USA Housing Price Analysis practice

Main objective of the analysis

My main objective of this project is to create an reliable model to predict the price of the house by the given features.

Brief description of the data set you chose and a summary of its attributes.

The data set I chose is the USA housing price data set downloaded from Kaggle.The data set contains the following attributes:

```
'Avg. Area Income': Avg. Income of residents of the city house is
located in.
'Avg. Area House Age': Avg Age of Houses in same city
'Avg. Area Number of Rooms': Avg Number of Rooms for Houses in same
city
'Avg. Area Number of Bedrooms': Avg Number of Bedrooms for Houses in
same city
'Area Population': Population of city house is located in
'Price': Price that the house sold at
'Address': Address for the house
```

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")
%matplotlib inline
# read data
df=pd.read_csv('USA_Housing.csv')
print("*"*100)
print(df.info())
print("*"*100)
print(df.describe())
```

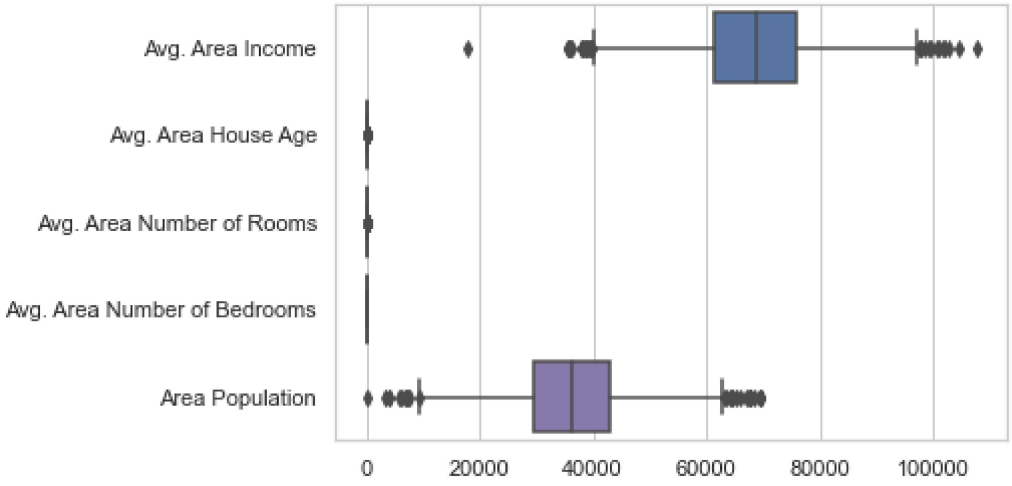
```
********************************************************************************
***************
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 7 columns):
 #   Column                       Non-Null Count  Dtype  
---  ------                       --------------  -----  
 0   Avg. Area Income             5000 non-null   float64
 1   Avg. Area House Age          5000 non-null   float64
 2   Avg. Area Number of Rooms    5000 non-null   float64
 3   Avg. Area Number of Bedrooms  5000 non-null   float64
 4   Area Population              5000 non-null   float64
 5   Price                        5000 non-null   float64
 6   Address                      5000 non-null   object 
dtypes: float64(6), object(1)
memory usage: 273.6+ KB
None
********************************************************************************
***************
       Avg. Area Income  Avg. Area House Age  Avg. Area Number of Rooms  \
count       5000.000000          5000.000000                5000.000000   
mean       68583.108984             5.977222                   6.987792   
std        10657.991214             0.991456                   1.005833   
min        17796.631190             2.644304                   3.236194   
25%        61480.562388             5.322283                   6.299250   
50%        68804.286404             5.970429                   7.002902   
75%        75783.338666             6.650808                   7.665871   
max       107701.748378             9.519088                  10.759588   

