**Data Science/Machine Learning Notebook Structure**

**Recommended Jupyter Notebook Organization for ML Projects**

**1. Header Section**

# [Project Title]: [ML Problem Type - Classification/Regression/Clustering]

\*\*Author:\*\* [Your Name]

\*\*Date:\*\* [Date]

\*\*Objective:\*\* [Predict X using Y features]

\*\*Model Type:\*\* [Supervised/Unsupervised Learning]

## Table of Contents

1. Problem Definition & Business Context

2. Data Import & Overview

3. Exploratory Data Analysis (EDA)

4. Data Preprocessing & Cleaning

5. Feature Engineering

6. Feature Selection

7. Data Transformation (Scaling/Encoding)

8. Model Development & Training

9. Model Evaluation & Validation

10. Model Interpretation & Feature Importance

11. Business Recommendations & Deployment Considerations

12. Conclusions & Next Steps

**2. Section 1: Problem Definition & Business Context**

## 1. Problem Definition & Business Context

### Machine Learning Problem Type

- \*\*Problem Type:\*\* [Classification/Regression/Clustering/Time Series]

- \*\*Target Variable:\*\* [What are we predicting?]

- \*\*Success Metric:\*\* [Accuracy, RMSE, F1-Score, etc.]

### Business Context

- \*\*Why is this prediction valuable?\*\*

- \*\*How will the model be used in practice?\*\*

- \*\*What's the cost of false positives vs false negatives?\*\*

### Assumptions & Constraints

- \*\*Data assumptions:\*\* [Stationarity, independence, etc.]

- \*\*Business constraints:\*\* [Real-time predictions? Interpretability required?]

- \*\*Performance requirements:\*\* [Minimum accuracy threshold, latency requirements]

**3. Section 2: Data Import & Overview**

# Import libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# ML libraries

from sklearn.model\_selection import train\_test\_split, cross\_val\_score, GridSearchCV

from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder

from sklearn.feature\_selection import SelectKBest, RFE

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report, confusion\_matrix, roc\_auc\_score

# ... other imports based on problem type

# Load data

df = pd.read\_csv('data.csv')

### Dataset Overview

- \*\*Source:\*\* [Data source and context]

- \*\*Target Variable:\*\* [What we're predicting]

- \*\*Features:\*\* [Number and types of features]

- \*\*Sample Size:\*\* [Rows and columns]

# Initial data exploration

print(f"Dataset shape: {df.shape}")

print(f"Target variable distribution:")

print(df['target'].value\_counts(normalize=True))

# Check data types and missing values

df.info()

df.describe()

**4. Section 3: Exploratory Data Analysis (EDA)**

## 3. Exploratory Data Analysis

### Target Variable Analysis

# Analyze target distribution

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

df['target'].hist(bins=30)

plt.title('Target Variable Distribution')

plt.subplot(1, 2, 2)

df['target'].value\_counts().plot(kind='bar')

plt.title('Target Classes Count')

plt.show()

# Statistical summary of target

print(f"Target statistics:")

print(df['target'].describe())

### Feature Analysis & Correlation

# Correlation matrix

plt.figure(figsize=(12, 8))

correlation\_matrix = df.select\_dtypes(include=[np.number]).corr()

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', center=0)

plt.title('Feature Correlation Matrix')

plt.show()

# Feature distributions

numerical\_features = df.select\_dtypes(include=[np.number]).columns

df[numerical\_features].hist(figsize=(15, 12), bins=30)

plt.tight\_layout()

plt.show()

### Target vs Features Relationship

# Analyze relationship between features and target

# For numerical features

for feature in numerical\_features:

if feature != 'target':

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

df.boxplot(column=feature, by='target')

plt.title(f'{feature} by Target')

plt.subplot(1, 2, 2)

for target\_class in df['target'].unique():

subset = df[df['target'] == target\_class]

plt.hist(subset[feature], alpha=0.7, label=f'Target: {target\_class}')

plt.legend()

plt.title(f'{feature} Distribution by Target')

plt.show()

**5. Section 4: Data Preprocessing & Cleaning**

## 4. Data Preprocessing & Cleaning

### Missing Values Treatment

# Analyze missing values

missing\_data = df.isnull().sum()

missing\_percentage = (missing\_data / len(df)) \* 100

missing\_df = pd.DataFrame({'Missing Count': missing\_data, 'Percentage': missing\_percentage})

print("Missing Data Analysis:")

print(missing\_df[missing\_df['Missing Count'] > 0])

# Handle missing values

# Strategy depends on feature type and business context

df\_clean = df.copy()

