

# **Machine Learning**

## **LAB**



### **Lab #2**

**Prepare the Data For Machine Learning Algorithm**

**Instructor: Saad Rashad**

**Course Code: AL3002**

**Semester Fall 2024**

**Department of Computer Science,  
National University of Computer and Emerging Sciences FAST  
Peshawar Campus**

## 1. Data Cleaning:

- Most Machine Learning algorithms cannot work with missing features, so let's create a few functions to take care of them.
- total\_bedrooms attribute has some missing values(207), so let's fix this.

```
▶ null_values=df['total_bedrooms'].isnull().sum()  
print(null_values)  
↔ 207
```

- You can accomplish these easily using DataFrame's dropna(), drop(), and fillna() methods:

### 1.1. Data Removal:

```
▶ #df.dropna(subset=["total_bedrooms"], inplace=True) Option 1  
  
#df.drop("total_bedrooms", axis=1) Option 2  
|  
median = df["total_bedrooms"].median() # option 3  
df["total_bedrooms"].fillna(median, inplace=True)
```

- Where inplace = true and axis =1
- “When *inplace=True*, the operation is performed directly on the original DataFrame.”
- ‘axis=0’ Refers to rows. And ‘axis=1’ Refers to columns. It means the operations will be performed either row or column wise.
- Instead of performing a destructive methodology we will have another method that is the less destrcutive ”Option 3”.
- We will use a handy Scikit-Learn class : **SimpleImputer**. The benefit is that it will 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 we can implement **imputer**.

## 1.2. Data Imputation:

- Here is how to use it. First, you need to create a SimpleImputer instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
import numpy as np
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='median')
```

- Since the median can only be computed on numerical attributes, you need to create a copy of the data with only the numerical attributes (this will exclude the text attribute ocean\_proximity):

```
housing_data=df.select_dtypes(include=[np.number])
```

- Now you can fit the imputer instance to the training data using the fit() method:

```
[19] imputer.fit(housing_data)
```

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

- 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 we 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:

```

▶ imputer.statistics_
housing_data.median().values
↵ array([-1.1849e+02,  3.4260e+01,  2.9000e+01,  2.1270e+03,  4.3500e+02,
        1.1660e+03,  4.0900e+02,  3.5348e+00,  1.7970e+05])

```

Loading...

- Now you can use this “trained” imputer to transform the training set by replacing missing values with the learned medians:

```

▶ X=imputer.transform(housing_data)
print(X)
↵ [[-1.2223e+02  3.7880e+01  4.1000e+01 ...  1.2600e+02  8.3252e+00
    4.5260e+05]
   [-1.2222e+02  3.7860e+01  2.1000e+01 ...  1.1380e+03  8.3014e+00
    3.5850e+05]
   [-1.2224e+02  3.7850e+01  5.2000e+01 ...  1.7700e+02  7.2574e+00
    3.5210e+05]
   ...
   [-1.2122e+02  3.9430e+01  1.7000e+01 ...  4.3300e+02  1.7000e+00
    9.2300e+04]
   [-1.2132e+02  3.9430e+01  1.8000e+01 ...  3.4900e+02  1.8672e+00
    8.4700e+04]
   [-1.2124e+02  3.9370e+01  1.6000e+01 ...  5.3000e+02  2.3886e+00
    8.9400e+04]]

```

- 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.
- The output of **imputer.transform(housing\_data)** is a NumPy array: X has neither column names nor index. Luckily, it's not too hard to wrap X in a DataFrame and recover the column names and index from **housing\_data**:

```

housing_tr = pd.DataFrame(X, columns=housing_data.columns,
index=housing_data.index)
print(housing_tr)

```

```

longitude  latitude  housing_median_age  total_rooms  total_bedrooms
0      -122.23      37.88              41.0         880.0           129.0
1      -122.22      37.86              21.0        7099.0          1106.0
2      -122.24      37.85              52.0        1467.0           190.0
3      -122.25      37.85              52.0        1274.0           235.0
4      -122.25      37.85              52.0        1627.0           280.0
...      ...      ...              ...          ...           ...
20635   -121.09      39.48              25.0        1665.0           374.0
20636   -121.21      39.49              18.0         697.0           150.0
20637   -121.22      39.43              17.0        2254.0           485.0
20638   -121.32      39.43              18.0        1860.0           409.0
20639   -121.24      39.37              16.0        2785.0           616.0

population  households  median_income  median_house_value

```

## 2. Data Transformation:

### 2.1. Handling Text and Categorical Data:

- In this dataset, there is just one: the **ocean\_proximity** attribute is in textual format.

