main

August 30, 2018

1 Data Import and Basic Overview

```
In [1]: # Load the data
        import pandas as pd
        df = pd.read_csv("housing.csv")
        df.head()
```

```
Out[1]:    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
```

```
In [2]: # Get some info about the data
        # total_bedrooms has some (207) missing values
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
```

```
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

In [3]:
```
# Since ocean_proximity is a categorcial variable, we want to see how many categories
df["ocean_proximity"].value_counts()
```

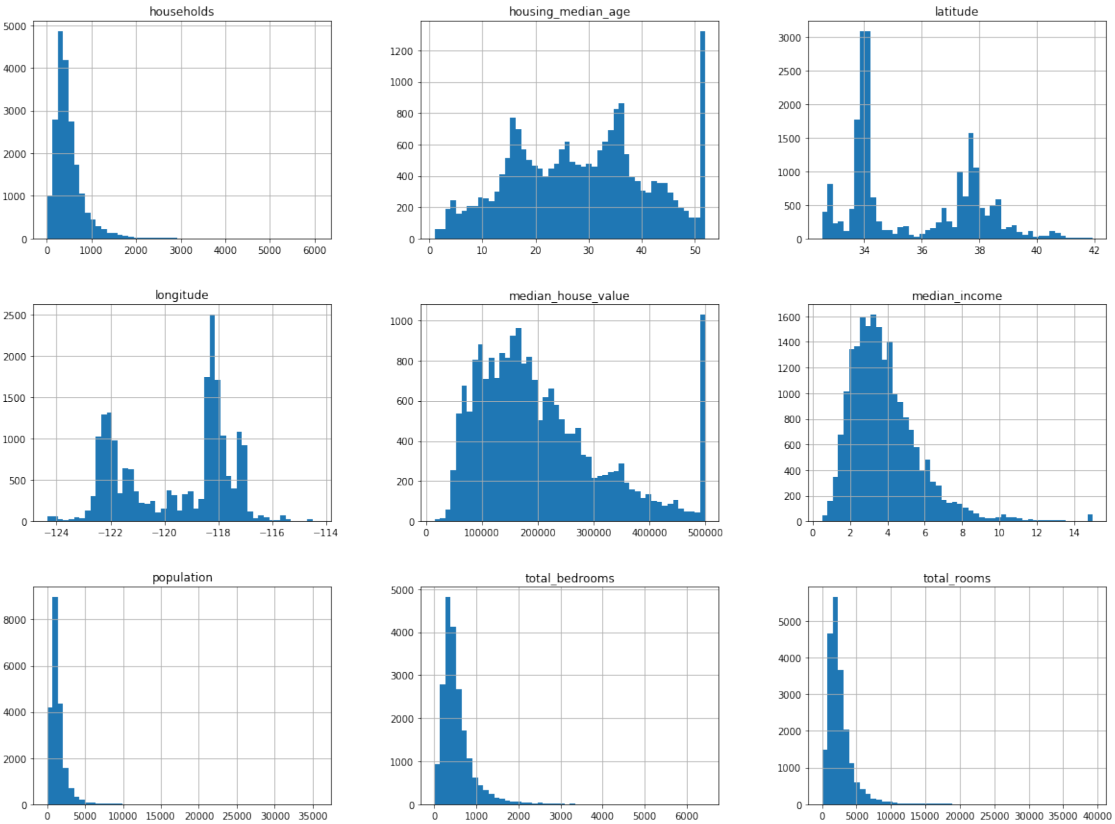
Out[3]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

In [4]:
```
# The describe() method shows a summary of the numerical attributes. Null values are i
df.describe()
```

Out[4]:
```
          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
```

In [5]:
```
# Now we visualize the numeric values with histograms, which shows the counts for cert
# Note: matplotlib inline only works in a Jupyter notebook This tells Jupyter to set u
```

```
# so it uses Jupyters own backend. Plots are then rendered within the notebook itself.
%matplotlib inline
import matplotlib.pyplot as plt
df.hist(bins = 50, figsize = (20, 15))
plt.show()
```




```
In [6]: # Split the data with sklearn function train_test_split
        from sklearn.model_selection import train_test_split
        train_set, test_set = train_test_split(df, test_size=0.2, random_state=42)
        print(len(train_set), " train +", len(test_set), " test")
```

```
16512  train + 4128  test
```

```
In [7]: # The following code creates an income category attribute by dividing the median incom
        # (to limit the number of income categories), and rounding up using ceil (to have disc
        # all the categories greater than 5 into category 5:
        import numpy as np

        df["income_cat"] = np.ceil(df["median_income"] / 1.5)
        df["income_cat"].where(df["income_cat"] < 5, 5.0, inplace = True)
```

