Report on California housing Dataset

Report by Team Caserta.

1. MAVILLAPALLI VENKATA TARUN KUMAR (P37000126)
2. GATTEM PRIYA MADHURI (P37000162)
3. KARRI RAHUL REDDY (P37000157)

Project Scope:

Our first task is to use the california census data to build a model of the housing prices in the state. This data includes features such as:

1.Population
2.Median Income
3.Median housing price for each block group in California

A block group is the smallest geographical unit for which census data is published. A Block group has a population between 600 to 3,000. We will call them "districts" for short.

Our model should be able to predict the median housing price for any district, given the other features.

Also to measure the performance of the model we have used the Root Mean Squared Error (RMSE).

Now lets just dive into the problem.

Step1: Importing libraries:

In [1]:

```
#Importing all the necessary libraries.
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from zlib import crc32
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit
from pandas.plotting import scatter_matrix
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.base import TransformerMixin, BaseEstimator
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
import joblib
from sklearn.model_selection import GridSearchCV
from scipy import stats
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import reciprocal, expon
from sklearn.base import BaseEstimator, TransformerMixin
```

Step 2: Load the data

In [2]:

```
#Step2.1: Read the "housing.csv" file from the folder into the program
housing = pd.read_csv(r'/Users/krahu/Downloads/housing.csv')

#Step2.2: Print first few rows of this data
housing.head()
```

Out[2]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Each row represents one district and has the following (10) attributes:

```
1.longitude
2.latitude
3.housing_median_age
4.total_rooms
5.total_bedrooms
6.population
7.households
8.median_income
9.median_house_value
10.ocean_proximity
```

In [3]:

```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

```
The info() method is useful to take a quick look at the data. It answers the following questions:

1.How many rows exist?
-> we found data has 20,640 rows

2.How many NaNs per column?
-> As we can see from above all columns have 20640 values apart from total_bedrooms. So we suspect that 207 values are missing.

3.What are the data types (per column)?
-> dtypes: float64(9), object(1)

However, all attributes except ocean_proximity are numerical.

Since we noticed repeated ocean_proximity values for the top 5 rows, we suspect that it is a categorical column, let's check it out:
```

In [4]:

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

Out[4]:

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

In [5]:

```
# note: .describe() ignores null values
housing.describe()
```

Out[5]:

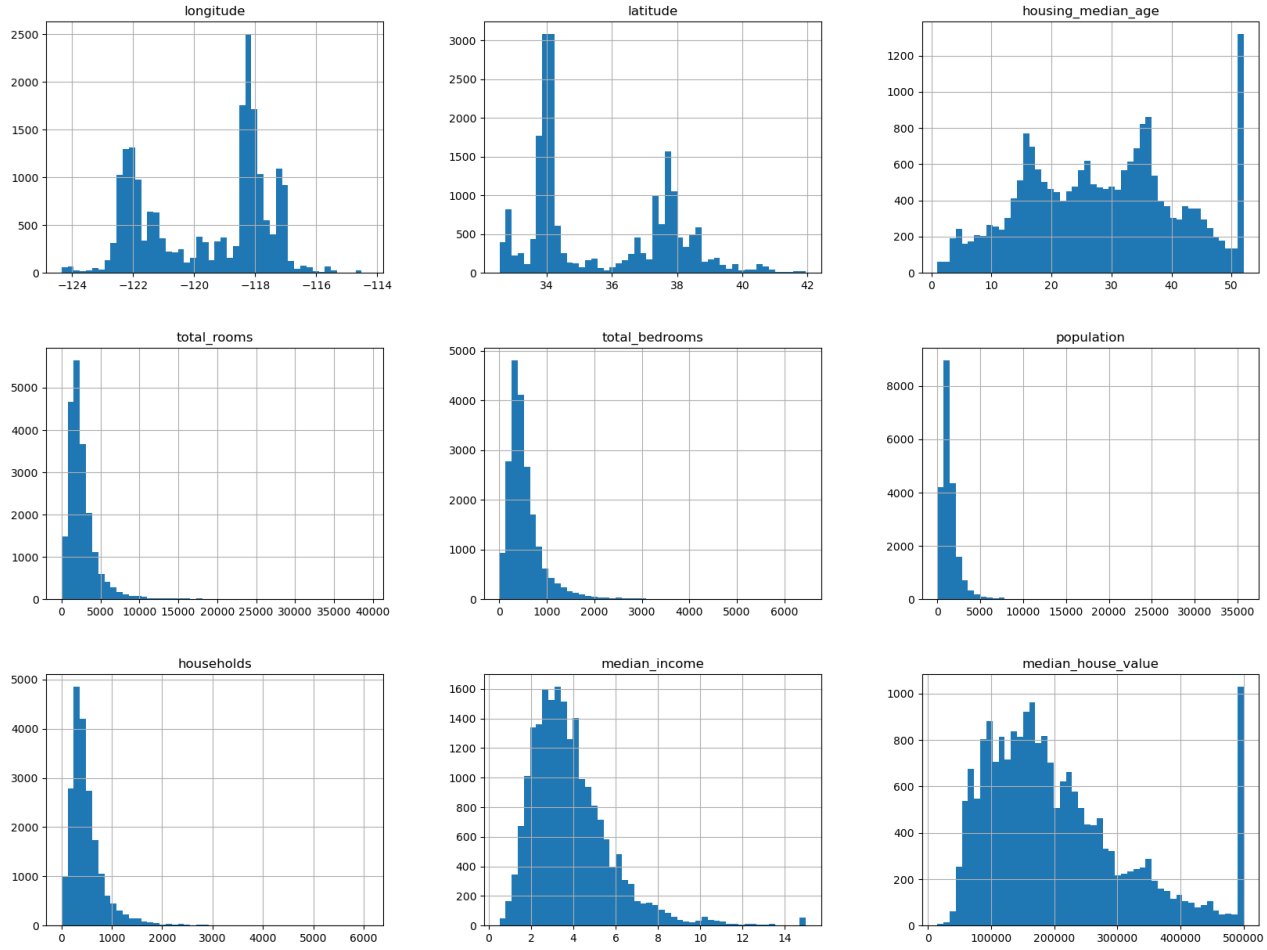
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

The standard deviation row displays the values of dispersion. The 25%, 50%, and 75% rows display each column's percentiles.

step 2.3: Lets plot histograms for the housing data.

In [6]:

```
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



Few things to notice from above :

The attributes have very different scales. The attributes are tail-heavy & they often extend to the right than to the left. As a result, It will be difficult for many machine learning algorithms to find patterns within the data.

step 3: split the data set to test and train

In [7]:

```
def split_train_test(data, test_ratio=0.2):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
# calling the function
train_set, test_set = split_train_test(data=housing)
len(train_set), len(test_set)
```

Out[7]:

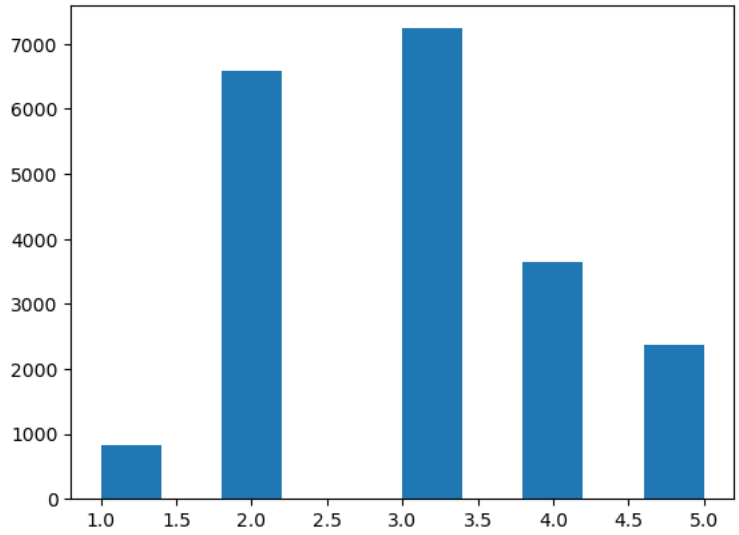
```
(16512, 4128)
```

```
As our dataset is not large enough when considering purely random sampling methods, we're running the chance of introducing a
significant sampling bias. so we are going for stratified sampling.


The code below transforms the median_income attribute into a categorical one by dividing the median income by 1.5 to limit the number of
income categories and rounds it up using "np.ceil()" to have discrete categories. It merges all the categories that are greater than 5
into category 5. The categories are represented in the histogram below the code.
```

In [8]:

```
housing["income_categories"] = np.ceil(housing["median_income"] / 1.5)
housing["income_categories"].where(housing["income_categories"] < 5, 5.0, inplace=True)
plt.hist(housing["income_categories"])
fig = plt.gcf()
```



Now we are ready to do stratified sampling based on income category:

In [9]:

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

Let's check if this worked as expected, we can start by checking the proportions of income categories in the test set:

In [10]:

```
strat_test_set['income_categories'].value_counts() / len(strat_test_set)
```

Out[10]:

```
3.0    0.350533
2.0    0.318798
4.0    0.176357
5.0    0.114341
1.0    0.039971
Name: income_categories, dtype: float64
```

Now that we have a test set that is representative of income_categories's distribution, it's time to remove it:

In [11]:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop('income_categories',axis=1,inplace=True)
```

We spent enough time on test set generation because this is an important part of any machine learning project. Moreover, many of these ideas will be useful later when we talk about cross-validation.

step 4: let's visualize and discover the Data to Gain Insights.

In [12]:

```
strat_train_set.shape, strat_test_set.shape
```

Out[12]:

```
((16512, 10), (4128, 10))
```

Let's create a copy of the training set for us to play with it without harming the original one:

In [13]:

```
housing = strat_train_set.copy(); housing.shape
```

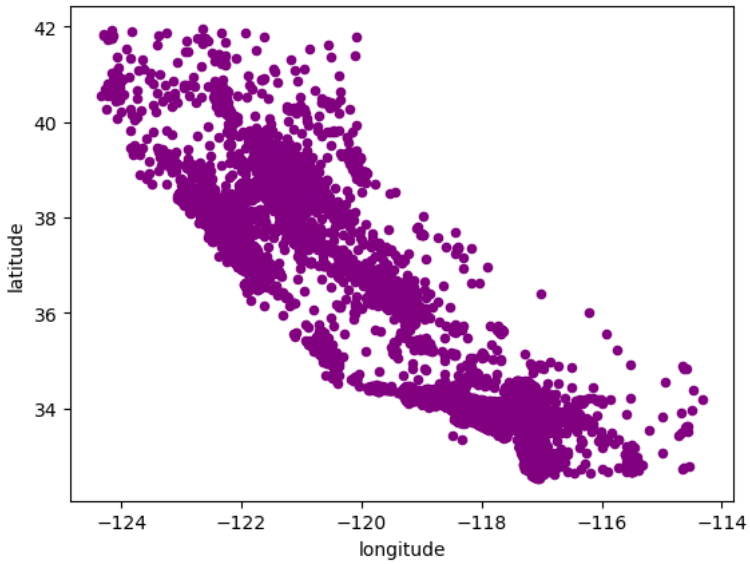
Out[13]:

```
(16512, 10)
```

step 4.1:Now we are going to visualize the Geographical Data

In [16]:

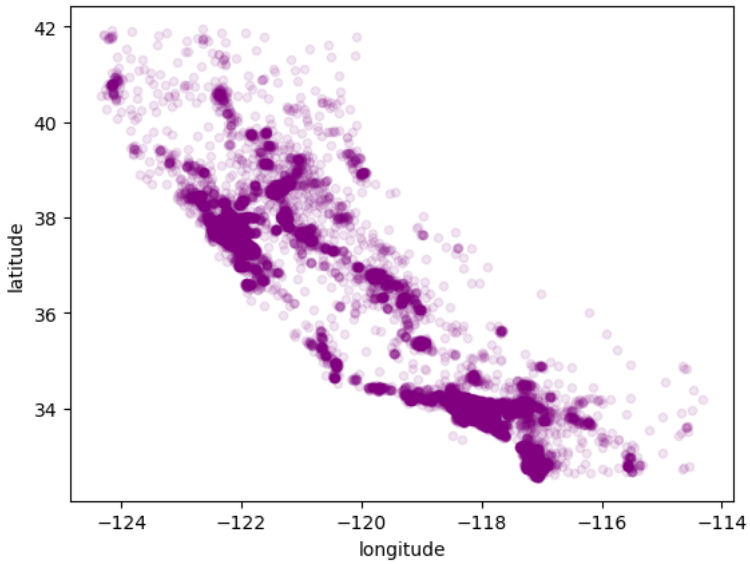
```
housing.plot(kind='scatter', x='longitude', y='latitude',color='purple')
plt.show()
```



This looks like california, but other than that, we can't really see any other pattern. Setting the alpha to 0.1 makes it much easier to estimate densities:

In [17]:

```
housing.plot(kind='scatter', x='longitude', y='latitude', alpha=0.1,color='purple')
plt.show()
```



As we can see from above the patterns hard to spot with naked eyes, so lets use matplotlib's visualization parameters to make the patterns stand out.

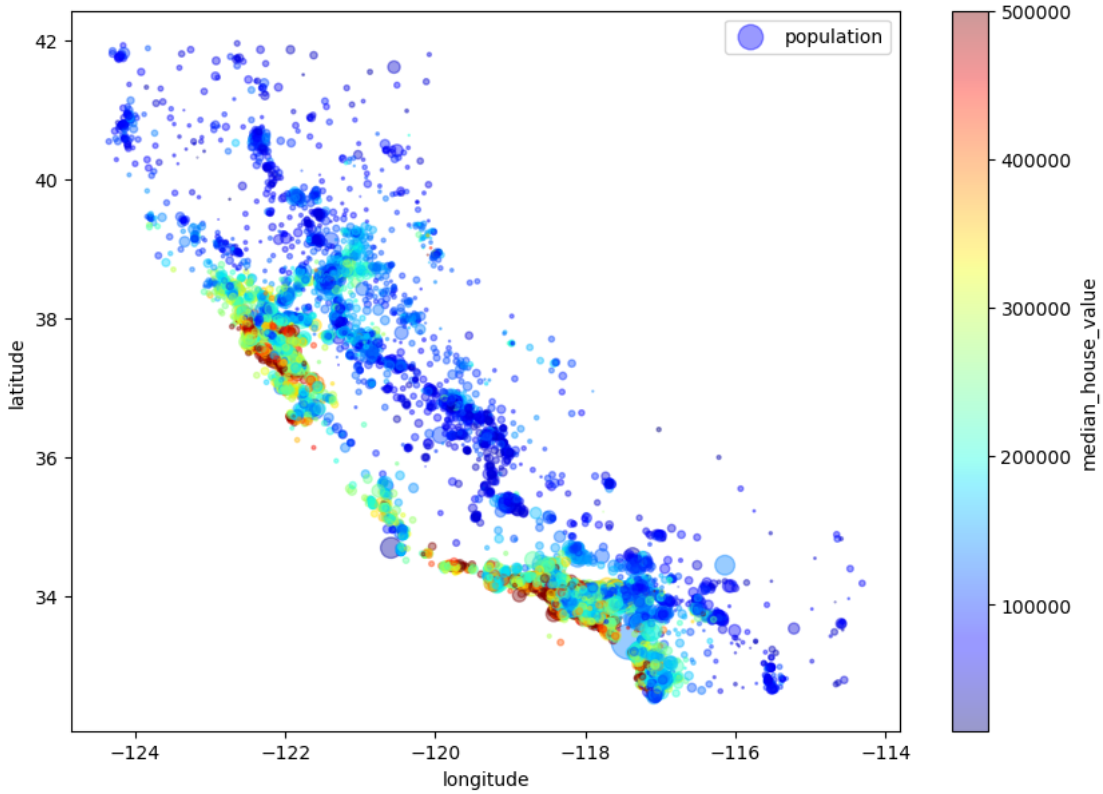
In the following figure, the radius of each circle represents the district's population (option s). The color represents the price(option c). We will also use a pre-defined color map called jet(option cmap) which ranges from blue(low levels) to red(high level).

In [18]:

```
housing.plot(kind='scatter', x='longitude', y='latitude', alpha=.4, s=housing['population']/100.,
             label='population', figsize=(10, 7), c='median_house_value', cmap=plt.get_cmap(name='jet'), colorbar=True)
plt.legend()
```

Out[18]:

```
<matplotlib.legend.Legend at 0x1f54d4c3520>
```



This image tells us that the median housing price is related to location for eg: the houses closer to the sea are more expensive. let's check the relation between population and price.

In [19]:

```
housing[['population', 'median_house_value']].corr()
```

Out[19]:

	population	median_house_value
population	1.000000	-0.026882
median_house_value	-0.026882	1.000000

