DSC326 Final Project: Nora Zakrzewski & Katie Harper

Set-up

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
%cd "/content/drive/Shared drives/DSC326 Final"
```

```
/content/drive/Shared drives/DSC326 Final
```

```
pip install scikit-optimize
```

```
Collecting scikit-optimize
  Downloading scikit_optimize-0.10.2-py2.py3-none-any.whl.metadata (9.7 kB)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (1.4.2)
Collecting pyaml>=16.9 (from scikit-optimize)
  Downloading pyaml-25.1.0-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: numpy>=1.20.3 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (2.0.2)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (1.15.2)
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (1.6.1)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (24.2)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.11/dist-packages (from pyaml>=16.9->scikit-optimize) (6.0.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=1.0.0->scikit-o
Downloading scikit_optimize-0.10.2-py2.py3-none-any.whl (107 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 107.8/107.8 kB 2.9 MB/s eta 0:00:00
Downloading pyaml-25.1.0-py3-none-any.whl (26 kB)
Installing collected packages: pyaml, scikit-optimize
Successfully installed pyaml-25.1.0 scikit-optimize-0.10.2
```

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.preprocessing import MinMaxScaler, StandardScaler, LabelEncoder, OneHotEncoder, OrdinalEncoder
from itertools import product
from skopt import BayesSearchCV
from skopt.space import Real, Categorical

import statsmodels.api as sm
import statsmodels.formula.api as smf

from sklearn.svm import SVC, SVR
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor, GradientBoostingClassifier, GradientBoostingRegressor
from sklearn.neural_network import MLPClassifier, MLPRegressor

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

Part I

(1)

This dataset is related to heart disease prediction and is likely collected to help identify factors that contribute to the likelihood of a patient suffering from heart-related issues, specifically focusing on predicting whether a person will experience a cardiac death event within a certain timeframe. The goal of the classification problem is to predict whether a patient will have a heart-related death event based on various medical and demographic factors. The target variable, DEATH_EVENT, is categorical and binary, indicating whether the patient died (1) or survived (0). The model aims to predict this outcome based on the features provided, such as age, medical history, and lab results.

The features in the dataset can be divided into numerical and categorical variables. Numerical features include age, creatinine_phosphokinase, ejection_fraction, platelets, serum_creatinine, serum_sodium, and time, which represent various measurements or quantities. Categorical features include anaemia, diabetes, high_blood_pressure, sex, and smoking. Among the categorical features, all are

nominal. The dataset is likely imbalanced, with a higher proportion of patients surviving compared to those who experience a death event, making it important to address this during model training to avoid biased predictions.

(2)

```
# load dataset
df = pd.read_csv('heart_failure_clinical_records_dataset.csv')
df.head()
```

	age	anaemia	creatinine_phosphokinase	diabetes	ejection_fraction	high_blood_pressure	platelets	serum_creatinine	serum_sodi
0	75.0	0	582	0	20	1	265000.00	1.9	
1	55.0	0	7861	0	38	0	263358.03	1.1	
2	65.0	0	146	0	20	0	162000.00	1.3	
3	50.0	1	111	0	20	0	210000.00	1.9	
4	65.0	1	160	1	20	0	327000.00	2.7	

```
# Extract features and target variable
X = df.iloc[:, :12]  # Select the 12 features
y = df['DEATH_EVENT']


# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=23)
```

(3)

```
nominal_features = ['anaemia', 'diabetes', 'high_blood_pressure', 'sex', 'smoking']
numerical_features = ['age', 'creatinine_phosphokinase', 'ejection_fraction', 'platelets', 'serum_creatinine', 'serum_sodium', 'time'
#no ordinal features


# One-hot encode nominal variables using training data only
encoder = OneHotEncoder(handle_unknown="ignore")
X_train_nominal = encoder.fit_transform(X_train[nominal_features]).toarray()


# Standardize numerical features using training data only
scaler = StandardScaler()
X_train_numerical = scaler.fit_transform(X_train[numerical_features])


# Concatenate transformed numerical and categorical features
X_train_transformed = np.hstack((X_train_numerical, X_train_nominal))
X_train_transformed
```

```
array([[-0.48482655, -0.41620564,  0.24056486, ...,  1.        ,
         1.        ,  0.        ],
       [-0.23317081, -0.52492379,  0.06256176, ...,  1.        ,
         1.        ,  0.        ],
       [-0.90425279, -0.3371379 , -1.09445837, ...,  1.        ,
         0.        ,  1.        ],
       ...,
       [ 0.77345217, -0.00900678, -1.53946612, ...,  1.        ,
         0.        ,  1.        ],
       [-0.06540031, -0.42015903, -0.64945063, ...,  0.        ,
         1.        ,  0.        ],
       [ 1.52841941, -0.52986552,  1.13058034, ...,  1.        ,
         1.        ,  0.        ]])
```

(4)

K-Nearest Neighbors

```
# Perform 5-fold cross-validation to select the best k
k_values = range(1, 31)
accuracy_scores = []
```

