# Machine Learning Practice - Week 3

## Import basic libraries

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import sklearn as sklearn
from sklearn import datasets
from pandas.plotting import scatter_matrix
plt.style.use('seaborn')
```

## Introducing the dataset

---

#### California Housing dataset

The original database is available from StatLib http://lib.stat.cmu.edu/datasets/. This dataset the following input variables (features):

- MedInc - median income in block
- HouseAge - median house age in block
- AveRooms - average number of rooms
- AveBedrms - average number of bedrooms
- Population - block population
- AveOccupancy - average house occupancy
- Latitude - house block latitude
- Longitude - house block longitude

The target variable is the median house value for California districts.

This dataset was derived from the 1990 U.S. census, using one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people).

```
==============   ==============
Samples total             20640
Dimensionality                8
Features                   real
```

```
    Target          real 0.15 - 5.
    ==============  ==============
```

```
# X, Y = sklearn.datasets.fetch_california_housing(return_X_y=True)
dataset = sklearn.datasets.fetch_california_housing()
X, y = dataset.data, dataset.target

print('shape of attributes', X.shape)
print('shape of target', y.shape)
```

```
Downloading Cal. housing from https://ndownloader.figshare.com/files/5976036 to /roo
shape of attributes (20640, 8)
shape of target (20640,)
```

```
data=pd.DataFrame(X,columns = ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms',
                               'Population', 'AveOccupancy', 'Latitude', 'Longitude'])
data
```

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccupancy | Latitude | Lo |
|---|---|---|---|---|---|---|---|---|
| **0** | 8.3252 | 41.0 | 6.984127 | 1.023810 | 322.0 | 2.555556 | 37.88 | |
| **1** | 8.3014 | 21.0 | 6.238137 | 0.971880 | 2401.0 | 2.109842 | 37.86 | |
| **2** | 7.2574 | 52.0 | 8.288136 | 1.073446 | 496.0 | 2.802260 | 37.85 | |
| **3** | 5.6431 | 52.0 | 5.817352 | 1.073059 | 558.0 | 2.547945 | 37.85 | |
| **4** | 3.8462 | 52.0 | 6.281853 | 1.081081 | 565.0 | 2.181467 | 37.85 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **20635** | 1.5603 | 25.0 | 5.045455 | 1.133333 | 845.0 | 2.560606 | 39.48 | |
| **20636** | 2.5568 | 18.0 | 6.114035 | 1.315789 | 356.0 | 3.122807 | 39.49 | |
| **20637** | 1.7000 | 17.0 | 5.205543 | 1.120092 | 1007.0 | 2.325635 | 39.43 | |
| **20638** | 1.8672 | 18.0 | 5.329513 | 1.171920 | 741.0 | 2.123209 | 39.43 | |
| **20639** | 2.3886 | 16.0 | 5.254717 | 1.162264 | 1387.0 | 2.616981 | 39.37 | |

20640 rows × 8 columns

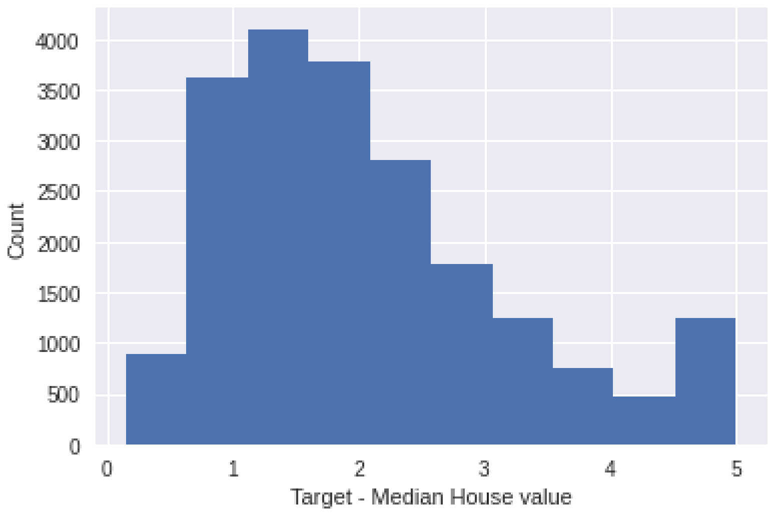
```
data.describe()
```

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccupa |
|---|---|---|---|---|---|---|
| **count** | 20640.000000 | 20640.000000 | 20640.000000 | 20640.000000 | 20640.000000 | 20640.000 |
| **mean** | 3.870671 | 28.639486 | 5.429000 | 1.096675 | 1425.476744 | 3.070 |
| **std** | 1.899822 | 12.585558 | 2.474173 | 0.473911 | 1132.462122 | 10.386 |
| **min** | 0.499900 | 1.000000 | 0.846154 | 0.333333 | 3.000000 | 0.692 |

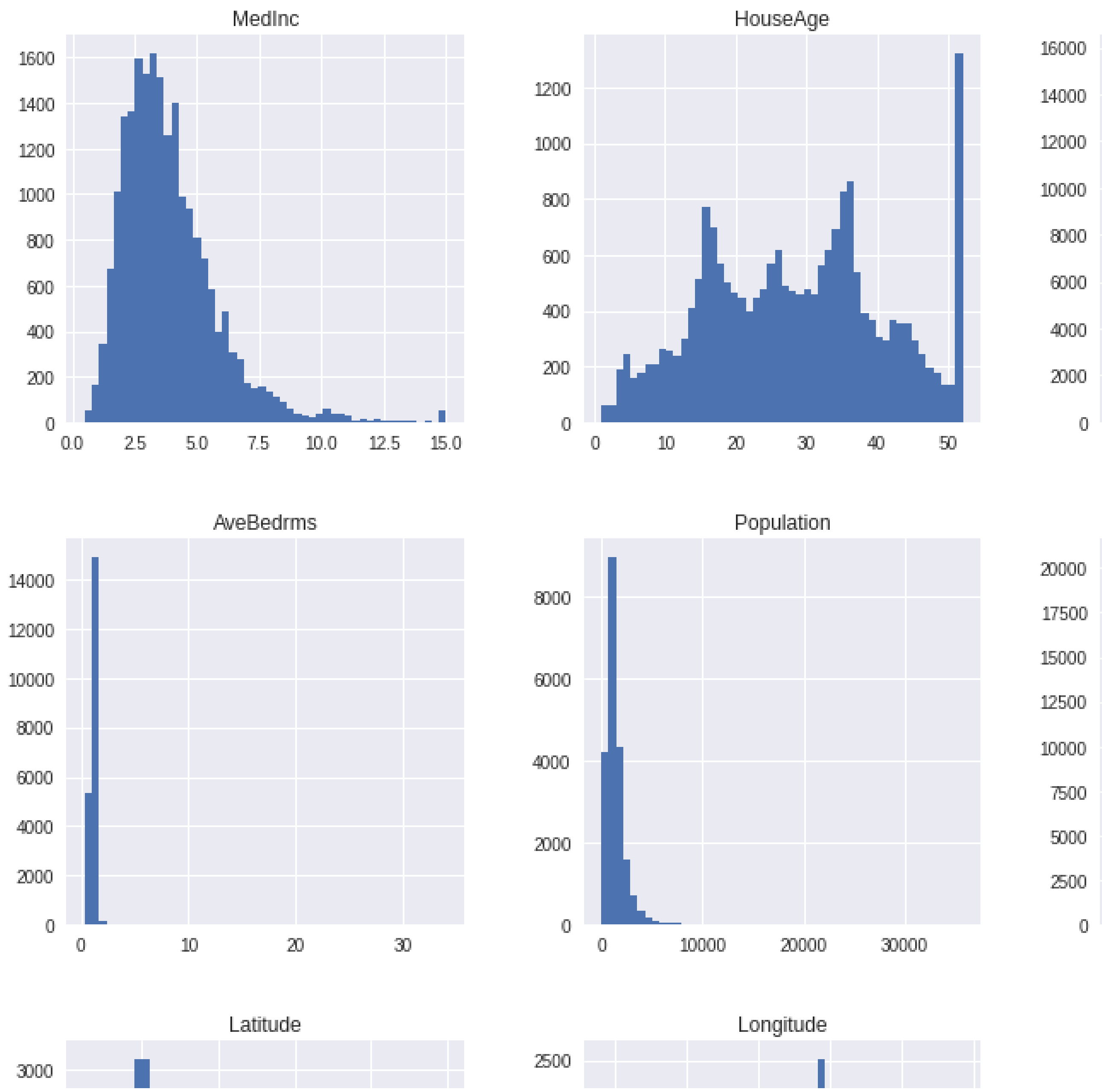
## Visualization of the data

Let us have a look at the range and distribution of the target and input features by plotting their histograms.

