Cap_2

February 24, 2021

1 Predict the Median Housing Price in California

2 Overview

In this chapter we chose the California Housing Prices dataset from the StatLib repository. This dataset was based on data from the 1990 California census. It is not exactly recent (you could still afford a nice house in the Bay Area at the time), but it has many qualities for learning, so we will pretend it is recent data. We also added a categorical attribute and removed a few features for teaching purposes.

3 Get the Data

3.1 Libraries

[1]:
```
# Get the important libraries

%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
```

3.2 Read the data and Take a Quick Look at the Data Structure

[2]:
```
# Set the file path
path = "./data/housing.csv"

# Read the data from local
raw_data = pd.read_csv(path)

# Get the overview of the raw data
raw_data.head()
```

[2]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	\
0	-122.23	37.88	41.0	880.0	129.0	
1	-122.22	37.86	21.0	7099.0	1106.0	

```
2    -122.24     37.85                52.0       1467.0           190.0
3    -122.25     37.85                52.0       1274.0           235.0
4    -122.25     37.85                52.0       1627.0           280.0

   population  households  median_income  median_house_value ocean_proximity
0       322.0       126.0         8.3252            452600.0        NEAR BAY
1      2401.0      1138.0         8.3014            358500.0        NEAR BAY
2       496.0       177.0         7.2574            352100.0        NEAR BAY
3       558.0       219.0         5.6431            341300.0        NEAR BAY
4       565.0       259.0         3.8462            342200.0        NEAR BAY
```

3.3 Preview some important info

[3]:
```
raw_data.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
```

Analize how many districts belong to each category

[4]:
```
# Get the values of districts of each ocean_proximity category
raw_data["ocean_proximity"].value_counts()
```

[4]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

[5]:
```
# Get the summary of the numerical attributes
raw_data.describe()
```

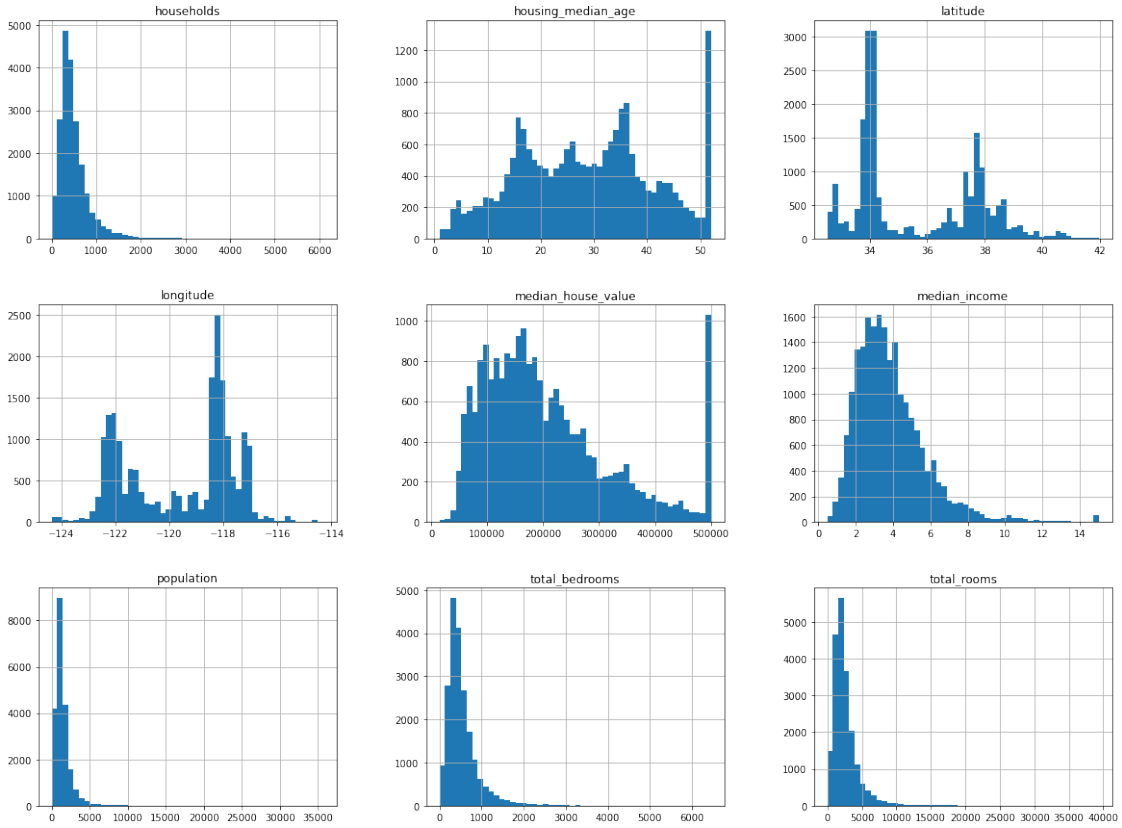
[5]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000     32.540000            1.000000      2.000000
25%     -121.800000     33.930000           18.000000   1447.750000
50%     -118.490000     34.260000           29.000000   2127.000000
75%     -118.010000     37.710000           37.000000   3148.000000
max     -114.310000     41.950000           52.000000  39320.000000