```
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train_transformed, y_train, cv=5, scoring='accuracy')
    overall_accuracy = np.mean(scores)  # Compute mean accuracy
    accuracy_scores.append(overall_accuracy)


accuracy_scores
```

```
[np.float64(0.6984929078014185),
 np.float64(0.7446808510638296),
 np.float64(0.7696808510638298),
 np.float64(0.7281028368794327),
 np.float64(0.7530141843971631),
 np.float64(0.7448581560283689),
 np.float64(0.7529255319148935),
 np.float64(0.7406028368794326),
 np.float64(0.7404255319148936),
 np.float64(0.7322695035460993),
 np.float64(0.7404255319148936),
 np.float64(0.7531914893617021),
 np.float64(0.7574468085106383),
 np.float64(0.7531914893617022),
 np.float64(0.774113475177305),
 np.float64(0.7448581560283687),
 np.float64(0.7532801418439716),
 np.float64(0.7490248226950355),
 np.float64(0.7531914893617022),
 np.float64(0.7447695035460994),
 np.float64(0.7531914893617021),
 np.float64(0.7281914893617021),
 np.float64(0.7366134751773049),
 np.float64(0.7282801418439717),
 np.float64(0.7323581560283688),
 np.float64(0.7323581560283688),
 np.float64(0.7323581560283688),
 np.float64(0.7407801418439717),
 np.float64(0.7449468085106383),
 np.float64(0.7407801418439717)]
```

```
# Find the best k with the highest accuracy
best_k = k_values[np.argmax(accuracy_scores)]
best_accuracy = max(accuracy_scores)

# Print results
print(f"Best k: {best_k}")
print(f"Best Accuracy from cross-validation: {best_accuracy:.4f}")
```

```
Best k: 15
Best Accuracy from cross-validation: 0.7741
```

SVC

```
# Define hyperparameter grid
kernel_func_options = ['linear', 'poly', 'rbf']
C_val_options = [0.001, 0.01, 0.1, 1]

# Store results
results = []

# Combinations of hyperparameters
prodvalues = product(kernel_func_options, C_val_options)
print(list(prodvalues))
```

```
[('linear', 0.001), ('linear', 0.01), ('linear', 0.1), ('linear', 1), ('poly', 0.001), ('poly', 0.01), ('poly', 0.1), ('poly', 1),
```

```
# Perform 5-fold cross-validation for each combination of hyperparameters
for kernel_func, C_val in product(kernel_func_options, C_val_options):
    svc_model = SVC(kernel=kernel_func, C=C_val, random_state=23)

    # Compute accuracy using 5-fold cross-validation on the training set
    cv_scores = cross_val_score(svc_model, X_train_transformed, y_train, cv=5, scoring='accuracy')

    # Compute overall accuracy from cross-validation
    mean_accuracy = np.mean(cv_scores)

    results.append((kernel_func, C_val, mean_accuracy))

    # print kernel_func, C_val and mean_accuracy
    print(f'kernel = {kernel_func}, C = {C_val}: mean accuracy = {mean_accuracy}')
```

```
kernel = linear, C = 0.001: mean accuracy = 0.6736702127659574
kernel = linear, C = 0.01: mean accuracy = 0.7659574468085106
kernel = linear, C = 0.1: mean accuracy = 0.8118794326241134
kernel = linear, C = 1: mean accuracy = 0.8035460992907802
kernel = poly, C = 0.001: mean accuracy = 0.6736702127659574
kernel = poly, C = 0.01: mean accuracy = 0.677836879432624
kernel = poly, C = 0.1: mean accuracy = 0.7156028368794327
kernel = poly, C = 1: mean accuracy = 0.7782801418439715
kernel = rbf, C = 0.001: mean accuracy = 0.6736702127659574
kernel = rbf, C = 0.01: mean accuracy = 0.6736702127659574
kernel = rbf, C = 0.1: mean accuracy = 0.6736702127659574
kernel = rbf, C = 1: mean accuracy = 0.8115248226950355
```

```
results_df = pd.DataFrame(results, columns=['kernel function', 'C', 'Accuracy'])
results_df
```

	kernel function	C	Accuracy
0	linear	0.001	0.673670
1	linear	0.010	0.765957
2	linear	0.100	0.811879
3	linear	1.000	0.803546
4	poly	0.001	0.673670
5	poly	0.010	0.677837
6	poly	0.100	0.715603
7	poly	1.000	0.778280
8	rbf	0.001	0.673670
9	rbf	0.010	0.673670
10	rbf	0.100	0.673670
11	rbf	1.000	0.811525

```
# Select the best hyperparameter setting
best_kernel_func, best_C_val, best_acc = max(results, key=lambda x: x[2])
print(best_kernel_func, best_C_val, best_acc)
```

```
linear 0.1 0.8118794326241134
```

Random Forest

```
# Define hyperparameter ranges
n_estimators_options = [10, 25, 40]
max_depth_options = [1, 3, 5]

results_rf = []


for n_estimators, max_depth in product(n_estimators_options, max_depth_options):
    rf_model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth, random_state=23)

    cv_scores = cross_val_score(rf_model, X_train_transformed, y_train, cv=5, scoring='accuracy')
    mean_accuracy = np.mean(cv_scores)

    results_rf.append((n_estimators, max_depth, mean_accuracy))

    print(f'n_estimators = {n_estimators}, max_depth = {max_depth}: mean accuracy = {mean_accuracy:.4f}')
```

