Memory updated

Handling Data Before Training – Everything You Need to Know

For an ML interview, you need to **fully understand** how to prepare raw data before feeding it into a model. This involves **data cleaning, transformation, feature engineering, and handling imbalances**. Below is a comprehensive breakdown.

---

1. Handling Missing Data

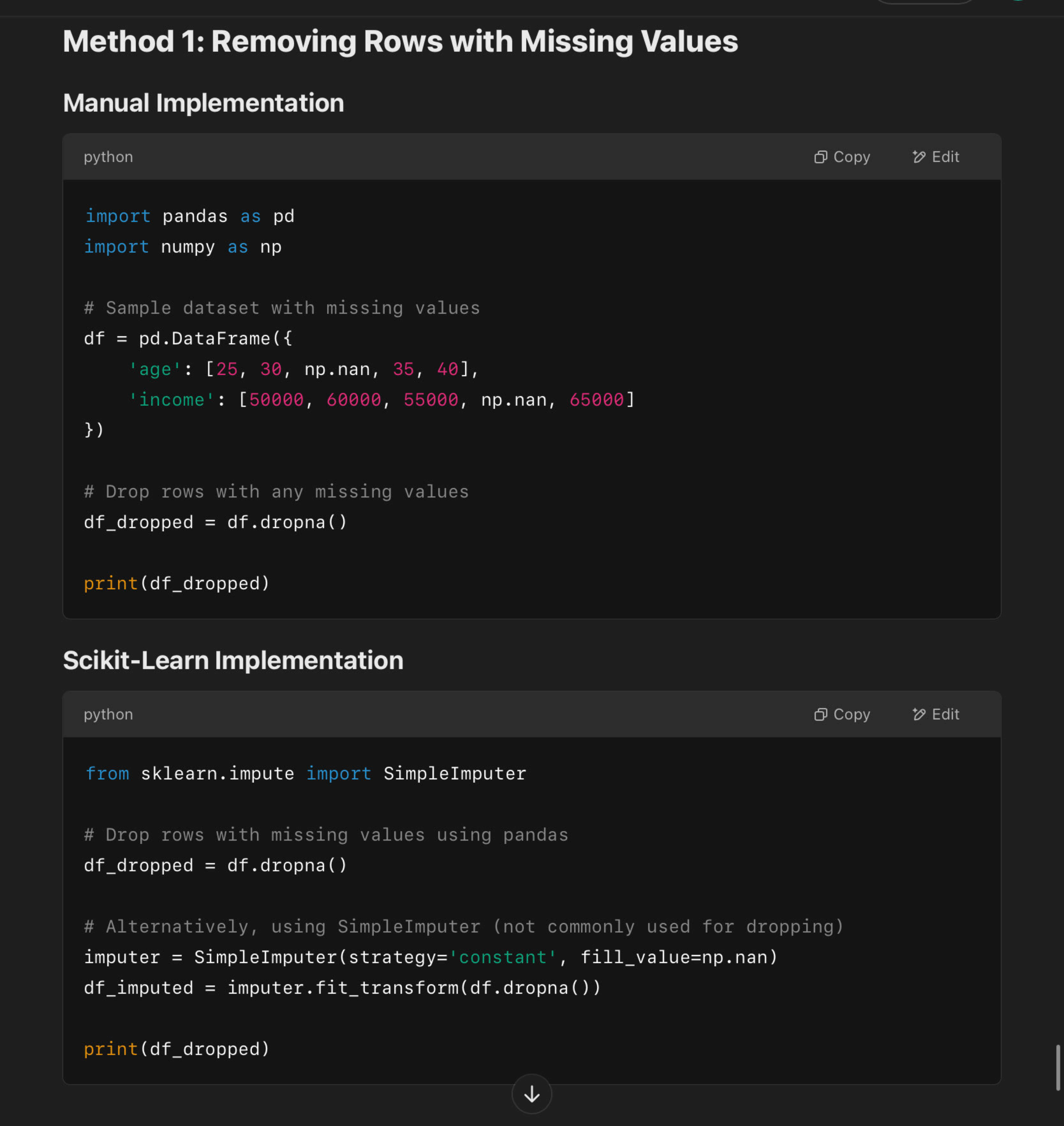
Missing data is common in real-world datasets and needs to be handled **before training**.

Types of Missing Data

- **MCAR (Missing Completely at Random)** → No pattern in missing values.
- **MAR (Missing at Random)** → Missingness depends on observed data.
- **MNAR (Missing Not at Random)** → Systematic missing pattern (e.g., people not reporting income).

