Machine Learning Practical

Data Cleaning in ML

What is Data Cleaning?

Data cleaning is a crucial step in the machine learning (ML) pipeline, as it involves identifying and removing any missing, duplicate, or irrelevant data. The goal of data cleaning is to ensure that the data is accurate, consistent, and free of errors, as incorrect or inconsistent data can negatively impact the performance of the ML model. Professional data scientists usually invest a very large portion of their time in this step because of the belief that **"Better data beats fancier algorithms"**.

Data cleaning, also known as **data cleansing** or **data preprocessing**, is a crucial step in the data science pipeline that involves identifying and correcting or removing errors, inconsistencies, and inaccuracies in the data to improve its quality and usability. Data cleaning is essential because raw data is often noisy, incomplete, and inconsistent, which can negatively impact the accuracy and reliability of the insights derived from it.

Data Cleaning

- **Removal of Unwanted Observations**: Identify and eliminate irrelevant or redundant observations from the dataset. The step involves scrutinizing data entries for duplicate records, irrelevant information, or data points that do not contribute meaningfully to the analysis. Removing unwanted observations streamlines the dataset, reducing noise and improving the overall quality.
- **Fixing Structure errors:** Address structural issues in the dataset, such as inconsistencies in data formats, naming conventions, or variable types. Standardize formats, correct naming discrepancies, and ensure uniformity in data representation. Fixing structure errors enhances data consistency and facilitates accurate analysis and interpretation.

- **Managing Unwanted outliers:** Identify and manage outliers, which are data points significantly deviating from the norm. Depending on the context, decide whether to remove outliers or transform them to minimize their impact on analysis. Managing outliers is crucial for obtaining more accurate and reliable insights from the data.
- **Handling Missing Data:** Devise strategies to handle missing data effectively. This may involve imputing missing values based on statistical methods, removing records with missing values, or employing advanced imputation techniques. Handling missing data ensures a more complete dataset, preventing biases and maintaining the integrity of analyses.

```
import pandas as pd
import numpy as np

# Load the dataset
df = pd.read_csv('titanic.csv')
df.head()
```

Output:

```
PassengerId    Survived    Pclass    Name    Sex    Age    SibSp
Parch    Ticket    Fare    Cabin    Embarked
0    1    0    3    Braund, Mr. Owen Harris    male    22.0    1
0    A/5 21171    7.2500    NaN    S
1    2    1    1    Cumings, Mrs. John Bradley (Florence Briggs
Th...    female    38.0    1    0    PC 17599    71.2833    C85    C
2    3    1    3    Heikkinen, Miss. Laina    female    26.0    0
0    STON/O2. 3101282    7.9250    NaN    S
3    4    1    1    Futrelle, Mrs. Jacques Heath (Lily May Peel)
female    35.0    1    0    113803    53.1000    C123    S
4    5    0    3    Allen, Mr. William Henry    male    35.0    0
0    373450    8.0500    NaN    S
```

Data Inspection and Exploration

Output:

```
0      False
1      False
2      False
3      False
4      False
       ...
886    False
887    False
888    False
889    False
890    False
Length: 891, dtype: bool
```

Check the data information using df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

From the above data info, we can see that Age and Cabin have an **unequal number of counts**. And some of the columns are categorical and have data type objects and some are integer and float values.

```
# Categorical columns
cat_col = [col for col in df.columns if df[col].dtype == 'object']
print('Categorical columns :',cat_col)
# Numerical columns
num_col = [col for col in df.columns if df[col].dtype != 'object']
print('Numerical columns :',num_col)
```

Output:

```
Categorical columns : ['Name', 'Sex', 'Ticket', 'Cabin', 'Embarked']
Numerical columns : ['PassengerId', 'Survived', 'Pclass', 'Age',
'SibSp', 'Parch', 'Fare']
```

Check the total number of Unique Values in the Categorical Columns

```
df[cat_col].nunique()
```

Output:

```
Name        891
Sex           2
Ticket      681
Cabin       147
Embarked      3
dtype: int64
```

Steps to Perform Data Cleansing

Removal of all Above Unwanted Observations

This includes deleting duplicate/ redundant or irrelevant values from your dataset. Duplicate observations most frequently arise during data collection and Irrelevant observations are those that don't actually fit the specific problem that you're trying to solve.

