Manil wagle Follow

Mar 17, 2020 · 15 min read · Listen

Save

Predicting House Prices using Machine Learning

An end to end Solution

ML to Predict House Prices

With house prices near all time high in Toronto, i wanted to see if there are advanced models for predicting house prices. Many of people i know work in Real Estate Industry especially in the sell side. I have asked people how do they determine what should be the sale price of the house price before putting it on the Market. And the common

My goal for this project is to build an end to end solution or application that is capable of predicting the house prices better than individuals. Another motivation for this project is to implement similar solution at my workplace and help Investments and Residential team make data driven decisions.

We will be building models to predict house prices in California using California Census data which consist of metrics such as population, median income, median house price and others for each block group in California which typically consists of population from 600 too 3,000. The ultimate goal of the project is to build a prediction engine capable of predicting district's median housing price. We know that this is supervised learning problem as our data set consists of labelled observations and it does looks like multivariate regression should be our got to option but we will explore multiple ways of building the model and finally pick the one with lowest error rate RMSE (Root Mean Square Error) or MAE (Mean Absolute Error) or any other metrics we choose.

Let's get started by loading the data and some common required libraries

1. **Understanding Data**

```
import sklearn
import numpy as np
import os
import seaborn as sns
import pandas as pd

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Load the data  import pandas as pd
housing= pd.read_csv("housing.csv")

# Explore the data
housing.head()
```

longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
-122.23	37.88	41	880	129	322	126	8.3252	452600	NEAR BAY
-122.22	37.86	21	7099	1106	2401	1138	8.3014	358500	NEAR BAY
-122.24	37.85	52	1467	190	496	177	7.2574	352100	NEAR BAY
-122.25	37.85	52	1274	235	558	219	5.6431	341300	NEAR BAY
-122.25	37.85	52	1627	280	565	259	3.8462	342200	NEAR BAY
-122.25	37.85	52	919	213	413	193	4.0368	269700	NEAR BAY
-122.25	37.84	52	2535	489	1094	514	3.6591	299200	NEAR BAY
-122.25	37.84	52	3104	687	1157	647	3.12	241400	NEAR BAY

Sample Data

As discussed earlier, each row represents one district and there are 10 features of this housing data set. Lets some some additional stats on the data sets.

```
housing.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
```

There are 20,640 observations in the data set with some missing value for total_bedroom feature. Also, we can see that all the features are numeric except ocean_proximity which is categorical variable. Let' see how many levels it has