```
A very weak pair-wise correlation between price and population.
```

step 4.2: Looking for Correlations

Since the dataset is not too large, we can easily compute the standard correlation coefficient of every pair of columns. Now let's look at how much each attribute correlates with house_median_value.

In [20]:

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

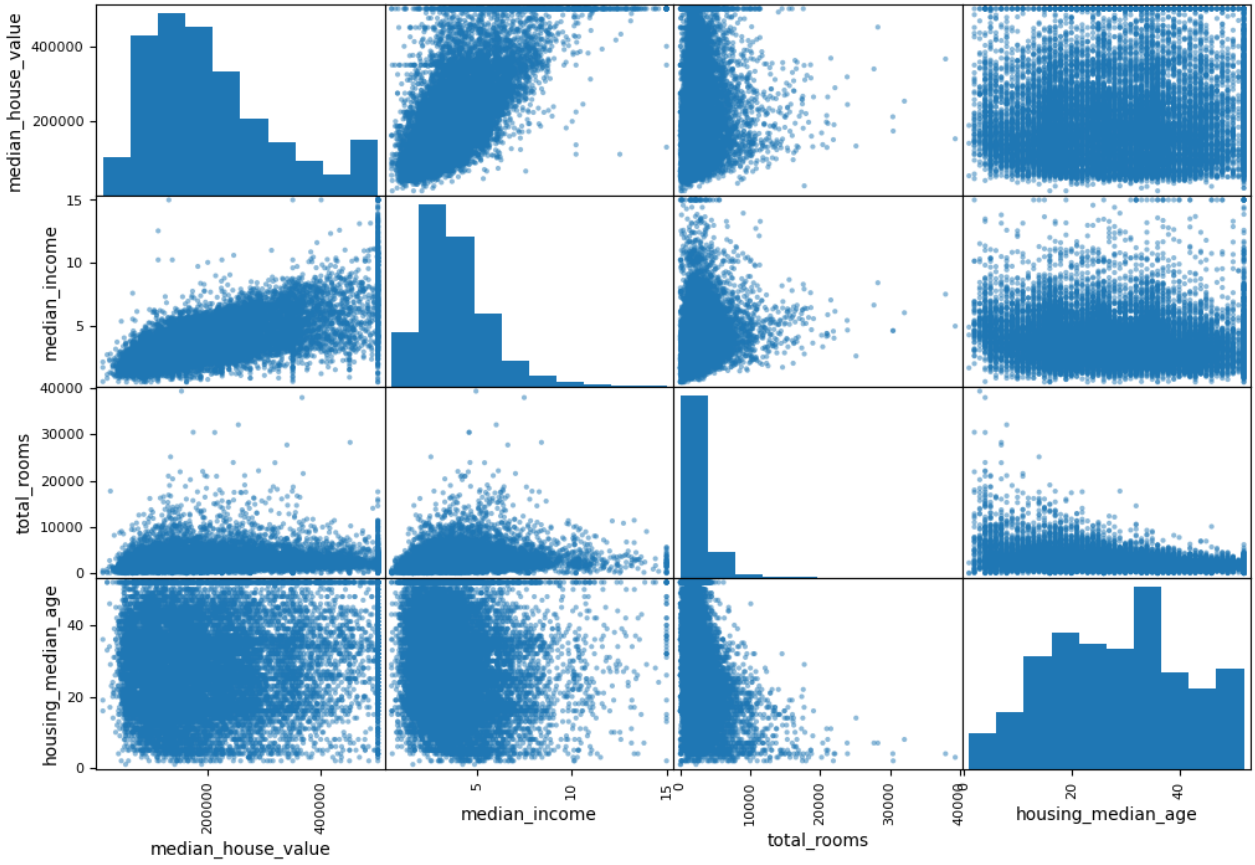
Out[20]:

```
median_house_value    1.000000
median_income         0.687151
total_rooms           0.135140
housing_median_age    0.114146
households            0.064590
total_bedrooms        0.047781
population           -0.026882
longitude            -0.047466
latitude             -0.142673
Name: median_house_value, dtype: float64
```

```
step 4.3: correlation using the Pandas .scatter_matrix() method to visualize
```

In [23]:

```
attributes = ['median_house_value', 'median_income', 'total_rooms', 'housing_median_age']
scatter_matrix(frame=housing[attributes], figsize=(12, 8))
plt.show()
```

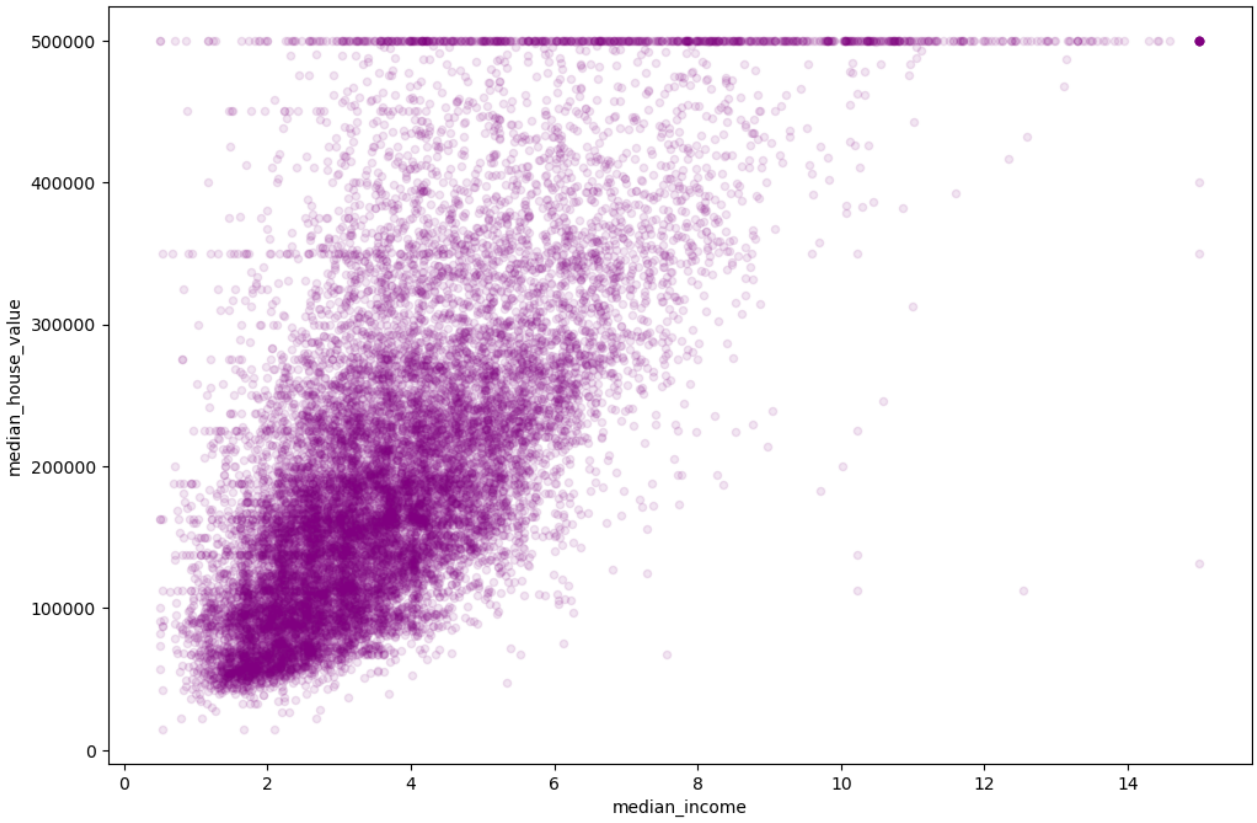


The most interesting attribute to predict median_house_value is median_income

Let's zoom in the correlation scatterplot:

In [24]:

```
housing.plot(kind='scatter', x='median_income', y='median_house_value', figsize=(12,8), alpha=0.1,color='purple')
plt.show()
```



This correlation is indeed very strong, we can clearly see the upward trend.

The price cap that we noticed earlier is clearly visible at 500,000 USD$, but the plot reveals other less obvious lines at: USD450K, USD350K, USD280K and so on.

We may want to remove the corresponding districts so that the model to not learn these quirks.

Step 4.4 : Experimenting with Attribute Combinations

We may want to transform tail heavy distributions using the logarithm function (log(.)).

One last thing we would want to do before feeding the data into an ML algorithm is to try to combine features.

Examples:

1.The number of rooms per household, not the total number of rooms in a district.
2.The total number of bedrooms isn't helpful either, we want to compare it with the number of rooms.
3.The number of poeple per household is also an interesting feature to look at.

Let's create them all:

In [25]:

```
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']
```

In [26]:

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

Out[26]:

```
median_house_value          1.000000
median_income               0.687151
rooms_per_household         0.146255
total_rooms                 0.135140
housing_median_age          0.114146
households                  0.064590
total_bedrooms              0.047781
population_per_household   -0.021991
population                 -0.026882
longitude                  -0.047466
latitude                   -0.142673
bedrooms_per_room          -0.259952
Name: median_house_value, dtype: float64
```

We have noticed that bedrooms_per_room is much more correlated with median_house_value. meaning that the more expensive the house, the less the bedrooms per room ratio. rooms_per_household have a moderate positive correlation with median_house_value, the more expensive a house is, the more rooms it will have.

step 5: Data preparation

In [27]:

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

Out[27]:

```
((16512, 9), (16512,))
```

step 5.1: Data Cleaning

As already observed that our data has some missing values and machine learning algorithms cannot deal with missing features. So let's try to deal with them using scikit-learn's SimpleImputer.

In [28]:

```
imputer = SimpleImputer(strategy='median')
```

Since the imputer can only work on numerical attributes, we need to create a copy of the dataFrame without the OCEAN_PROXIMITY text attribute:

In [29]:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Now we can just fit the imputer to the dataframe:

In [30]:

```
imputer.fit(housing_num)
```

Out[30]:

```
SimpleImputer(strategy='median')
```

The imputer has calculated the median of all attributes and stored them in .statistics_.

In [31]:

```
imputer.statistics_
```

Out[31]:

```
array([-118.51   ,   34.26   ,   29.     , 2119.     ,  433.     ,
       1164.     ,  408.     ,    3.54155])
```

In [32]:

```
housing_num.median().values
```

Out[32]:

```
array([-118.51   ,   34.26   ,   29.     , 2119.     ,  433.     ,
       1164.     ,  408.     ,    3.54155])
```

Now we can use the "trained or fitted" imputer to transform the numerical attributes by replacing missing values with their corresponding medians:

In [33]:

```
X = imputer.transform(housing_num)
X.shape
```

Out[33]:

```
(16512, 8)
```

The result is a numpy array containing the transformed features. If we want to put it back into a Pandas DataFrame, it's simple:

In [34]:

```
housing_tr = pd.DataFrame(data=X, index=housing_num.index, columns=housing_num.columns)
housing_tr.head()
```

Out[34]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
12655	-121.46	38.52	29.0	3873.0	797.0	2237.0	706.0	2.1736
15502	-117.23	33.09	7.0	5320.0	855.0	2015.0	768.0	6.3373
2908	-119.04	35.37	44.0	1618.0	310.0	667.0	300.0	2.8750
14053	-117.13	32.75	24.0	1877.0	519.0	898.0	483.0	2.2264
20496	-118.70	34.28	27.0	3536.0	646.0	1837.0	580.0	4.4964

step 5.2 : Handling Text & Categorical Attributes

So far, we have only dealt with numerical attributes, now, let's check text/categorical attributes.

We have only 1 categorical attribute, which is ocean_proximity. let's look at its values for the 10 first instances:

In [35]:

```
housing_cat = housing[['ocean_proximity']]
housing_cat.head(10)
```

Out[35]:

	ocean_proximity
12655	INLAND
15502	NEAR OCEAN
2908	INLAND
14053	NEAR OCEAN
20496	<1H OCEAN
1481	NEAR BAY
18125	<1H OCEAN
5830	<1H OCEAN
17989	<1H OCEAN
4861	<1H OCEAN

There is a limited number of values, each of which represents a category:

In [36]:

```
housing_cat['ocean_proximity'].value_counts()
```

Out[36]:

```
<1H OCEAN     7277
INLAND        5262
NEAR OCEAN    2124
NEAR BAY      1847
ISLAND           2
Name: ocean_proximity, dtype: int64
```

step 5.3: convert the text into ordinal categorical numbers

In [37]:

```
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat.values)
housing_cat_encoded.shape
```

Out[37]:

```
(16512, 1)
```

In [38]:

```
housing_cat_encoded[:10]
```

Out[38]:

```
array([[1.],
       [4.],
       [1.],
       [4.],
       [0.],
       [3.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

We can get the list of categories using the categories_ attribute of the OrdinalEncoder:

In [39]:

```
ordinal_encoder.categories_
```

Out[39]:

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

One issue with this representation is that the encoder will assume that two nearby categories are more similar than distant ones, but this is not the case for us (ex. categories 0 and 4 are clearly more similar than 0 and 1). To fix this issue, we create one binary attribute per category:

-One attribute is equal to 1 if the category is equal to <1H OCEAN and 0 otherwise. -One attribute is equal to 1 if the category is equal to INLAND and 0 otherwise.

The new attributes are sometimes called dummy attributes, let's create them:

In [40]:

```
one_hot_encoder = OneHotEncoder()
housing_cat_1hot = one_hot_encoder.fit_transform(housing_cat.values)
housing_cat_1hot
```

Out[40]:

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

-The output is a sparse scipy matrix instead of a numpy array. If we use numpy, we have to store all of the zeros in memory, comprising of most of the array.

-Instead, we store the information as a Scipy sparse matrix which only stores the locations of the non-zeros (which is more efficient).

- We can mostly use it as a normal 2D array, but if we want to convert it into a dense numpy array:

In [41]:

```
housing_cat_1hot.toarray()
```

Out[41]:

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

In [42]:

```
one_hot_encoder.categories_
```

Out[42]:

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

step 6: Custom Transformers

In [43]:

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

In [44]:

```
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    """Engineers new features from existing ones: `rooms_per_household`, `population_per_household`, `bedrooms_per_room`

    # Arguments:
        add_bedrooms_per_room, bool: defaults to True. Indicates if we want to add the feature `bedrooms_per_room`.
    """
    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  # We don't have any internal parameters. Only interested in transforming data.

    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]
```

In [45]:

```
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
```

In [46]:

```
housing_extra_attribs = attr_adder.transform(housing.values)
```

Step 6.1: Transformation Pipline

Scikit-learn provides the Pipeline class to help us chain transformations in a sequence. Here is a small pipeline for the numerical attribtues:

In [47]:

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

In [48]:

```
housing_num_tr = num_pipeline.fit_transform(housing_num)
housing_num_tr.shape
```

Out[48]:

```
(16512, 11)
```

Till here, we have handeled categorical/continuous columns separately. It would be better if we had a single transformer that is able to transform all columns.

step 6.2 : we will use ColumnTransformers.

In [49]:

```
num_attribs = housing_num.columns.tolist()
cat_attribs = ["ocean_proximity"]
full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", OneHotEncoder(), cat_attribs)
])
```

In [50]:

```
housing_prepared = full_pipeline.fit_transform(housing)
```

In [51]:

```
housing_prepared.shape
```

Out[51]:

```
(16512, 16)
```

Great!, we have a pre-processing pipeline that takes the data and transforms it into data that we can feed to an ML algorithm.

step 7: Model development and training

step 7.1: Linear Regression

In [52]:

```
lin_reg = LinearRegression()
```

In [53]:

```
lin_reg.fit(X=housing_prepared, y=housing_labels)
```

Out[53]:

```
LinearRegression()
```

Let's try the model on a few instances from the training set:

In [54]:

```
some_data = housing.iloc[:5]
```

In [55]:

```
some_labels = housing_labels.iloc[:5]
```

In [56]:

```
some_data_prepared = full_pipeline.transform(some_data)
print("Predictions: ", lin_reg.predict(some_data_prepared))
```

```
Predictions:  [ 85657.90192014 305492.60737488 152056.46122456 186095.70946094
 244550.67966089]
```

In [57]:

```
print("Labels: ", some_labels.tolist())
```

```
Labels:  [72100.0, 279600.0, 82700.0, 112500.0, 238300.0]
```

Even though the predictions are not exactly accurate, still it works.

Let's measure the performance of our model using the RMSE metric.

In [58]:

```
housing_predictions = lin_reg.predict(housing_prepared)
```

In [59]:

```
lin_mse = mean_squared_error(housing_labels, housing_predictions)
```

In [60]:

```
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

Out[60]:

```
68627.87390018745
```

```
Most districts median housing values range between 120K to 265K, so an average error of 68K is not good.

This is an example of a model overfitting the data. When this happens, it can mean two things:

-The features do not provide enough information to make better predictions.
-The model is not powerful enough, meaning its hypothesis space is narrow.

The main ways to tackle underfitting:

1.To feed the model better features.
2.To select a more powerful model.
3.To loosen the model's restrictions.