Ways to Handle Missing Data

Method	When to Use	Implementation
Remove rows with missing values	When missing data is small (<5%)	`df.dropna()`
Remove columns with missing values	When an entire feature has many missing values (>50%)	`df.drop(columns=['feature'])`
Mean/Median/Mode Imputation	When missing values are MCAR and the feature is numerical	`df.fillna(df.mean())`
Forward/Backward Fill	When data has a sequential nature (e.g., time series)	`df.fillna(method='ffill')`
Predict Missing Values (KNN, Regression, etc.)	When missing values are MAR/MNAR and can be estimated using other features	`from sklearn.impute import KNNImputer`



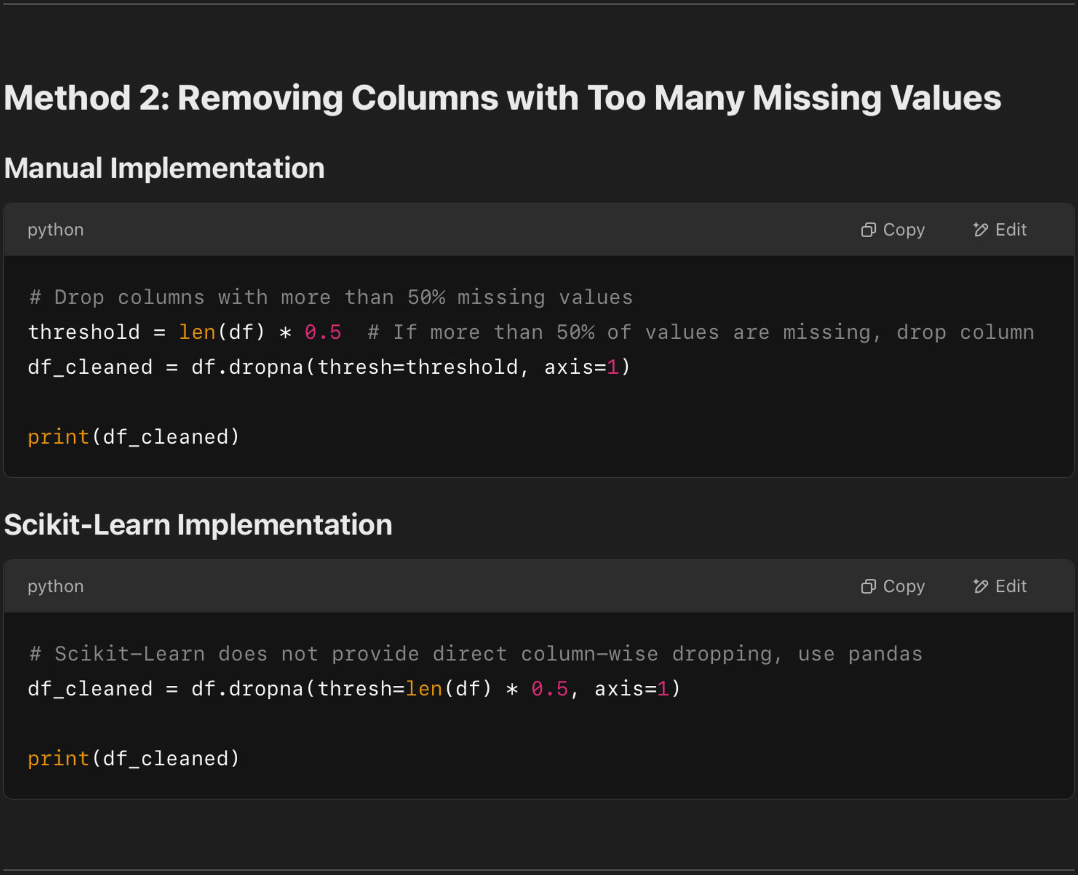
Method 1: Removing Rows with Missing Values

Manual Implementation

```python
import pandas as pd
import numpy as np

# Sample dataset with missing values
df = pd.DataFrame({
    'age': [25, 30, np.nan, 35, 40],
    'income': [50000, 60000, 55000, np.nan, 65000]
})

# Drop rows with any missing values
df_dropped = df.dropna()

print(df_dropped)
```

Scikit-Learn Implementation

```python
from sklearn.impute import SimpleImputer

# Drop rows with missing values using pandas
df_dropped = df.dropna()

# Alternatively, using SimpleImputer (not commonly used for dropping)
imputer = SimpleImputer(strategy='constant', fill_value=np.nan)
df_imputed = imputer.fit_transform(df.dropna())

print(df_dropped)
```



