Effat University
Department of Computer Science
Energy and Information Technology Research Center

LAB 3

Data Preprocessing & Feature Engineering

CS4082 – Machine Learning

Preparing Real-World Data for Machine Learning Models in Python

Prepared by: Dr. Naila Marir
Semester: Spring 2026

Contents

1 Lab Overview

In Lab 2, you learned how to load data, train models, and evaluate predictions using scikit-learn. However, we used *clean* datasets that were ready to go. In the real world, data is **messy** – it has missing values, different scales, text columns, and irrelevant features. If you feed messy data into a model, you get messy results.

In this lab, you will learn the essential skill of **data preprocessing**: cleaning, transforming, and engineering features so that your ML models perform at their best.

1.1 What You Will Learn

- How to identify and handle missing values
- Encoding categorical (text) variables into numbers
- Feature scaling: StandardScaler and MinMaxScaler
- How scaling impacts KNN performance (a practical demonstration)
- Building reproducible pipelines with scikit-learn's `Pipeline`
- Basic feature selection using correlation analysis
- Applying the full preprocessing workflow on a realistic dataset

1.2 Prerequisites

- Completion of Lab 2 (scikit-learn basics: fit → predict → evaluate)
- Basic Python and Pandas knowledge
- Google Colab account (recommended) or local Python 3.8+ installation

Why Does Preprocessing Matter?

Industry practitioners estimate that **80% of a data scientist's time** is spent on data preprocessing and cleaning. A well-preprocessed dataset can often improve model accuracy more than switching to a fancier algorithm!

2 Part 1: Understanding the Problem with Messy Data

Before we learn the solutions, let's first *see* the problem. We will create a realistic messy dataset that mimics what you would encounter in a real project.

2.1 Step 1: Create a Messy Dataset

Run this code to generate a dataset about hospital patients in Saudi Arabia:

```
import pandas as pd
import numpy as np

np.random.seed(42)
n = 200

data = {
    'age': np.random.randint(18, 80, n).astype(float),
    'blood_pressure': np.round(np.random.uniform(90, 180, n), 1),
    'cholesterol': np.round(np.random.uniform(150, 350, n), 1),
    'bmi': np.round(np.random.uniform(18, 42, n), 1),
    'gender': np.random.choice(['Male', 'Female'], n),
    'city': np.random.choice(
        ['Jeddah', 'Riyadh', 'Dammam', 'Makkah'], n),
    'smoker': np.random.choice(['Yes', 'No'], n, p=[0.3, 0.7]),
    'heart_disease': np.random.choice([0, 1], n, p=[0.6, 0.4])
}

df = pd.DataFrame(data)

# Inject missing values (realistic!)
missing_idx = np.random.choice(n, 20, replace=False)
df.loc[missing_idx[:10], 'age'] = np.nan
df.loc[missing_idx[10:15], 'blood_pressure'] = np.nan
df.loc[missing_idx[15:], 'cholesterol'] = np.nan

print(f'Dataset shape: {df.shape}')
print(f'\nFirst 5 rows:')
print(df.head())
print(f'\nMissing values per column:')
print(df.isnull().sum())
```

2.2 Step 2: Identify the Issues

Let's systematically check what needs fixing:

```
print('=== Data Types ===')
print(df.dtypes)

print('\n=== Missing Values ===')
print(df.isnull().sum())

print('\n=== Numeric Ranges (notice the scale differences!) ===')
print(df.describe().round(2))

print('\n=== Categorical Columns ===')
for col in ['gender', 'city', 'smoker']:
    print(f'{col}: {df[col].unique()}')
```

Three Problems We Need to Fix

1. **Missing values**: `age`, `blood_pressure`, and `cholesterol` have NaN entries.
2. **Categorical columns**: `gender`, `city`, and `smoker` are text – ML models need numbers.
3. **Different scales**: `age` ranges from 18–80, but `cholesterol` ranges from 150–350. This can confuse distance-based algorithms like KNN.

Task 1: Explore the Messy Data

- Use `df.info()` to see column types and non-null counts in one view.
- Calculate the *percentage* of missing values for each column (hint: divide by `len(df)` and multiply by 100).
- Use `df['city'].value_counts()` to see how many patients come from each city.
- Which column has the most missing values?