```
plt.hist(y)
plt.xlabel('Target - Median House value')
plt.ylabel('Count')
plt.show()
```

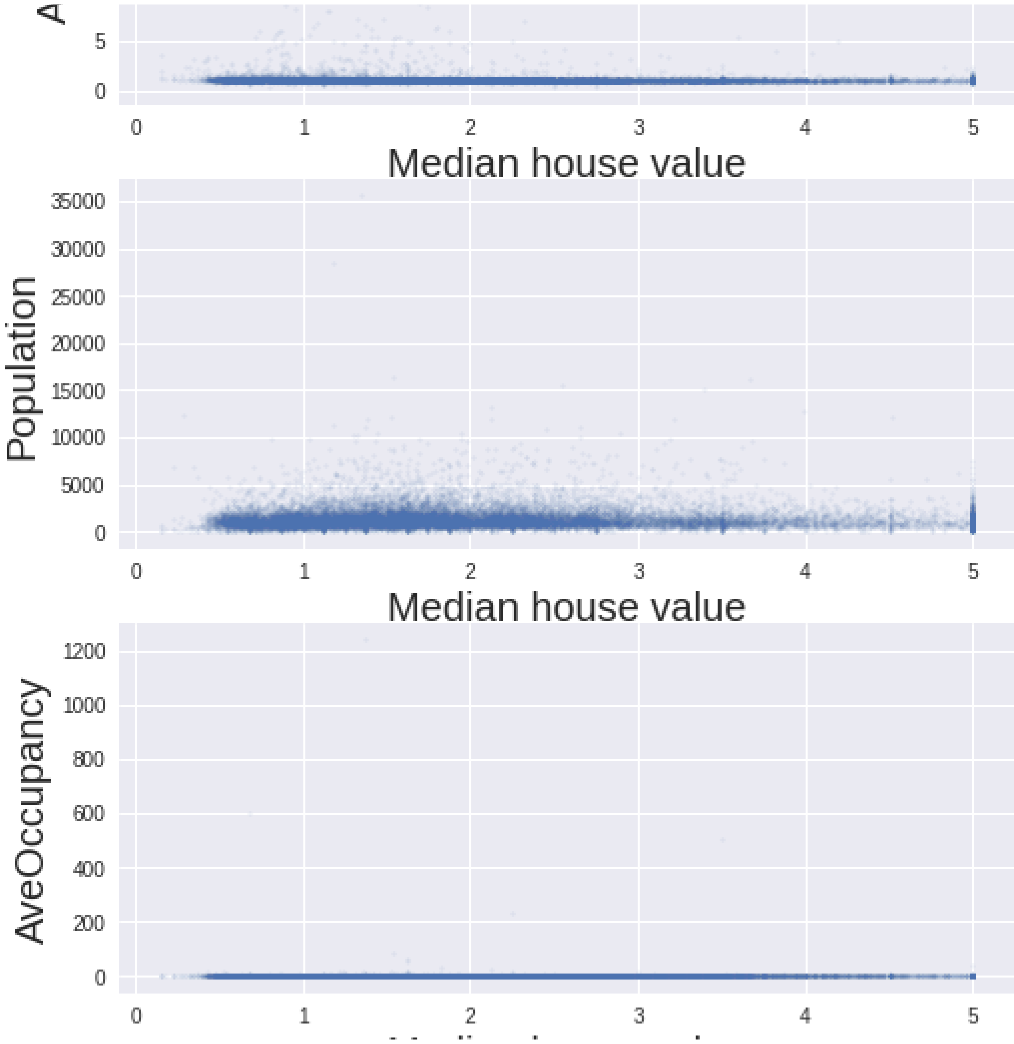