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

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
```

We see that ocean proximity has 5 levels. Lets explore our data little more using describe feature in python.

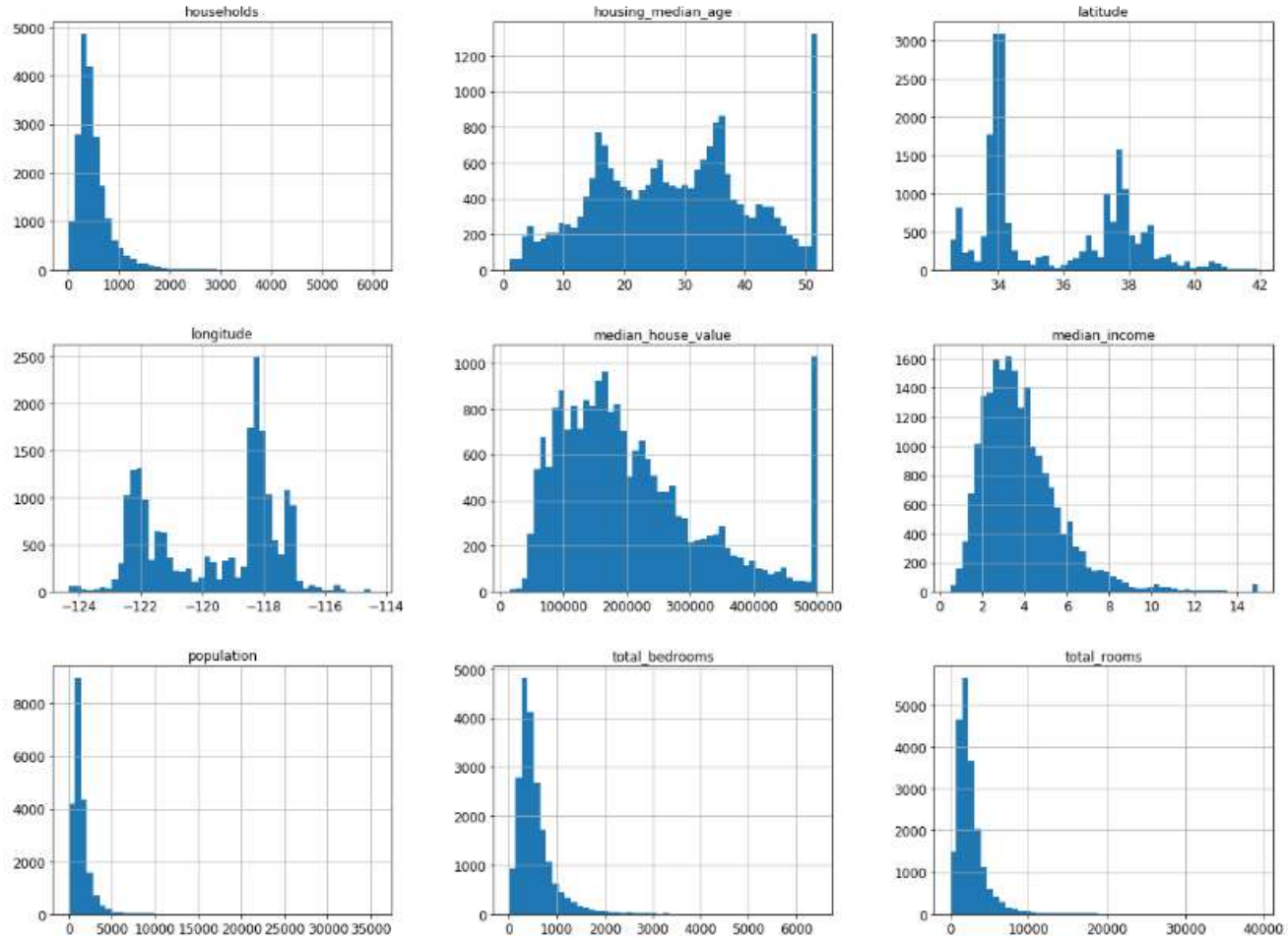
```
# Use describe to explore the numerical variables
housing.describe()
```

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

Exploration

This way we can quickly see basic metrics like average, median, percentile for different features. Lets see their distribution using histograms.

```
# Lets look at the distribution of all the numeric variables
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



Distribution plots

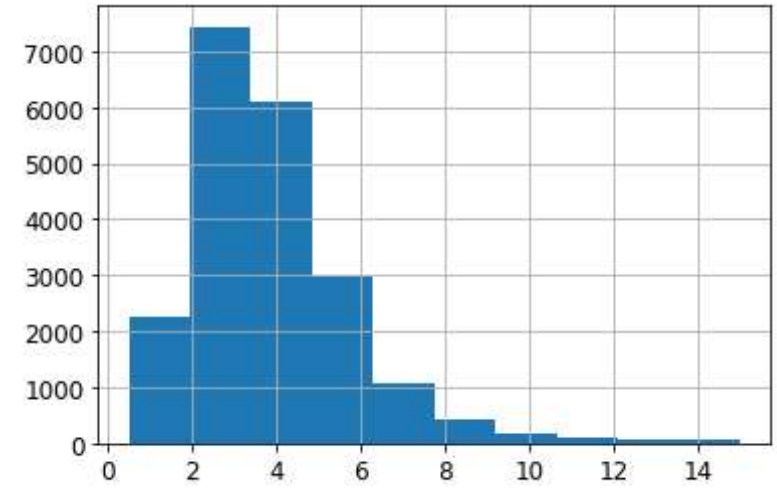
We can quickly see that over 800 districts have median house value of around $100,000. Median income plot seems little strange, as data has been scaled and capped at 15 for higher median income and 0.5 for lower median incomes. Similarly, we can see that median age is capped at 50 and median house value is capped at $500,000. If capped value possess problem, we can either collect proper values for the capped values or remove those district for the data sets. We also see that not all features are in same scale and many features are tail heavy, i.e they extend much farther to the right of the median than to the left.