- Redundant observations alter the efficiency to a great extent as the data repeats and may add towards the correct side or towards the incorrect side, thereby producing unfaithful results.
- Irrelevant observations are any type of data that is of no use to us and can be removed directly.

Now we have to make a decision according to the subject of analysis, which factor is important for our discussion.

As we know our machines don't understand the text data. So, we have to either drop or convert the categorical column values into numerical types. Here we are dropping the Name columns because the Name will be always unique and it hasn't a great influence on target variables. For the ticket, Let's first print the 50 unique tickets.

```
df['Ticket'].unique()[:50]
```

Output:

```
array(['A/5 21171', 'PC 17599', 'STON/O2. 3101282', '113803',
'373450',
       '330877', '17463', '349909', '347742', '237736', 'PP 9549',
       '113783', 'A/5. 2151', '347082', '350406', '248706',
'382652',
       '244373', '345763', '2649', '239865', '248698', '330923',
'113788',
       '347077', '2631', '19950', '330959', '349216', 'PC 17601',
       'PC 17569', '335677', 'C.A. 24579', 'PC 17604', '113789',
'2677',
       'A./5. 2152', '345764', '2651', '7546', '11668', '349253',
       'SC/Paris 2123', '330958', 'S.C./A.4. 23567', '370371',
'14311',
       '2662', '349237', '3101295'], dtype=object)
```

From the above tickets, we can observe that it is made of two like first values 'A/5 21171' is joint from of 'A/5' and '21171' this may influence our target variables. It will the case of **Feature Engineering**. where we derived new features from a column or a group of columns. In the current case, we are dropping the "Name" and "Ticket" columns.

Drop Name and Ticket Columns

```
df1 = df.drop(columns=['Name','Ticket'])
df1.shape
```

Output:

```
(891, 10)
```

Handling Missing Data

Missing data is a common issue in real-world datasets, and it can occur due to various reasons such as human errors, system failures, or data collection issues. Various techniques can be used to handle missing data, such as imputation, deletion, or substitution.

Let's check the % missing values columns-wise for each row using df.isnull() it checks whether the values are null or not and gives returns boolean values. and .sum() will sum the total number of null values rows and we divide it by the total number of rows present in the dataset then we multiply to get values in % i.e per 100 values how much values are null.

```
round((df1.isnull().sum()/df1.shape[0])*100,2)
```

Output:

```
PassengerId     0.00
Survived        0.00
Pclass          0.00
Sex             0.00
Age            19.87
SibSp           0.00
Parch           0.00
Fare            0.00
Cabin          77.10
Embarked        0.22
dtype: float64
```

We cannot just ignore or remove the missing observation. They must be handled carefully as they can be an indication of something important.

The two most common ways to deal with missing data are:

- **Dropping Observations with missing values**.
    - The fact that the value was missing may be informative in itself.
    - Plus, in the real world, you often need to make predictions on new data even if some of the features are missing!

As we can see from the above result that Cabin has 77% null values and Age has 19.87% and Embarked has 0.22% of null values.

So, it's not a good idea to fill 77% of null values. So, we will drop the Cabin column. Embarked column has only 0.22% of null values so, we drop the null values rows of Embarked column.

```
df2 = df1.drop(columns='Cabin')
df2.dropna(subset=['Embarked'], axis=0, inplace=True)
df2.shape
```

Output:

```
(889, 9)
```

- **Imputing the missing values from past observations.**
    - Again, "missingness" is almost always informative in itself, and you should tell your algorithm if a value was missing.
    - Even if you build a model to impute your values, you're not adding any real information. You're just reinforcing the patterns already provided by other features.

We can use **Mean imputation** or **Median imputations** for the case.

Note:

- Mean imputation is suitable when the data is normally distributed and has no extreme outliers.
- Median imputation is preferable when the data contains outliers or is skewed.

```
# Mean imputation
df3 = df2.fillna(df2.Age.mean())
# Let's check the null values again
df3.isnull().sum()
```

Output:

```
PassengerId    0
Survived       0
Pclass         0
Sex            0
Age            0
SibSp          0
Parch          0
Fare           0
Embarked       0
dtype: int64
```

Handling Outliers

Outliers are extreme values that deviate significantly from the majority of the data. They can negatively impact the analysis and model performance. Techniques such as clustering, interpolation, or transformation can be used to handle outliers.