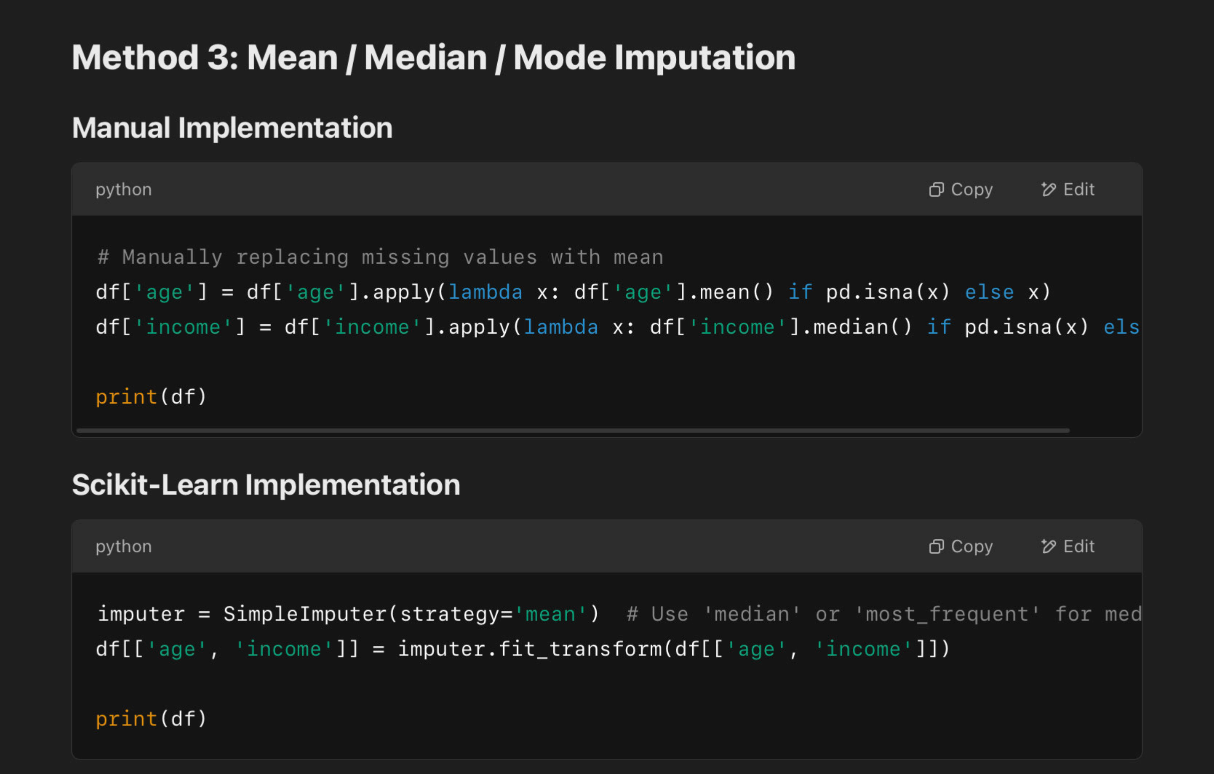
Method 2: Removing Columns with Too Many Missing Values

Manual Implementation

```python
# Drop columns with more than 50% missing values
threshold = len(df) * 0.5  # If more than 50% of values are missing, drop column
df_cleaned = df.dropna(thresh=threshold, axis=1)

print(df_cleaned)
```

Scikit-Learn Implementation

```python
# Scikit-Learn does not provide direct column-wise dropping, use pandas
df_cleaned = df.dropna(thresh=len(df) * 0.5, axis=1)

print(df_cleaned)
```



