0-Overfitting_Underfitting

October 20, 2024

1 Overfitting and Underfitting

Overfitting occurs when a machine learning model learns not only the underlying patterns in the training data but also the noise or random fluctuations in that data. As a result, the model performs very well on the training data but poorly on new, unseen data (test data or validation data), as it fails to generalize beyond the training set.

In essence, an overfitted model becomes too complex, capturing irrelevant details that don't contribute to the actual patterns in the data.

What is Underfitting? Underfitting occurs when a machine learning model is too simple to capture the underlying patterns in the data. This results in poor performance on both the training data and the test data. The model doesn't learn enough from the training data to make accurate predictions.

In essence, an underfitted model has high bias and cannot model the complexity of the data.

1.0.1 Why do Overfitting and Underfitting happen?

Overfitting happens when the model is too flexible or too complex, often due to:

- Too many features or irrelevant features in the model.
- A high degree of the polynomial function (in the case of regression).
- Too long training time, which allows the model to fit even random noise in the training data.

Underfitting happens when the model is too simplistic, often due to:

- Too few features or not enough relevant features.
- Too simple a model (e.g., using linear regression for data that has a non-linear relationship).
- Insufficient training, meaning the model hasn't been given enough time to learn from the data.

1.0.2 How to detect Overfitting and Underfitting?

Overfitting:

- Low error on the training data.
- High error on the test data or new data.
- The model seems to memorize the training data rather than generalize patterns.

Underfitting:

- High error on both the training data and the test data.
- The model cannot capture the relationship between the input features and target variable effectively.

1.0.3 How to prevent Overfitting and Underfitting?

For Overfitting:

- **Simplify the model**: Use fewer features or reduce the complexity of the model.
- **Regularization**: Apply techniques like L1 (Lasso), L2 (Ridge), or Elastic Net regularization, which penalize overly complex models.
- **Cross-validation**: Use techniques like k-fold cross-validation to assess the model on different subsets of data.
- **Prune decision trees**: For tree-based models, pruning helps limit the depth and complexity of the model.
- **Early stopping**: In algorithms like gradient boosting, stop the training process when performance on the validation set begins to deteriorate.
- **Increase the size of the dataset**: More training data helps the model generalize better.

For Underfitting:

- **Increase model complexity**: Use more powerful models, add more features, or use higher-degree polynomial functions in regression.
- **Remove constraints**: For models like decision trees, allow the tree to grow deeper.
- **Reduce regularization**: Lessen the penalization of model complexity to allow the model to capture more details in the data.
- **Train for more epochs**: Let the model train longer (for models like neural networks) to capture more patterns.

1.0.4 Examples in Classification and Regression

Classification Example: Decision Trees Let's start by illustrating overfitting and underfitting in classification using a Decision Tree.

1. Underfitting in Classification: If you limit the depth of the decision tree (e.g., a maximum depth of 1 or 2), the tree may not be complex enough to capture the relationships in the data. As a result, it underfits the data, meaning the model cannot make accurate predictions for both the training and test sets.

2. Overfitting in Classification: If you allow the decision tree to grow too deep without any constraints (e.g., depth = 20), it may fit the training data perfectly. However, the model will likely memorize the data, fitting noise and making poor predictions on new data (overfitting).

[1]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load dataset
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,␣
 ↪test_size=0.2, random_state=42)

# Underfitting: Shallow tree (max_depth=1)
clf_underfit = DecisionTreeClassifier(max_depth=1)
clf_underfit.fit(X_train, y_train)
y_pred_train_underfit = clf_underfit.predict(X_train)
y_pred_test_underfit = clf_underfit.predict(X_test)

# Overfitting: Deep tree (max_depth=20)
clf_overfit = DecisionTreeClassifier(max_depth=20)
clf_overfit.fit(X_train, y_train)
y_pred_train_overfit = clf_overfit.predict(X_train)
y_pred_test_overfit = clf_overfit.predict(X_test)

# Calculate accuracy
train_acc_underfit = accuracy_score(y_train, y_pred_train_underfit)
test_acc_underfit = accuracy_score(y_test, y_pred_test_underfit)

train_acc_overfit = accuracy_score(y_train, y_pred_train_overfit)
test_acc_overfit = accuracy_score(y_test, y_pred_test_overfit)

print(f"Underfitting - Train Accuracy: {train_acc_underfit}, Test Accuracy:␣
 ↪{test_acc_underfit}")
print(f"Overfitting - Train Accuracy: {train_acc_overfit}, Test Accuracy:␣
 ↪{test_acc_overfit}")
```

```
Underfitting - Train Accuracy: 0.675, Test Accuracy: 0.6333333333333333
Overfitting - Train Accuracy: 1.0, Test Accuracy: 1.0
```

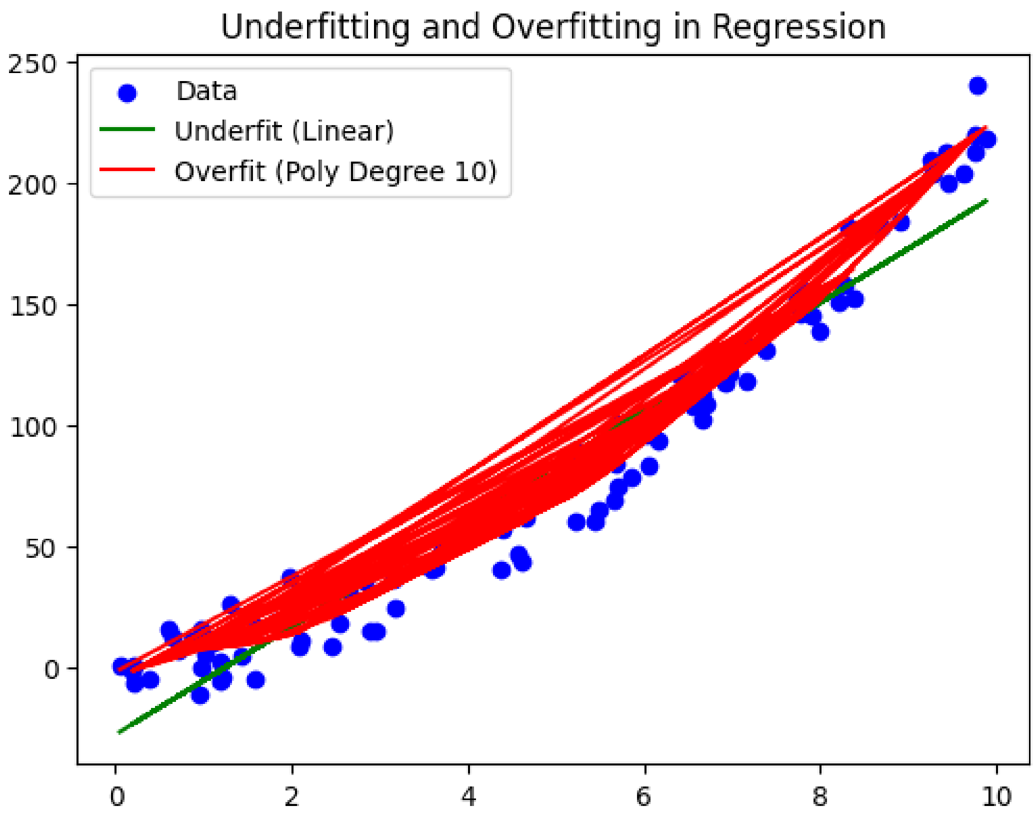
1.0.5 Regression Example: Polynomial Regression

1. Underfitting in Regression: Using a linear regression model for data that has a non-linear relationship (e.g., quadratic data) will cause underfitting. The model will be too simple to capture the patterns in the data.

2.Overfitting in Regression: If you use a high-degree polynomial regression (e.g., 9th or 10th degree) on data that has a simple relationship, the model will capture noise in the training data and won't generalize well to new data.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

```python
from sklearn.metrics import root_mean_squared_error

# Create a dataset (quadratic data)
np.random.seed(0)
X = np.random.rand(100, 1) * 10  # Random points from 0 to 10
y = 2 * (X ** 2) + 3 * X + np.random.randn(100, 1) * 10  # Quadratic relation␣
 ↪with some noise

# Underfitting: Linear model (degree = 1)
poly_features_underfit = PolynomialFeatures(degree=1)
X_poly_underfit = poly_features_underfit.fit_transform(X)
lin_reg_underfit = LinearRegression()
lin_reg_underfit.fit(X_poly_underfit, y)
y_pred_underfit = lin_reg_underfit.predict(X_poly_underfit)

# Overfitting: High-degree polynomial model (degree = 10)
poly_features_overfit = PolynomialFeatures(degree=10)
X_poly_overfit = poly_features_overfit.fit_transform(X)
lin_reg_overfit = LinearRegression()
lin_reg_overfit.fit(X_poly_overfit, y)
y_pred_overfit = lin_reg_overfit.predict(X_poly_overfit)

# Plot results
plt.scatter(X, y, color='blue', label="Data")
plt.plot(X, y_pred_underfit, color='green', label="Underfit (Linear)")
plt.plot(X, y_pred_overfit, color='red', label="Overfit (Poly Degree 10)")
plt.title("Underfitting and Overfitting in Regression")
plt.legend()
plt.show()
```


Underfitting and Overfitting in Regression
Data
Underfit (Linear)
Overfit (Poly Degree 10)
250
200
150
100
50
0
0
2
4
6
8
10

1-train_test_split

October 20, 2024

1 Train Test Split

The train-test split is a critical concept in machine learning for evaluating the performance of a model. It involves dividing the dataset into two distinct parts: the training set and the test set. This method ensures that we can assess how well the model generalizes to unseen data, which is the key to making accurate predictions in real-world applications.

Purpose of Train-Test Split: The main goal is to evaluate the model's ability to generalize to new, unseen data by testing it on a separate test set that the model has never encountered during training. If a model performs well on both the training data and the test data, it's likely to perform well on new, real-world data.

Key Terminology:

1. Training Set: The subset of the dataset used to train the model. The model learns patterns, relationships, and trends from this data.
2. Test Set: The subset used to evaluate the model after training. It provides an unbiased estimate of the model's performance on unseen data.

Why Train-Test Split is Important:

- Avoid Overfitting: Without a train-test split, the model might memorize the training data (overfitting) rather than learning generalizable patterns. This would result in poor performance on new data.
- Unbiased Evaluation: It provides an unbiased measure of how well the model generalizes beyond the training data.
- Generalization Performance: It allows us to assess the model's generalization error, i.e., how it will perform on future data.

Train-Test Split Ratio:

- 80:20 Split: Commonly used when the dataset is sufficiently large. This balance ensures that there is enough data for the model to learn (80%) while still keeping a good portion (20%) for evaluating the model's performance.
- 70:30 Split: Used when you want more data for testing, but still have enough data for training. This might be more appropriate when the dataset is relatively smaller, as having more test data can give a clearer picture of generalization.

- 90:10 Split: Used when the dataset is very large. A small test set is enough to provide a good estimate of model performance since the model has a huge amount of data to train on.

[23]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
```

[24]:
```
data = pd.read_csv('500hits.csv', encoding='latin-1')
data.head()
```

[24]:
```
          PLAYER  YRS     G     AB     R     H   2B   3B   HR   RBI    BB  \
0        Ty Cobb   24  3035  11434  2246  4189  724  295  117   726  1249
1    Stan Musial   22  3026  10972  1949  3630  725  177  475  1951  1599
2   Tris Speaker   22  2789  10195  1882  3514  792  222  117   724  1381
3    Derek Jeter   20  2747  11195  1923  3465  544   66  260  1311  1082
4  Honus Wagner   21  2792  10430  1736  3430  640  252  101     0   963

     SO   SB   CS     BA  HOF
0   357  892  178  0.366    1
1   696   78   31  0.331    1
2   220  432  129  0.345    1
3  1840  358   97  0.310    1
4   327  722   15  0.329    1
```

[25]:
```
X = data.drop(columns=['PLAYER', 'HOF'])
y = data['HOF']

X.shape
y.shape
```

[25]:
```
(465,)
```

[26]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=11,␣
 ↪test_size=0.2)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(372, 14)
(93, 14)
(372,)
(93,)
```

[27]:
```
print(X_train.describe().round(3))
```

```
           YRS         G         AB         R         H        2B        3B  \
count  372.000   372.000    372.000   372.000   372.000   372.000   372.000
mean    17.011  2046.522   7526.078  1154.126  2177.995   382.505    78.094
std      2.662   351.233   1302.406   291.308   426.615    97.173    48.798
```

```
min     11.000  1331.000   4981.000   651.000  1660.000  177.000    3.000
25%     15.000  1797.500   6507.500   936.000  1838.000  312.000   41.000
50%     17.000  1992.000   7237.000  1099.000  2080.500  367.000   67.000
75%     19.000  2245.500   8198.250  1305.000  2383.750  436.250  108.000
max     26.000  3308.000  12364.000  2295.000  4189.000  792.000  309.000

            HR       RBI        BB        SO        SB       CS       BA
count  372.000   372.000   372.000   372.000   372.000  372.000  372.000
mean   202.642   901.073   780.105   850.323   196.927   58.987    0.289
std    141.726   484.370   327.453   472.918   185.586   49.322    0.021
min      9.000     0.000   239.000     0.000     7.000    0.000    0.246
25%     79.750   645.000   536.500   448.000    64.500   23.000    0.274
50%    185.500   977.500   719.000   844.000   141.000   52.000    0.288
75%    293.250  1218.500   961.250  1234.250   285.500   84.000    0.300
max    755.000  2297.000  2190.000  1936.000  1406.000  335.000    0.366
```

[28]:
```
print(X_test.describe().round(3))
```

```
          YRS         G         AB         R         H       2B       3B  \
count  93.000    93.000     93.000    93.000    93.000   93.000   93.000
mean   17.204  2057.409   7452.968  1135.065  2139.258  374.742   80.398
std     3.154   368.580   1265.371   283.877   415.165   93.929   51.792
min    11.000  1399.000   5472.000   601.000  1660.000  206.000   14.000
25%    15.000  1820.000   6622.000   935.000  1818.000  310.000   45.000
50%    17.000  1997.000   7359.000  1108.000  2054.000  361.000   68.000
75%    19.000  2282.000   8096.000  1283.000  2256.000  432.000   99.000
max    25.000  2850.000  10876.000  1859.000  3430.000  668.000  252.000

            HR       RBI        BB        SO       SB       CS      BA
count   93.000    93.000    93.000    93.000   93.000   93.000  93.000
mean   194.677   867.011   797.387   836.065  191.817   54.473   0.287
std    151.600   495.127   328.755   552.309  166.926   42.508   0.022
min     15.000     0.000   266.000    15.000    8.000    0.000   0.248
25%     78.000   618.000   527.000   359.000   61.000   15.000   0.272
50%    151.000   926.000   750.000   745.000  137.000   50.000   0.285
75%    291.000  1138.000   937.000  1179.000  271.000   83.000   0.299
max    612.000  1922.000  1747.000  2597.000  744.000  173.000   0.340
```

2-feature_scaling

October 20, 2024

1 Feature Scaling

is a crucial preprocessing step in machine learning where we transform the data to ensure that the features are on a similar scale. Many machine learning algorithms compute distances (e.g., Euclidean distance) or gradients that are highly sensitive to the magnitude of the input data. If one feature has a large range of values compared to others, it can dominate and distort the model's predictions. Feature scaling addresses this issue by bringing the features into comparable ranges.

Why Feature Scaling is Important:

1. **Improves Model Convergence**: Algorithms like Gradient Descent and its variants benefit from feature scaling as it makes the optimization process faster and more stable.
2. **Distance-based Algorithms**: Algorithms like KNN, K-Means clustering, and SVM rely on distance metrics. If the features have different ranges, the model could misinterpret the importance of features based on their magnitude.
3. **Regularization and PCA**: Feature scaling ensures that regularization techniques (like Lasso or Ridge) and dimensionality reduction methods (like PCA) treat all features equally.

Techniques for Feature Scaling

1. **Normalization (Min-Max Scaling)** Normalization scales the data into a fixed range, typically between 0 and 1, or sometimes -1 and 1. It's commonly used when you know that the distribution of data does not follow a Gaussian distribution, or when the algorithm expects the data to be within a particular range.
2. **Standardization (Z-Score Normalization)** Standardization transforms the data to have a mean of 0 and a standard deviation of 1. It's useful when the feature distribution follows a Gaussian distribution (bell-shaped curve) or when the algorithm assumes the data is normally distributed (e.g., logistic regression, linear regression, or SVM).

1.0.1 When to Use Normalization vs. Standardization:

- Normalization is preferred for algorithms like:
  - Neural Networks (often benefit from data scaled between 0 and 1),
  - K-Nearest Neighbors (KNN),
  - Support Vector Machines (SVM) when using an RBF kernel.
- Standardization is preferred for:
  - Linear regression,
  - Logistic regression,
  - SVM (with linear kernel),

– Principal Component Analysis (PCA),
– Regularized models (Lasso, Ridge).

[1]:
```
import pandas as pd

df = pd.read_csv('500hits.csv', encoding='latin-1')

df = df.drop(columns=['PLAYER', 'CS'])
df.describe().round(3)
```

[1]:
```
             YRS         G          AB         R         H        2B        3B  \
count    465.000   465.000     465.000   465.000   465.000   465.000   465.000
mean      17.049  2048.699    7511.456  1150.314  2170.247   380.953    78.555
std        2.765   354.392    1294.066   289.635   424.191    96.483    49.363
min       11.000  1331.000    4981.000   601.000  1660.000   177.000     3.000
25%       15.000  1802.000    6523.000   936.000  1838.000   312.000    41.000
50%       17.000  1993.000    7241.000  1104.000  2076.000   366.000    67.000
75%       19.000  2247.000    8180.000  1296.000  2375.000   436.000   107.000
max       26.000  3308.000   12364.000  2295.000  4189.000   792.000   309.000

