**Loan approval data analysis**

1. **Check for missing values and outliers.**

**missing\_values = df.isnull().sum()**

This will return the count of missing values for each column in the DataFrame.

For Checking Outliers.

**Z-Score Method:**

The z-score method identifies outliers by calculating the number of standard deviations an observation deviates from the mean.

Values with z-scores greater than a certain threshold (e.g., 2 or 3) are considered outliers.

Example:

**from scipy.stats import zscore**

**z\_scores = zscore(df['column\_name'])**

**outliers = df[np.abs(z\_scores) > threshold]**

**IQR Method (Interquartile Range):**

The IQR method defines outliers as observations that fall below Q1 - 1.5 \* IQR or above Q3 + 1.5 \* IQR, where Q1 is the first quartile, Q3 is the third quartile, and IQR is the interquartile range.

**Q1 = df['column\_name'].quantile(0.25)**

**Q3 = df['column\_name'].quantile(0.75)**

**IQR = Q3 - Q1**

**outliers = df[(df['column\_name'] < Q1 - 1.5 \* IQR) | (df['column\_name'] > Q3 + 1.5 \* IQR)]**

**Tukey's Fences:**

Tukey's fences method is similar to the IQR method but uses different multiplier values (typically 1.5 or 3) to define the lower and upper fences.

Values outside the fences are considered outliers.

Example:

Q1 = df['column\_name'].quantile(0.25)

Q3 = df['column\_name'].quantile(0.75)

lower\_fence = Q1 - 1.5 \* (Q3 - Q1)

upper\_fence = Q3 + 1.5 \* (Q3 - Q1)

outliers = df[(df['column\_name'] < lower\_fence) | (df['column\_name'] > upper\_fence)]

**Modified Z-Score Method:**

The modified z-score method is a robust alternative to the standard z-score method that uses the median and median absolute deviation (MAD) instead of the mean and standard deviation.

Values with modified z-scores greater than a certain threshold (e.g., 2.5) are considered outliers.

Example:

median = df['column\_name'].median()

mad = (df['column\_name'] - median).abs().median()

modified\_z\_scores = 0.6745 \* (df['column\_name'] - median) / mad

outliers = df[np.abs(modified\_z\_scores) > threshold]

**Box Plot Visualization:**

Box plots visually represent the distribution of data and help identify outliers as individual data points beyond the whiskers.

Example:

**import seaborn as sns**

**sns.boxplot(x=df['column\_name'])**

These are some commonly used techniques to check for outliers in Python. The choice of method depends on the characteristics of your data and the specific analysis you are conducting. It's recommended to use a combination of statistical methods and visualizations to identify and handle outliers effectively.

1. **Imput the missing values .**

**Impute with Mean:**

This method replaces missing values with the mean value of the respective column.

df['column\_name'].fillna(df['column\_name'].mean(), inplace=True)

**Impute with Median:**

This method replaces missing values with the median value of the respective column.

df['column\_name'].fillna(df['column\_name'].median(), inplace=True)

**Impute with Mode:**

This method replaces missing values with the mode (most frequent value) of the respective column.

df['column\_name'].fillna(df['column\_name'].mode()[0], inplace=True)

**Impute with Constant Value:**

This method replaces missing values with a specific constant value.

df['column\_name'].fillna(value, inplace=True)

Replace value with the desired constant value.

**Forward Fill (ffill) or Backward Fill (bfill):**

These methods fill missing values with the last known non-missing value (forward fill) or the next known non-missing value (backward fill).

df['column\_name'].fillna(method='ffill', inplace=True) # Forward fill

df['column\_name'].fillna(method='bfill', inplace=True) # Backward fill

**Interpolation:**

Interpolation methods estimate missing values based on the surrounding data points.

Examples include linear interpolation (linear), polynomial interpolation (polynomial), and time-based interpolation (time).

df['column\_name'].interpolate(method='linear', inplace=True)

1. **Perform data normalization or standardization**

**Min-Max Normalization**:

Min-Max normalization scales the data to a fixed range between 0 and 1.

It preserves the relative distribution of the data and is suitable when you have a known minimum and maximum value for the feature.

Example:

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

normalized\_data = scaler.fit\_transform(df[['column\_name']])

**Standardization (Z-score normalization):**

Standardization transforms the data to have a mean of 0 and a standard deviation of 1.

It makes the data suitable for algorithms that assume normally distributed features or when comparing variables with different scales.

Example:

**from sklearn.preprocessing import StandardScaler**

**scaler = StandardScaler()**

**standardized\_data = scaler.fit\_transform(df[['column\_name']])**

Replace 'column\_name' with the actual column name in your DataFrame.

In both cases, the fit\_transform() method is used to compute the normalization or standardization parameters from the data and apply the transformation. The resulting normalized\_data or standardized\_data will be a NumPy array.

1. **Remove the Duplicates :-**

**Check for duplicate records:**

To identify duplicate records in your DataFrame, you can use the duplicated() method:

**duplicates = df.duplicated()**

This will return a boolean Series indicating whether each row is a duplicate or not.