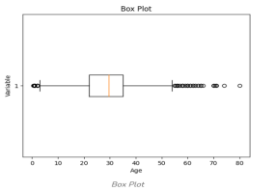
To check the outliers, We generally use a box plot. A box plot, also referred to as a box-and-whisker plot, is a graphical representation of a dataset's distribution. It shows a variable's median, quartiles, and potential outliers. The line inside the box denotes the median, while the box itself denotes the interquartile range (IQR). The whiskers extend to the most extreme non-outlier values within 1.5 times the IQR. Individual points beyond the whiskers are considered potential outliers. A box plot offers an easy-to-understand overview of the range of the data and makes it possible to identify outliers or skewness in the distribution.

Let's plot the box plot for Age column data.

```
import matplotlib.pyplot as plt

plt.boxplot(df3['Age'], vert=False)
plt.ylabel('Variable')
plt.xlabel('Age')
plt.title('Box Plot')
plt.show()
```

Output:



Box Plot

As we can see from the above Box and whisker plot, Our age dataset has outliers values. The values less than 5 and more than 55 are outliers.

```
# calculate summary statistics
mean = df3['Age'].mean()
std  = df3['Age'].std()

# Calculate the lower and upper bounds
lower_bound = mean - std*2
upper_bound = mean + std*2

print('Lower Bound :',lower_bound)
print('Upper Bound :',upper_bound)

# Drop the outliers
df4 = df3[(df3['Age'] >= lower_bound)
                & (df3['Age'] <= upper_bound)]
```

Output:

```
Lower Bound : 3.705400107925648
Upper Bound : 55.578785285332785
```

Similarly, we can remove the outliers of the remaining columns.

Data Transformation

Data transformation involves converting the data from one form to another to make it more suitable for analysis. Techniques such as normalization, scaling, or encoding can be used to transform the data.

Data validation and verification

Data validation and verification involve ensuring that the data is accurate and consistent by comparing it with external sources or expert knowledge.

For the machine learning prediction, First, we separate independent and target features. Here we will consider only **'Sex' 'Age' 'SibSp', 'Parch' 'Fare' 'Embarked'** only as the independent features and **Survived** as target variables. Because PassengerId will not affect the survival rate.

```
X = df3[['Pclass','Sex','Age', 'SibSp','Parch','Fare','Embarked']]
Y = df3['Survived']
```

Data formatting

Data formatting involves converting the data into a standard format or structure that can be easily processed by the algorithms or models used for analysis. Here we will discuss commonly used data formatting techniques i.e. Scaling and Normalization.

Scaling

- Scaling involves transforming the values of features to a specific range. It maintains the shape of the original distribution while changing the scale.
- Particularly useful when features have different scales, and certain algorithms are sensitive to the magnitude of the features.
- Common scaling methods include Min-Max scaling and Standardization (Z-score scaling).

Min-Max Scaling: Min-Max scaling rescales the values to a specified range, typically between 0 and 1. It preserves the original distribution and ensures that the minimum value maps to 0 and the maximum value maps to 1.

```
from sklearn.preprocessing import MinMaxScaler

# initialising the MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))

# Numerical columns
num_col_ = [col for col in X.columns if X[col].dtype != 'object']
x1 = X
# learning the statistical parameters for each of the data and
transforming
x1[num_col_] = scaler.fit_transform(x1[num_col_])
x1.head()
```

Output:

	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	1.0	male	0.271174	0.125	0.0	0.014151	S
1	0.0	female	0.472229	0.125	0.0	0.139136	C
2	1.0	female	0.321438	0.000	0.0	0.015469	S
3	0.0	female	0.434531	0.125	0.0	0.103644	S
4	1.0	male	0.434531	0.000	0.0	0.015713	S

Standardization (Z-score scaling): Standardization transforms the values to have a mean of 0 and a standard deviation of 1. It centers the data around the mean and scales it based on the standard deviation. Standardization makes the data more suitable for algorithms that assume a Gaussian distribution or require features to have zero mean and unit variance.

```
z = (x - μ) / σ
```

Where,

- X = Data
- μ = Mean value of X
- σ = Standard deviation of X

Data Cleansing Tools

Some data cleansing tools:

- OpenRefine
- Trifacta Wrangler
- TIBCO Clarity
- Cloudingo
- IBM Infosphere Quality Stage

Advantages of Data Cleaning in Machine Learning:

- **Improved model performance**: Removal of errors, inconsistencies, and irrelevant data, helps the model to better learn from the data.
- **Increased accuracy**: Helps ensure that the data is accurate, consistent, and free of errors.
- Better representation of the data: Data cleaning allows the data to be transformed into a format that better represents the underlying relationships and patterns in the data.
- Improved data quality: Improve the quality of the data, making it more reliable and accurate.
- Improved data security: Helps to identify and remove sensitive or confidential information that could compromise data security.