```
n_estimators = 10, max_depth = 1: mean accuracy = 0.6906
n_estimators = 10, max_depth = 3: mean accuracy = 0.8200
n_estimators = 10, max_depth = 5: mean accuracy = 0.7947
n_estimators = 25, max_depth = 1: mean accuracy = 0.7156
n_estimators = 25, max_depth = 3: mean accuracy = 0.8368
n_estimators = 25, max_depth = 5: mean accuracy = 0.8366
n_estimators = 40, max_depth = 1: mean accuracy = 0.7491
n_estimators = 40, max_depth = 3: mean accuracy = 0.8410
n_estimators = 40, max_depth = 5: mean accuracy = 0.8407
```

```
best_rf = max(results_rf, key=lambda x: x[2])
print(f"\nBest Random Forest: n_estimators = {best_rf[0]}, max_depth = {best_rf[1]}, accuracy = {best_rf[2]:.4f}")
```

```

Best Random Forest: n_estimators = 40, max_depth = 3, accuracy = 0.8410
```

MLPClassifier

```
# Define hyperparameter ranges
hidden_layer_options = [(5,), (10,), (5,5,)]
alpha_options = [0.01, 0.1, 1.0]
activation_options = ['relu', 'tanh']

results = []


for hidden_layers, alpha, activation in product(hidden_layer_options, alpha_options, activation_options):
    mlp_model = MLPClassifier(
        hidden_layer_sizes=hidden_layers,
        alpha=alpha,
        activation=activation,
        max_iter=1700,
        early_stopping=True,
        random_state=23
    )


# 5-fold cross-validation
cv_scores = cross_val_score(mlp_model, X_train_transformed, y_train, cv=5, scoring='accuracy')
mean_accuracy = np.mean(cv_scores)

results.append((hidden_layers, alpha, activation, mean_accuracy))

# Print results
print(f'hidden_layers = {hidden_layers}, alpha = {alpha}, activation = {activation}: mean accuracy = {mean_accuracy:.4f}')
```

```
hidden_layers = (5, 5), alpha = 1.0, activation = tanh: mean accuracy = 0.6988
```

```
# Find best hyperparameters
best_config = max(results, key=lambda x: x[3])
print("\nBest MLP configuration:")
print(f"hidden_layers = {best_config[0]}, alpha = {best_config[1]}, activation = {best_config[2]}, accuracy = {best_config[3]:.4f}")
```

```
Best MLP configuration:
hidden_layers = (5, 5), alpha = 1.0, activation = tanh, accuracy = 0.6988
```

Logistic Regression

```
# Define hyperparameter range
C_val_options = [0.0001, 0.001, 0.01, 0.1]
penalty_options = ['l2']

results_logreg = []


for C_val, penalty in product(C_val_options, penalty_options):
    logreg_model = LogisticRegression(C=C_val, penalty=penalty, solver='liblinear', max_iter=1000, random_state=23)

    cv_scores = cross_val_score(logreg_model, X_train_transformed, y_train, cv=5, scoring='accuracy')
    mean_accuracy = np.mean(cv_scores)

    results_logreg.append((C_val, penalty, mean_accuracy))

    print(f'C = {C_val}, penalty = {penalty}: mean accuracy = {mean_accuracy:.4f}')
```

```
C = 0.0001, penalty = l2: mean accuracy = 0.7366
C = 0.001, penalty = l2: mean accuracy = 0.7492
C = 0.01, penalty = l2: mean accuracy = 0.8077
C = 0.1, penalty = l2: mean accuracy = 0.8285
```

```
best_logreg = max(results_logreg, key=lambda x: x[2])
print(f"\nBest Logistic Regression: C = {best_logreg[0]}, penalty = {best_logreg[1]}, accuracy = {best_logreg[2]:.4f}")
```

```
Best Logistic Regression: C = 0.1, penalty = l2, accuracy = 0.8285
```

(5)

```
final_results = []
```

```
final_results.clear()
```

K-nn

```
# Train and evaluate the model with the best k
knn_best = KNeighborsClassifier(n_neighbors=best_k)
knn_best.fit(X_train_transformed, y_train)
X_train_numerical = scaler.transform(X_train[numerical_features])
X_train_nominal = encoder.transform(X_train[nominal_features]).toarray()
X_train_transformed = np.hstack((X_train_numerical, X_train_nominal))
y_pred_train = knn_best.predict(X_train_transformed)
X_test_numerical = scaler.transform(X_test[numerical_features])
X_test_nominal = encoder.transform(X_test[nominal_features]).toarray()
X_test_transformed = np.hstack((X_test_numerical, X_test_nominal))
y_pred_test = knn_best.predict(X_test_transformed)

train_accuracy = accuracy_score(y_train, y_pred_train)
test_accuracy = accuracy_score(y_test, y_pred_test)
final_results.append(("K-NN", train_accuracy, test_accuracy))

print(f'Train Accuracy with best k: {train_accuracy:.4f}')
print(f"Test Accuracy with best k: {test_accuracy:.4f}")
```

```
Train Accuracy with best k: 0.7866
Test Accuracy with best k: 0.7667
```

SVC

```
# Train final model on the full training set with the best hyperparameters
final_svc_model = SVC(kernel=best_kernel_func, C=best_C_val, random_state=23, class_weight='balanced')
final_svc_model.fit(X_train_transformed, y_train)
```

```
SVC(C=0.1, class_weight='balanced', kernel='linear', random_state=23)
```

```
# Calculate accuracy
train_accuracy = final_svc_model.score(X_train_transformed, y_train)
test_accuracy = final_svc_model.score(X_test_transformed, y_test)
final_results.append(("SVC", train_accuracy, test_accuracy))

print("Training Accuracy with best C:", train_accuracy)
print("Test Accuracy with best C:", test_accuracy)
```