```
data.hist(bins=50,figsize=(15,15))
# display histogram
plt.show()
```



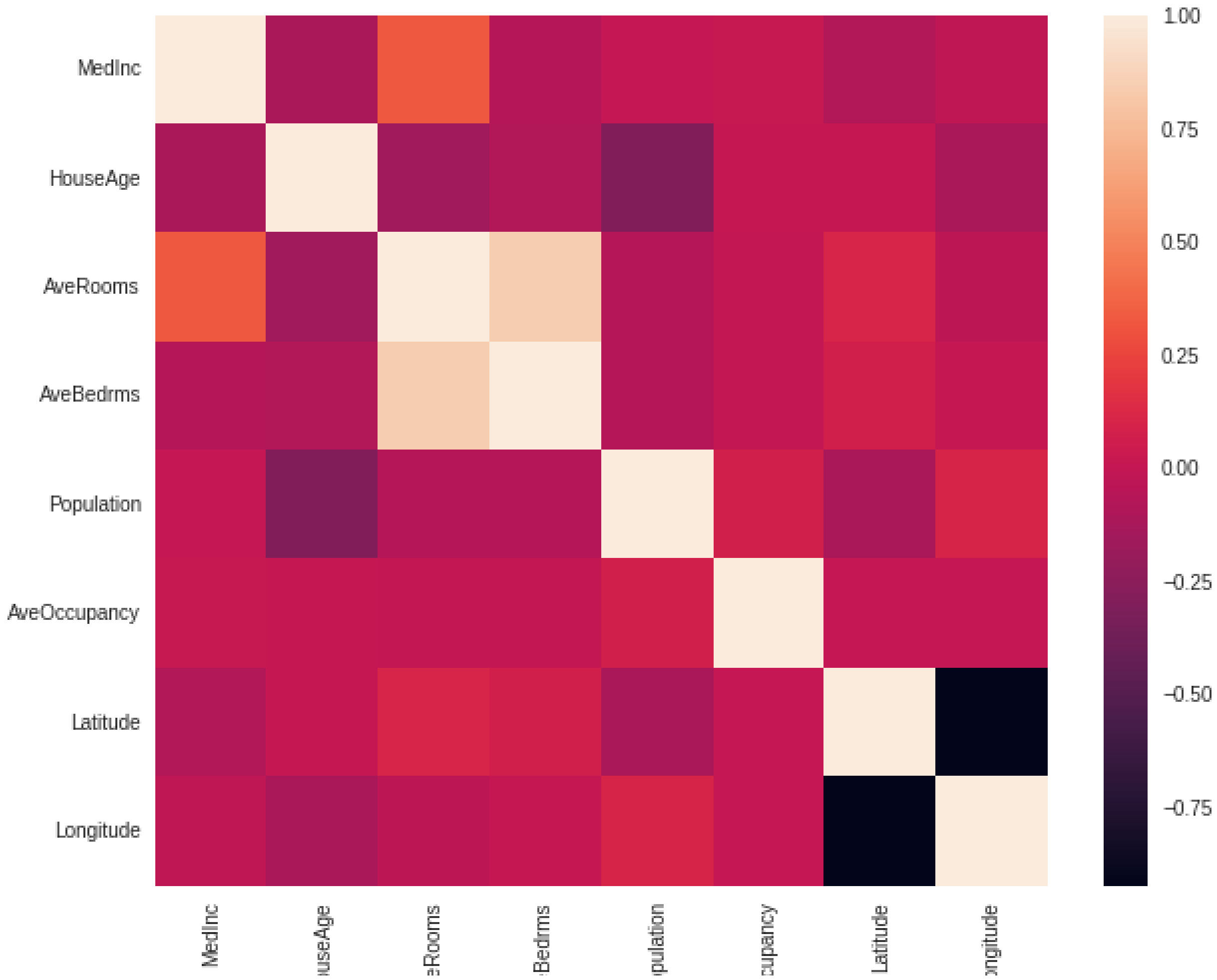

```
i=1
plt.figure(figsize=(8,32))

for colname in data:
  plt.subplot(8,1,i)
  # plt.subplot(4,2,i)
  plt.scatter(y,data[colname].values, alpha=0.08, s=3)
  plt.xlabel('Median house value', fontsize = 20)
  plt.ylabel(colname, fontsize = 20)
  i+=1
plt.suptitle("Relation between Median house value and features")
plt.show()
```

Relation between Median house value and features




You can notice a wide range of feature distribution. Also there are outliers in some features.

