10 Linear Regression Algorithms with Python

> Linear Regression is the most straightforward algorithm in machine learning, it can be trained differently. In this notebook we will cover the following linear algorithms:
>
> 1. Linear Regression
> 2. Robust Regression
> 3. Ridge Regression
> 4. LASSO Regression
> 5. Elastic Net
> 6. Polynomial Regression
> 7. Stochastic Gradient Descent
> 8. Artificial Neural Networks
> 9. Random Forest Regressor
> 10. Support Vector Machine

Data

> We are going to use the `USA_Housing` dataset. Since house price is a continuous variable, this is a regression problem. The data contains the following columns:
>
> - '`Avg. Area Income`': Avg. The income of residents of the city house is located in.
> - '`Avg. Area House Age`': Avg Age of Houses in the same city
> - '`Avg. Area Number of Rooms`': Avg Number of Rooms for Houses in the same city
> - '`Avg. Area Number of Bedrooms`': Avg Number of Bedrooms for Houses in the same city
> - '`Area Population`': The population of city house is located in
> - '`Price`': Price that the house sold at
> - '`Address`': Address for the house

In [16]:
```
!pip install -q hvplot
```

Import Libraries

In [17]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import hvplot.pandas

%matplotlib inline

# sns.set_style("whitegrid")
# plt.style.use("fivethirtyeight")
```

Check out the Data

In [18]:
```
USAhousing = pd.read_csv('/input/usa-housing/USA_Housing.csv')
USAhousing.head()
```

Out[18]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	Address
0	79545.458574	5.682861	7.009188	4.09	23086.800503	1.059034e+06	208 Michael Ferry Apt. 674\nLaurabury, NE 3701...
1	79248.642455	6.002900	6.730821	3.09	40173.072174	1.505891e+06	188 Johnson Views Suite 079\nLake Kathleen, CA...
2	61287.067179	5.865890	8.512727	5.13	36882.159400	1.058988e+06	9127 Elizabeth Stravenue\nDanieltown, WI 06482...
3	63345.240046	7.188236	5.586729	3.26	34310.242831	1.260617e+06	USS Barnett\nFPO AP 44820
4	59982.197226	5.040555	7.839388	4.23	26354.109472	6.309435e+05	USNS Raymond\nFPO AE 09386

In [19]:
```
USAhousing.info()
```

```
<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
```

In [20]:
```
USAhousing.describe()
```

Out[20]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5.000000e+03
mean	68583.108984	5.977222	6.987792	3.981330	36163.516039	1.232073e+06
std	10657.991214	0.991456	1.005833	1.234137	9925.650114	3.531176e+05
min	17796.631190	2.644304	3.236194	2.000000	172.610686	1.593866e+04
25%	61480.562388	5.322283	6.299250	3.140000	29403.928702	9.975771e+05
50%	68804.286404	5.970429	7.002902	4.050000	36199.406689	1.232669e+06
75%	75783.338666	6.650808	7.665871	4.490000	42861.290769	1.471210e+06
max	107701.748378	9.519088	10.759588	6.500000	69621.713378	2.469066e+06

In [21]:
```
USAhousing.columns
```

Out[21]:
```
Index(['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population', 'Price', 'Address'],
      dtype='object')
```

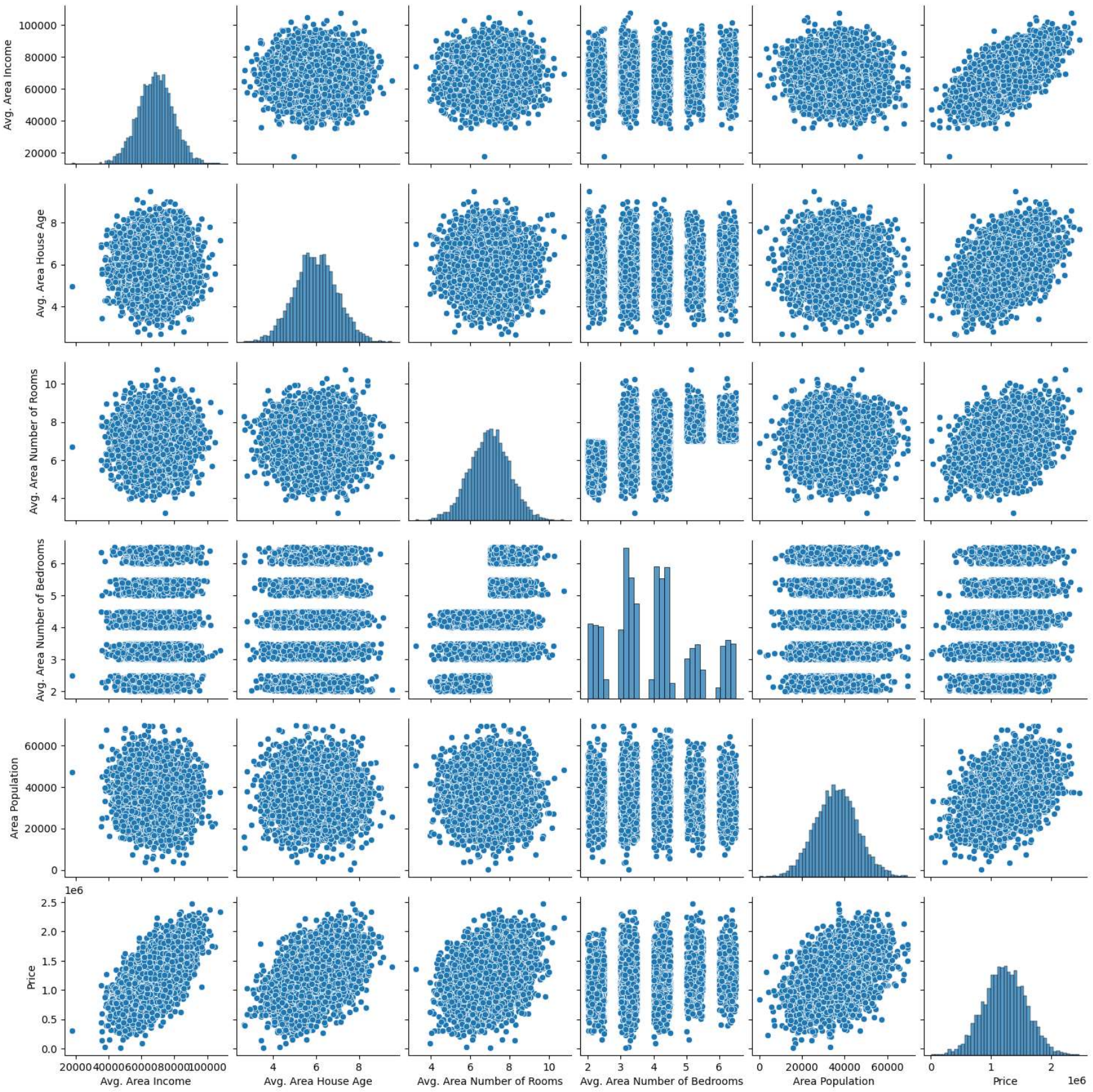
Exploratory Data Analysis (EDA)

Let's create some simple plots to check out the data

In [22]:
```
sns.pairplot(USAhousing)
```