Disadvantages of Data Cleaning in Machine Learning

- Time-consuming: Time-Consuming task, especially for large and complex datasets.
- Error-prone: Data cleaning can be error-prone, as it involves transforming and cleaning the data, which can result in the loss of important information or the introduction of new errors.
- Cost and resource-intensive: Resource-intensive process that requires significant time, effort, and expertise. It can also require the use of specialized software tools, which can add to the cost and complexity of data cleaning.
- Overfitting: Data cleaning can inadvertently contribute to overfitting by removing too much data.

What does it mean to cleanse our data?

Cleansing data involves identifying and rectifying errors, inconsistencies, and inaccuracies in a dataset to improve its quality, ensuring reliable results in analyses and decision-making.

What is an example of cleaning data?

Removing duplicate records in a customer database ensures accurate and unbiased analysis, preventing redundant information from skewing results or misrepresenting the customer base.

What is the meaning of data wash?

"Data wash" is not a standard term in data management. If used, it could refer to cleaning or processing data, but it's not a widely recognized term in the field.

How is data cleansing done?

Data cleansing involves steps like removing duplicates, handling missing values, and correcting inconsistencies. It requires systematic examination and correction of data issues.

What is data cleansing in cyber security?

In cybersecurity, data cleansing involves identifying and removing malicious code or unauthorized access points from datasets to protect sensitive information and prevent cyber threats.

How to clean data using SQL?

Use SQL commands like `DELETE` for removing duplicates, `UPDATE` for correcting values, and `ALTER TABLE` for modifying data structures. Employ `WHERE` clauses to target specific records for cleaning.

Linear Regression Example steps

Suppose we have the following dataset that shows the weight and height of seven individuals:

Weight (lbs)	Height (inches)
140	60
155	62
159	67
179	70
192	71
200	72
212	75

Use the following steps to fit a linear regression model to this dataset, using weight as the predictor variable and height as the response variable.

Step 1: Calculate X*Y, X^2, and Y^2

Weight (lbs)	Height (inches)	X*Y	X^2	Y^2
140	60	8400	19600	3600
155	62	9610	24025	3844
159	67	10653	25281	4489
179	70	12530	32041	4900
192	71	13632	36864	5041
200	72	14400	40000	5184
212	75	15900	44944	5625

Step 2: Calculate ΣX, ΣY, ΣX*Y, ΣX^2, and ΣY^2

	Weight (lbs)	Height (inches)	X*Y	X^2	Y^2
	140	60	8400	19600	3600
	155	62	9610	24025	3844
	159	67	10653	25281	4489
	179	70	12530	32041	4900
	192	71	13632	36864	5041
	200	72	14400	40000	5184
	212	75	15900	44944	5625
Σ	**1237**	**477**	**85125**	**222755**	**32683**

Step 3: Calculate b_0

The formula to calculate b_0 is: $[(\Sigma Y)(\Sigma X^2) - (\Sigma X)(\Sigma XY)] \ / \ [n(\Sigma X^2) - (\Sigma X)^2]$

In this example, $b_0 = [(477)(222755) - (1237)(85125)] \ / \ [7(222755) - (1237)^2] =$ **32.783**

Step 4: Calculate b_1

The formula to calculate b_1 is: $[n(\Sigma XY) - (\Sigma X)(\Sigma Y)] \ / \ [n(\Sigma X^2) - (\Sigma X)^2]$

In this example, $b_1 = [7(85125) - (1237)(477)] \ / \ [7(222755) - (1237)^2]$ = **0.2001**

Step 5: Place b_0 and b_1 in the estimated linear regression equation.

