Data Science and Artificial Intelliegence Practicum

5-modul. Machine Learning

Original Notebook -> https://jovian.ai/anvarnarz/05-ml-04-machinelearning

5.4 - Machine Learning

Importing libraries for data analysis

In [1]:
```python
# importing libraries
import numpy as np
import pandas as pd
```

Loading dataset

In [2]:
```python
# loading dataset
URL = "https://github.com/ageron/handson-ml2/blob/master/datasets/housing/housing.csv?raw=true"
df = pd.read_csv(URL)

df.head()
```

Out[2]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Splitting data into `test_set` and `train_set`

In [3]:
```
df.shape
```

Out[3]:
```
(20640, 10)
```

In [4]:
```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(df, test_size=0.2, random_state=42)

X_train = train_set.drop('median_house_value', axis=1)
y = train_set['median_house_value'].copy()

X_num = X_train.drop('ocean_proximity', axis=1)
```

Building Pipeline

In [5]:
```
from sklearn.base import BaseEstimator, TransformerMixin

# indices of columns that we need
rooms_idx, bedrooms_idx, population_idx, households_idx = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):

    def __init__(self, add_bedrooms_per_room=True):
        self.add_bedrooms_per_room = add_bedrooms_per_room
```

```python
    def fit(self, X, y=None):
        return self  # our function is only a transformer (not an estimator)

    def transform(self, X):
        rooms_per_household = X[:, rooms_idx] / X[:, households_idx]
        population_per_household = X[:, population_idx] / X[:, households_idx]
        if self.add_bedrooms_per_room:  # add_bedrooms_per_room column is optional
            bedrooms_per_room = X[:, bedrooms_idx] / X[:, rooms_idx]
            return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
```

Pipeline for *numerical features*

In [6]:
```python
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler

num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('attribs_adder', CombinedAttributesAdder(add_bedrooms_per_room=True)),
    ('std_scaler', StandardScaler())
])
```

Pipeline for *categorical features*

In [7]:
```python
from sklearn.compose import ColumnTransformer

num_attribs = list(X_num)
cat_attribs = ['ocean_proximity']

full_pipeline = ColumnTransformer([
    ('num', num_pipeline, num_attribs),
    ('cat', OneHotEncoder(), cat_attribs)
])
```

The final and complete pipeline(`full_pipeline`) is ready.

Using pipeline: call `.fit_transform()` method.

In [8]:
```python
X_prepared = full_pipeline.fit_transform(X_train)
X_prepared
```

Out[8]:
```
array([[ 1.27258656, -1.3728112 ,  0.34849025, ...,  0.        ,
         0.        ,  1.        ],
       [ 0.70916212, -0.87669601,  1.61811813, ...,  0.        ,
         0.        ,  1.        ],
       [-0.44760309, -0.46014647, -1.95271028, ...,  0.        ,
         0.        ,  1.        ],
       ...,
       [ 0.59946887, -0.75500738,  0.58654547, ...,  0.        ,
         0.        ,  0.        ],
       [-1.18553953,  0.90651045, -1.07984112, ...,  0.        ,
         0.        ,  0.        ],
       [-1.41489815,  0.99543676,  1.85617335, ...,  0.        ,
         1.        ,  0.        ]])
```

Dataset is read for Machine Learning.

Machine Learning

Our goal is *prediction*, for this there are several Machine Learning algorithms.

Linear Regression

`sklearn.linear_model.LinearRegression` - Ordinary least squares Linear Regression.

In [9]:
```python
from sklearn.linear_model import LinearRegression

LR_model = LinearRegression()
```

`LinearRegression` is an *estimator*. Estimator receives data and *learns* to predict by `fit` method.

In [10]:
```python
LR_model.fit(X_prepared, y)
```

```
Out[10]: ▾ LinearRegression
LinearRegression()
```

Linear regression model is ready!