```
housing_categorical=df["ocean_proximity"]
housing_categorical.head(500)
```

	ocean_proximity
0	NEAR BAY
1	NEAR BAY
2	NEAR BAY
3	NEAR BAY
4	NEAR BAY
...	...
495	INLAND
496	<1H OCEAN
497	<1H OCEAN
498	<1H OCEAN
499	NEAR OCEAN

500 rows × 1 columns

**dtype:** object

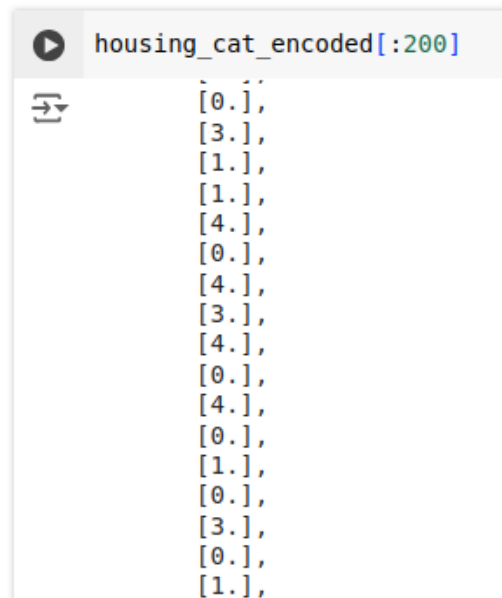
- Machine Learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's OrdinalEncoder class:

### 2.1.1. Working of Ordinal encoder :

- Encode categorical features as an integer array.
- Ordinal encoding is a preprocessing technique used for converting categorical data into numeric values that preserve their inherent ordering.
- Ordinal encoding works by mapping each unique category value to a different integer. Typically, integers start at 0 and increase by 1 for each additional category

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_categorical)
```

- Here's what the first few encoded values in `housing_cat_encoded` look like:

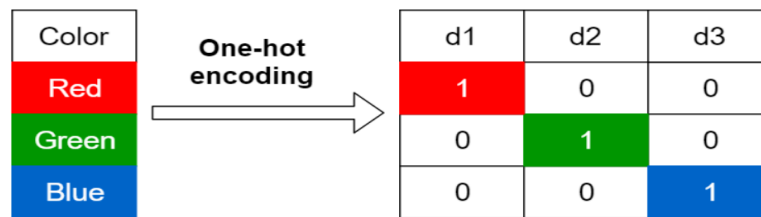


```
housing_cat_encoded[:200]
[0.],
[3.],
[1.],
[1.],
[4.],
[0.],
[4.],
[3.],
[4.],
[0.],
[4.],
[0.],
[1.],
[0.],
[3.],
[0.],
[1.]
```

- One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as “bad,” “average,” “good,” and “excellent”), but it is obviously not the case for the `ocean_proximity` column (for example, categories 0 and 4 are clearly more similar than categories 0 and 1)

### 2.1.2. Working of One-Hot encoder :

- To fix the above issue, a common solution 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), while the others will be 0 (cold).
- The new attributes are sometimes called dummy attributes.
- Scikit-Learn provides a `OneHotEncoder` class to convert categorical values into one-hot vectors:



```
[37] from sklearn.preprocessing import OneHotEncoder
```

```
cat_encoder=OneHotEncoder()  
housing_cat_1hot=cat_encoder.fit_transform(housing_categorical)  
housing_cat_1hot
```

```
<20640x5 sparse matrix of type '<class 'numpy.float64'>'  
with 20640 stored elements in Compressed Sparse Row format>
```

```
[39] housing_cat_1hot.toarray()
```

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

```
cat_encoder.categories_
```

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

- By default, the output of a OneHotEncoder is a SciPy sparse matrix, instead of a NumPy array:
- A sparse matrix is a very efficient representation for matrices that contain mostly zeroes.
- Indeed, internally it only stores the non-zero values and their positions.
- When a categorical attribute has hundreds or thousands of categories, then one-hot-encoding it results in a very large matrix full of zeros except for a single 1 per row.
- In this case, a sparse matrix is exactly what we need: it will save plenty of memory and speed up computations.
- You can use a sparse matrix mostly like a normal 2D array, but if you want to convert it to a (dense) NumPy array, just call the **toarray()** method



- Pandas has a function called `get_dummies()` which also converts each categorical feature into a one-hot-representation, with one binary feature per category:

```
df_test = pd.DataFrame({"ocean_proximity": ["<1H OCEAN", "INLAND", "ISLAND", "NEAR BAY", "NEAR OCEAN"]})
pd.get_dummies(df_test)
```

	ocean_proximity_<1H OCEAN	ocean_proximity_INLAND	ocean_proximity_ISLAND	ocean_proximity_NEAR BAY	ocean_proximity_NEAR OCEAN
0	True	False	False	False	False
1	False	True	False	False	False
2	False	False	True	False	False
3	False	False	False	True	False
4	False	False	False	False	True