```
Training Accuracy with best C: 0.8493723849372385
Test Accuracy with best C: 0.7833333333333333
```

Random Forest

```
rf_params = best_rf  # (n_estimators, max_depth)
rf = RandomForestClassifier(n_estimators=rf_params[0], max_depth=rf_params[1], random_state=23, class_weight='balanced')
rf.fit(X_train_transformed, y_train)
rf_train_acc = accuracy_score(y_train, rf.predict(X_train_transformed))
rf_test_acc = accuracy_score(y_test, rf.predict(X_test_transformed))
final_results.append(("Random Forest", rf_train_acc, rf_test_acc))

print("Training Accuracy with best Random Forest:", rf_train_acc)
print("Test Accuracy with best Random Forest:", rf_test_acc)
```

```
Training Accuracy with best Random Forest: 0.9288702928870293
Test Accuracy with best Random Forest: 0.8
```

MLPClassification

```
mlp_params = best_config  # (hidden_layers, alpha, activation, accuracy)
mlp = MLPClassifier(
    hidden_layer_sizes=mlp_params[0],
    alpha=mlp_params[1],
    activation=mlp_params[2],
    max_iter=1700,
    random_state=23
)
```

```
mlp.fit(X_train_transformed, y_train)
mlp_train_acc = accuracy_score(y_train, mlp.predict(X_train_transformed))
mlp_test_acc = accuracy_score(y_test, mlp.predict(X_test_transformed))
final_results.append(("MLP Classifier", mlp_train_acc, mlp_test_acc))

print("Training Accuracy with best MLP:", mlp_train_acc)
print("Test Accuracy with best MLP:", mlp_test_acc)
```

```
Training Accuracy with best MLP: 0.8619246861924686
Test Accuracy with best MLP: 0.75
```

Logistic Regression

```
logreg_C = best_logreg[0]
logreg = LogisticRegression(C=logreg_C, penalty='l2', solver='liblinear', max_iter=1000, random_state=23, class_weight='balanced')
logreg.fit(X_train_transformed, y_train)
logreg_train_acc = accuracy_score(y_train, logreg.predict(X_train_transformed))
logreg_test_acc = accuracy_score(y_test, logreg.predict(X_test_transformed))
final_results.append(("Logistic Regression", logreg_train_acc, logreg_test_acc))

print("Training Accuracy with best Logistic Regression:", logreg_train_acc)
print("Test Accuracy with best Logistic Regression:", logreg_test_acc)
```

```
Training Accuracy with best Logistic Regression: 0.8326359832635983
Test Accuracy with best Logistic Regression: 0.7833333333333333
```

Final Results

```
# Create DataFrame and sort by test accuracy
results_df = pd.DataFrame(final_results, columns=["Model", "Train Accuracy", "Test Accuracy"])
best_per_model = results_df.loc[results_df.groupby("Model")["Test Accuracy"].idxmax()]
best_per_model_sorted = best_per_model.sort_values(by="Test Accuracy", ascending=False).reset_index(drop=True)

# Display
print("\nFinal Results:")
print(best_per_model_sorted.to_string(index=False))
```

```
Final Results:
               Model  Train Accuracy  Test Accuracy
       Random Forest        0.928870       0.800000
                 SVC        0.849372       0.783333
Logistic Regression        0.832636       0.783333
               K-NN        0.786611       0.766667
     MLP Classifier        0.861925       0.750000
```

Random Forest had the highest test accuracy (80%) and a good balance between train and test accuracy.

(6)

This dataset shows that tree-based methods like Random Forest excel due to their ability to capture non-linear relationships. However, linear models (SVC and Logistic Regression) perform nearly as well, suggesting a mix of linear and non-linear patterns. K-NN struggled, likely due to high-dimensionality or noisy features, while the MLP Classifier showed good training accuracy but had difficulty generalizing to the test data, suggesting an unnecessary level of complexity for this dataset. This indicates that simpler models such as Random Forest, SVC, and Logistic Regression are more suitable for this data, while more complex models like MLP may need further tuning or might not provide additional value in this case.

Part II

(1)

The "LasVegasTripAdvisorReviews" dataset contains information about 504 online TripAdvisor reviews from 21 hotels in the Las Vegas Strip. The target variable, Score, is ordinal, with a score of 5 being the highest rating and a score of 1 being the lowest rating. The goal of the regression problem is to predict the score that a hotel receives from a review, based on features such as the characteristics of the reviewer, when the review was posted, and whether or not the hotel offers certain amenities.

The features in the dataset can be divided into numerical and categorical variables. Numerical features include the number of reviews a user has left, number of hotel reviews, number of helpful votes, number of rooms, and member years. The nominal categorical features include user country, period of stay, traveler type, hotel name, review month, review weekday, and whether the hotel has a pool, gym, tennis court, spa, and free internet. The final feature is hotel stars, which is ordinal since the number of stars is ranked with 5 being the best and 1 being the worst, though the data only goes down to 3 stars.

(2)

