

Dataset Name: Bank Marketing Dataset

Dataset link: <https://www.kaggle.com/datasets/janiobachmann/bank-marketing-dataset>

Goal: Predict if a customer will subscribe to a bank product.

Size: ~3,000 records.

```
import pandas as pd
import numpy as np
import os
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
import warnings
warnings.filterwarnings('ignore')
```

```
# import the Dataset:
from google.colab import drive
drive.mount('/content/drive')
os.chdir('/content/drive/MyDrive/Colab Notebooks/Datasets/')
!ls
df = pd.read_csv('bank.csv')
df.head()
df.dtypes
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)

bank.csv heart.csv UCI_Credit_Card.csv
feature_importance.png melb_data.csv WA_Fn-UseC_-Telco-Customer-Churn.csv

```
0
age      int64
job      object
marital  object
education object
default  object
balance  int64
housing  object
loan     object
contact  object
day      int64
month    object
duration int64
campaign int64
pdays   int64
previous int64
poutcome object
deposit  object
```

dtype: object

```
df.columns
```

```
Index(['age', 'job', 'marital', 'education', 'default', 'balance', 'housing',
       'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays',
       'previous', 'poutcome', 'deposit'],
      dtype='object')
```

```
# Check for missing values
df.isnull().sum()
```

```

      0
age    0
job    0
marital 0
education 0
default 0
balance 0
housing 0
loan    0
contact 0
day     0
month   0
duration 0
campaign 0
pdays  0
previous 0
poutcome 0
deposit 0

```

dtype: int64

```
len(df)
```

11162

```
missing_percentage = df.isnull().mean() * 100
threshold = 30
columns_to_drop = missing_percentage[missing_percentage >
threshold].index
df = df.drop(columns=columns_to_drop)
print(f"Columns dropped: {list(columns_to_drop)}")
print(f"Remaining columns: {df.columns}")
```

```
Columns dropped: []
Remaining columns: Index(['age', 'job', 'marital', 'education', 'default', 'balance', 'housing',
                        'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays',
                        'previous', 'poutcome', 'deposit'],
                        dtype='object')
```

```
# Pre-processing on dataset
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
for column in df.select_dtypes(include=['object']).columns: df[column] = label_encoder.fit_transform(df[column])
df.head()
```

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome
0	59	0	1	1	0	2343	1	0	2	5	8	1042	1	-1	0	not in default
1	56	0	1	1	0	45	0	0	2	5	8	1467	1	-1	0	not in default
2	41	9	1	1	0	1270	1	0	2	5	8	1389	1	-1	0	not in default
3	55	7	1	1	0	2476	1	0	2	5	8	579	1	-1	0	not in default
4	54	0	1	2	0	184	0	0	2	5	8	673	2	-1	0	not in default

Next steps: [Generate code with df](#) [New interactive sheet](#)

```
for column in df.select_dtypes(include=['float64']).columns:
    mean_value = df[column].mean()
    df[column] = df[column].fillna(mean_value)

for column in df.select_dtypes(include=['object']).columns:
    mode_value = df[column].mode()[0]
    df[column] = df[column].fillna(mode_value)
```

```
df.isnull().sum()
```

	0
age	0
job	0
marital	0
education	0
default	0
balance	0
housing	0
loan	0
contact	0
day	0
month	0
duration	0
campaign	0
pdays	0
previous	0
poutcome	0
deposit	0
dtype:	int64

Start coding or [generate](#) with AI.

