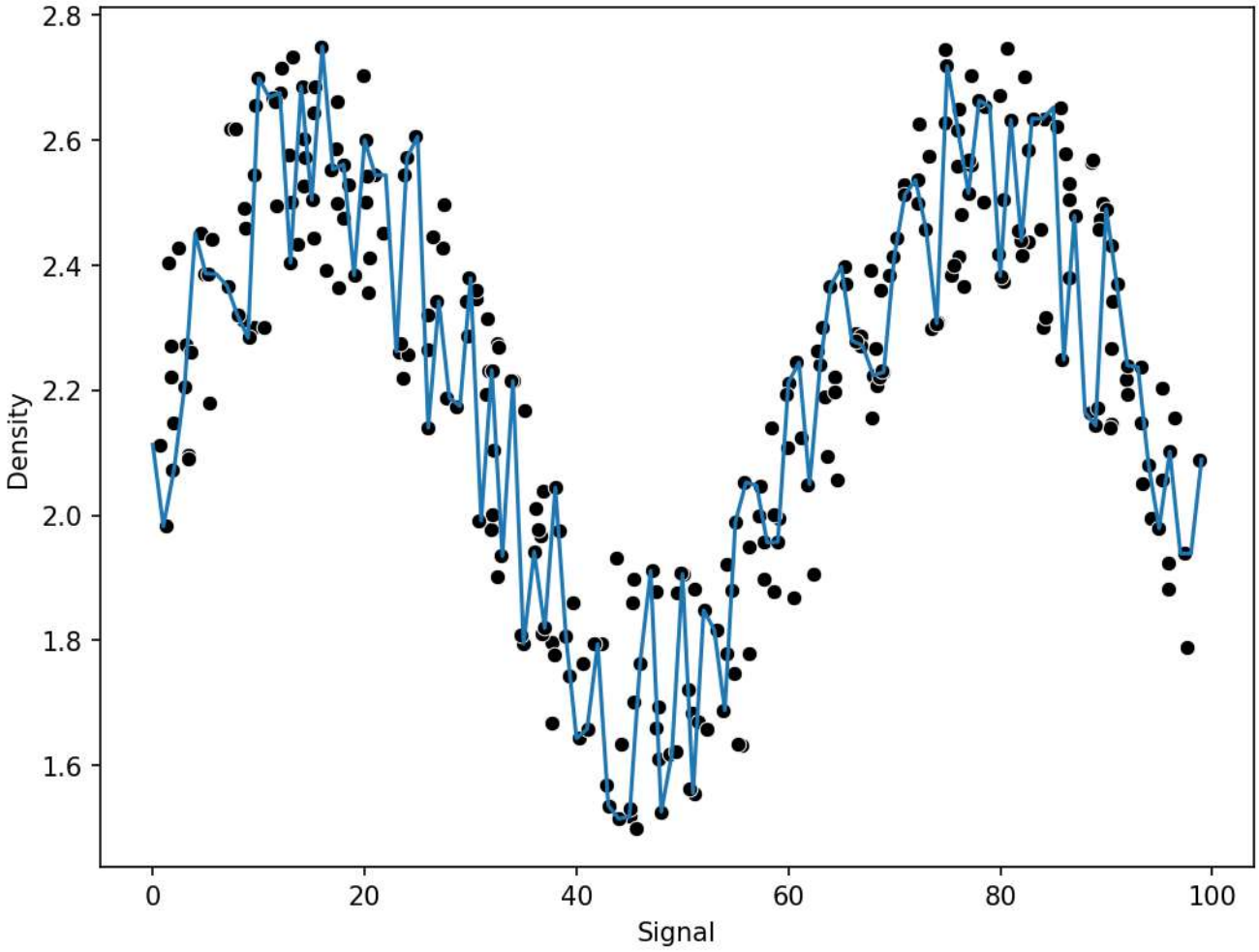
from sklearn.neighbors import KNeighborsRegressor