```
In [8]: # Now we are ready to do stratified sampling based on the income category. For this we
        # StratifiedShuffleSplit class:

        from sklearn.model_selection import StratifiedShuffleSplit

        ssplit = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

        for train_index, test_index in ssplit.split(df, df["income_cat"]):
            strat_train_set = df.loc[train_index]
            strat_test_set = df.loc[test_index]
```

```
In [9]: # Let's see if it worked.
        print(df["income_cat"].value_counts() / len(df))
        print("And the sum is, of course, ", sum(df["income_cat"].value_counts() / len(df)))
```

```
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
And the sum is, of course,  1.0
```

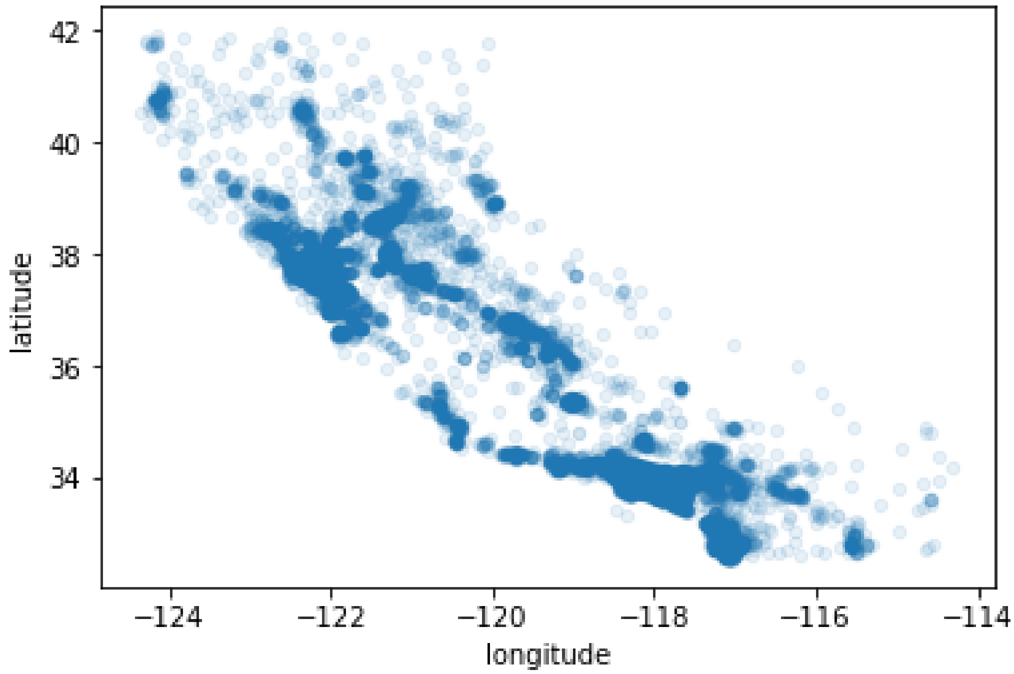
```
In [10]: # Now we should remove the "income_cat" column so that the data is back to its origin
         #for set in (strat_train_set, strat_test_set):
         #    set.drop(["income_cat"], axis=1, inplace=True)
```

2 Exploratory Data Analysis: Plotting and Correlation Matrices

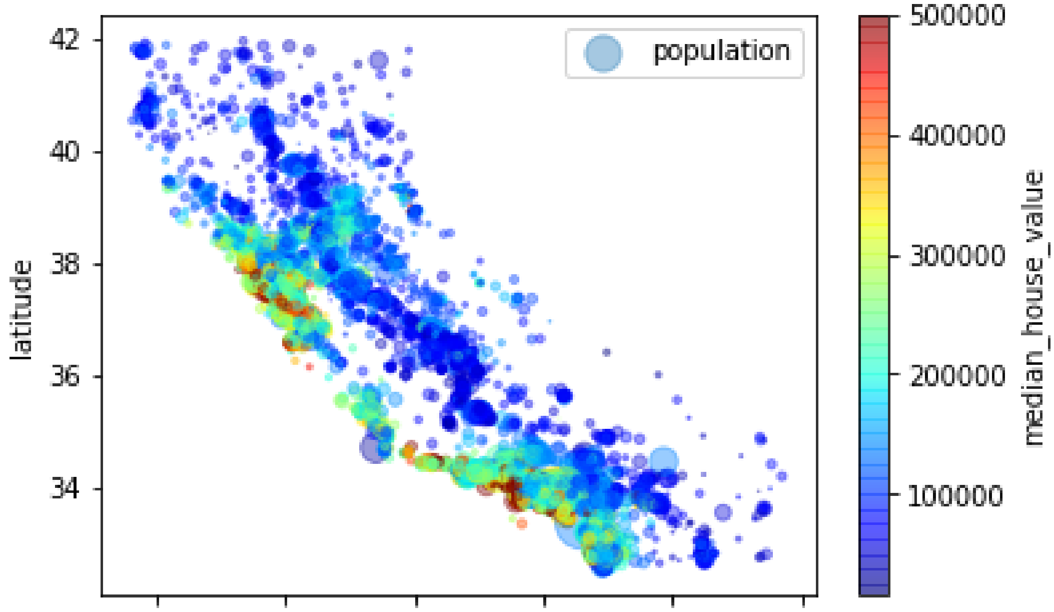
```
In [11]: # First, we make sure you have put the test set aside and you are only exploring the
         df = strat_train_set.copy()
```