       total_bedrooms    population    households  median_income  \
count    20433.000000  20640.000000  20640.000000   20640.000000
mean       537.870553   1425.476744    499.539680       3.870671
std        421.385070   1132.462122    382.329753       1.899822
min          1.000000      3.000000      1.000000       0.499900
25%        296.000000    787.000000    280.000000       2.563400
50%        435.000000   1166.000000    409.000000       3.534800
75%        647.000000   1725.000000    605.000000       4.743250
max       6445.000000  35682.000000   6082.000000      15.000100

       median_house_value
count        20640.000000
mean        206855.816909
std         115395.615874
min          14999.000000
25%         119600.000000
50%         179700.000000
75%         264725.000000
max         500001.000000
```

Plot the attributes in a Histogram for get another point of view.

[6]:
```
# Plot a histogram
raw_data.hist(bins=50, figsize=(20,15))
plt.show()
```



4 Data preparation

4.1 Split the data into train and test dataset

[7]:
```
data = raw_data
```

Split the data randomly

[8]:
```
from sklearn.model_selection import train_test_split

# random_state parameter makes replicable
train_set, test_set = train_test_split(data, test_size=0.2, random_state=42)
```

Create a categorical attribute for income median values

[9]:
```
data["income_cat"] = pd.cut(data["median_income"],
                            bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                            labels=[1, 2, 3, 4, 5])
```

Split the data using stratified sampling

[10]:
```
from sklearn.model_selection import StratifiedShuffleSplit

# Split the data
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(data, data["income_cat"]):
    strat_train_set = data.loc[train_index]
    strat_test_set = data.loc[test_index]
```

Drop the categorical attirbute created before, and return to the original state

[11]:
```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

5 Discover and Visualize the data to gain Insights

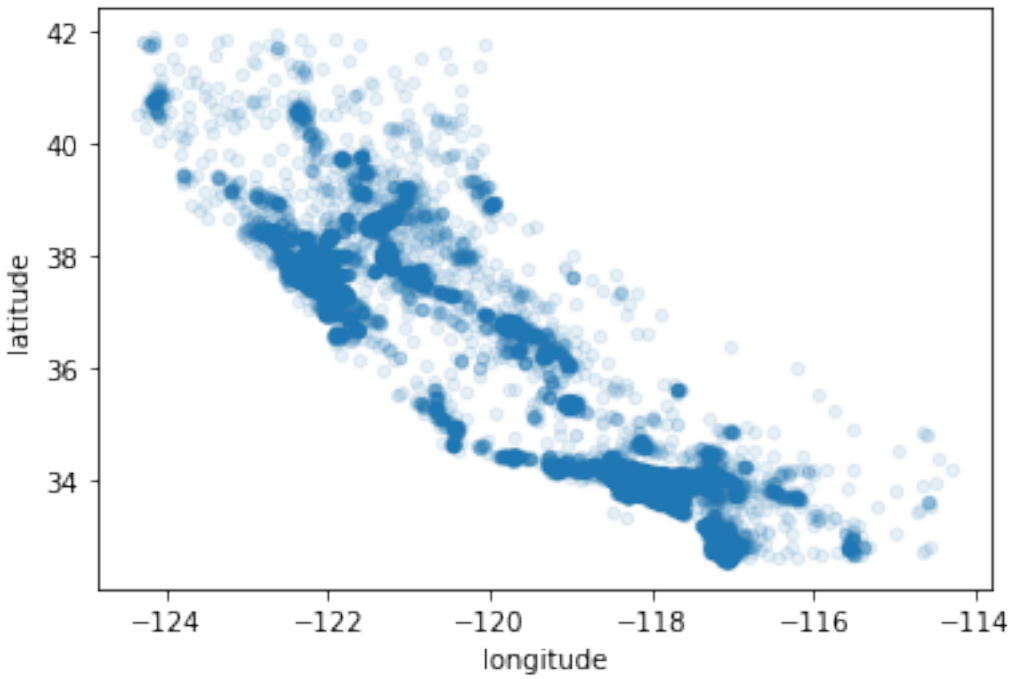
[12]:
```
# Make a copy of the training data
housing = strat_train_set.copy()
```