       Avg. Area Number of Bedrooms  Area Population         Price  
count                   5000.000000      5000.000000  5.000000e+03  
mean                       3.981330     36163.516039  1.232073e+06  
std                        1.234137      9925.650114  3.531176e+05  
min                        2.000000       172.610686  1.593866e+04  
25%                        3.140000     29403.928702  9.975771e+05  
50%                        4.050000     36199.406689  1.232669e+06  
75%                        4.490000     42861.290769  1.471210e+06  
max                        6.500000     69621.713378  2.469066e+06  
```

> **From the above analysis, we can see the data without any missing values.**
> **The target column will be Price and the rest will be the features.**

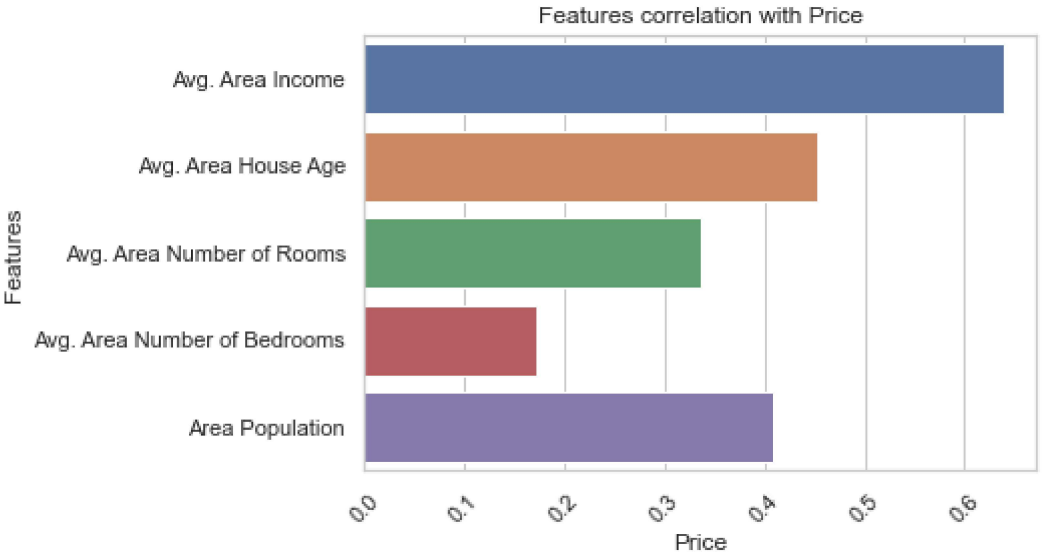
Brief summary of data exploration and actions taken for data cleaning and feature engineering.

In [68]:
```
# box plot
sns.boxplot(data=df.iloc[:,:-2],orient='h')
plt.show()
```



From the above analysis, we can see that the data is clean and there are no missing values, but the scale for numeric attributes is quite different and theoretically we need to scaling the data before we use them for model training.

In [69]:
```
df1=df.corr()[['Price']].reset_index()
df1.columns=['Features','Price']
df1=df1.iloc[:-1,:]
sns.barplot(y=df1['Features'],x=df1['Price'],orient='h')
# tilt x axis
plt.xticks(rotation=45)
plt.title('Features correlation with Price')
plt.show()
```



From this chart we know how big the influence of each attribute is on the price of the house.

Re-scale the data and then check the data distribution again.

Summary of training at least three linear regression models which should be variations that cover using a simple linear regression as a baseline, adding polynomial effects, and using a regularization regression. Preferably, all use the same training and test splits, or the same cross-validation method.

1. By using simple linear regression, I get the score of 0.918. After scaling the data, I get the same score. For Linear Regression, in terms of the raw predictions, scaling will not cause any influence.

r2_score = 0.918, MAE: 80879.097 MSE: 10089009300.895 RMSE: 100444.061

Features to be used:

'Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms', 'Avg. Area Number of Bedrooms', 'Area Population'

Target column:

'Price'

Simple linear_model compare scale vs not scale:

In [56]:
```
# scale data
MSE_dic={}
from sklearn.preprocessing import StandardScaler
features=['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population']
target='Price'
y=df['Price']
X=df[features]
scaler=StandardScaler()
X_scaled=scaler.fit_transform(X)
# sns.boxplot(data=pd.DataFrame(X_scaled,columns=features),orient='h')
# train test split
from sklearn.model_selection import train_test_split

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)