```
data = df.copy()
def detect_outliers_iqr(column):
    Q1 = column.quantile(0.25)
    Q3 = column.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return (column < lower_bound) | (column > upper_bound)
for col in data.columns:
    data[f'{col}_Outlier'] = detect_outliers_iqr(data[col])
outliers_only = data[data.filter(like='_Outlier').any(axis=1)]
outliers_only
```

	age	job	marital	education	default	balance	housing	loan	contact	day	...	loan_Outlier	contact_Outlier	day_0
0	59	0	1	1	0	2343	1	0	2	5	...	False	False	
1	56	0	1	1	0	45	0	0	2	5	...	False	False	
2	41	9	1	1	0	1270	1	0	2	5	...	False	False	
5	42	4	2	2	0	0	1	1	2	5	...	True	False	
6	56	4	1	2	0	830	1	1	2	6	...	True	False	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
11145	60	5	0	2	0	-134	0	0	0	12	...	False	False	
11146	30	0	1	1	1	23	0	1	0	4	...	True	False	
11154	52	9	1	2	0	523	1	1	0	8	...	True	False	
11155	35	1	1	1	0	80	1	1	0	21	...	True	False	
11160	43	9	1	1	0	0	0	1	0	8	...	True	False	

5666 rows × 34 columns

```
data = df.copy()
def replace_outliers_with_nan(column):
    Q1 = column.quantile(0.25)
    Q3 = column.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
```

```

upper_bound = Q3 + 1.5 * IQR
# Replace outliers with NaN
return column.where((column >= lower_bound) & (column <= upper_bound), np.nan)
# Replace outliers with NaN for each feature
for col in data.columns:
    data[col] = replace_outliers_with_nan(data[col])
null_counts = data.isnull().sum()
print("Sum of null values in each column:")
print(null_counts)

```

Sum of null values in each column:

```

age      171
job       0
marital   0
education 0
default   168
balance  1055
housing   0
loan     1460
contact   0
day        0
month      0
duration  636
campaign  601
pdays    2750
previous  1258
poutcome  1228
deposit   0
dtype: int64

```

```

from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=2)
data_imputed_knn = pd.DataFrame(imputer.fit_transform(data),
                                columns=data.columns)
data_imputed_knn.isnull().sum()

```

```

0
age      0
job       0
marital   0
education 0
default   0
balance   0
housing   0
loan       0
contact   0
day        0
month      0
duration   0
campaign   0
pdays     0
previous   0
poutcome   0
deposit    0

```

dtype: int64

```

df = pd.read_csv("bank.csv")
data = df.copy()

def remove_outliers_iqr_numerical(data_df):
    # Select only the numerical columns for IQR calculation
    numerical_data = data_df.select_dtypes(include=np.number)
    Q1 = numerical_data.quantile(0.25)
    Q3 = numerical_data.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # Create a boolean mask: True if a value in any numerical column is an outlier
    outlier_mask = ((numerical_data < lower_bound) | (numerical_data > upper_bound)).any(axis=1)
    # Filter the entire original DataFrame using the mask

```

```
data_cleaned = data_df[~outlier_mask]
return data_cleaned

cleaned_data = remove_outliers_iqr_numerical(data)
print("Original Data:")
print(len(data))
print("\nCleaned Data (without outliers):")
print(len(cleaned_data))
```

Original Data:
11162

Cleaned Data (without outliers):
6557

```
data['deposit'].value_counts()
```

	count
deposit	
no	5873
yes	5289

dtype: int64

```
#Convert categorical features to numerical data
label_encoders = {}
for column in data.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])
    label_encoders[column] = le
df.head()
```

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous
0	59	admin.	married	secondary	no	2343	yes	no	unknown	5	may	1042	1	-1	
1	56	admin.	married	secondary	no	45	no	no	unknown	5	may	1467	1	-1	
2	41	technician	married	secondary	no	1270	yes	no	unknown	5	may	1389	1	-1	
3	55	services	married	secondary	no	2476	yes	no	unknown	5	may	579	1	-1	
4	54	admin.	married	tertiary	no	184	no	no	unknown	5	may	673	2	-1	

Next steps: [Generate code with df](#) [New interactive sheet](#)

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 1. Define X and y (Using 'deposit' as your target)
X = data.drop('deposit', axis=1).values
y = data['deposit'].values

# 2. Split (70% Train, 30% Test - Page 4 of Lecture 5)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# 3. Standardize (Crucial for Logistic Regression, KNN, and SVM)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

print("Data successfully split and scaled!")
print(f"X_train shape: {X_train.shape}")
```