5.1 View geographical data

Since there is a geographical information, it is a good idea to create a scatterplot of all districts to visualize the data. The value of alpha = 0.1 makes it much easier to visualize the places where there is a high density of data points.

[13]:
```
# Plotting geographical data
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

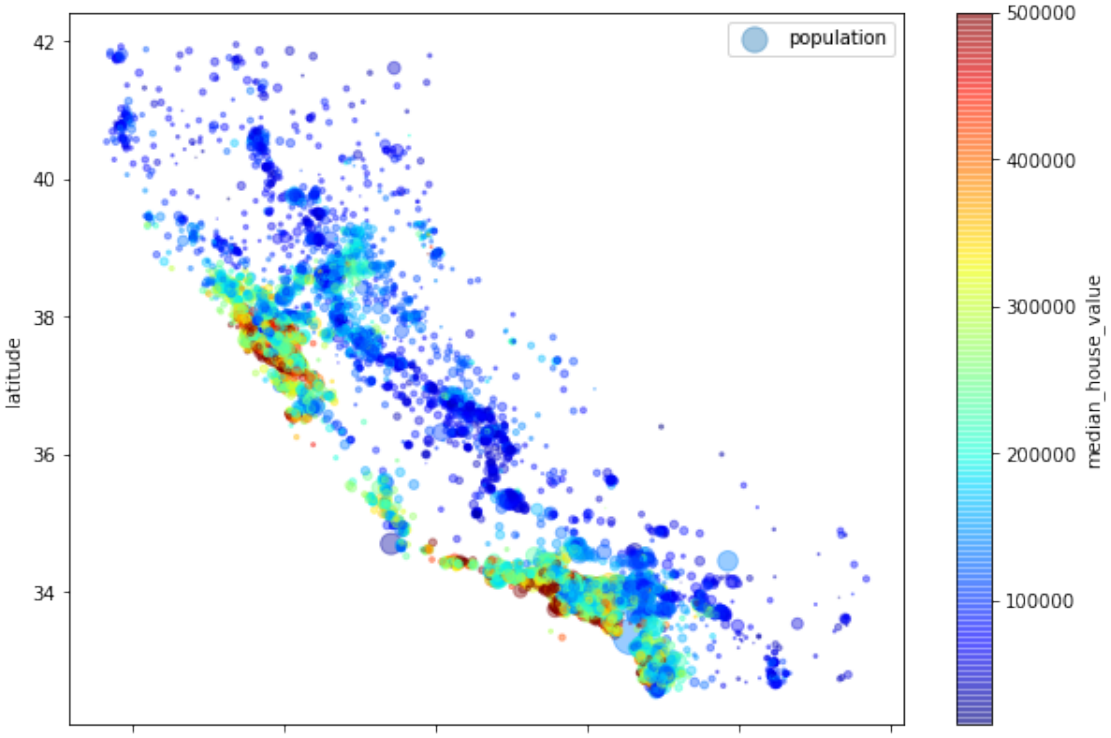
[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x20816f17880>
```



Now let´s look at the prices information, in this ocation the we will play with some options for obtain a better representation of the data. The radius of each circle (Option s) represents the distric's population, the color represents the price (Option c). At the same time a predefined color map it is used (Option cmap) caled jet.