3 Part 2: Handling Missing Values

Missing data is one of the most common problems in real datasets. Scikit-learn provides the `SimpleImputer` class to handle this systematically.

3.1 Strategy Overview

Strategy	When to Use	Code
Mean	Numeric, roughly symmetric data	`strategy='mean'`
Median	Numeric, skewed data or outliers	`strategy='median'`
Most Frequent	Categorical data	`strategy='most_frequent'`
Drop Rows	Very few missing values (<5%)	`df.dropna()`

Table 1: Common strategies for handling missing values.

3.2 Step 1: Impute Numeric Columns

```
from sklearn.impute import SimpleImputer

# Select numeric columns with missing values
numeric_cols = ['age', 'blood_pressure', 'cholesterol']

# Create an imputer that fills NaN with the median
imputer = SimpleImputer(strategy='median')

# Fit on the data and transform
df[numeric_cols] = imputer.fit_transform(df[numeric_cols])

# Verify: no more missing values!
print('Missing values after imputation:')
print(df[numeric_cols].isnull().sum())
```

Why Median Over Mean?

The median is more robust to **outliers**. If one patient has a blood pressure of 300 (a data entry error), the mean would be pulled upward, but the median stays stable. For medical data, median is usually the safer choice.

3.3 Step 2: Verify the Fix

```
print(f'Total missing values in entire dataset: {df.isnull().sum().sum()}')
print(f'\nDataset shape (no rows lost!): {df.shape}')
print(df.describe().round(2))
```

Task 2: Experiment with Imputation

- Re-create the messy dataset (run the creation code again).
- This time, use `strategy='mean'` instead of `'median'`. Compare the filled values – are they different? By how much?
- Try using `df.dropna()` instead. How many rows do you lose? Is that acceptable for 200 samples?

- **Think**: In what scenario would dropping rows be better than imputing?

4 Part 3: Encoding Categorical Variables

Machine learning models work with numbers, not text. We need to convert categorical columns like `gender`, `city`, and `smoker` into numeric format.

4.1 Two Main Approaches

Method	Best For	Example
Label Encoding	Binary or ordinal categories	Male/Female → 0/1
One-Hot Encoding	Nominal categories (no order)	Jeddah/Riyadh/... → separate columns

Table 2: Categorical encoding methods.

4.2 Step 1: Label Encoding for Binary Variables

For columns with only two values, we can simply map them to 0 and 1:

```
from sklearn.preprocessing import LabelEncoder

# Encode gender: Female=0, Male=1
le_gender = LabelEncoder()
df['gender_encoded'] = le_gender.fit_transform(df['gender'])

# Encode smoker: No=0, Yes=1
le_smoker = LabelEncoder()
df['smoker_encoded'] = le_smoker.fit_transform(df['smoker'])

print('Original vs Encoded:')
print(df[['gender', 'gender_encoded',
          'smoker', 'smoker_encoded']].head(8))
```

4.3 Step 2: One-Hot Encoding for Multi-Category Variables

The `city` column has 4 categories (Jeddah, Riyadh, Dammam, Makkah). Label encoding would assign them numbers 0–3, but this implies an *order* (Riyadh > Jeddah?) that doesn't exist. One-Hot encoding creates a separate binary column for each category:

```
# One-Hot Encode the city column
city_dummies = pd.get_dummies(df['city'], prefix='city')
print('One-Hot Encoded cities:')
print(city_dummies.head())

# Add to dataframe and drop original text columns
df = pd.concat([df, city_dummies], axis=1)
df = df.drop(columns=['gender', 'city', 'smoker'])

print(f'\nNew shape: {df.shape}')
print(f'New columns: {list(df.columns)}')
```

The Label Encoding Trap

Never use Label Encoding for nominal categories with more than 2 values. If you encode Jeddah=0, Riyadh=1, Dammam=2, Makkah=3, the model might learn that "Makkah is greater than Jeddah" – which is meaningless! Always use One-Hot encoding for unordered

categories.

Task 3: Practice Encoding

- Print the first 10 rows of the fully encoded dataset. How many columns do you have now?
- What would happen if a city column had 50 unique values? How many new columns would One-Hot encoding create? Is this a problem?
- **Bonus**: Use `pd.get_dummies(df['city'], prefix='city', drop_first=True)` – what changes and why might this be useful?