How can we test the model? Let's feed a row from the `housing` dataset to the model and compare the result with the existing result (label).

```
In [11]: # choosing 5 random sample rows
test_data = X_train.sample(5)
test_data
```

Out[11]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
10960	-117.89	33.76	36.0	2656.0	572.0	2370.0	571.0	3.8056	<1H OCEAN
14665	-117.12	32.80	31.0	1727.0	342.0	879.0	345.0	3.8125	NEAR OCEAN
3759	-118.38	34.18	32.0	3553.0	1060.0	3129.0	1010.0	2.5603	<1H OCEAN
7663	-118.22	33.83	43.0	1426.0	272.0	871.0	276.0	3.7083	<1H OCEAN
70	-122.29	37.81	26.0	768.0	152.0	392.0	127.0	1.7719	NEAR BAY

```
In [12]: # extract the labels corresponding to the test_data
test_label = y.loc[test_data.index]
test_label
```

```
Out[12]: 10960    177200.0
14665    166300.0
3759     174200.0
7663     175200.0
70        82500.0
Name: median_house_value, dtype: float64
```

We pass the `test_data` through the pipeline.
Note that this time we call the `.transform()` method because we called the `.fit()` method before.

```
In [13]: test_data_prepared = full_pipeline.transform(test_data)
test_data_prepared
```

Out[13]:
```
array([[ 8.43785663e-01, -8.81376346e-01,  5.86545474e-01,
         6.43582202e-03,  7.99608506e-02,  8.29840618e-01,
         1.86407359e-01, -3.94668767e-02, -3.28297892e-01,
         9.10015508e-02,  4.31680370e-02,  1.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 1.22771205e+00, -1.33068821e+00,  1.89786762e-01,
        -4.20772935e-01, -4.68972490e-01, -4.81479715e-01,
        -4.06836340e-01, -3.58433767e-02, -1.79884160e-01,
        -4.74275734e-02, -2.55660923e-01,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00],
       [ 5.99468871e-01, -6.84802404e-01,  2.69138504e-01,
         4.18929100e-01,  1.24465420e+00,  1.49737391e+00,
         1.33877012e+00, -6.93429847e-01, -8.03171435e-01,
         9.14295503e-05,  1.47397998e+00,  1.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 6.79245783e-01, -8.48614022e-01,  1.14200767e+00,
        -5.59190412e-01, -6.36039159e-01, -4.88515639e-01,
        -5.87959416e-01, -9.05634775e-02, -1.12498665e-01,
         5.08152592e-03, -3.81328680e-01,  1.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [-1.35007941e+00,  1.01415809e+00, -2.06971950e-01,
        -8.61777454e-01, -9.22439163e-01, -9.09791588e-01,
        -9.79080261e-01, -1.10745410e+00,  2.56360077e-01,
        -8.93649143e-04, -2.57637026e-01,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  1.00000000e+00,
         0.00000000e+00]])
```

In [14]:
```
# predicting
predicted_data = LR_model.predict(test_data_prepared)
predicted_data
```

Out[14]:
```
array([193853.07920142, 222840.04513555, 184762.98229178, 222315.6310429 ,
       131675.72884619])
```

What you see these are the predicted values. Let's compare how they differ from actual values:

In [15]:
```
pd.DataFrame({
    'Prediction': predicted_data,
    'Real price': test_label
})
```

Out[15]:

	Prediction	Real price
10960	193853.079201	177200.0
14665	222840.045136	166300.0
3759	184762.982292	174200.0
7663	222315.631043	175200.0
70	131675.728846	82500.0