```
/opt/conda/lib/python3.10/site-packages/seaborn/axisgrid.py:118: UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

Out[22]:
```
<seaborn.axisgrid.PairGrid at 0x7cdb0961aa10>
```



In [23]:
```
USAhousing.hvplot.hist(by='Price', subplots=False, width=1000)
```

Out[23]:

In [24]:
```
USAhousing.hvplot.hist("Price")
```

Out[24]:

In [25]:
```
USAhousing.hvplot.scatter(x='Avg. Area House Age', y='Price')
```

Out[25]:

In [26]:
```
USAhousing.hvplot.scatter(x='Avg. Area Income', y='Price')
```

Out[26]:

In [27]:
```
USAhousing.columns
```

Out[27]:
```
Index(['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population', 'Price', 'Address'],
      dtype='object')
```

Training a Linear Regression Model

> We will need to first split up our data into an X array that contains the features to train on, and a y array with the target variable, in this case, the Price column. We will toss out the Address column because it only has text info that the linear regression model can't use.

X and y arrays

In [28]:
```
X = USAhousing[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
                'Avg. Area Number of Bedrooms', 'Area Population']]
y = USAhousing['Price']
```

Train Test Split

Now, let's split the data into a training set and a testing set. We will train our model on the training set and then use the test set to evaluate the model.

In [29]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

In [30]:
```
from sklearn import metrics
from sklearn.model_selection import cross_val_score

def cross_val(model):
    pred = cross_val_score(model, X, y, cv=10)
    return pred.mean()

def print_evaluate(true, predicted):
    mae = metrics.mean_absolute_error(true, predicted)
    mse = metrics.mean_squared_error(true, predicted)
    rmse = np.sqrt(metrics.mean_squared_error(true, predicted))
    r2_square = metrics.r2_score(true, predicted)
    print('MAE:', mae)
    print('MSE:', mse)
    print('RMSE:', rmse)
    print('R2 Square', r2_square)
    print('__________________________________')

def evaluate(true, predicted):
    mae = metrics.mean_absolute_error(true, predicted)
    mse = metrics.mean_squared_error(true, predicted)
    rmse = np.sqrt(metrics.mean_squared_error(true, predicted))
    r2_square = metrics.r2_score(true, predicted)
    return mae, mse, rmse, r2_square
```

Preparing Data For Linear Regression

> Try different preparations of your data using these heuristics and see what works best for your problem.
>
> - **Linear Assumption.** Linear regression assumes a linear relationship between your input and output. It does not support any other type of relationship. While this may seem obvious, it's important to remember, especially when dealing with numerous attributes. In such cases, you might need to transform the data to establish a linear relationship, for instance, by applying a log transform for data exhibiting an exponential relationship.
> - **Remove Noise.** Linear regression assumes that your input and output variables are not noisy. Consider using data cleaning operations that let you better expose and clarify the signal in your data. This is most important for the output variable and you want to remove outliers in the output variable (y) if possible.
> - **Remove Collinearity.** Linear regression will over-fit your data when you have highly correlated input variables. Consider calculating pairwise correlations for your input data and removing the most correlated.
> - **Gaussian Distributions.** Linear regression will make more reliable predictions if your input and output variables have a Gaussian distribution. You may get some benefit using transforms (e.g. log or BoxCox) on your variables to make their distribution more Gaussian looking.
> - **Rescale Inputs:** Linear regression will often make more reliable predictions if you rescale input variables using standardization or normalization.

