# Predicting House Prices using Machine Learning



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 answer i get is we look at the comparable properties in the areas. Can we add some science behind this process?

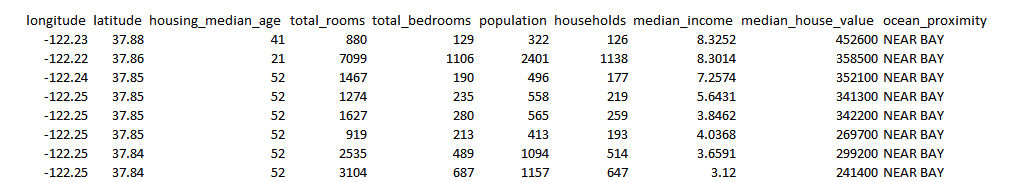
My goal for this project is to build an end to end solution or application that is capable of predicting the house prices better than individuals. Another motivation for this project is to implement similar solution at my workplace and help Investments and Residential team make data driven decisions.

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**

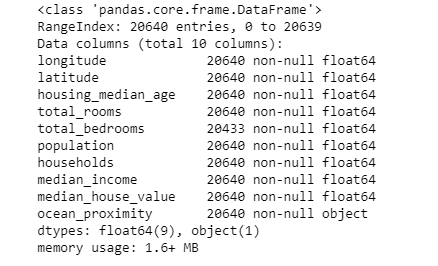
import sklearn  
import numpy as np  
import os  
import seaborn as sns  
import pandas as pd# To plot pretty figures  
%matplotlib inline  
import matplotlib as mpl  
import matplotlib.pyplot as plt  
mpl.rc('axes', labelsize=14)  
mpl.rc('xtick', labelsize=12)  
mpl.rc('ytick', labelsize=12)# Load the data import pandas as pd   
housing= pd.read\_csv("housing.csv")# Explore the data  
housing.head()



Sample Data

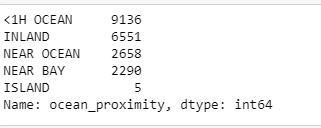
As discussed earlier, each row represents one district and there are 10 features of this housing data set. Lets some some additional stats on the data sets.

housing.info()



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

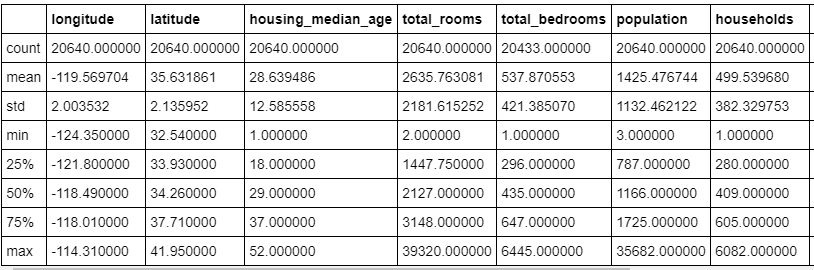
housing["ocean\_proximity"].value\_counts()



Levels of ocean\_proximity

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

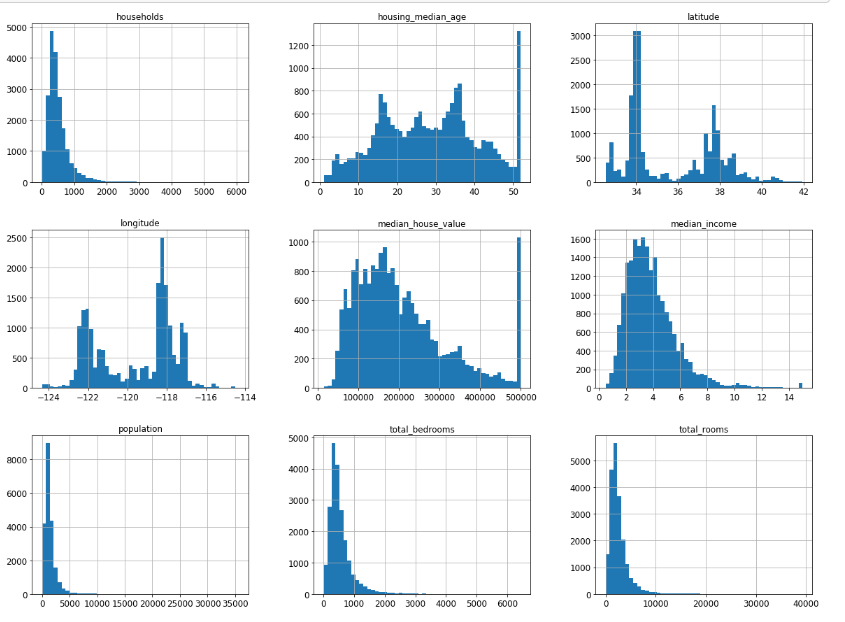
# Use describe to explore the numerical variables   
housing.describe()