The estimated linear regression equation is: $\hat{y} = b_0 + b_1 x$

In our example, it is $\hat{y} = 0.32783 + (0.2001)x$

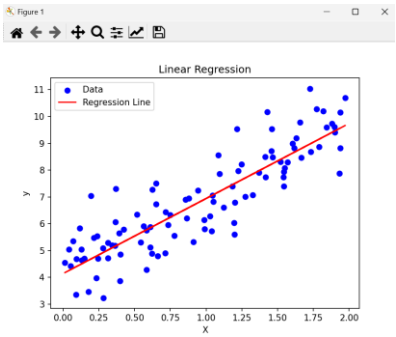
How to Interpret a Simple Linear Regression Equation

Here is how to interpret this estimated linear regression equation: $\hat{y} = 32.783 + 0.2001x$

b_0 = 32.7830. When weight is zero pounds, the predicted height is 32.783 inches. Sometimes the value for b_0 can be useful to know, but in this example it doesn't actually make sense to interpret b_0 since a person can't weigh zero pounds.

b_1 = 0.2001. A one pound increase in weight is associated with a 0.2001 inch increase in height.

```
Slope (m): 2.7993236574802762
Intercept (b): 4.142913319458566
Mean Squared Error: 0.6536995137170021
R² Score: 0.8072059636181392
```




```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Generate dataset
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Model parameters
print(f"Slope (m): {model.coef_[0][0]}")
print(f"Intercept (b): {model.intercept_[0]}")

# Predict and evaluate
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error: {mse}")
print(f"R² Score: {r2}")

# Visualization
plt.scatter(X, y, color='blue', label='Data')
plt.plot(X, model.predict(X), color='red', label='Regression Line')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.title('Linear Regression')
plt.show()
```

[6]:
```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import fetch_california_housing

# Load California housing dataset
data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['Price'] = data.target
```

```
# Display the first few rows of the dataset
print(df.head())

# Select features (e.g., MedInc - Median Income) and target (Price)
X = df[['MedInc']]  # Using median income as the feature
y = df['Price']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)
```

```
# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")
print(f"R-squared: {r2:.2f}")

# Visualize the best-fit line
plt.figure(figsize=(8, 6))
plt.scatter(X_test, y_test, color='blue', label='Actual Prices')
plt.plot(X_test, y_pred, color='red', label='Predicted Prices')
plt.xlabel('Median Income (in $10,000s)')
plt.ylabel('House Price (in $100,000s)')
plt.title('Linear Regression: Predicting House Prices')
plt.legend()
plt.show()
```

```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude  Price
0    -122.23  4.526
1    -122.22  3.585
2    -122.24  3.521
3    -122.25  3.413
4    -122.25  3.422
Mean Squared Error: 0.71
R-squared: 0.46
```

What is gradient descent?

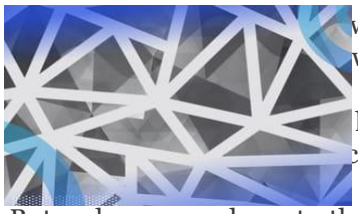
Gradient descent is an optimization algorithm often used to train machine learning models by locating the minimum values within a cost function.

Through this process, gradient descent minimizes the cost function and reduces the margin between predicted and actual results, improving a machine learning model's accuracy over time.

A gradient simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function.

The higher the gradient, the steeper the slope and the faster a model can learn.

But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.



who wants to climb
west steps possible.

hill by taking really
ction.

But as he comes closer to the top, his steps will get smaller and smaller to avoid overshooting it.

To use the **gradient descent algorithm** to minimize the cost function in a linear regression problem, you need to:

1. **Define the Hypothesis Function**: $h_\theta(x) = \theta_0 + \theta_1 x$, where θ_0 is the intercept and θ_1 is the slope.
2. **Define the Cost Function**: $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$, where m is the number of data points.
3. **Compute the Gradients**:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)$$

$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i) x_i$$

4. **Update the Parameters**:

$$\theta_0 := \theta_0 - \alpha \frac{\partial J(\theta)}{\partial \theta_0}$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial J(\theta)}{\partial \theta_1}$$

where α is the learning rate.

```
[7]: # Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing

# Load California housing dataset
data = fetch_california_housing()

# Create a DataFrame for convenience
df = pd.DataFrame(data.data, columns=data.feature_names)
df['MedHouseVal'] = data.target

# Use 'MedInc' (median income) as the feature and 'MedHouseVal' (house value) as the target
X = df['MedInc'].values  # Feature
y = df['MedHouseVal'].values.reshape(-1, 1)  # Target

# Normalize the feature for better convergence
X = (X - np.mean(X)) / np.std(X)

# Add an intercept term (bias)
X_b = np.c_[np.ones((X.shape[0], 1)), X]  # Add a column of ones to X

# Initialize parameters
theta = np.random.randn(2, 1)  # Two parameters: theta_0 (intercept), theta_1 (slope)
learning_rate = 0.01
iterations = 1000
```

```
# Define the cost function
def compute_cost(X, y, theta):
    m = len(y)
    predictions = X @ theta
    cost = (1 / (2 * m)) * np.sum((predictions - y) ** 2)
    return cost

# Gradient descent implementation
def gradient_descent(X, y, theta, learning_rate, iterations):
    m = len(y)
    cost_history = []

    for i in range(iterations):
        gradients = (1 / m) * X.T @ (X @ theta - y)  # Compute gradients
        theta -= learning_rate * gradients  # Update parameters
        cost_history.append(compute_cost(X, y, theta))  # Record cost

    return theta, cost_history
```