- It looks nice and simple, so why not use it instead of `OneHotEncoder`?
- Well, the advantage of `OneHotEncoder` is that it remembers which categories it was trained on.
- This is very important because once your model is in production, it should be fed exactly the same features as during training: no more, no less.
- Watch what our trained `cat_encoder` outputs when we make it transform the same `df_test` (using **`transform()`**, not `fit_transform()`):

```
cat_encoder.transform(df_test).toarray()
```

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

- See the difference? `get_dummies()` saw only two categories, so it output two columns, whereas `OneHotEncoder` output one column per learned category, in the right order. Moreover, if you feed `get_dummies()` a `DataFrame` containing an unknown category (e.g., "<2H OCEAN"), it will happily generate a column for it.
- But `OneHotEncoder` is smarter: it will detect the unknown category and raise an exception. If you prefer, you can set the `handle_unknown` hyperparameter to "ignore", in which case it will just represent the unknown category with zeros:

```

df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN", "ISLAND"]})
pd.get_dummies(df_test_unknown)
cat_encoder.handle_unknown = "ignore"
cat_encoder.transform(df_test_unknown).toarray()

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

```

### 2.1.3. Working of Label encoder :

- A technique that is used to convert categorical columns into numerical ones so that they can be fitted by machine learning models which only take numerical data.
- Similar to Ordinal encoding but works on nominal data instead
- It is usually implemented on the Y variable instead of the input X variable.

```

[18] from sklearn.preprocessing import LabelEncoder
label_encode=LabelEncoder()
housing_cat_label=label_encode.fit_transform(housing_categorical)
print(housing_cat_label)

```

```

[3 3 3 ... 1 1 1]

```

Or

```

from sklearn.preprocessing import LabelEncoder
label_encode = LabelEncoder()
# If housing_categorical is a DataFrame or a 2D array, convert it to a 1D array
housing_categorical = housing_categorical.values.ravel() if hasattr(housing_categorical, 'values') else housing_categorical.ravel()
housing_cat_label = label_encode.fit_transform(housing_categorical)
print(housing_cat_label.tolist())

```

```

, 0, 4, 0, 0, 1, 0, 4, 1, 1, 1, 1, 4, 0, 1, 0, 1, 1, 1, 0, 1, 4, 0, 0, 1, 4, 1, 3, 0, 4, 4, 0, 3, 3, 0, 1, 3, 1, 3, 0, 4, 0, 0, 1, 3, 1,

```

## 3. Outliers:

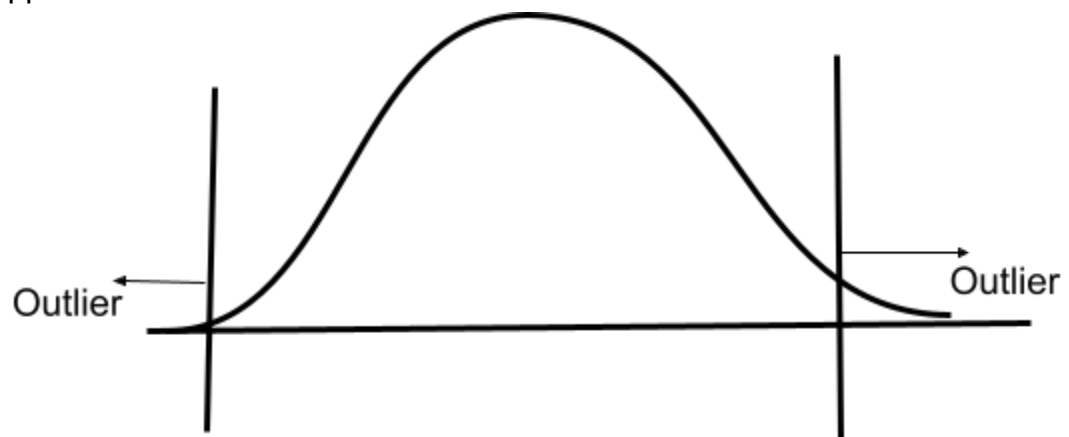
- An *outlier* is an observation that lies an abnormal distance from other values in a random sample from a population. In a sense, **this definition leaves it up to the analyst (or a consensus process) to decide what will be considered abnormal.**

### 3.1 When are Outliers Dangerous?

- They impact statistical measures like mean or variance
- They can affect the strength and direction of correlation between variables, potentially leading to misleading interpretations
- Create a learning bias in the model. Specially those models that deal with weights like linear and logistic regression, Adaboost and Deep learning

### 3.2 Treatment:

- **Trimming:** Remove the detected outliers from both the ends of the data distribution.
- **Capping:**
  - a. Decide on the thresholds that define the acceptable range for your data. Common choices are based on percentiles, such as the 1st and 99th percentiles, or the 5th and 95th percentiles.
  - b. Find the data points that fall outside these thresholds. These are considered extreme values.
  - c. **Lower Bound:** Replace any value below the lower threshold with the lower threshold value.
  - d. **Upper Bound:** Replace any value above the upper threshold with the upper threshold value.