```
In [12]: # Since geographical information is given, we can create a scatterplot of all distric
         # we can see the density better
         df.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

```
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x237e5825b00>
```

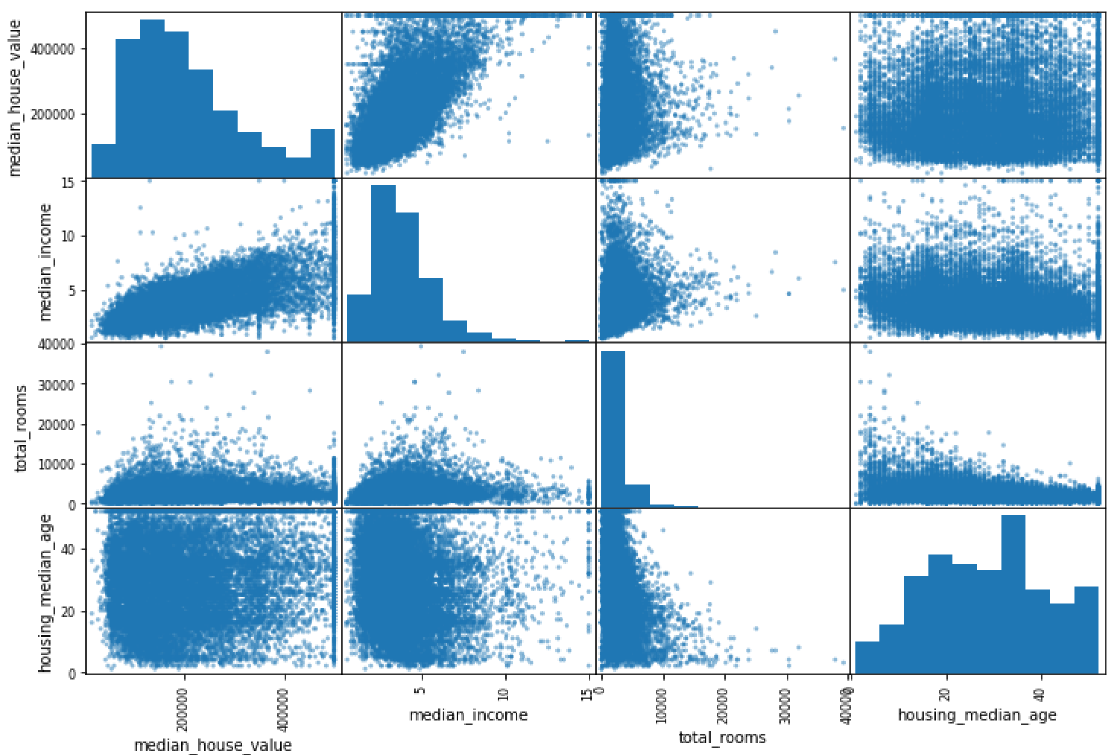



```
In [13]: # In the next plot we look at housing prices geographically. We will use a predefined
         # from blue to red
         df.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
                s=df["population"]/100, label="population",
                c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True)
         plt.legend()

Out[13]: <matplotlib.legend.Legend at 0x237e592d2b0>
```