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

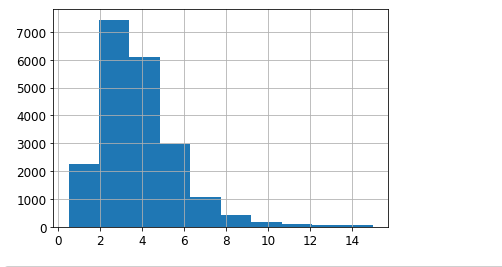
We can quickly see that over 800 districts have median house value of around $100,000. Median income plot seems little strange, as data has been scaled and capped at 15 for higher median income and 0.5 for lower median incomes. Similarly, we can see that median age is capped at 50 and median house value 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  
  
train\_set, test\_set= train\_test\_split(housing, test\_size=0.2, random\_state=42)

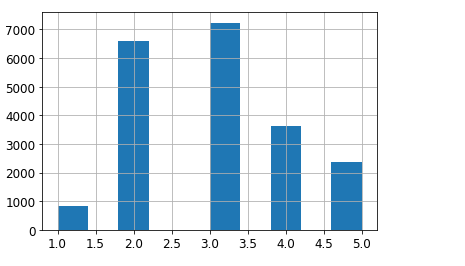
Here, we used random sampling to create train and test datasets. Usually, median income of any neighborhood is great indicator of wealth distribution in that area. So, we want to make sure that test datasets is representative of various categories of income which is actually numeric variable. This means 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()

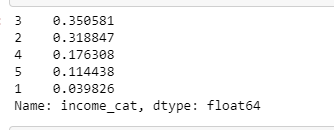


Here, we looked at the distribution of median income and created 5 levels of income category.

# 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.

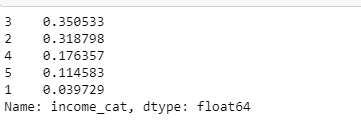
## Check if the strata worked for entire datasets  
housing["income\_cat"].value\_counts() / len(housing)



Entire datasets

Now, lets see if the same proportion has been applied in the test sets.

strat\_test\_set["income\_cat"].value\_counts() / len(strat\_test\_set)



Test sets

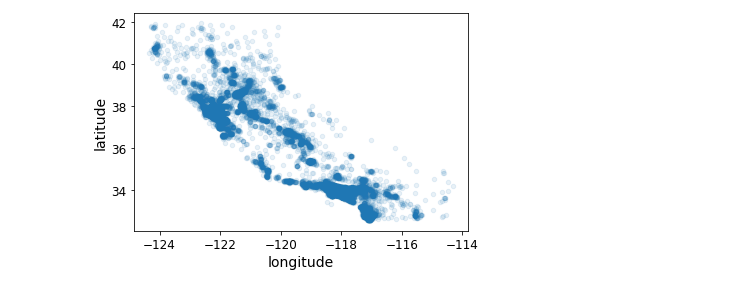
We see the same distribution of income category variable in the test sets as in the entire datasets. Now, lets get the data back to original state by dropping income category variable.

## 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.

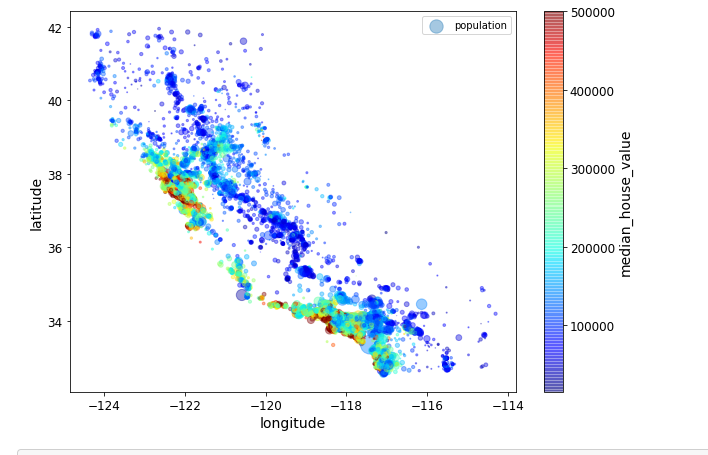
## Exploring high density areas  
  
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)



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.