[14]:
```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

[14]: <matplotlib.legend.Legend at 0x20816ffbb20>



This image tell us that the housing pricing are very much related to the location (e.g, close to the ocean) and to the population density. It will be probably be useful to use a clustering algorithm to detect the main clusters and add new features that measure the proximity to the cluster centers.

5.2 Looking for Correlations

In this section it will be calculated the correlations, in particular the "standard correlation coefficient (Pearson's r)" this values measures how an attribute affects each other.

[15]:
```
# Get the correlation values between the attributes
corr_matrix = housing.corr()
```

Now let's look at how much each attribute correlates with the median house value:

[16]:
```
# Get the correlations vector related to the target variable
corr_matrix["median_house_value"].sort_values(ascending=False)
```

[16]:
```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
```

```
longitude           -0.047432
latitude            -0.142724
Name: median_house_value, dtype: float64
```

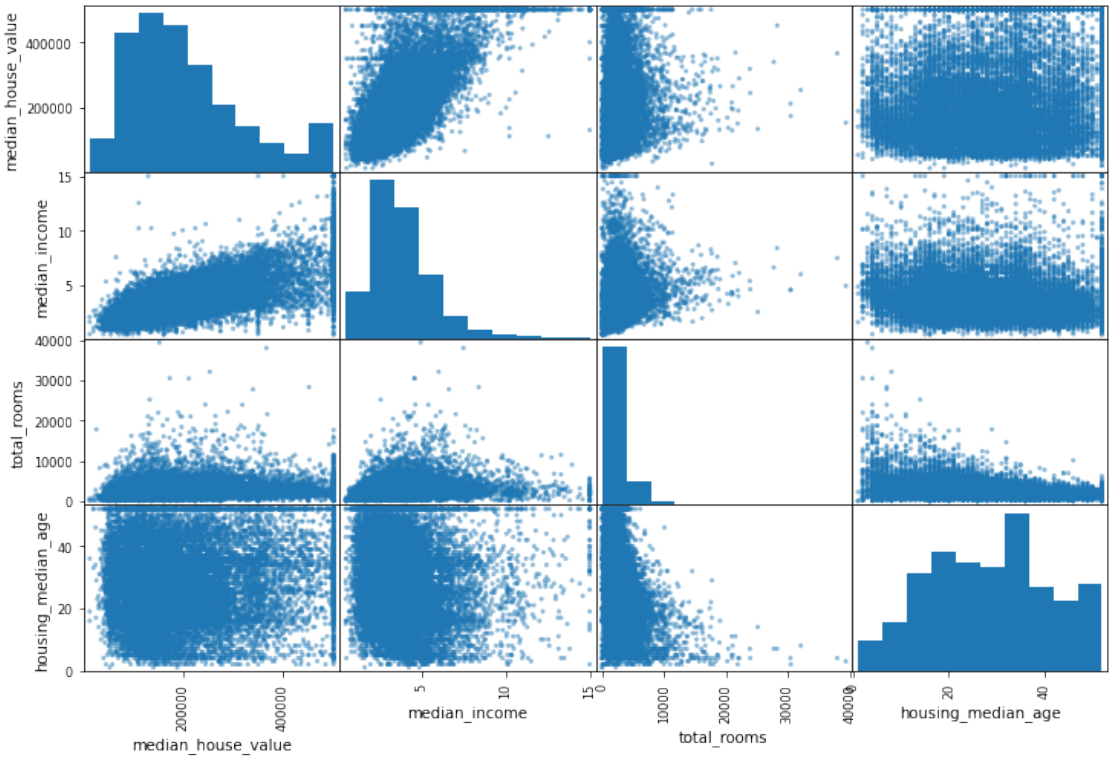
It is clear that the most influential variable in the houses values is the value of the median_income variable.

Another way to check for correlation between attributes is to use Pandas' scatter_matrix function, which plots every numerical attribute against every other numerical attribute (target included). In this plot the main diagonal shows a histogram of each attribute, due to the corresponding correlation against itself it is one and this factor would not be very useful.

[17]:
```
from pandas.plotting import scatter_matrix

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

[17]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000002081718B850>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000020816EFB670>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000020816ECE430>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000002081716D6D0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x00000208170425B0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000020817AA7970>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000020817AA7A60>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x00000208170A8F10>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x0000020816FC2760>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000020816F73BB0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000020816EC90A0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000002081707A490>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x00000208171358E0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000020816F30D30>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x00000208171AD1C0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x00000208171EA610>]],
      dtype=object)
```



6 Attribute Combinations

In this section we will try out various attribute combinations, using the actual attribute combined with other and together make a new one. For example can be intresting to compute the value os the number of rooms in each huoseholds, or the total number of bedrooms per room. In the next chunk, we will compute the new attributes.

[18]:
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]

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

[18]:
```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
```