# import linear regression model
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score,mean_squared_error,mean_absolute_error
# train model on not scaled data
lr=LinearRegression()
```

```
lr.fit(X_train,y_train)
y_pred=lr.predict(X_test)
MSE_dic['Simple LR MSE(not scaled)']=mean_squared_error(y_test,y_pred)
print('*'*50+"Not scaled"+'*'*50)
print('r2_score:',r2_score(y_test,y_pred).round(3))
print('MAE:', mean_absolute_error(y_test, y_pred).round(3))
print('MSE:', mean_squared_error(y_test, y_pred).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test, y_pred)).round(3))
```

```
**************************************************Not scaled**********************
**********
r2_score: 0.918
MAE: 80879.097
MSE: 10089009300.895
RMSE: 100444.061
```

In [57]:
```
# train model on scaled data
X_train_s,X_test_s,y_train,y_test=train_test_split(X_scaled,y,test_size=0.2,random_sta
lr_s=LinearRegression()
lr_s.fit(X_train_s,y_train)
y_pred_s=lr_s.predict(X_test_s)
MSE_dic['Simple LR MSE(scaled)']=mean_squared_error(y_test,y_pred_s)
print('*'*50+"Scaled"+'*'*50)
print('r2_score:',r2_score(y_test,y_pred_s).round(3))
print('MAE:', mean_absolute_error(y_test, y_pred_s).round(3))
print('MSE:', mean_squared_error(y_test, y_pred_s).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test, y_pred_s)).round(3))
```

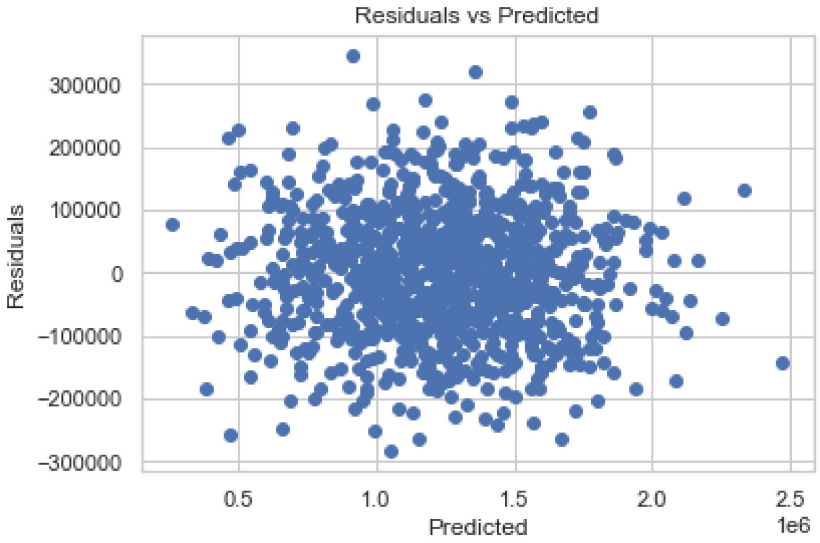
```
**************************************************Scaled**************************
**********
r2_score: 0.918
MAE: 80879.097
MSE: 10089009300.894
RMSE: 100444.061
```

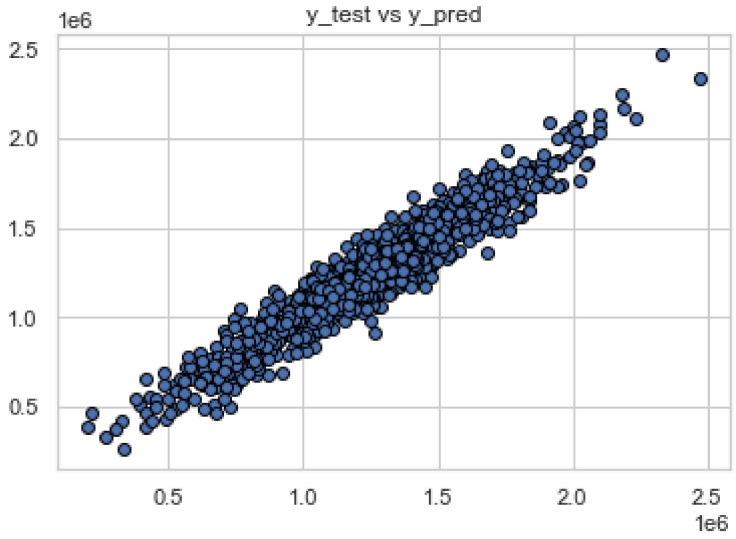
For Linear Regression it seems that scaling the data will not affect the raw predictions!

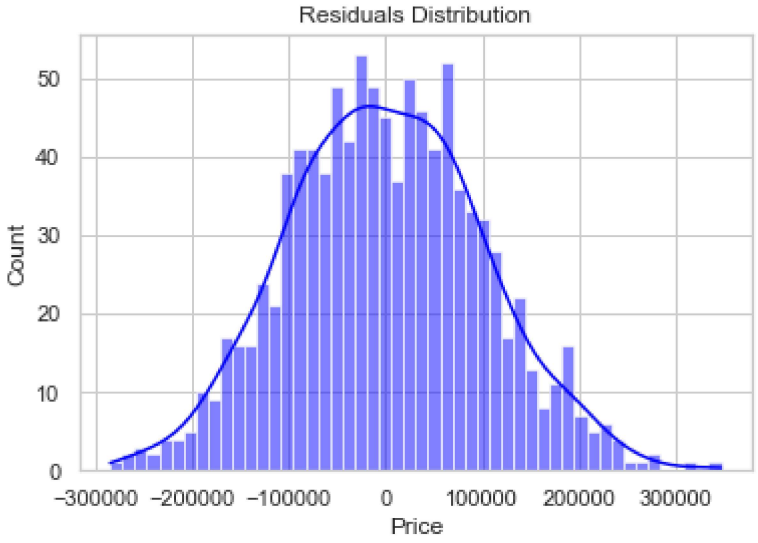
In [12]:
```
# plot residuals
plt.scatter(y_pred,y_test-y_pred)
plt.xlabel('Predicted')
plt.ylabel('Residuals')
plt.title('Residuals vs Predicted')
plt.show()