Before doing any feature engineering, we will divide the datasets into train and test split with 80% of the data for model building and 20% for testing the model.

```
from sklearn.model_selection import train_test_split
```

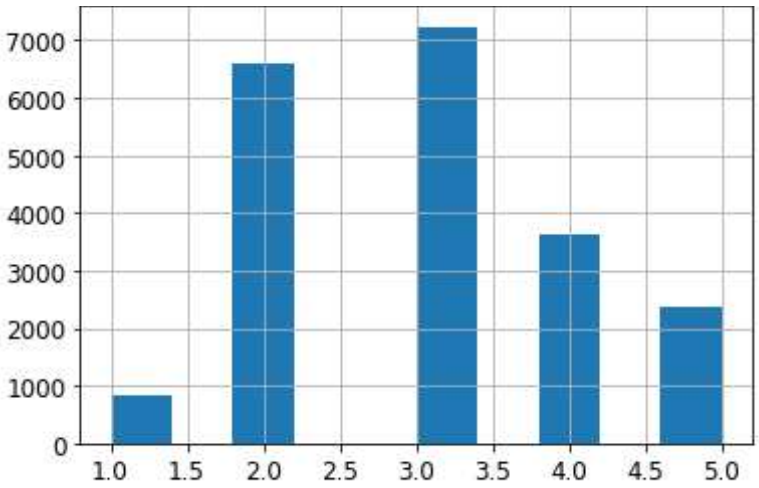
Here, we used random sampling to create train and test datasets. Usually, median income of any neighborhood is great indicator of wealth distribution in that area. So, we want to make sure that test datasets is representative of various categories of income which is actually numeric variable. This means we have to convert it into categorical variables and create different levels of income and use stratified sampling instead of random sampling.

```
housing["median_income"].hist()
```



Median Income

```
# Checking for the right number of bins for the response variable

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



Here, we looked at the distribution of median income and created 5 levels of income category.

```
# Startified sampling based on income_cat to make the datasets more
random and representative

from sklearn.model_selection import StratifiedShuffleSplit

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

Let's check if the income category variable is distributed evenly.

```
## Check if the strata worked for entire datasets
housing["income_cat"].value_counts() / len(housing)
```

```
3    0.350581
2    0.318847
4    0.176308
5    0.114438
1    0.039826
Name: income_cat, dtype: float64
```

```
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

```
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

Test sets

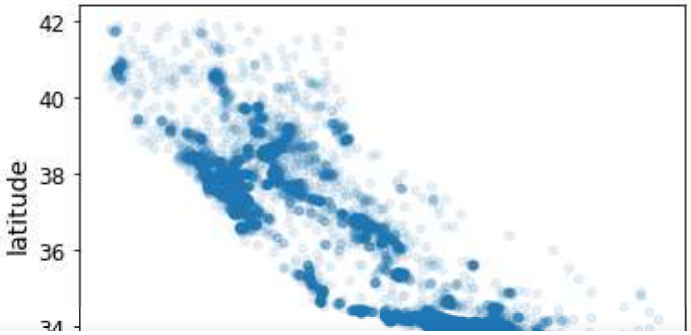
We see the same distribution of income category variable in the test sets as in the entire datasets. Now, lets get the data back to original state by dropping income category variable.

```
## Removing income _cat from the dataset so data goes back to
original state  for set_ in (strat_train_set, strat_test_set):
set_.drop("income_cat", axis=1, inplace=True)
```

2. Visualizing and Exploring Data

Let's do little more exploration now on training sets leaving test sets alone for now.