```
# load dataset
df = pd.read_csv('LasVegasTripAdvisorReviews.csv', sep=';')
df.head()
```

	User country	Nr. reviews	Nr. hotel reviews	Helpful votes	Score	Period of stay	Traveler type	Pool	Gym	Tennis court	Spa	Casino	Free internet	Hotel name	Hotel stars	Nr. rooms	Us contine
0	USA	11	4	13	5	Dec-Feb	Friends	NO	YES	NO	NO	YES	YES	Circus Circus Hotel & Casino Las Vegas	3	3773	No Amer
1	USA	119	21	75	3	Dec-Feb	Business	NO	YES	NO	NO	YES	YES	Circus Circus Hotel & Casino Las Vegas	3	3773	No Amer
2	USA	36	9	25	5	Mar-May	Families	NO	YES	NO	NO	YES	YES	Circus Circus Hotel & Casino Las Vegas	3	3773	No Amer
3	UK	14	7	14	4	Mar-May	Friends	NO	YES	NO	NO	YES	YES	Circus Circus Hotel & Casino Las Vegas	3	3773	Euro
4	Canada	5	5	2	4	Mar-May	Solo	NO	YES	NO	NO	YES	YES	Circus Circus Hotel & Casino Las Vegas	3	3773	No Amer

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 504 entries, 0 to 503
Data columns (total 20 columns):
 #   Column            Non-Null Count  Dtype 
---  ------            --------------  ----- 
 0   User country      504 non-null    object
 1   Nr. reviews       504 non-null    int64 
 2   Nr. hotel reviews 504 non-null    int64 
 3   Helpful votes     504 non-null    int64 
 4   Score             504 non-null    int64 
 5   Period of stay    504 non-null    object
 6   Traveler type     504 non-null    object
 7   Pool              504 non-null    object
 8   Gym               504 non-null    object
 9   Tennis court      504 non-null    object
 10  Spa               504 non-null    object
 11  Casino            504 non-null    object
 12  Free internet     504 non-null    object
 13  Hotel name        504 non-null    object
 14  Hotel stars       504 non-null    object
 15  Nr. rooms         504 non-null    int64 
 16  User continent    504 non-null    object
```

```
 17  Member years       504 non-null    int64
 18  Review month       504 non-null    object
 19  Review weekday     504 non-null    object
dtypes: int64(6), object(14)
memory usage: 78.9+ KB
```

```
# Split features and target
X = df.drop(columns=['Score'])
y = df['Score']


# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=23)
```

(3)

```
numerical_features = ['Nr. reviews', 'Nr. hotel reviews', 'Helpful votes', 'Nr. rooms', 'Member years']
nominal_features = ['User country', 'Period of stay', 'Traveler type', 'Pool', 'Gym', 'Tennis court', 'Spa', 'Free internet', 'Hotel
ordinal_features = ['Hotel stars']


# One-hot encode nominal variables using training data only
encoder = OneHotEncoder(drop="first", sparse_output=False, handle_unknown="ignore")
X_train_nominal = encoder.fit_transform(X_train[nominal_features])


# Ordinal encode ordinal variables using training data only
ord_encoder = OrdinalEncoder()
X_train_ordinal = ord_encoder.fit_transform(X_train[ordinal_features])


# Standardize numerical features using training data only
scaler = StandardScaler()
X_train_numerical = scaler.fit_transform(X_train[numerical_features])


# Concatenate transformed numerical and categorical features
X_train_transformed = np.hstack((X_train_numerical, X_train_nominal, X_train_ordinal))
X_train_transformed
```

```
array([[-0.47542577, -0.24909591, -0.33201556, ...,  0.        ,
         1.        ,  2.        ],
       [ 0.22344684,  0.65351159,  0.54263901, ...,  0.        ,
         0.        ,  4.        ],
       [-0.55454343, -0.48455874, -0.49104366, ...,  0.        ,
         0.        ,  4.        ],
       ...,
       [-0.26444536, -0.32758352, -0.23262299, ...,  0.        ,
         1.        ,  0.        ],
       [-0.60728853, -0.52380254, -0.61031474, ...,  0.        ,
         0.        ,  4.        ],
       [-0.5677297 , -0.44531493, -0.5506792 , ...,  1.        ,
         0.        ,  2.        ]])
```

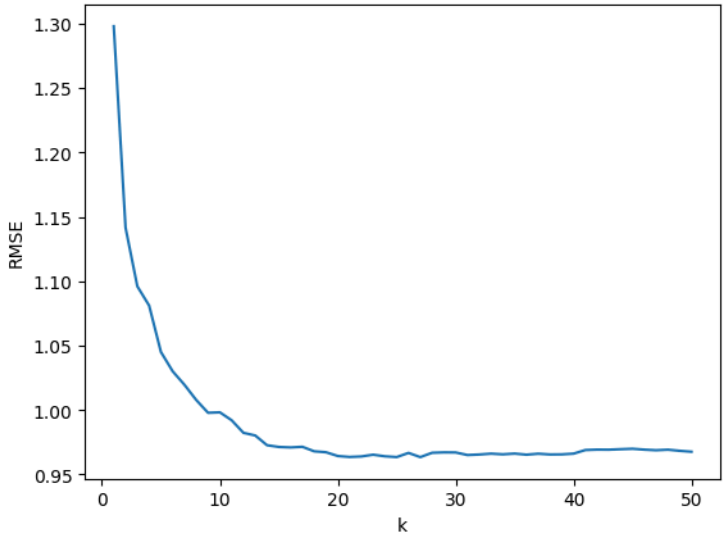
(4)