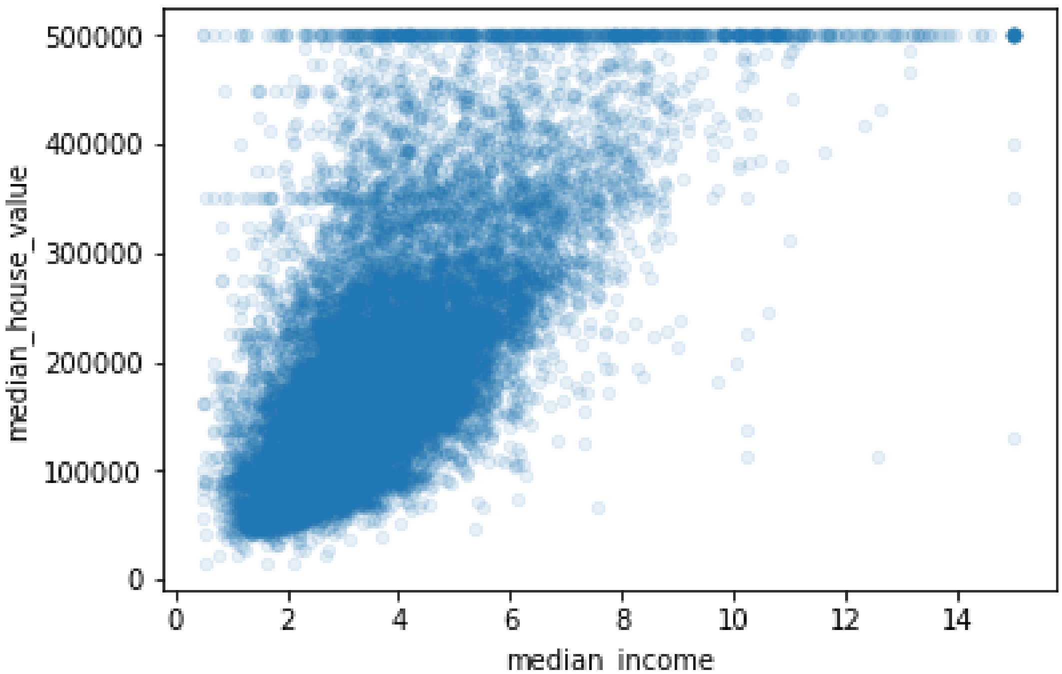



```
In [14]: # Next we compute a correlation matrix for all features.
         corr_matrix = df.corr()
         # ... and look how they correlate to the median house value
         corr_matrix["median_house_value"].sort_values(ascending=False)

Out[14]: median_house_value    1.000000
         median_income         0.687160
         income_cat            0.642274
         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

In [15]: # We now have a look at the most promising features in explaining median house value
         from pandas.tools.plotting import scatter_matrix

         attributes = ["median_house_value", "median_income", "total_rooms", "housing_median_ag
         scatter_matrix(df[attributes], figsize=(12, 8))

C:\Users\Clem\Anaconda3\lib\site-packages\ipykernel_launcher.py:5: FutureWarning: 'pandas.tools
  """
```

```
Out[15]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x00000237E3CBE320>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5ED1F28>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5D55588>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5D7CBA8>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5DAD278>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5DAD2B0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5E00F98>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5E31668>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5F07CF8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E5F3A3C8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E6341A58>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E6374128>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x00000237E639C7B8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E63C5E48>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E3B16390>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000237E30AE630>]],
               dtype=object)
```




```
In [16]: # The most promising attribute to predict the median house value is the median
         # income, so lets zoom in on their correlation scatterplot
         df.plot(kind="scatter", x="median_income", y="median_house_value", alpha=0.1)

Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x237e324dd68>
```



3 Experimenting with Attribute Combinations

In [17]:
```
# Creating new, more purposeful, variables
df["rooms_per_household"] = df["total_rooms"]/df["households"]
df["bedrooms_per_room"] = df["total_bedrooms"]/df["total_rooms"]
df["population_per_household"] = df["population"]/df["households"]

# Look at the correlation matrix again
corr_matrix = df.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)

# Apparently houses with a lower bedroom/room ratio tend to be more expensive.
```

Out[17]:
```
median_house_value          1.000000
median_income               0.687160
income_cat                  0.642274
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
```

4 Prepare the Data for Machine Learning Algorithms: Cleaning, Scaling, Encoding

In [18]:
```
# First, we start off with a fresh training set
df = strat_train_set.drop("median_house_value", axis=1)
df_labels = strat_train_set["median_house_value"].copy()
```

