End-to-end Machine Learning project

Week 5 - Homework 1
CS550 - Machine Learning and Business Intelligence

Ademilton Marcelo da Cruz Nunes (19679)

Table of Content

Links

Google Slides:
https://docs.google.com/presentation/d/1r3xAyv16LOgcbKmez4AI3kF6sWvPXEqhl_KLa8DrYnU/edit?usp=sharing

GitHub:https://github.com/ademiltonnunes/Machine-Learning/tree/main/End%20to%20End%20Project

Introduction

Understanding the process that is done to make predictions through machine learning is essential to implement good models and make good predictions without noise, and bias.

These slides are intended to comment on the main code parts from the end_to_end_machine_learning_project.ipynb file, giving my understanding of the code and executions.

Getting Data

When studying Machine Learning, having data to train the models is just as important as learning about how to generate a model. It is interesting to use real data to apply the acquired knowledge.

For this examples and studying it is being used the California Housing Prices dataset from the 1990 California census.

Download the data from California 1990 Census

```python
import os
import tarfile
import urllib.request

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

```python
fetch_housing_data()
```

```python
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

```python
housing = load_housing_data()
housing.head()
```

Download the data from California 1990 Census

Through the provided link in the code, where the data from census is, the data retrieved from a housing.tgz file is saved in the datasets/housing/ directory in a csv format file.

The function loads that CSV file into an object containing all the data.

	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

Download the data from California 1990 Census

```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Through the info() method, we can get some information about the data type of each column of the csv file.

For example:

- The data type. In this example all columns are numeric, except ocen_proximity.
- The column is null or not.

Dealing with non-numeric fields

```
housing["ocean_proximity"].value_counts()
```

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

Since the ocean_proximity column is the only non-numeric one, we can use some functions to understand this data. For example, the count for each type in that column.

Details about numeric columns

```
housing.describe()
```

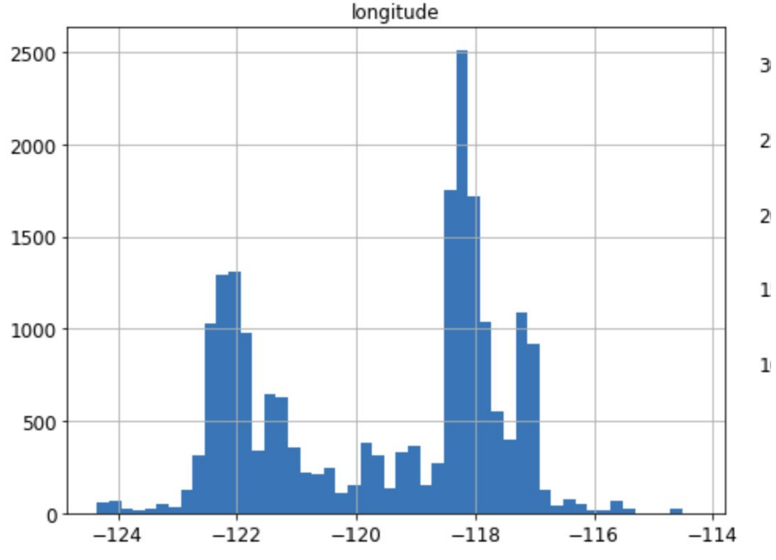
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

Another interesting function is the description(). For numeric columns, this returns important information about that data, like the mean, and quantity, etc.

Histograms

```
[15] %matplotlib inline
     import matplotlib.pyplot as plt
     housing.hist(bins=50, figsize=(20,15))
     save_fig("attribute_histogram_plots")
     plt.show()
```

```
Saving figure attribute_histogram_plots
```



Making column histograms is also an important tool to better understand the data we are dealing with.

The histogram shows the number of instances that have a specific value.

Training and Test set

```
[16] # to make this notebook's output identical at every run
     np.random.seed(42)
```

```
[17] import numpy as np

     # For illustration only. Sklearn has train_test_split()
     def split_train_test(data, test_ratio):
         shuffled_indices = np.random.permutation(len(data))
         test_set_size = int(len(data) * test_ratio)
         test_indices = shuffled_indices[:test_set_size]
         train_indices = shuffled_indices[test_set_size:]
         return data.iloc[train_indices], data.iloc[test_indices]
```

```
[18] train_set, test_set = split_train_test(housing, 0.2)
     print(len(train_set), "train +", len(test_set), "test")