In [31]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('std_scalar', StandardScaler())
])

X_train = pipeline.fit_transform(X_train)
X_test = pipeline.transform(X_test)
```

Linear Regression

In [32]:
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X_train,y_train)
```

Out[32]:
```
LinearRegression()
```

Model Evaluation

Let's evaluate the model by checking out it's coefficients and how we can interpret them.

In [33]:
```
# print the intercept
print(lin_reg.intercept_)
```

```
1228219.1492415662
```

In [34]:
```
coeff_df = pd.DataFrame(lin_reg.coef_, X.columns, columns=['Coefficient'])
coeff_df
```

Out[34]:

	Coefficient
Avg. Area Income	232679.724643
Avg. Area House Age	163841.046593
Avg. Area Number of Rooms	121110.555478
Avg. Area Number of Bedrooms	2892.815119
Area Population	151252.342377

> Interpreting the coefficients:
>
> - Holding all other features fixed, a 1 unit increase in **Avg. Area Income** is associated with an **increase of $23.26.**
> - Holding all other features fixed, a 1 unit increase in **Avg. Area House Age** is associated with an **increase of $163841.**
> - Holding all other features fixed, a 1 unit increase in **Avg. Area Number of Rooms** is associated with an **increase of $121110.**
> - Holding all other features fixed, a 1 unit increase in **Avg. Area Number of Bedrooms** is associated with an **increase of $2892.81**.
> - Holding all other features fixed, a 1 unit increase in **Area Population** is associated with an **increase of $15.12.**

Does this make sense? Probably not because I made up this data.

Predictions from our Model

Let's grab predictions off our test set and see how well it did

In [35]:
```
pred = lin_reg.predict(X_test)
```

In [36]:
```
pd.DataFrame({'True Values': y_test, 'Predicted Values': pred}).hvplot.scatter(x='True Values', y='Predicted Values')
```

Out[36]:

Residual Histogram

In [37]:
```
pd.DataFrame({'Error Values': (y_test - pred)}).hvplot.kde()
```

Out[37]:

Regression Evaluation Metrics

Here are three common evaluation metrics for regression problems:

> - **Mean Absolute Error** (MAE) is the mean of the absolute value of the errors:
>
> $$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$
>
> - **Mean Squared Error** (MSE) is the mean of the squared errors:
>
> $$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$
>
> - **Root Mean Squared Error** (RMSE) is the square root of the mean of the squared errors:
>
> $$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$
>
> Comparing these metrics:
>
> - **MAE** is the easiest to understand, because it's the average error.
> - **MSE** is more popular than MAE, because MSE "punishes" larger errors, which tends to be useful in the real world.
> - **RMSE** is even more popular than MSE, because RMSE is interpretable in the "y" units.
>
> All of these are **loss functions**, because we want to minimize them.

In [38]:
```
test_pred = lin_reg.predict(X_test)
train_pred = lin_reg.predict(X_train)
```

```
print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)
print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df = pd.DataFrame(data=[["Linear Regression", *evaluate(y_test, test_pred) , cross_val(LinearRegression())]],
                          columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', "Cross Validation"])
```

```
Test set evaluation:
_____________________________________
MAE: 81135.56609336878
MSE: 10068422551.40088
RMSE: 100341.52954485436
R2 Square 0.9146818498754016
_____________________________________
Train set evaluation:
_____________________________________
MAE: 81480.49973174892
MSE: 10287043161.197224
RMSE: 101425.06180031257
R2 Square 0.9192986579075526
_____________________________________
```

Robust Regression

> Robust regression is a form of regression analysis designed to overcome some limitations of traditional parametric and non-parametric methods. Robust regression methods are designed to be not overly affected by violations of assumptions by the underlying data-generating process.
>
> One instance in which robust estimation should be considered is when there is a strong suspicion of `heteroscedasticity`.
>
> A common situation in which robust estimation is used occurs when the data contain outliers. In the presence of outliers that do not come from the same data-generating process as the rest of the data, least squares estimation is inefficient and can be biased. Because the least squares predictions are dragged towards the outliers, and because the variance of the estimates is artificially inflated, the result is that outliers can be masked. (In many situations, including some areas of geostatistics and medical statistics, it is precisely the outliers that are of interest.)

Random Sample Consensus - RANSAC

> Random sample consensus (`RANSAC`) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. Therefore, it also can be interpreted as an outlier detection method.
>
> A basic assumption is that the data consists of "inliers", i.e., data whose distribution can be explained by some set of model parameters, though may be subject to noise, and "outliers" which are data that do not fit the model. The outliers can come, for example, from extreme values of the noise or from erroneous measurements or incorrect hypotheses about the interpretation of data. RANSAC also assumes that, given a (usually small) set of inliers, there exists a procedure which can estimate the parameters of a model that optimally explains or fits this data.