In [19]:
```
# DATA CLEANING
# We have three options how to deal with missing data:
# housing.dropna(subset=["total_bedrooms"]) # gets rid of missing records
# housing.drop("total_bedrooms", axis=1) # gets rid of the missing attributes
# median = housing["total_bedrooms"].median()
# housing["total_bedrooms"].fillna(median) # computes median imputation

# We can also use scikit-learn to perform median imputation
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")

# But since the median can only be computed on numerical attributes, we need to creat
# copy of the data without the text attribute ocean_proximity:
df_num = df.drop("ocean_proximity", axis=1)

# Now we can fit the imputer instance to the training data using the fit() method:
imputer.fit(df_num)

# Now you can use this trained imputer to transform the training set by replacing
# missing values by the learned medians:
X = imputer.transform(df_num)

# Finally, we can put it back into a DataFrame
df_tr = pd.DataFrame(X, columns = df_num.columns)
```

In [20]:
```
# HANDLING TEXT AND CATEGORICAL ATTRIBUTES
# We need to encode categorical attributes
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()

df_cat = df["ocean_proximity"]
df_cat_encoded = encoder.fit_transform(df_cat)
df_cat_encoded
```

Out[20]:
```
array([0, 0, 4, ..., 1, 0, 3], dtype=int64)
```

In [21]:
```
print(encoder.classes_)
```

```
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

```
In [22]: # Since our categories aren't ordinal, One Hot Encoding would be the better choice
         from sklearn.preprocessing import OneHotEncoder
         encoder = OneHotEncoder()
         df_cat_1hot = encoder.fit_transform(df_cat_encoded.reshape(-1,1))
         df_cat_1hot

Out[22]: <16512x5 sparse matrix of type '<class 'numpy.float64'>'
                 with 16512 stored elements in Compressed Sparse Row format>

In [23]: # Due to data saving, SciPy saves the result of the encoding in a sparse matrix, which
         # about the 1s. We can convert it to a 2D array:
         df_cat_1hot.toarray()

Out[23]: array([[1., 0., 0., 0., 0.],
                [1., 0., 0., 0., 0.],
                [0., 0., 0., 0., 1.],
                ...,
                [0., 1., 0., 0., 0.],
                [1., 0., 0., 0., 0.],
                [0., 0., 0., 1., 0.]])

In [24]: # We can apply both transformations (from text categories to integer categories, then
         # from integer categories to one-hot vectors) in one shot using the LabelBinarizer cla
         from sklearn.preprocessing import LabelBinarizer
         encoder = LabelBinarizer()
         df_cat_1hot = encoder.fit_transform(df["ocean_proximity"])
         df_cat_1hot

Out[24]: array([[1, 0, 0, 0, 0],
                [1, 0, 0, 0, 0],
                [0, 0, 0, 0, 1],
                ...,
                [0, 1, 0, 0, 0],
                [1, 0, 0, 0, 0],
                [0, 0, 0, 1, 0]])

In [25]: ##### NOTES
         # 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 fun
         # tionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not
         # itance), 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 BaseEstima
         # tor 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 auto
```

```
# matic hyperparameter tuning. For example, here is a small transformer class that ad
# the combined attributes we discussed earlier:

from sklearn.base import BaseEstimator, TransformerMixin
import numpy as np

rooms_ix, bedrooms_ix, population_ix, household_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[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_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_pe
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
df_extra_attribs = attr_adder.transform(df.values)
```

In [26]:
```
# SCALING - two options:
# (1) min-max scaling (usually 0-1)
# (2) standardization (not 0-1, but less affected by outliers)

# PIPELINE
# We now scale the numeric features using the Pipeline and StandardScaler(), which co

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import Imputer

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

housing_num_tr = num_pipeline.fit_transform(df_num)
```

In [27]:
```
# FULL PIPELINE HANDLING CATEGORICAL AND NUMERIC VARIABLES

from sklearn.pipeline import FeatureUnion
```

```python
# There is nothing in Scikit-Learn to handle Pandas DataFrames, so we need to write a
# this task:
from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names=attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values

# Then we can write our Custom Label Binarizer

class CustomLabelBinarizer(BaseEstimator, TransformerMixin):
    def __init__(self, sparse_output=False):
        self.sparse_output = sparse_output
    def fit(self, X, y=None):
        return self
    def transform(self, X, y=None):
        enc = LabelBinarizer(sparse_output=self.sparse_output)
        return enc.fit_transform(X)