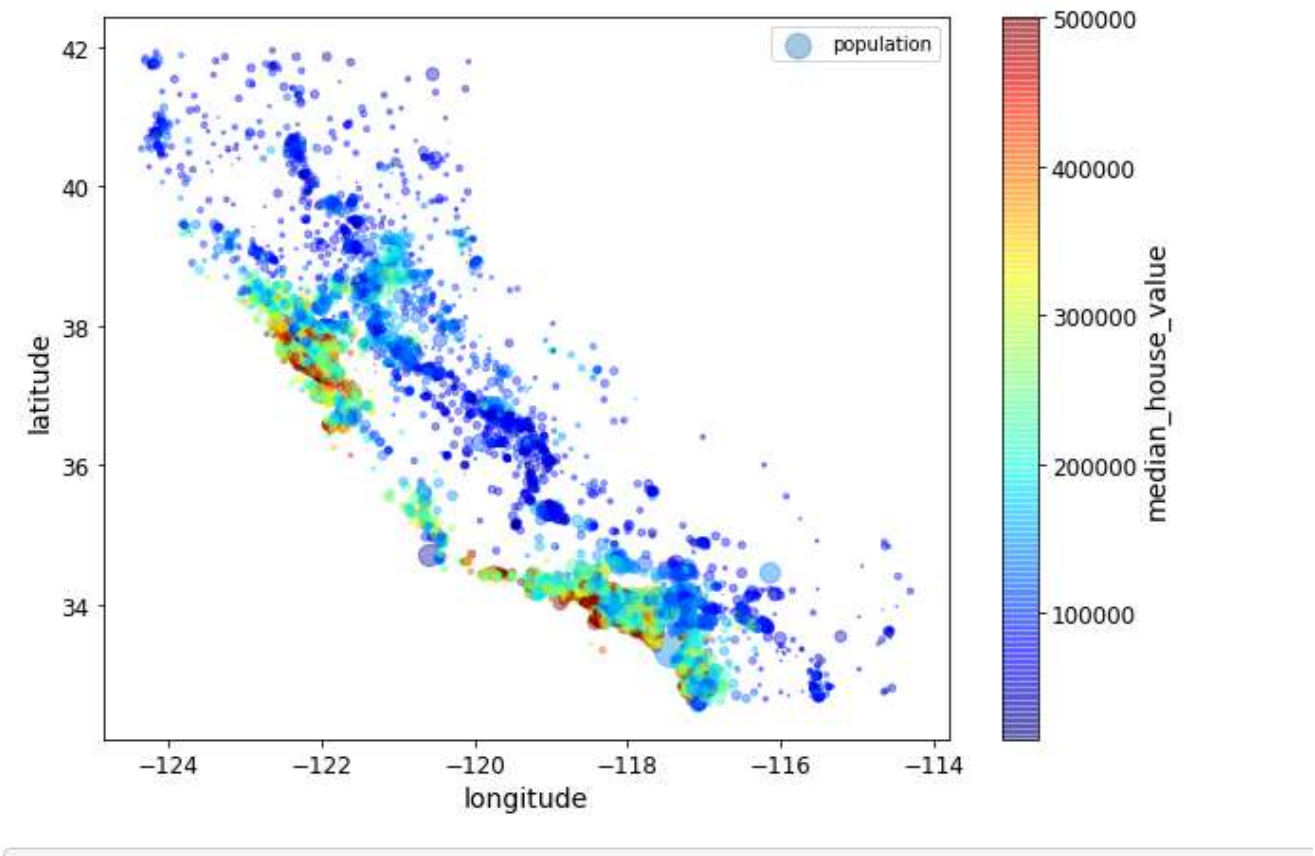
```
## Exploring high density areas

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



High Density Areas

We can now quickly see high density around areas like around Bay, Los Angeles and San Diego. Now lets look at housing prices in these areas.

```
## Lets look at housing prices with circle representing district
population and color representing price

