CALIFORNIA HOUSING PREDICTION

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

Pinikeshi Varshith Reddy

12017306

Supervisor

Mr. VED PRAKASH

School of Computer Science and Engineering

Lovely Professional University

Phagwara, Punjab (India)

April 2023

DECLARATION STATEMENT

I hereby declare that the project work reported in the dissertation/dissertation proposal entitled "CALIFORNIA HOUSE PREDICTION" in partial fulfilment of the requirement for the award of Degree for Master of Technology in Computer Science and Engineering at Lovely Professional University, Phagwara, Punjab is an authentic work carried out under supervision of my research supervisor Mr. Ved Prakash I have not submitted this work elsewhere for any degree or diploma.

I understand that the work presented herewith is in direct compliance with Lovely Professional University's Policy on plagiarism, intellectual property rights, and highest standards of moral and ethical conduct. Therefore, to the best of my knowledge, the content of this dissertation represents authentic and honest research effort conducted, in its entirety, by me. I am fully responsible for the contents of my dissertation work.

ABSTRACT

The dataset we will use is the "California Housing Prices" dataset from the statlib repository, which is based on data from the 1990 census. This dataset offers great opportunities for learning. The prediction task for this dataset will be to predict housing prices based on several features.

The US Census Bureau has published California Census Data which has 10 types of metrics such as the population, median income, median housing price, households and so on for each block group in California. The dataset also serves as an input for project scoping and tries to specify the functional and non-functional r requirements for it.

To implement machine learning algorithms, it requires rudimentary data cleaning, understandable list of variables and sits at an optimal size between being too toyish and too cumbersome. The data contains information from the California census. So, although it may not help you with predicting current housing prices dataset, it does provide an accessible introductory dataset for teaching people about the basics of machine learning.

TABLE OF CONTENTS

PROBLEM OBJECTIVE

The project aims at building a model of housing prices to predict median house values in California using the provided dataset. This model should learn from the data and be able to predict the median housing price in any district, given all the other metrics. Districts or block groups are the smallest geographical units for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). There are 20,640 districts in the project dataset.

ABOUT THE DATASET

The data aren't cleaned so there are some pre-processing steps required! The columns are as follows their names are self-explanatory.

Longitude

Latitude

Housing_median_age

Total_rooms

Total_bedrooms

Population Households

Median_income

Median_house_value

ocean_proximity

Field and Description

1. Longitude (signed numeric - float): Longitude value for the block in California, USA

2. latitude (numeric - float): Latitude value for the block in California, USA

3. housing_median_age (numeric - int): Median age of the house in the block

4. total_rooms (numeric - int): Count of the total number of rooms (excluding bedrooms) in all houses in the block

5. total_bedrooms (numeric - float): Count of the total number of bedrooms in all houses in the block

6. population (numeric - int): Count of the total number of populations in the block

7. households (numeric - int): Count of the total number of households in the block

8. median_income (numeric - float): Median of the total household income of all the houses in the block

9. Ocean_proximity (numeric - categorical): Type of the landscape of the block [Unique Values: 'NEAR BAY', '

Analysis Tasks:

• Build a model of housing prices to predict median house values in California using the provided dataset.

• Train the model to learn from the data to predict the median housing price in any district, given all the other metrics.

• Predict housing prices based on median_income and plot the regression chart for it.

• Load the data

• Read the “housing.csv” file from the folder into the program.

• Print first few rows of this data.

• Extract input (X) and output (Y) data from the dataset.

• Handle missing values:

• Fill the missing values with the mean of the respective column.

• Encode categorical data:

• Convert categorical column in the dataset to numerical data.

• Split the dataset:

• Split the data into 70% training dataset and 30% test dataset.

• Standardize data:

• Standardize training and test datasets.

• Perform Linear Regression:

• Perform Linear Regression on training data.

• Predict output for test dataset using the fitted model.

• Print root mean squared error (RMSE) from Linear Regression.

• Extract just the median_income column from the independent variables (from X_train and X_test). • Perform Linear Regression to predict housing values based on median_income.

• Predict output for test dataset using the fitted model.

• Plot the fitted model for training data as well as for test data to check if the fitted model satisfies the test data.

• *Dataset Size: 20640 rows x 10 columns*

TASKS PERFORMED

Step1: Import all libraries.