5 Part 4: Feature Scaling

Feature scaling is the process of putting all numeric features on a similar range. This is **critical** for distance-based algorithms like KNN and SVM.

5.1 Why Scaling Matters

Consider our dataset: `age` ranges from 18–80 while `cholesterol` ranges from 150–350. Without scaling, `cholesterol` will *dominate* the distance calculation in KNN simply because its numbers are bigger – not because it's more important!

5.2 Two Common Scalers

Scaler	Formula	Result
StandardScaler	$z = \frac{x-\mu}{\sigma}$	Mean = 0, Std = 1
MinMaxScaler	$x' = \frac{x-x_{\min}}{x_{\max}-x_{\min}}$	Range [0, 1]

Table 3: Comparison of common scaling methods.

5.3 Step 1: Apply StandardScaler

```
from sklearn.preprocessing import StandardScaler

# Prepare features and target
feature_cols = ['age', 'blood_pressure', 'cholesterol', 'bmi',
                'gender_encoded', 'smoker_encoded',
                'city_Dammam', 'city_Jeddah',
                'city_Makkah', 'city_Riyadh']
X = df[feature_cols].values
y = df['heart_disease'].values

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Compare before and after
import pandas as pd
comparison = pd.DataFrame({
    'Feature': feature_cols,
    'Original Mean': X.mean(axis=0).round(2),
    'Original Std': X.std(axis=0).round(2),
    'Scaled Mean': X_scaled.mean(axis=0).round(2),
    'Scaled Std': X_scaled.std(axis=0).round(2)
})
print(comparison.to_string(index=False))
```

Expected Output

After scaling, every feature should have a mean close to 0 and a standard deviation close to 1. This puts all features on an equal footing for the model.

5.4 Step 2: Visualize the Effect of Scaling

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Before scaling
axes[0].boxplot([X[:, 0], X[:, 1], X[:, 2], X[:, 3]],
                labels=['Age', 'BP', 'Cholesterol', 'BMI'])
axes[0].set_title('Before Scaling')
axes[0].set_ylabel('Value')

# After scaling
axes[1].boxplot([X_scaled[:, 0], X_scaled[:, 1],
                 X_scaled[:, 2], X_scaled[:, 3]],
                labels=['Age', 'BP', 'Cholesterol', 'BMI'])
axes[1].set_title('After StandardScaler')
axes[1].set_ylabel('Value')

plt.tight_layout()
plt.show()
```

Task 4: Compare Scalers

- Apply `MinMaxScaler` to the same data. What range do the values fall into?
- Create the same box plot comparison for MinMaxScaler. How does it differ from StandardScaler?
- **Think**: If your data has many outliers, which scaler would be more robust? Why?

6 Part 5: The Scaling Impact – KNN Before and After

Let's prove that scaling matters with a concrete experiment. We will train KNN on the *same data* with and without scaling and compare the accuracy.

6.1 Experiment: KNN Without Scaling vs. With Scaling

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

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

X_train_s, X_test_s, _, _ = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y)

# --- KNN WITHOUT Scaling ---
knn_raw = KNeighborsClassifier(n_neighbors=5)
knn_raw.fit(X_train, y_train)
raw_acc = accuracy_score(y_test, knn_raw.predict(X_test))

# --- KNN WITH Scaling ---
knn_scaled = KNeighborsClassifier(n_neighbors=5)
knn_scaled.fit(X_train_s, y_train)
scaled_acc = accuracy_score(y_test, knn_scaled.predict(X_test_s))

print(f'KNN Accuracy WITHOUT scaling: {raw_acc:.2%}')
print(f'KNN Accuracy WITH scaling:    {scaled_acc:.2%}')
print(f'Improvement: {(scaled_acc - raw_acc):.2%}')
```

What to Expect

You should see a noticeable improvement in KNN accuracy after scaling. This is because KNN uses Euclidean distance, and unscaled features with larger ranges dominate the distance calculation. Scaling ensures every feature contributes fairly.