plt.scatter(y_test, y_pred, edgecolor='black')
plt.title('y_test vs y_pred')
plt.show()

sns.histplot((y_test - y_pred), bins = 50, kde=1,color='Blue')
plt.title('Residuals Distribution')
plt.show()
```







LASSO Regression L1 Regularization

In [73]:
```
# import lasso regression model
from sklearn.linear_model import LassoCV
alphas=[0.01,0.001,0.1,0.5,1,5,10,100,1000]
lasso=LassoCV(alphas=alphas,
```

```
                  max_iter=50000,
                  cv=3)
lasso.fit(X_train_s,y_train)
y_pred_s=lasso.predict(X_test_s)
MSE_dic['Lasso MSE(scaled)']=mean_squared_error(y_test,y_pred_s)
df_lasso_coef=pd.DataFrame(lasso.coef_,index=features,columns=['Coefficients'])
# print(lasso_coef)
print('*'*50+"LASSO_Regression_Scaled"+'*'*50)
print("alpha:",lasso.alpha_)
print('r2_score:',r2_score(y_test,y_pred_s).round(3))
print('MAE:', mean_absolute_error(y_test, y_pred_s).round(3))
print('MSE:', mean_squared_error(y_test, y_pred_s).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test, y_pred_s)).round(3))
```

```
**************************************************LASSO_Regression_Scaled**********
**********************************
alpha: 1.0
r2_score: 0.918
MAE: 80879.051
MSE: 10088999229.808
RMSE: 100444.01
```

In [59]:

```
# import lasso regression model
from sklearn.linear_model import LassoCV
alphas=[0.01,0.001,0.1,0.5,1,5,10,100,1000]
lasso=LassoCV(alphas=alphas,
                  max_iter=50000,
                  cv=3)
lasso.fit(X_train,y_train)
y_pred=lasso.predict(X_test)