             HR       RBI        BB        SO        SB        BA       HOF
count    465.000   465.000   465.000   465.000   465.000   465.000   465.000
mean     201.049   894.260   783.561   847.471   195.905     0.289     0.329
std      143.623   486.193   327.432   489.224   181.846     0.021     0.475
min        9.000     0.000   239.000     0.000     7.000     0.246     0.000
25%       79.000   640.000   535.000   436.000    63.000     0.273     0.000
50%      178.000   968.000   736.000   825.000   137.000     0.287     0.000
75%      292.000  1206.000   955.000  1226.000   285.000     0.300     1.000
max      755.000  2297.000  2190.000  2597.000  1406.000     0.366     2.000
```

[2]:
```
X1 = df.iloc[:, 0:13]
```

[3]:
```
X2 = df.iloc[:, 0:13]
```

[4]:
```
from sklearn.preprocessing import StandardScaler

scaleStandard = StandardScaler()
X1 = scaleStandard.fit_transform(X1)
X1 = pd.DataFrame(X1,␣
 ↪columns=['YRS','G','AB','R','H','2B','3B','HR','RBI','BB','SO','SB','BA'])
X1.head()
```

[4]:
```
        YRS         G        AB         R         H        2B        3B  \
0  2.516295  2.786078  3.034442  3.787062  4.764193  3.559333  4.389485
1  1.792237  2.760655  2.677044  2.760530  3.444971  3.569709  1.996457
2  1.792237  2.091184  2.075964  2.528955  3.171214  4.264876  2.909053
3  1.068180  1.972543  2.849554  2.670665  3.055576  1.691719 -0.254611
4  1.430208  2.099658  2.257758  2.024329  2.972977  2.687780  3.517449
```

```
         HR       RBI        BB        SO        SB        BA
0 -0.585841 -0.346449  1.423013 -1.003628  3.832067  3.648290
1  1.909487  2.175837  2.493089 -0.309948 -0.649080  1.996159
2 -0.585841 -0.350567  1.826585 -1.283965  1.299723  2.657012
3  0.410896  0.858071  0.912434  2.030966  0.892346  1.004881
4 -0.697364 -1.841290  0.548609 -1.065016  2.896201  1.901752
```

[5]:
```
X1.describe().round(3)
```

[5]:
```
           YRS        G       AB        R        H       2B       3B       HR  \
count  465.000  465.000  465.000  465.000  465.000  465.000  465.000  465.000
mean    -0.000    0.000    0.000    0.000    0.000    0.000   -0.000   -0.000
std      1.001    1.001    1.001    1.001    1.001    1.001    1.001    1.001
min     -2.190   -2.027   -1.958   -1.899   -1.204   -2.116   -1.532   -1.339
25%     -0.742   -0.697   -0.765   -0.741   -0.784   -0.715   -0.762   -0.851
50%     -0.018   -0.157   -0.209   -0.160   -0.222   -0.155   -0.234   -0.161
75%      0.706    0.560    0.517    0.504    0.483    0.571    0.577    0.634
max      3.240    3.557    3.754    3.956    4.764    4.265    4.673    3.861

           RBI       BB       SO       SB       BA
count  465.000  465.000  465.000  465.000  465.000
mean     0.000    0.000   -0.000    0.000    0.000
std      1.001    1.001    1.001    1.001    1.001
min     -1.841   -1.665   -1.734   -1.040   -2.016
25%     -0.524   -0.760   -0.842   -0.732   -0.742
50%      0.152   -0.145   -0.046   -0.324   -0.081
75%      0.642    0.524    0.775    0.490    0.533
max      2.888    4.300    3.580    6.662    3.648
```

[6]:
```
from sklearn.preprocessing import MinMaxScaler

scaleMinMax = MinMaxScaler(feature_range=(0,1))
X2 = scaleMinMax.fit_transform(X2)
X2 = pd.DataFrame(X2,␣
 ↪columns=['YRS','G','AB','R','H','2B','3B','HR','RBI','BB','SO','SB','BA'])
X2.head()
```

[6]:
```
        YRS         G        AB         R         H        2B        3B  \
0  0.866667  0.861912  0.874035  0.971074  1.000000  0.889431  0.954248
1  0.733333  0.857360  0.811459  0.795750  0.778964  0.891057  0.568627
2  0.733333  0.737481  0.706217  0.756198  0.733096  1.000000  0.715686
3  0.600000  0.716237  0.841663  0.780401  0.713721  0.596748  0.205882
4  0.666667  0.738998  0.738047  0.670012  0.699881  0.752846  0.813725

         HR       RBI        BB        SO        SB        BA
0  0.144772  0.316064  0.517683  0.137466  0.632595  1.000000
```

```
1  0.624665  0.849369  0.697078  0.268002  0.050751  0.708333
2  0.144772  0.315194  0.585341  0.084713  0.303788  0.825000
3  0.336461  0.570744  0.432086  0.708510  0.250893  0.533333
4  0.123324  0.000000  0.371092  0.125915  0.511079  0.691667
```

[7]:
```
X2.describe().round(3)
```

[7]:
```
           YRS        G       AB        R        H       2B       3B       HR  \
count  465.000  465.000  465.000  465.000  465.000  465.000  465.000  465.000
mean     0.403    0.363    0.343    0.324    0.202    0.332    0.247    0.257
std      0.184    0.179    0.175    0.171    0.168    0.157    0.161    0.193
min      0.000    0.000    0.000    0.000    0.000    0.000    0.000    0.000
25%      0.267    0.238    0.209    0.198    0.070    0.220    0.124    0.094
50%      0.400    0.335    0.306    0.297    0.164    0.307    0.209    0.227
75%      0.533    0.463    0.433    0.410    0.283    0.421    0.340    0.379
max      1.000    1.000    1.000    1.000    1.000    1.000    1.000    1.000

           RBI       BB       SO       SB       BA
count  465.000  465.000  465.000  465.000  465.000
mean     0.389    0.279    0.326    0.135    0.356
std      0.212    0.168    0.188    0.130    0.177
min      0.000    0.000    0.000    0.000    0.000
25%      0.279    0.152    0.168    0.040    0.225
50%      0.421    0.255    0.318    0.093    0.342
75%      0.525    0.367    0.472    0.199    0.450
max      1.000    1.000    1.000    1.000    1.000
```

3-one_hot_encoder

October 20, 2024

1 One Hot Encoder

One-Hot Encoding is a technique used to represent categorical data as binary vectors. Machine learning models typically expect numerical input, but many datasets contain categorical variables (e.g., "color" can be "red," "blue," or "green"). One-hot encoding allows us to transform these categorical features into a format that the model can understand without assuming any ordinal relationship between the categories.

Why One-Hot Encoding is Important:

1. **Handling Categorical Data**: Many machine learning models (e.g., linear regression, decision trees, SVMs) cannot work with raw categorical data directly. One-hot encoding converts the categorical data into binary vectors that the model can process.
2. **Avoiding Misinterpretation of Ordinal Relationships**: By assigning each category its own binary feature, one-hot encoding ensures that the model does not assume any inherent order or relationship between categories, which could mislead the learning process.

How One-Hot Encoding Works: Assume we have a categorical feature "Color" with three possible values: "Red," "Green," and "Blue." One-hot encoding creates a binary vector for each category:

Color	One-Hot Encoding
Red	[1, 0, 0]
Green	[0, 1, 0]
Blue	[0, 0, 1]

When to Use One-Hot Encoding:

- **Non-Ordinal Categorical Data**: One-hot encoding is used for nominal (non-ordinal) categorical data, where there is no intrinsic order among the categories (e.g., "color" or "country").
- **Features with Small Cardinality**: Works well when the categorical feature has a small number of unique categories. When the number of unique categories (cardinality) is large, one-hot encoding can lead to a high-dimensional, sparse matrix, which can become computationally expensive and increase memory usage.

Summary:

- One-Hot Encoding is used to convert categorical variables into binary vectors.

- It is essential for handling non-ordinal categorical data in machine learning.
- It works by creating a separate binary column for each unique category.
- Care should be taken with features that have high cardinality, as it can lead to large, sparse matrices.
- The dummy variable trap can be avoided by dropping one of the one-hot encoded columns when working with linear models.

[2]:
```
import pandas as pd
d = {
    'sales':␣
  ↪[100000,222000,1000000,522000,111111,222222,1111111,20000,75000,90000,1000000,10000],
    'city':␣
  ↪['Tampa','Tampa','Orlando','Jacksonville','Miami','Jacksonville','Miami','Miami','Orlando',
    'size': ['Small',␣
  ↪'Medium','Large','Large','Small','Medium','Large','Small','Medium','Medium','Medium','Small
}

df = pd.DataFrame(data=d)
df.head()
```

[2]:
```
     sales          city    size
0   100000         Tampa   Small
1   222000         Tampa  Medium
2  1000000       Orlando   Large
3   522000  Jacksonville   Large
4   111111         Miami   Small
```

[3]:
```
df['city'].unique()
```

[3]:
```
array(['Tampa', 'Orlando', 'Jacksonville', 'Miami'], dtype=object)
```

[7]:
```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False).
  ↪set_output(transform='pandas')
ohetransform = ohe.fit_transform(df)
ohetransform
```

[7]:
```
    sales_10000  sales_20000  sales_75000  sales_90000  sales_100000  \
0           0.0          0.0          0.0          0.0           1.0
1           0.0          0.0          0.0          0.0           0.0
2           0.0          0.0          0.0          0.0           0.0
3           0.0          0.0          0.0          0.0           0.0
4           0.0          0.0          0.0          0.0           0.0
5           0.0          0.0          0.0          0.0           0.0
6           0.0          0.0          0.0          0.0           0.0
```

```
7           0.0           1.0           0.0           0.0            0.0
8           0.0           0.0           1.0           0.0            0.0
9           0.0           0.0           0.0           1.0            0.0
10          0.0           0.0           0.0           0.0            0.0
11          1.0           0.0           0.0           0.0            0.0

    sales_111111  sales_222000  sales_222222  sales_522000  sales_1000000  \
0            0.0           0.0           0.0           0.0            0.0
1            0.0           1.0           0.0           0.0            0.0
2            0.0           0.0           0.0           0.0            1.0
3            0.0           0.0           0.0           1.0            0.0
4            1.0           0.0           0.0           0.0            0.0
5            0.0           0.0           1.0           0.0            0.0
6            0.0           0.0           0.0           0.0            0.0
7            0.0           0.0           0.0           0.0            0.0
8            0.0           0.0           0.0           0.0            0.0
9            0.0           0.0           0.0           0.0            0.0
10           0.0           0.0           0.0           0.0            1.0
11           0.0           0.0           0.0           0.0            0.0

    sales_1111111  city_Jacksonville  city_Miami  city_Orlando  city_Tampa  \
0             0.0                0.0         0.0           0.0         1.0
1             0.0                0.0         0.0           0.0         1.0
2             0.0                0.0         0.0           1.0         0.0
3             0.0                1.0         0.0           0.0         0.0
4             0.0                0.0         1.0           0.0         0.0
5             0.0                1.0         0.0           0.0         0.0
6             1.0                0.0         1.0           0.0         0.0
7             0.0                0.0         1.0           0.0         0.0
8             0.0                0.0         0.0           1.0         0.0
9             0.0                0.0         0.0           1.0         0.0
10            0.0                0.0         0.0           1.0         0.0
11            0.0                0.0         0.0           1.0         0.0

    size_Large  size_Medium  size_Small
0          0.0          0.0         1.0
1          0.0          1.0         0.0
2          1.0          0.0         0.0
3          1.0          0.0         0.0
4          0.0          0.0         1.0
5          0.0          1.0         0.0
6          1.0          0.0         0.0
7          0.0          0.0         1.0
8          0.0          1.0         0.0
9          0.0          1.0         0.0
10         0.0          1.0         0.0
11         0.0          0.0         1.0
```

```
[9]: df = pd.concat([df, ohetransform], axis=1)
     df
```

```
[9]:       sales          city    size  sales_10000  sales_20000  sales_75000  \
     0    100000         Tampa   Small          0.0          0.0          0.0
     1    222000         Tampa  Medium          0.0          0.0          0.0
     2   1000000       Orlando   Large          0.0          0.0          0.0
     3    522000  Jacksonville   Large          0.0          0.0          0.0
     4    111111         Miami   Small          0.0          0.0          0.0
     5    222222  Jacksonville  Medium          0.0          0.0          0.0
     6   1111111         Miami   Large          0.0          0.0          0.0
     7     20000         Miami   Small          0.0          1.0          0.0
     8     75000       Orlando  Medium          0.0          0.0          1.0
     9     90000       Orlando  Medium          0.0          0.0          0.0
     10  1000000       Orlando  Medium          0.0          0.0          0.0
     11    10000       Orlando   Small          1.0          0.0          0.0

         sales_90000  sales_100000  sales_111111  sales_222000  …  sales_522000  \
     0           0.0           1.0           0.0           0.0  …           0.0
     1           0.0           0.0           0.0           1.0  …           0.0
     2           0.0           0.0           0.0           0.0  …           0.0
     3           0.0           0.0           0.0           0.0  …           1.0
     4           0.0           0.0           1.0           0.0  …           0.0
     5           0.0           0.0           0.0           0.0  …           0.0
     6           0.0           0.0           0.0           0.0  …           0.0
     7           0.0           0.0           0.0           0.0  …           0.0
     8           0.0           0.0           0.0           0.0  …           0.0
     9           1.0           0.0           0.0           0.0  …           0.0
     10          0.0           0.0           0.0           0.0  …           0.0
     11          0.0           0.0           0.0           0.0  …           0.0

         sales_1000000  sales_1111111  city_Jacksonville  city_Miami  city_Orlando  \
     0             0.0            0.0                0.0         0.0           0.0
     1             0.0            0.0                0.0         0.0           0.0
     2             1.0            0.0                0.0         0.0           1.0
     3             0.0            0.0                1.0         0.0           0.0
     4             0.0            0.0                0.0         1.0           0.0
     5             0.0            0.0                1.0         0.0           0.0
     6             0.0            1.0                0.0         1.0           0.0
     7             0.0            0.0                0.0         1.0           0.0
     8             0.0            0.0                0.0         0.0           1.0
     9             0.0            0.0                0.0         0.0           1.0
     10            1.0            0.0                0.0         0.0           1.0
     11            0.0            0.0                0.0         0.0           1.0

         city_Tampa  size_Large  size_Medium  size_Small
     0          1.0         0.0          0.0         1.0
```

```
1          1.0         0.0          1.0         0.0
2          0.0         1.0          0.0         0.0
3          0.0         1.0          0.0         0.0
4          0.0         0.0          0.0         1.0
5          0.0         0.0          1.0         0.0
6          0.0         1.0          0.0         0.0
7          0.0         0.0          0.0         1.0
8          0.0         0.0          1.0         0.0
9          0.0         0.0          1.0         0.0
10         0.0         0.0          1.0         0.0
11         0.0         0.0          0.0         1.0