Data successfully split and scaled!
X_train shape: (7813, 16)

```
from sklearn.metrics import classification_report, accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

def evaluate_model(model_name, y_true, y_pred):
    print(f"\n--- {model_name} Evaluation ---")

    # Calculate Metrics
    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred, pos_label=1) # Assumes 1 is 'yes'
    rec = recall_score(y_true, y_pred, pos_label=1)
    f1 = f1_score(y_true, y_pred, pos_label=1)
```

```
# Create a results dictionary
results = {
    'Accuracy': acc,
    'Precision': prec,
    'Recall': rec,
    'F1-Score': f1,
    'Report': classification_report(y_true, y_pred)
}

return results
```

```
# Random Forest Classifier with grid search
from sklearn.model_selection import GridSearchCV
# Define the Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42)
# Define the hyperparameters for grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)
# Fit the grid search to the data
grid_search.fit(X_train, y_train)
# Get the best estimator from grid search
best_rf_classifier = grid_search.best_estimator_
# Make predictions using the best model
y_pred = best_rf_classifier.predict(X_test)

# Evaluate the model
evaluation_results = evaluate_model('RandomForestClassifier', y_test, y_pred)

# Print the best parameters ONCE
print("\n" + "="*30)
print("BEST HYPERPARAMETERS FOUND:")
print(grid_search.best_params_)
print("="*30)

# Print evaluation results
for key, value in evaluation_results.items():
    if key == 'Report': # Adjusted to match your function's key
        print(f"\n{key}: \n{value}")
    else:
        print(f"{key}: {value:.4f}")
```

```
--- RandomForestClassifier Evaluation ---

=====
BEST HYPERPARAMETERS FOUND:
{'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
=====
Accuracy: 0.8504
Precision: 0.8240
Recall: 0.8702
F1-Score: 0.8465

Report:

```

	precision	recall	f1-score	support
0	0.88	0.83	0.85	1762
1	0.82	0.87	0.85	1587
accuracy			0.85	3349
macro avg	0.85	0.85	0.85	3349
weighted avg	0.85	0.85	0.85	3349

```
#LogisticRegression with grid search
# Define the Logistic Regression model
LR = LogisticRegression(random_state=42)
# Define the hyperparameters for grid search
param_grid = {
    'C': [0.01, 0.1, 1, 10], # Regularization strength
    'solver': ['liblinear', 'lbfgs'], # Solver type
    'max_iter': [100, 200, 300] # Maximum iterations for convergence
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=LR, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)
```

```
# Fit the grid search to the data
grid_search.fit(X_train, y_train)
# Get the best estimator from grid search
best_LR = grid_search.best_estimator_
# Make predictions using the best model
y_pred = best_LR.predict(X_test)
# Evaluate the model
evaluation_results = evaluate_model('LogisticRegression', y_test, y_pred)

# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report': print(value) # Print report separately for better readability
    else: print(f"{key}: {value:.4f}" if isinstance(value, float) else f"{key}: \n{value}")

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)
```

```
--- LogisticRegression Evaluation ---
Accuracy: 0.8011
Precision: 0.8032
Recall: 0.7687
F1-Score: 0.7856
Report:
```