In [45]:

```
from sklearn.linear_model import RANSACRegressor

model = RANSACRegressor(base_estimator=LinearRegression(), max_trials=100)
model.fit(X_train, y_train)

test_pred = model.predict(X_test)
train_pred = model.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)
print('====================================')
print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Robust Regression", *evaluate(y_test, test_pred) , cross_val(RANSACRegressor())]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', "Cross Validation"])
```

```
/opt/conda/lib/python3.10/site-packages/sklearn/linear_model/_ransac.py:343: FutureWarning: `base_estimator` was renamed to `estimator` in version 1.1 and will be removed in 1.3.
  warnings.warn(
Test set evaluation:
_____________________________________
MAE: 82029.41475274497
MSE: 10283624826.83934
RMSE: 101408.20887304607
R2 Square 0.9128582613292039
_____________________________________
====================================
Train set evaluation:
_____________________________________
MAE: 82712.18012372882
MSE: 10599060832.325783
RMSE: 102951.74030741677
R2 Square 0.9168508947921401
_____________________________________
```

Ridge Regression

> Ridge regression addresses some of the problems of **Ordinary Least Squares** by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares,

$$\min_w ||Xw - y||_2^2 + \alpha||w||_2^2$$

> $\alpha >= 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.
>
> Ridge regression is an L2 penalized model. Add the squared sum of the weights to the least-squares cost function.

In [46]:
```
from sklearn.linear_model import Ridge

model = Ridge(alpha=100, solver='cholesky', tol=0.0001, random_state=42)
model.fit(X_train, y_train)
pred = model.predict(X_test)

test_pred = model.predict(X_test)
train_pred = model.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)
print('==================================')
print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Ridge Regression", *evaluate(y_test, test_pred) , cross_val(Ridge())]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', "Cross Validation"])
```

```
Test set evaluation:

_____________________________________
MAE: 81428.64835535336
MSE: 10153269900.892612
RMSE: 100763.43533689497
R2 Square 0.9139628674464607

_____________________________________
==================================
Train set evaluation:

_____________________________________
MAE: 81972.39058585512
MSE: 10382929615.14346
RMSE: 101896.66145239233
R2 Square 0.9185464334441484

_____________________________________
```

LASSO Regression

> A linear model that estimates sparse coefficients.
>
> Mathematically, it consists of a linear model trained with ℓ_1 prior as regularizer. The objective function to minimize is:

$$\min_w \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha||w||_1$$

> The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha||w||_1$ added, where α is a constant and $||w||_1$ is the $\ell_1 - norm$ of the parameter vector.

In [48]:
```
from sklearn.linear_model import Lasso

model = Lasso(alpha=0.1,
              precompute=True,
#              warm_start=True,
              positive=True,
              selection='random',
              random_state=42)
model.fit(X_train, y_train)

test_pred = model.predict(X_test)
train_pred = model.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)
print('==================================')
print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Lasso Regression", *evaluate(y_test, test_pred) , cross_val(Lasso())]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', "Cross Validation"])
```

```
Test set evaluation:
_____________________________________
MAE: 81135.69851726218
MSE: 10068453390.364521
RMSE: 100341.68321472648
R2 Square 0.914681588551116

_____________________________________
==================================
Train set evaluation:
_____________________________________
MAE: 81480.63002185506
MSE: 10287043196.634295
RMSE: 101425.0619750084
R2 Square 0.9192986576295505

_____________________________________
```

Elastic Net

> A linear regression model trained with L1 and L2 prior as regularizer. This combination allows for learning a sparse model where few of the weights are non-zero like Lasso, while still maintaining the regularization properties of Ridge. Elastic-net is useful when there are multiple features which are correlated with one another. Lasso is likely to pick one of these at random, while elastic-net is likely to pick both. A practical advantage of trading-off between Lasso and Ridge is it allows Elastic-Net to inherit some of Ridge's stability under rotation. The objective function to minimize is in this case

$$\min_w \frac{1}{2n_{samples}} ||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2$$

In [49]:
```python
from sklearn.linear_model import ElasticNet

model = ElasticNet(alpha=0.1, l1_ratio=0.9, selection='random', random_state=42)
model.fit(X_train, y_train)

test_pred = model.predict(X_test)
train_pred = model.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)
print('==================================')
print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Elastic Net Regression", *evaluate(y_test, test_pred) , cross_val(ElasticNet())]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', "Cross Validation"])
```

```
Test set evaluation:
_____________________________________
MAE: 81184.43147330944
MSE: 10078050168.470106
RMSE: 100389.49232100991
R2 Square 0.9146002670381437

_____________________________________
==================================
Train set evaluation:
_____________________________________
MAE: 81577.88831531754
MSE: 10299274948.101461
RMSE: 101485.34351373829
R2 Square 0.9192027001474953

_____________________________________
```

Polynomial Regression

> One common pattern within machine learning is to use linear models trained on nonlinear functions of the data. This approach maintains the generally fast performance of linear methods, while allowing them to fit a much wider range of data.
>
> For example, a simple linear regression can be extended by constructing polynomial features from the coefficients. In the standard linear regression case, you might have a model that looks like this for two-dimensional data:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2$$

> If we want to fit a paraboloid to the data instead of a plane, we can combine the features in second-order polynomials, so that the model looks like this:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

> The (sometimes surprising) observation is that this is still a linear model: to see this, imagine creating a new variable

$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

> With this re-labeling of the data, our problem can be written

$$\hat{y}(w, x) = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5$$

> We see that the resulting polynomial regression is in the same class of linear models we'd considered above (i.e. the model is linear in w) and can be solved by the same techniques. By considering linear fits within a higher-dimensional

> space built with these basis functions, the model has the flexibility to fit a much broader range of data.

