IMPORT MODULES AND PREPARE DATA:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
Xmatplotlib inline
import seaborn as sns
from sklearn.model_selection import train_test_split
from pandas.plotting import scatter_matrix
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import TransformerMixin
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
```

First, here we have imported the modules.

```
import os
import tarfile
import urllib
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"
def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tz__path = os.path.join(housing_path, "housing_tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
          housing_tgz = tarfile.open(tgz_path)
housing_tgz.extractall(path=housing_path)
housing_tgz.close()
def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
fetch_housing_data()
housing = load_housing_data()
 rooms_ix, bedrooms_ix, population_ix, household_ix = [
          list(housing.columns).index(col)
for col in ("total_rooms", "total_bedrooms", "population", "households")]
 class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
        inst CombinedAttributesAdder(BaseEstimator, TransformerWixin):

def __init__(self, add_bedrooms_per_room = True):
    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_noom = 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]
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
housing = train_set.drop("median_house_value", axis=1)
housing_labels = train_set["median_house_value"].copy()
housing_num = housing.drop("ocean_proximity", axis=1)
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
 num_pipeline = Pipeline([
  ('imputer', SimpleImputer(strategy="median")),
('attribs_adder', CombinedAttributesAdder()),
('std_scaler', StandardScaler())])
full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", OneHotEncoder(), cat_attribs)])
 housing_prepared = full_pipeline.fit_transform(housing)
```

We have loaded housing data.

- We have **created** our transformer to calculate new features. To create **transform**, we have inherited the **BaseEstimator** class and **TransformerMixin** class. We have **defined** a Boolean variable that indicates whether we want to create the feature of the proportion of **bedrooms** with respect to number of rooms.
- Split the data into training and testing using train_test_split from scikit-learn.
- Create the predictor and labels for the training.
- We have **created** our pipeline which we first **fill** the null value with the **median** of each column then **add** new features and finally **apply** one hot encode for the categorical value.
- We **apply** the **pipeline** to the data.

SELECT AND TRAIN A MODEL

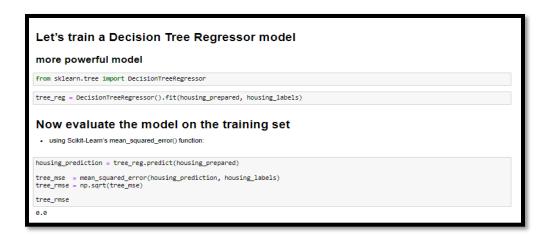
FIRST TRAIN A LINEARREGRESSION MODEL:

```
1- Select and Train a Model
Let's first train a LinearRegression model
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression().fit(housing_prepared, housing_labels)
First try it out on a few instances from the training set:
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)
print('Prediction:', lin_reg.predict(some_data_prepared))
print('label:', list(some_labels))
Prediction: [181746.54359616 290558.74973505 244957.50017771 146498.51061398
163230.42393939]
label: [103000.0, 382100.0, 172600.0, 93400.0, 96500.0]
measure this regression model's RMSE on the whole training set
· sing Scikit-Learn's mean_squared_error() function:
from sklearn.metrics import mean_squared_error
housing_prediction = lin_reg.predict(housing_prepared)
lin mse = mean_squared_error(housing_prediction, housing_labels)
lin_rmse = np.sqrt(lin_mse)
67593.20745775253
```

- First, we train the data on a simple model. So, we choose linear regression.
- We have **trained** it on only 5 **sample data** and the output as seen is **near** to the true value.
- We have **computed** the error using **root mean squared error** and we have **found** the error is **67593** which is very large.

- The error is very large even if we use **training data** to **evaluate** the model.
- This is **due** to **selecting** a very simple model for our problem.
- We can solve this issue by:
 - Choosing a more complex model.
 - Adding constraints to our data (regularization).
 - Trying to add some data.

TRAIN A DECISION TREE REGRESSOR MODEL



- We have chosen a more complex model, "DecisionTreeRegressor".
- When we **evaluate** the model on the training data, we have **0 error**, but this doesn't mean that **the** model is perfect, but rather **it** is **overfit**.
- We can solve this problem by **choosing** another model or **applying** cross-validation or **fine-tuning**.

EVALUATION USING CROSS-VALIDATION:

```
Evaluation Using Cross-Validation

1-split the training set into 10 distinct subsets then train and evaluate the Decision Tree model

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)

2- display the resultant scores and calculate its Mean and Standard deviation

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

display_scores(tree_rmse_scores)

Scores: [65078.25891532 70696.02750088 69317.03471236 71590.76310531
73540.25698882 67900.28180135 67155.89636265 68593.47233815
67226.35976615 70788.91303135]

Mean: 69196.72653223416

Standard deviation: 2371.663291454293
```

- We have applied K-fold cross-validation techniques using 10 validation data.
- It **splits** the training into **10 sets** and **trains** the data on 9 and **evaluates** on the 10th.
- The value that is returned is the **negative mean squared error** (this is negative because it **uses** utility function; **more better** is good instead of **less values**; **more better**).