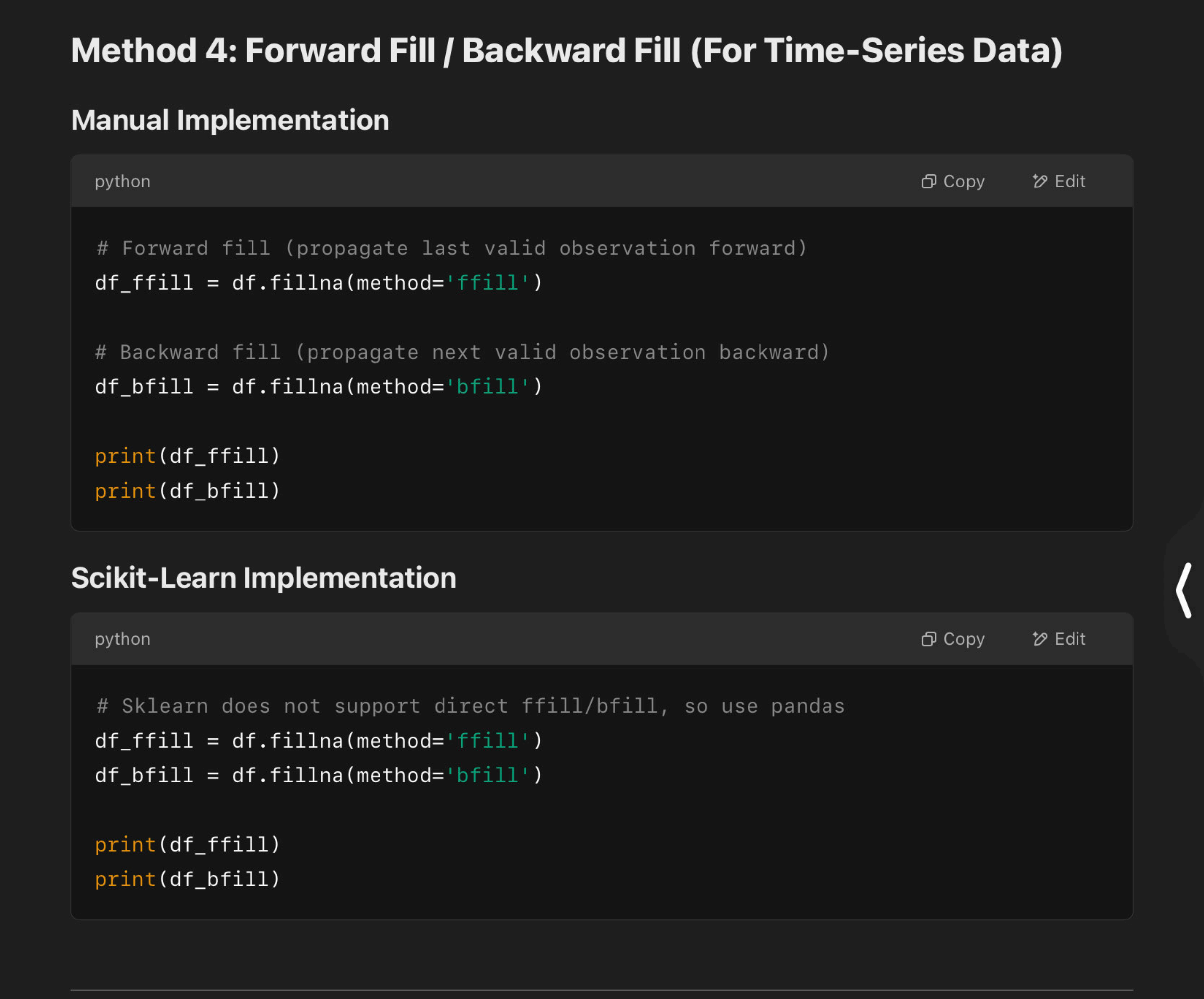
Method 3: Mean / Median / Mode Imputation

Manual Implementation

```python
# Manually replacing missing values with mean
df['age'] = df['age'].apply(lambda x: df['age'].mean() if pd.isna(x) else x)
df['income'] = df['income'].apply(lambda x: df['income'].median() if pd.isna(x) els

print(df)
```

Scikit-Learn Implementation

```python
imputer = SimpleImputer(strategy='mean')  # Use 'median' or 'most_frequent' for med
df[['age', 'income']] = imputer.fit_transform(df[['age', 'income']])

print(df)
```



Method 4: Forward Fill / Backward Fill (For Time-Series Data)

Manual Implementation

```python
# Forward fill (propagate last valid observation forward)
df_ffill = df.fillna(method='ffill')

# Backward fill (propagate next valid observation backward)
df_bfill = df.fillna(method='bfill')

print(df_ffill)
print(df_bfill)
```

Scikit-Learn Implementation

```python
# Sklearn does not support direct ffill/bfill, so use pandas
df_ffill = df.fillna(method='ffill')
df_bfill = df.fillna(method='bfill')

print(df_ffill)
print(df_bfill)
```