     16512 train + 4128 test
```

```
[19] from zlib import crc32

     def test_set_check(identifier, test_ratio):
         return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

     def split_train_test_by_id(data, test_ratio, id_column):
         ids = data[id_column]
         in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
         return data.loc[~in_test_set], data.loc[in_test_set]
```

In order to make the models, it is necessary to separate the data into training and development sets.

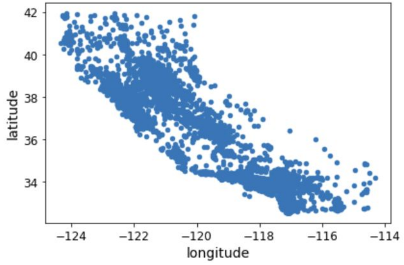
In this code part, 80% of the data will be used for training and 20% for testing.

Discover and visualize the data

```
[37] housing = strat_train_set.copy()
```

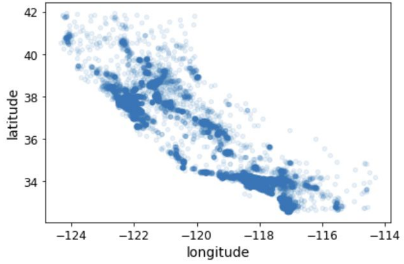
```
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

```
Saving figure bad_visualization_plot
```




```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1
save_fig("better_visualization_plot")
```

```
Saving figure better_visualization_plot
```



Creating a graph to look at the data is also a good technique for understanding and finding patterns, but data can be overloaded.

The definition of alpha makes it possible to do with data that are in areas with higher density

Correlation

```
[47] corr_matrix = housing.corr()
```

```
[48] corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value    1.000000
median_income         0.687151
total_rooms           0.135140
housing_median_age    0.114146
households            0.064590
total_bedrooms        0.047781
population           -0.026882
longitude            -0.047466
latitude             -0.142673
Name: median_house_value, dtype: float64
```

When we analyze data, we have to see each data is more important or more correlate to the prediction we are looking for.

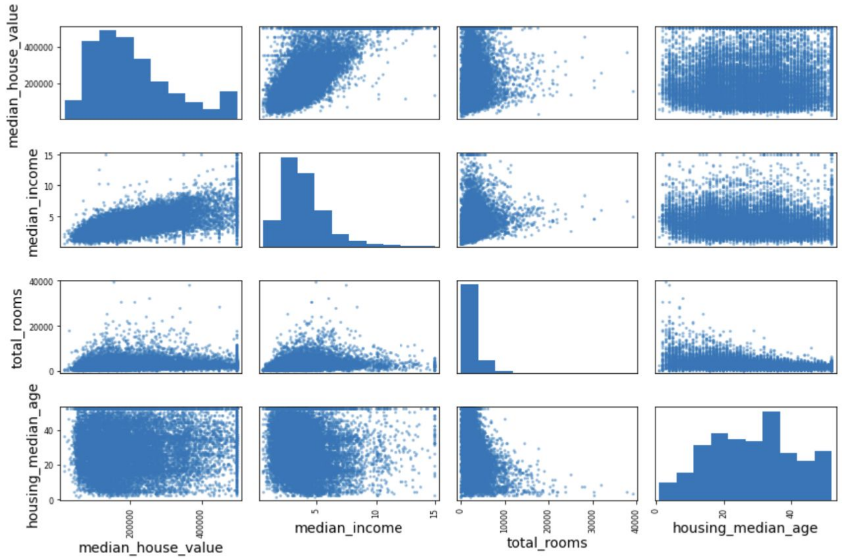
Using the function corr(), we can analyze which data is more correlated. The correlation coefficient goes to -1 to 1. If it is close to 1, means that data is more correlated, but when close to -1 it means less correlations.

Correlations

```python
# from pandas.tools.plotting import scatter_matrix # For older versions of Pandas
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")
```

```
Saving figure scatter_matrix_plot
```



Other way to find correlations is through the scatter_matrix() function. This function can show each attribute against the others or make a histogram of them.