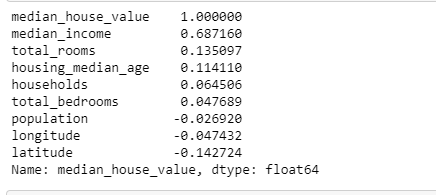
## Lets look at housing prices with circle representing district population and color representing price  
  
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,  
 s=housing["population"]/100, label="population", figsize=(10,7),  
 c="median\_house\_value", cmap=plt.get\_cmap("jet"), 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 value.

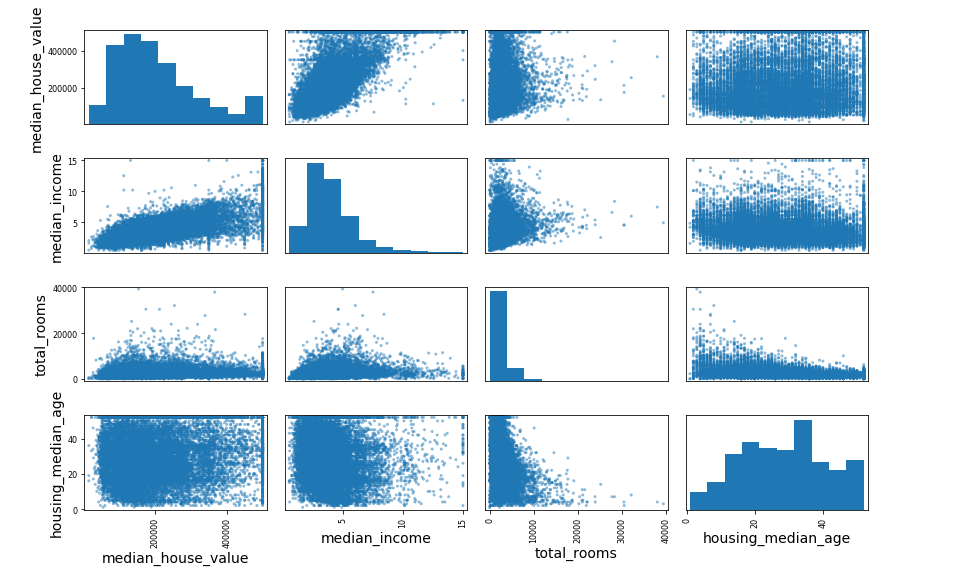
## How other variables relate with our target variable  
  
corr\_matrix = housing.corr()  
corr\_matrix["median\_house\_value"].sort\_values(ascending=False)



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.

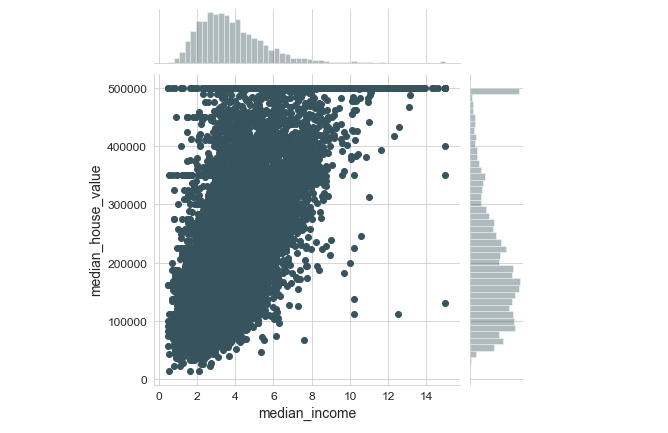
from pandas.plotting import scatter\_matrixattributes = ["median\_house\_value", "median\_income", "total\_rooms",  
 "housing\_median\_age"]  
scatter\_matrix(housing[attributes], figsize=(12, 8))  
save\_fig("scatter\_matrix\_plot")



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.