```
3-repaet the same steps to compute the same scores for the Linear Regression model

notice the difference between the results of the two models

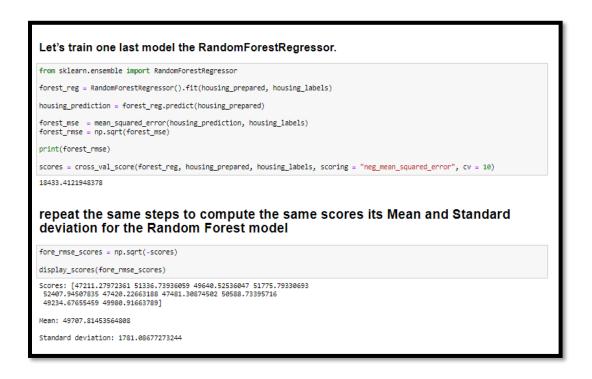
| scores= cross_val_score(lin_reg, housing_prepared, housing_labels, scoring = "neg_mean_squared_error", cv=10)
| lin_rmse_scores = np.sqrt(-scores)
| display_scores(lin_rmse_scores)

| scores: [65000.67382615 70960.56056304 67122.63935124 66089.63153865
| 68402.54686442 65266.34735288 65218.78174481 68525.46981754
| 72739.87555996 68957.34111906]

| Mean: 67828.38677377408
| Standard deviation: 2468.0913950652284
```

- We have applied the same previous step to the linear regression model.
- According to the **root mean square error** and **standard deviation**, the linear regression is **better** than **decision tree**, but it **still performs poorly**.

TRAIN ONE LAST MODEL THE RANDOMFORESTREGRESSOR:



- The **final model** we will **try** is **RandomForestAlgorithm**.
- It will apply the **decision tree algorithm** to many **decision tree models** with different **hyperparameters** then take the **average** for this model. Here we have **chosen 10 models**.
- We have **applied** the **same previous step**.

From the root mean square error and standard deviation, it is much better than the linear regression model and the decision tree algorithms.

SAVING MODELS:

```
Save every model you experiment with

using the joblib library

import joblib

joblib.dump(lin_reg, "linear_regression.pkl")
joblib.dump(tree_reg, "decision_tree.pkl")
joblib.dump(forest_reg, "random_forest.pkl")

['random_forest.pkl']
```

We have saved our models and their hyperparameters to be able to load and use them later.

FINE-TUNE YOUR MODEL

- To **overcome** the **overfitting**, we have used **fine-tuning** technique using **GridSearchCV**.
- In this algorithm, we **define hyperparameters** and **train** all the combinations and **choose** the **best one**.
- It **takes much time** because it will try all possible combinations, which is **90** on the random forest algorithm.
- This algorithm is **effective** here, but if we have many possible combinations of **hyperparameters** or we **do not actually know** what **hyperparameters** we can choose, **you** can use **randomized search**.

ANALYZE THE BEST MODELS:

- Now we have **analyzed** the **best model** to **gain** some **insights**.
- We have **printed** the **relative importance** of each **feature** to **determine** which **features** are most **influential** in increasing accuracy.
- We can **remove** the **features** that have **low relative importance** and **combine** new ones to **obtain** more **impactful** data.
- We can **note** that **from** the **categorical feature**, only **INLAND** is **highly influential**, while the **others** are **less influential**.

EVALUATE YOUR SYSTEM ON THE TEST SET

```
Now is the time to evaluate the final model on the test set.
Evaluate Your System on the Test Set
1-get the predictors and the labels from your test set
final_model = grid_search.best_estimator_
X_test = test_set.drop("median_house_value", axis=1)
y_test = test_set["median_house_value"].copy()
2-run your full pipeline to transform the data
X_test_prepared = full_pipeline.transform(X_test)
3-evaluate the final model on the test set
final_prediction = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_prediction)
final_rmse = np.sqrt(final_mse)
final_rmse
49924.83143250667
compute a 95% confidence interval for the generalization error
using scipy.stats.t.interval().
from scipy import stats
squared_errors = (final_prediction - y_test) ** 2
np.sqrt(stats.t.interval(.95, len(squared errors) - 1,
        loc=squared_errors.mean(),
       scale=stats.sem(squared_errors)))
array([47747.38671616, 52011.19734159])
```

- Finally, we evaluate the best model we have obtained on the test data, generalizing it to data it never sees.
- We **obtain** the **predictors** and the **labels** of the test data.
- We pass it through the pipeline, but only apply transform to it to prevent it from learning from this data and use the values we obtained from the training data.
- We have **calculated** the **error**, which is **considered good** compared to the earlier model, but **still cannot** be **used** because of the **large error**.
- It is better to calculate a range in which the data can change rather than calculating a simple error value.
- We have **calculated** the **95% confidence interval**, obtaining the **bounds** within which **95% of errors** can **lie**.