# Define which attributes are numeric and which are categorical

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

num_pipeline = Pipeline([
    ('selector', DataFrameSelector(num_attribs)),
    ('imputer', Imputer(strategy='median')),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scalar', StandardScaler())
])

cat_pipeline = Pipeline([
    ('selector', DataFrameSelector(cat_attribs)),
    ('label_binarizer', CustomLabelBinarizer())
])

full_pipeline = FeatureUnion(transformer_list=[
    ('num_pipeline', num_pipeline),
    ('cat_pipeline', cat_pipeline)
])
```

In [28]:
```python
# Now we can run the full Pipeline
df_prepared = full_pipeline.fit_transform(df)
df_prepared
```

Out[28]:
```
array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
         0.        ,  0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.        ,
         0.        ,  0.        ],
       [ 1.18684903, -1.34218285,  0.18664186, ...,  0.        ,
         0.        ,  1.        ],
       ...,
       [ 1.58648943, -0.72478134, -1.56295222, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.78221312, -0.85106801,  0.18664186, ...,  0.        ,
         0.        ,  0.        ],
       [-1.43579109,  0.99645926,  1.85670895, ...,  0.        ,
         1.        ,  0.        ]])
```

5 Select and Train a Model: Linear Regression & Random Tree

In [29]:
```
# LINEAR REGRESSION
import sklearn
from sklearn.linear_model import LinearRegression

lin_reg = sklearn.linear_model.LinearRegression()
lin_reg.fit(df_prepared, df_labels)
```

Out[29]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

In [30]:
```
# We can now try out our model on some instances of the training set.
some_data = df.iloc[:20]
some_labels = df_labels.iloc[:20]
some_data_prepared = full_pipeline.transform(some_data)

# Somehow the dimensions don't align, so we add a dummy constant (find out with print
import statsmodels.api as sm
some_data_prepared2 = sm.add_constant(some_data_prepared)

print("Predictions:\t", lin_reg.predict(some_data_prepared2))
```

```
Predictions:	 [141205.61156451 183554.62242061  38415.60035182 280725.88619738
 277510.37386787 361575.48916583 107465.26267364 251545.28327809
  58067.56709854   9981.20859169 423021.615525   326121.36861741
 298847.30719306 101312.91924905 308937.7105179   80948.31201093
 123918.46382858 220715.37529613 227857.02125055 224954.00410057]
```

In [31]:
```
# Measuring RMSE for the whole training set
# We see that the model underfits
from sklearn.metrics import mean_squared_error

df_pred = lin_reg.predict(df_prepared)
```

```
        lin_mse = mean_squared_error(df_labels, df_pred)
        lin_rmse = np.sqrt(lin_mse)
        lin_rmse
```

```
Out[31]: 68376.64295459939
```

```
In [32]: # DECISION TREE
         from sklearn.tree import DecisionTreeRegressor

         tree_reg = DecisionTreeRegressor()
         tree_reg.fit(df_prepared, df_labels)
```

```
Out[32]: DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                    max_leaf_nodes=None, min_impurity_decrease=0.0,
                    min_impurity_split=None, min_samples_leaf=1,
                    min_samples_split=2, min_weight_fraction_leaf=0.0,
                    presort=False, random_state=None, splitter='best')
```

```
In [33]: # Compute RMSE for the Tree Model
         # We see that there is no error at all - the model badly overfits the training data!
         df_pred = tree_reg.predict(df_prepared)
         tree_mse = mean_squared_error(df_labels, df_pred)
         tree_rmse = np.sqrt(tree_mse)
         tree_rmse
```

```
Out[33]: 0.0
```

6 Cross Validation

```
In [34]: # We want to randomly split the tree model and evaluate 10 (= k) times
         # The result is an array containing the 10 evaluation scores
         from sklearn.model_selection import cross_val_score

         scores = cross_val_score(tree_reg, df_prepared, df_labels, scoring="neg_mean_squared_
         tree_rmse_scores = np.sqrt(-scores)

         # Let's look at the results:
         def display_scores(scores):
             print("Scores:", scores)
             print("Mean:", scores.mean())
             print("Standard deviation:", scores.std())

         display_scores(tree_rmse_scores)
```

```
Scores: [68388.35970408 66307.44519964 70204.39032561 68298.06791616
 71599.34853504 74480.67948806 70321.47476413 70808.0505509
 76998.85765856 69980.32451557]
Mean: 70738.69986577544
Standard deviation: 2934.092883342242
```

```
In [35]: # Let's compute the same scores for the linear model to be sure
         # We see that the Tree Model is overfitting so badly that it performs worse than the l
         lin_scores = cross_val_score(lin_reg, df_prepared, df_labels, scoring="neg_mean_square
         lin_rmse_scores = np.sqrt(-lin_scores)
         display_scores(lin_rmse_scores)
```

```
Scores: [66877.52325028 66608.120256   70575.91118868 74179.94799352
 67683.32205678 71103.16843468 64782.65896552 67711.29940352
 71080.40484136 67687.6384546 ]
Mean: 68828.99948449331
Standard deviation: 2662.7615706103434
```

7 Random Forest & Cross-Validation