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, sharex=False)
plt.legend()
```



Housing Prices

We can see that house prices are very much correlated with locations and dense areas. What about the correlation of all these features with our target variable; median house

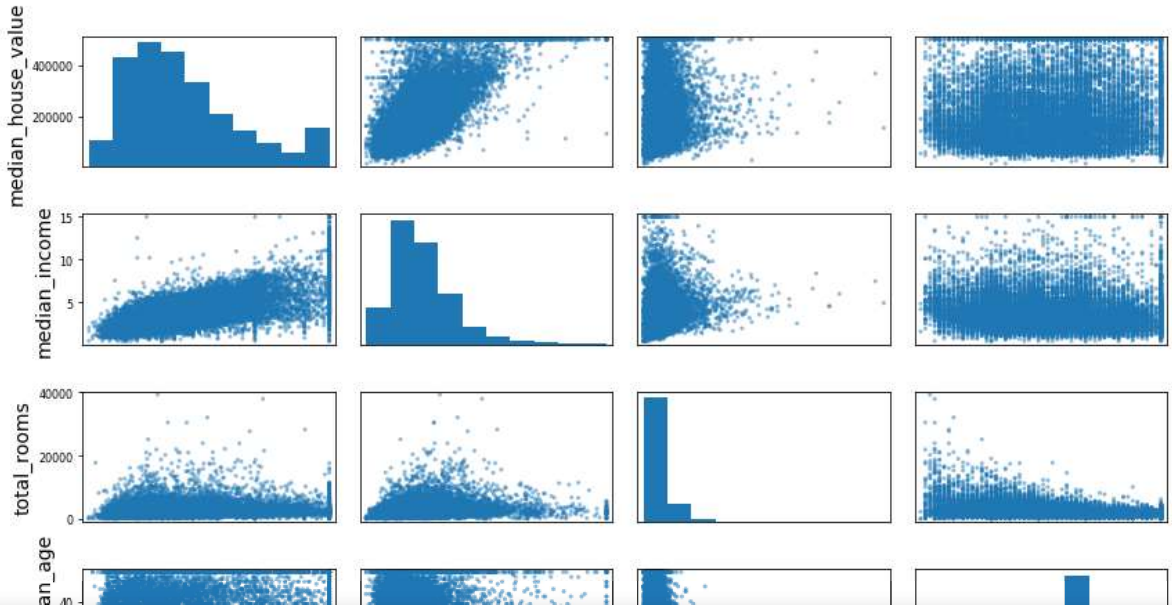
```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
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
```

Correlation Table

Our usual suspects, median income, total rooms and age are top 3 variables in terms of correlation with our target variable. We can even look at the correlation plots.

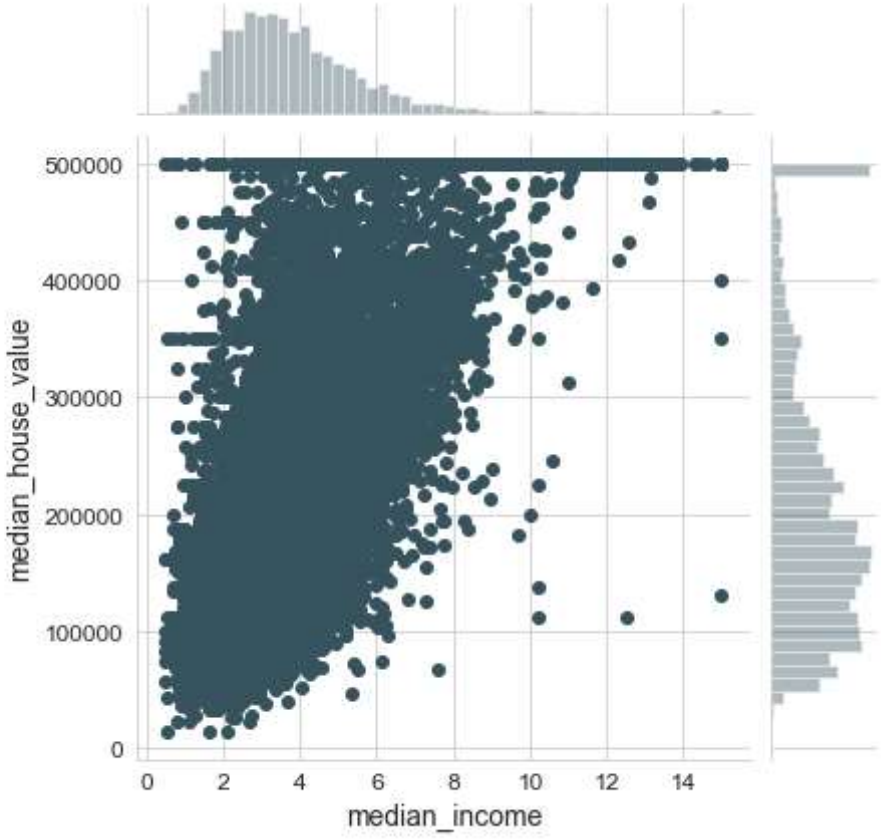
```
from pandas.plotting import scatter_matrix