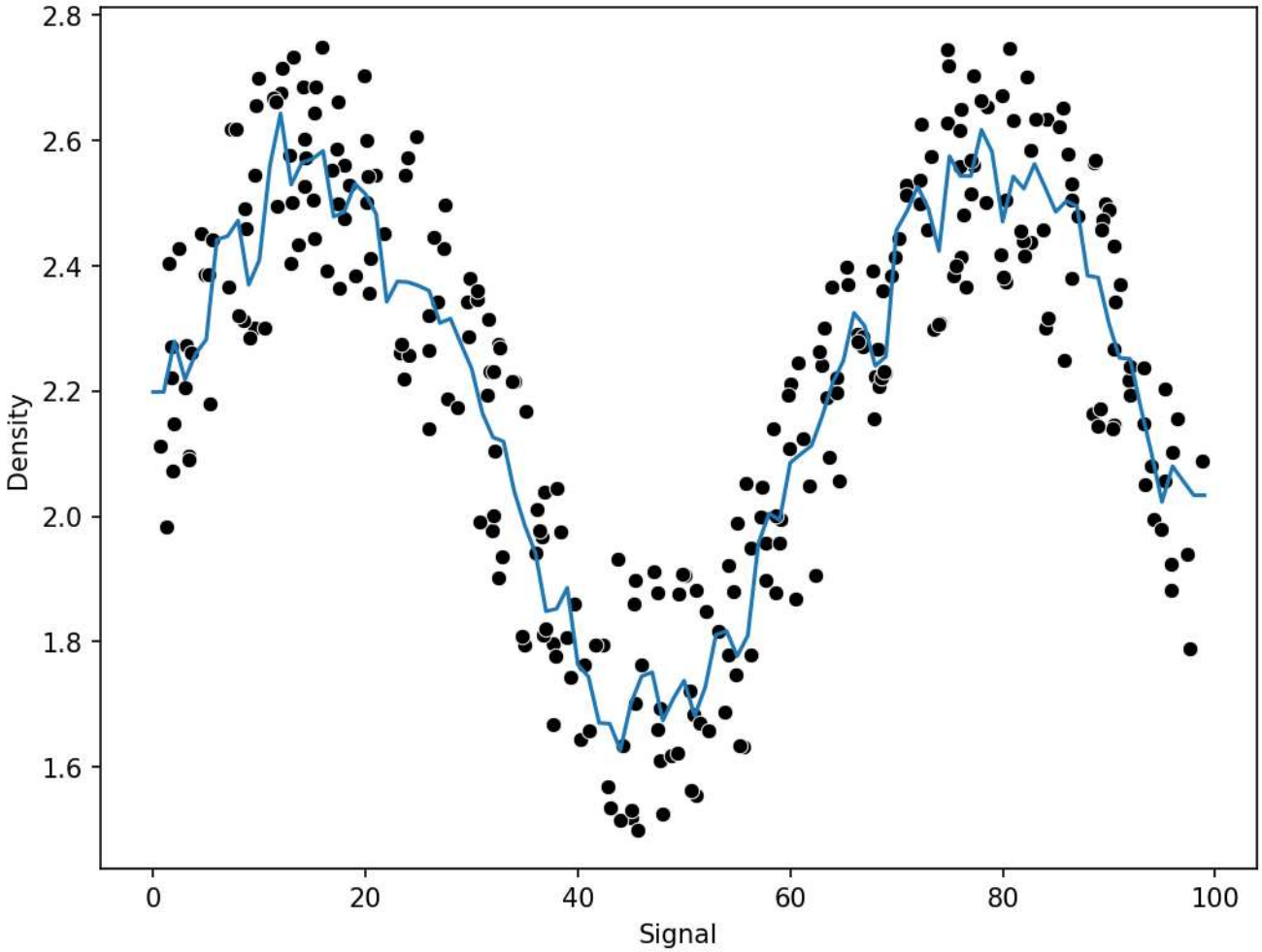
We will test the model for three different values of k.