```
plt.figure(figsize=(10,8))
sns.heatmap(data.corr())
plt.show()
```




You can observe that Median Income is positively correlated with Average Rooms but negatively correlated with HouseAge

# Cleaning the data

---

## 1. Identification of features that only have a single value.

```
# get number of unique values for each column
counts = data.nunique()
print(counts)

# record columns to delete
to_del = [i for i,v in enumerate(counts) if v == 1]
print('Columns with single value', to_del)
# drop useless columns
data.drop(to_del, axis=1, inplace=True)
print(data.shape)
```

```
MedInc        12928
HouseAge         52
AveRooms      19392
AveBedrms     14233
Population     3888
```

```
AveOccupancy    18841
Latitude          862
Longitude         844
dtype: int64
Columns with single value []
(20640, 8)
```

Since there are no columns with single value, no need to drop any column at this stage.

## 2. Identification of features with very few unique values.

```
Name_List = ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms',
             'Population', 'AveOccupancy', 'Latitude', 'Longitude']
```

```
col_todel=[]
print('Feature name, Number of unique values, Percentage of unique values out of all rows

for i in range(data.shape[1]):
  col=list(data[Name_List[i]])
  num = np.unique(col).size
  percentage = float(num / data.shape[0]) * 100
  if percentage < 1:
      col_todel.append(i)
  print('%s, %d, %.1f%%' % (Name_List[i], num, percentage))
print('\n Column to delete', col_todel)
for j in col_todel:
  print('\n Feature to delete', Name_List[j])
```

```
Feature name, Number of unique values, Percentage of unique values out of all rows ir
MedInc, 12928, 62.6%
HouseAge, 52, 0.3%
AveRooms, 19392, 94.0%
AveBedrms, 14233, 69.0%
Population, 3888, 18.8%
AveOccupancy, 18841, 91.3%
Latitude, 862, 4.2%
Longitude, 844, 4.1%

 Column to delete [1]

 Feature to delete HouseAge
```

```
data1=data.copy() # original features will be retained in data
# drop useless columns
for i in col_todel:
  data1.drop(Name_List[i], axis=1, inplace=True)
print(data1.shape)
```

```
(20640, 7)
```

## 3. Identification of rows that contain duplicate observations.

```
# delete duplicate rows
data1.drop_duplicates(inplace=True)
print(data1.shape)
```

```
(20640, 7)
```

# Create Train and Test data

---

```
import sklearn
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(data1, y, test_size=0.2, random_state=
```

```
print('Shape of training data',X_train.shape)
print('Shape of training labels',y_train.shape)
print('Shape of testing data',X_test.shape)
print('Shape of testing labels',y_test.shape)
```

```
Shape of training data (16512, 7)
Shape of training labels (16512,)
Shape of testing data (4128, 7)
Shape of testing labels (4128,)
```

# **Linear Regression**

Import basic libraries

```
# import model
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn import metrics
```

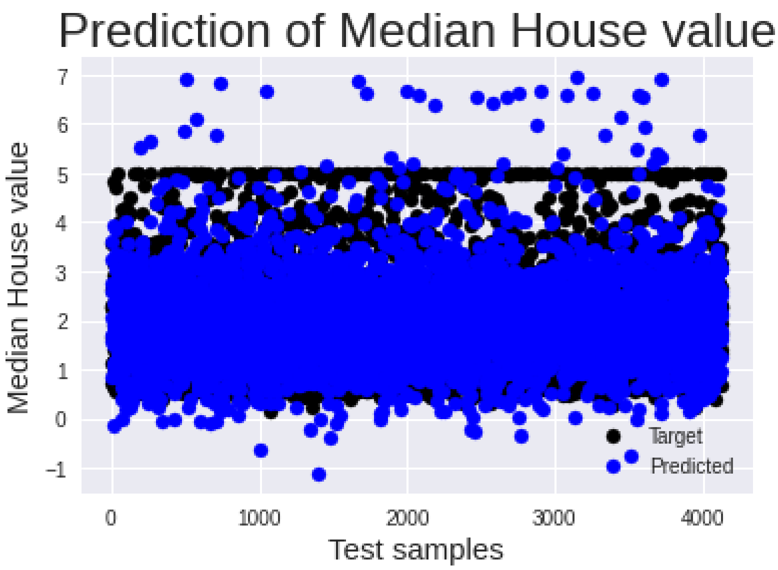
Selection of scalers from sklearn.preprocessing

```
from sklearn.preprocessing import MinMaxScaler, MaxAbsScaler, StandardScaler, RobustScaler
# from sklearn.preprocessing import QuantileTransformer, PowerTransformer
```

```
linear_regression = LinearRegression()
mmscaler = MinMaxScaler()
```

```
X_train_norm = mmscaler.fit_transform(X_train)
X_test_norm = mmscaler.transform(X_test)

linear_regression.fit(X_train_norm, y_train)
y_pred = linear_regression.predict(X_test_norm)
mse = metrics.mean_squared_error(y_test, y_pred)
print('MSE = ', mse)
# Plot outputs
x_range=range(X_test.shape[0])
plt.scatter(x_range, y_test,  color='black')
plt.scatter(x_range, y_pred, color='blue')
plt.title('Prediction of Median House value', size=24)
plt.xlabel('Test samples', size=15)
plt.ylabel('Median House value',size=15)
plt.legend(labels=['Target', 'Predicted'])
plt.show()

# Evaluate the models using pipeline and crossvalidation
linear_regression1 = LinearRegression()
pipe_1 = Pipeline([('scaler', MinMaxScaler()),
                         ("regression", linear_regression1)])
pipe_1.fit(X_train,y_train)
scores = cross_val_score(linear_regression1, X_train, y_train,cv=10)
print("Score: {:.2f} %".format(scores.mean()))
```

```
MSE =  0.5380990250708308
```




```
Score: 0.60 %
```

**Exercise:** Try with other scalers.

## PolynomialFeatures

Polynomial Features(degree=d) transforms an array containing $n$ features into an array containing $\frac{(n+d)!}{d!n!}$ features. Let us try with a $2^{nd}$ degree polynomial.