	precision	recall	f1-score	support
0	0.80	0.83	0.81	1762
1	0.80	0.77	0.79	1587
accuracy			0.80	3349
macro avg	0.80	0.80	0.80	3349
weighted avg	0.80	0.80	0.80	3349

```
Best hyperparameters found by GridSearchCV:
{'C': 1, 'max_iter': 100, 'solver': 'lbfgs'}
```

```
#DecisionTreeClassifier with grid search

# Define the Decision Tree model
DT = DecisionTreeClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'criterion': ['gini', 'entropy'], # The function to measure the quality of a split
    'max_depth': [None, 10, 20, 30], # The maximum depth of the tree
    'min_samples_split': [2, 5, 10], # The minimum number of samples required to split an internal node
    'min_samples_leaf': [1, 2, 4], # The minimum number of samples required to be at a leaf node
    'max_features': [None, 'sqrt', 'log2'] # The number of features to consider when looking for the best split
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=DT, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)
# Fit the grid search to the data
grid_search.fit(X_train, y_train)
# Get the best estimator from grid search
best_DT = grid_search.best_estimator_
# Make predictions using the best model
y_pred = best_DT.predict(X_test)
# Evaluate the model
evaluation_results = evaluate_model('DecisionTreeClassifier', y_test, y_pred)
# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report': print(value) # Print report separately for better readability
    else: print(f"{key}: {value:.4f}" if isinstance(value, float) else f"{key}: \n{value}")

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)
```

```
--- DecisionTreeClassifier Evaluation ---
Accuracy: 0.8196
Precision: 0.8236
Recall: 0.7883
F1-Score: 0.8055
Report:
```

	precision	recall	f1-score	support
0	0.82	0.85	0.83	1762
1	0.82	0.79	0.81	1587
accuracy			0.82	3349
macro avg	0.82	0.82	0.82	3349

```
weighted avg      0.82      0.82      0.82      3349
```

```
Best hyperparameters found by GridSearchCV:
{'criterion': 'entropy', 'max_depth': 10, 'max_features': None, 'min_samples_leaf': 2, 'min_samples_split': 5}
```

```
# DecisionTreeClassifier with grid search

# Define the Decision Tree model
DT = DecisionTreeClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [None, 'sqrt', 'log2']
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=DT, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best estimator from grid search
best_DT = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_DT.predict(X_test)

# Evaluate the model
evaluation_results = evaluate_model('DecisionTreeClassifier', y_test, y_pred)

# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report':
        print(value)
    else: print(f"{key}: {value:.4f}" if isinstance(value, float) else f"{key}: \n{value}")

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)
```

```
--- DecisionTreeClassifier Evaluation ---
Accuracy: 0.8196
Precision: 0.8236
Recall: 0.7883
F1-Score: 0.8055
Report:
```

	precision	recall	f1-score	support
0	0.82	0.85	0.83	1762
1	0.82	0.79	0.81	1587
accuracy			0.82	3349
macro avg	0.82	0.82	0.82	3349
weighted avg	0.82	0.82	0.82	3349

```
Best hyperparameters found by GridSearchCV:
{'criterion': 'entropy', 'max_depth': 10, 'max_features': None, 'min_samples_leaf': 2, 'min_samples_split': 5}
```

```
# KNeighborsClassifier with grid search

# Define the K-Nearest Neighbors model
knn = KNeighborsClassifier()

# Define the hyperparameters for grid search
param_grid = {
    'n_neighbors': [1, 3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']
}

grid_search = GridSearchCV(estimator=knn, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

grid_search.fit(X_train, y_train)

best_knn = grid_search.best_estimator_

y_pred = best_knn.predict(X_test)
```

```

evaluation_results = evaluate_model('K-Nearest Neighbors', y_test, y_pred)

# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report': print(value)
    else: print(f"{key}: {value:.4f}" if isinstance(value, float) else f"{key}: \n{value}")

print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)

```

--- K-Nearest Neighbors Evaluation ---

Accuracy: 0.7961
Precision: 0.8206
Recall: 0.7290
F1-Score: 0.7721
Report:

	precision	recall	f1-score	support
0	0.78	0.86	0.82	1762
1	0.82	0.73	0.77	1587
accuracy			0.80	3349
macro avg	0.80	0.79	0.79	3349
weighted avg	0.80	0.80	0.79	3349

Best hyperparameters found by GridSearchCV:
{'algorithm': 'auto', 'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'distance'}