Step2: Load the data. Step2.1: Read the “housing.csv” file from the folder into the program.

Step2.2: Print first few rows of this data.

Step2.3: Extract input (X) and output (y) data from the dataset

Step3: Handle missing values: Fill the missing values with the mean of the respective column.

Step4: Encode categorical data: Convert categorical column in the dataset to numerical data.

Step5: Split the dataset: Split the data into 80% training dataset and 20% test dataset. Step6: Standardize data: Standardize training and test datasets.

```
# Common imports
import pandas as pd
import numpy as np

# data visualization
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
import pandas as pd

def load_data(housing_path=path):
    csv = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv)
```

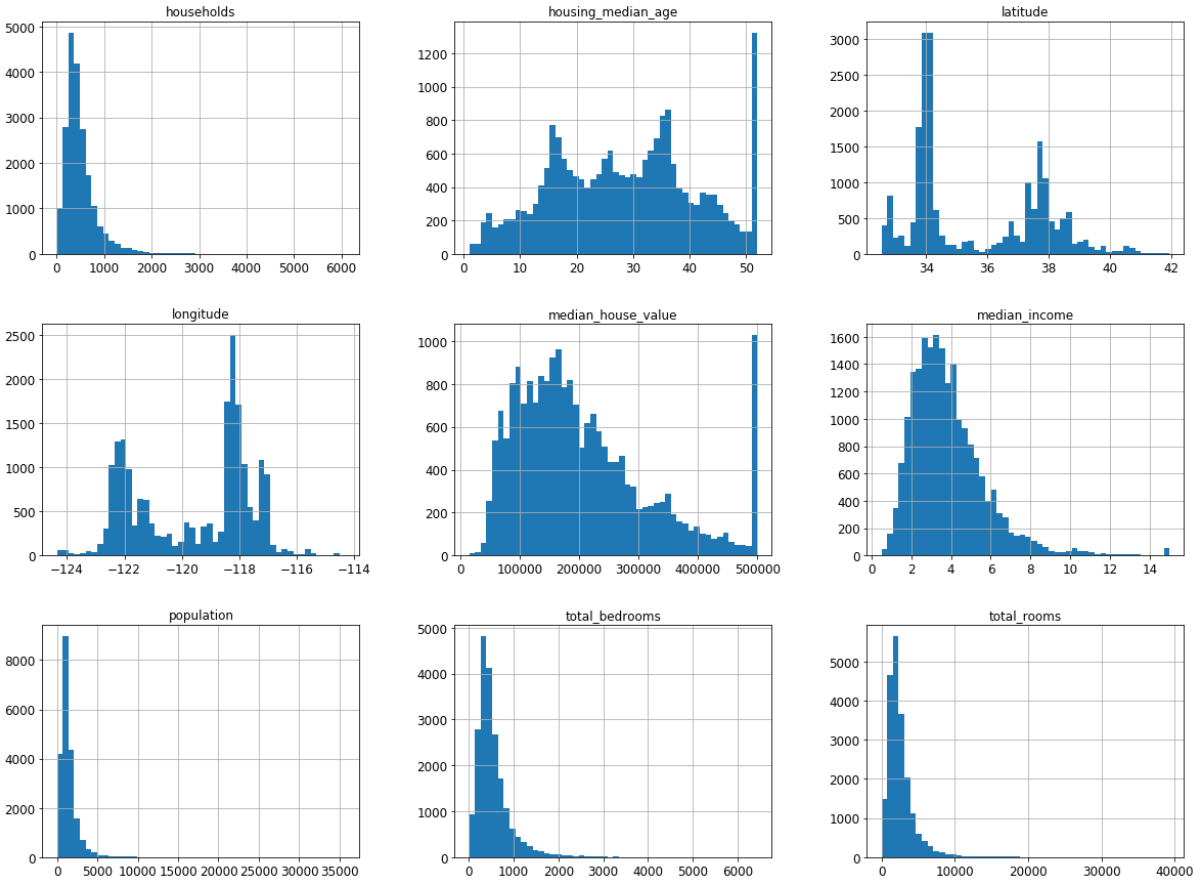
Below you can see the top 5 rows of the dataset with the "head ()" method.