In [52]:
```
from sklearn.preprocessing import PolynomialFeatures

poly_reg = PolynomialFeatures(degree=2)

X_train_2_d = poly_reg.fit_transform(X_train)
X_test_2_d = poly_reg.transform(X_test)

lin_reg = LinearRegression()
lin_reg.fit(X_train_2_d,y_train)

test_pred = lin_reg.predict(X_test_2_d)
train_pred = lin_reg.predict(X_train_2_d)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)
print('==================================')
print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Polynomail Regression", *evaluate(y_test, test_pred), 0]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', 'Cross Validation'])
```

```
Test set evaluation:
_____________________________________
MAE: 81174.51844119694
MSE: 10081983997.620693
RMSE: 100409.08324260656
R2 Square 0.914566932419506
_____________________________________
==================================
Train set evaluation:
_____________________________________
MAE: 81363.0618562117
MSE: 10266487151.007816
RMSE: 101323.67517519198
R2 Square 0.9194599187853729
_____________________________________
```

Stochastic Gradient Descent

> Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Sescent is to tweak parameters iteratively in order to minimize a cost function. Gradient Descent measures the local gradient of the error function with regards to the parameters vector, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum.

In [53]:
```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(n_iter_no_change=250, penalty=None, eta0=0.0001, max_iter=100000)
sgd_reg.fit(X_train, y_train)

test_pred = sgd_reg.predict(X_test)
train_pred = sgd_reg.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)
print('==================================')
print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Stochastic Gradient Descent", *evaluate(y_test, test_pred), 0]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', 'Cross Validation'])
```

```
Test set evaluation:
_____________________________________
MAE: 81135.56653706348
MSE: 10068422540.702381
RMSE: 100341.52949154393
R2 Square 0.914681849966059
_____________________________________
==================================
Train set evaluation:
_____________________________________
MAE: 81480.49988440325
MSE: 10287043161.199207
RMSE: 101425.06180032235
R2 Square 0.919298657907537
_____________________________________
```

Artficial Neural Network

In [54]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Activation, Dropout
from tensorflow.keras.optimizers import Adam