Evaluating the Model

In [16]:
```
test_set
```

Out[16]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_prox
20046	-119.01	36.06	25.0	1505.0	NaN	1392.0	359.0	1.6812	47700.0	IN
3024	-119.46	35.14	30.0	2943.0	NaN	1565.0	584.0	2.5313	45800.0	IN
15663	-122.44	37.80	52.0	3830.0	NaN	1310.0	963.0	3.4801	500001.0	NEA
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	495.0	5.7376	218600.0	<1H O
9814	-121.93	36.62	34.0	2351.0	NaN	1063.0	428.0	3.7250	278000.0	NEAR O
...	...	...	...	...	...	...	...	...	...	
15362	-117.22	33.36	16.0	3165.0	482.0	1351.0	452.0	4.6050	263300.0	<1H O
16623	-120.83	35.36	28.0	4323.0	886.0	1650.0	705.0	2.7266	266800.0	NEAR O
18086	-122.05	37.31	25.0	4111.0	538.0	1585.0	568.0	9.2298	500001.0	<1H O
2144	-119.76	36.77	36.0	2507.0	466.0	1227.0	474.0	2.7850	72300.0	IN
3665	-118.37	34.22	17.0	1787.0	463.0	1671.0	448.0	3.5521	151500.0	<1H O

4128 rows × 10 columns

In [17]:

```
# separating predictors
X_test = test_set.drop('median_house_value', axis=1)
X_test
```

Out[17]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
20046	-119.01	36.06	25.0	1505.0	NaN	1392.0	359.0	1.6812	INLAND
3024	-119.46	35.14	30.0	2943.0	NaN	1565.0	584.0	2.5313	INLAND
15663	-122.44	37.80	52.0	3830.0	NaN	1310.0	963.0	3.4801	NEAR BAY
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	495.0	5.7376	<1H OCEAN
9814	-121.93	36.62	34.0	2351.0	NaN	1063.0	428.0	3.7250	NEAR OCEAN
...	...	...	...	...	...	...	...	...	...
15362	-117.22	33.36	16.0	3165.0	482.0	1351.0	452.0	4.6050	<1H OCEAN
16623	-120.83	35.36	28.0	4323.0	886.0	1650.0	705.0	2.7266	NEAR OCEAN
18086	-122.05	37.31	25.0	4111.0	538.0	1585.0	568.0	9.2298	<1H OCEAN
2144	-119.76	36.77	36.0	2507.0	466.0	1227.0	474.0	2.7850	INLAND
3665	-118.37	34.22	17.0	1787.0	463.0	1671.0	448.0	3.5521	<1H OCEAN

4128 rows × 9 columns

In [18]:

```
# separating labels
y_test = test_set['median_house_value'].copy()
y_test
```

Out[18]:

```
20046     47700.0
3024      45800.0
15663    500001.0
20484    218600.0
9814     278000.0
           ...
15362    263300.0
16623    266800.0
18086    500001.0
2144      72300.0
3665     151500.0
Name: median_house_value, Length: 4128, dtype: float64
```

In [19]:
```python
# pass test_set through pipeline
X_test_prepared = full_pipeline.transform(X_test)
X_test_prepared
```

Out[19]:
```
array([[ 0.28534728,  0.1951    , -0.28632369, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.06097472, -0.23549054,  0.11043502, ...,  0.        ,
         0.        ,  0.        ],
       [-1.42487026,  1.00947776,  1.85617335, ...,  0.        ,
         1.        ,  0.        ],
       ...,
       [-1.23041404,  0.78014149, -0.28632369, ...,  0.        ,
         0.        ,  0.        ],
       [-0.08860699,  0.52740357,  0.58654547, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.60445493, -0.66608108, -0.92113763, ...,  0.        ,
         0.        ,  0.        ]])
```

In [20]:
```python
# Prediction
y_predicted = LR_model.predict(X_test_prepared)
```

We use **Root Mean Square Error** (RMSE) to compare prediction and real data:

In [21]:
```python
# Evaluation
from sklearn.metrics import mean_squared_error

lin_mse = mean_squared_error(y_test, y_predicted)
# calculate RMSE
lin_rmse = np.sqrt(lin_mse)
print(lin_rmse)
```

```
72701.32600762133
```

So, `RMSE = $72701` came out. Not bad, but not good either. That is, our model makes an average error of `$72,000` when evaluating houses.

There is no single, universal solution to improve model accuracy. Things we can try:

- Finding better parameters
- Choosing a better model (algorithm).
- Collecting more information, etc.

We will try another model now.