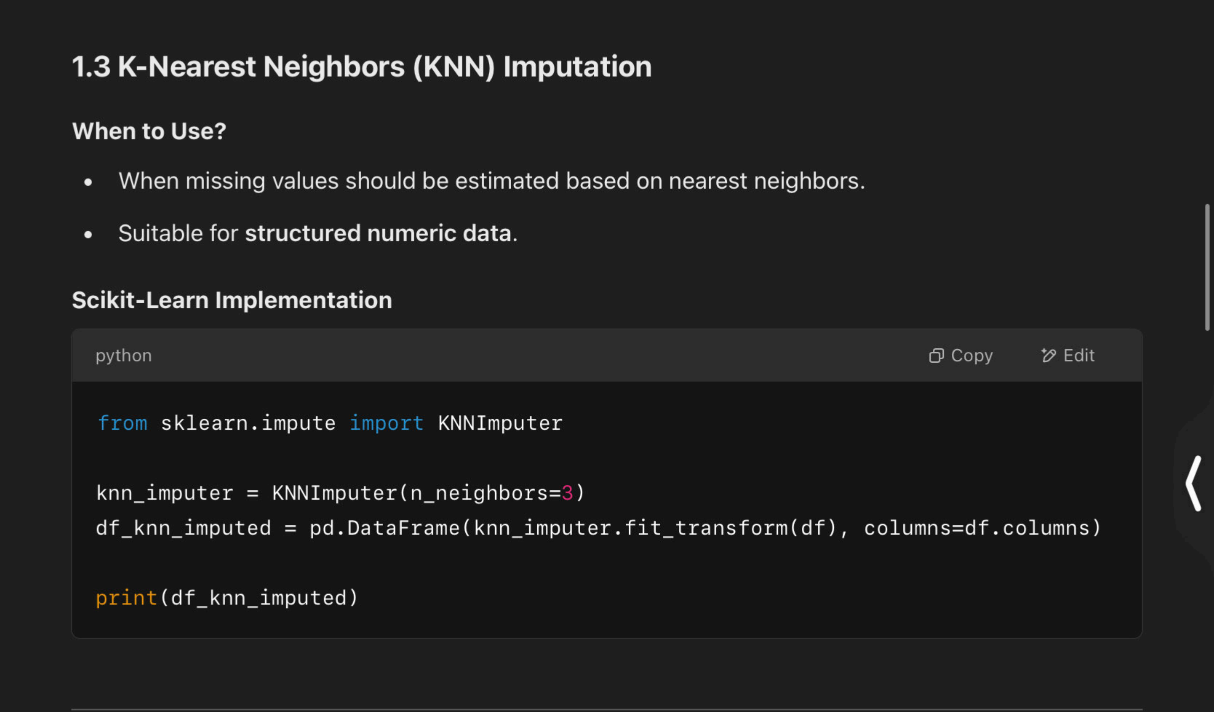
Method 5: Predicting Missing Values (Using KNN)

Manual Implementation

```python
from sklearn.neighbors import KNeighborsRegressor

# Example dataset
df_knn = pd.DataFrame({
    'feature1': [10, 20, 30, np.nan, 50],
    'feature2': [5, 10, 15, 20, 25]
})

# Separate rows with missing values
missing_rows = df_knn[df_knn['feature1'].isna()]
known_rows = df_knn[~df_knn['feature1'].isna()]

# Train KNN Regressor
knn = KNeighborsRegressor(n_neighbors=2)
knn.fit(known_rows[['feature2']], known_rows['feature1'])

# Predict missing values
df_knn.loc[df_knn['feature1'].isna(), 'feature1'] = knn.predict(missing_rows[['feat

print(df_knn)
```

Scikit-Learn Implementation

```python
from sklearn.impute import KNNImputer

knn_imputer = KNNImputer(n_neighbors=2)
df_knn_imputed = knn_imputer.fit_transform(df_knn)

print(df_knn_imputed)
```