X_train = np.array(X_train)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

```
model = Sequential()

model.add(Dense(X_train.shape[1], activation='relu'))
model.add(Dense(32, activation='relu'))
# model.add(Dropout(0.2))

model.add(Dense(64, activation='relu'))
# model.add(Dropout(0.2))

model.add(Dense(128, activation='relu'))
# model.add(Dropout(0.2))

model.add(Dense(512, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(1))

model.compile(optimizer=Adam(0.00001), loss='mse')

r = model.fit(X_train, y_train,
              validation_data=(X_test,y_test),
              batch_size=1,
              epochs=100)
```

```
Epoch 1/100
3500/3500 [==============================] - 17s 3ms/step - loss: 1635959308288.0000 - val_loss: 1658109558784.0000
Epoch 2/100
3500/3500 [==============================] - 11s 3ms/step - loss: 1635207741440.0000 - val_loss: 1656222253056.0000
Epoch 3/100
3500/3500 [==============================] - 11s 3ms/step - loss: 1630735171584.0000 - val_loss: 1647849373696.0000
Epoch 4/100
3500/3500 [==============================] - 11s 3ms/step - loss: 1615667134464.0000 - val_loss: 1623587160064.0000
Epoch 5/100
3500/3500 [==============================] - 11s 3ms/step - loss: 1578034659328.0000 - val_loss: 1568294567936.0000
Epoch 6/100
3500/3500 [==============================] - 11s 3ms/step - loss: 1500520251392.0000 - val_loss: 1461720580096.0000
Epoch 7/100
3500/3500 [==============================] - 11s 3ms/step - loss: 1362529615872.0000 - val_loss: 1283692560384.0000
Epoch 8/100
3500/3500 [==============================] - 12s 3ms/step - loss: 1151140757504.0000 - val_loss: 1031173898240.0000
Epoch 9/100
3500/3500 [==============================] - 11s 3ms/step - loss: 884213612544.0000 - val_loss: 744122023936.0000
Epoch 10/100
3500/3500 [==============================] - 11s 3ms/step - loss: 632530993152.0000 - val_loss: 522820157440.0000
Epoch 11/100
3500/3500 [==============================] - 12s 3ms/step - loss: 476224061440.0000 - val_loss: 410893287424.0000
Epoch 12/100
3500/3500 [==============================] - 11s 3ms/step - loss: 408242946048.0000 - val_loss: 364421283840.0000
Epoch 13/100
3500/3500 [==============================] - 12s 3ms/step - loss: 375205691392.0000 - val_loss: 337152376832.0000
Epoch 14/100
3500/3500 [==============================] - 11s 3ms/step - loss: 350786945024.0000 - val_loss: 315824963584.0000
Epoch 15/100
3500/3500 [==============================] - 11s 3ms/step - loss: 329439346688.0000 - val_loss: 296637693952.0000
Epoch 16/100
3500/3500 [==============================] - 11s 3ms/step - loss: 310290513920.0000 - val_loss: 278684532736.0000
Epoch 17/100
3500/3500 [==============================] - 11s 3ms/step - loss: 291863527424.0000 - val_loss: 261565693952.0000
Epoch 18/100
3500/3500 [==============================] - 11s 3ms/step - loss: 272524886016.0000 - val_loss: 245535424512.0000
Epoch 19/100
3500/3500 [==============================] - 12s 3ms/step - loss: 255604457472.0000 - val_loss: 230537510912.0000
Epoch 20/100
3500/3500 [==============================] - 11s 3ms/step - loss: 239092252672.0000 - val_loss: 216159371264.0000
Epoch 21/100
3500/3500 [==============================] - 11s 3ms/step - loss: 223985336320.0000 - val_loss: 202297655296.0000
Epoch 22/100
3500/3500 [==============================] - 12s 3ms/step - loss: 209053646848.0000 - val_loss: 189402415104.0000
Epoch 23/100
3500/3500 [==============================] - 11s 3ms/step - loss: 195379871744.0000 - val_loss: 177145479168.0000
Epoch 24/100
3500/3500 [==============================] - 12s 3ms/step - loss: 180810350592.0000 - val_loss: 165425446912.0000
Epoch 25/100
3500/3500 [==============================] - 11s 3ms/step - loss: 169834512384.0000 - val_loss: 154421116928.0000
Epoch 26/100
3500/3500 [==============================] - 11s 3ms/step - loss: 158785880064.0000 - val_loss: 144324870144.0000
Epoch 27/100
3500/3500 [==============================] - 12s 3ms/step - loss: 146980306944.0000 - val_loss: 134498590720.0000
Epoch 28/100
3500/3500 [==============================] - 11s 3ms/step - loss: 136268660736.0000 - val_loss: 125185179648.0000
Epoch 29/100
3500/3500 [==============================] - 11s 3ms/step - loss: 126436106240.0000 - val_loss: 116454137856.0000
Epoch 30/100
3500/3500 [==============================] - 12s 3ms/step - loss: 117105737728.0000 - val_loss: 108315385856.0000
Epoch 31/100
3500/3500 [==============================] - 11s 3ms/step - loss: 109463683072.0000 - val_loss: 100691501056.0000
Epoch 32/100
3500/3500 [==============================] - 11s 3ms/step - loss: 101181554688.0000 - val_loss: 93567795200.0000
Epoch 33/100
3500/3500 [==============================] - 12s 3ms/step - loss: 93273227264.0000 - val_loss: 86965608448.0000
Epoch 34/100
3500/3500 [==============================] - 11s 3ms/step - loss: 86881157120.0000 - val_loss: 80709943296.0000
Epoch 35/100
3500/3500 [==============================] - 12s 3ms/step - loss: 80447799296.0000 - val_loss: 74884964352.0000
Epoch 36/100
3500/3500 [==============================] - 11s 3ms/step - loss: 74682400768.0000 - val_loss: 69521555456.0000
Epoch 37/100
3500/3500 [==============================] - 11s 3ms/step - loss: 69652742144.0000 - val_loss: 64572534784.0000
Epoch 38/100
3500/3500 [==============================] - 12s 3ms/step - loss: 64600694784.0000 - val_loss: 60053512192.0000
Epoch 39/100
3500/3500 [==============================] - 11s 3ms/step - loss: 59956658176.0000 - val_loss: 55881830400.0000
Epoch 40/100
3500/3500 [==============================] - 11s 3ms/step - loss: 55683854336.0000 - val_loss: 52117303296.0000
Epoch 41/100
3500/3500 [==============================] - 12s 3ms/step - loss: 52298625024.0000 - val_loss: 48704151552.0000
Epoch 42/100
3500/3500 [==============================] - 11s 3ms/step - loss: 48704208896.0000 - val_loss: 45662937088.0000
Epoch 43/100
3500/3500 [==============================] - 12s 3ms/step - loss: 45889564672.0000 - val_loss: 42953994240.0000
Epoch 44/100
3500/3500 [==============================] - 11s 3ms/step - loss: 43336740864.0000 - val_loss: 40526344192.0000
Epoch 45/100
3500/3500 [==============================] - 11s 3ms/step - loss: 41343074304.0000 - val_loss: 38329667584.0000
Epoch 46/100
3500/3500 [==============================] - 12s 3ms/step - loss: 39019712512.0000 - val_loss: 36401291264.0000
Epoch 47/100
3500/3500 [==============================] - 11s 3ms/step - loss: 37273694208.0000 - val_loss: 34713907200.0000
Epoch 48/100
```

```