6.2 Bonus: Decision Tree Comparison

```
from sklearn.tree import DecisionTreeClassifier

# Decision Tree WITHOUT scaling
dt_raw = DecisionTreeClassifier(random_state=42)
dt_raw.fit(X_train, y_train)
dt_raw_acc = accuracy_score(y_test, dt_raw.predict(X_test))

# Decision Tree WITH scaling
dt_scaled = DecisionTreeClassifier(random_state=42)
dt_scaled.fit(X_train_s, y_train)
dt_scaled_acc = accuracy_score(y_test, dt_scaled.predict(X_test_s))

print(f'DT Accuracy WITHOUT scaling: {dt_raw_acc:.2%}')
print(f'DT Accuracy WITH scaling:    {dt_scaled_acc:.2%}')
```

Key Insight

Decision Trees are **not affected by scaling** because they split based on thresholds within each feature independently – they don't calculate distances between samples. This is an important distinction: **scaling is essential for distance-based models (KNN, SVM) but optional for tree-based models (Decision Tree, Random Forest).**

Task 5: Investigate Scaling Effects

- Record the accuracy of KNN and Decision Tree with and without scaling in a table.
- Try KNN with `n_neighbors=3` and `n_neighbors=10`, both with and without scaling. Does scaling always help?
- Write 2–3 sentences explaining *why* KNN benefits from scaling but Decision Trees do not.

7 Part 6: Building a Preprocessing Pipeline

So far we applied each step separately. In real projects, this leads to messy, error-prone code. Scikit-learn's `Pipeline` chains multiple steps together into a single, clean object.

7.1 Why Pipelines?

- **Reproducibility**: The same preprocessing is applied to training and testing data automatically.
- **No data leakage**: The pipeline ensures that the scaler is fit *only* on training data.
- **Clean code**: One object to fit, predict, and evaluate.

What is Data Leakage?

If you scale *all* the data before splitting, the scaler "sees" the test set during fitting. This leaks information from the test set into training, making your accuracy look artificially good. A pipeline prevents this by fitting the scaler only on `X_train`.

7.2 Step 1: Create a Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

# Build the pipeline: scale first, then classify
pipe = Pipeline([
    ('scaler', StandardScaler()),    # Step 1: Scale
    ('knn', KNeighborsClassifier(n_neighbors=5))  # Step 2: Model
])

# Split the original (unscaled) data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)

# The pipeline handles scaling internally!
pipe.fit(X_train, y_train)
pipe_acc = accuracy_score(y_test, pipe.predict(X_test))

print(f'Pipeline KNN Accuracy: {pipe_acc:.2%}')
```

7.3 Step 2: Swap Models Easily

The beauty of pipelines is that you can swap the model without changing anything else:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

# Pipeline with SVM
pipe_svm = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC(kernel='rbf', random_state=42))
])
pipe_svm.fit(X_train, y_train)
svm_acc = accuracy_score(y_test, pipe_svm.predict(X_test))
```

```
# Pipeline with Decision Tree
pipe_dt = Pipeline([
    ('scaler', StandardScaler()),
    ('dt', DecisionTreeClassifier(random_state=42))
])
pipe_dt.fit(X_train, y_train)
dt_acc = accuracy_score(y_test, pipe_dt.predict(X_test))

print(f'Pipeline KNN Accuracy: {pipe_acc:.2%}')
print(f'Pipeline SVM Accuracy: {svm_acc:.2%}')
print(f'Pipeline DT Accuracy:  {dt_acc:.2%}')
```

Task 6: Build Your Own Pipeline

- Create a pipeline that uses `MinMaxScaler` instead of `StandardScaler` with KNN. Compare the accuracy.
- Add `SimpleImputer` as the first step in your pipeline (before the scaler). Now the pipeline handles missing values AND scaling automatically.
- **Hint**: The 3-step pipeline would look like: `[('imputer', SimpleImputer(...)), ('scaler', StandardScaler()), ('knn', KNeighborsClassifier())]`

8 Part 7: Feature Selection with Correlation Analysis

Not all features are equally useful. Some may be irrelevant or redundant. Feature selection helps you identify which features actually contribute to prediction.