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



Correlation plots

Since median income is most important variable, lets explore this one little more.

```
sns.jointplot(x="median_income", y="median_house_value",
data=housing)
```



Correlation between Median Income and House Value

We see some unusual lines around $450,000, $350,000 and around $280,000. And as noted earlier, we see strong line around $500,000 which is capped line. It is usually good practice to remove those districts which are creating solid lines.

3. Feature Engineering

W have feature called total_rooms and total_bedrooms which are basically total number of rooms and bedrooms in that district. These features are not useful to us unless we convert them into total numbers of rooms and bedrooms per household. We

```
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["h
ouseholds"]
```

Now, let's look at the correlation table again to make sure that these new features are useful to predict our target variable.

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

```
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
```

Correlation Table with added Features

We see that rooms per household is much more correlated than total rooms.

4. Getting Data Ready for Feeding into Machine Learning Models

This is one of the most essential part of Machine Learning Task as our result will depend on how well we execute this step. Here, we will write as many functions as possible to to make the data ready so that we will be able to reproduce these transformations easily on any datasets. The first step always is to start from the training set and apply same transformation to the test sets. But we will also want to separate features from target variable as in many cases, they need different sort of

```
# Here first we will create a copy and separate the target variable
as we do not want to do the same transformation

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

4 a. Imputing Missing Values

We will start by replacing the missing values of numerical features. To do that, lets first create a dataset without text attributes. We will replace the missing values of all the numerical features using median.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")

# Remove ocean_proximity feature which is text
housing_num = housing.drop("ocean_proximity", axis=1)

# Now, lets impute missing values
imputer.fit(housing_num)
```

Since, only total_bedrooms attribute has missing values, we can just impute missing value for that feature. But just to be sure, we can apply imputer to all numeric features.

```
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
              missing_values=nan, strategy='median', verbose=0)
```

```
imputer.statistics_
## Apply same logic to all the numeric datasets in case fute data
has missing values
housing_num.median().values
```

```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

```
# Now lets use this trained imputer to transform the training sets
X = imputer.transform(housing_num)
```

Now, lets put it back into pandas data frame.

```
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):
longitude           16512 non-null float64
latitude            16512 non-null float64
housing_median_age  16512 non-null float64
total_rooms         16512 non-null float64
total_bedrooms      16512 non-null float64
population          16512 non-null float64
households          16512 non-null float64
median_income       16512 non-null float64
dtypes: float64(8)
memory usage: 1.0 MB
```

4 b. Transforming Categorical Variables

We can see that now there is no missing values. Now, it's time to deal with the text attribute ocean proximity and convert it into numbers so that we can feed it into the ML models. We will use one hot encoding technique for this.

```
from sklearn.preprocessing import OneHotEncoder

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

```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
```

What is happening here is one binary attribute is being created per category. one attribute equals to 1 when category is 'INLAND' and 0 otherwise for all the levels and only one attribute will be equal to 1 (hot), while others will be 0 (cold).

```
cat_encoder.categories_
```

```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

4 c. Custom Transformation

Now, we will write custom functions/transformers to add extra attributes we discussed earlier in the dataset. Writing custom transformers helps in automatic hyper parameter tuning. The code below has one hyper parameter, add_bedrooms_per_room, set to True by default. Which will help us to determine if adding this attribute will help the model or not.

```
#### Creating custom transformation to add extra attributes

from sklearn.base import BaseEstimator, TransformerMixin

# column index
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,
```

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

Now, lets add the attribute back to the dataset.