[12 rows x 39 columns]
```

4-ordinal_encoder

October 20, 2024

1 Ordinal Encoder

is a technique used to convert categorical features into numerical values, where the categories have a clear order or ranking. Unlike one-hot encoding, which treats all categories as independent, ordinal encoding assigns each unique category an integer value based on its rank or order.

Why Ordinal Encoding is Important:

- Ordered Categories: It is specifically used for ordinal data, where the categories have a meaningful order or ranking, but the distances between these categories might not be uniform.
- Simpler Representation: It provides a simpler, more compact representation of categorical features compared to one-hot encoding, especially when the feature has an inherent order and a small number of categories.

When to Use Ordinal Encoding:

- Ordinal Categorical Data: Use ordinal encoding when the categorical feature has an intrinsic order. For example, “low,” “medium,” and “high” represent ordered categories, but one-hot encoding would not capture the ranking.
- When Preserving Order is Important: If the relationship between categories is important for the model to understand, ordinal encoding helps preserve this information. Smaller Cardinality: When dealing with ordinal features with a small number of categories, ordinal encoding is efficient and works well with most machine learning algorithms.

How Ordinal Encoding Works: Let’s take an example where we have a categorical feature “Quality” with values: “Low,” “Medium,” and “High.” Ordinal encoding assigns each category a numerical value based on the order:

Quality Low 0 Medium 1 High 2

Color	Ordinal Encoding
Low	0
Medium	1
High	2

In this case, “Low” is mapped to 0, “Medium” to 1, and “High” to 2, which captures the inherent ranking of the categories.

When Not to Use Ordinal Encoding:

- Non-Ordinal Categorical Data: If the categorical feature has no inherent order (e.g., “color” or “country”), ordinal encoding may mislead the model because it implies a relationship or ranking between the categories that doesn’t exist.
- Assumed Numerical Relationships: Some machine learning models, like linear regression, may interpret ordinal values as having linear relationships (i.e., assuming that the difference between “Medium” and “Low” is the same as between “High” and “Medium”), which is not always correct.

Use Cases of Ordinal Encoding:

- Education Levels: Categories such as “Primary,” “Secondary,” and “Tertiary” have an inherent order, making them a perfect candidate for ordinal encoding.
- Customer Satisfaction Levels: Responses like “Very Dissatisfied,” “Dissatisfied,” “Neutral,” “Satisfied,” and “Very Satisfied” can be encoded in their natural order.
- Rating Scales: Ratings such as “Low,” “Medium,” and “High” in product reviews can be ordinal encoded since they imply a ranking.

Summary:

- Ordinal Encoding converts categorical data into numerical data, preserving the order of categories.
- It is used when the categorical variable has an inherent order (e.g., “low,” “medium,” “high”).
- It is compact and efficient but may lead to misleading interpretations if used on non-ordinal categorical data.
- Some algorithms (e.g., decision trees, random forests) work well with ordinal encoding, but others (e.g., linear models) may misinterpret the encoding.

[1]:
```python
import pandas as pd

d = {'sales':␣
 ↪[100000,222000,1000000,522000,111111,222222,1111111,20000,75000,90000,1000000,10000],␣
 ↪'city':␣
 ↪['Tampa','Tampa','Orlando','Jacksonville','Miami','Jacksonville','Miami','Miami','Orlando',
 ↪'size': ['Small',␣
 ↪'Medium','Large','Large','Small','Medium','Large','Small','Medium','Medium','Medium','Small

df = pd.DataFrame(data=d)
df.head()
```

[1]:
```
      sales          city    size
0    100000         Tampa   Small
1    222000         Tampa  Medium
2   1000000       Orlando   Large
3    522000  Jacksonville   Large
```

```
4   111111         Miami   Small
```

[2]:
```
df['size'].unique()
sizes = ['Small', 'Medium', 'Large']
```

[3]:
```
from sklearn.preprocessing import OrdinalEncoder
enc = OrdinalEncoder(categories = [sizes])
size_enc = enc.fit_transform(df[['size']])
size_enc
```

[3]:
```
array([[0.],
       [1.],
       [2.],
       [2.],
       [0.],
       [1.],
       [2.],
       [0.],
       [1.],
       [1.],
       [1.],
       [0.]])
```

[4]:
```
df['size_enc'] = size_enc
df
```

[4]:
```
      sales          city    size  size_enc
0    100000         Tampa   Small       0.0
1    222000         Tampa  Medium       1.0
2   1000000       Orlando   Large       2.0
3    522000  Jacksonville   Large       2.0
4    111111         Miami   Small       0.0
5    222222  Jacksonville  Medium       1.0
6   1111111         Miami   Large       2.0
7     20000         Miami   Small       0.0
8     75000       Orlando  Medium       1.0
9     90000       Orlando  Medium       1.0
10  1000000       Orlando  Medium       1.0
11    10000       Orlando   Small       0.0
```

5-handling-missing-data

October 20, 2024

1 Handling Missing Data

Handling missing data is a crucial step in the data preprocessing phase of machine learning projects. Missing values can arise from various reasons, such as incomplete data collection, sensor errors, or data entry issues. If not handled properly, missing data can significantly degrade model performance or cause errors during training and testing.

Why Handling Missing Data is Important:

1. Maintaining Data Integrity: Missing values can skew analysis results, leading to incorrect conclusions and poor model performance.
2. Ensuring Model Robustness: Most machine learning models (especially algorithms like linear regression, logistic regression, or SVM) cannot handle missing values directly, so preprocessing is necessary to ensure smooth execution.
3. Preserving Dataset Size: Rather than discarding records with missing values, handling them appropriately allows you to use as much of your data as possible.

Different Techniques for Handling Missing Data:

1. Removing Missing Data:
   - Dropping Rows: If the number of rows with missing values is small, you can drop those rows.
   - Dropping Columns: If an entire column has too many missing values (e.g., more than 50%), you may choose to drop that feature.
2. Imputation (Filling Missing Data):
   - Mean/Median/Mode Imputation: Replace missing values with the mean, median, or mode of the non-missing values in that column.
   - Forward/Backward Fill: Use previous or next values to fill missing entries (mainly used in time-series data).
   - Predictive Imputation: Use a machine learning model to predict the missing values based on other features.
   - K-Nearest Neighbors (KNN) Imputation: Use KNN to find similar data points and fill missing values based on the nearest neighbors.
3. Flagging/Indicator Variables:

  - Create a new binary feature indicating whether the value in the original column was missing or not. This approach allows the model to know which data points had missing values.

4. Using Algorithms that Handle Missing Data:
   - Some machine learning algorithms like decision trees and random forests can handle missing values internally without requiring imputation.

Practical Considerations:

1. Nature of Missing Data: Before choosing a method, understand the nature of missing data. Is it Missing Completely at Random (MCAR), Missing at Random (MAR), or Not Missing at Random (NMAR)? Different techniques are better suited for different types of missingness.
2. Preserving Relationships: While imputation preserves data, it may introduce bias or distort relationships between features, especially if the missing data is not random..
3. Multiple Imputation: For more advanced scenarios, techniques like Multiple Imputation (e.g., MICE - Multiple Imputation by Chained Equations) are used to generate multiple datasets with different imputations, combining their results to reduce imputation bias.

Summary:

- Removing Missing Data: Drop rows or columns with missing values if the missing proportion is small.
- Imputation: Replace missing values with mean, median, mode, or more advanced techniques like KNN or predictive imputation.
- Flagging: Create binary flags for missing values to allow the model to capture this information.
- Advanced Algorithms: Some algorithms like decision trees and XGBoost handle missing values naturally without preprocessing.

[1]:
```
import pandas as pd
import numpy as np

miles = pd.DataFrame({'farthest_run_mi': [50,62, np.nan,100,26,13,31,50]})
miles
```

[1]:
```
   farthest_run_mi
0             50.0
1             62.0
2              NaN
3            100.0
4             26.0
5             13.0
6             31.0
7             50.0
```

[2]:
```
miles.isna().sum()
```

[2]:
```
farthest_run_mi    1
dtype: int64
```

[3]:
```python
from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer(strategy='mean')
imp_mean.fit_transform(miles)
```

[3]:
```
array([[ 50.        ],
       [ 62.        ],
       [ 47.42857143],
       [100.        ],
       [ 26.        ],
       [ 13.        ],
       [ 31.        ],
       [ 50.        ]])
```

[4]:
```python
imp_median = SimpleImputer(strategy='median')
imp_median.fit_transform(miles)
```

[4]:
```
array([[ 50.],
       [ 62.],
       [ 50.],
       [100.],
       [ 26.],
       [ 13.],
       [ 31.],
       [ 50.]])
```

[5]:
```python
imp_mode = SimpleImputer(strategy='most_frequent')
imp_mode.fit_transform(miles)
```

[5]:
```
array([[ 50.],
       [ 62.],
       [ 50.],
       [100.],
       [ 26.],
       [ 13.],
       [ 31.],
       [ 50.]])
```

[6]:
```python
imp_constant = SimpleImputer(strategy='constant', fill_value=13)
imp_constant.fit_transform(miles)
```

[6]:
```
array([[ 50.],
       [ 62.],
       [ 13.],
       [100.],
       [ 26.],
        [ 13.],
        [ 31.],
        [ 50.]])
```

```
[7]: names = pd.DataFrame({'names':['Ryan', 'Nolan', 'Honus', 'Wagner', np.nan,␣
      ↪'Ruth']})
     names
```

```
[7]:     names
     0     Ryan
     1    Nolan
     2    Honus
     3   Wagner
     4      NaN
     5     Ruth
```

```
[8]: imp_constant_cat = SimpleImputer(strategy='constant', fill_value='babe')
     imp_constant_cat.fit_transform(names)
```

```
[8]: array([['Ryan'],
            ['Nolan'],
            ['Honus'],
            ['Wagner'],
            ['babe'],
            ['Ruth']], dtype=object)
```

```
[9]: imp_mean_marked = SimpleImputer(strategy='mean', add_indicator= True)
     imp_mean_marked.fit_transform(miles)
```

```
[9]: array([[ 50.        ,   0.        ],
            [ 62.        ,   0.        ],
            [ 47.42857143,   1.        ],
            [100.        ,   0.        ],
            [ 26.        ,   0.        ],
            [ 13.        ,   0.        ],
            [ 31.        ,   0.        ],
            [ 50.        ,   0.        ]])
```

```
[13]: new_df = pd.DataFrame({
          'Name': ['Ryan', 'Nolan', 'Walter', 'Honus', 'Christy', np.nan, 'Napoleon',␣
       ↪'Tris'],
          'farthest_run_mi': [50,62, np.nan,100,26,13,31,50]
      })
      new_df
```

```
[13]:       Name  farthest_run_mi
      0      Ryan             50.0
```

```
1     Nolan             62.0
2    Walter              NaN
3     Honus            100.0
4   Christy             26.0
5       NaN             13.0
6  Napoleon             31.0
7      Tris             50.0
```

[14]:
```
from sklearn.compose import make_column_transformer
ct = make_column_transformer(
    (imp_constant_cat, ['Name']),
    (imp_mean, ['farthest_run_mi']),
    remainder='drop'
)
ct.set_output(transform='pandas')
```

[14]:
```
ColumnTransformer(transformers=[('simpleimputer-1',
                                 SimpleImputer(fill_value='babe',
                                               strategy='constant'),
                                 ['Name']),
                                ('simpleimputer-2', SimpleImputer(),
                                 ['farthest_run_mi'])])
```

[15]:
```
df_pandas = ct.fit_transform(new_df)
df_pandas
```

[15]:
```
  simpleimputer-1__Name  simpleimputer-2__farthest_run_mi
0                  Ryan                         50.000000
1                 Nolan                         62.000000
2                Walter                         47.428571
3                 Honus                        100.000000
4               Christy                         26.000000
5                  babe                         13.000000
6              Napoleon                         31.000000
7                  Tris                         50.000000
```

6-data_preprocessing_with_column_transformer

October 20, 2024

1 Data Preprocessing With Column Transformer

The ColumnTransformer in Python's sklearn library is a powerful tool that allows you to apply different preprocessing steps to different columns of your dataset. This is particularly useful when you have a mixture of numerical and categorical data and need to apply distinct transformations to each type of data. Using the ColumnTransformer, you can combine transformations like scaling numerical features, encoding categorical variables, and imputing missing values, all in a single unified process.

Why Use ColumnTransformer?

1. **Multiple Data Types**: It handles datasets with mixed data types (numerical, categorical, text, etc.).
2. **Efficient Pipeline**: You can combine different preprocessing steps into a pipeline, making the code cleaner and easier to maintain.
3. **Selective Preprocessing**: Apply different preprocessing steps to specific columns, instead of preprocessing the entire dataset uniformly.
4. **Integration with Pipelines**: It can be combined with machine learning models in a Pipeline, allowing for a streamlined workflow from preprocessing to model training and evaluation.

[2]:
```
import pandas as pd

d = {'sales':␣
  ↪[100000,222000,1000000,522000,111111,222222,1111111,20000,75000,90000,1000000,10000],␣
  ↪'city':␣
  ↪['Tampa','Tampa','Orlando','Jacksonville','Miami','Jacksonville','Miami','Miami','Orlando',
  ↪'size': ['Small',␣
  ↪'Medium','Large','Large','Small','Medium','Large','Small','Medium','Medium','Medium','Small

df = pd.DataFrame(data=d)
df
```

[2]:
```
     sales          city    size
0   100000         Tampa   Small
1   222000         Tampa  Medium
2  1000000       Orlando   Large
```

```
3    522000  Jacksonville   Large
4    111111         Miami   Small
5    222222  Jacksonville  Medium
6   1111111         Miami   Large
7     20000         Miami   Small
8     75000       Orlando  Medium
9     90000       Orlando  Medium
10  1000000       Orlando  Medium
11    10000       Orlando   Small
```

[7]:
```
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
from sklearn.compose import make_column_transformer

ohe = OneHotEncoder(sparse_output=False)
ode = OrdinalEncoder()

ct = make_column_transformer(
    (ohe, ['city']),
    (ode, ['size']),
    remainder='passthrough'
)

ct.set_output(transform='pandas')
```

[7]:
```
ColumnTransformer(remainder='passthrough',
                  transformers=[('onehotencoder',
                                 OneHotEncoder(sparse_output=False), ['city']),
                                ('ordinalencoder', OrdinalEncoder(), ['size'])])
```

[8]:
```
df_pandas = ct.fit_transform(df)
df_pandas
```

[8]:
```
    onehotencoder__city_Jacksonville  onehotencoder__city_Miami  \
0                                0.0                        0.0
1                                0.0                        0.0
2                                0.0                        0.0
3                                1.0                        0.0
4                                0.0                        1.0
5                                1.0                        0.0
6                                0.0                        1.0
7                                0.0                        1.0
8                                0.0                        0.0
9                                0.0                        0.0
10                               0.0                        0.0
11                               0.0                        0.0

    onehotencoder__city_Orlando  onehotencoder__city_Tampa  \
0                          0.0                         1.0
1                          0.0                         1.0
2                          1.0                         0.0
3                          0.0                         0.0
4                          0.0                         0.0
5                          0.0                         0.0
6                          0.0                         0.0
7                          0.0                         0.0
8                          1.0                         0.0
9                          1.0                         0.0
10                         1.0                         0.0
11                         1.0                         0.0

    ordinalencoder__size  remainder__sales
0                    2.0            100000
1                    1.0            222000
2                    0.0           1000000
3                    0.0            522000
4                    2.0            111111
5                    1.0            222222
6                    0.0           1111111
7                    2.0             20000
8                    1.0             75000
9                    1.0             90000
10                   1.0           1000000
11                   2.0             10000
```

[9]:
```
ct2 = make_column_transformer(
    (ohe, [1]),
    (ode, [2]),
    remainder='drop'
)

ct2.set_output(transform='pandas')
```

[9]:
```
ColumnTransformer(transformers=[('onehotencoder',
                                 OneHotEncoder(sparse_output=False), [1]),
                                ('ordinalencoder', OrdinalEncoder(), [2])])
```

[11]:
```
df_pandas2 = ct2.fit_transform(df)
df_pandas2
```

[11]:
```
    onehotencoder__city_Jacksonville  onehotencoder__city_Miami  \
0                                0.0                        0.0
1                                0.0                        0.0
2                                0.0                        0.0
3                                1.0                        0.0
```