GaussianNB with grid search

Define the Gaussian Naive Bayes model

gnb = GaussianNB()

param_grid = { 'var_smoothing': [1e-9, 1e-8, 1e-7, 1e-6] }

grid_search = GridSearchCV(estimator=gnb, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

grid_search.fit(X_train, y_train)

best_gnb = grid_search.best_estimator_

y_pred = best_gnb.predict(X_test)

evaluation_results = evaluate_model('Gaussian Naive Bayes', y_test, y_pred)

Print the evaluation results

```

for key, value in evaluation_results.items():
    if key == 'Classification Report': print(value)
    else: print(f"{key}: {value:.4f}" if isinstance(value, float) else f"{key}: \n{value}")

```

Print the best hyperparameters found by grid search

print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)

--- Gaussian Naive Bayes Evaluation ---

Accuracy: 0.7498

Best hyperparameters found by GridSearchCV:

{'var_smoothing': 1e-09}

Precision: 0.7039

Best hyperparameters found by GridSearchCV:

{'var_smoothing': 1e-09}

Recall: 0.8147

Best hyperparameters found by GridSearchCV:

{'var_smoothing': 1e-09}

F1-Score: 0.7553

Best hyperparameters found by GridSearchCV:

{'var_smoothing': 1e-09}

Report:

	precision	recall	f1-score	support
0	0.81	0.69	0.74	1762
1	0.70	0.81	0.76	1587
accuracy			0.75	3349
macro avg	0.75	0.75	0.75	3349
weighted avg	0.76	0.75	0.75	3349

```
Best hyperparameters found by GridSearchCV:
{'var_smoothing': 1e-09}
```