Each row represents one district. The dataset has 10 attributes: longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, median_house_value and ocean_proximity.

```
In [5]: housing = load_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	45
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	35
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	35
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	34
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	34



What you should notice about the histograms above:

- The attributes have varying scales, which we will discuss later in this post.
- Many of the histograms are "tail heavy" which means that they extend further to the right of the median than to the left which makes it harder for an algorithm to detect patterns. We will transform the later.
- The median_income attribute is not in US Dollar because the data has been scaled and capped at 15.0001 and at 0.4999. This is called a "pre-processed attribute" and is common within machine learning, but you should understand how the data was pre-processed.
- The housing_median_age and the median_house_value attributes are also capped. That the median_house_value is capped could be a serious problem, because this is your label (what you want to predict) and your model could learn that the price never goes beyond that limit. In this case we only have the option to remove the capped one or to collect the right labels for them.

Train/Test Split

It is important that we now set a part of the data aside. Your brain is an amazing pattern detection system and therefore extremely prune to overfitting. If you would also look at the test set during the visual exploration, which we will do now, you may find some patterns unconsciously within the data, which let you select a particular algorithm that leads to an overfitted model. Because of that you could launch a system that performs not as well on new data than expected.

We use the straightforward "train_test_split ()" method from sklearn to split our data into train and test subsets.

```
In [10]: train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
         print(len(train_set), "Train Instances +", len(test_set), "Test Instances")
```

```
16512 Train Instances + 4128 Test Instances
```

Now we spited our data purely random, which is fine if you have a large dataset, but if it is not, you could have a sampling bias. When a company decides to call 10,000 people because of a survey, they want to make sure that these people represent the whole population. For an example the US population consists out of 49% male and 51% female, so a well conducted survey would try to maintain this ratio, which is called **stratified sampling**. The population is divided into homogeneous subgroups, called **strata** and the right number of instances is sampled from each **stratum** to makes sure that the data really represents the population.

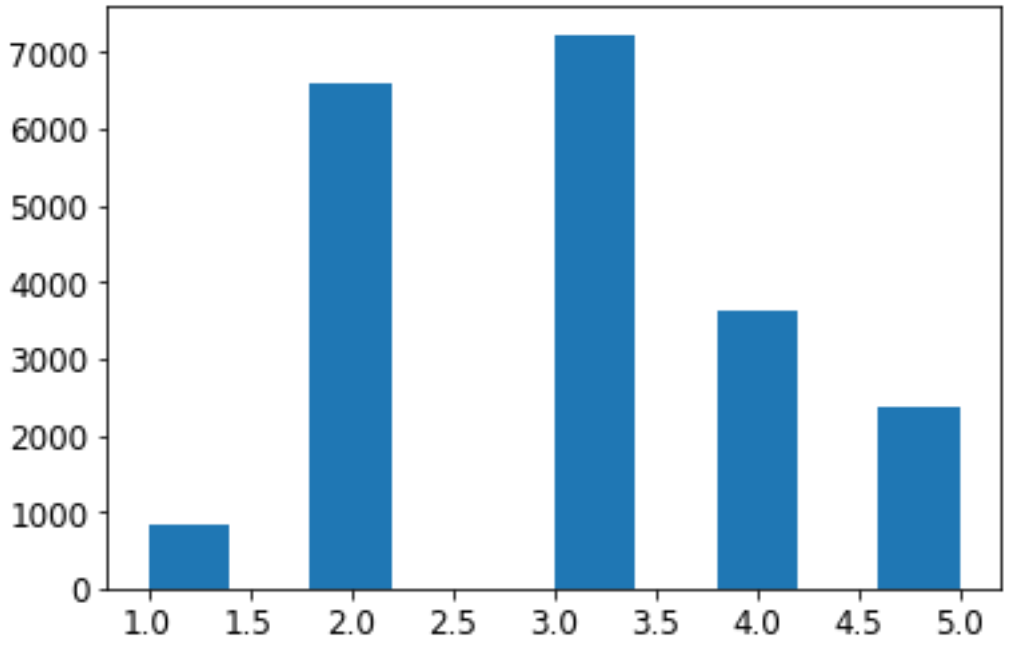
Imagine that you talked with real estate experts, and they tell you, that the median_income attribute is very important, when you want to predict housing prices. If that's the case, you want a test set that is representative of the income categories of the dataset. Because median_income is continuous, you need to convert it into a categorical attribute. If you look again at the holograms, we analysed detailed above, you may notice that most median income values are clustered around 20,000 - 50,000 but some go far beyond 60,000. It is very important that you don't have too many strata and that each stratum should have

enough instances. If this is not the case, the estimate of the stratum's importance may be biased, and your model could think that a stratum is less important.

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.