```
4                                0.0                          1.0
5                                1.0                          0.0
6                                0.0                          1.0
7                                0.0                          1.0
8                                0.0                          0.0
9                                0.0                          0.0
10                               0.0                          0.0
11                               0.0                          0.0

    onehotencoder__city_Orlando  onehotencoder__city_Tampa  \
0                           0.0                        1.0
1                           0.0                        1.0
2                           1.0                        0.0
3                           0.0                        0.0
4                           0.0                        0.0
5                           0.0                        0.0
6                           0.0                        0.0
7                           0.0                        0.0
8                           1.0                        0.0
9                           1.0                        0.0
10                          1.0                        0.0
11                          1.0                        0.0

    ordinalencoder__size
0                    2.0
1                    1.0
2                    0.0
3                    0.0
4                    2.0
5                    1.0
6                    0.0
7                    2.0
8                    1.0
9                    1.0
10                   1.0
11                   2.0
```

[12]:
```
ct3 = make_column_transformer(
    (ohe, [1]),
    ('passthrough', ['size']),
    remainder='drop'
)

ct3.set_output(transform='pandas')
```

[12]:
```
ColumnTransformer(transformers=[('onehotencoder',
                                 OneHotEncoder(sparse_output=False), [1]),
                                ('passthrough', 'passthrough', ['size'])])
```

[13]:
```
ct3.fit_transform(df)
```

[13]:
```
    onehotencoder__city_Jacksonville  onehotencoder__city_Miami  \
0                                0.0                        0.0
1                                0.0                        0.0
2                                0.0                        0.0
3                                1.0                        0.0
4                                0.0                        1.0
5                                1.0                        0.0
6                                0.0                        1.0
7                                0.0                        1.0
8                                0.0                        0.0
9                                0.0                        0.0
10                               0.0                        0.0
11                               0.0                        0.0

    onehotencoder__city_Orlando  onehotencoder__city_Tampa passthrough__size
0                           0.0                        1.0             Small
1                           0.0                        1.0            Medium
2                           1.0                        0.0             Large
3                           0.0                        0.0             Large
4                           0.0                        0.0             Small
5                           0.0                        0.0            Medium
6                           0.0                        0.0             Large
7                           0.0                        0.0             Small
8                           1.0                        0.0            Medium
9                           1.0                        0.0            Medium
10                          1.0                        0.0            Medium
11                          1.0                        0.0             Small
```

7-KNN

October 20, 2024

1 K Nearest Neighbor

K-Nearest Neighbor (KNN) is a supervised machine learning algorithm used for classification and regression tasks. It classifies a new data point based on how its neighbors (the closest data points in the feature space) are classified. In essence, KNN assigns the class of the majority of its K nearest neighbors to the new point. It's a non-parametric and lazy learning algorithm, meaning it makes decisions based on the entire dataset rather than assuming a specific structure (non-parametric), and it doesn't explicitly learn a model during training (lazy learning).

- **Non-parametric**: No assumptions about the data distribution.
- **Lazy learning**: The model simply stores the data and waits until a query (test data) is made before computing anything.

When to Use KNN? KNN is suitable when you have a labeled dataset and you want to classify new, unseen data. However, it is most effective under the following conditions:

- **Small to medium-sized datasets**: KNN can become computationally expensive with large datasets, as it requires scanning through all data points to find the neighbors.
- **Low-dimensional data**: With high-dimensional data, the distance between data points tends to increase, making it harder for the algorithm to distinguish between neighbors (this is known as the curse of dimensionality).
- **Non-linear decision boundaries**: If the decision boundaries between classes are not linear (i.e., they can't be split by a line or hyperplane), KNN can work well since it uses local information. Balanced classes: KNN works best when the number of samples in each class is roughly equal.

How Does KNN Work?

1. **hoose K (the number of neighbors)**: This is a user-defined parameter. Typically, an odd number is chosen to avoid ties in classification. For example, K=3 means the algorithm looks at the 3 nearest neighbors of a data point.
2. **Calculate Distance**: For a new data point, the algorithm calculates the distance between this point and all other points in the training dataset. The most common distance metric used is Euclidean distance.

Other distance metrics can be used, like Manhattan distance or Minkowski distance, depending on the dataset and the problem.

3. **Find the K Nearest Neighbors**: After calculating the distance between the new data point and every other point in the dataset, KNN selects the K closest points.

4. Assign a Class or Predict Value:

   - **For classification**: The class label that appears most frequently among the K nearest neighbors is assigned to the new data point.
   - **For regression**: The mean (or sometimes median) of the values of the K nearest neighbors is taken as the predicted value for the new data point.

5. **Result**: The algorithm outputs either the predicted class label (for classification) or a predicted value (for regression).

Advantages of KNN:

- **Simple to implement**: KNN requires no assumptions about the underlying data, making it easy to use.
- **Flexible**: Can be used for both classification and regression problems.
- **No training phase**: As a lazy learner, KNN does not require a separate training phase, making it efficient in terms of time before making predictions.

Disadvantages of KNN:

- **Computationally expensive**: As it compares every new point with all points in the training data, it can be slow with large datasets.
- **Memory-intensive**: Since the entire dataset needs to be stored, it requires more memory as the dataset grows.
- **Sensitive to irrelevant features**: If the dataset has many irrelevant features, the distance metric may be distorted, leading to poor performance.
- **Sensitive to the choice of K**: A poor choice of K can lead to underfitting (if K is too large) or overfitting (if K is too small).

Real-World Applications:

1. **Image Recognition**: KNN can classify images based on similarity, such as facial recognition or object classification.
2. **Recommender Systems**: KNN can be used to find similar items to recommend to users based on their preferences.
3. **Medical Diagnosis**: It's used to classify diseases based on the symptoms and historical data of patients.
4. **Finance**: KNN is used to detect outliers or categorize financial transactions (e.g., fraud detection).
5. **Document Classification**: Used in text mining to classify documents into predefined categories.

[1]:
```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, classification_report
```

[2]:
```
df = pd.read_csv('500hits.csv', encoding="latin-1")
df.head()
```

[2]:
```
          PLAYER  YRS     G     AB     R     H   2B   3B   HR   RBI    BB  \
0        Ty Cobb   24  3035  11434  2246  4189  724  295  117   726  1249
1     Stan Musial   22  3026  10972  1949  3630  725  177  475  1951  1599
2   Tris Speaker   22  2789  10195  1882  3514  792  222  117   724  1381
3    Derek Jeter   20  2747  11195  1923  3465  544   66  260  1311  1082
4  Honus Wagner   21  2792  10430  1736  3430  640  252  101     0   963

     SO   SB   CS     BA  HOF
0   357  892  178  0.366    1
1   696   78   31  0.331    1
2   220  432  129  0.345    1
3  1840  358   97  0.310    1
4   327  722   15  0.329    1
```

[3]:
```
df = df.drop(columns=['PLAYER', 'CS'])
df.head()
```

[3]:
```
   YRS     G     AB     R     H   2B   3B   HR   RBI    BB    SO   SB     BA  \
0   24  3035  11434  2246  4189  724  295  117   726  1249   357  892  0.366
1   22  3026  10972  1949  3630  725  177  475  1951  1599   696   78  0.331
2   22  2789  10195  1882  3514  792  222  117   724  1381   220  432  0.345
3   20  2747  11195  1923  3465  544   66  260  1311  1082  1840  358  0.310
4   21  2792  10430  1736  3430  640  252  101     0   963   327  722  0.329

   HOF
0    1
1    1
2    1
3    1
4    1
```

[6]:
```
X = df.iloc[:,0:13]
y = df.iloc[:,13]
```

[7]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=11,␣
 ↪test_size=0.2)

scaler = MinMaxScaler(feature_range=(0,1))
```

```
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

[10]:
```
knn = KNeighborsClassifier(n_neighbors=8)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
y_pred
```

[10]:
```
array([0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0,
       0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0,
       1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0,
       0, 0, 0, 1, 1], dtype=int64)
```

[11]:
```
knn.score(X_test, y_test)
```

[11]:
```
0.8279569892473119
```

1.1 Confusion Matrix

[12]:
```
cm = confusion_matrix(y_test, y_pred)
cm
```

[12]:
```
array([[55, 12],
       [ 4, 22]], dtype=int64)
```

1.2 Classification Report

[15]:
```
cr = classification_report(y_test, y_pred)
print(cr)
```

```
              precision    recall  f1-score   support

           0       0.93      0.82      0.87        67
           1       0.65      0.85      0.73        26

    accuracy                           0.83        93
   macro avg       0.79      0.83      0.80        93
weighted avg       0.85      0.83      0.83        93
```

[16]:
```
print(knn.n_samples_fit_)
```

```
372
```

8-Decision_Tree

October 20, 2024

1 Decision Tree

A Decision Tree is a supervised machine learning algorithm used for classification and regression tasks. It works by breaking down a dataset into smaller and smaller subsets while incrementally developing a tree structure. Each internal node of the tree represents a test or decision based on an attribute (feature), each branch represents the outcome of the decision, and each leaf node represents a class label (in classification) or a continuous value (in regression).

- **Root node**: The top node, which represents the entire dataset and is split based on the most important feature.
- **Internal nodes**: Represent the attributes or features that are used to split the data.
- **Leaf nodes**: Represent the final class or output value.

The main goal of a Decision Tree is to create a model that predicts the target value by learning simple decision rules inferred from the data features.

When to Use Decision Trees? Decision Trees are useful when you want a model that:

- **Can handle both categorical and numerical data**: Decision Trees are flexible and can work well with different types of data.
- **Is easy to interpret**: Since Decision Trees mimic human decision-making processes, they are intuitive and easy to understand, even for non-experts.
- **Requires minimal data preprocessing**: Unlike many other algorithms, Decision Trees do not require normalization or scaling of data.
- **Can model non-linear relationships**: Decision Trees can capture complex patterns in data, including interactions between features.

However, Decision Trees are especially effective when:

- The dataset is not too large, as deep trees can become computationally expensive.
- There is a need for a simple, interpretable model.
- You are working with a problem that involves a sequence of decision

How Does a Decision Tree Work?

1. **Feature Selection and Splitting**: The tree starts with all the data at the root node. It then selects a feature and splits the data based on this feature to form child nodes. The feature is chosen by evaluating different splitting criteria (more on this later).

2. **Recursive Partitioning**: This process of splitting the data continues recursively at each child node, selecting the best feature at each step until one of the following conditions is met:

   - All samples at a node belong to the same class (for classification).
   - The node reaches a pre-defined depth (maximum depth of the tree).
   - The number of samples at a node is less than the minimum split size.

3. **Prediction**: Once the tree has been constructed, it can be used to classify new samples by passing them down the tree, following the decisions at each node until a leaf node is reached. The class label (or value for regression) at the leaf node is the model's prediction.

Who Should Use Decision Trees? Decision Trees are ideal for:

- **Business analysts and decision-makers**: The model is easy to interpret and can provide insights into the data and important decision points.
- **Data scientists and machine learning engineers** who need to solve classification or regression problems.
- **Researchers** in fields like biology, medicine, or finance who deal with complex datasets involving interactions between multiple features.

Advantages of Decision Trees:

- **Easy to interpret**: Even non-technical people can understand the decision process.
- **Handles both numerical and categorical data**: Can work with different types of features without requiring feature transformation.
- **No need for feature scaling**: Unlike algorithms like SVM or KNN, Decision Trees don't require scaling.
- **Can capture non-linear relationships**: Decision Trees can split data at any point, allowing them to model complex relationships.

Disadvantages of Decision Trees:

- **Prone to overfitting**: Decision Trees can create very complex models that overfit the training data, especially when the tree grows too deep.
- **Unstable**: Small changes in the data can result in significantly different trees.
- **Bias towards dominant classes**: In unbalanced datasets, the tree might be biased toward the dominant class.

To avoid these issues, Decision Trees are often pruned (cutting off branches that do not provide additional information) or ensemble methods like Random Forest or Gradient Boosting are used to create a more robust model.

Real-World Applications:

1. **Loan Default Prediction**: Financial institutions use Decision Trees to classify loan applicants based on their risk profile.
2. **Medical Diagnosis**: Decision Trees help in diagnosing diseases based on patient symptoms and medical records.
3. **Customer Churn Prediction**: Companies use Decision Trees to predict if a customer will leave based on historical data.
4. **Marketing**: Decision Trees are used to segment customers based on purchasing behavior and demographics.
5. **Fraud Detection**: Decision Trees help identify fraudulent transactions by learning decision rules from historical data.

[142]:
```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix, classification_report

df = pd.read_csv('500hits.csv', encoding="latin-1")
df.head()
```

[142]:
```
         PLAYER  YRS     G     AB     R     H   2B   3B   HR   RBI    BB  \
0       Ty Cobb   24  3035  11434  2246  4189  724  295  117   726  1249
1   Stan Musial   22  3026  10972  1949  3630  725  177  475  1951  1599
2  Tris Speaker   22  2789  10195  1882  3514  792  222  117   724  1381
3   Derek Jeter   20  2747  11195  1923  3465  544   66  260  1311  1082
4  Honus Wagner   21  2792  10430  1736  3430  640  252  101     0   963

     SO   SB   CS     BA  HOF
0   357  892  178  0.366    1
1   696   78   31  0.331    1
2   220  432  129  0.345    1
3  1840  358   97  0.310    1
4   327  722   15  0.329    1
```

[143]:
```python
df = df.drop(columns=['PLAYER', 'CS'])
df.head()
```

[143]:
```
   YRS     G     AB     R     H   2B   3B   HR   RBI    BB    SO   SB     BA  \
0   24  3035  11434  2246  4189  724  295  117   726  1249   357  892  0.366
1   22  3026  10972  1949  3630  725  177  475  1951  1599   696   78  0.331
2   22  2789  10195  1882  3514  792  222  117   724  1381   220  432  0.345
3   20  2747  11195  1923  3465  544   66  260  1311  1082  1840  358  0.310
4   21  2792  10430  1736  3430  640  252  101     0   963   327  722  0.329

   HOF
```

```
0    1
1    1
2    1
3    1
4    1
```

[144]:
```
X = df.iloc[:, 0:13]
y = df.iloc[:, 13]

X_train, X_test, y_train, y_test = train_test_split(X,y, random_state=17,␣
 ↪test_size=0.2)

dtc = DecisionTreeClassifier()

dtc.get_params()
```

[144]:
```
{'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'random_state': None,
 'splitter': 'best'}
```

[145]:
```
dtc.fit(X_train, y_train)
```

[145]:
```
DecisionTreeClassifier()
```

[146]:
```
y_prediction = dtc.predict(X_test)

print(confusion_matrix(y_test, y_prediction))
```

```
[[52  9]
 [11 21]]
```

[147]:
```
print(classification_report(y_test, y_prediction))
```

```
              precision    recall  f1-score   support

           0       0.83      0.85      0.84        61
           1       0.70      0.66      0.68        32

    accuracy                           0.78        93
```

```
   macro avg       0.76      0.75      0.76        93
weighted avg       0.78      0.78      0.78        93
```

[148]:
```
dtc.feature_importances_
```

[148]:
```
array([0.        , 0.02598978, 0.03173403, 0.03633493, 0.39759474,
       0.06589131, 0.01565832, 0.05810452, 0.04515849, 0.12784069,
       0.04098071, 0.05207056, 0.10264192])
```

[151]:
```
features = pd.DataFrame(dtc.feature_importances_, index=X.columns)
features
```

[151]:
```
            0
YRS  0.000000
G    0.025990
AB   0.031734
R    0.036335
H    0.397595
2B   0.065891
3B   0.015658
HR   0.058105
RBI  0.045158
BB   0.127841
SO   0.040981
SB   0.052071
BA   0.102642
```

[156]:
```
dtc2 = DecisionTreeClassifier(criterion='entropy', ccp_alpha=0.04)

dtc2.fit(X_train, y_train)

y_pred2 = dtc2.predict(X_test)

print(confusion_matrix(y_test, y_pred2))

print(classification_report(y_test, y_pred2))
```

```
[[50 11]
 [ 9 23]]
              precision    recall  f1-score   support

           0       0.85      0.82      0.83        61
           1       0.68      0.72      0.70        32

    accuracy                           0.78        93
   macro avg       0.76      0.77      0.77        93
weighted avg       0.79      0.78      0.79        93
```