```
population                -0.026920
longitude                 -0.047432
latitude                  -0.142724
bedrooms_per_room         -0.259984
Name: median_house_value, dtype: float64
```

As you can see, the new value of bedrooms_per_room it it much more correlated with th median value than the total number or rooms/bedrooms, apparently, houses with a lower bedroom/room ratio tend to be more expensive.

7 Data Preparation

The first step in this section will be come back toa a clean training dataset, and let's separate the predictors and the labels since we don't necessary want to apply same transfomrations to the predictors and the target values.

[19]:
```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Tip: Instead of just doing manually, write functions, it will be better for makes reproducible and scalable. Also, you can test all your transformation and mixe it eachother and compare which works better for your code. Finally you can use these functions in a live system to transform the new data before fedding the algorirthm.

7.1 Data Cleaning

Firstly, we will trate the missing values, as we noticed earlier the feature *total_bedrooms* has some missing values, so let's create some functions to take care of them.

Scikit-Learn provides a handy class to take care of missing values: SimpleImputer. Here is how to use it. First, you need to create a SimpleImputer instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

[20]:
```
from sklearn.impute import SimpleImputer

# Create the imputer
imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes, so we need to create a copy of our data without categorical features.

[21]:
```
# Drop the categorical variable
housing_num = housing.drop("ocean_proximity", axis=1)
```

Now we can fit the imputer with the numerical dataframe.

[22]:
```
imputer.fit(housing_num)
```

[22]:
```
SimpleImputer(strategy='median')
```

Now it is time to transform the training dataset by replacing missing values by the median value.

[23]:
```
X = imputer.transform(housing_num)
```

The result is a plain NumPy array, if you wanto to put it back into a Pandas dataframe it is simple.

[24]:
```
# Create a Pandas dataframe from NumPy array.
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
housing_tr.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16512 entries, 0 to 16511
Data columns (total 8 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   total_bedrooms      16512 non-null  float64
 5   population          16512 non-null  float64
 6   households          16512 non-null  float64
 7   median_income       16512 non-null  float64
dtypes: float64(8)
memory usage: 1.0 MB
```

7.2 Text and Categorical Attributes

Most Machine Learning algorithms prefer to work with numbers anyway, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's OrdinalEncoder class

[25]:
```
from sklearn.preprocessing import OrdinalEncoder

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

[25]:
```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

Another solutions could be use one hot encoding, which create one binary attribute per category, avoiding the issue of distances with these non-ordinal attribute. Scikit-Learn also provides a OneHotEncoder class to convert categorical values into one-hot vectors

[26]:
```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

[26]:
```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

A really good news it it that the result is stored in a SciPy sparse matrix instead of a NumPy array. This is very useful when you have categorical variables with thousands of categories, after one_hot_encoding we get a matrix with thousands of columns, and is full zeros except for a single one per row. Using up tons of extra space and memory to store zeros, so instead a sparse matrix only store the location of non zero elements.

7.3 Custom Transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes. You will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not inheritance), all you need is to create a class and implement three methods: fit() (returning self), transform(), and fit_transform(). You can get the last one for free by simply adding TransformerMixin as a base class. Also, if you add BaseEstimator as a base class (and avoid *args and* kargs in your constructor) you will get two extra methods (get_params() and set_params()) that will be useful for automatic hyperparameter tuning. For example, here is a small transformer class that adds the combined attributes we discussed earlier:

[27]:
```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
```

```
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

In this example the transformer has one hyperparameter, add_bedrooms_per_room, set to True by default (it is often helpful to provide sensible defaults). This hyperparameter will allow you to easily find out whether adding this attribute helps the Machine Learning algorithms or not. More generally, you can add a hyperparameter to gate any data preparation step that you are not 100% sure about. The more you automate these data preparation steps, the more combinations you can automatically try out, making it much more likely that you will find a great combination (and saving you a lot of time).

7.4 Feature Scaling

Feature scaling it is one of the most important transfromations we need to apply to our data, mostly ML algorithms works better with scaled data, when the input numerical attributes have very different scales some variables can get more influence in the results due to the scale of the data. The scaling of the data will be implemented in the next section.