```
poly_features = PolynomialFeatures(degree=2, include_bias=False)
mmscaler = MinMaxScaler()
X_trainpoly = poly_features.fit_transform(X_train)
X_testpoly = poly_features.fit_transform(X_test)

X_train_norm = mmscaler.fit_transform(X_trainpoly)
X_test_norm = mmscaler.transform(X_testpoly)
print(X_train_norm[0].shape)
```

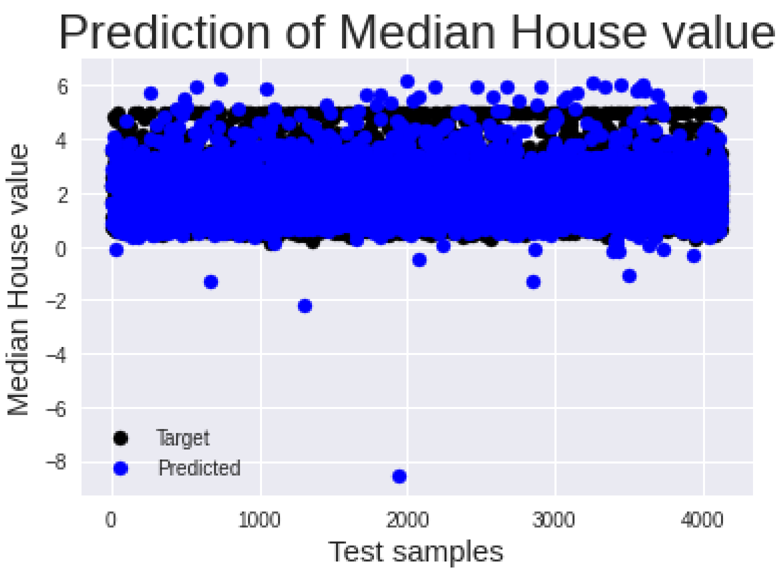
```
(35,)
```

```
print(X_trainpoly[0].shape)
```

```
(35,)
```

```
lin_reg = LinearRegression()
lin_reg.fit(X_trainpoly, y_train)
y_pred = lin_reg.predict(X_testpoly)
mse = metrics.mean_squared_error(y_test, y_pred)
print('MSE = ', mse)
# Plot outputs
x_range=range(X_test.shape[0])
plt.scatter(x_range, y_test,  color='black')
plt.scatter(x_range, y_pred, color='blue')
plt.title('Prediction of Median House value', size=24)
plt.xlabel('Test samples', size=15)
plt.ylabel('Median House value',size=15)
plt.legend(labels=['Target', 'Predicted'])
plt.show()
```

```
MSE =  0.47799573854496114
```

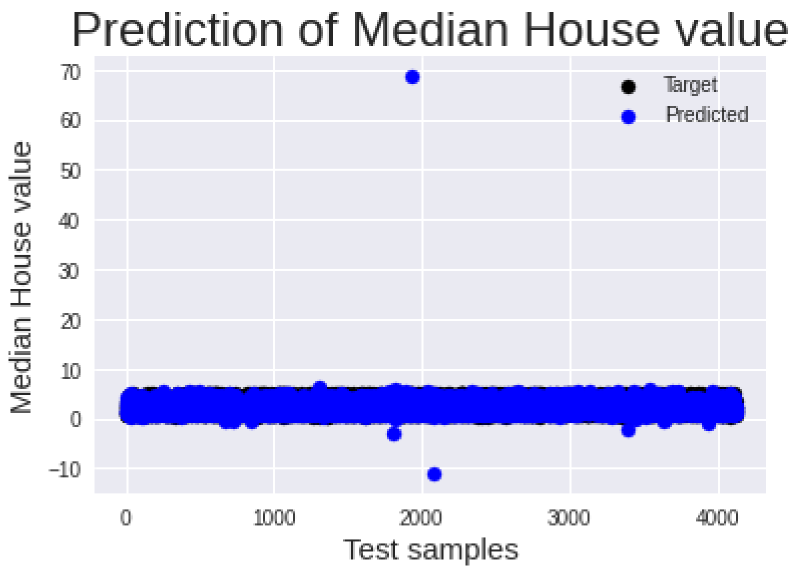



Let us increase the degree of polynomial to 3 and see what happens.