```
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()
```



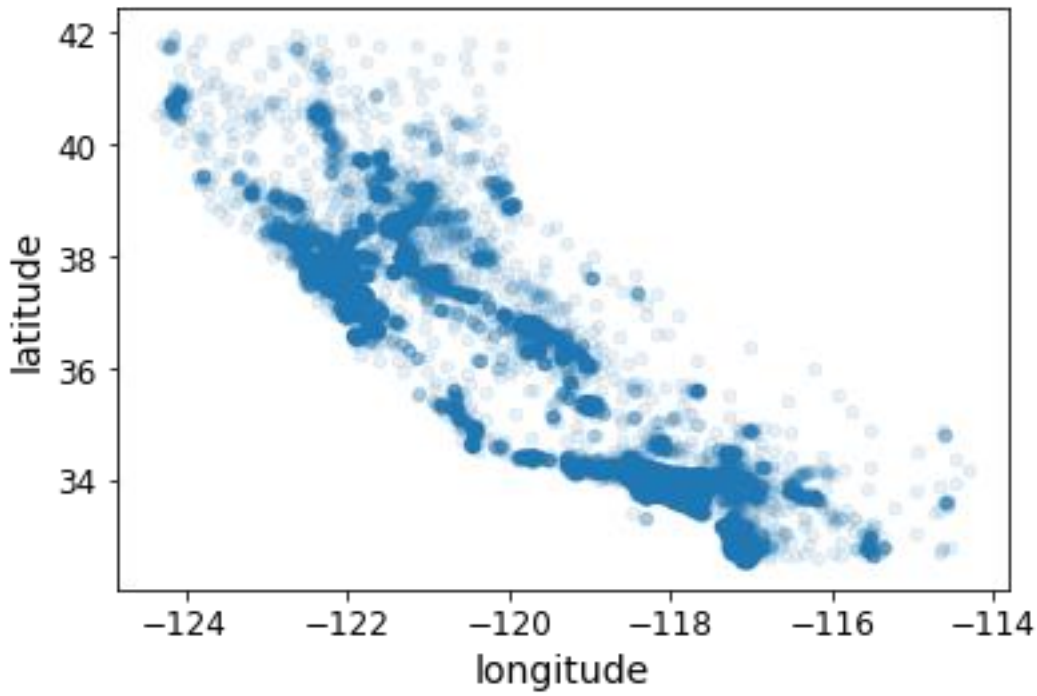
Finally, you need to do stratified sampling based on the income categories. You can use sklearn's "StratifiedShuffleSplit" class.

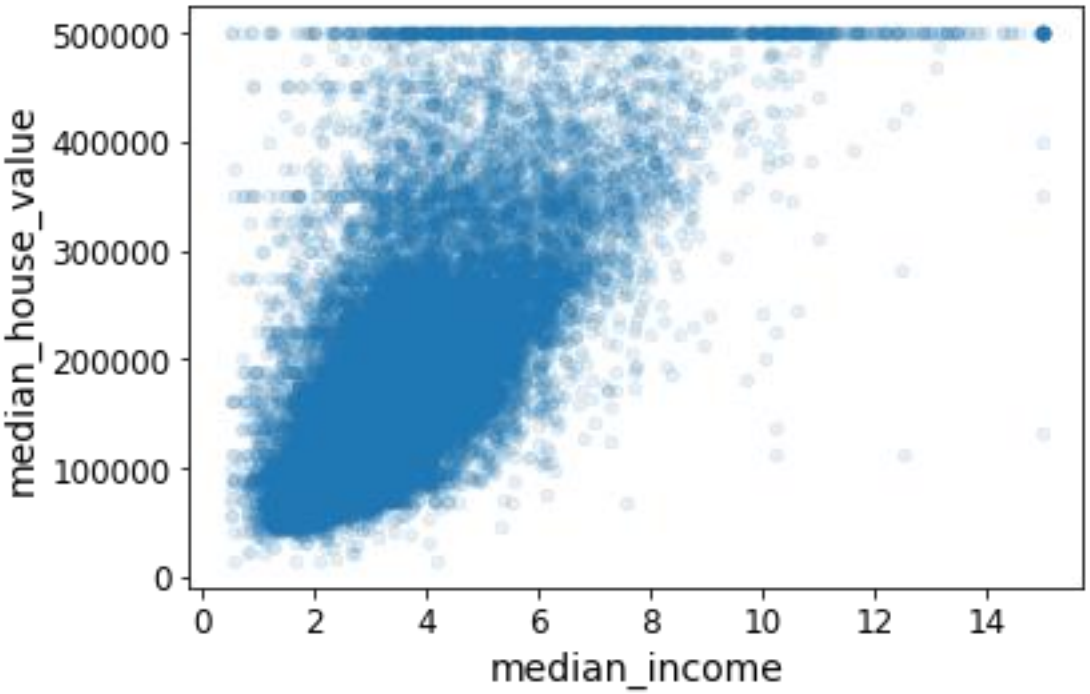
Now it is time for exploring the data. First, we want to visualize the geographical data with latitude and longitude. A good way to do this is to create a scatterplot of all the districts. It is important that you set alpha equal to 0.1, because then the scatterplot has a high density and therefore it is much easier to visualize.

SCATTER PLOT

Down below, if you are familiar with California, you can clearly see, the high density in the Bay Area, Los Angelos, San Diego and in the Central Valley around Sacramento and Fresno. Our brains are very good at finding patterns on pictures, but you often need to play around with the parameters to make the important patterns really stand out.

Now we will look at the housing prices at the scatterplot below. The colour represents the price, and the radius of each circle represents the districts population.





The scatterplot reveals that the correlation is indeed very strong because we can clearly see an upward trend and the points are not to dispersed. We can also clearly see the price-cap, we talked about earlier, at 500 000 as a horizontal line. Other less obvious lines are around 450 000, 350 000 and 280 000. We may have to remove the corresponding districts to prevent the model from learning to reproduce these data faults.

I hope that the previous explanation and visualizations made you more comfortable with the concepts of exploring data to gain insights. We identified a few very important faults in the data that we need to clean up before we can put the data into the machine learning model, that we will build later. We also found some interesting correlations between the different attributes, and we recognized the tail-heavy distribution, that we will also clean up later. Of course, a lot of things are different at every project you will work on, but the general guidelines will be the same and therefore you now already have a good understanding of some of those guidelines.

Before we now actually prepare the data to fed it into the model, we should think about combination a few attributes. For example, the number of rooms within a district is of course not very helpful, if you don't know how many households are within that district. You want the number of rooms per household. The number of bedrooms isn't that helpful for the same reason, but it would make sense to compare it with the total number of rooms within a household. Also, the population per household would be an interesting attribute.

It will create these new attributes in the code below and then we will look at the correlation matrix again.

