Machine Learning

| Prepare The Data for the ML Algorithms

Prof. Dr. Mostafa Elhosseini

https://youtube.com/drmelhosseini

Agenda

- Prepare the Date
- Missing Values

Note

- Revert to a clean training set (by copying **strat_train_set** once again).
- You should also **separate the predictors and the labels**, since you don’t necessarily want to apply the same transformations to the predictors and the target values
- Note that **drop()** creates a copy of the data and does not affect strat_train_set)

Note

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

```
housing.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
13096	-122.42	37.80	52.0	3321.0	1115.0	1576.0	1034.0	2.0987	NEAR BAY
14973	-118.38	34.14	40.0	1965.0	354.0	666.0	357.0	6.0876	<1H OCEAN
3785	-121.98	38.36	33.0	1083.0	217.0	562.0	203.0	2.4330	INLAND
14689	-117.11	33.75	17.0	4174.0	851.0	1845.0	780.0	2.2618	INLAND
20507	-118.15	33.77	36.0	4366.0	1211.0	1912.0	1172.0	3.5292	NEAR OCEAN

Note

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

```
housing_labels.head()
```

```
13096    458300.0
14973    483800.0
3785     101700.0
14689     96100.0
20507    361800.0
Name: median_house_value, dtype: float64
```

Prepare the Data for Machine Learning

- To ensure efficient preparation of data for machine learning algorithms, it is advisable to **automate this process** through the development of **specific functions**. This approach is beneficial for several reasons:
  - Ensures consistent application of transformations across **different datasets**.
  - Builds a **reusable library** of transformation functions for future projects.
  - Facilitates the **preprocessing of new data** in live systems before analysis.
  - Allows for **easy experimentation with different transformations** to identify the most effective techniques.

Missing Values

- Everywhere, there are missing values, and you don't want those to interfere with your work.
- **NaN** is the symbol for not a number in Pandas Dataframe, and it indicates missing values.
- Most machine learning algorithms cannot work with missing features, so you'll need to take care of these.

Missing Values

```
<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
```

Missing Values

```
housing.isna()
```

- Every value is given a Boolean stating whether a value is missing

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...
20635	False	False	False	False	False	False	False	False	False	False
20636	False	False	False	False	False	False	False	False	False	False
20637	False	False	False	False	False	False	False	False	False	False
20638	False	False	False	False	False	False	False	False	False	False
20639	False	False	False	False	False	False	False	False	False	False

Missing Values

```
housing.isna().any()
```

- If there are any missing value in that column

```
longitude             False
latitude              False
housing_median_age    False
total_rooms           False
total_bedrooms         True
population            False
households            False
median_income         False
median_house_value    False
ocean_proximity       False
dtype: bool
```

Missing Values

```
housing.isna().sum()
```

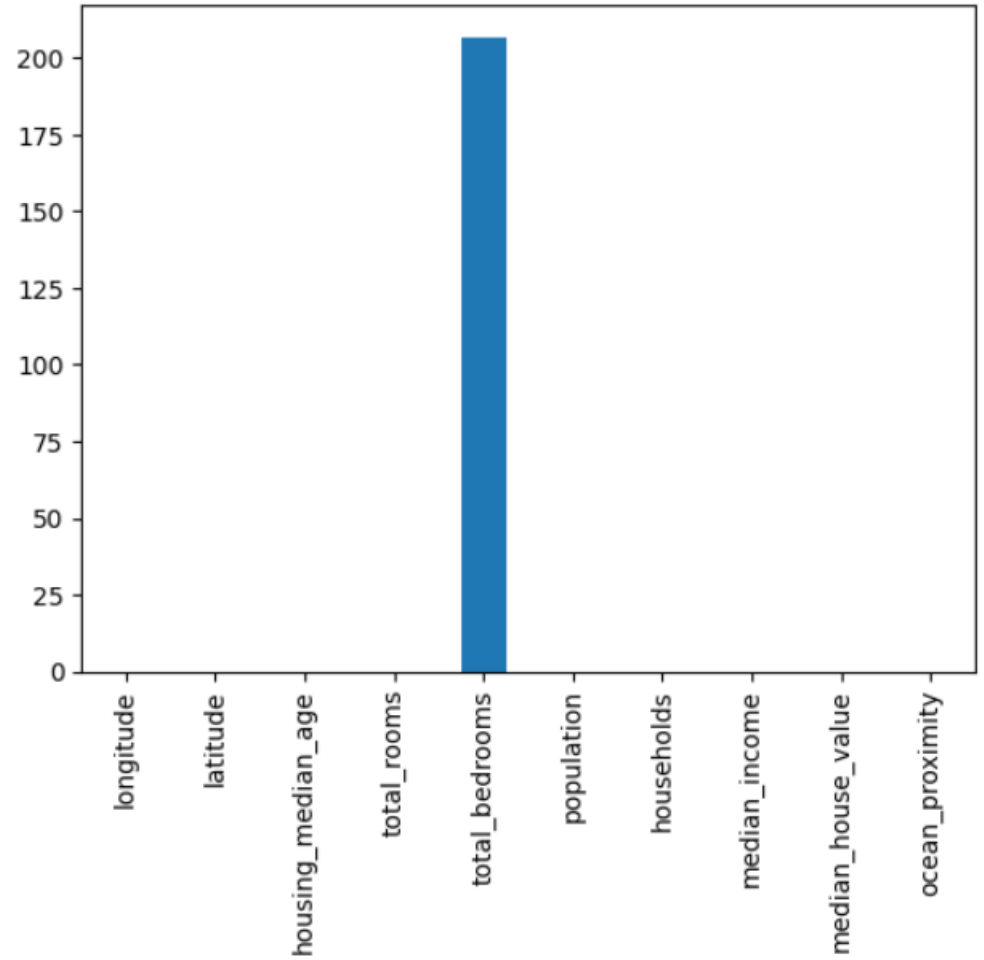
- Count the number of NaN in each column