**Remove duplicate records:**

To remove duplicate records from your DataFrame, you can use the drop\_duplicates() method:

**df\_no\_duplicates = df.drop\_duplicates()**

This will create a new DataFrame (df\_no\_duplicates) with the duplicate records removed.

If you want to modify the original DataFrame in place, you can use the inplace=True parameter:

**df.drop\_duplicates(inplace=True)**

By default, drop\_duplicates() considers all columns in the DataFrame. If you want to check for duplicates based on specific columns, you can pass a subset of columns to the method:

**df\_no\_duplicates = df.drop\_duplicates(subset=['column1', 'column2'])**

1. **Remove Specific Column :-**

**Using drop() function:**

Specify the column(s) you want to remove by providing their names in a list to the drop() function:

**df = df.drop(['column1', 'column2'], axis=1)**

Replace 'column1' and 'column2' with the actual column names you want to remove. The axis=1 parameter indicates that you are removing columns.

If you want to modify the DataFrame in-place, you can use the inplace=True parameter:

**df.drop(['column1', 'column2'], axis=1, inplace=True)**

**Using del keyword:**

You can use the del keyword to delete columns from the DataFrame directly:

**del df['column1']**

**del df['column2']**

1. **Handle categorical variables by encoding or creating dummy variables.**

**Label Encoding:**

Label encoding assigns a unique numeric label to each unique category in a categorical variable.

You can use the LabelEncoder class from the sklearn.preprocessing module:

**from sklearn.preprocessing import LabelEncoder**

**label\_encoder = LabelEncoder()**

**df['encoded\_column'] = label\_encoder.fit\_transform(df['categorical\_column'])**

Replace 'categorical\_column' with the actual column name containing the categorical variable, and 'encoded\_column' with the desired name for the encoded column in your DataFrame.

**One-Hot Encoding (Dummy Variables):**

One-hot encoding creates dummy variables for each unique category in a categorical variable.

You can use the get\_dummies() function from pandas to perform one-hot encoding:

**df\_encoded = pd.get\_dummies(df, columns=['categorical\_column'])**

Replace 'categorical\_column' with the actual column name containing the categorical variable. The get\_dummies() function will create new columns in the DataFrame, with each column representing a unique category and containing binary values (0 or 1) indicating the presence of that category.

If you want to drop one of the encoded columns to avoid multicollinearity, you can use the drop\_first parameter:

**df\_encoded = pd.get\_dummies(df, columns=['categorical\_column'], drop\_first=True)**

* **7.** **Address any inconsistencies or errors in data entry.**

**Standardizing values:**

In cases where there are variations in how data is entered (e.g., capitalization, abbreviations), you can standardize the values to a consistent format.

Use string methods, such as lower(), upper(), or title(), to convert values to lowercase, uppercase, or proper case, respectively.

**Correcting data errors:**

If you identify specific errors or inconsistencies, you can use string methods, regular expressions, or specific rules to correct them.

Examples include removing leading or trailing spaces, replacing or removing specific characters, or applying data transformation rules.

1. **Resolve any formatting issues in data.**

**Removing leading/trailing whitespaces:**

Leading or trailing whitespaces can cause issues when performing string operations or comparisons. You can use the strip() method to remove these whitespaces.

**df['column\_name'] = df['column\_name'].str.strip()**

**Converting data types:**

Sometimes, data may be stored in the wrong data type, leading to formatting issues. You can use the appropriate methods to convert data types to their desired formats.

**df['column\_name'] = df['column\_name'].astype(int) # Convert to integer**

**df['column\_name'] = df['column\_name'].astype(float) # Convert to float**

**df['column\_name'] = pd.to\_datetime(df['column\_name']) # Convert to datetime**

**Handling date formats:**

If you have date data with varying formats, you can use the to\_datetime() function from pandas to convert them into a standardized date format.

**df['column\_name'] = pd.to\_datetime(df['column\_name'], format='%Y-%m-%d') # Convert to 'YYYY-MM-DD' format**

**Formatting numerical values:**

You can use string formatting techniques to control the display of numerical values, such as setting the number of decimal places or adding thousand separators.

**df['column\_name'] = df['column\_name'].map('{:.2f}'.format) # Format to two decimal places**

**df['column\_name'] = df['column\_name'].map('{:,}'.format) # Format with thousand separators**

**Handling inconsistent capitalization:**

If you have text data with inconsistent capitalization, you can convert it to a standardized format using string methods like lower() or upper().

**df['column\_name'] = df['column\_name'].str.lower() # Convert to lowercase**

**df['column\_name'] = df['column\_name'].str.upper() # Convert to uppercase**

Parsing and extracting information from strings:

If you have strings with specific patterns and need to extract certain information, you can use regular expressions (re module) or string manipulation techniques to extract the desired data.

import re

**df['extracted\_value'] = df['column\_name'].str.extract(r'(\d+)') # Extract nume**

1. **Perform feature scaling if needed**

**Same as Min Max scalling and Standard Scaler.**