MSE_dic['Lasso MSE(not scaled)']=mean_squared_error(y_test,y_pred)

print('*'*50+"LASSO_Regression_NOT Scaled"+'*'*50)
print("alpha:",lasso.alpha_)
print('r2_score:',r2_score(y_test,y_pred).round(3))
print('MAE:', mean_absolute_error(y_test, y_pred).round(3))
print('MSE:', mean_squared_error(y_test, y_pred).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test, y_pred)).round(3))
```

```
**************************************************LASSO_Regression_NOT Scaled********
****************************************
alpha: 100.0
r2_score: 0.918
MAE: 80885.373
MSE: 10090240529.054
RMSE: 100450.189
```

Using LASSO with alpha=0.1, I get the similar result comparing with simple linear regression model above.

LASSO Regression with polynomial effect

In [62]:

```
from sklearn.preprocessing import PolynomialFeatures
poly=PolynomialFeatures(degree=2, include_bias=False)
X_poly=poly.fit_transform(X_scaled)
X_train_poly,X_test_poly,y_train_poly,y_test_poly=train_test_split(X_poly,y,test_size=
alphas=[0.01,0.001,0.1,0.5,1,5,10,100,1000]
lass_poly=LassoCV(alphas=alphas,max_iter=50000,cv=3)
```

```
lass_poly.fit(X_train_poly,y_train_poly)
y_pred_poly=lass_poly.predict(X_test_poly)

MSE_dic['Lasso MSE(poly=2 scaled)']=mean_squared_error(y_test_poly,y_pred_poly)
print('*'*30+"LASSO Regression with polynomial effect"+'*'*30)
print("alpha:",lasso.alpha_)
print('r2_score:',r2_score(y_test_poly,y_pred_poly).round(3))
print('MAE:', mean_absolute_error(y_test_poly, y_pred_poly).round(3))
print('MSE:', mean_squared_error(y_test_poly, y_pred_poly).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test_poly, y_pred_poly)).round(3))
```

```
******************************LASSO Regression with polynomial effect**************
**************
alpha: 100.0
r2_score: 0.918
MAE: 80886.062
MSE: 10096966797.078
RMSE: 100483.664
```

Similar result with polynomial effect. The result is quite similar with simple linear regression model above. Even a little bit worse when we look at MAE, MSE, and RMSE.

Ridge Regression L2 Regularization

In [61]:
```
from sklearn.linear_model import RidgeCV
alphas=[0.01,0.001,0.1,0.5,1,5,10,100,1000]
ridge=RidgeCV(alphas=alphas,cv=3)
ridge.fit(X_train_s,y_train)
y_pred_ridge=ridge.predict(X_test_s)

MSE_dic['Ridge Regression MSE(scaled)']=mean_squared_error(y_test,y_pred_ridge)

print('*'*30+"Ridge Regression"+'*'*30)
print('r2_score:',r2_score(y_test,y_pred_ridge).round(3))
print('MAE:', mean_absolute_error(y_test, y_pred_ridge).round(3))
print('MSE:', mean_squared_error(y_test, y_pred_ridge).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test, y_pred_ridge)).round(3))
```

```
******************************Ridge Regression******************************
r2_score: 0.918
MAE: 80878.442
MSE: 10089005429.399
RMSE: 100444.041
```

In [63]:
```
from sklearn.linear_model import RidgeCV
alphas=[0.01,0.001,0.1,0.5,1,5,10,100,1000]
ridge=RidgeCV(alphas=alphas,cv=3)
ridge.fit(X_train_poly,y_train_poly)
y_pred_ridge=ridge.predict(X_test_poly)
MSE_dic['Ridge Regression MSE(poly=2 scaled)']=mean_squared_error(y_test_poly,y_pred_r
print('*'*30+"Ridge Regression with polynomial effect"+'*'*30)
print('r2_score:',r2_score(y_test_poly,y_pred_ridge).round(3))
print('MAE:', mean_absolute_error(y_test_poly, y_pred_ridge).round(3))
print('MSE:', mean_squared_error(y_test_poly, y_pred_ridge).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test_poly, y_pred_ridge)).round(3))
```