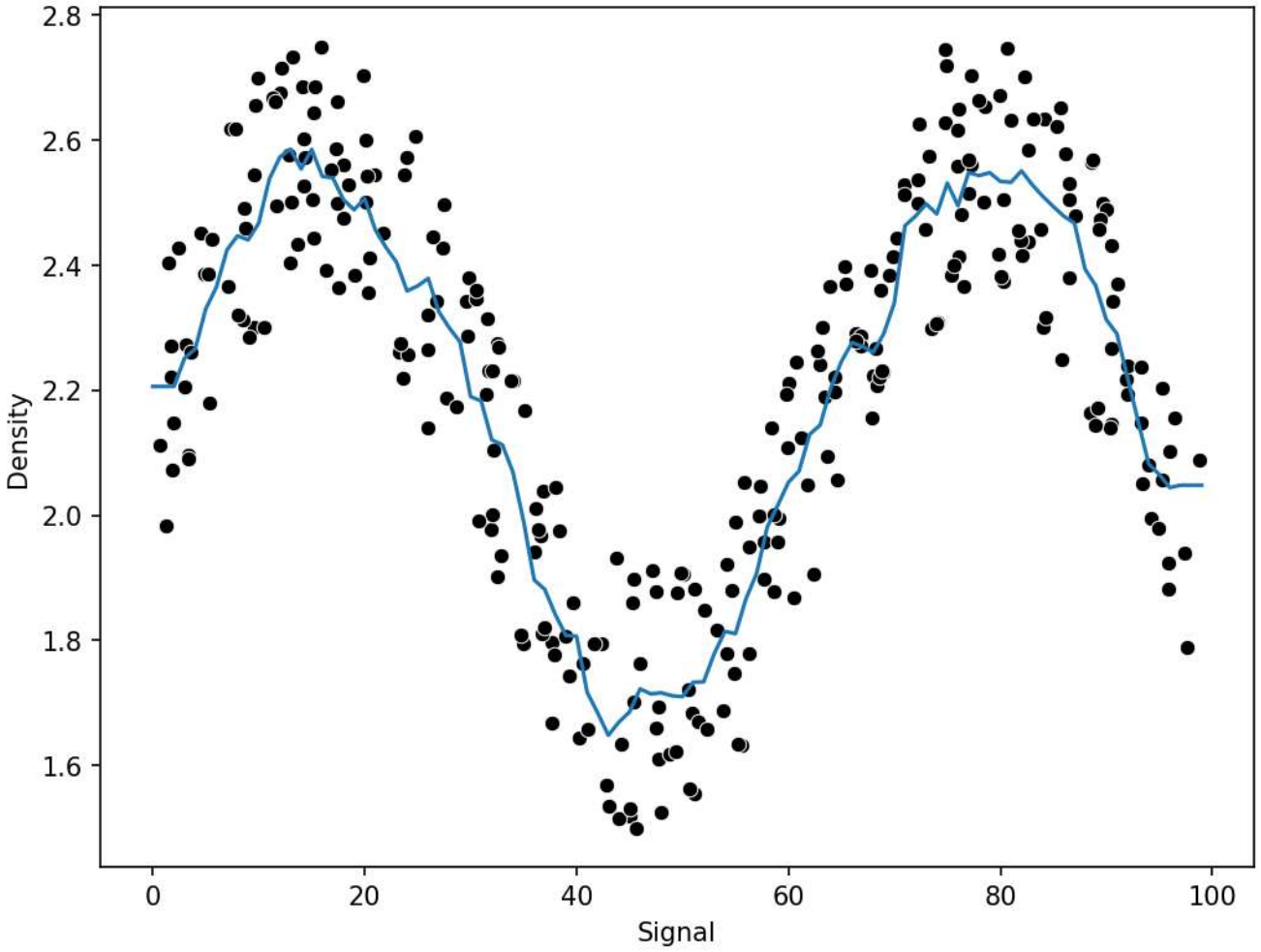
```
preds = {}
k_values = [1,5,10]
for n in k_values:

    model = KNeighborsRegressor(n_neighbors=n)
    run_model(model,X_train,y_train,X_test,y_test)
```

RMSE : 0.1523487028635337
RMSE : 0.13730685016923647
RMSE : 0.13277855732740926




2.8
2.6
2.4
2.2
2.0
1.8
1.6
Density
0
20
40
60
80
100
Signal



As we increase k from 1 to 5, then to 10, the curve becomes much smoother. Overall RMSE decreases, and the model does not try to fit to individual points.