```
poly_features = PolynomialFeatures(degree=3, include_bias=False)
mmscaler = MinMaxScaler()
```

```
X_trainpoly = poly_features.fit_transform(X_train)
X_testpoly = poly_features.fit_transform(X_test)

X_train_norm = mmscaler.fit_transform(X_trainpoly)
X_test_norm = mmscaler.transform(X_testpoly)

print(X_trainpoly[0].shape)
lin_reg = LinearRegression()
lin_reg.fit(X_trainpoly, y_train)
y_pred = lin_reg.predict(X_testpoly)
mse = metrics.mean_squared_error(y_test, y_pred)
print('MSE = ', mse)

# Plot outputs
x_range=range(X_test.shape[0])
plt.scatter(x_range, y_test,  color='black')
plt.scatter(x_range, y_pred, color='blue')
plt.title('Prediction of Median House value', size=24)
plt.xlabel('Test samples', size=15)
plt.ylabel('Median House value',size=15)
plt.legend(labels=['Target', 'Predicted'])
plt.show()
```

```
(119,)
MSE =  1.551614116199306
```




What happened?

---

MSE has increased beyond 1.

## Learning Curves

- Plots of the model's performance on the training set and the validation set as a function of the training set size (or the training iteration).

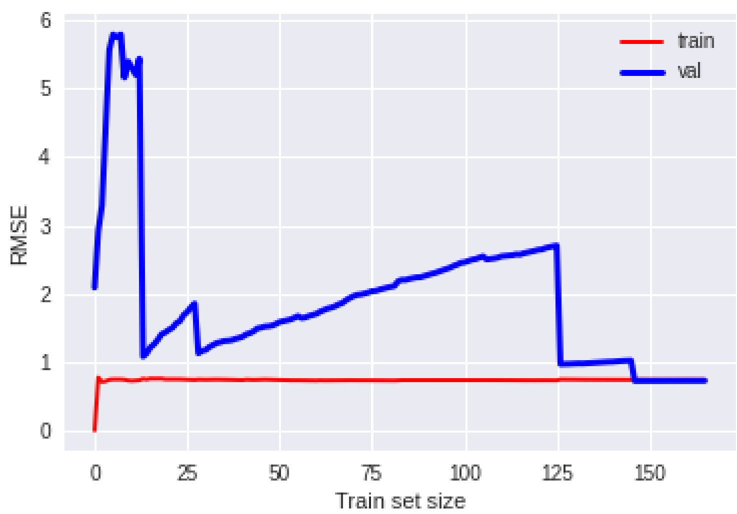
- To generate the plots, simply train the model several times on different sized subsets of the training set.

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
  scaler = StandardScaler()
  X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
  X_train_norm = scaler.fit_transform(X_train)
  X_val_norm = scaler.transform(X_val)
  train_errors, val_errors = [], []
  for m in range(1, len(X_train_norm),100):
    model.fit(X_train_norm[:m], y_train[:m])
    y_train_predict = model.predict(X_train_norm[:m])
    y_val_predict = model.predict(X_val_norm)
    train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
    val_errors.append(mean_squared_error(y_val, y_val_predict))

  plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
  plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
  plt.xlabel('Train set size', fontsize = 22)
  plt.ylabel('RMSE', fontsize = 22)
  plt.legend()
  print('Train Erros', train_errors)
```

Let's look at the learning curves of the plain Linear Regression model

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg,data1,y)
```

```
Train Erros [0.0, 0.5924496562085271, 0.4957569646821756, 0.5091571110603256, 0.5413(
```




**Inference**:

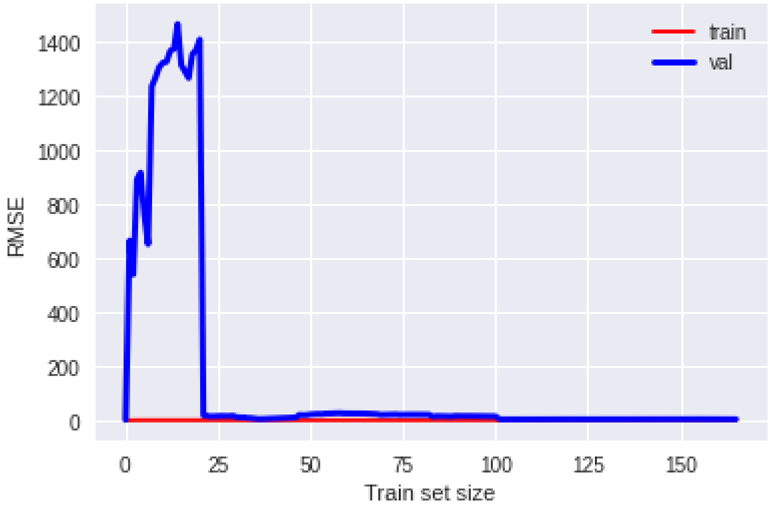
- These learning curves are typical of an underfitting model. Both curves have reached a plateau; they are close and fairly high.
- Few instances in the training set means the model can fit them perfectly. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly.
- When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite big. Then as the model is shown more training examples, it learns and thus the validation error slowly goes down.

Let's look at the learning curves of the Polynomial Regression model

---

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline([("poly_features", PolynomialFeatures(degree=2, include_b
                                  ("lin_reg", LinearRegression()),])
plot_learning_curves(polynomial_regression,data1, y)
```

```
Train Erros [0.0, 0.1709999969665855, 0.21698272427749002, 0.34020002907050134, 0.35:
```




**Inference:** Do these learning curves look a bit like the previous ones? No

- The error on the training data is lower than with the Linear Regression model. This means that the model performs better on the training data than on the validation data. Overfitting occured.
- One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