```
[157]: features2 = pd.DataFrame(dtc2.feature_importances_, index=X.columns)
       features2
```

```
[157]:             0
       YRS  0.000000
       G    0.000000
       AB   0.000000
       R    0.000000
       H    0.837977
       2B   0.000000
       3B   0.000000
       HR   0.000000
       RBI  0.000000
       BB   0.000000
       SO   0.000000
       SB   0.000000
       BA   0.162023
```

9-Random_Forest

October 20, 2024

1 Random Forest

Random Forest is a supervised learning algorithm that is used for both classification and regression tasks. It builds multiple decision trees (hence the term "forest") and combines their outputs to make a final prediction. It is an ensemble method, meaning it aggregates the predictions of many base models (in this case, decision trees) to create a more robust and accurate model.

Each tree in the Random Forest is trained on a different random subset of the data, and the splits within each tree are made using random subsets of features. This randomness helps the model avoid overfitting and improves generalization to unseen data.

- **For classification**, Random Forest uses a majority vote to determine the class label.
- **For regression**, it averages the predictions from all trees.

When to Use Random Forest? Random Forest is versatile and can be used in many situations, but it is especially effective when:

- **You need a powerful model that handles both classification and regression problems**.
- **The dataset has many features**: Random Forest can handle high-dimensional datasets and can even provide feature importance, helping identify the most influential variables.
- **The dataset is large**: Unlike a single decision tree, Random Forest can handle large datasets more effectively because it mitigates overfitting by averaging the predictions of many trees.
- **The data has missing values**: Random Forest can handle datasets with some missing values without requiring extensive preprocessing.
- **You need a model that can generalize well to unseen data**: Random Forest is often one of the best choices in terms of accuracy and robustness because it reduces overfitting.

How Does Random Forest Work?

1. **Create Multiple Subsets (Bootstrapping)**: Random Forest randomly samples the training data with replacement to create multiple different training subsets (called bootstrap samples). Each decision tree is trained on one of these subsets.
2. **Random Feature Selection (Feature Bagging)**: At each split in a tree, Random Forest only considers a random subset of the features rather than all of them. This ensures that individual trees are diverse and reduces the risk that some features dominate the model.

3. **Build Decision Trees**: Each bootstrap sample is used to build a decision tree. Each tree is grown to its maximum depth (typically unpruned), and the algorithm selects the best split from the subset of features at each node.

4. **Make Predictions**:
   - **For classification**: After training all the trees, Random Forest classifies new data points by making each tree "vote" on the predicted class. The final classification is the class with the most votes.
   - **For regression**: The prediction for each data point is the average of the predictions from all trees.

5. **Aggregate Predictions**: Random Forest aggregates the results of all the trees to make a final prediction. By combining the output of multiple decision trees, Random Forest can achieve better accuracy and stability than any single tree.

Who Should Use Random Forest? Random Forest is suitable for:

- **Data scientists and machine learning engineers** who need a strong baseline model.
- **Business analysts and decision-makers** who want to interpret feature importance or detect important decision-making factors.
- **Researchers and practitioners** in fields like finance, healthcare, and bioinformatics, where datasets can be large, complex, and noisy.
- **Beginners in machine learning** who want an easy-to-use and effective algorithm without requiring much fine-tuning.

Advantages of Random Forest:

- **Reduces overfitting**: By averaging the predictions of multiple decision trees, Random Forest mitigates overfitting that often occurs with a single decision tree.
- **Handles large datasets well**: Random Forest can scale to large datasets and many features effectively.
- **Works with both classification and regression problems**: It is versatile and can handle different types of machine learning problems.
- **Feature importance**: Random Forest can provide insights into which features are most important for making predictions.
- **Robust to outliers**: Since individual trees are built on random subsets of data, outliers have less impact on the final model.

Disadvantages of Random Forest:

- **Slower and more computationally expensive**: Since it trains multiple decision trees, Random Forest can be slower and require more memory compared to simpler models like decision trees or linear models.
- **Less interpretable**: While decision trees are easy to interpret, a Random Forest with hundreds of trees is more difficult to explain and understand.

- **Risk of overfitting in noisy data**: While it generally reduces overfitting, Random Forest can still overfit if there is too much noise in the data or if there are too many trees.

Real-World Applications:

- **Credit Scoring and Risk Assessment**: Banks and financial institutions use Random Forest to assess the risk of loan applicants defaulting based on their historical data.
- **Customer Churn Prediction**: Random Forest helps predict which customers are likely to leave a company based on their behavior and interaction history.
- **Fraud Detection**: Random Forest can identify fraudulent transactions by learning patterns from historical financial transaction data.
- **Healthcare and Medical Diagnosis**: Random Forest is used to classify patients based on symptoms, helping doctors make diagnostic decisions.
- **Stock Market Prediction**: Random Forest can be used to predict stock prices based on historical trends and market conditions.

[1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

df = pd.read_csv('500hits.csv', encoding="latin-1")
df.head()
```

[1]:
```
         PLAYER  YRS     G     AB     R     H   2B   3B   HR   RBI    BB  \
0       Ty Cobb   24  3035  11434  2246  4189  724  295  117   726  1249
1   Stan Musial   22  3026  10972  1949  3630  725  177  475  1951  1599
2  Tris Speaker   22  2789  10195  1882  3514  792  222  117   724  1381
3   Derek Jeter   20  2747  11195  1923  3465  544   66  260  1311  1082
4  Honus Wagner   21  2792  10430  1736  3430  640  252  101     0   963

     SO   SB   CS     BA  HOF
0   357  892  178  0.366    1
1   696   78   31  0.331    1
2   220  432  129  0.345    1
3  1840  358   97  0.310    1
4   327  722   15  0.329    1
```

[2]:
```
df = df.drop(columns=['PLAYER', 'CS'])

X = df.iloc[:, 0:13]
y = df.iloc[:, 13]
```

[3]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, random_state=17,␣
 ↪test_size=0.2)
```

```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)
y_pred
```

```
[3]: array([0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0,
            0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,
            1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
            0, 0, 1, 0, 0], dtype=int64)
```

```
[4]: rf.score(X_test, y_test)
```

```
[4]: 0.8279569892473119
```

```
[5]: print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.85      0.90      0.87        61
           1       0.79      0.69      0.73        32

    accuracy                           0.83        93
   macro avg       0.82      0.79      0.80        93
weighted avg       0.83      0.83      0.82        93
```

```
[6]: features = pd.DataFrame(rf.feature_importances_, index=X.columns)
     features
```

```
[6]:            0
     YRS  0.028679
     G    0.082555
     AB   0.081906
     R    0.117477
     H    0.129573
     2B   0.061198
     3B   0.047405
     HR   0.055006
     RBI  0.109791
     BB   0.048961
     SO   0.041604
     SB   0.046483
     BA   0.149363
```

```
[7]: rf2 = RandomForestClassifier(
         n_estimators=1000,
         criterion='entropy',
         min_samples_split=10,
         max_depth=14, random_state=42
     )

     rf2.fit(X_train, y_train)
```

[7]:
```
RandomForestClassifier(criterion='entropy', max_depth=14, min_samples_split=10,
                       n_estimators=1000, random_state=42)
```

[8]:
```
rf2.score(X_test, y_test)
```

[8]:
```
0.8494623655913979
```

[9]:
```
y_pred2 = rf2.predict(X_test)
print(classification_report(y_test, y_pred2))
```

```
              precision    recall  f1-score   support

           0       0.85      0.93      0.89        61
           1       0.85      0.69      0.76        32

    accuracy                           0.85        93
   macro avg       0.85      0.81      0.82        93
weighted avg       0.85      0.85      0.85        93
```

10-SVM

October 20, 2024

1 Support Vector Machine

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression tasks, though it is mostly used for classification. The goal of SVM is to find a hyperplane in an N-dimensional space (where N is the number of features) that distinctly classifies the data points. The best hyperplane is the one that maximizes the margin between the two classes, meaning it is as far as possible from the nearest data points of each class.

- **Support vectors** are the data points that are closest to the hyperplane. They are the most critical elements of the dataset because they define the decision boundary.
- **Margin** is the distance between the hyperplane and the nearest data points from either class. SVM seeks to maximize this margin to improve model generalization.

When to Use Support Vector Machine? SVM is particularly useful in scenarios where:

- **You have small or medium-sized datasets**: SVM works best on smaller datasets as it can be computationally expensive for large datasets.
- **The data is linearly separable**: SVM is most effective when the data can be clearly separated into two distinct classes by a straight line (in 2D) or a hyperplane (in higher dimensions).
- **You need a model that can handle high-dimensional data**: SVM is capable of performing well even when the number of features exceeds the number of samples, making it suitable for problems like text classification.
- **The classes are well separated**: SVM works well when there is a clear margin between the classes.
- **You need to avoid overfitting**: SVM focuses on maximizing the margin between classes, which often helps avoid overfitting in classification tasks.

How Does Support Vector Machine Work?

1. **Hyperplane Calculation**: In a binary classification problem, the goal of SVM is to find a hyperplane that best separates the data into two classes. In two-dimensional space, this hyperplane is a line, but in higher-dimensional space, it becomes a plane or hyperplane.
2. **Maximizing the Margin**: SVM tries to find the hyperplane that maximizes the margin between the two classes. The margin is the distance between the hyperplane and the closest

data points from each class (called support vectors). The larger the margin, the better the generalization ability of the classifier.

3. **Linear vs. Non-Linear Classification**:
   - **Linear SVM**: If the data is linearly separable, SVM will find the optimal hyperplane that separates the two classes.
   - **Non-Linear SVM (Kernel SVM)**: In many real-world problems, the data is not linearly separable. SVM uses a technique called the kernel trick to map the data into a higher-dimensional space where a linear separation becomes possible. Some commonly used kernels include:
     - **Polynomial kernel**: For capturing polynomial relationships between features.
     - **Radial Basis Function (RBF) kernel (Gaussian kernel)**: Useful for complex, non-linear relationships.
4. **Soft Margin vs. Hard Margin**:
   - **Hard margin SVM**: Enforces that no data points from either class are allowed to be within the margin or misclassified. This is rarely used because real-world data often contains noise and overlaps.
   - **Soft margin SVM**: Allows some misclassification or violation of the margin (controlled by a parameter C) to account for noisy or overlapping data.
5. **Classification and Prediction**: Once the SVM model has found the optimal hyperplane, new data points are classified by determining on which side of the hyperplane they fall.

Who Should Use Support Vector Machine? SVM is a good choice for:

- **Researchers and practitioners** in fields like bioinformatics, image recognition, and text categorization, where datasets might be small but have many features.
- **Data scientists** looking for a robust classifier that works well in high-dimensional spaces.
- **Machine learning engineers** dealing with classification problems where the data is non-linearly separable but a strong decision boundary is needed.
- **Business analysts** who need an accurate and interpretable model when the classes are well-separated.###

Advantages of Support Vector Machine:

- **Effective in high-dimensional spaces**: SVM works well when the number of features is large, even more than the number of samples.
- **Robust to overfitting**: By maximizing the margin, SVM often generalizes well to unseen data, especially with appropriate tuning of the C parameter.
- **Flexibility with non-linear data**: With the use of kernels, SVM can effectively classify data that is not linearly separable.
- **Memory efficient**: SVM uses only a subset of the training points (the support vectors), making it efficient in memory usage.

Disadvantages of Support Vector Machine:

- **Computationally expensive**: SVM can be slow and memory-intensive, especially for large datasets.
- **Hard to interpret**: SVMs are not as easy to interpret as models like decision trees or logistic regression.
- **Requires careful tuning of parameters**: The performance of SVM depends on the choice of kernel, regularization parameter C, and other hyperparameters. Poor tuning can lead to suboptimal results.
- **Not suitable for large datasets**: SVM can struggle with very large datasets because of its computational complexity.

Real-World Applications:

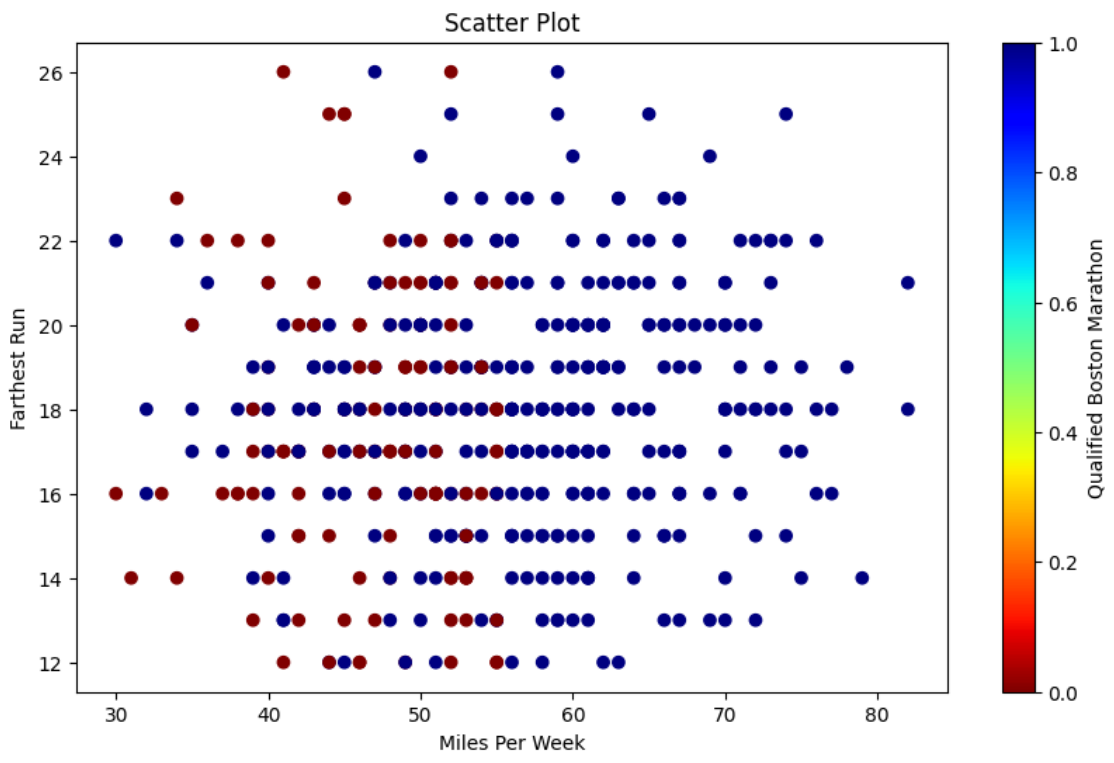
- Image Recognition: SVM is widely used in image classification tasks, such as handwriting recognition or face detection.
- Bioinformatics: SVMs are used to classify proteins and genes, making them useful in genomics and biological sequence analysis.
- Text Classification: SVM is commonly used in text categorization (e.g., spam detection) because of its ability to handle high-dimensional data.
- Medical Diagnosis: SVM helps classify diseases based on symptoms or medical images, such as classifying tumors as malignant or benign.
- Stock Market Prediction: SVM is used to predict stock market trends by classifying historical data into different categories.

[21]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

mean1 = 55
std_dev1 = 10
num_samples = 500

column1_numbers = np.random.normal(mean1, std_dev1, num_samples)
column1_numbers = np.clip(column1_numbers, 30, 120)
column1_numbers = np.round(column1_numbers).astype(int)

mean2 = 18
std_dev2 = 3

column2_numbers = np.random.normal(mean2, std_dev2, num_samples)
column2_numbers = np.clip(column2_numbers, 12, 26)
```

```
column2_numbers = np.round(column2_numbers).astype(int)

column3_numbers = np.random.randint(2, size=num_samples)
column3_numbers[column1_numbers > mean1] =1

data = {"Miles_Per_week": column1_numbers,
        "Farthest_run": column2_numbers,
        "Qualified_Boston_Marathon": column3_numbers}

df = pd.DataFrame(data)

df
```

[21]:
```
     Miles_Per_week  Farthest_run  Qualified_Boston_Marathon
0                53            16                          1
1                60            20                          1
2                44            12                          1
3                65            18                          1
4                70            21                          1
..              …             …                          …
495              58            17                          1
496              55            12                          0
497              43            18                          1
498              73            22                          1
499              47            19                          0

[500 rows x 3 columns]
```