This model is not regularized, which rules out the last option. We could try to input more features, but let's start by testing a more
powerful model.
```

step 7.2: Let's try out DecisionTreeRegressor as this is a powerful model which is capable of finding non-linear relationships within the data.

In [61]:

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

Out[61]:

```
DecisionTreeRegressor()
```

In [62]:

```
housing_predictions = tree_reg.predict(housing_prepared)
```

In [63]:

```
tree_mse = mean_squared_error(y_true=housing_labels, y_pred=housing_predictions)
```

In [64]:

```
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

Out[64]:

```
0.0
```

It is either the model is absolutely perfect, or it badly overfit the data.

As we know, we shouldn't touch the test set. The solution is to partition the training data itself and extract a validation set.

step 8: Model Evaluation using Cross-Validation

One way to evaluate our model is to use train_test_split() again on the training set, extract a validation set and evaluate our iterative models on it.

A great alternative is to use K-fold cross-validation. We randomly split the training data into 10 folds, we iteratively train the model on 9 folds and evaluate on 1, doing this 10 times.

We will endup with 10 metric scores.

Step 8.1: Cross - Validation for Decision tree

In [65]:

```
scores = cross_val_score(estimator=tree_reg, X=housing_prepared,
                         y=housing_labels, scoring='neg_mean_squared_error', cv=10)
```

From the above cross validation output standard deviation (Std) tells us how precise our model is, and mean is the performance of the model.

In [66]:

```
tree_rmse_scores = np.sqrt(-scores)
```

scikit-learn's cross validation features expect a utility function (the greater the better) rather than a cost function (the lower the better). That's why we used ned_mean_squared_error and we negated it at RMSE evaluation

In [67]:

```
def display_scores(scores):
    """Displays the scores, their mean, and the standard deviation.

    # Arguments:
        scores, np.array: list of scores given by the cross validation procedure.
    """
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard Deviation:", scores.std())
```

In [68]:

```
display_scores(tree_rmse_scores)
```

```
Scores: [73376.18420354 71754.69777021 68960.93936081 70091.94689164
 69099.73396532 78331.85382317 71267.51352558 74130.86696342
 67981.56535076 71693.91047833]
Mean: 71668.92123328042
Standard Deviation: 2890.6233768452944
```

The decision tree seems to perform worse than the linear regression model!

We should notice that cross validation allows us to not only get an estimate of the performance of our model (mean), but how precise it is (std). We would not have this estimation if we used only one validation set. However, cross-validation comes at the cost of training the model several times, which is not always possible.

Let's compute the same scores for the linear regression model just to be sure:

Step 8.2: Cross - Validation for Linear Regression

In [69]:

```
scores = cross_val_score(estimator=lin_reg, X=housing_prepared,
                         y=housing_labels, scoring='neg_mean_squared_error', cv=10)
```

In [70]:

```
lin_rmse_scores = np.sqrt(-scores)
```

In [71]:

```
display_scores(lin_rmse_scores)
```

```
Scores: [71762.76364394 64114.99166359 67771.17124356 68635.19072082
 66846.14089488 72528.03725385 73997.08050233 68802.33629334
 66443.28836884 70135.96556127]
Mean: 69103.69661464215
Standard Deviation: 2880.190583299432
```

The decision tree model is overfitting so badly that it performs nearly worse as linear regression model.

step 7.3: Let's try one last model now, the random forest regressor. Random forests work by training many decision trees on random feature subsets then average out their predictions.

In [72]:

```
forest_reg = RandomForestRegressor()
forest_reg.fit(X=housing_prepared, y=housing_labels)
```

Out[72]:

```
RandomForestRegressor()
```

In [73]:

```
forest_mse = mean_squared_error(y_true=housing_labels, y_pred=forest_reg.predict(X=housing_prepared))
```

In [74]:

```
forest_rmse = np.sqrt(forest_mse)
```

In [75]:

```
forest_rmse
```

Out[75]:

```
18706.530001586234
```

Step 8.3: Cross - Validation for Random Forest

In [76]:

```
scores = cross_val_score(estimator=forest_reg, X=housing_prepared,
                         y=housing_labels, scoring='neg_mean_squared_error', cv=10)
```

In [77]:

```
forest_rmse_scores = np.sqrt(-scores)
```

In [78]:

```
display_scores(scores=forest_rmse_scores)
```

```
Scores: [51221.64533192 49045.75383536 46443.99285446 52279.65742396
 47536.34968552 51805.51485041 52358.72165003 50312.22905882
 48211.66063487 53946.36627571]
Mean: 50316.18916010512
Standard Deviation: 2302.3411567631238
```

```
Random forests seem promissing. We should notice, that the RMSE on the training set is still much lower then the validation RMSE,
meaning the model overfitted, but not as badly as the decision tree model.

Possible solutions to overfitting are:

1.Getting more training data
2.Simplifying the model
3.Regularizing the model
```

step 9. Fine-Tune our Model using Grid Search

As we can see from above random forest performs best among all the regressors we tried, so we choose random forest to perform fine-tuning.

we will use scikit-learn's GridSearchCV.

In [79]:

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

In [80]:

```
forest_reg = RandomForestRegressor()
```

In [81]:

```
grid_search = GridSearchCV(estimator=forest_reg, param_grid=param_grid, scoring='neg_mean_squared_error', cv=5, return_train_score=True, r
```

In [82]:

```
grid_search.fit(X=housing_prepared, y=housing_labels)
```

Out[82]:

```
GridSearchCV(cv=5, estimator=RandomForestRegressor(), n_jobs=-1,
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4],
                          'n_estimators': [3, 10]}],
             return_train_score=True, scoring='neg_mean_squared_error')
```

The model will first explore 3*4 combinations of hyper-parameters, then jump to the 2nd hyper-parameter space and try 1 x 2 x 3. For each combination, it will train 5 times using the cross validation strategy.