K-nearest neighbors

```
# Perform 10-fold cross-validation to select the best k
k_values = range(1, 51)
rmse_scores = []

for k in k_values:
    knn = KNeighborsRegressor(n_neighbors=k)
    scores = cross_val_score(knn, X_train_transformed, y_train, cv=10, scoring='neg_root_mean_squared_error')
    rmse_scores.append(-np.mean(scores))  # Convert negative RMSE to positive

# Find the best k with the lowest RMSE
best_k = k_values[np.argmin(rmse_scores)]
best_rmse = min(rmse_scores)
print(f"Best k: {best_k}")
print(f"Best RMSE from cross-validation: {best_rmse:.4f}")

# Ensure rmse_scores and k_values have the same length before plotting
# This assumes rmse_scores has fewer elements than k_values
k_values = k_values[:len(rmse_scores)]
```

```
plt.plot(k_values, rmse_scores)
plt.xlabel('k')
plt.ylabel('RMSE')
plt.show()
```

```
Best k: 27
Best RMSE from cross-validation: 0.9635
```




```
rmse_scores
```

```
[np.float64(1.2979177210828619),
 np.float64(1.1415822518972778),
 np.float64(1.0960428971114462),
 np.float64(1.0811236894398373),
 np.float64(1.0451108573977448),
 np.float64(1.0300656455845698),
 np.float64(1.019595298741965),
 np.float64(1.007776492691177),
 np.float64(0.9979029458306503),
 np.float64(0.9982613396284024),
 np.float64(0.9921061844946315),
 np.float64(0.9823538653620518),
 np.float64(0.9802917661027302),
 np.float64(0.9727365452621706),
 np.float64(0.9713868166391574),
 np.float64(0.9710269913099145),
 np.float64(0.9715118983822162),
 np.float64(0.9680099048661445),
 np.float64(0.9672981957793642),
 np.float64(0.9643627938067049),
 np.float64(0.9636314661893893),
 np.float64(0.964020922426071),
 np.float64(0.9653735262305843),
 np.float64(0.9641364988067627),
 np.float64(0.9635373653266548),
 np.float64(0.9667580814956477),
 np.float64(0.9634954979597753),
 np.float64(0.9668666622802833),
 np.float64(0.967180810966075),
 np.float64(0.9671143437296366),
 np.float64(0.9651281294374738),
 np.float64(0.9655274273712833),
 np.float64(0.9661948694988233),
 np.float64(0.9656679880142756),
 np.float64(0.9662452619822085),
 np.float64(0.9653974603580278),
 np.float64(0.9661462983373669),
 np.float64(0.9655670968152548),
 np.float64(0.9656361391413597),
 np.float64(0.966209350662016),
 np.float64(0.9690047701278404),
 np.float64(0.9693192372803523),
 np.float64(0.9692693969885362),
 np.float64(0.9696532383112529),
 np.float64(0.969990066042004),
 np.float64(0.9693063599154697),
 np.float64(0.9688400393331953),
 np.float64(0.9692659819793257),
 np.float64(0.9683901546270259),
 np.float64(0.9676819768041753)]
```

Multilayer Perceptron Regressor

```
# Define hyperparameter grid
hidden_layer_options = [1, 2]
hidden_nodes_options = [5, 10, 20]
activation_functions = ['logistic', 'relu']

# Store results
results = []


# Combinations of hyperparameters
prodvalues = product(hidden_layer_options, hidden_nodes_options, activation_functions)
print(list(prodvalues))
```

```
[(1, 5, 'logistic'), (1, 5, 'relu'), (1, 10, 'logistic'), (1, 10, 'relu'), (1, 20, 'logistic'), (1, 20, 'relu'), (2, 5, 'logistic'
```

```
# Perform 5-fold cross-validation for each combination of hyperparameters
for layers, nodes, activation in product(hidden_layer_options, hidden_nodes_options, activation_functions):
    hidden_layer_sizes = tuple([nodes] * layers)
    mlp = MLPRegressor(hidden_layer_sizes=hidden_layer_sizes, activation=activation, random_state=23, max_iter=1000)

    # Compute RMSE using cross-validation
    scores = cross_val_score(mlp, X_train_transformed, y_train, cv=5, scoring='neg_root_mean_squared_error')
    mean_rmse = -np.mean(scores)

    results.append((hidden_layer_sizes, activation, mean_rmse))
```

Show hidden output

```
# Select the best hyperparameter setting
best_hidden_layer_sizes, best_activation, best_rmse = min(results, key=lambda x: x[2])
print(best_hidden_layer_sizes, best_activation, best_rmse)
```

```
(5,) logistic 0.9868475087431486
```

```
# Convert results to DataFrame for better visualization
results_df = pd.DataFrame(results, columns=["Hidden Layer Sizes", "Activation Function", "Cross-Validation RMSE"])
results_df
```