```
longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms      207
population            0
households            0
median_income         0
median_house_value    0
ocean_proximity       0
dtype: int64
```

Missing Values

- Visualize missing values in the dataset

```
housing.isna().sum().plot(kind = 'bar')
```



Missing Values

- After we have identified that we have any missing values, what can we do about them
- The **dropna( )** function comes to your aid

Missing Values

- After we have identified that we have any missing values, what can we do about them
- The **dropna( )** function comes to your aid
- You have three options to fix this:
  - Get rid of the corresponding districts.
  - Get rid of the whole attribute.
  - Set the missing values to some value (zero, the mean, the median, etc.). This is called **imputation**.

Missing Values

- Rows that originally contained a NaN value.

```
null_rows_idx = housing.isnull().any(axis=1)
housing.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	NaN	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	NaN	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	NaN	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	NaN	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	NaN	375.0	183.0	9.8020	<1H OCEAN

Missing Values

- Get rid of the corresponding districts.

```
housing_option1 = housing.copy()

housing_option1.dropna(subset=["total_bedrooms"], inplace=True)  # option 1

housing_option1.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity

Missing Values

- Get rid of the whole attribute.

```
housing_option2 = housing.copy()

housing_option2.drop("total_bedrooms", axis=1, inplace=True)  # option 2

housing_option2.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	375.0	183.0	9.8020	<1H OCEAN

Missing Values

- Set the missing values to some value (zero, the mean, the median, etc.). This is called **imputation**.

```
housing_option3 = housing.copy()

median = housing["total_bedrooms"].median()
housing_option3["total_bedrooms"].fillna(median, inplace=True)  # option 3

housing_option3.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	434.0	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	434.0	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	434.0	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	434.0	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	434.0	375.0	183.0	9.8020	<1H OCEAN

Missing Values

SimpleImputer

- Store the median value of each feature: this will make it possible to impute missing values not only on the training set, but also on the validation set, the test set, and any new data fed to the model.

```
from sklearn.impute import SimpleImputer

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

- Since the **median** can only be computed on **numerical attributes**, you then need to create a copy of the data with only the numerical attributes

```
housing_num = housing.select_dtypes(include=[np.number])
```

Missing Values

SimpleImputer

- Fit the imputer instance to the training data using the fit() method

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

- The imputer has simply **computed the median of each attribute** and stored the result in its **statistics_ instance** variable.
- Only the total_bedrooms attribute had missing values, but you **cannot be sure** that there won't be any missing values in new data after the system goes live, so it is safer to apply the imputer to all the numerical attributes

Missing Values

SimpleImputer

- Fit the imputer instance to the training data using the fit() method

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

- The imputer has simply **computed the median of each attribute** and stored the result in its **statistics_ instance** variable.
- Only the total_bedrooms attribute had missing values, but you **cannot be sure** that there won't be any missing values in new data after the system goes live, so it is safer to apply the imputer to all the numerical attributes

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

Missing Values

SimpleImputer

- The imputer has simply computed the median of each attribute and stored the result in its **statistics_** instance variable.

```
imputer.statistics_
```

```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

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

```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

Missing Values

SimpleImputer

- Use this “trained” imputer to transform the training set by replacing missing values with the learned medians.
- Transform the training set

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

```
imputer.feature_names_in_
```

```
array(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income'],
      dtype=object)
```

Missing Values

SimpleImputer

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

```
housing_tr.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
14452	-120.67	40.50	15.0	5343.0	434.0	2503.0	902.0	3.5962
18217	-117.96	34.03	35.0	2093.0	434.0	1755.0	403.0	3.4115
11889	-118.05	34.04	33.0	1348.0	434.0	1098.0	257.0	4.2917
20325	-118.88	34.17	15.0	4260.0	434.0	1701.0	669.0	5.1033
14360	-117.87	33.62	8.0	1266.0	434.0	375.0	183.0	9.8020

Missing Values

- Missing values can also be replaced with the mean value (strategy="**mean**"), or with the most frequent value (strategy="**most_frequent**"), or with a constant value (strategy="constant", **fill_value**=…).
- The last two strategies support non-numerical data.

Missing Values

- There are **also more powerful imputers** available in the sklearn.impute package (both for **numerical features** only):
- **KNNImputer** replaces each missing value with the **mean of the k-nearest neighbors'values** for that feature. The distance is based on all the available features.
- **IterativeImputer** trains a regression model per feature to predict the missing values based on all the other available features. It then trains the model again on the updated data, and repeats the process several times, improving the models and the replacement values at each iteration.