[22]:
```
plt.figure(figsize=(10,6))
plt.scatter(df['Miles_Per_week'], df['Farthest_run'],␣
  ↪c=df['Qualified_Boston_Marathon'], cmap='jet_r')
plt.xlabel("Miles Per Week")
plt.ylabel("Farthest Run")
plt.title("Scatter Plot")
plt.colorbar(label="Qualified Boston Marathon")
plt.show()
```



```
[23]: X = df.iloc[:,0:2]
      y = df.iloc[:,2]

      X_train, X_test, y_train, y_test = train_test_split(X,y, random_state=33,␣
       ↪test_size=0.2)
      model = SVC()

      model.fit(X_train, y_train)
      print(model.score(X_test, y_test))
```

```
0.76
```

```
[24]: model_reg0 = SVC(C=0.1)
      model_reg0.fit(X_train, y_train)
      print(model_reg0.score(X_test, y_test))
```

```
0.76
```

```
[25]: model_reg1 = SVC(C=1)
      model_reg1.fit(X_train, y_train)
      print(model_reg1.score(X_test, y_test))
```

```
0.76
```

[26]:
```
model_reg2 = SVC(C=1000)
model_reg2.fit(X_train, y_train)
print(model_reg2.score(X_test, y_test))
```

```
0.73
```

1.0.1 Gamma

[27]:
```
model_gamma0 = SVC(gamma=0.1)
model_gamma0.fit(X_train, y_train)
model_gamma0.score(X_test, y_test)
```

[27]:
```
0.71
```

[28]:
```
model_gamma1 = SVC(gamma=1)
model_gamma1.fit(X_train, y_train)
model_gamma1.score(X_test, y_test)
```

[28]:
```
0.74
```

[29]:
```
model_gamma2 = SVC(gamma=1000)
model_gamma2.fit(X_train, y_train)
model_gamma2.score(X_test, y_test)
```

[29]:
```
0.74
```

1.0.2 Kernel

[30]:
```
model_kernel0 = SVC(kernel='linear')
model_kernel0.fit(X_train, y_train)
model_kernel0.score(X_test, y_test)
```

[30]:
```
0.74
```

[31]:
```
model_kernel0 = SVC(kernel='rbf')
model_kernel0.fit(X_train, y_train)
model_kernel0.score(X_test, y_test)
```

[31]:
```
0.76
```

11-Naive_Bayes

October 20, 2024

1 Naive Bayes Classifier

The **Naive Bayes classifier** is a family of probabilistic classifiers that rely on Bayes' Theorem and the assumption of feature independence. The classifier is called naive because it assumes that all the features in a dataset are independent of each other, which is rarely true in real-world scenarios, but it simplifies the computation.

Naive Bayes works by calculating the probabilities for each class and then predicting the class with the highest probability based on the input features. It is a powerful and simple algorithm for classification tasks, especially in domains like text classification (e.g., spam detection, sentiment analysis).

When to Use Naive Bayes Classifier? Naive Bayes is particularly useful in the following scenarios:

- **When working with high-dimensional datasets**: It performs well with many features, such as in text classification where each word can be a feature.
- **When the assumption of conditional independence holds**: Even when this assumption is not completely true, Naive Bayes often performs surprisingly well in practice.
- **For real-time predictions**: Naive Bayes is computationally efficient and can handle large datasets quickly, making it suitable for real-time applications.
- **Text classification, sentiment analysis, and spam filtering**: These tasks involve high-dimensional and sparse data, where Naive Bayes shines.
- **Categorical input features**: The algorithm handles categorical variables well, especially in its Multinomial Naive Bayes variant.

Who Should Use Naive Bayes Classifier? Naive Bayes is an excellent choice for:

- **Data scientists and machine learning practitioners** who need a fast, simple, and interpretable model for classification tasks.
- **Business analysts** looking for text-based solutions, such as customer sentiment analysis, email spam filtering, or document categorization.
- **Researchers and developers** working with large-scale text processing, like in Natural Language Processing (NLP).
- **Beginners in machine learning**, as Naive Bayes is easy to implement and understand, while still being effective in many real-world scenarios.

Advantages of Naive Bayes Classifier:

- **Fast and efficient**: Naive Bayes is computationally efficient and works well with large datasets.
- **Handles high-dimensional data**: Especially useful for text classification problems where the data is sparse and high-dimensional.
- **Simple to implement**: The algorithm is simple to code and interpret, making it a good starting point for beginners.
- **Requires less training data**: It performs well with relatively small datasets because it estimates fewer parameters.
- **Performs well in practice**: Even though the independence assumption rarely holds true, Naive Bayes often works surprisingly well.

1.0.1 Disadvantages of Naive Bayes Classifier:

- **Strong independence assumption**: The assumption that features are conditionally independent given the class can lead to suboptimal results if this condition is not met in the data.
- **Zero probability problem**: If a feature value that wasn't seen in the training data appears in the test set, the model will assign a zero probability to that class. This can be mitigated using techniques like Laplace smoothing.
- **Not suitable for continuous data without modifications**: The basic form of Naive Bayes does not handle continuous data well unless it is adapted (e.g., Gaussian Naive Bayes).
- **Not ideal for highly correlated features**: Naive Bayes struggles when features are correlated since the independence assumption is violated.

Real-World Applications of Naive Bayes Classifier:

- **Email Spam Detection**: Classifies emails as spam or not based on the occurrence of certain words.
- **Text Classification**: Categorizes documents or news articles into different topics based on word occurrences (Multinomial Naive Bayes).
- **Sentiment Analysis**: Determines whether a review or a tweet is positive or negative by analyzing word frequencies (commonly used in NLP).
- **Medical Diagnosis**: Predicts diseases based on symptoms by calculating the probability of each condition.
- **Recommender Systems**: Helps in filtering and recommending content based on a user's previous interactions and preferences.

[55]:
```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV

num_items = 41
data = {
    'City Population': np.random.randint(10000, 1000000, num_items),
    'Continent': np.random.choice(['Asia', 'Europe', 'North America', 'South␣
 ↪America'], num_items),
    'Venue Capacity': np.random.randint(500, 20000, num_items),
    'Day Of Week': np.random.choice(['Monday', 'Tuesday', 'Wednesday',␣
 ↪'Thursday', 'Friday', 'Saturday', 'Sunday'], num_items),
    'Multiple Concerts': np.random.randint(0, 2, num_items),
    'Sold Out': np.random.randint(0, 2, num_items)
}

df = pd.DataFrame(data)
#df = pd.read_csv('pear_jam_tour2.csv', encoding="unicode_escape")
df
```

```
[55]:     City Population      Continent  Venue Capacity Day Of Week  \
      0            491817  North America           16665    Thursday
      1            701437         Europe            9246      Monday
      2            166930         Europe            8403    Thursday
      3            588655  South America            4672    Saturday
      4            388841  South America            1405      Friday
      5            882813           Asia           18118      Sunday
      6            158157           Asia            1666   Wednesday
      7            117802           Asia           14382      Sunday
      8             44163  North America            2502    Saturday
      9            893911  North America           17897      Friday
      10           981604         Europe           15859     Tuesday
      11           780283  North America           19257   Wednesday
      12           265129           Asia            4524    Thursday
      13           131209           Asia           14857    Thursday
      14           405167         Europe            9712      Monday
      15            84701  North America            2482    Thursday
      16           822257  North America            5842      Sunday
      17           424221  South America           17013    Saturday
      18           305672           Asia            2628      Sunday
      19           720117  North America            3999    Thursday
      20           285205  North America            6290    Saturday
      21           121093  North America           10833    Thursday
      22           240183           Asia            8535    Saturday
      23           368908  North America           19451     Tuesday
      24           989683  North America            8082    Saturday
      25           743758  South America           10294      Monday
```

```
26            951668           Asia            15488      Friday
27            356715         Europe             1282   Wednesday
28            428998  South America            17548      Friday
29            490859         Europe             9096      Friday
30            815163           Asia            12759     Tuesday
31            568414  North America             3550      Friday
32            569051  South America             5245      Sunday
33            327147           Asia             4974   Wednesday
34            703689  North America            19689      Monday
35            368834         Europe            15419      Monday
36            259567  North America            16369     Tuesday
37            654837  North America            19461      Friday
38            560703  North America            19339    Thursday
39            466105  South America            12232     Tuesday
40            167668  South America             2890    Thursday

    Multiple Concerts  Sold Out
0                   0         0
1                   0         0
2                   0         0
3                   0         1
4                   1         1
5                   1         1
6                   1         0
7                   0         0
8                   0         0
9                   1         1
10                  1         1
11                  0         1
12                  0         1
13                  1         1
14                  0         0
15                  0         0
16                  0         1
17                  1         0
18                  0         1
19                  0         0
20                  1         0
21                  1         1
22                  1         1
23                  1         1
24                  0         0
25                  0         0
26                  0         1
27                  0         0
28                  0         0
29                  0         0
```

```
30                   1          1
31                   0          1
32                   1          1
33                   0          0
34                   0          0
35                   0          0
36                   0          0
37                   1          1
38                   1          0
39                   0          1
40                   0          1
```

[56]:
```
df2 = pd.get_dummies(df[['Continent', 'Day Of Week']])

df2
```

[56]:
```
    Continent_Asia  Continent_Europe  Continent_North America  \
0            False             False                     True
1            False              True                    False
2            False              True                    False
3            False             False                    False
4            False             False                    False
5             True             False                    False
6             True             False                    False
7             True             False                    False
8            False             False                     True
9            False             False                     True
10           False              True                    False
11           False             False                     True
12            True             False                    False
13            True             False                    False
14           False              True                    False
15           False             False                     True
16           False             False                     True
17           False             False                    False
18            True             False                    False
19           False             False                     True
20           False             False                     True
21           False             False                     True
22            True             False                    False
23           False             False                     True
24           False             False                     True
25           False             False                    False
26            True             False                    False
27           False              True                    False
28           False             False                    False
29           False              True                    False
```

```
30          True            False                      False
31         False            False                       True
32         False            False                      False
33          True            False                      False
34         False            False                       True
35         False             True                      False
36         False            False                       True
37         False            False                       True
38         False            False                       True
39         False            False                      False
40         False            False                      False

    Continent_South America  Day Of Week_Friday  Day Of Week_Monday  \
0                     False               False               False
1                     False               False                True
2                     False               False               False
3                      True               False               False
4                      True                True               False
5                     False               False               False
6                     False               False               False
7                     False               False               False
8                     False               False               False
9                     False                True               False
10                    False               False               False
11                    False               False               False
12                    False               False               False
13                    False               False               False
14                    False               False                True
15                    False               False               False
16                    False               False               False
17                     True               False               False
18                    False               False               False
19                    False               False               False
20                    False               False               False
21                    False               False               False
22                    False               False               False
23                    False               False               False
24                    False               False               False
25                     True               False                True
26                    False                True               False
27                    False               False               False
28                     True                True               False
29                    False                True               False
30                    False               False               False
31                    False                True               False
32                     True               False               False
33                    False               False               False
```

```
34                     False               False                True
35                     False               False                True
36                     False               False               False
37                     False                True               False
38                     False               False               False
39                      True               False               False
40                      True               False               False

    Day Of Week_Saturday  Day Of Week_Sunday  Day Of Week_Thursday  \
0                  False               False                  True
1                  False               False                 False
2                  False               False                  True
3                   True               False                 False
4                  False               False                 False
5                  False                True                 False
6                  False               False                 False
7                  False                True                 False
8                   True               False                 False
9                  False               False                 False
10                 False               False                 False
11                 False               False                 False
12                 False               False                  True
13                 False               False                  True
14                 False               False                 False
15                 False               False                  True
16                 False                True                 False
17                  True               False                 False
18                 False                True                 False
19                 False               False                  True
20                  True               False                 False
21                 False               False                  True
22                  True               False                 False
23                 False               False                 False
24                  True               False                 False
25                 False               False                 False
26                 False               False                 False
27                 False               False                 False
28                 False               False                 False
29                 False               False                 False
30                 False               False                 False
31                 False               False                 False
32                 False                True                 False
33                 False               False                 False
34                 False               False                 False
35                 False               False                 False
36                 False               False                 False
37                 False               False                 False
```

```
38                 False               False                 True
39                 False               False                False
40                 False               False                 True

    Day Of Week_Tuesday  Day Of Week_Wednesday
0                 False                  False
1                 False                  False
2                 False                  False
3                 False                  False
4                 False                  False
5                 False                  False
6                 False                   True
7                 False                  False
8                 False                  False
9                 False                  False
10                 True                  False
11                False                   True
12                False                  False
13                False                  False
14                False                  False
15                False                  False
16                False                  False
17                False                  False
18                False                  False
19                False                  False
20                False                  False
21                False                  False
22                False                  False
23                 True                  False
24                False                  False
25                False                  False
26                False                  False
27                False                   True
28                False                  False
29                False                  False
30                 True                  False
31                False                  False
32                False                  False
33                False                   True
34                False                  False
35                False                  False
36                 True                  False
37                False                  False
38                False                  False
39                 True                  False
40                False                  False
```

```
[57]: df3 = pd.concat([df, df2], axis=1)

      df3
```

```
[57]:     City Population      Continent  Venue Capacity Day Of Week  \
      0              491817  North America           16665    Thursday
      1              701437         Europe            9246      Monday
      2              166930         Europe            8403    Thursday
      3              588655  South America            4672    Saturday
      4              388841  South America            1405      Friday
      5              882813           Asia           18118      Sunday
      6              158157           Asia            1666   Wednesday
      7              117802           Asia           14382      Sunday
      8               44163  North America            2502    Saturday
      9              893911  North America           17897      Friday
      10             981604         Europe           15859     Tuesday
      11             780283  North America           19257   Wednesday
      12             265129           Asia            4524    Thursday
      13             131209           Asia           14857    Thursday
      14             405167         Europe            9712      Monday
      15              84701  North America            2482    Thursday
      16             822257  North America            5842      Sunday
      17             424221  South America           17013    Saturday
      18             305672           Asia            2628      Sunday
      19             720117  North America            3999    Thursday
      20             285205  North America            6290    Saturday
      21             121093  North America           10833    Thursday
      22             240183           Asia            8535    Saturday
      23             368908  North America           19451     Tuesday
      24             989683  North America            8082    Saturday
      25             743758  South America           10294      Monday
      26             951668           Asia           15488      Friday
      27             356715         Europe            1282   Wednesday
      28             428998  South America           17548      Friday
      29             490859         Europe            9096      Friday
      30             815163           Asia           12759     Tuesday
      31             568414  North America            3550      Friday
      32             569051  South America            5245      Sunday
      33             327147           Asia            4974   Wednesday
      34             703689  North America           19689      Monday
      35             368834         Europe           15419      Monday
      36             259567  North America           16369     Tuesday
      37             654837  North America           19461      Friday
      38             560703  North America           19339    Thursday
      39             466105  South America           12232     Tuesday
      40             167668  South America            2890    Thursday
```

```
   Multiple Concerts  Sold Out  Continent_Asia  Continent_Europe  \
0                  0         0           False             False
1                  0         0           False              True
2                  0         0           False              True
3                  0         1           False             False
4                  1         1           False             False
5                  1         1            True             False
6                  1         0            True             False
7                  0         0            True             False
8                  0         0           False             False
9                  1         1           False             False
10                 1         1           False              True
11                 0         1           False             False
12                 0         1            True             False
13                 1         1            True             False
14                 0         0           False              True
15                 0         0           False             False
16                 0         1           False             False
17                 1         0           False             False
18                 0         1            True             False
19                 0         0           False             False
20                 1         0           False             False
21                 1         1           False             False
22                 1         1            True             False
23                 1         1           False             False
24                 0         0           False             False
25                 0         0           False             False
26                 0         1            True             False
27                 0         0           False              True
28                 0         0           False             False
29                 0         0           False              True
30                 1         1            True             False
31                 0         1           False             False
32                 1         1           False             False
33                 0         0            True             False
34                 0         0           False             False
35                 0         0           False              True
36                 0         0           False             False
37                 1         1           False             False
38                 1         0           False             False
39                 0         1           False             False
40                 0         1           False             False

   Continent_North America  Continent_South America  Day Of Week_Friday  \
0                     True                    False               False
1                    False                    False               False
2                    False                    False               False
```

```
3                     False                    True                  False
4                     False                    True                   True
5                     False                   False                  False
6                     False                   False                  False
7                     False                   False                  False
8                      True                   False                  False
9                      True                   False                   True
10                    False                   False                  False
11                     True                   False                  False
12                    False                   False                  False
13                    False                   False                  False
14                    False                   False                  False
15                     True                   False                  False
16                     True                   False                  False
17                    False                    True                  False
18                    False                   False                  False
19                     True                   False                  False
20                     True                   False                  False
21                     True                   False                  False
22                    False                   False                  False
23                     True                   False                  False
24                     True                   False                  False
25                    False                    True                  False
26                    False                   False                   True
27                    False                   False                  False
28                    False                    True                   True
29                    False                   False                   True
30                    False                   False                  False
31                     True                   False                   True
32                    False                    True                  False
33                    False                   False                  False
34                     True                   False                  False
35                    False                   False                  False
36                     True                   False                  False
37                     True                   False                   True
38                     True                   False                  False
39                    False                    True                  False
40                    False                    True                  False

    Day Of Week_Monday  Day Of Week_Saturday  Day Of Week_Sunday  \
0                False                 False               False
1                 True                 False               False
2                False                 False               False
3                False                  True               False
4                False                 False               False
5                False                 False                True
6                False                 False               False
```

```
7                 False                 False                True
8                 False                  True               False
9                 False                 False               False
10                False                 False               False
11                False                 False               False
12                False                 False               False
13                False                 False               False
14                 True                 False               False
15                False                 False               False
16                False                 False                True
17                False                  True               False
18                False                 False                True
19                False                 False               False
20                False                  True               False
21                False                 False               False
22                False                  True               False
23                False                 False               False
24                False                  True               False
25                 True                 False               False
26                False                 False               False
27                False                 False               False
28                False                 False               False
29                False                 False               False
30                False                 False               False
31                False                 False               False
32                False                 False                True
33                False                 False               False
34                 True                 False               False
35                 True                 False               False
36                False                 False               False
37                False                 False               False
38                False                 False               False
39                False                 False               False
40                False                 False               False

    Day Of Week_Thursday  Day Of Week_Tuesday  Day Of Week_Wednesday
0                   True                False                  False
1                  False                False                  False
2                   True                False                  False
3                  False                False                  False
4                  False                False                  False
5                  False                False                  False
6                  False                False                   True
7                  False                False                  False
8                  False                False                  False
9                  False                False                  False
10                 False                 True                  False
```

```
11               False               False               True
12                True               False              False
13                True               False              False
14               False               False              False
15                True               False              False
16               False               False              False
17               False               False              False
18               False               False              False
19                True               False              False
20               False               False              False
21                True               False              False
22               False               False              False
23               False                True              False
24               False               False              False
25               False               False              False
26               False               False              False
27               False               False               True
28               False               False              False
29               False               False              False
30               False                True              False
31               False               False              False
32               False               False              False
33               False               False               True
34               False               False              False
35               False               False              False
36               False                True              False
37               False               False              False
38                True               False              False
39               False                True              False
40                True               False              False
```