3500/3500 [==============================] - 11s 3ms/step - loss: 35525615616.0000 - val_loss: 33283368960.0000
Epoch 49/100
3500/3500 [==============================] - 12s 3ms/step - loss: 34034669568.0000 - val_loss: 31990720512.0000
Epoch 50/100
3500/3500 [==============================] - 11s 3ms/step - loss: 32745592832.0000 - val_loss: 30802511872.0000
Epoch 51/100
3500/3500 [==============================] - 12s 3ms/step - loss: 31757709312.0000 - val_loss: 29763026944.0000
Epoch 52/100
3500/3500 [==============================] - 12s 3ms/step - loss: 30725447680.0000 - val_loss: 28870731776.0000
Epoch 53/100
3500/3500 [==============================] - 11s 3ms/step - loss: 30301433856.0000 - val_loss: 28002928640.0000
Epoch 54/100
3500/3500 [==============================] - 12s 3ms/step - loss: 29413881856.0000 - val_loss: 27256612864.0000
Epoch 55/100
3500/3500 [==============================] - 11s 3ms/step - loss: 28519575552.0000 - val_loss: 26560821248.0000
Epoch 56/100
3500/3500 [==============================] - 11s 3ms/step - loss: 27828357120.0000 - val_loss: 25949415424.0000
Epoch 57/100
3500/3500 [==============================] - 12s 3ms/step - loss: 27362172928.0000 - val_loss: 25376983040.0000
Epoch 58/100
3500/3500 [==============================] - 11s 3ms/step - loss: 26997155840.0000 - val_loss: 24801363968.0000
Epoch 59/100
3500/3500 [==============================] - 11s 3ms/step - loss: 26366113792.0000 - val_loss: 24315068416.0000
Epoch 60/100
3500/3500 [==============================] - 12s 3ms/step - loss: 25624805376.0000 - val_loss: 23842801664.0000
Epoch 61/100
3500/3500 [==============================] - 12s 3ms/step - loss: 25158729728.0000 - val_loss: 23431553024.0000
Epoch 62/100
3500/3500 [==============================] - 12s 3ms/step - loss: 24828473344.0000 - val_loss: 22952957952.0000
Epoch 63/100
3500/3500 [==============================] - 11s 3ms/step - loss: 24133113856.0000 - val_loss: 22608713728.0000
Epoch 64/100
3500/3500 [==============================] - 11s 3ms/step - loss: 24152397824.0000 - val_loss: 22187180032.0000
Epoch 65/100
3500/3500 [==============================] - 12s 3ms/step - loss: 23455307776.0000 - val_loss: 21802670080.0000
Epoch 66/100
3500/3500 [==============================] - 11s 3ms/step - loss: 22972102656.0000 - val_loss: 21466863616.0000
Epoch 67/100
3500/3500 [==============================] - 12s 3ms/step - loss: 22910437376.0000 - val_loss: 21139642368.0000
Epoch 68/100
3500/3500 [==============================] - 12s 3ms/step - loss: 22670950400.0000 - val_loss: 20811978752.0000
Epoch 69/100
3500/3500 [==============================] - 12s 3ms/step - loss: 22494793728.0000 - val_loss: 20500830208.0000
Epoch 70/100
3500/3500 [==============================] - 12s 3ms/step - loss: 22153525248.0000 - val_loss: 20192782336.0000
Epoch 71/100
3500/3500 [==============================] - 12s 3ms/step - loss: 22092113920.0000 - val_loss: 19905064960.0000
Epoch 72/100
3500/3500 [==============================] - 12s 3ms/step - loss: 21601464320.0000 - val_loss: 19627452416.0000
Epoch 73/100
3500/3500 [==============================] - 12s 3ms/step - loss: 21141583872.0000 - val_loss: 19381852160.0000
Epoch 74/100
3500/3500 [==============================] - 11s 3ms/step - loss: 20818229248.0000 - val_loss: 19158253568.0000
Epoch 75/100
3500/3500 [==============================] - 12s 3ms/step - loss: 20810127360.0000 - val_loss: 18913705984.0000
Epoch 76/100
3500/3500 [==============================] - 12s 3ms/step - loss: 20644577280.0000 - val_loss: 18656931840.0000
Epoch 77/100
3500/3500 [==============================] - 12s 3ms/step - loss: 20164083712.0000 - val_loss: 18444519424.0000
Epoch 78/100
3500/3500 [==============================] - 12s 3ms/step - loss: 19818045440.0000 - val_loss: 18173810688.0000
Epoch 79/100
3500/3500 [==============================] - 12s 3ms/step - loss: 19892738048.0000 - val_loss: 17975521280.0000
Epoch 80/100
3500/3500 [==============================] - 11s 3ms/step - loss: 19654350848.0000 - val_loss: 17776916480.0000
Epoch 81/100
3500/3500 [==============================] - 12s 3ms/step - loss: 19405084672.0000 - val_loss: 17564684288.0000
Epoch 82/100
3500/3500 [==============================] - 12s 3ms/step - loss: 19170838528.0000 - val_loss: 17357641728.0000
Epoch 83/100
3500/3500 [==============================] - 12s 3ms/step - loss: 19056832512.0000 - val_loss: 17147287552.0000
Epoch 84/100
3500/3500 [==============================] - 12s 3ms/step - loss: 18851735552.0000 - val_loss: 16961145856.0000
Epoch 85/100
3500/3500 [==============================] - 12s 3ms/step - loss: 18791458816.0000 - val_loss: 16794759168.0000
Epoch 86/100
3500/3500 [==============================] - 12s 3ms/step - loss: 18444681216.0000 - val_loss: 16653736960.0000
Epoch 87/100
3500/3500 [==============================] - 12s 3ms/step - loss: 18259365888.0000 - val_loss: 16436895744.0000
Epoch 88/100
3500/3500 [==============================] - 12s 3ms/step - loss: 18135013376.0000 - val_loss: 16301491200.0000
Epoch 89/100
3500/3500 [==============================] - 12s 3ms/step - loss: 18294607872.0000 - val_loss: 16206785536.0000
Epoch 90/100
3500/3500 [==============================] - 12s 3ms/step - loss: 17888196608.0000 - val_loss: 15975628800.0000
Epoch 91/100
3500/3500 [==============================] - 12s 3ms/step - loss: 17678673920.0000 - val_loss: 15897917440.0000
Epoch 92/100
3500/3500 [==============================] - 12s 3ms/step - loss: 17463703552.0000 - val_loss: 15738961920.0000
Epoch 93/100
3500/3500 [==============================] - 11s 3ms/step - loss: 17638373376.0000 - val_loss: 15568631808.0000
Epoch 94/100
3500/3500 [==============================] - 11s 3ms/step - loss: 17264066560.0000 - val_loss: 15433594880.0000
Epoch 95/100
3500/3500 [==============================] - 12s 3ms/step - loss: 17110662144.0000 - val_loss: 15322685440.0000
```

```
Epoch 96/100
3500/3500 [==============================] - 11s 3ms/step - loss: 17199093760.0000 - val_loss: 15301575680.0000
Epoch 97/100
3500/3500 [==============================] - 12s 3ms/step - loss: 16939251712.0000 - val_loss: 15160739840.0000
Epoch 98/100
3500/3500 [==============================] - 12s 3ms/step - loss: 16743148544.0000 - val_loss: 14951175168.0000
Epoch 99/100
3500/3500 [==============================] - 12s 3ms/step - loss: 16887285760.0000 - val_loss: 14833034240.0000
Epoch 100/100
3500/3500 [==============================] - 12s 3ms/step - loss: 16801883136.0000 - val_loss: 14721144832.0000
```