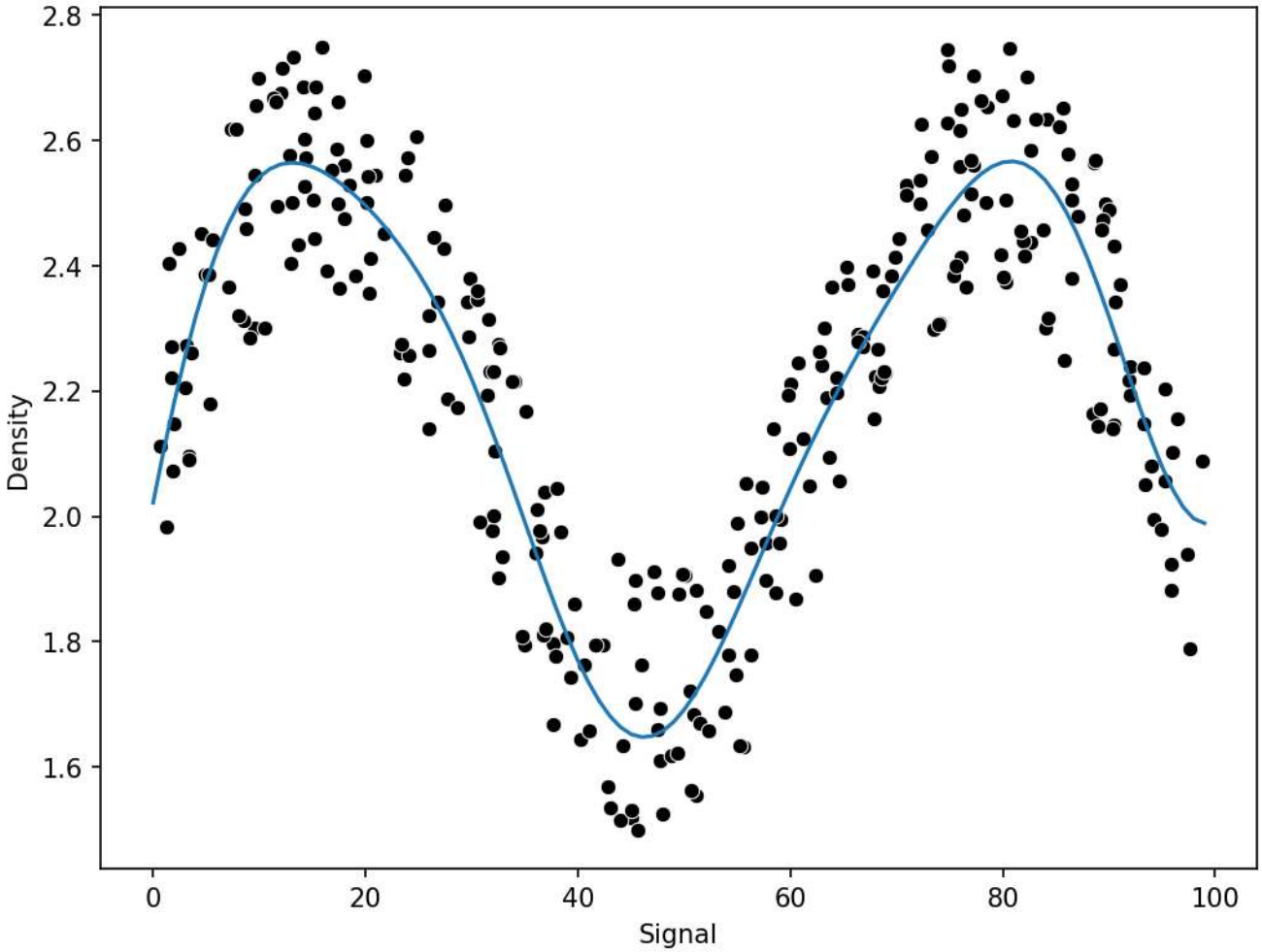
Support Vector Regression

from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV

In this model we will perform hyper parameter tuning and grid search.

param_grid = {'C':[0.01,0.1,1,5,10,100,1000],'gamma':['auto','scale']}
svr = SVR()
grid = GridSearchCV(svr,param_grid)
run_model(grid,X_train,y_train,X_test,y_test)

RMSE : 0.12646999302046696



Looks like the smallest RMSE 0.127 so far and the plot looks really well fitted to the over all trend.

And finally let's check out how a boosting of the Decision Tree Method will perform.