8.1 Step 1: Compute the Correlation Matrix

```
# Create a DataFrame with our processed features
df_processed = pd.DataFrame(X_scaled, columns=feature_cols)
df_processed['heart_disease'] = y

# Compute correlations
corr_matrix = df_processed.corr()

# Show correlation with the target
print('Correlation with heart_disease:')
target_corr = corr_matrix['heart_disease'].drop('heart_disease')
print(target_corr.sort_values(ascending=False).round(3))
```

8.2 Step 2: Visualize with a Heatmap

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(10, 8))
im = plt.imshow(corr_matrix.values, cmap='coolwarm',
                vmin=-1, vmax=1)
plt.colorbar(im, label='Correlation')

# Add labels
ticks = range(len(corr_matrix.columns))
plt.xticks(ticks, corr_matrix.columns, rotation=45, ha='right')
plt.yticks(ticks, corr_matrix.columns)

# Add correlation values on the heatmap
for i in range(len(corr_matrix)):
    for j in range(len(corr_matrix)):
        val = corr_matrix.values[i, j]
        color = 'white' if abs(val) > 0.5 else 'black'
        plt.text(j, i, f'{val:.2f}', ha='center', va='center',
                 fontsize=7, color=color)

plt.title('Feature Correlation Matrix')
plt.tight_layout()
plt.show()
```

8.3 Step 3: Select Top Features

```
# Select features with absolute correlation > threshold
threshold = 0.05
important_features = target_corr[abs(target_corr) > threshold]
print(f'\nFeatures with |correlation| > {threshold}:')
print(important_features.sort_values(ascending=False).round(3))

# Train with only the top features
```

```
top_features = important_features.index.tolist()
X_selected = df_processed[top_features].values

X_tr, X_te, y_tr, y_te = train_test_split(
    X_selected, y, test_size=0.2, random_state=42, stratify=y)

pipe_selected = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier(n_neighbors=5))
])
pipe_selected.fit(X_tr, y_tr)
sel_acc = accuracy_score(y_te, pipe_selected.predict(X_te))

print(f'\nAccuracy with ALL features:      {pipe_acc:.2%}')
print(f'Accuracy with SELECTED features: {sel_acc:.2%}')
print(f'Features used: {len(top_features)} out of {len(feature_cols)}'
    )
```

Interpreting Correlations

- Values close to **+1**: Strong positive relationship (as one increases, the other increases).
- Values close to **−1**: Strong negative relationship (as one increases, the other decreases).
- Values close to **0**: Little or no linear relationship.

Note: Correlation only captures *linear* relationships. Two features can have a strong non-linear relationship but low correlation.

Task 7: Feature Selection

- Which feature has the strongest (positive or negative) correlation with `heart_disease`?
- Try different threshold values (0.03, 0.08, 0.10). How does accuracy change?
- **Think**: Why might using fewer features sometimes give *better* accuracy? (Hint: consider the "curse of dimensionality.")

9 Part 8: Putting It All Together – Full Workflow

Let's apply every technique from this lab in one clean, end-to-end workflow. This is the pattern you should follow in your own projects.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# ===== STEP 1: Load Raw Data =====
# (Re-create the messy dataset)
np.random.seed(42)
n = 200
data = {
    'age': np.random.randint(18, 80, n).astype(float),
    'blood_pressure': np.round(np.random.uniform(90, 180, n), 1),
    'cholesterol': np.round(np.random.uniform(150, 350, n), 1),
    'bmi': np.round(np.random.uniform(18, 42, n), 1),
    'gender': np.random.choice(['Male', 'Female'], n),
    'city': np.random.choice(
        ['Jeddah', 'Riyadh', 'Dammam', 'Makkah'], n),
    'smoker': np.random.choice(['Yes', 'No'], n, p=[0.3, 0.7]),
    'heart_disease': np.random.choice([0, 1], n, p=[0.6, 0.4])
}
df = pd.DataFrame(data)
missing_idx = np.random.choice(n, 20, replace=False)
df.loc[missing_idx[:10], 'age'] = np.nan
df.loc[missing_idx[10:15], 'blood_pressure'] = np.nan
df.loc[missing_idx[15:], 'cholesterol'] = np.nan

print('=== Step 1: Raw Data ===')
print(f'Shape: {df.shape}, Missing: {df.isnull().sum().sum()}')

# ===== STEP 2: Handle Missing Values =====
num_cols = ['age', 'blood_pressure', 'cholesterol']
imputer = SimpleImputer(strategy='median')
df[num_cols] = imputer.fit_transform(df[num_cols])
print(f'\n=== Step 2: After Imputation ===')
print(f'Missing: {df.isnull().sum().sum()}')

# ===== STEP 3: Encode Categorical Variables =====
le = LabelEncoder()
df['gender'] = le.fit_transform(df['gender'])
df['smoker'] = le.fit_transform(df['smoker'])
df = pd.concat([df, pd.get_dummies(df['city'], prefix='city')],
               axis=1)
df = df.drop(columns=['city'])
print(f'\n=== Step 3: After Encoding ===')
print(f'Columns: {list(df.columns)}')

# ===== STEP 4: Separate Features and Target =====
X = df.drop(columns=['heart_disease']).values
```

```
y = df['heart_disease'].values

# ===== STEP 5: Split Data =====
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)

# ===== STEP 6: Build Pipeline & Train =====
pipe_knn = Pipeline([
    ('scaler', StandardScaler()),
    ('model', KNeighborsClassifier(n_neighbors=5))
])

pipe_dt = Pipeline([
    ('scaler', StandardScaler()),
    ('model', DecisionTreeClassifier(random_state=42))
])

pipe_knn.fit(X_train, y_train)
pipe_dt.fit(X_train, y_train)

# ===== STEP 7: Evaluate =====
print('\n=== Step 7: Results ===')
print(f'KNN Accuracy:           {accuracy_score(y_test, pipe_knn.
    predict(X_test)):.2%}')
print(f'Decision Tree Accuracy: {accuracy_score(y_test, pipe_dt.
    predict(X_test)):.2%}')

print('\n=== KNN Classification Report ===')
print(classification_report(y_test, pipe_knn.predict(X_test),
      target_names=['No Disease', 'Disease']))
```