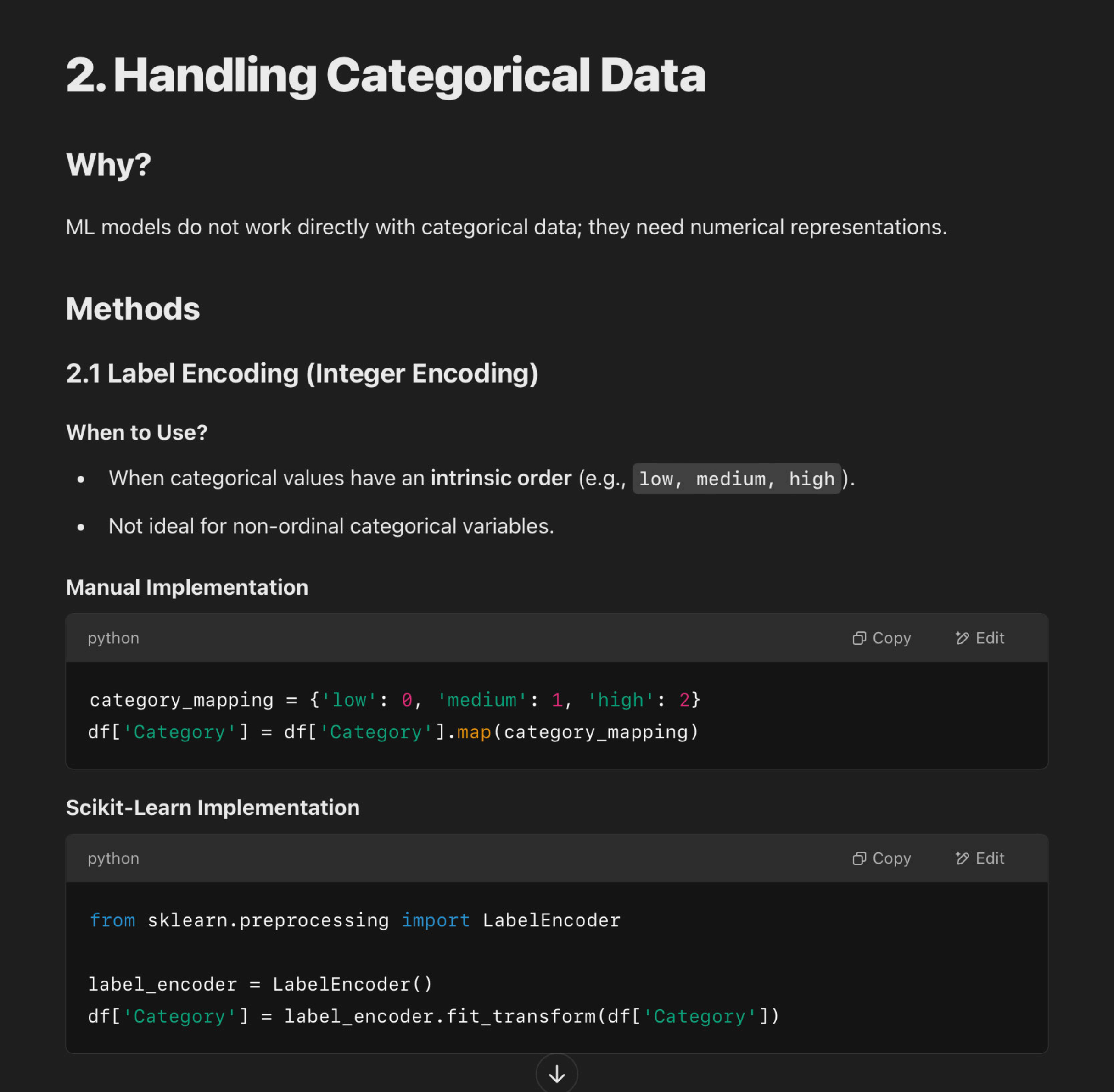
1.3 K-Nearest Neighbors (KNN) Imputation

When to Use?

- When missing values should be estimated based on nearest neighbors.
- Suitable for **structured numeric data**.

Scikit-Learn Implementation

```python
from sklearn.impute import KNNImputer

knn_imputer = KNNImputer(n_neighbors=3)
df_knn_imputed = pd.DataFrame(knn_imputer.fit_transform(df), columns=df.columns)

print(df_knn_imputed)
```



2. Handling Categorical Data

Why?

ML models do not work directly with categorical data; they need numerical representations.

Methods

2.1 Label Encoding (Integer Encoding)

When to Use?

- When categorical values have an **intrinsic order** (e.g., `low, medium, high`).
- Not ideal for non-ordinal categorical variables.

Manual Implementation

```python
category_mapping = {'low': 0, 'medium': 1, 'high': 2}
df['Category'] = df['Category'].map(category_mapping)
```

Scikit-Learn Implementation

```python
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
df['Category'] = label_encoder.fit_transform(df['Category'])
```

2.2 One-Hot Encoding

When to Use?