[58]:
```
df4 = df3.drop(columns=['Continent', 'Day Of Week'], axis=1)
df4
```

[58]:
```
    City Population  Venue Capacity  Multiple Concerts  Sold Out  \
0            491817           16665                  0         0
1            701437            9246                  0         0
2            166930            8403                  0         0
3            588655            4672                  0         1
4            388841            1405                  1         1
5            882813           18118                  1         1
6            158157            1666                  1         0
7            117802           14382                  0         0
8             44163            2502                  0         0
9            893911           17897                  1         1
10           981604           15859                  1         1
11           780283           19257                  0         1
```

```
12          265129            4524                 0         1
13          131209           14857                 1         1
14          405167            9712                 0         0
15           84701            2482                 0         0
16          822257            5842                 0         1
17          424221           17013                 1         0
18          305672            2628                 0         1
19          720117            3999                 0         0
20          285205            6290                 1         0
21          121093           10833                 1         1
22          240183            8535                 1         1
23          368908           19451                 1         1
24          989683            8082                 0         0
25          743758           10294                 0         0
26          951668           15488                 0         1
27          356715            1282                 0         0
28          428998           17548                 0         0
29          490859            9096                 0         0
30          815163           12759                 1         1
31          568414            3550                 0         1
32          569051            5245                 1         1
33          327147            4974                 0         0
34          703689           19689                 0         0
35          368834           15419                 0         0
36          259567           16369                 0         0
37          654837           19461                 1         1
38          560703           19339                 1         0
39          466105           12232                 0         1
40          167668            2890                 0         1

    Continent_Asia  Continent_Europe  Continent_North America  \
0            False             False                     True
1            False              True                    False
2            False              True                    False
3            False             False                    False
4            False             False                    False
5             True             False                    False
6             True             False                    False
7             True             False                    False
8            False             False                     True
9            False             False                     True
10           False              True                    False
11           False             False                     True
12            True             False                    False
13            True             False                    False
14           False              True                    False
15           False             False                     True
```

```
16          False             False                     True
17          False             False                    False
18           True             False                    False
19          False             False                     True
20          False             False                     True
21          False             False                     True
22           True             False                    False
23          False             False                     True
24          False             False                     True
25          False             False                    False
26           True             False                    False
27          False              True                    False
28          False             False                    False
29          False              True                    False
30           True             False                    False
31          False             False                     True
32          False             False                    False
33           True             False                    False
34          False             False                     True
35          False              True                    False
36          False             False                     True
37          False             False                     True
38          False             False                     True
39          False             False                    False
40          False             False                    False

    Continent_South America  Day Of Week_Friday  Day Of Week_Monday  \
0                     False               False               False
1                     False               False                True
2                     False               False               False
3                      True               False               False
4                      True                True               False
5                     False               False               False
6                     False               False               False
7                     False               False               False
8                     False               False               False
9                     False                True               False
10                    False               False               False
11                    False               False               False
12                    False               False               False
13                    False               False               False
14                    False               False                True
15                    False               False               False
16                    False               False               False
17                     True               False               False
18                    False               False               False
19                    False               False               False
```

```
20                    False               False               False
21                    False               False               False
22                    False               False               False
23                    False               False               False
24                    False               False               False
25                     True               False                True
26                    False                True               False
27                    False               False               False
28                     True                True               False
29                    False                True               False
30                    False               False               False
31                    False                True               False
32                     True               False               False
33                    False               False               False
34                    False               False                True
35                    False               False                True
36                    False               False               False
37                    False                True               False
38                    False               False               False
39                     True               False               False
40                     True               False               False

    Day Of Week_Saturday  Day Of Week_Sunday  Day Of Week_Thursday  \
0                  False               False                  True
1                  False               False                 False
2                  False               False                  True
3                   True               False                 False
4                  False               False                 False
5                  False                True                 False
6                  False               False                 False
7                  False                True                 False
8                   True               False                 False
9                  False               False                 False
10                 False               False                 False
11                 False               False                 False
12                 False               False                  True
13                 False               False                  True
14                 False               False                 False
15                 False               False                  True
16                 False                True                 False
17                  True               False                 False
18                 False                True                 False
19                 False               False                  True
20                  True               False                 False
21                 False               False                  True
22                  True               False                 False
23                 False               False                 False
```

```
24                 True               False                 False
25                False               False                 False
26                False               False                 False
27                False               False                 False
28                False               False                 False
29                False               False                 False
30                False               False                 False
31                False               False                 False
32                False                True                 False
33                False               False                 False
34                False               False                 False
35                False               False                 False
36                False               False                 False
37                False               False                 False
38                False               False                  True
39                False               False                 False
40                False               False                  True

    Day Of Week_Tuesday  Day Of Week_Wednesday
0                 False                  False
1                 False                  False
2                 False                  False
3                 False                  False
4                 False                  False
5                 False                  False
6                 False                   True
7                 False                  False
8                 False                  False
9                 False                  False
10                 True                  False
11                False                   True
12                False                  False
13                False                  False
14                False                  False
15                False                  False
16                False                  False
17                False                  False
18                False                  False
19                False                  False
20                False                  False
21                False                  False
22                False                  False
23                 True                  False
24                False                  False
25                False                  False
26                False                  False
27                False                   True
```

```
28                  False                    False
29                  False                    False
30                   True                    False
31                  False                    False
32                  False                    False
33                  False                     True
34                  False                    False
35                  False                    False
36                   True                    False
37                  False                    False
38                  False                    False
39                   True                    False
40                  False                    False
```

[59]:
```
X = df4.drop(columns=['Sold Out'], axis=1)
y = df4['Sold Out']

X_train, X_test, y_train, y_test = train_test_split(X,y, random_state=33,␣
 ↪test_size=0.2)
gnb = GaussianNB()

gnb.fit(X_train, y_train)
y_pred = gnb.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.80      0.80      0.80         5
           1       0.75      0.75      0.75         4

    accuracy                           0.78         9
   macro avg       0.78      0.78      0.78         9
weighted avg       0.78      0.78      0.78         9
```

[60]:
```
gnb.score(X_train, y_train)
```

[60]:
```
0.5625
```

[61]:
```
gnb.score(X_test, y_test)
```

[61]:
```
0.7777777777777778
```

1.0.2 Add in parameter

```
[66]: param_grid = {
          'var_smoothing': [0.00000001, 0.000000001, 0.00000001]
      }

      grid_search = GridSearchCV(gnb,param_grid, cv=5, scoring='accuracy', n_jobs=-1)
      grid_search.fit(X_train, y_train)
```

```
[66]: GridSearchCV(cv=5, estimator=GaussianNB(), n_jobs=-1,
                   param_grid={'var_smoothing': [1e-08, 1e-09, 1e-08]},
                   scoring='accuracy')
```

```
[67]: grid_search.best_params_
```

```
[67]: {'var_smoothing': 1e-08}
```

```
[68]: grid_search.best_score_
```

```
[68]: 0.4095238095238095
```

12-Logistical_Regression

October 20, 2024

1 Logistic Regression

Logistic Regression is a supervised learning algorithm used for classification tasks, not regression, despite its name. It estimates the probability that a given input point belongs to a certain class by applying a logistic (sigmoid) function to a linear combination of input features. The output is a probability between 0 and 1, making it ideal for binary classification problems (two possible outcomes, like yes/no, spam/not spam).

Logistic regression predicts a binary outcome based on one or more predictor variables (features) and is one of the foundational techniques in machine learning and statistics.

When to Use Logistic Regression? Logistic Regression is commonly used when:

- **The dependent variable is binary**: For example, a classification problem where the output is either "1" or "0" (e.g., true/false, success/failure).
- **You need probabilistic interpretations**: Logistic regression outputs probabilities that a data point belongs to a certain class, which makes it useful in areas where you need both the prediction and the confidence of that prediction.
- **When interpretability is important**: It provides clear and interpretable coefficients, which can explain the impact of each feature on the prediction.
- **The relationship between features and the target is linear in the log-odds**: While logistic regression assumes a linear relationship between the features and the log-odds of the outcome, it can still handle non-linear relationships by transforming the input features (e.g., polynomial or interaction terms).

Typical use cases include:

- **Medical Diagnosis**: Predicting whether a patient has a disease or not.
- **Email Classification**: Determining whether an email is spam or not.
- **Customer Churn**: Predicting whether a customer will leave a service or remain.
- **Credit Scoring**: Estimating the probability of a customer defaulting on a loan.

Who Should Use Logistic Regression? Logistic Regression is ideal for:

- **Data scientists and machine learning practitioners** who need a simple, interpretable model for binary classification tasks.

- **Business analysts and researchers** looking for an explainable model where they can interpret the effect of each feature on the outcome.
- **Medical professionals** making decisions based on classification problems like disease detection.
- **Beginner machine learning students** since logistic regression is one of the simplest and easiest algorithms to learn and apply.

Advantages of Logistic Regression:

- **Simplicity and Interpretability**: It is easy to implement and interpret, making it a popular choice for binary classification problems.
- **Probabilistic Output**: Logistic regression provides probabilities for class membership, which can be useful for understanding the confidence of predictions.
- **Fast and Efficient**: It performs well on relatively small datasets and is computationally efficient.
- **No Feature Scaling Required**: While performance can be improved by scaling features, logistic regression does not require it.
- **Linear Decision Boundary**: The decision boundary is linear, which makes it suitable for linearly separable datasets.

Disadvantages of Logistic Regression:

- **Limited to Linear Boundaries**: Logistic regression assumes a linear relationship between the features and the log-odds, making it unsuitable for complex or highly non-linear problems.
- **Assumes Independence of Features**: Like Naive Bayes, logistic regression assumes that the features are independent of each other, which may not hold true in many cases. Cannot Handle Multiclass Problems Directly: Logistic regression is a binary classifier by default, but techniques like One-vs-Rest or Softmax Regression can extend it to multiclass problems.
- **Sensitive to Outliers**: Logistic regression can be sensitive to outliers, which can skew the decision boundary.

Real-World Applications of Logistic Regression:

1. **Credit Scoring**: Predicting the probability of a customer defaulting on a loan.
2. **Medical Diagnosis**: Predicting the likelihood of a disease based on symptoms.
3. **Customer Retention**: Estimating the probability that a customer will churn or stay with the company.
4. **Marketing Campaigns**: Predicting whether a customer will respond to an advertisement or not.
5. **Email Classification**: Classifying emails as spam or non-spam.

```
[1]: import pandas as pd

     d = {
             'miles_per_week':␣
       ↪[37,39,46,51,88,17,18,20,21,22,23,24,25,27,28,29,30,31,32,33,34,38,40,42,57,68,35,36,41,43,
             'completed_50m_ultra':␣
       ↪['no','no','no','no','no','no','no','no','no','no','no','no','no','no','no','no','no','no',
         }

     df = pd.DataFrame(data=d)
     df
```

```
[1]:      miles_per_week completed_50m_ultra
     0                37                  no
     1                39                  no
     2                46                  no
     3                51                  no
     4                88                  no
     ..              …                   …
     96               67                 yes
     97               74                 yes
     98               79                 yes
     99               90                 yes
     100             112                 yes

     [101 rows x 2 columns]
```

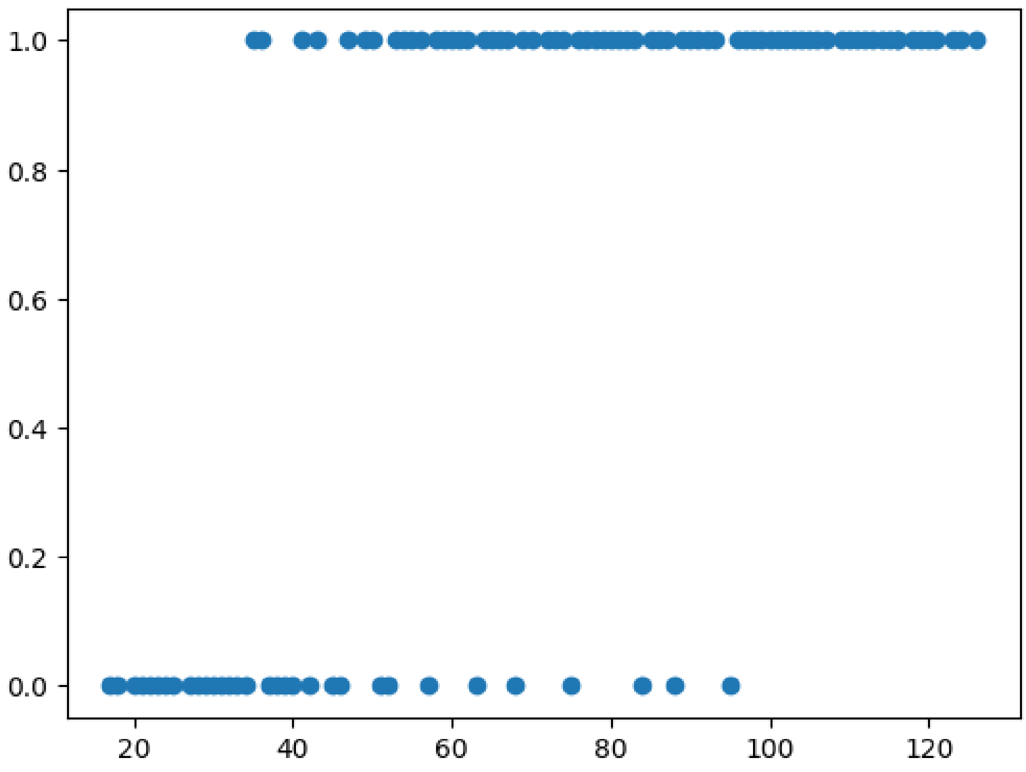
```
[2]: from sklearn.model_selection import train_test_split
     from sklearn.metrics import confusion_matrix, classification_report
     from sklearn.preprocessing import OrdinalEncoder
     from sklearn.linear_model import LogisticRegression
     from matplotlib import pyplot as plt

     import seaborn as sns

     finished_race = ['no', 'yes']
     enc = OrdinalEncoder(categories=[finished_race])
     df['completed_50m_ultra'] = enc.fit_transform(df[['completed_50m_ultra']])

     df
```

```
[2]:      miles_per_week  completed_50m_ultra
     0                37                  0.0
     1                39                  0.0
     2                46                  0.0
     3                51                  0.0
     4                88                  0.0
```

```
..                …                   …
96               67                 1.0
97               74                 1.0
98               79                 1.0
99               90                 1.0
100             112                 1.0

[101 rows x 2 columns]
```