7.5 Transformation Pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations. Here is a small pipeline for the numerical attributes:

[28]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformations to each column. In version 0.20, Scikit-Learn introduced the ColumnTransformer for this purpose, and the good news is that it works great with Pandas DataFrames. Let's use it to apply all the transformations to the housing data:

[29]:
```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
```

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

housing_prepared = full_pipeline.fit_transform(housing)
```

Here is how this works: first we import the ColumnTransformer class, next we get the list of numerical column names and the list of categorical column names, and we construct a ColumnTransformer. The constructor requires a list of tuples, where each tuple contains a name21, a transformer and a list of names (or indices) of columns that the transformer should be applied to. In this example, we specify that the numerical columns should be transformed using the num_pipeline that we defined earlier, and the categorical columns should be transformed using a OneHotEncoder. Finally, we apply this ColumnTransformer to the housing data: it applies each transformer to the appropriate columns and concatenates the outputs along the second axis (the transformers must return the same number of rows).

8 Select and Train a Model

In this section as final part of the project, we will select and train a Machine Learning model.

8.1 Training and Evaluating on the Training Set

Now things are going to be simpler than you might think, it is due to the previous work. Let's first train a Linear Regresion model.

[30]:
```
from sklearn.linear_model import LinearRegression

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

[30]:
```
LinearRegression()
```

You now have a working Linear Regression model. Let's try it out on a few instances from the training set:

[31]:
```
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))
print("Labels:", list(some_labels))
```

```
Predictions: [210644.60459286 317768.80697211 210956.43331178  59218.98886849
 189747.55849879]
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

It works, although the predictions are not exactly accurate (e.g., the first prediction is off by close to 40%!). Let's measure this regression model's RMSE on the whole training set using Scikit-Learn's mean_squared_error function:

[32]:
```
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
```

[32]:
```
68628.19819848923
```

Okay, this is better than nothing but clearly not a great score. This is a example of a model underfitting the training data. The main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model. Let's try the first options selecting a differente model.

8.2 Decision Tree Regressor

This time we will try a DecisionTreeRegressor.

[33]:
```
from sklearn.tree import DecisionTreeRegressor

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

[33]:
```
DecisionTreeRegressor()
```

[34]:
```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

[34]:
```
0.0
```

Absolutely amazing!!! Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data. How can you be sure? As we saw earlier, you don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use part of the training set for training, and part for model validation.

8.3 Evaluation Using Cross-Validation

Scikit-Learn's cross-validation feature it is a great tool. The following code performs K-fold cross-validation: it randomly splits the training set into 10 distinct subsets called folds, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores

[35]:
```
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)
```

The results are:

[36]:
```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
```

```
Scores: [70080.5626825  67867.20975265 72334.53796405 69134.31145699
 71145.72499385 75374.1719049  70049.00668828 70738.98205283
 75387.40562744 69711.0580683 ]
Mean: 71182.29711918108
Standard deviation: 2379.6435941800673
```

Now this model doesn't look as good as it did earlier, in fact seems to perform worse than the Linear Regression model. Notice that cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is. You would not have this information if you just used one validation set, but at the same time cross-validation comes at the cost of the training the model several times, so it is not always possible.

8.4 Random Forest Regressor

Now let's try one last model, RandomForestRegressor.

[37]:
```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)

housing_predictions = forest_reg.predict(housing_prepared)
forest_rmse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_rmse)
forest_rmse
```

[37]:
```
18680.369219500815
```

This is much better: Random Forests look very promising. However, note that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set.

9 Tune the model

In case that you have a shortlist of models, yo now need to fine/tune them. In this section we will discover a few ways to to that.

9.1 Grid Search

Instead of to fiddle with the hyperparameters manually you should get Scikit-learn's GridSearchCV to search for you. With this approrach all you need to do is telling to the model which hyperparameters you want to play with, and what values to try uot, them it will evaluate all the posible comninations between the values and hyperparameters, using as before Cross-Validation.

The folowing code searches for the best combination of hyperparameter values for the RandomForestRegresssor.

[38]:
```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