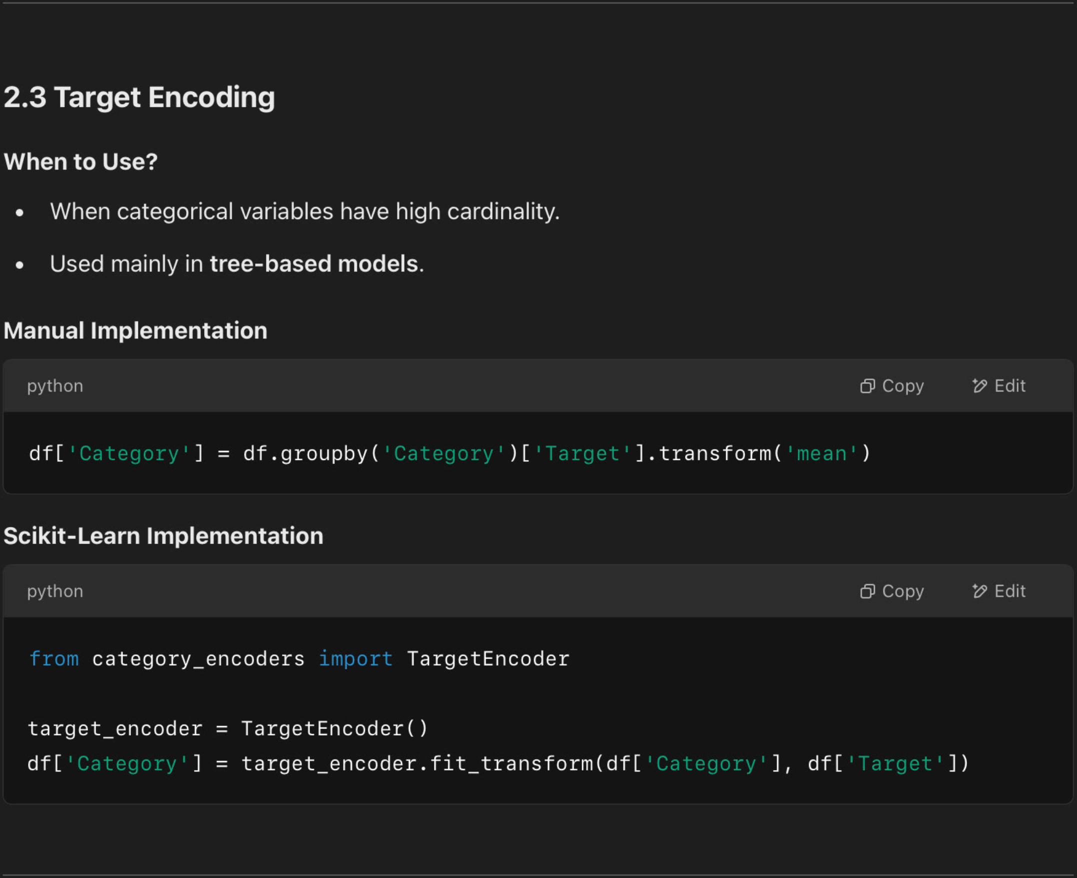
- When categorical values are **nominal (no order)**.
- Works well for **small cardinality categories** (not too many unique values).

Manual Implementation

```python
df = pd.get_dummies(df, columns=['Category'])
```

Scikit-Learn Implementation

```python
from sklearn.preprocessing import OneHotEncoder

onehot_encoder = OneHotEncoder(sparse=False)
encoded_columns = onehot_encoder.fit_transform(df[['Category']])
df_encoded = pd.DataFrame(encoded_columns, columns=onehot_encoder.get_feature_names

df = pd.concat([df, df_encoded], axis=1).drop(columns=['Category'])
```



2.3 Target Encoding

When to Use?

- When categorical variables have high cardinality.
- Used mainly in **tree-based models**.

Manual Implementation

```python
df['Category'] = df.groupby('Category')['Target'].transform('mean')
```

Scikit-Learn Implementation

```python
from category_encoders import TargetEncoder

target_encoder = TargetEncoder()
df['Category'] = target_encoder.fit_transform(df['Category'], df['Target'])
```

3. Feature Scaling

Why?

Some ML models (e.g., Logistic Regression, SVM, KNN) require features to be on the same scale to perform well.

Methods

3.1 Min-Max Scaling (Normalization)

When to Use?

- When data has a **fixed range**.
- Works well for **image processing** and neural networks.

Manual Implementation

python Copy Edit

```python
df['Feature1'] = (df['Feature1'] - df['Feature1'].min()) / (df['Feature1'].max() -
```

Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
```

3.2 Standardization (Z-score Normalization)

When to Use?

- When data follows a **normal distribution.**
- Common for **linear models, PCA, and clustering.**

Manual Implementation

```python
df['Feature1'] = (df['Feature1'] - df['Feature1'].mean()) / df['Feature1'].std()
```

Scikit-Learn Implementation

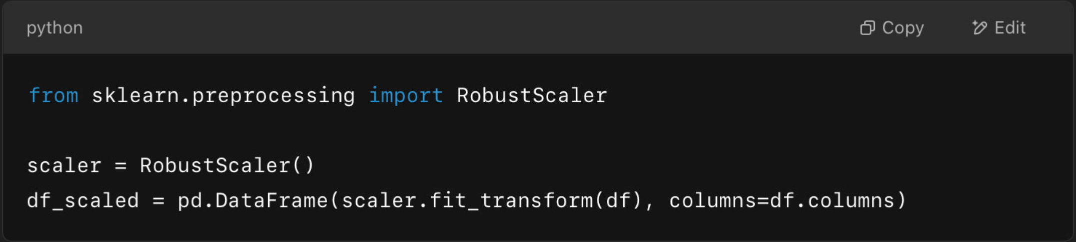
```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
```

3.3 Robust Scaling

When to Use?

- When data has **many outliers**.

Scikit-Learn Implementation