The Complete Preprocessing Checklist

Before training any model, always check:

1. ✓ Missing values handled (imputed or dropped)
2. ✓ Categorical variables encoded (Label or One-Hot)
3. ✓ Features scaled (especially for KNN, SVM)
4. ✓ Data split into train/test (no data leakage!)
5. ✓ Pipeline built for reproducibility

10 Part 9: Summary and Key Takeaways

10.1 The Preprocessing Workflow

Step	Action	Tool	Why
1	Explore data	`df.info()`, `df.describe()`	Understand the problems
2	Handle missing values	`SimpleImputer`	Models can't handle NaN
3	Encode categories	`LabelEncoder`, `get_dummies`	Models need numbers
4	Scale features	`StandardScaler`	Equal contribution
5	Select features	Correlation analysis	Remove noise
6	Build pipeline	`Pipeline`	Reproducibility

Table 4: The data preprocessing workflow.

10.2 Key Rules to Remember

1. **Always explore before preprocessing** – understand what's wrong before trying to fix it.
2. **Scale for distance-based models** (KNN, SVM) – tree-based models don't need it.
3. **Use One-Hot for nominal categories** – Label Encoding implies false ordering.
4. **Use Pipelines** – they prevent data leakage and keep your code clean.
5. **Fit on training data only** – then transform both training and testing data.

10.3 What to Explore Next

- **Advanced imputation**: Try `KNNImputer` which uses K-Nearest Neighbors to fill missing values based on similar samples.
- **Feature engineering**: Create new features from existing ones (e.g., BMI category from BMI values).
- **ColumnTransformer**: Apply different preprocessing to different columns in a single pipeline using `sklearn.compose.ColumnTransformer`.
- **Cross-validation**: Use `cross_val_score()` for more robust evaluation than a single train/test split.

11 Submission Requirements

11.1 What to Submit

1. Your completed Colab notebook (`.ipynb`) with all code cells executed and outputs visible.
2. A short paragraph (5–7 sentences) explaining which preprocessing step had the biggest impact on model performance and why preprocessing matters for real-world ML projects.

11.2 Grading Rubric

Criterion	Points	Weight
Data exploration and missing values (Tasks 1–2)	15	15%
Categorical encoding (Task 3)	15	15%
Feature scaling and comparison (Tasks 4–5)	20	20%
Pipeline construction (Task 6)	20	20%
Feature selection and analysis (Task 7)	15	15%
Written reflection and overall quality	15	15%
Total	**100**	**100%**

Table 5: Lab 3 grading rubric.

Final Tip

In real ML projects, a well-preprocessed dataset often matters more than the choice of algorithm. Master these skills, and you will build better models – no matter which algorithm you use!