Prepare the data - Clean the data

```python
housing = strat_train_set.drop("median_house_value", axis=1) # drop labels for training set
housing_labels = strat_train_set["median_house_value"].copy()
```

```python
[56] sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
1606	-122.08	37.88	26.0	2947.0	NaN	825.0	626.0	2.9330	NEAR BAY
10915	-117.87	33.73	45.0	2264.0	NaN	1970.0	499.0	3.4193	<1H OCEAN
19150	-122.70	38.35	14.0	2313.0	NaN	954.0	397.0	3.7813	<1H OCEAN
4186	-118.23	34.13	48.0	1308.0	NaN	835.0	294.0	4.2891	<1H OCEAN
16885	-122.40	37.58	26.0	3281.0	NaN	1145.0	480.0	6.3580	NEAR OCEAN

```python
[57] sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity

```python
[58] sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income	ocean_proximity
1606	-122.08	37.88	26.0	2947.0	825.0	626.0	2.9330	NEAR BAY
10915	-117.87	33.73	45.0	2264.0	1970.0	499.0	3.4193	<1H OCEAN
19150	-122.70	38.35	14.0	2313.0	954.0	397.0	3.7813	<1H OCEAN
4186	-118.23	34.13	48.0	1308.0	835.0	294.0	4.2891	<1H OCEAN
16885	-122.40	37.58	26.0	3281.0	1145.0	480.0	6.3580	NEAR OCEAN

```python
[59] median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3
sample_incomplete_rows
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
1606	-122.08	37.88	26.0	2947.0	433.0	825.0	626.0	2.9330	NEAR BAY
10915	-117.87	33.73	45.0	2264.0	433.0	1970.0	499.0	3.4193	<1H OCEAN
19150	-122.70	38.35	14.0	2313.0	433.0	954.0	397.0	3.7813	<1H OCEAN
4186	-118.23	34.13	48.0	1308.0	433.0	835.0	294.0	4.2891	<1H OCEAN
16885	-122.40	37.58	26.0	3281.0	433.0	1145.0	480.0	6.3580	NEAR OCEAN

Sometimes, some data can be null, as the column total_bedrooms. To handle it, we can:

1. Remove data that has it column nullable
2. Remove this column at all
3. Define a standard value for this column

Dealing with non-numeric data

Remove the text attribute because median can only be calculated on numerical attributes:

```
[62] housing_num = housing.drop('ocean_proximity', axis=1)
     # alternatively: housing_num = housing.select_dtypes(include=[np.number])
```

```
[63] imputer.fit(housing_num)

     SimpleImputer(strategy='median')
```

```
[64] imputer.statistics_

     array([-118.51    ,   34.26    ,   29.     , 2119.     ,  433.     ,
            1164.     ,  408.     ,    3.54155])
```

Check that this is the same as manually computing the median of each attribute:

```
[65] housing_num.median().values

     array([-118.51    ,   34.26    ,   29.     , 2119.     ,  433.     ,
            1164.     ,  408.     ,    3.54155])
```

In regression algorithm, it is used only numeric data. We have to deal with non-numeric data in that situation, by not including them.

Dealing with non-numeric data

```
[72] try:
         from sklearn.preprocessing import OrdinalEncoder
     except ImportError:
         from future_encoders import OrdinalEncoder # Scikit-Learn < 0.20
```

```
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

```
array([[1.],
       [4.],
       [1.],
       [4.],
       [0.],
       [3.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

If remove the data is not possible, we can change the text to numbers. For example, where it is “In-land” changes to number 1.

Select and train a model

```
[157] from sklearn.linear_model import LinearRegression

      lin_reg = LinearRegression()
      lin_reg.fit(housing_prepared, housing_labels)

      LinearRegression()
```

```
[158] # let's try the full preprocessing pipeline on a few training instances
      some_data = housing.iloc[:5]
      some_labels = housing_labels.iloc[:5]
      some_data_prepared = full_pipeline.transform(some_data)

      print("Predictions:", lin_reg.predict(some_data_prepared))

      Predictions: [ 85657.90192014 305492.60737488 152056.46122456 186095.70946094
       244550.67966089]
```

With those modification, data is cleaner and well prepared to make predictions.