```
median_house_value          1.000000
median_income               0.688075
rooms_per_household         0.151948
total_rooms                 0.134153
housing_median_age          0.105623
households                  0.065843
total_bedrooms              0.049686
population_per_household   -0.023737
population                 -0.024650
longitude                  -0.045967
latitude                   -0.144160
bedrooms_per_room          -0.255880
Name: median_house_value, dtype: float64
```

Now we see that the bedrooms_per_room attribute is more correlated with the median house value than the total number of rooms or bedrooms. Houses with a lower bedroom/room ratio tend to be more expensive. The rooms_per_household attribute is also better than the total number of rooms in a district. Obviously the larger the house, the higher the price.

The part of exploring the data does not have to be fully thorough and accurate. The reason why we do it, is to gain a few important insights, that we can use when we build our model. Building proper Machine Learning models is a very iterative process, which means you gain some insights, build a model and then you try to gain more insights and update the model that it works better and so on. You will continue to do this, till you reached a satisfying accuracy and will probably try out several algorithms and parameter adjustments on your way.

DATA PREPARATION

Now it is time to prepare the data so that our model can process it. We will write functions that do this instead of doing it manually. The reason for this is, that you can reuse these on a new dataset and at new projects you will work on. But first, let's revert to a clean training set by copying start_train_set and let's separate the predictors and the labels since we don't necessarily want to apply the same transformations to the predictors and the target values.