## Regularized Linear Models

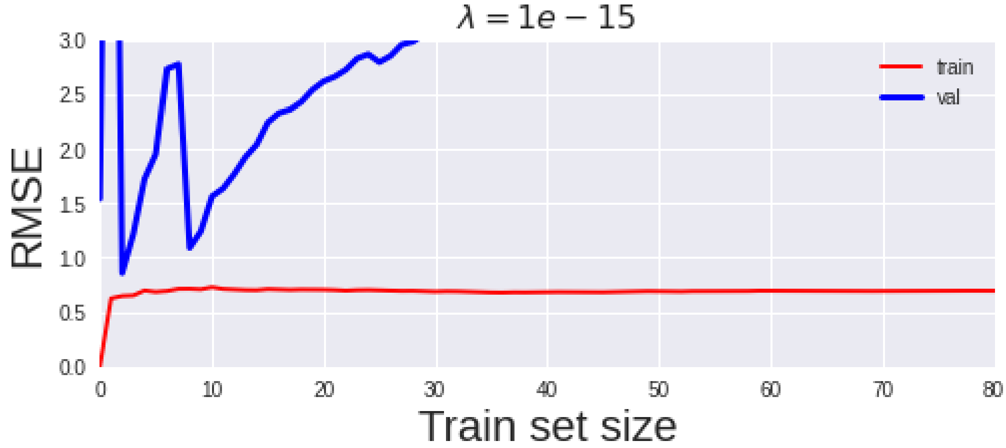
For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at two ways to constrain the weights.

- Ridge Regression
- Lasso Regression

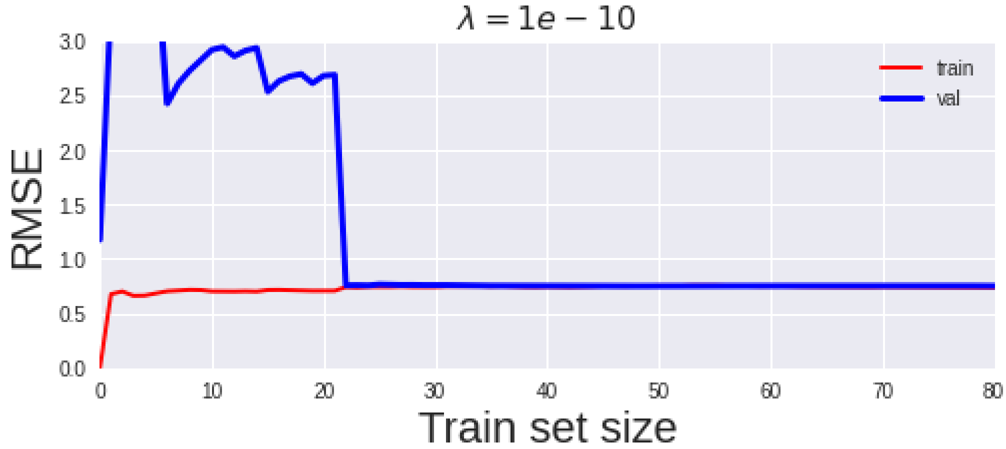
## Ridge Regression

```
from sklearn.linear_model import Ridge

for alpha_range in [1e-15, 1e-10, 1e-8, 1e-4, 1e-3, 1e-2, 1, 10, 20, 50, 70, 100]:
  plt.figure(figsize=(8,3))
  ridge_regression = Ridge(alpha=alpha_range, solver='cholesky')
  plot_learning_curves(ridge_regression, X_train, y_train)
  plt.axis([0, 80, 0, 3])
  plt.title(r"$\lambda = {}$".format(alpha_range), fontsize=16)
  plt.show()
```

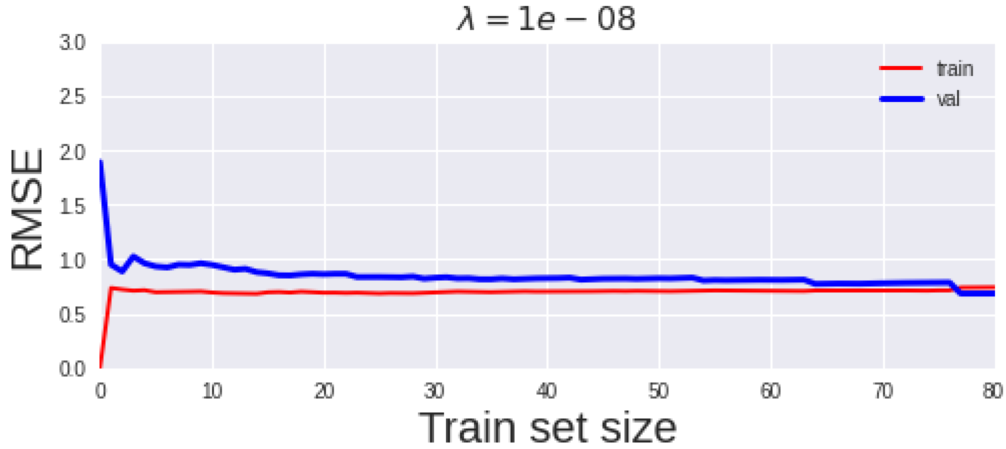
Train Erros [0.0, 0.3892265062360528, 0.4177128262322926, 0.42388255846505746, 0.4853