Evaluate on the Training Set

```
from sklearn.metrics import mean_squared_error

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

```
68627.87390018745
```

```
[162] from sklearn.metrics import mean_absolute_error

      lin_mae = mean_absolute_error(housing_labels, housing_predictions)
      lin_mae
```

```
49438.66860915802
```

```
[163] from sklearn.tree import DecisionTreeRegressor

      tree_reg = DecisionTreeRegressor(random_state=42)
      tree_reg.fit(housing_prepared, housing_labels)
```

```
DecisionTreeRegressor(random_state=42)
```

```
[164] housing_predictions = tree_reg.predict(housing_prepared)
      tree_mse = mean_squared_error(housing_labels, housing_predictions)
      tree_rmse = np.sqrt(tree_mse)
      tree_rmse
```

```
0.0
```

With the real data it is possible to make predictions and evaluate the predictions to see if the model is close to reality.

Fine-tune your model

```
[165] from sklearn.model_selection import cross_val_score

      scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                               scoring="neg_mean_squared_error", cv=10)
      tree_rmse_scores = np.sqrt(-scores)
```

```
[166] def display_scores(scores):
          print("Scores:", scores)
          print("Mean:", scores.mean())
          print("Standard deviation:", scores.std())

      display_scores(tree_rmse_scores)
```

```
Scores: [72831.45749112 69973.18438322 69528.56551415 72517.78229792
 69145.50006909 79094.74123727 68960.045444   73344.50225684
 69826.02473916 71077.09753998]
Mean: 71629.89009727491
Standard deviation: 2914.035468468928
```

```
[167] lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
                                   scoring="neg_mean_squared_error", cv=10)
      lin_rmse_scores = np.sqrt(-lin_scores)
      display_scores(lin_rmse_scores)
```

```
Scores: [71762.76364394 64114.99166359 67771.17124356 68635.19072082
 66846.14089488 72528.03725385 73997.08050233 68802.33629334
 66443.28836884 70139.79923956]
Mean: 69104.07998247063
Standard deviation: 2880.3282098180634
```

Note: we specify `n_estimators=10` to avoid a warning about the fact that the default value is going to change to 100 in Scikit-Learn 0.22.

```
[168] from sklearn.ensemble import RandomForestRegressor

      forest_reg = RandomForestRegressor(n_estimators=10, random_state=42)
      forest_reg.fit(housing_prepared, housing_labels)
```

```
RandomForestRegressor(n_estimators=10, random_state=42)
```

```
[169] housing_predictions = forest_reg.predict(housing_prepared)
      forest_mse = mean_squared_error(housing_labels, housing_predictions)
      forest_rmse = np.sqrt(forest_mse)
      forest_rmse
```

```
22413.454658589766
```

```
from sklearn.model_selection import cross_val_score

forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,
                                scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

```
Scores: [53519.05518628 50467.33817051 48924.16513902 53771.72056856
 50810.90996358 54876.09682033 56012.79985518 52256.88927227
 51527.73185039 55762.56008531]
Mean: 52792.92669114079
Standard deviation: 2262.8151900582
```

```
[171] scores = cross_val_score(lin_reg, housing_prepared, housing_labels, scoring="neg_mean_squared_error", cv=10)
      pd.Series(np.sqrt(-scores)).describe()
```

```
count       10.000000
mean     69104.079982
std       3036.132517
min      64114.991664
25%      67077.398482
50%      68718.763507
75%      71357.022543
max      73997.080502
dtype: float64
```

To improve predictions, one option is to manually change the hyperparameters until you find an optimal combination of hyperparameter values and make better predictions.

Fine-tune your model

```
[172] from sklearn.svm import SVR

      svm_reg = SVR(kernel="linear")
      svm_reg.fit(housing_prepared, housing_labels)
      housing_predictions = svm_reg.predict(housing_prepared)
      svm_mse = mean_squared_error(housing_labels, housing_predictions)
      svm_rmse = np.sqrt(svm_mse)
      svm_rmse
```

```
111095.06635291968
```

```
[173] from sklearn.model_selection import GridSearchCV

      param_grid = [
          # try 12 (3×4) combinations of hyperparameters
          {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
          # then try 6 (2×3) combinations with bootstrap set as False
          {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
        ]

      forest_reg = RandomForestRegressor(random_state=42)
      # train across 5 folds, that's a total of (12+6)*5=90 rounds of training
      grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                                 scoring='neg_mean_squared_error', return_train_score=True)
      grid_search.fit(housing_prepared, housing_labels)
```

```
GridSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4],
                          'n_estimators': [3, 10]}],
             return_train_score=True, scoring='neg_mean_squared_error')
```

The best hyperparameter combination found:

```
[174] grid_search.best_params_
```

```
{'max_features': 8, 'n_estimators': 30}
```

```
[175] grid_search.best_estimator_
```

```
RandomForestRegressor(max_features=8, n_estimators=30, random_state=42)
```