# Example strategies:

# Numerical: median/mean imputation or advanced methods

# Categorical: mode imputation or 'Unknown' category

# Time series: forward/backward fill

# Document your imputation strategy for each feature

### Outlier Detection & Treatment

# Detect outliers using IQR method

def detect\_outliers\_iqr(data, feature):

Q1 = data[feature].quantile(0.25)

Q3 = data[feature].quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

return data[(data[feature] < lower\_bound) | (data[feature] > upper\_bound)]

# Check outliers for numerical features

for feature in numerical\_features:

if feature != 'target':

outliers = detect\_outliers\_iqr(df\_clean, feature)

print(f"{feature}: {len(outliers)} outliers ({len(outliers)/len(df\_clean)\*100:.1f}%)")

# Visualize outliers

plt.figure(figsize=(15, 8))

df\_clean[numerical\_features].boxplot()

plt.xticks(rotation=45)

plt.title('Outlier Detection - Boxplots')

plt.show()

# Handle outliers (document strategy: remove, cap, transform)

**6. Section 5: Feature Engineering**

## 5. Feature Engineering

### Creating New Features

# Create new features based on domain knowledge

df\_engineered = df\_clean.copy()

# Examples of feature engineering:

# 1. Polynomial features

# df\_engineered['feature1\_squared'] = df\_engineered['feature1'] \*\* 2

# 2. Interaction features

# df\_engineered['feature1\_x\_feature2'] = df\_engineered['feature1'] \* df\_engineered['feature2']

# 3. Binning continuous variables

# df\_engineered['age\_group'] = pd.cut(df\_engineered['age'], bins=[0, 25, 50, 75, 100])

# 4. Date/time features (if applicable)

# df\_engineered['hour'] = pd.to\_datetime(df\_engineered['timestamp']).dt.hour

# df\_engineered['day\_of\_week'] = pd.to\_datetime(df\_engineered['timestamp']).dt.dayofweek

# 5. Text features (if applicable)

# df\_engineered['text\_length'] = df\_engineered['text\_column'].str.len()

print(f"Original features: {df\_clean.shape[1]}")

print(f"After feature engineering: {df\_engineered.shape[1]}")

print(f"New features added: {df\_engineered.shape[1] - df\_clean.shape[1]}")

### Feature Engineering Validation

# Check correlation of new features with target

new\_features = [col for col in df\_engineered.columns if col not in df\_clean.columns]

if new\_features:

for feature in new\_features:

if df\_engineered[feature].dtype in ['int64', 'float64']:

correlation = df\_engineered[feature].corr(df\_engineered['target'])

print(f"{feature}: correlation with target = {correlation:.3f}")

**7. Section 6: Feature Selection**

## 6. Feature Selection

### Statistical Feature Selection

from sklearn.feature\_selection import SelectKBest, f\_classif, chi2, mutual\_info\_classif

# Separate features and target

X = df\_engineered.drop('target', axis=1)

y = df\_engineered['target']

# Select numerical and categorical features

numerical\_features = X.select\_dtypes(include=[np.number]).columns.tolist()

categorical\_features = X.select\_dtypes(exclude=[np.number]).columns.tolist()

print(f"Numerical features: {len(numerical\_features)}")

print(f"Categorical features: {len(categorical\_features)}")

# Statistical tests for feature selection

# For numerical features - F-test

if numerical\_features:

selector\_f = SelectKBest(score\_func=f\_classif, k='all')

X\_numerical = X[numerical\_features]

selector\_f.fit(X\_numerical, y)

feature\_scores\_f = pd.DataFrame({

'Feature': numerical\_features,

'F\_Score': selector\_f.scores\_,

'P\_Value': selector\_f.pvalues\_

}).sort\_values('F\_Score', ascending=False)

print("\nTop numerical features by F-score:")

print(feature\_scores\_f.head(10))

### Recursive Feature Elimination (RFE)

from sklearn.feature\_selection import RFE

from sklearn.ensemble import RandomForestClassifier

# Handle categorical variables first (basic encoding for RFE)

X\_for\_rfe = X.copy()

for col in categorical\_features:

le = LabelEncoder()

X\_for\_rfe[col] = le.fit\_transform(X\_for\_rfe[col].astype(str))

# RFE with Random Forest

rf = RandomForestClassifier(n\_estimators=100, random\_state=42)

selector\_rfe = RFE(estimator=rf, n\_features\_to\_select=10, step=1)

selector\_rfe.fit(X\_for\_rfe, y)

# Get selected features

selected\_features\_rfe = X\_for\_rfe.columns[selector\_rfe.support\_].tolist()

print(f"RFE selected features: {selected\_features\_rfe}")