```
housing = strat_train_set.drop("median_house_value", axis=1) # drop labels for training set
housing_labels = strat_train_set["median_house_value"].copy()
```

We noticed earlier that the total_bedrooms attribute has some missing values. Most Machine Learning algorithms can't work with datasets that have missing values.

There are 3 ways to solve this problem:

1.) You could remove the whole attribute.

2.) You could get rid of the districts that contain missing values.

3.) You could replace them with zeros, the median or the mean.

We chose option 3 and will compute the median on the training set. Sklearn provides you with "Imputer" to do this. You first need to specify an Imputer instance, that specifies that you want to replace each attributes missing values with the median of that attribute. Because the median can only be computed on numerical attributes, we need to make a copy of the data without the ocean_proximity attribute that contains text and no numbers.

How to process Categorical Attributes and Text

Like I already mentioned most of the machine learning algorithms can just work with numerical data. The ocean proximity attribute is still a categorical feature, and we will convert it now. We can use Pandas factorize () method to convert this string categorical feature to an integer categorical feature.

```
housing_cat = housing['ocean_proximity']
housing_cat_encoded, housing_categories = housing_cat.factorize()
housing_cat_encoded[:10]
```

```
array([0, 0, 1, 2, 0, 2, 0, 2, 0, 0])
```

```
housing_categories
```

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

One issues with this solution is that our algorithm will assume that two nearby values are more similar than two distant values, which is obviously not the case. For example, categories 0 and 4 are more similar than categories 0 and 1. A common solution for this problem is to create one binary attribute per category: one attribute equal to 1 when the category is "<1H OCEAN" and 0 otherwise, another attribute equal to 1 when the category is "INLAND" and 0 otherwise, and so on. This is called "one-hot-encoding", because only one attribute will be equal to 1 (hot) and the others will be 0 (cold).

We will do exactly this in the code cell below. We use the OneHotEncoder encoder, that sklearn provides, to convert numeric categorical values into one-hot vectors. Note that "fit_transform()" expects a 2D array but "housing_cat" is a 1D array. Therefore, we need to reshape it.

FEATURE SCALING

Feature scaling is one of the most important transformations you need to apply, since nearly all machine learning algorithms perform bad when the input numerical attributes have widely varying scales, which is the case at our current dataset. For example, the median incomes range from o to 15, but the total number of rooms from 6 to 39,320. Note that scaling the target values is not required.

There are two common ways:

1.) min-max scaling

2.) standardization

With min-max scaling the values are shifted and rescaled that they range from 0 to 1. This can be done by subtracting the **min value** and dividing by the **max** minus the **min**. Sklearn's transformer called **MinMaxScaler**, which has a feature_range hyperparameter, that lets you change the range to whatever you want to (instead of 0 to 1), can do this for you.

Standardization subtracts the mean value and divides it by the variance. It does not bound the values into a specific range, which can be good or bad. Neural Networks for example often need input values that range from 0 to1. The good thing is, that standardization is less effected by outliers. If for example, you suppose a district has a median income of 100 (by mistake), min-max scaling would map the values that range from 0 to 15 down to 0 to 0.15, whereas standardization would give you a much more accurate outcome. With sklearn you can do standardization with the **StandardScaler** transformer. And always just do transformations with the training set only, to prevent overfitting.

PIPELINES FOR TRANSFORMATION

The pipeline is a Python scikit-learn utility for orchestrating machine learning operations. Pipelines function by allowing a linear series of data transforms to be linked together, resulting in a measurable modelling process.

There are a lot of data transformation steps that also need to be done in the right order. To help you with that, sklearn provides the **Pipeline** class. In the code below, you can see a small pipeline of attributes:

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

num_pipeline = Pipeline([
        ('imputer', Imputer(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]])
```

The Pipeline constructor takes a list of name/estimator pairs, which define a sequence of steps. Note that all instead of the last estimator must be transformers, which means that they must have a fit_transform () method. Also note that if you call the pipelines fit () method, it calls fit_transform () on all transformers. In our example, the last estimator is a StandardScaler (for standardization), so the pipeline has a transform () method that applies all the transform to the data in sequence.