Let's look at the score of each hyperparameter combination tested during the grid search:

```
[176] cvres = grid_search.cv_results_
      for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
          print(np.sqrt(-mean_score), params)
```

```
63895.161577951665 {'max_features': 2, 'n_estimators': 3}
54916.32386349543 {'max_features': 2, 'n_estimators': 10}
52885.86715332332 {'max_features': 2, 'n_estimators': 30}
60075.3680329983 {'max_features': 4, 'n_estimators': 3}
52495.01284985185 {'max_features': 4, 'n_estimators': 10}
50187.24324926565 {'max_features': 4, 'n_estimators': 30}
58064.73529982314 {'max_features': 6, 'n_estimators': 3}
51519.32062366315 {'max_features': 6, 'n_estimators': 10}
49969.80441627874 {'max_features': 6, 'n_estimators': 30}
58895.824998155826 {'max_features': 8, 'n_estimators': 3}
52459.79624724529 {'max_features': 8, 'n_estimators': 10}
49898.98913455217 {'max_features': 8, 'n_estimators': 30}
62381.765106921855 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54476.57050944266 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59974.60028085155 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52754.5632813202 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57831.136061214274 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51278.37877140253 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Executing (9s) Cell > fit() > run_search() >

However, this job can be done by GridSearchCV.

It requires selecting which hyperparameters and values to experiment with, and using cross-validation to evaluate all possible combinations of hyperparameter values.

Fine-tune your model

```python
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distribs,
                                n_iter=10, cv=5, scoring='neg_mean_squared_error', random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

```
RandomizedSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
                   param_distributions={'max_features': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f7e4d6f9100>,
                                        'n_estimators': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f7e4d473280>},
                   random_state=42, scoring='neg_mean_squared_error')
```

```python
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
49117.55344336652 {'max_features': 7, 'n_estimators': 180}
51450.63202856348 {'max_features': 5, 'n_estimators': 15}
50692.53588182537 {'max_features': 3, 'n_estimators': 72}
50783.614493515 {'max_features': 5, 'n_estimators': 21}
49162.89877456354 {'max_features': 7, 'n_estimators': 122}
50655.798471042704 {'max_features': 3, 'n_estimators': 75}
50513.856319990606 {'max_features': 3, 'n_estimators': 88}
49521.17201976928 {'max_features': 5, 'n_estimators': 100}
50302.90440763418 {'max_features': 3, 'n_estimators': 150}
65167.02018649492 {'max_features': 5, 'n_estimators': 2}
```

```python
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

```
array([6.96542523e-02, 6.04213840e-02, 4.21882202e-02, 1.52450557e-02,
       1.55545295e-02, 1.58491147e-02, 1.49346552e-02, 3.79009225e-01,
       5.47789150e-02, 1.07031322e-01, 4.82031213e-02, 6.79266007e-03,
       1.65706303e-01, 7.83480660e-05, 1.52473276e-03, 3.02816106e-03])
```

```python
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # old solution
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

```
[(0.3790092248170967, 'median_income'),
 (0.16570630316895876, 'INLAND'),
 (0.10703132208204354, 'pop_per_hhold'),
 (0.06965425227942929, 'longitude'),
 (0.0604213840080722, 'latitude'),
 (0.054778915018283726, 'rooms_per_hhold'),
 (0.048203121338269206, 'bedrooms_per_room'),
 (0.04218822024391753, 'housing_median_age'),
 (0.015849114744428634, 'population'),
 (0.015554529490469328, 'total_bedrooms'),
 (0.01524505568840977, 'total_rooms'),
 (0.014934655161887776, 'households'),
 (0.006792660074259966, '<1H OCEAN'),
 (0.0030281610628962747, 'NEAR OCEAN'),
 (0.0015247327555504937, 'NEAR BAY'),
 (7.834806602687504e-05, 'ISLAND')]
```

```python
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

Another technique is to use Randomized Search. this is used in the same way as the Grid Search, but instead of trying every possible combination, it evaluates a fixed number of combinations, selecting a random value for each hyperparameter on each iteration.

Conclusion

This project gave a good idea of what a machine learning project looks like. I could see that the biggest work is in the data preparation stage. Machine learning algorithms are important, but having good data is what makes the model better at making predictions.