In [83]:

```
grid_search.best_params_
```

Out[83]:

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

```
To get the best estimator directly
```

In [84]:

```
grid_search.best_estimator_
```

Out[84]:

```
RandomForestRegressor(max_features=8, n_estimators=30)
```

Re-training the best model on the whole training data is generally a good practice. & ofcoures, the evaluation scores are also available.

In [85]:

```
cvres = grid_search.cv_results_
```

In [86]:

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

```
63654.623813293896 {'max_features': 2, 'n_estimators': 3}
55639.20288717565 {'max_features': 2, 'n_estimators': 10}
52671.14767038927 {'max_features': 2, 'n_estimators': 30}
59528.186222983 {'max_features': 4, 'n_estimators': 3}
51847.13028417252 {'max_features': 4, 'n_estimators': 10}
50588.50763683963 {'max_features': 4, 'n_estimators': 30}
59414.720739946926 {'max_features': 6, 'n_estimators': 3}
52208.0162447802 {'max_features': 6, 'n_estimators': 10}
50267.82673978195 {'max_features': 6, 'n_estimators': 30}
58599.67969097404 {'max_features': 8, 'n_estimators': 3}
52077.082768518194 {'max_features': 8, 'n_estimators': 10}
50131.05171283584 {'max_features': 8, 'n_estimators': 30}
62130.168953806795 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
53799.95044648434 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59511.28691045055 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52042.109051614265 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
58882.89394634933 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
52123.85550178776 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

In this example, the best hyper-parameter combination is 50138.25601132559 {'max_features': 8, 'n_estimators': 30} with an average RMSE of 50138. The model performs slightly better than a random forest with default hyper-parameters.

step 10 :Analyze the best models & their errors

As we often gain good insights about the problem by inspecting good models. For example, the random forest model can give us estimates over feature importance.

In [87]:

```
feature_importances = grid_search.best_estimator_.feature_importances_
(feature_importances*100).astype(int)
```

Out[87]:

```
array([ 6,  5,  4,  1,  1,  1,  1, 39,  3, 11,  6,  1, 14,  0,  0,  0])
```

In [88]:

```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
```

In [89]:

```
cat_encoder = full_pipeline.named_transformers_['cat']
```

In [90]:

```
cat_one_hot_attributes = cat_encoder.categories_[0].tolist()
```

In [91]:

```
attributes = num_attribs + extra_attribs + cat_one_hot_attributes
```

In [92]:

```
# sorted(zip(feature_importances, attributes), reverse=True)
dict(zip(feature_importances, attributes))
```

Out[92]:

```
{0.0669391525926582: 'longitude',
 0.05955357081145778: 'latitude',
 0.045448507449804096: 'housing_median_age',
 0.01536883386574255: 'total_rooms',
 0.015282786252789839: 'total_bedrooms',
 0.01563164606819322: 'population',
 0.015062293909834833: 'households',
 0.3905296814352555: 'median_income',
 0.037534844046205315: 'rooms_per_hhold',
 0.11372277395935236: 'pop_per_hhold',
 0.06013114695517695: 'bedrooms_per_room',
 0.011922117463731929: '<1H OCEAN',
 0.14630269122610845: 'INLAND',
 3.521357652513692e-05: 'ISLAND',
 0.0019802600457921385: 'NEAR BAY',
 0.004554480341371749: 'NEAR OCEAN'}
```

With this information, we might want to start dropping some of the attributes to simplify the model (ex. only one ocean_proximity value is important).

step 11: Evaluate our system on the test set

In [93]:

```
final_model = grid_search.best_estimator_
```

In [94]:

```
X_test = strat_test_set.drop(labels='median_house_value', axis=1)
```

In [95]:

```
y_test = strat_test_set['median_house_value'].copy()
```

In [96]:

```
X_test_prepared = full_pipeline.transform(X=X_test)
```

In [97]:

```
final_predictions = final_model.predict(X=X_test_prepared)
```

In [98]:

```
final_mse = mean_squared_error(y_true=y_test, y_pred=final_predictions)
```

In [99]:

```
final_rmse = np.sqrt(final_mse)
final_rmse
```

Out[99]:

```
47680.98841678697
```

```
We now have a final prediciton error as shown above and learned a lot of things on the way. Note that usually the performance on the
test set is slightly worse, because our system is fine tuned to perform well on the training set.
```

In some cases, such a point estimate of the generalization error won't be enough. We want to create a confidence interval of 95% around the metric.

For this, we use the individual predictions for each test set element.

In [100]:

```
confidence = .95
```

In [101]:

```
squared_errors = (y_test - final_predictions) ** 2
```

In [102]:

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

Out[102]:

```
array([45714.24158649, 49569.76325317])
```

If we do a lot of hyper-parameter fine-tuning, we will endup with a slightly worse performance on the test set because we will sometimes overfit to the changing validation set.

Summary

```
-We have framed a problem, plot histograms & draw conclusions from it, split data into training & testing subsets and we
understood tailheavyness.

-We worked on stratified sampling which is used on the income_categories feature. We also plotted scatterplots, computed a
correlation coefficient table.

-In terms of data preparation and cleaning, we talked about missing values and filled using scikit-learn's SimpleImputer. We
processed categorical attributes and text, using One hot encoding and reshaped arrays along the way.

-Furthermore we used the sklearn pipeline class,we used ColumnTransformers and used RMSE to evaluate our models and learned       about
underfitting.

-In Model development and training we used a Linear Regression model, a DecisionTreeRegressor and a RandomForestRegressor.

-we also performed RMSE and cross validation for each and every model and found out RandomForestRegressor best suites for our     data
set.

-we choose random forest regressor for hyper parameters with fine tuning using grid search.finally we evaluated our system on     the
test set.
```