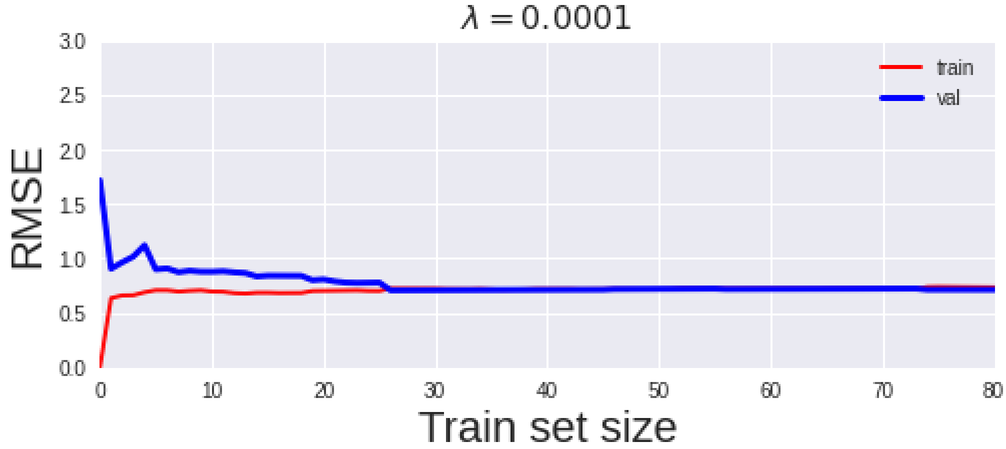

Train Erros [0.0, 0.4567828100747261, 0.49130581772437154, 0.43660219804026945, 0.438



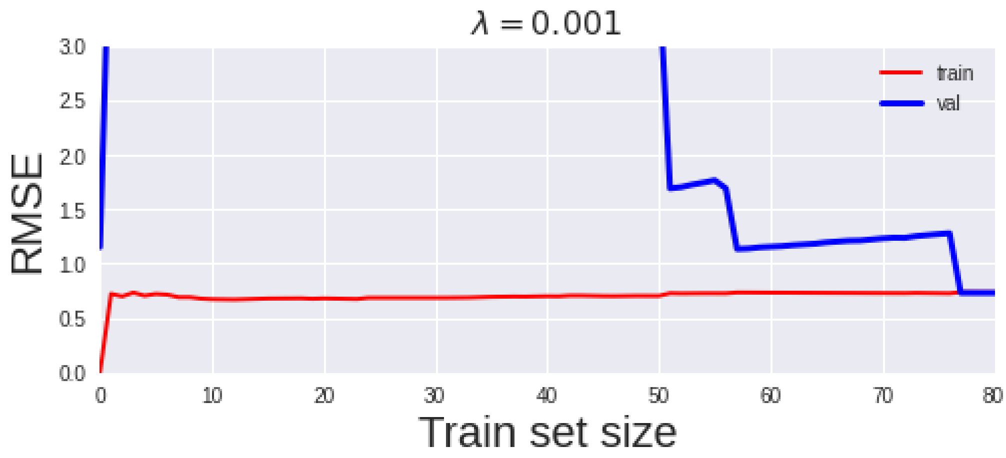

Train Erros [0.0, 0.5438768237773459, 0.5227512314896334, 0.5068364688860119, 0.5138



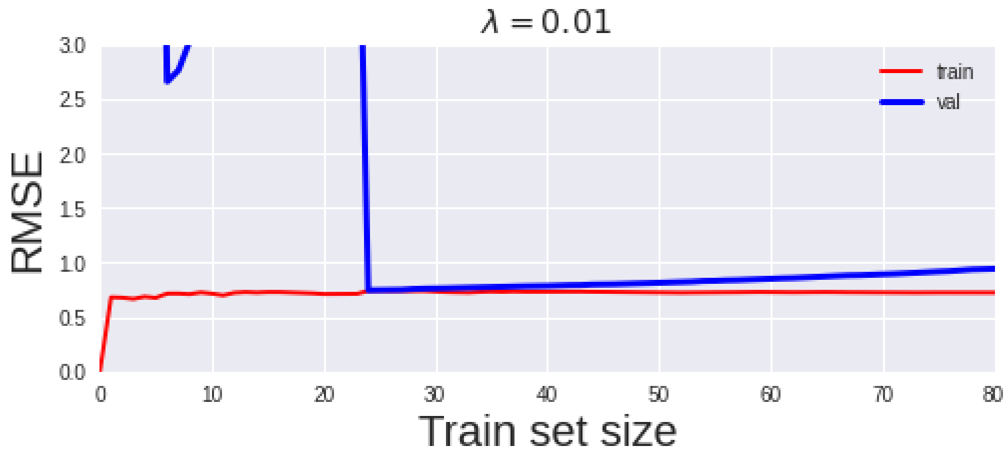

Train Erros [0.0, 0.4023899704510929, 0.4347544444699565, 0.43717291815989195, 0.4737




Train Erros [0.0, 0.5208928742792114, 0.48685107875433237, 0.5368430706531254, 0.4961




```
Train Erros [0.0, 0.46513938820338674, 0.4604705205286685, 0.4451881566451582, 0.4727
```



## Using Gridsearch to select the optimal values of $\lambda$

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge

ridge = Ridge()
scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)

parameters = {"alpha":[1e-15, 1e-10, 1e-8, 1e-4, 1e-3, 1e-2, 1, 10, 20, 50, 70, 100]}
ridge_regression = GridSearchCV(ridge, parameters, scoring='neg_mean_squared_error', cv=5)
ridge_regression.fit(X_train_norm, y_train)

print('Best parameter', ridge_regression.best_params_)
print('Best Score',-ridge_regression.best_score_)

pred_test_rr= ridge_regression.predict(X_test_norm)
print('MSE for test prediction', mean_squared_error(y_test,pred_test_rr))
```

```
Best parameter {'alpha': 20}
Best Score 0.5397676233324156
MSE for test prediction 0.5378202207225878
```

## Lasso Regression

```
from sklearn.linear_model import Lasso

for alpha_range in [1e-15, 1e-10, 1e-8, 1e-4, 1e-3, 1e-2, 1, 10, 20, 50, 70, 100]:
```