```
### Adding attributes to the datasets

housing_extra_attribs = pd.DataFrame(
    housing_extra_attribs,
    columns=list(housing.columns)+["rooms_per_household",
"population_per_household"])
housing_extra_attribs.head()
```

housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity	rooms_per_household	population_per_household
38	1568	351	710	339	2.7042	<1H OCEAN	4.62537	2.0944
14	679	108	306	113	6.4214	<1H OCEAN	6.00885	2.70796
31	1952	471	936	462	2.8621	NEAR OCEAN	4.22511	2.02597
25	1847	371	1460	353	1.8839	INLAND	5.23229	4.13598
17	6592	1525	4459	1463	3.0347	<1H OCEAN	4.50581	3.04785

4 d. Transformation Pipelines

ML models don't perform well when input features are in different scales. So, we will standardize the all the numeric features except for target variable.

```
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)
housing_num_tr
```

```
array([[-1.15604281,  0.77194962,  0.74333089, ..., -0.31205452,
        -0.08649871,  0.15531753],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.21768338,
        -0.03353391, -0.83628902],
       [ 1.18684903, -1.34218285,  0.18664186, ..., -0.46531516,
        -0.09240499,  0.4222004 ],
       ...,
       [ 1.58648943, -0.72478134, -1.56295222, ...,  0.3469342 ,
        -0.03055414, -0.52177644],
       [ 0.78221312, -0.85106801,  0.18664186, ...,  0.02499488,
         0.06150916, -0.30340741],
       [-1.43579109,  0.99645926,  1.85670895, ..., -0.22852947,
        -0.09586294,  0.10180567]])
```

Now, lets transform categorical variable

```
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)

housing_prepared.shape
```

```
(16512, 16)
```

Now, our training dataset has 16,512 rows and 16 variables

5. Training a Machine Learning Model

In this section, we will train several ML models with the goal of finding the best model that fits our data, especially the test datasets. We will start with the basic one, i.e Linear Regression. In this section, i will be going through the results more rather than the code itself.

```
from sklearn.linear_model import LinearRegression

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

Now, we have a working linear regression. Let's try doing some prediction on few of the 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: [210644.60459286 317768.80697211 210956.43331178  59218.98886849
 189747.55849879]
```

Let's compare this with the actual values.

```
### Compare against actual values

print("Labels:", list(some_labels))
```

```
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

In first instance, our model is off by around $76,000. Let's measure RMSE of the regression model.

```
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
```




The RMSE tells us that model has typical prediction error of $68,628 which is pretty big. We could try to add more feature or try more complex model to make model more accurate. As part of this project, we will try more complex models.

5 b. Decision Trees

Let's see if we are able to produce better model using decision trees.

```
### Using DecisionTreeRegressor

from sklearn.tree import DecisionTreeRegressor

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

Now we have build a mode, lets evaluate the decision tree model again using RMSE.

```
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
```

RMSE of Decision Tree

Something doesn't look right as the model can't be 100% accurate. Since, we don't want to touch the test dataset until we find our final model, let's use 10 fold cross validation technique to split the training set into further training and validation set.

```
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)
```




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

display_scores(tree_rmse_scores)
```

```
Scores: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
 71115.88230639 75585.14172901 70262.86139133 70273.6325285
 75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

Error Scores on Decision Trees

Now, we can see that linear regression was even better than decision tree which has mean error of $71,407 with standard deviation of +- $2,439 compare to $68,628 RMSE of linear regression. Let's find out what will be the RMSE if we apply 10 fold cross validation in the regression as well.

```
## Using cross validation on linear regression

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: [66782.73843989 66960.118071   70347.95244419 74739.57052552
 68031.13388938 71193.84183426 64969.63056405 68281.61137997
 71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798349
```

Error Scores on Linear Regression

So, our linear regression is indeed better than decision tree for the problem we have as linear regression still has mean error of only $69,000 compare to $71,000 for decision trees.

Random forest works by building multiple trees on random subset of features and averaging out their predictions.

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=100,
random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

Let's see RMSE of random forest on training sets.

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

```
18603.515021376355
```

RMSE of Random Forest

Wow, this is great; it means model prediction error is just $18,603 on training sets. Will we get different result if we use cross validation on random forest?