# Feature ranking

feature\_ranking = pd.DataFrame({

'Feature': X\_for\_rfe.columns,

'Ranking': selector\_rfe.ranking\_,

'Selected': selector\_rfe.support\_

}).sort\_values('Ranking')

print("\nFeature Ranking (RFE):")

print(feature\_ranking.head(15))

### Feature Importance (Tree-based)

# Feature importance using Random Forest

rf\_importance = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_importance.fit(X\_for\_rfe, y)

feature\_importance = pd.DataFrame({

'Feature': X\_for\_rfe.columns,

'Importance': rf\_importance.feature\_importances\_

}).sort\_values('Importance', ascending=False)

# Visualize feature importance

plt.figure(figsize=(10, 8))

sns.barplot(data=feature\_importance.head(15), x='Importance', y='Feature')

plt.title('Top 15 Feature Importances (Random Forest)')

plt.show()

# Select final features (combine multiple methods)

top\_features\_importance = feature\_importance.head(15)['Feature'].tolist()

print(f"Top features by importance: {top\_features\_importance}")

**8. Section 7: Data Transformation (Scaling/Encoding)**

## 7. Data Transformation

### Prepare Final Feature Set

# Combine feature selection results to choose final features

final\_features = list(set(selected\_features\_rfe) & set(top\_features\_importance))

print(f"Final selected features: {len(final\_features)}")

print(final\_features)

# Create final dataset

X\_final = X[final\_features].copy()

y\_final = y.copy()

print(f"Final dataset shape: {X\_final.shape}")

### Categorical Encoding

from sklearn.preprocessing import OneHotEncoder, LabelEncoder

# Identify categorical features in final set

categorical\_features\_final = X\_final.select\_dtypes(exclude=[np.number]).columns.tolist()

numerical\_features\_final = X\_final.select\_dtypes(include=[np.number]).columns.tolist()

print(f"Categorical features to encode: {categorical\_features\_final}")

print(f"Numerical features: {len(numerical\_features\_final)}")

# Apply encoding

X\_encoded = X\_final.copy()

# One-hot encoding for categorical features with few categories

# Label encoding for categorical features with many categories

for col in categorical\_features\_final:

unique\_values = X\_encoded[col].nunique()

print(f"{col}: {unique\_values} unique values")

if unique\_values <= 10: # One-hot encode

ohe = OneHotEncoder(sparse\_output=False, drop='first')

encoded\_features = ohe.fit\_transform(X\_encoded[[col]])

feature\_names = [f"{col}\_{category}" for category in ohe.categories\_[0][1:]]

encoded\_df = pd.DataFrame(encoded\_features, columns=feature\_names, index=X\_encoded.index)

X\_encoded = pd.concat([X\_encoded.drop(col, axis=1), encoded\_df], axis=1)

else: # Label encode

le = LabelEncoder()

X\_encoded[col] = le.fit\_transform(X\_encoded[col].astype(str))

print(f"After encoding shape: {X\_encoded.shape}")

### Feature Scaling

from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

# Split data before scaling to prevent data leakage

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_encoded, y\_final, test\_size=0.2, random\_state=42, stratify=y\_final

)

print(f"Training set: {X\_train.shape}")

print(f"Test set: {X\_test.shape}")

# Choose scaler based on data distribution

# StandardScaler: normal distribution

# RobustScaler: outliers present

# MinMaxScaler: bounded range needed

scaler = StandardScaler() # Most common choice

X\_train\_scaled = pd.DataFrame(

scaler.fit\_transform(X\_train),

columns=X\_train.columns,

index=X\_train.index

)

X\_test\_scaled = pd.DataFrame(

scaler.transform(X\_test),

columns=X\_test.columns,

index=X\_test.index

)

print("Scaling completed!")

print(f"Training mean after scaling: {X\_train\_scaled.mean().mean():.3f}")

print(f"Training std after scaling: {X\_train\_scaled.std().mean():.3f}")

**9. Section 8: Model Development & Training**

## 8. Model Development & Training

### Baseline Model

from sklearn.dummy import DummyClassifier

from sklearn.metrics import accuracy\_score, classification\_report

# Always start with a baseline

baseline\_model = DummyClassifier(strategy='most\_frequent')

baseline\_model.fit(X\_train\_scaled, y\_train)

baseline\_pred = baseline\_model.predict(X\_test\_scaled)

baseline\_accuracy = accuracy\_score(y\_test, baseline\_pred)

print(f"Baseline Accuracy (Most Frequent): {baseline\_accuracy:.3f}")

### Model Selection & Training