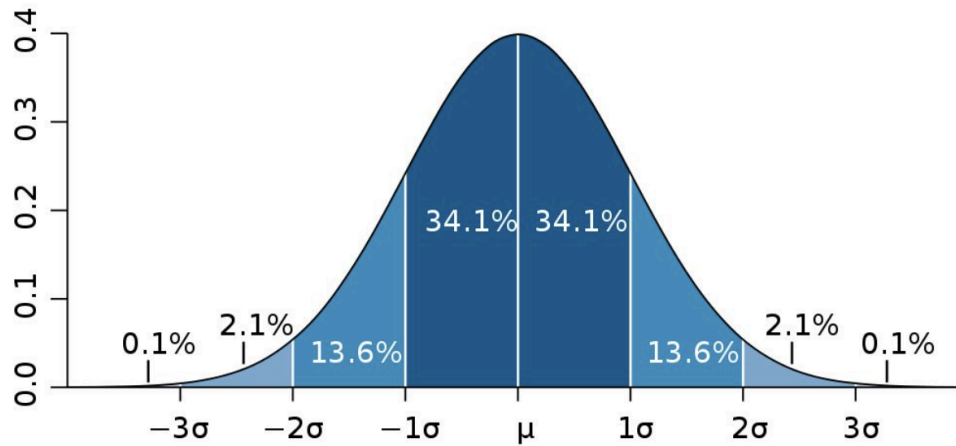
- e.
- **Discretization:**

Divide the range of the continuous variable into equal-width intervals. Each interval represents a discrete category.

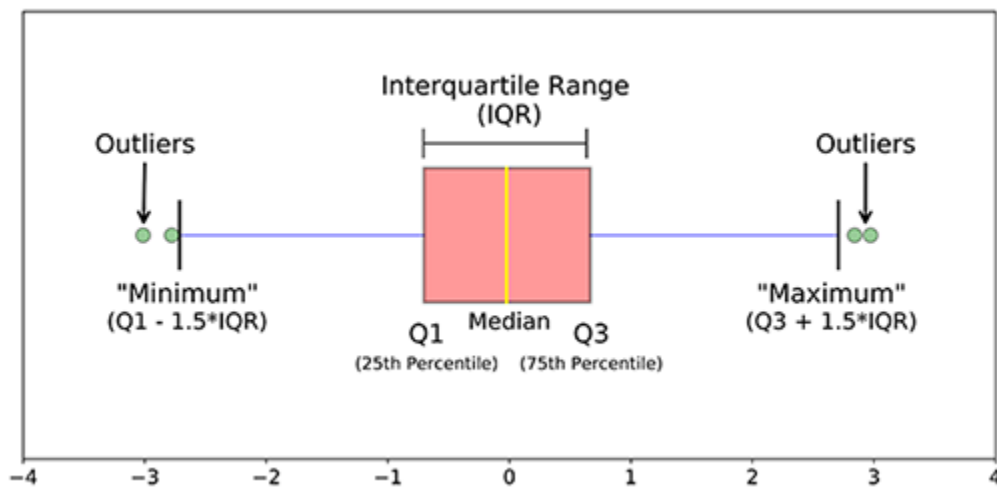
Example: For a variable ranging from 0 to 100, create bins of width 10 (0-10, 10-20, 20-30, etc.).

### 3.2 How to detect Outliers:

- **Normal Distribution:**  
**Z-score:**



- **Skewed Distribution:**  
**IQR:**



**Task: Download a dataset and implement the following steps on it**

1. Get basic insights on the data set including its features
2. Visualize the data set and overall data distribution
3. Clean the data using different cleaning techniques
4. Transform the data using different transformation techniques and justify why you chose the technique.

## References:

[1]. Hands On Machine Learning with Scikit Learn, Keras and TensorFlow pg.147-pg.156

[2].

[https://www.youtube.com/watch?v=LIn1PKgGr\\_M&list=PLKnIA16\\_Rmvbr7zKYQuBfsVkjoLcJgxHH&index=41](https://www.youtube.com/watch?v=LIn1PKgGr_M&list=PLKnIA16_Rmvbr7zKYQuBfsVkjoLcJgxHH&index=41)