```
### Using cross validation in random forest

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: [49519.80364233 47461.9115823  50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574  47585.73832311
 53490.10699751 50021.5852922 ]
```

Not bad, so far one of the best model with the error rate of $50,182 even though we see that error rate is pretty high in validation datasets compare to training sets suggesting there might be over fitting issue.Let's try one last model before starting to fine tune our final model.

5 d. Support Vector Machine

```
### Lets see how SVM performs

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
```

```
111094.6308539982
```

RMSE of Support Vector Machine

RMSE of $111,094 rules Support Vector Machine from the final consideration.

6. Fine Tuning the Model

We will fine tune our random forest model using grid search technique. Where we will need to tell which hyper parameters we want to experiment and what values to try out, and grid search technique will evaluate all the possible combination of hyper parameters values, using cross validation.

```
### Using grid search to fine tune the model. Random forest

from sklearn.model_selection import GridSearchCV

param_grid = [
    # try 12 (3×4) combinations of hyperparameters
```

```
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)

grid_search.best_params_
```

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

Grid search Result

Let's look at the result of hyper parameter combination tested during the grid search.

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

```
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Grid Search Results

```
# Randomized hyper parameter search

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)

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

```
49150.657232934034 {'max_features': 7, 'n_estimators': 180}
51389.85295710133 {'max_features': 5, 'n_estimators': 15}
50796.12045980556 {'max_features': 3, 'n_estimators': 72}
50835.09932039744 {'max_features': 5, 'n_estimators': 21}
49280.90117886215 {'max_features': 7, 'n_estimators': 122}
50774.86679035961 {'max_features': 3, 'n_estimators': 75}
50682.75001237282 {'max_features': 3, 'n_estimators': 88}
49608.94061293652 {'max_features': 5, 'n_estimators': 100}
50473.57642831875 {'max_features': 3, 'n_estimators': 150}
64429.763804893395 {'max_features': 5, 'n_estimators': 2}
```

Randomized Search Results

For the purpose of this project, we will stay with grid search. Now, its time to analyze the best model and its error. Lets start by looking the importance of features in the random forest model.

```
# Feature Importance

feature_importances =
grid_search.best_estimator_.feature_importances_
```

```
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)
```

```
[(0.36615898061813423, 'median_income'),
 (0.16478099356159054, 'INLAND'),
 (0.10879295677551575, 'pop_per_hhold'),
 (0.07334423551601243, 'longitude'),
 (0.06290907048262032, 'latitude'),
 (0.056419179181954014, 'rooms_per_hhold'),
 (0.053351077347675815, 'bedrooms_per_room'),
 (0.04114379847872964, 'housing_median_age'),
 (0.014874280890402769, 'population'),
 (0.014672685420543239, 'total_rooms'),
 (0.014257599323407808, 'households'),
 (0.014106483453584104, 'total_bedrooms'),
 (0.010311488326303788, '<1H OCEAN'),
 (0.002856474637320158, 'NEAR OCEAN'),
 (0.0019604155994780706, 'NEAR BAY'),
 (6.0280386727366e-05, 'ISLAND')]
```

Feature Importance

This step allows us the opportunity to understand which feature are most important and which are of low importance, i.e candidate that can be dropped. As we seen earlier, median income is top feature for the model.

7. Evaluate the Model on the Test Set

Finally, its time to evaluate the random forest model on the test set and deploy it into production.

```
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
```

RMSE of test set

The RMSE of 47,730 is really g[o]odel and we will be deploying this random forest model into the production. Computing the prediction interval of model is always a good ideas as it makes us aware how much the error can fluctuate.


```
# Computing 95% confidence interval
from scipy import stats

confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors)))
```

```
array([45685.10470776, 49691.25001878])
```

95% Confidence Interval

This tells us that prediction error can fluctuate anywhere between $45,685 to $49,691. Around $4,000 gap in confidence interval is something we can live with. So, this is our final model.

8. Conclusions and Next Steps

I believe this model could be optimized and tuned more to add accuracy either by adding new features or engineering new features. This model can be used to predict the house prices in any geographic location by just slightly fine tuning the features and parameters.

What would be more interesting in my view is; if we could add second layer to the model output or may be second step where results from this model are fed into second model which would then forecast district house prices 6 months, 18 months and so on into the future. This would allow the opportunity not only to predict the house prices but also to see what the future holds for the house prices. And this is exactly the type of insights Real Estate Investment teams need to make right investments.