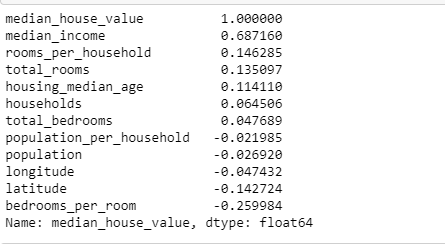
**3. Feature Engineering**

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 will also create a new variable called population per household using population variable.

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"]

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)



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 transformation.

# 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.



imputer.statistics\_  
## Apply same logic to all the numeric datasets in case fute data has missing values  
housing\_num.median().values

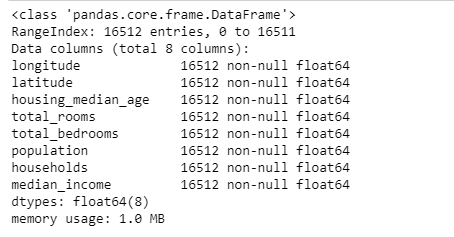


Now, we will use this trained imputer to transform the training set by replacing values by the learned medians,

# 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.

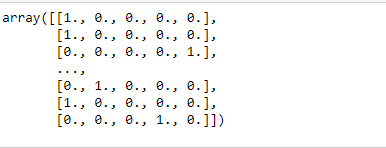
housing\_tr = pd.DataFrame(X, columns=housing\_num.columns)  
housing\_tr.info()



**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



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\_



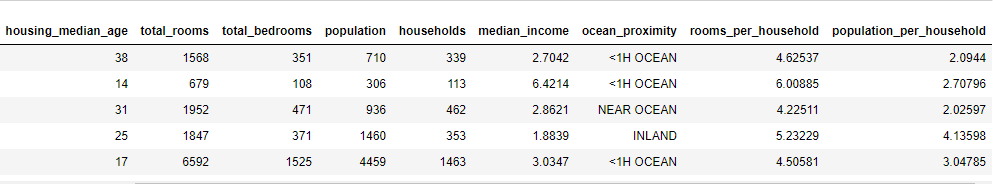
**4 c. Custom Transformation**

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 attributesfrom sklearn.base import BaseEstimator, TransformerMixin# column index  
rooms\_ix, bedrooms\_ix, population\_ix, households\_ix = 3, 4, 5, 6class 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)

Now, lets add the attribute back to the dataset.

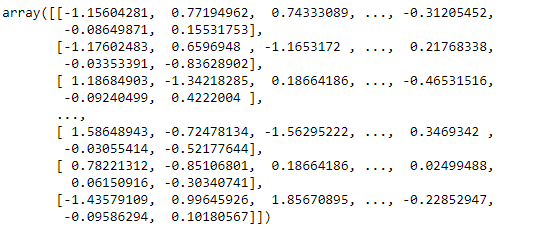
### Adding attributes to the datasetshousing\_extra\_attribs = pd.DataFrame(  
 housing\_extra\_attribs,  
 columns=list(housing.columns)+["rooms\_per\_household", "population\_per\_household"])  
housing\_extra\_attribs.head()



**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 StandardScalernum\_pipeline = Pipeline([  
 ('imputer', SimpleImputer(strategy="median")),  
 ('attribs\_adder', CombinedAttributesAdder()),  
 ('std\_scaler', StandardScaler()),  
 ])housing\_num\_tr = num\_pipeline.fit\_transform(housing\_num)  
housing\_num\_tr



Now, lets transform categorical variable

from sklearn.compose import ColumnTransformernum\_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



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.

**5 a. Linear Regression**

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))



Let’s compare this with the actual values.

### Compare against actual values  
  
print("Labels:", list(some\_labels))



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



RMSE of the Regression Model

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



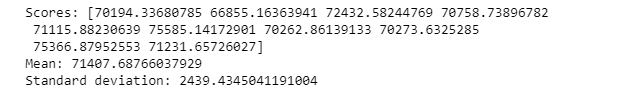
RMSE of Decision Tree

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 result decision tree model after cross validation.

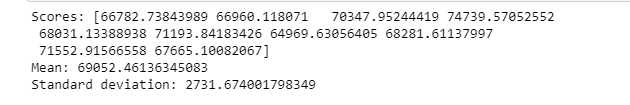
### Let's look at the scores  
def display\_scores(scores):  
 print("Scores:", scores)  
 print("Mean:", scores.mean())  
 print("Standard deviation:", scores.std())  
  
display\_scores(tree\_rmse\_scores)



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)



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.