```
In [36]: # RANDOM FOREST
         from sklearn.ensemble import RandomForestRegressor

         forest_reg = RandomForestRegressor()
         forest_reg.fit(df_prepared, df_labels)
```

```
Out[36]: RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                    max_features='auto', max_leaf_nodes=None,
                    min_impurity_decrease=0.0, min_impurity_split=None,
                    min_samples_leaf=1, min_samples_split=2,
                    min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
                    oob_score=False, random_state=None, verbose=0, warm_start=False)
```

```
In [37]: # Compute RMSE scores for the Forest Model
         forest_scores = cross_val_score(forest_reg, df_prepared, df_labels, scoring="neg_mean_
         forest_rmse_scores = np.sqrt(-forest_scores)
         display_scores(forest_rmse_scores)
```

```
Scores: [52193.50219189 50709.74259614 53631.26562596 54258.28640521
 51847.6322984  55403.90767205 50147.73716523 49815.09345125
 55076.99967574 52632.75877272]
Mean: 52571.692585458295
Standard deviation: 1894.1377044806732
```

```
In [38]: # We can save promising models to load them again later
         from sklearn.externals import joblib

         joblib.dump(forest_reg, "forest_reg.pkl")
         # ... and later:
         # forest_reg_loaded = joblib.load("forest_reg.pkl")
```

```
Out[38]: ['forest_reg.pkl']
```

8 Fine-Tune the Model: Grid Search, Randomized Search

In [39]:
```
# GRID SEARCH
# Going through all the possible hyperparameters by hand is very tedious => solution:
from sklearn.model_selection import GridSearchCV

# Define the hyperparameters to test
param_grid = [
    {"n_estimators": [3, 10, 30], "max_features": [2, 4, 6, 8]},
    {"bootstrap": [False], "n_estimators": [3, 10], "max_features": [2, 3, 4]}
]

# Define the model
forest_reg = RandomForestRegressor()

# Compute the Grid Search
grid_search = GridSearchCV(forest_reg, param_grid, cv=5, scoring = "neg_mean_squared_e
grid_search.fit(df_prepared, df_labels)

# Display the best combination of parameters (the one that minimizes RMSE)
grid_search.best_params_
```

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

In [40]:
```
# We can also get the best estimator directly
grid_search.best_estimator_
```

Out[40]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

In [41]:
```
# ... or have a look at the evaluation scores
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
64037.23293749993 {'max_features': 2, 'n_estimators': 3}
56251.19190624636 {'max_features': 2, 'n_estimators': 10}
53170.86836019155 {'max_features': 2, 'n_estimators': 30}
61239.74097575497 {'max_features': 4, 'n_estimators': 3}
53324.4811997054 {'max_features': 4, 'n_estimators': 10}
51852.46942632737 {'max_features': 4, 'n_estimators': 30}
59630.27419182698 {'max_features': 6, 'n_estimators': 3}
53077.26189069585 {'max_features': 6, 'n_estimators': 10}
50853.9342137766 {'max_features': 6, 'n_estimators': 30}
58687.72464032654 {'max_features': 8, 'n_estimators': 3}
```

```
53303.73799842968 {'max_features': 8, 'n_estimators': 10}
50468.37914839918 {'max_features': 8, 'n_estimators': 30}
62658.418239385435 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54687.6772346913 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
61322.80674037472 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
53351.11649186429 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
59261.49207198239 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
53105.89020161202 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

In [42]:
```
# RANDOMIZED SEARCH
# The grid search approach is fine when you are exploring relatively few combinations
# like in the previous example, but when the hyperparameter search space is large, it
# often preferable to use RandomizedSearchCV instead. This class can be used in much
# the same way as the GridSearchCV class, but instead of trying out all possible comb
# nations, it evaluates a given number of random combinations by selecting a random
# value for each hyperparameter at every iteration.
from sklearn.model_selection import RandomizedSearchCV

# Define the hyperparameters to test
# In Randomized Search, we could also give a range for the hyperparameters
# The number of iterations defines how many random combinations of parameters the Ran

n_features = df_prepared.shape[1]

param_dist = {"max_depth": [3, None],
              "max_features": range(1, n_features),
              "n_estimators": range(1, 100),
              "min_samples_split": range(1, 100),
              "min_samples_leaf": range(1, 100),
              "bootstrap": [True, False]}

# Define the model
forest_reg = RandomForestRegressor()

# Compute the Grid Search
grid_search = RandomizedSearchCV(forest_reg, param_dist, cv=5, scoring="neg_mean_squa
grid_search.fit(df_prepared, df_labels)

# Display the best combination of parameters (the one that minimizes RMSE)
grid_search.best_params_
```