Show hidden output

Random Forest

```
# Define hyperparameter grid
num_trees_options = [100, 200]
max_depth_options = [10, 20]
min_samples_leaf_options = [5, 10]

# Store results
results = []

# Combinations of hyperparameters
prodvalues = product(num_trees_options, max_depth_options, min_samples_leaf_options)
print(list(prodvalues))
```

```
[(100, 10, 5), (100, 10, 10), (100, 20, 5), (100, 20, 10), (200, 10, 5), (200, 10, 10), (200, 20, 5), (200, 20, 10)]
```

```
# Perform 5-fold cross-validation for each combination of hyperparameters
for num_trees, max_depth, min_samples_leaf in product(num_trees_options, max_depth_options, min_samples_leaf_options):
    rf_pruned_reg = RandomForestRegressor(n_estimators=num_trees, max_depth=max_depth, min_samples_leaf=min_samples_leaf,\
                                          random_state=23)

    # Compute RMSE using cross-validation
    rmse_scores = cross_val_score(rf_pruned_reg, X_train_transformed, y_train, cv=5, scoring='neg_mean_squared_error')
    mean_rmse = np.sqrt(-np.mean(rmse_scores))  # Correct overall RMSE calculation

    results.append((num_trees, max_depth, min_samples_leaf, mean_rmse))
    print(f"num_trees: {num_trees}, max_depth: {max_depth}, min_samples_leaf: {min_samples_leaf}, RMSE: {mean_rmse}")
```

```
num_trees: 100, max_depth: 10, min_samples_leaf: 5, RMSE: 0.9752526122536477
num_trees: 100, max_depth: 10, min_samples_leaf: 10, RMSE: 0.9705427192406126
num_trees: 100, max_depth: 20, min_samples_leaf: 5, RMSE: 0.9759284730014615
num_trees: 100, max_depth: 20, min_samples_leaf: 10, RMSE: 0.9705754617024587
num_trees: 200, max_depth: 10, min_samples_leaf: 5, RMSE: 0.9761719704789232
num_trees: 200, max_depth: 10, min_samples_leaf: 10, RMSE: 0.9709170932134014
```

```
num_trees: 200, max_depth: 20, min_samples_leaf: 5, RMSE: 0.9768649104140984
num_trees: 200, max_depth: 20, min_samples_leaf: 10, RMSE: 0.97093533254901
```

```
results_df = pd.DataFrame(results, columns=["Number of Trees", "Max Depth", "Min Samples Leaf", "Cross-Validation RMSE"])
results_df
```

	Number of Trees	Max Depth	Min Samples Leaf	Cross-Validation RMSE
0	100	10	5	0.975253
1	100	10	10	0.970543
2	100	20	5	0.975928
3	100	20	10	0.970575
4	200	10	5	0.976172
5	200	10	10	0.970917
6	200	20	5	0.976865
7	200	20	10	0.970935

```
# Select the best hyperparameter setting
best_num_trees, best_max_depth, best_min_samples_leaf, best_rmse = min(results, key=lambda x: x[3])
print(best_num_trees, best_max_depth, best_min_samples_leaf, best_rmse)
```

```
100 10 10 0.9705427192406126
```

Support Vector Regressor

```
# scale feature and target variable in the training data
scaler_y = StandardScaler()
y_train_scaled = scaler_y.fit_transform(y_train.values.reshape(-1, 1))
y_train_scaled = y_train_scaled.ravel()
```

```
# Set up SVR and hyperparameter grid
svr = SVR()
param_grid = {
    'C': [0.1, 1, 10, 50, 100],
    'epsilon': [0.1, 0.5, 1],
    'kernel': ['linear','poly','rbf']
}
```

```
# Perform GridSearchCV
grid_search = GridSearchCV(svr, param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=-1)
grid_search.fit(X_train_transformed, y_train_scaled)
```

```
▸ GridSearchCV
▸ best_estimator_:
      SVR
    ▸ SVR
```

```
# Best model
best_model = grid_search.best_estimator_
print(best_model)
```

```
SVR(C=0.1, epsilon=0.5, kernel='linear')
```

(5)

```
# Test data
X_test_nominal = encoder.transform(X_test[nominal_features])
X_test_ordinal = ord_encoder.transform(X_test[ordinal_features])
X_test_numerical = scaler.transform(X_test[numerical_features])
X_test_transformed = np.hstack((X_test_numerical, X_test_nominal, X_test_ordinal))
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/preprocessing/_encoders.py:246: UserWarning: Found unknown categories in columns
  warnings.warn(
```

```
final_results = []
final_results.clear()
```

K-nearest neighbors

```
# Train the model with the best k
knn_best = KNeighborsRegressor(n_neighbors=best_k)
knn_best.fit(X_train_transformed, y_train)

# Evaluate the model with the best k
y_pred_train = knn_best.predict(X_train_transformed)
train_rmse = root_mean_squared_error(y_train, y_pred_train)
y_pred_test = knn_best.predict(X_test_transformed)
test_rmse = root_mean_squared_error(y_test, y_pred_test)
final_results.append(("K-NN", train_rmse, test_rmse))

# Print results
print(f"Best k: {best_k}")
print(f"Best RMSE from cross-validation: {best_rmse:.4f}")
print(f"Test RMSE with best k: {test_rmse:.4f}")
```

```
Best k: 27
Best RMSE from cross-validation: 0.9705
Test RMSE with best k: 1.0147
```

Multilayer Perceptron Regressor

```
# Train final model on the full training set with the best hyperparameters
final_model = MLPRegressor(hidden_layer_sizes=best_hidden_layer_sizes, activation=best_activation, random_state=42, max_iter=1000)
final_model.fit(X_train_transformed, y_train)

# Evaluate on training and test set
y_pred_train = final_model.predict(X_train_transformed)
train_rmse = root_mean_squared_error(y_train, y_pred_train)
y_pred_test = final_model.predict(X_test_transformed)
test_rmse = root_mean_squared_error(y_test, y_pred_test)
final_results.append(("MLP", train_rmse, test_rmse))

print("Test RMSE with best MLP:", test_rmse)
```

```
Test RMSE with best MLP: 0.9967814644584271
```