DecisionTree

In [22]:
```
from sklearn.tree import DecisionTreeRegressor

Tree_model = DecisionTreeRegressor()
Tree_model.fit(X_prepared, y)
```

Out[22]:
```
▾ DecisionTreeRegressor
DecisionTreeRegressor()
```

In [27]:
```
# Prediction
y_predicted = Tree_model.predict(X_test_prepared)

# Evaluation
lin_mse = mean_squared_error(y_test, y_predicted)
# calculate RMSE
lin_rmse = np.sqrt(lin_mse)
print(lin_rmse)
```

```
72099.3810531719
```

It is not much different from the previous result.

RandomForest

In [25]:
```
from sklearn.ensemble import RandomForestRegressor

RF_model = RandomForestRegressor()
RF_model.fit(X_prepared, y)
```

Out[25]:
```
▾ RandomForestRegressor
RandomForestRegressor()
```

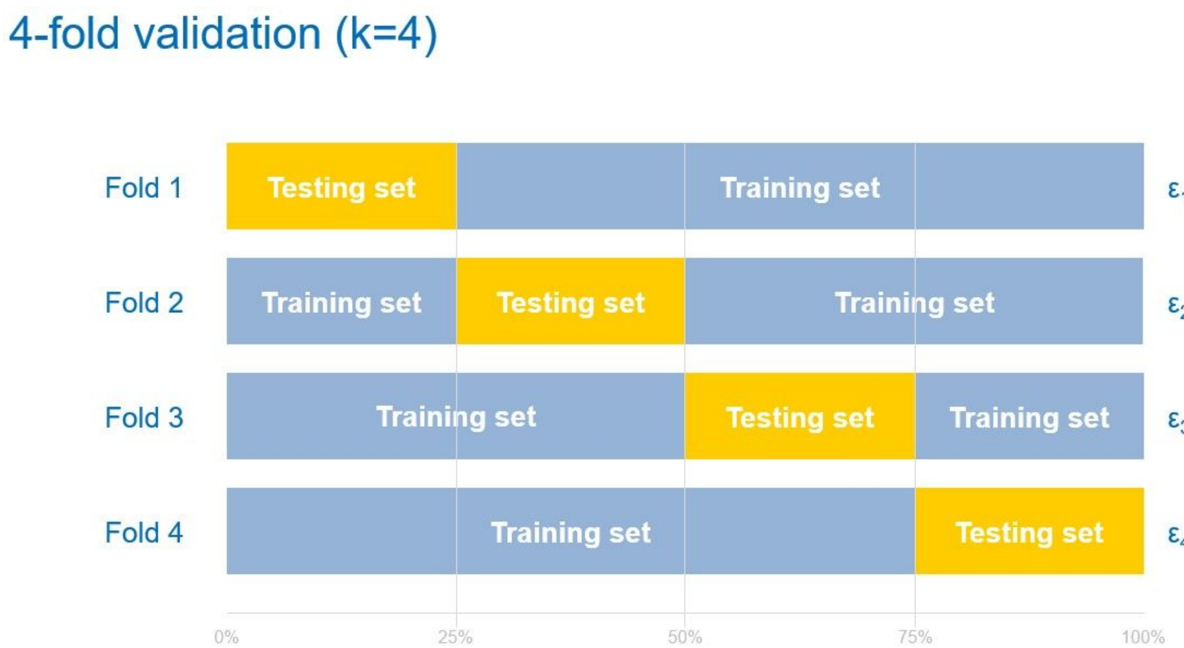
In [34]:
```
# Prediction
y_predicted = RF_model.predict(X_test_prepared)

# Evaluation
lin_mse = mean_squared_error(y_test, y_predicted)
# calculate RMSE
lin_rmse = np.sqrt(lin_mse)
print(lin_rmse)
```

```
50302.97033623816
```

Better than previous result.

Cross-validation

With cross-validation, we can divide the dataset into several parts and train & test the model several times using different parts of the dataset.



© https://mathworks.com/discovery/cross-validation

For cross validation, it is not necessary to divide the data into train and test, it is done by `sklearn` itself.