**5 c. Random Forest**

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.

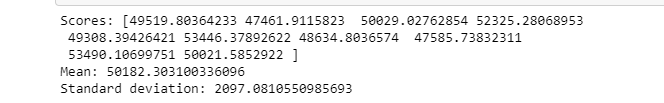
housing\_predictions = forest\_reg.predict(housing\_prepared) forest\_mse = mean\_squared\_error(housing\_labels, housing\_predictions) forest\_rmse = np.sqrt(forest\_mse)   
forest\_rmse



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)



Error Scores of Random Forest

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



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.

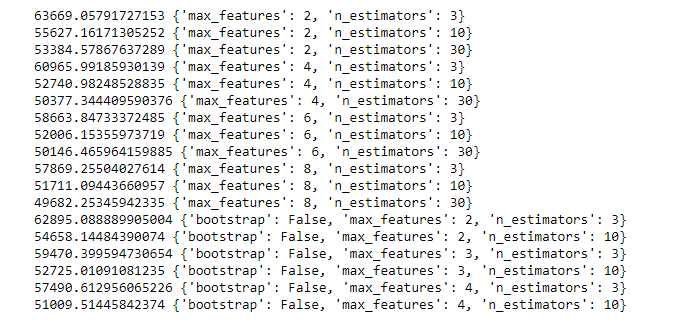
### Using grid search to fine tune the model. Random forestfrom sklearn.model\_selection import GridSearchCVparam\_grid = [  
 # try 12 (3×4) combinations of hyperparameters  
 {'n\_estimators': [3, 10, 30], 'max\_features': [2, 4, 6, 8]},  
 # then try 6 (2×3) combinations with bootstrap set as False  
 {'bootstrap': [False], 'n\_estimators': [3, 10], 'max\_features': [2, 3, 4]},  
 ]forest\_reg = RandomForestRegressor(random\_state=42)  
# train across 5 folds, that's a total of (12+6)\*5=90 rounds of training   
grid\_search = GridSearchCV(forest\_reg, param\_grid, cv=5,  
 scoring='neg\_mean\_squared\_error',  
 return\_train\_score=True)  
grid\_search.fit(housing\_prepared, housing\_labels)grid\_search.best\_params\_



Grid search Result

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

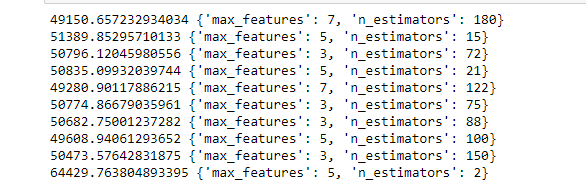
cvres = grid\_search.cv\_results\_  
for mean\_score, params in zip(cvres["mean\_test\_score"], cvres["params"]):  
 print(np.sqrt(-mean\_score), params)



Grid Search Results

We see that combination of 8 feature and 30 estimators gives the lowest RMSE of $49,682. When the problem and data in hand is massive, it is usually recommended to use randomized search rather than grid search like below.

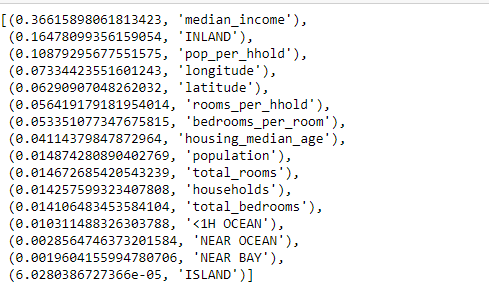
# Randomized hyper parameter searchfrom sklearn.model\_selection import RandomizedSearchCV  
from scipy.stats import randintparam\_distribs = {  
 'n\_estimators': randint(low=1, high=200),  
 'max\_features': randint(low=1, high=8),  
 }forest\_reg = RandomForestRegressor(random\_state=42)  
rnd\_search = 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)



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 Importancefeature\_importances = grid\_search.best\_estimator\_.feature\_importances\_  
feature\_importancesextra\_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)



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 good , so this is our final model 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 statsconfidence = 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)))



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.