4. Use two feature selection methods to identify the most important features from the datasets.

```
# Method 1: Correlation Analysis (Statistical Selection)

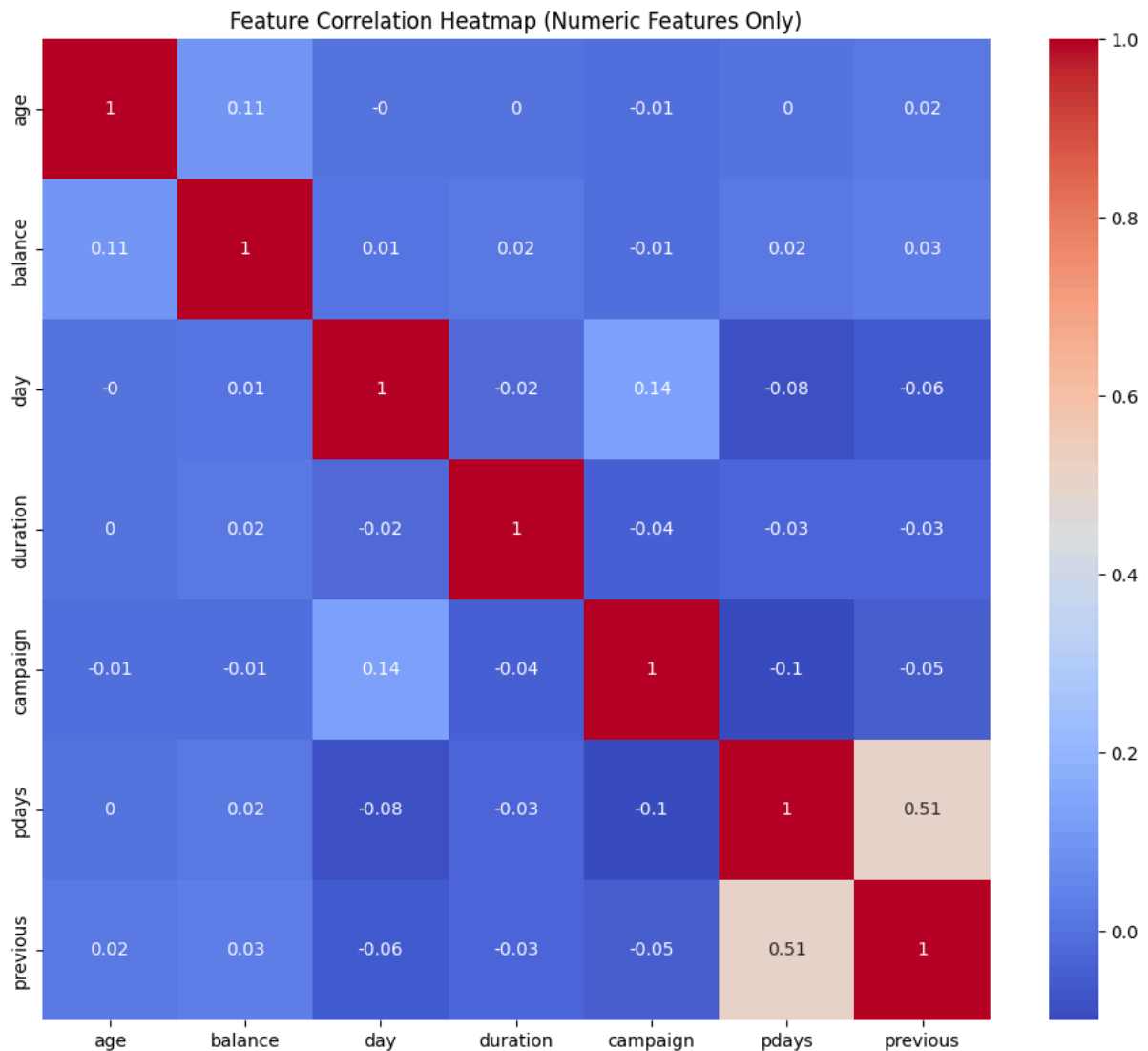
import seaborn as sns
import matplotlib.pyplot as plt

# 1. Select only the numeric columns (This avoids the 'admin.' error)
numeric_df = df.select_dtypes(include=['number'])

# 2. Calculate correlation matrix on only numeric data
corr = numeric_df.corr().round(2)

# 3. Generate the Heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(corr, cmap='coolwarm', annot=True)
plt.title('Feature Correlation Heatmap (Numeric Features Only)')
plt.show()

# 4. Print top features related to the target 'y'
# (Note: Ensure your target 'y' was converted to 0 and 1 earlier)
if 'y' in corr.columns:
    print("Top Features by Correlation to Target:")
    print(corr['y'].abs().sort_values(ascending=False))
```



```
# Method 2: Random Forest Feature Importance (Model-Based Selection)

import pandas as pd
import matplotlib.pyplot as plt

# Using the 'best_rf_classifier' you already trained
importances = best_rf_classifier.feature_importances_
```

```

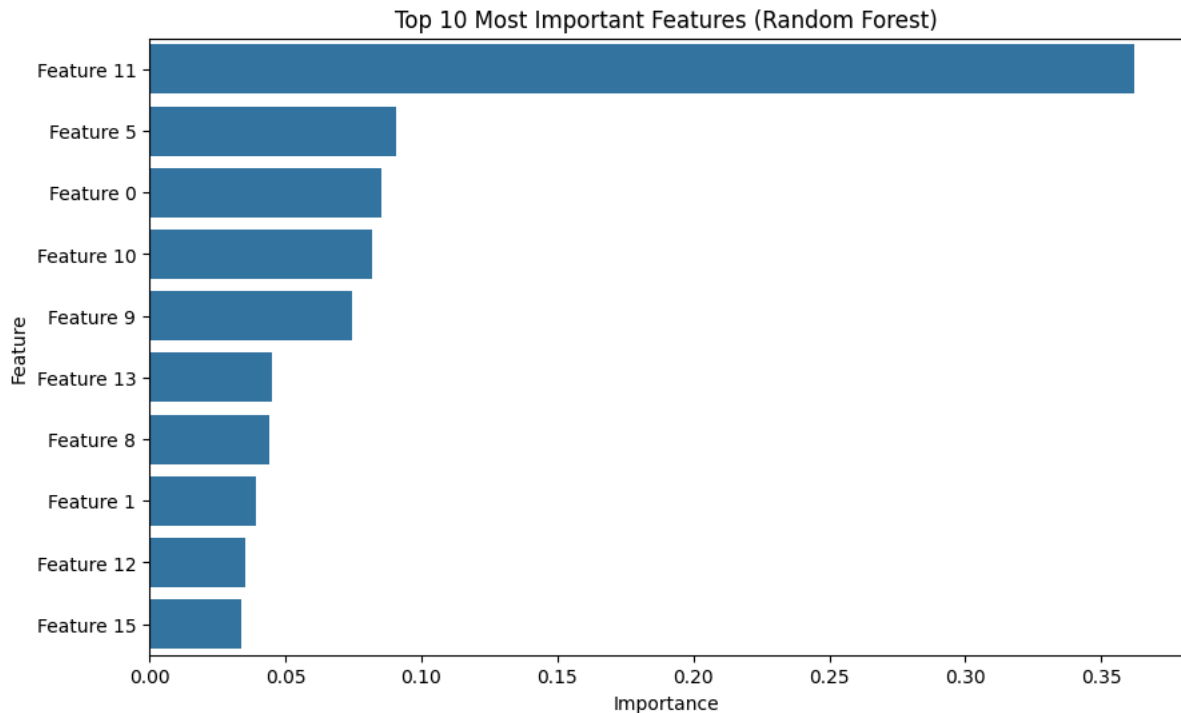
feature_names = X_train.columns if hasattr(X_train, 'columns') else [f"Feature {i}" for i in range(X_train.shape[1])]

# Create importance table
feature_importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
}).sort_values(by='Importance', ascending=False)

# Plotting
plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df.head(10))
plt.title('Top 10 Most Important Features (Random Forest)')
plt.show()

print(feature_importance_df.head(10))