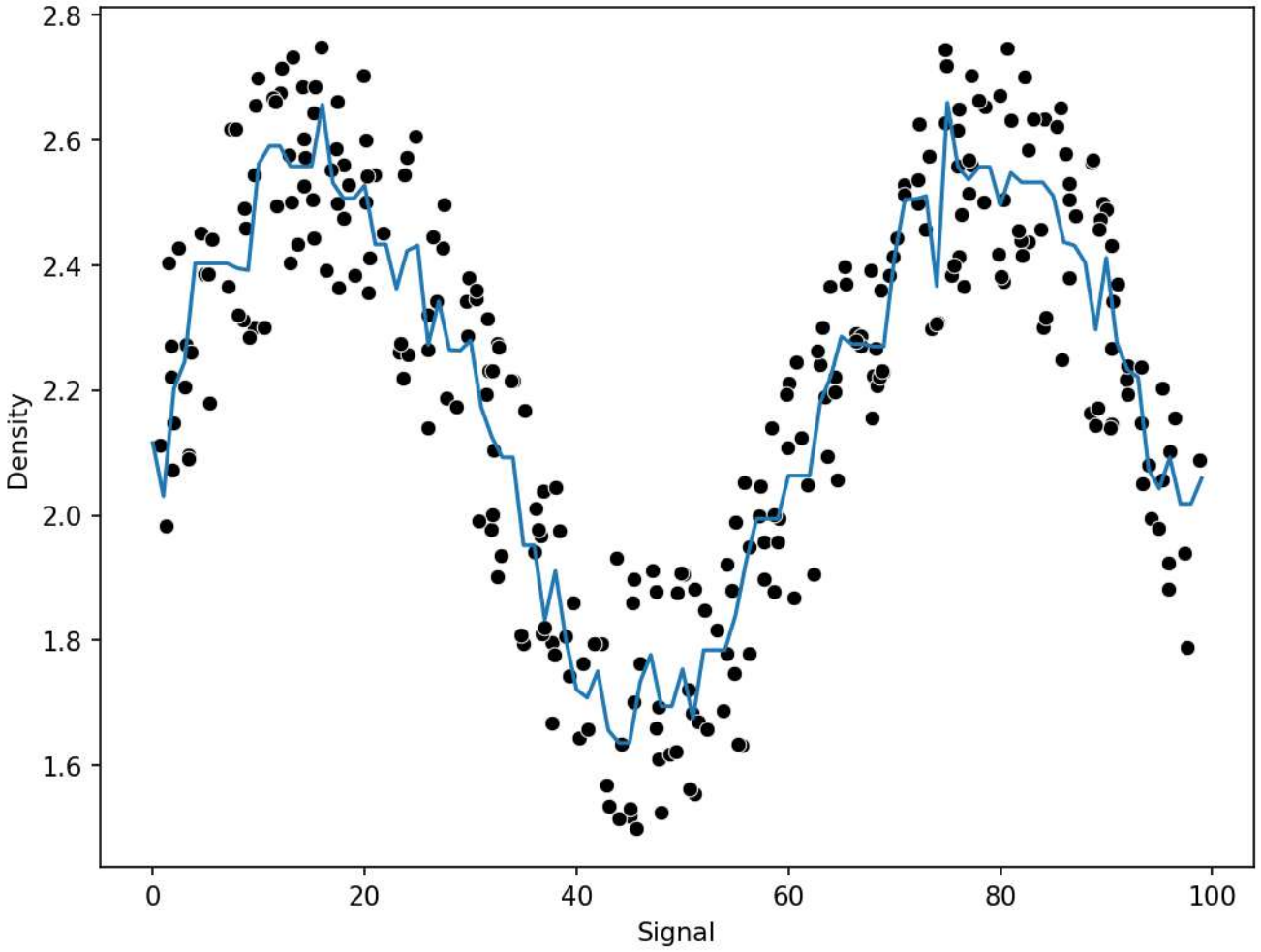
Gradient Boosting.

from sklearn.ensemble import GradientBoostingRegressor

ewfmodel = GradientBoostingRegressor()

run_model(model,X_train,y_train,X_test,y_test)

RMSE : 0.13294148649584667



This method also gives good results.