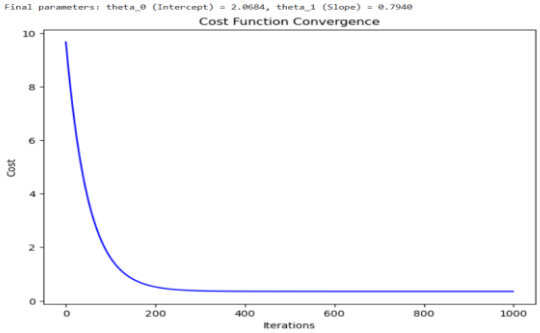
```
# Run gradient descent
theta_final, cost_history = gradient_descent(X_b, y, theta, learning_rate, iterations)

# Print the final parameters
print(f"Final parameters: theta_0 (Intercept) = {theta_final[0][0]:.4f}, theta_1 (Slope) = {theta_final[1][0]:.4f}")

# Plot the cost function convergence
plt.figure(figsize=(8, 6))
plt.plot(range(iterations), cost_history, color="blue")
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.title("Cost Function Convergence")
plt.show()

# Visualize the best-fit line
plt.figure(figsize=(8, 6))
plt.scatter(X, y, color="blue", label="Data Points")
plt.plot(X, X_b @ theta_final, color="red", label="Best Fit Line")
plt.xlabel("Median Income (Normalized)")
plt.ylabel("Median House Value")
plt.title("Linear Regression with Gradient Descent")
plt.legend()
plt.show()
```

```
Final parameters: theta_0 (Intercept) = 2.0684, theta_1 (Slope) = 0.7940
```



[8]:
```python
from sklearn.metrics import mean_squared_error, r2_score

# Predictions on the training set
y_pred = X_b @ theta_final

# Compute MSE
mse = mean_squared_error(y, y_pred)

# Compute R-squared
r2 = r2_score(y, y_pred)

# Print the evaluation metrics
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R-squared (R^2): {r2:.4f}")
```

```
Mean Squared Error (MSE): 0.7011
R-squared (R^2): 0.4734
```

In a simple linear regression model, the parameters θ_0 and θ_1 represent the coefficients of the linear equation:

$$h_\theta(x) = \theta_0 + \theta_1 \cdot x$$

Meanings of θ_0 and θ_1:

1. θ_0 **(Intercept)**:
   - This is the **y-intercept** of the regression line.
   - It represents the predicted value of the target (y) when the feature (x) is 0.
   - In the context of housing prices:
     - If $\theta_0 = 2.5$, this means when the median income ($x = 0$, after normalization), the predicted median house value is 2.5.
2. θ_1 **(Slope)**:
   - This is the **slope** of the regression line.
   - It represents the rate of change in the target (y) for a unit change in the feature (x).
   - In the context of housing prices:
     - If $\theta_1 = 0.85$, it means that for every 1-unit increase in normalized median income, the median house price increases by 0.85.

Intuition Behind θ_0 and θ_1:

Imagine you are trying to draw the best-fit line through your data points:

- θ_0 determines **where the line starts** on the y-axis.
- θ_1 determines **how steep or flat the line is**, based on how strongly x influences y.

Example:

For a simple equation:

$$\text{Predicted House Price} = \theta_0 + \theta_1 \cdot \text{Normalized Median Income}$$

- If $\theta_0 = 2.5$ and $\theta_1 = 0.85$:
  - A normalized median income of 1.0 predicts:

    $$\text{Price} = 2.5 + 0.85 \cdot 1.0 = 3.35$$
  - A normalized median income of 2.0 predicts:

    $$\text{Price} = 2.5 + 0.85 \cdot 2.0 = 4.2$$