In [55]:
```
pd.DataFrame({'True Values': y_test, 'Predicted Values': pred}).hvplot.scatter(x='True Values', y='Predicted Values')
```

Out[55]:

In [56]:
```
pd.DataFrame(r.history)
```

Out[56]:

	loss	val_loss
0	1.635959e+12	1.658110e+12
1	1.635208e+12	1.656222e+12
2	1.630735e+12	1.647849e+12
3	1.615667e+12	1.623587e+12
4	1.578035e+12	1.568295e+12
...	...	...
95	1.719909e+10	1.530158e+10
96	1.693925e+10	1.516074e+10
97	1.674315e+10	1.495118e+10
98	1.688729e+10	1.483303e+10
99	1.680188e+10	1.472114e+10

100 rows × 2 columns

In [57]:
```
pd.DataFrame(r.history).hvplot.line(y=['loss', 'val_loss'])
```

Out[57]:

In [59]:
```
test_pred = model.predict(X_test)
train_pred = model.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)

print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Artficial Neural Network", *evaluate(y_test, test_pred), 0]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', 'Cross Validation'])
```

```
47/47 [==============================] - 0s 1ms/step
110/110 [==============================] - 0s 1ms/step
Test set evaluation:

_____________________________________
MAE: 94912.90125053687
MSE: 14721153465.518364
RMSE: 121330.76059070248
R2 Square 0.8752553764041622

_____________________________________
Train set evaluation:

_____________________________________
MAE: 98039.4612827314
MSE: 15688677933.532608
RMSE: 125254.45274932387
R2 Square 0.876923101706429

_____________________________________
```

Random Forest Regressor

In [60]:
```
from sklearn.ensemble import RandomForestRegressor

rf_reg = RandomForestRegressor(n_estimators=1000)
rf_reg.fit(X_train, y_train)

test_pred = rf_reg.predict(X_test)
train_pred = rf_reg.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)

print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["Random Forest Regressor", *evaluate(y_test, test_pred), 0]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', 'Cross Validation'])
```

```
Test set evaluation:
_____________________________________
MAE: 94384.86904540827
MSE: 14193674223.405785
RMSE: 119137.20755249295
R2 Square 0.8797251484003632
_____________________________________
Train set evaluation:
_____________________________________
MAE: 35320.769058241756
MSE: 1986369265.366525
RMSE: 44568.7027561553
R2 Square 0.9844170318823071
_____________________________________
```

Support Vector Machine

In [61]:
```
from sklearn.svm import SVR

svm_reg = SVR(kernel='rbf', C=1000000, epsilon=0.001)
svm_reg.fit(X_train, y_train)

test_pred = svm_reg.predict(X_test)
train_pred = svm_reg.predict(X_train)

print('Test set evaluation:\n_____________________________________')
print_evaluate(y_test, test_pred)

print('Train set evaluation:\n_____________________________________')
print_evaluate(y_train, train_pred)

results_df_2 = pd.DataFrame(data=[["SVM Regressor", *evaluate(y_test, test_pred), 0]],
                            columns=['Model', 'MAE', 'MSE', 'RMSE', 'R2 Square', 'Cross Validation'])
```

```
Test set evaluation:
_____________________________________
MAE: 87205.73051021632
MSE: 11720932765.275507
RMSE: 108263.25676458984
R2 Square 0.9006787511983232
_____________________________________
Train set evaluation:
_____________________________________
MAE: 73692.56848073176
MSE: 9363827731.41128
RMSE: 96766.87310960957
R2 Square 0.9265412370487788
_____________________________________
```

Summary

We covered a lot of ground including:

> - The common linear regression models (Ridge, Lasso, ElasticNet, ...).
> - The representation used by the model.
> - Learning algorithms used to estimate the coefficients in the model.
> - Rules of thumb to consider when preparing data for use with linear regression.
> - How to evaluate a linear regression model.