Out[42]:
```
{'n_estimators': 75,
 'min_samples_split': 18,
 'min_samples_leaf': 18,
 'max_features': 12,
 'max_depth': None,
 'bootstrap': True}
```

```
In [43]: # Have a look at the results
         cvres = grid_search.cv_results_
         for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
             print(np.sqrt(-mean_score), params)

         # And we can see that our more expansive Grid Search was more successful
```

```
53672.678244024566 {'n_estimators': 94, 'min_samples_split': 72, 'min_samples_leaf': 1, 'max_fe
59743.727232743884 {'n_estimators': 28, 'min_samples_split': 56, 'min_samples_leaf': 98, 'max_
62552.27571390002 {'n_estimators': 63, 'min_samples_split': 69, 'min_samples_leaf': 92, 'max_fe
63039.9877533101 {'n_estimators': 1, 'min_samples_split': 20, 'min_samples_leaf': 97, 'max_fea
59383.47945308758 {'n_estimators': 54, 'min_samples_split': 47, 'min_samples_leaf': 86, 'max_fe
72164.46447545003 {'n_estimators': 68, 'min_samples_split': 8, 'min_samples_leaf': 10, 'max_fea
54659.245044030315 {'n_estimators': 35, 'min_samples_split': 78, 'min_samples_leaf': 14, 'max_
73731.15294322852 {'n_estimators': 96, 'min_samples_split': 24, 'min_samples_leaf': 92, 'max_fe
91964.27128311332 {'n_estimators': 65, 'min_samples_split': 77, 'min_samples_leaf': 90, 'max_fe
53378.345773801455 {'n_estimators': 75, 'min_samples_split': 18, 'min_samples_leaf': 18, 'max_
```

9 Random Forest Feature Importance

```
In [44]: # The Random Forest Regression can deliver insights about feature importance
         feature_importances = grid_search.best_estimator_.feature_importances_
         feature_importances
```

```
Out[44]: array([0.04707705, 0.04160708, 0.03608319, 0.00366621, 0.00380163,
                0.00296816, 0.00319686, 0.41304862, 0.14474247, 0.01291231,
                0.12008443, 0.0095269 , 0.00342597, 0.15567209, 0.        ,
                0.00084991, 0.00133711])
```

```
In [45]: # Lets display these importance scores next to their corresponding attribute names
         extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
         cat_one_hot_attribs = list(encoder.classes_)
         attributes = num_attribs + extra_attribs + cat_one_hot_attribs
         sorted(zip(feature_importances, attributes), reverse=True)
```

```
Out[45]: [(0.41304861580566804, 'median_income'),
          (0.15567208988234324, 'INLAND'),
          (0.14474247421756342, 'income_cat'),
          (0.12008442675040658, 'pop_per_hhold'),
          (0.04707705144577667, 'longitude'),
          (0.04160708445236289, 'latitude'),
          (0.03608318795736406, 'housing_median_age'),
          (0.012912305055734602, 'rooms_per_hhold'),
          (0.009526898210955762, 'bedrooms_per_room'),
          (0.0038016334737196373, 'total_bedrooms'),
          (0.003666214435749274, 'total_rooms'),
          (0.003425972518995523, '<1H OCEAN'),
```

```
(0.003196861270053063, 'households'),
 (0.0029681628533946023, 'population'),
 (0.0013371068500214816, 'NEAR OCEAN'),
 (0.0008499148198910899, 'NEAR BAY'),
 (0.0, 'ISLAND')]
```

10 Evaluate System on the Test Set

In [46]:
```
# As a final step, it's time to evaluate the model on the test set
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
```

Out[46]:
```
50421.1337360698
```

11 APPENDIX. Additional Model: Support Vector Machine Regression (SVR)

In [47]:
```
# Support Vector Machine
from sklearn.svm import SVR

svr_model = SVR()
svr_model.fit(df_prepared, df_labels)
```

Out[47]:
```
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

In [48]:
```
# Compute RMSE scores for the SVM Model
svr_scores = cross_val_score(svr_model, df_prepared, df_labels, scoring="neg_mean_squa
svr_rmse_scores = np.sqrt(-svr_scores)
display_scores(svr_rmse_scores)
```

```
Scores: [115384.13990747 118545.00860081 119939.42694795 119529.25609
 119168.81353278]
Mean: 118513.32901580425
Standard deviation: 1630.2362501167422
```