```
******************************Ridge Regression with polynomial effect***************
**************
r2_score: 0.918
MAE: 80884.109
MSE: 10099225432.696
RMSE: 100494.903
```

In [64]:
```
from sklearn.linear_model import ElasticNetCV

l1_ratios = np.linspace(0.1, 0.9, 9)
alphas=[0.01,0.001,0.1,0.5,1,5,10,100,1000]
elasticNetCV = ElasticNetCV(alphas=alphas,
                            l1_ratio=l1_ratios,
                            max_iter=50000).fit(X_train_s, y_train)
elasticNetCV.fit(X_train_s, y_train)
y_pred_ela=elasticNetCV.predict(X_test_s)
MSE_dic['ElasticNet Regression MSE(scaled)']=mean_squared_error(y_test_poly,y_pred_ric
print('*'*30+"ElasticNet Regression"+'*'*30)
print('L1 ratio:',elasticNetCV.l1_ratio_)
print('alphas:',elasticNetCV.alphas_)
print('r2_score:',r2_score(y_test,y_pred_ela).round(3))
print('MAE:', mean_absolute_error(y_test, y_pred_ela).round(3))
print('MSE:', mean_squared_error(y_test, y_pred_ela).round(3))
print('RMSE:', np.sqrt(mean_squared_error(y_test, y_pred_ela)).round(3))
```

```
******************************ElasticNet Regression******************************
L1 ratio: 0.6
alphas: [1.e+03 1.e+02 1.e+01 5.e+00 1.e+00 5.e-01 1.e-01 1.e-02 1.e-03]
r2_score: 0.918
MAE: 80877.0
MSE: 10089010528.303
RMSE: 100444.067
```

A paragraph explaining which of your regressions you recommend as a final model that best fits your needs in terms of accuracy and explainability.

From the models I've trained, I think the best model is Lasso Regression(alpha=1) on Scaled data which gives the lowest MSE value. The base model is simple linear regression model and also gives relative good result which is out of expectation. For the polynomial model, the higher the polynomial degree, the worse the MSE. The Ridge model has similar result with Lasso model.

In [69]:
```
sorted(MSE_dic.items(),key=lambda x: x[1])
```

Out[69]:
```
[('Lasso MSE(scaled)', 10088999229.807938),
 ('Ridge Regression MSE(scaled)', 10089005429.398758),
 ('Simple LR MSE(scaled)', 10089009300.893988),
 ('Simple LR MSE(not scaled)', 10089009300.894522),
 ('Lasso MSE(not scaled)', 10090240529.05382),
 ('Lasso MSE(poly scaled)', 10096966797.078465),
 ('Lasso MSE(poly=2 scaled)', 10096966797.078465),
 ('Ridge Regression MSE(poly=2 scaled)', 10099225432.69609),
 ('ElasticNet Regression MSE(scaled)', 10099225432.69609)]
```

Summary Key Findings and Insights, which walks your reader through the main drivers of your model and insights from your data derived from your linear regression model.

use the following formulas to get the weights of the model, since the features are scaled so the higher the Coefficient of the feature, the higher the weight of the feature.

df_lasso*coef=pd.DataFrame(lasso.coef*,index=features,columns=['Coefficients'])

So the Avg. Area Income has the highest weight, which has higher influence to the price of the house. And the House age comes the second. The Avg. Area Number of Bedrooms has the least influence to the price of the house.

In [76]:
```
df_lasso_coef.sort_values(by='Coefficients',ascending=False)
```

Out[76]:

	Coefficients
Avg. Area Income	230744.994452
Avg. Area House Age	163242.303534
Area Population	151551.584552
Avg. Area Number of Rooms	120309.749033
Avg. Area Number of Bedrooms	3010.465641

Suggestions for next steps in analyzing this data, which may include suggesting revisiting this model adding specific data features to achieve a better explanation or a better prediction.

To get deeper understanding of the data and the problem. Take "Address" for example, we may re-construct the feature by extracting the city and state from the address. As the geo-location might also be an important feature. And also the birth rate of the city could also be an important feature.