```



	Feature	Importance
11	Feature 11	0.362124
5	Feature 5	0.090610
0	Feature 0	0.085313
10	Feature 10	0.081729
9	Feature 9	0.074509
13	Feature 13	0.045365
8	Feature 8	0.044195
1	Feature 1	0.039418
12	Feature 12	0.035382
15	Feature 15	0.033775

5. Apply machine learning models to the selected features and evaluate the models

```

from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

def evaluate_model(model_name, y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)

    results = {
        "MAE": mae,
        "MSE": mse,
        "RMSE": rmse,
        "R^2 Score": r2
    }

    # Plotting actual vs predicted (as shown in Lecture 6, Page 8)
    plt.figure(figsize=(8, 6))

```

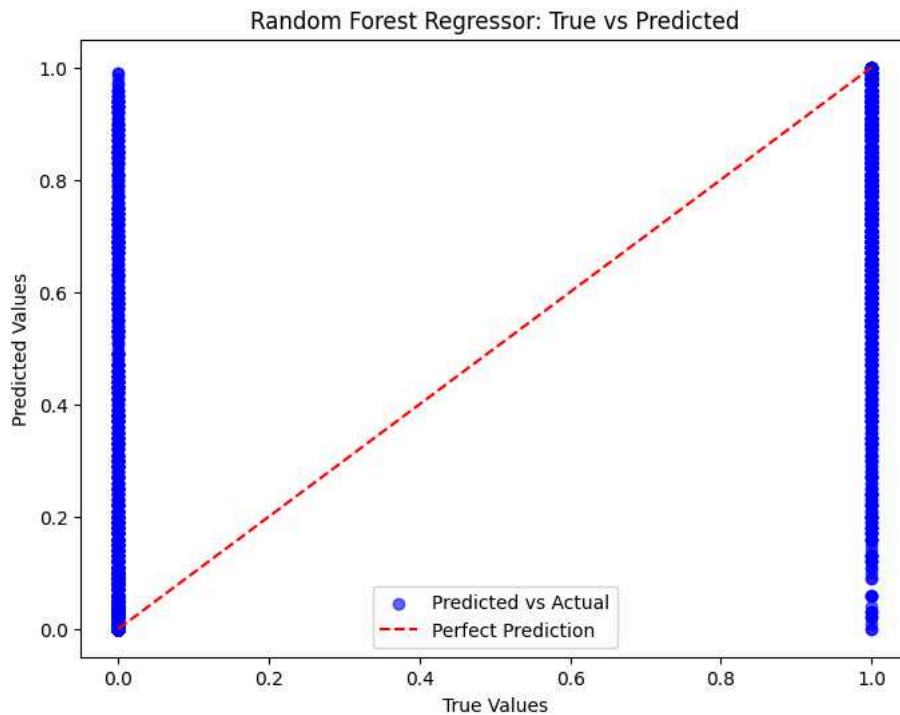
```
plt.scatter(y_true, y_pred, alpha=0.6, color='blue', label='Predicted vs Actual')
plt.plot([min(y_true), max(y_true)], [min(y_true), max(y_true)], 'r--', label='Perfect Prediction')
plt.title(f'{model_name}: True vs Predicted')
plt.xlabel('True Values')
plt.ylabel('Predicted Values')
plt.legend()
plt.show()

return results
```

Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor
```

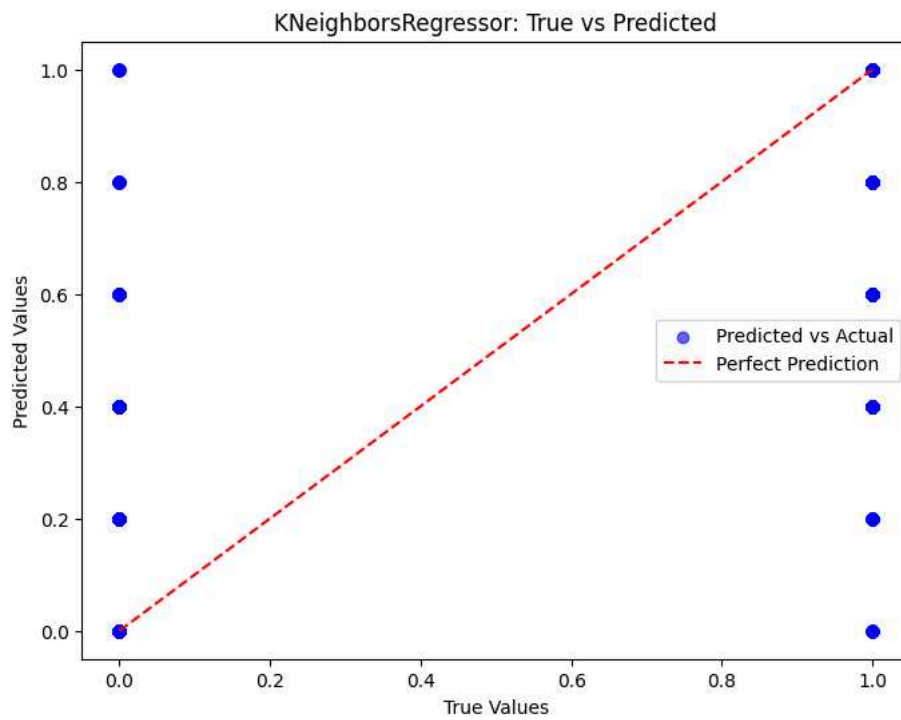
```
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
results_rf = evaluate_model('Random Forest Regressor', y_test, y_pred_rf)
print(results_rf)
```



```
{'MAE': 0.23342788892206628, 'MSE': 0.11446590026873693, 'RMSE': np.float64(0.3383280955947007), 'R^2 Score': 0.54088276887}
```

K-Nearest Neighbors Regressor

```
knn = KNeighborsRegressor()
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
results_knn = evaluate_model('KNeighborsRegressor', y_test, y_pred_knn)
print(results_knn)
```



{'MAE': 0.2755449387876978, 'MSE': 0.15753956404896985, 'RMSE': np.float64(0.39691253954614464), 'R^2 Score': 0.368116371168}

-
-
-
-
-

Thank you so much Dr. Hager and "جزاك الله خيراً"

-