Random Forest

```
# Train final model on the full training set with the best hyperparameters
final_model = RandomForestRegressor(n_estimators=best_num_trees, max_depth=best_max_depth, min_samples_leaf=best_min_samples_leaf,\
                                    random_state=23)
final_model.fit(X_train_transformed, y_train)
```

```
RandomForestRegressor
RandomForestRegressor(max_depth=10, min_samples_leaf=10, random_state=23)
```

```
# Predict on training set
y_train_pred = final_model.predict(X_train_transformed)
train_rmse = root_mean_squared_error(y_train, y_train_pred)
y_test_pred = final_model.predict(X_test_transformed)
test_rmse = root_mean_squared_error(y_test, y_test_pred)
final_results.append(("Random Forest", train_rmse, test_rmse))

# Print RMSE scores
print(f"Training RMSE: {train_rmse:.4f}")
print(f"Test RMSE: {test_rmse:.4f}")
```

```
Training RMSE: 0.8119
Test RMSE: 0.9895
```

Support Vector Regressor

```
# Fit best model on full training set
best_model.fit(X_train_transformed, y_train_scaled)
```

```
SVR
SVR(C=0.1, epsilon=0.5, kernel='linear')
```

```
# Predict and inverse transform
y_train_pred_scaled = best_model.predict(X_train_transformed)
y_test_pred_scaled = best_model.predict(X_test_transformed)

y_train_pred = scaler_y.inverse_transform(y_train_pred_scaled.reshape(-1, 1)).ravel()
y_test_pred = scaler_y.inverse_transform(y_test_pred_scaled.reshape(-1, 1)).ravel()

# RMSE calculation
train_rmse = root_mean_squared_error(y_train, y_train_pred)
test_rmse = root_mean_squared_error(y_test, y_test_pred)
final_results.append(("SVR", train_rmse, test_rmse))

print("Train RMSE:", train_rmse)
print("Test RMSE:", test_rmse)
print("Best Hyperparameters:", grid_search.best_params_)
```

```
Train RMSE: 0.9059957997320988
Test RMSE: 1.0241276760782283
Best Hyperparameters: {'C': 0.1, 'epsilon': 0.5, 'kernel': 'linear'}
```

Multiple Linear Regression

```
train_data = pd.concat([pd.DataFrame(X_train_transformed), y_train.reset_index(drop=True)], axis=1)

# First-order multiple regression model
train_df = sm.add_constant(train_data)  # include intercept

# Create column names for the features in X_train_transformed
num_features = X_train_transformed.shape[1]
feature_names = [f'feature_{i}' for i in range(num_features)]

# Add feature names to the DataFrame
train_df.columns = ['const'] + feature_names + ['Score']

# Update the formula to use the new feature names
reg = smf.ols('Score ~ ' + ' + '.join(feature_names), data=train_df) # Assuming 'price' is the name of your target variable column


# Fit multiple regression model
res = reg.fit()


# Calculate rmse on training set
y_train_pred = res.predict(train_df.drop(columns=['Score']))
train_rmse = root_mean_squared_error(y_train, y_train_pred)

test_df = pd.DataFrame(X_test_transformed, columns=feature_names)
test_df = sm.add_constant(test_df)  # include intercept

# Predict for the test set
y_pred_reg = res.predict(test_df)
test_rmse = root_mean_squared_error(y_test, y_pred_reg)

final_results.append(("Multiple Linear Regression", train_rmse, test_rmse))

print("Training RMSE:", train_rmse)
print("Test RMSE:", test_rmse)
```

```
Training RMSE: 0.8070821638006869
Test RMSE: 1.1648286476571108
```

Final Results

```
# Create DataFrame and sort by test accuracy
results_df = pd.DataFrame(final_results, columns=["Model", "Train RMSE", "Test RMSE"])
best_per_model = results_df.loc[results_df.groupby("Model")["Test RMSE"].idxmax()]
best_per_model_sorted = best_per_model.sort_values(by="Test RMSE", ascending=True).reset_index(drop=True)

# Display
print("\nFinal Results:")
print(best_per_model_sorted.to_string(index=False))
```

```
Final Results:
                        Model  Train RMSE  Test RMSE
                Random Forest    0.811923   0.989500
                          MLP    0.934563   0.996781
                         K-NN    0.928166   1.014703
                          SVR    0.905996   1.024128
Multiple Linear Regression     0.807082   1.164829
```

Random Forest had the lowest test RMSE, 0.9895, and a reasonable variation between train and test RMSE that suggests there was not too much influence of overfitting in the performance of the model.

(6)

This dataset shows that tree-based methods like Random Forest are best for predicting the score a hotel receives from TripAdvisor reviews because they are able to capture the non-linear relationships between several numerical and categorical variables. Multilayer Perceptron Regressor and K-nearest neighbors methods also performed fairly well, which suggests that more complex models are more suitable for analyzing this dataset due to its high dimensionality. Support Vector Regression performed only slightly worse than k-NN on the test data, while the Multiple Linear Regression model performed the worst on the test data but best on the training data. This indicates that linear models may be more likely to run the risk of overfitting this dataset and less effective for representing the data overall.