```python
from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
```

4. Feature Engineering

Why?

Creating meaningful features improves model performance.

Methods

4.1 Polynomial Features

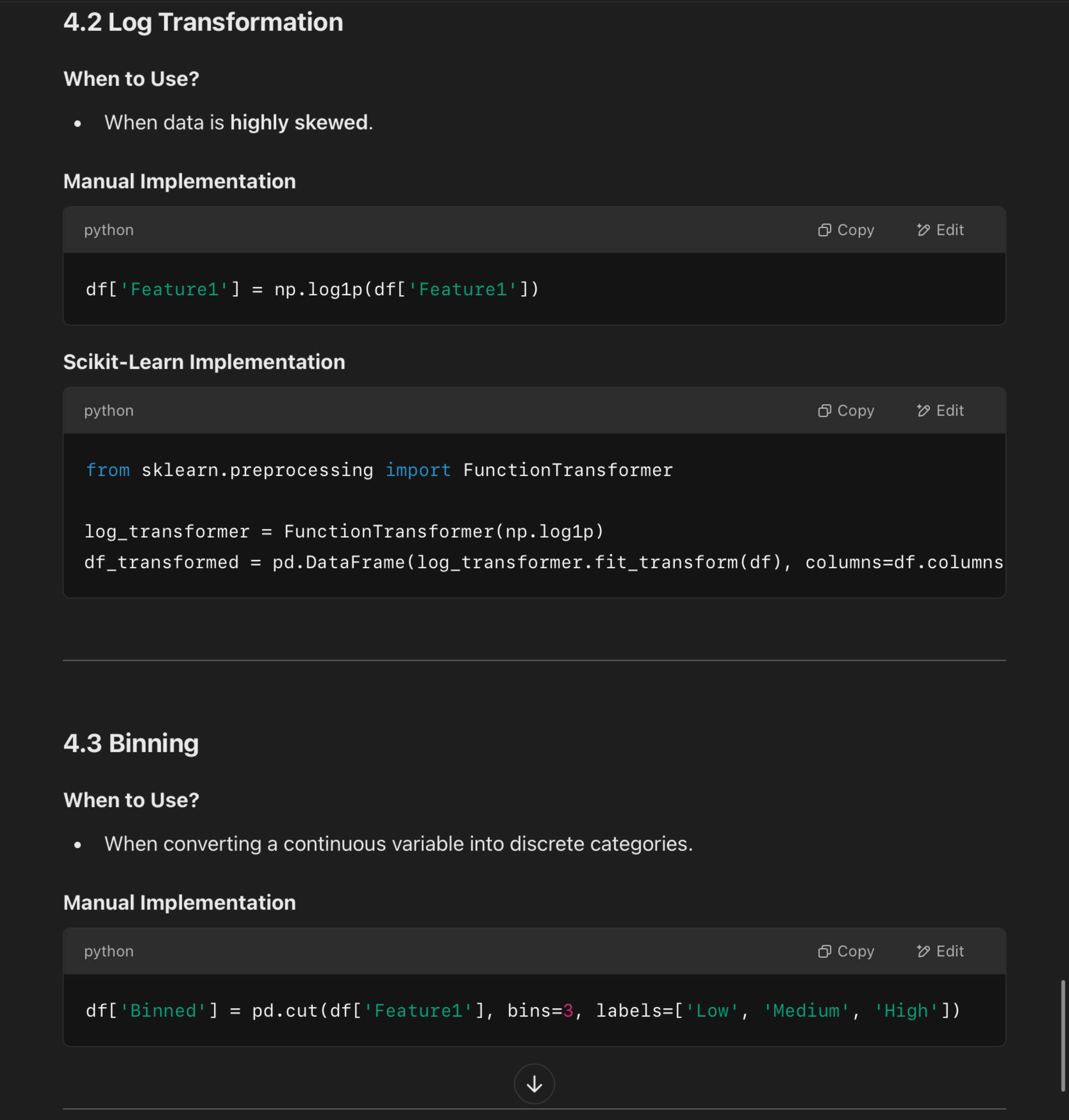
When to Use?

- When relationships are **non-linear**.

Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2)
df_poly = pd.DataFrame(poly.fit_transform(df), columns=poly.get_feature_names_out(d
```



4.2 Log Transformation

When to Use?

- When data is **highly skewed**.

Manual Implementation

```python
df['Feature1'] = np.log1p(df['Feature1'])
```

Scikit-Learn Implementation

```python
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log1p)
df_transformed = pd.DataFrame(log_transformer.fit_transform(df), columns=df.columns
```

4.3 Binning

When to Use?

- When converting a continuous variable into discrete categories.

Manual Implementation

```python
df['Binned'] = pd.cut(df['Feature1'], bins=3, labels=['Low', 'Medium', 'High'])
```