In [35]:
```
X = df.drop('median_house_value', axis=1)
y = df['median_house_value'].copy()

X_prepared = full_pipeline.transform(X)
```

We create a simple function to display the validation results:

In [36]:
```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Std.dev:", scores.std())
```

LogisticRegression validation

In [46]:
```
# import cross validation score
from sklearn.model_selection import cross_val_score

scores = cross_val_score(LR_model, X_prepared, y, scoring="neg_mean_squared_error", cv=10)
LR_rmse_scores = np.sqrt(-scores)

display_scores(LR_rmse_scores)
```

```
Scores: [84188.51219065 61197.24357613 86752.24346334 62289.14292385
 80540.40041898 68919.39949642 52503.82940087 90910.07884989
 77674.67507925 53941.60539478]
Mean: 71891.71307941682
Std.dev: 13249.525989444988
```

DecisionTree validation

In [48]:
```
scores = cross_val_score(Tree_model, X_prepared, y, scoring="neg_mean_squared_error", cv=10)
LR_rmse_scores = np.sqrt(-scores)

display_scores(LR_rmse_scores)
```

```
Scores: [118471.89596147  72616.0672963   82803.41491496  74062.62707203
  90787.70572359  79654.15982782  68368.94540824 100515.26980479
  95207.73294116  77760.11062863]
Mean:  86024.79295789878
Std.dev:  14572.863972286626
```

RandomForest validation

In [49]:
```python
scores = cross_val_score(RF_model, X_prepared, y, scoring="neg_mean_squared_error", cv=10)
LR_rmse_scores = np.sqrt(-scores)

display_scores(LR_rmse_scores)
```

```
Scores: [95820.18183358 46972.73919635 65442.46141267 56983.29046103
 61284.90178136 60254.09207594 46836.18841014 78995.99562563
 74334.86912631 49559.12380717]
Mean:  63648.38437301737
Std.dev:  14861.892467887077
```

Saving the Model

`joblib` is faster in saving/loading large NumPy arrays, whereas `pickle` is faster with large collections of Python objects. Therefore, if your model contains large NumPy arrays (as the majority of models does), `joblib` should be faster.

Saving with `pickle`

In [51]:
```python
import pickle

filename = 'RF_model.pkl'  # we can give any name to file and extension
with open(filename, 'wb') as file:
    pickle.dump(RF_model, file)
```

Loading the model:

In [52]:
```python
with open('RF_model.pkl', 'rb') as file:
    model = pickle.load(file)
```

Let's test the model:

In [59]:
```
scores = cross_val_score(model, X_prepared, y, scoring="neg_mean_squared_error", cv=5)
LR_rmse_scores = np.sqrt(-scores)
display_scores(LR_rmse_scores)
```

```
Scores: [76771.23270137 64033.48046861 61137.85889808 82093.07560208
 62220.15694484]
Mean: 69251.16092299437
Std.dev: 8531.718797395
```

Saving with `joblib`

`pip install joblib`

In [61]:
```
import joblib

filename = 'RF_model.jbl'  # we can give any name to file and extension
joblib.dump(RF_model, filename)
```

Out[61]:
```
['RF_model.jbl']
```

Loading the model:

In [62]:
```
model = joblib.load('RF_model.jbl')
```

Testing the model:

In [63]:
```
scores = cross_val_score(model, X_prepared, y, scoring="neg_mean_squared_error", cv=5)
LR_rmse_scores = np.sqrt(-scores)

display_scores(LR_rmse_scores)
```

```
Scores: [77606.97987952 64273.62119636 61140.26982307 81551.3988362
 62291.84698398]
Mean: 69372.82334382611
Std.dev: 8485.706347729905
```

Saving `pipeline` with `joblib`:

In [64]:
```
filename = 'pipeline.jbl'
joblib.dump(full_pipeline, filename)
```

Out[64]:
```
['pipeline.jbl']
```

In [67]:
```
print("Done!")
```

```
Done!
```