Train Models

We looked at the big picture of your problem and framed it, we explored and visualized the data and pre-processed it. Now it is time to select and train a Machine Learning model. The hard part were the previous steps. What we do now, is going to be much simpler and easier.

LINEAR REGRESSION

Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

Task1.1: Perform Linear Regression on training data.

Task1.2: Predict output for test dataset using the fitted model.

Task1.3: Print root mean squared error (RMSE) from Linear Regression

First, let's test whether a Linear Regression model gives us a satisfying result.

```
from sklearn.linear_model import LinearRegression

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

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

ROOT MEAN SQUARE

RSME (Root mean square error) calculates the transformation between values predicted by a model and actual values. In other words, it is one such error in the technique of measuring the precision and error rate of any machine learning algorithm of a regression problem.

```
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
```

```
68376.642954599374
```

This is clearly not a great score. Since most districts median_housing_values range between 120,000 and 265,000 dollars, a prediction error of $68,376 is not very satisfying and an example of a model underfitting the data. This either means that the features do not provide enough information to make proper predictions, or that the model is just not powerful enough.

The main ways to fix underfitting are:

1.) feed the model with better features

2.) select a more powerful model

3.) reduce the constraints on the model

First let's try out a more powerful model since we just only tested one.

Let's use a DecisionTreeRegressor, which can find complex nonlinear relationships in the data.

DECISION TREE REGRESSSION

Decision tree builds regression or classification models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed.

Task2.1: Perform Decision Tree Regression on training data.

Task2.2: Predict output for test dataset using the fitted model.

Task2.3: Print root mean squared error from Decision Tree Regression.

```
from sklearn.tree import DecisionTreeRegressor

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

```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=42, splitter='best')
```

Let's try it out on the training set:

```
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
```

This gives you no error at all, which means that we strongly overfitted our data. How can, we be sure? As we already discussed earlier, you don't want to use the test set until you are confident about your model. But how can we test how our model performs if we can't use the test data? One way to do this is using **K-Fold Cross-Validation**, which uses part of the training set for training and a part for validation. The following code randomly splits the training set into 10 subset called **folds**. Then it trains and evaluates 10 times, using every fold once for either training or validation:

Now we have a real estimate of how the decision tree performs. In fact, it does even perform worse than our previous linear regression model. Note that cross validation not only gives you an estimate of your models' performance, but also of how precise this estimate is (standard deviation). Here we have a prediction error of 71,085 dollar. The standard deviation is 2,431 Dollar, which means that our prediction error could be $2,431 more or less.

K FOLD CROSS VALIDATION

K-fold Cross-Validation is when the dataset is split into a K number of folds and is used to evaluate the model's ability when given new data. K refers to the number of groups the data sample is split into.

In [52]:
```
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: [ 66877.52325028  66608.120256    70575.91118868  74179.94799352
  67683.32205678  71103.16843468  64782.65896552  67711.29940352
  71080.40484136  67687.6384546 ]
Mean: 68828.9994845
Standard deviation: 2662.76157061
```

Now we know that our decision tree model is overfitting and performs worse than our linear regression model. Now let's try one last model, the RandomForestRegressor. Random Forest works by training many Decision Trees on random subsets of the features and then it averages their predictions. Building a model on top of other models is called **Ensemble Learning** and often pushes Machine Learning algorithms further.

RANDOM FOREST REGRESSION

Random Forest Regression is a supervised learning algorithm that uses ensemble learning method for regression. Ensemble learning method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model.

Task3.1: Perform Random Forest Regression on training data.

Task3.2: Predict output for test dataset using the fitted model.

Task3.3: Print root mean squared error from Random Forest Regression.

```
from sklearn.ensemble import RandomForestRegressor

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

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
           oob_score=False, random_state=42, verbose=0, warm_start=False)
```

This is a lot better. Note that the score on the training set is still much lower, than on the validation set, which indicates that the model is overfitting the training set and that we should optimize the model to solve this problem. However, you shouldn't fine tune a model too much before you haven't tried out various other machine learning models from different categories of algorithms, without spending too much time tweaking it. You could use different Support Vector Machines with different Kernels, maybe a neural network and so on. The goal is that you have a rough estimate about, what algorithms performs the best and then you should spend time fine tuning them for a better accuracy.