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)
```

[38]:
```
GridSearchCV(cv=5, estimator=RandomForestRegressor(),
             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')
```

This param_grid tells Scikit-Learn to first evaluate all $3 \times 4 = 12$ combinations of n_estimators and max_features hyperparameter values specified in the first dict (don't worry about what these hyperparameters mean for now; they will be explained in Chapter 7), then try all $2 \times 3 = 6$ combinations of hyperparameter values in the second dict, but this time with the bootstrap hyperparameter set to False instead of True (which is the default value for this hyperparameter).

All in all, the grid search will explore $12 + 6 = 18$ combinations of RandomForestRegressor hyperparameter values, and it will train each model five times (since we are using five-fold cross validation). In other words, all in all, there will be $18 \times 5 = 90$ rounds of training! It may take quite a long time, but when it is done you can get the best combination of parameters like this:

[39]:
```
grid_search.best_params_
```

[39]:
```
{'max_features': 8, 'n_estimators': 30}
```

Since the results involve the maximun values that were evaluated, you should probably try searching again woth higher values, since the score may continue to improve, but this time we will stop here.

We can also get the best estimator directly

[40]:
```
grid_search.best_estimator_
```

[40]:
```
RandomForestRegressor(max_features=8, n_estimators=30)
```

[41]:
```
cvres = grid_search.cv_results_

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

```
64119.04482863937 {'max_features': 2, 'n_estimators': 3}
55262.82044462242 {'max_features': 2, 'n_estimators': 10}
53019.73319852918 {'max_features': 2, 'n_estimators': 30}
60422.14955841005 {'max_features': 4, 'n_estimators': 3}
52566.80393350787 {'max_features': 4, 'n_estimators': 10}
50369.316336569136 {'max_features': 4, 'n_estimators': 30}
59362.41640853842 {'max_features': 6, 'n_estimators': 3}
52101.52279311987 {'max_features': 6, 'n_estimators': 10}
50259.17517066402 {'max_features': 6, 'n_estimators': 30}
59987.681820357364 {'max_features': 8, 'n_estimators': 3}
52624.57943108585 {'max_features': 8, 'n_estimators': 10}
49966.51138708826 {'max_features': 8, 'n_estimators': 30}
62053.6194199234 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54757.9469410517 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59006.17929563765 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52730.15975808623 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57977.43294611441 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51864.4425485659 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Note : If GridSearchCV is initialized with refit=True (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea since feeding it more data will likely improve its performance.

As we can see the RSME score for the winner combination is 49847.93219931037, which is slightly better than the score we got earlier using the default hyperparameter values (over 50k).

9.2 Analyze the best model and their errors

You will often gain good insights on the problem by inspecting the best models. For example, the RandomForestRegressor can indicate the relative importance of each attribute for making accurate predictions:

[42]:
```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

[42]:
```
array([6.93041725e-02, 6.71490692e-02, 4.53564217e-02, 1.60844253e-02,
       1.48075969e-02, 1.52770608e-02, 1.48489683e-02, 3.82252388e-01,
       4.45655052e-02, 1.13836190e-01, 4.76698974e-02, 8.86100491e-03,
```

```
       1.53658578e-01, 1.33370279e-04, 2.40561102e-03, 3.78974052e-03])
```

Let's display these importance scores next to their corresponding attribute names:

[43]:
```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
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)
```

[43]:
```
[(0.38225238839183556, 'median_income'),
 (0.1536585779590582, 'INLAND'),
 (0.11383618972364792, 'pop_per_hhold'),
 (0.06930417248525109, 'longitude'),
 (0.06714906923165523, 'latitude'),
 (0.04766989735748221, 'bedrooms_per_room'),
 (0.04535642170799949, 'housing_median_age'),
 (0.044565505191646024, 'rooms_per_hhold'),
 (0.01608442528372946, 'total_rooms'),
 (0.015277060808083857, 'population'),
 (0.014848968276926628, 'households'),
 (0.014807596852430804, 'total_bedrooms'),
 (0.00886100490767259, '<1H OCEAN'),
 (0.0037897405200039967, 'NEAR OCEAN'),
 (0.00240561102355084, 'NEAR BAY'),
 (0.0001333702790261435, 'ISLAND')]
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one ocean_proximity category is really useful, so you could try dropping the others).

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem (adding extra features or, on the contrary, getting rid of uninformative ones, cleaning up outliers, etc.).

10 Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. Now is the time to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from your test set, run your full_pipeline to transform the data (call transform(), not fit_transform(), you do not want to fit the test set!), and evaluate the final model on the test set:

[44]:
```
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)
final_rmse
```

[44]: 47613.458191772326

The performance will usually be slightly worse than what you measured using cross-validation if you did a lot of hyperparameter tuning (because your system ends up fine-tuned to perform well on the validation data, and will likely not perform as well on unknown datasets). It is not the case in this example, but when this happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

11 Future work

- Try a Support Vector Machine regressor (sklearn.svm.SVR)
- Implement a Randomized Search in Tune the model section