FINE TUNING

Now let's assume you played around with different algorithms and have a rough estimate over the best performing ones. We will now look at Grid Search as an example to fine tune your model.

GRID SEARCH

Grid search is a method for performing hyper-parameter optimisation, that is, with a given model (e.g., a CNN) and test dataset, it is a method for finding the optimal combination of hyper-parameters (an example of a hyper-parameter is the learning rate of the optimiser).

Instead of trying out different hyperparameters manually until you find some great combinations, you can let Grid Search do that for you. All you need to do is tell it which hyperparameters you want to experiment with and what values. The code below searches for the best combination of hyperparameter values for the RandomForestRegressor.

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

```
64265.8521009 {'max_features': 2, 'n_estimators': 3}
55882.9848402 {'max_features': 2, 'n_estimators': 10}
53472.5297705 {'max_features': 2, 'n_estimators': 30}
61320.6172102 {'max_features': 4, 'n_estimators': 3}
53834.6661703 {'max_features': 4, 'n_estimators': 10}
51273.2598733 {'max_features': 4, 'n_estimators': 30}
59851.1600773 {'max_features': 6, 'n_estimators': 3}
53108.4926792 {'max_features': 6, 'n_estimators': 10}
50804.4906775 {'max_features': 6, 'n_estimators': 30}
59225.2197785 {'max_features': 8, 'n_estimators': 3}
52883.782581 {'max_features': 8, 'n_estimators': 10}
50942.1591379 {'max_features': 8, 'n_estimators': 30}
62801.357247 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54452.7056215 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
61122.9494918 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
53014.3329645 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
60252.6537668 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
52716.5505964 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

The RMSE score for this combination is 50,804. Now you have successfully fine-tuned your model 1 Alternatives to Grid Search would be randomized search.

CONCLUSION

To sum up, we have got the solution that can predict the mean house value in the block with RMSE $46k using our best model - LGB. It is not an extremely precise prediction: $46k is about 20% of the average mean house price, but it seems that it is near the possible solution for these classes of model based on this data (it is popular dataset, but I have not found any solution with significantly better results).

We have used old Californian data from 1990 so it is not useful right now. But the same approach can be used to predict modern house prices (if applied to the resent market data).

link code
We have done a lot, but the results surely can be improved, at least one could try:

- feature engineering: polynomial features, better distances to cities (not Euclidean ones, ellipse representation of cities), average values of target for the geographically closest neighbours (requires custom estimator function for correct cross validation)
- PCA for dimensionality reduction (I have mentioned it but didn't used)
- more time and effort can be spent on RF and LGB parameters tuning.

EVALUATION

Now we can evaluate the final model on the test set. You just must get the labels from your test set, run your full_pipeline to transform the data and evaluate the model.

```
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
```

```
49036.586427241818
```

We now have a final prediction error of $49,036 and learned a lot of things on the way. Note that usually the performance on the test set is slightly worse, because your system is fine tuned to perform well on the training set.

Hopefully all of this gave you a good idea of what a machine learning project looks like and a blueprint for you to build great systems in the future. As I already mentioned, most of the work must be done before you select and train an algorithm. Note that it is probably better to know four or five machine learning algorithms very well rather than spending all your time exploring advanced algorithms and not enough time on the overall process.

SUMMARY

Now we have a pretty good understanding of how supervised regression works and learned a massive number of things on the way.

We learned how to frame a problem, plot histograms & draw conclusions from it, split data into training & testing subsets and what tail-heaviest is & how to solve it.

We now know why stratified sampling can be very important and we used it on the income_categories feature, after we created it. We also plotted scatterplots, computed a correlation coefficient table, discussed scaled & capped features, and even wrote functions for feature engineering.

In terms of data preparation, we talked about missing values and filled some. We processed categorical attributes and text, using One hot encoding and reshaped arrays along the way. We did some feature scaling and learned that most machine learning algorithms perform bad when the input numerical attributes have widely varying scales.

Furthermore, we used the sklearn pipeline class, joined two pipelines into one, used RMSE to evaluate our models and learned a lot about over- and underfitting.

On top of that we used K-fold cross validation and tuned our hyperparameters with grid search.

Along the project we used a Linear Regression model, a DecisionTreeRegressor and a RandomForestRegressor.