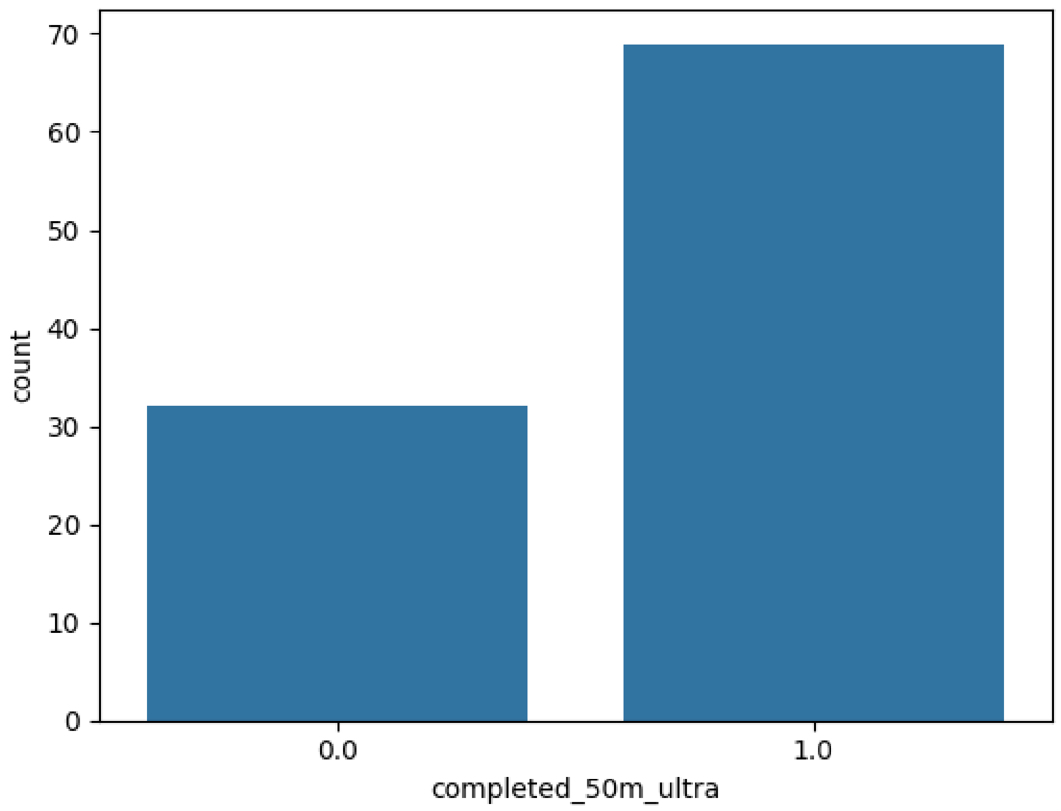
[3]:
```
plt.scatter(df['miles_per_week'], df['completed_50m_ultra'])
```

[3]:
```
<matplotlib.collections.PathCollection at 0x1f0ce21e410>
```



[4]:
```
sns.countplot(x='completed_50m_ultra', data=df)
```

[4]:
```
<Axes: xlabel='completed_50m_ultra', ylabel='count'>
```



[5]:
```
X = df.iloc[:,0:1]
y = df.iloc[:,1]

X_train, X_test, y_train, y_test = train_test_split(X,y, train_size=0.8,␣
↪random_state=11)

model = LogisticRegression()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
model.score(X_test, y_test)
```

[5]:
```
0.9047619047619048
```

[6]:
```
print(confusion_matrix(y_test, y_pred))
```

```
[[ 5  1]
 [ 1 14]]
```

[7]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support
```

```
         0.0       0.83      0.83      0.83         6
         1.0       0.93      0.93      0.93        15

    accuracy                           0.90        21
   macro avg       0.88      0.88      0.88        21
weighted avg       0.90      0.90      0.90        21
```

13-Machine_Learning_Pipeline

October 20, 2024

1 Machine Learning Pipeline

[96]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

from sklearn.tree import DecisionTreeClassifier

import joblib
```

[97]:
```
d1 = {
    'Social_media_followers' : [1000000, np.nan, 2000000, 1310000, 1700000, np.
 ↪nan, 4100000, 1600000, 2200000, 1000000],
    'Sold_out': [1,0,0,1,0,0,0,1,0,1]
}

d2 = {
        'Genre':['Rock', 'Metal', 'Bluegrass', 'Rock', np.nan, 'Rock', 'Rock',␣
 ↪np.nan, 'Bluegrass', 'Rock'],
        'Social_media_followers':[1000000, np.nan, 2000000, 1310000, 1700000,␣
 ↪np.nan, 4100000, 1600000, 2200000, 1000000],
        'Sold_out':[1,0,0,1,0,0,0,1,0,1]
    }

df1 = pd.DataFrame(d1)
df1
```

[97]:
```
   Social_media_followers  Sold_out
0               1000000.0         1
1                     NaN         0
2               2000000.0         0
3               1310000.0         1
```

```
4          1700000.0        0
5                NaN        0
6          4100000.0        0
7          1600000.0        1
8          2200000.0        0
9          1000000.0        1
```

[98]:
```
X1 = df1[['Social_media_followers']]
y1 = df1[['Sold_out']]

X1_train, X1_test, y1_train, y1_test = train_test_split(X1, y1, test_size=0.3,␣
 ↪random_state=19)

imputer = SimpleImputer(strategy='mean')
lr = LogisticRegression()

pipe1 = make_pipeline(imputer, lr)

pipe1.fit(X1_train, y1_train)
```

```
c:\Users\ikiga\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\utils\validation.py:1229: DataConversionWarning: A column-
vector y was passed when a 1d array was expected. Please change the shape of y
to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

[98]:
```
Pipeline(steps=[('simpleimputer', SimpleImputer()),
                ('logisticregression', LogisticRegression())])
```

[99]:
```
pipe1.score(X1_train, y1_train)
```

[99]:
```
1.0
```

[100]:
```
pipe1.score(X1_test, y1_test)
```

[100]:
```
0.6666666666666666
```

[101]:
```
pipe1.named_steps.simpleimputer.statistics_
```

[101]:
```
array([2051666.66666667])
```

[102]:
```
pipe1.named_steps.logisticregression.coef_
```

[102]:
```
array([[-9.72872687e-05]])
```

1.0.1 More Advance Pipeline

[103]:
```
df = pd.DataFrame(data=d2)
df
```

[103]:
```
      Genre  Social_media_followers  Sold_out
0      Rock               1000000.0         1
1     Metal                     NaN         0
2  Bluegrass              2000000.0         0
3      Rock               1310000.0         1
4       NaN               1700000.0         0
5      Rock                     NaN         0
6      Rock               4100000.0         0
7       NaN               1600000.0         1
8  Bluegrass              2200000.0         0
9      Rock               1000000.0         1
```

[104]:
```
X = df.iloc[:,0:2]
y = df.iloc[:,2]

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3,␣
 ↪random_state=17)

num_cols = ["Social_media_followers"]
cat_cols = ['Genre']

num_pipeline = Pipeline(
    steps = [
        ('impute', SimpleImputer(strategy='mean')),
        ('scale', StandardScaler())
    ]
)

cat_pipeline = Pipeline(steps=[
    ('impute', SimpleImputer(strategy='most_frequent')),
    ('one-hot-encoder', OneHotEncoder(handle_unknown='ignore',␣
 ↪sparse_output=False))
])
cat_pipeline
```

[104]:
```
Pipeline(steps=[('impute', SimpleImputer(strategy='most_frequent')),
                ('one-hot-encoder',
                 OneHotEncoder(handle_unknown='ignore', sparse_output=False))])
```

[105]:
```
col_transformer = ColumnTransformer(transformers= [
    ('num_pipeline', num_pipeline, num_cols),
    ('cat_pipeline', cat_pipeline, cat_cols),
],
remainder='drop', n_jobs=-1
)
```

[106]:
```
dtc = DecisionTreeClassifier()
pipefinal = make_pipeline(col_transformer, dtc)
pipefinal.fit(X_train, y_train)
```

[106]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(n_jobs=-1,
                                   transformers=[('num_pipeline',
                                                  Pipeline(steps=[('impute',
SimpleImputer()),
                                                                  ('scale',
StandardScaler())]),
                                                  ['Social_media_followers']),
                                                 ('cat_pipeline',
                                                  Pipeline(steps=[('impute',
SimpleImputer(strategy='most_frequent')),
                                                                  ('one-hot-
encoder',
OneHotEncoder(handle_unknown='ignore',
 sparse_output=False))]),
                                                  ['Genre'])])),
                ('decisiontreeclassifier', DecisionTreeClassifier())])
```

[107]:
```
pipefinal.score(X_test, y_test)
```

[107]:
```
0.6666666666666666
```

1.1 How to save your pipeline

[108]:
```
joblib.dump(pipefinal, 'pipe.joblib')
```

[108]:
```
['pipe.joblib']
```

[109]:
```
pipefinal2 = joblib.load('pipe.joblib')
pipefinal2
```

[109]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(n_jobs=-1,
                                   transformers=[('num_pipeline',
                                                  Pipeline(steps=[('impute',
SimpleImputer()),
                                                                  ('scale',
StandardScaler())]),
                                                  ['Social_media_followers']),
                                                 ('cat_pipeline',
                                                Pipeline(steps=[('impute',
SimpleImputer(strategy='most_frequent')),
                                                                ('one-hot-
encoder',
OneHotEncoder(handle_unknown='ignore',
 sparse_output=False))]),
                                                ['Genre'])])),
                ('decisiontreeclassifier', DecisionTreeClassifier())])
```

14-Cross_Validation

October 20, 2024

1 Cross Validation

Cross-Validation is a technique used in machine learning to assess the performance of a model by testing it on different subsets of the data. Instead of using just a single training and testing split, cross-validation helps evaluate the model's ability to generalize to unseen data by dividing the dataset into multiple parts (or folds) and training/testing the model on each part.

The most commonly used form of cross-validation is k-fold cross-validation, where the dataset is divided into k subsets or "folds." The model is trained on k−1 folds and tested on the remaining fold. This process is repeated k times, each time using a different fold as the test set and the rest as the training set. The final performance is averaged over all folds.

Cross-validation is important because it gives a better estimate of the model's performance by reducing the risk of overfitting or underfitting the data.

When to Use Cross-Validation? Cross-Validation should be used when:

- **You want to evaluate model performance**: It's used to estimate how well a model generalizes to an independent dataset.
- **You're working with limited data**: Cross-validation helps make the most out of the available data by using all of it for both training and testing across different iterations.
- **You're tuning hyperparameters**: It's often employed when trying to find the optimal hyperparameters of a model (e.g., in grid search).
- **To prevent overfitting**: When you want to avoid overfitting on a particular training set by providing a more robust evaluation of model performance.
- **In model selection**: It is used to compare different models and choose the best-performing one based on a more reliable performance metric.

Use cases include:

- **Machine learning competitions**: Such as Kaggle competitions, where high generalization performance is crucial.
- **Scientific research**: Where precise model evaluation is necessary to make reliable predictions.
- **Production-ready systems**: To validate that the model will work well in the real world, where unseen data is used.

1.0.1 Common Types of Cross-Validation:

1. k-Fold Cross-Validation:

- The most commonly used form, as described above. The number of folds, k, is usually 5 or 10.
- Advantages:
  - Provides a better estimation of model performance.
  - Each instance is used for both training and validation.
- Disadvantages:
  - More computationally expensive than simple train-test split, as the model is trained and evaluated k times.

2. Leave-One-Out Cross-Validation (LOOCV):

- A special case of k-fold where k equals the number of data points. That means for each iteration, only one instance is left out for testing, and the rest is used for training.
- Advantages:
  - No data point is wasted.
  - Provides an unbiased estimate of the generalization error.
- Disadvantages:
  - Computationally expensive for large datasets.
  - High variance, as training on almost the entire dataset may lead to overfitting.

3. Stratified k-Fold Cross-Validation:

- A variation of k-fold cross-validation where the data is split such that the proportion of classes in each fold is the same as the original dataset. This is especially useful in cases of imbalanced datasets.
- Advantages:
  - Better representation of the minority and majority classes in each fold.
  - Reduces bias in datasets with imbalanced class distributions.
- Disadvantages:
  - Still computationally intensive, similar to k-fold cross-validation.

4. Time Series Cross-Validation (Rolling or Expanding Window):

- In time series data, data points are not independent, so traditional k-fold cross-validation may not be suitable. Instead, the training and test splits are done based on time order.
- Advantages:
  - Maintains the temporal order of data, which is important for time series problems.

- Disadvantages:
  - More complex to implement and evaluate.

1.0.2 Who Should Use Cross-Validation?

Cross-validation is suitable for:

- **Data scientists and machine learning practitioners** looking for a robust model evaluation technique.
- **Machine learning beginners** who want to prevent overfitting or underfitting and need to compare different models and tune hyperparameters.
- **Researchers and engineers** working with limited data, where splitting the data into a simple train-test set might not be enough.
- **Kaggle competitors** or those involved in machine learning competitions where cross-validation provides a better estimate of performance on unseen data.

Advantages of Cross-Validation:

- **More reliable evaluation**: By using all the data for training and testing in different iterations, cross-validation gives a better estimate of model performance.
- **Reduces overfitting risk**: Prevents the model from overfitting to a particular train-test split.
- **Maximizes data usage**: Every data point is used for both training and validation, ensuring the model is evaluated on the entire dataset.
- **Model comparison**: Helps compare the performance of different models using a fair and unbiased technique.
- **Hyperparameter tuning**: Aids in finding the best hyperparameters for a model by testing performance across multiple folds.

Disadvantages of Cross-Validation:

- **Computationally expensive**: Training and testing the model multiple times (depending on k) can be time-consuming, especially for large datasets or complex models.
- **Bias-variance tradeoff**: While k-fold cross-validation provides a more reliable estimate, it can still have some variance if k is small. LOOCV reduces bias but may increase variance.

Real-World Applications of Cross-Validation:

- **Model Selection**: Comparing several machine learning algorithms (e.g., decision trees, logistic regression, SVM) and selecting the one with the best cross-validated performance.
- **Hyperparameter Tuning**: Tuning parameters like the depth of a decision tree, the learning rate of a neural network, or the C parameter in an SVM using cross-validation as part of a grid search or randomized search.
- **Time Series Prediction**: Using time-based cross-validation (rolling windows) to validate models for stock prices, weather forecasting, and other time-dependent data.

- **Medical Research**: Evaluating predictive models that classify diseases using limited patient dat

1.1 Example

[26]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import KFold, cross_val_score, StratifiedKFold
from sklearn.linear_model import LogisticRegression

# Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Initialize the model
model = LogisticRegression(max_iter=200)

# Define k-fold cross-validation with 5 folds
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Perform cross-validation and get scores
cv_scores = cross_val_score(model, X, y, cv=kf)

# Print cross-validated accuracy scores
print(f"Cross-validated scores: {cv_scores}")
print(f"Mean score: {cv_scores.mean()}")
```

```
Cross-validated scores: [1.         1.         0.93333333 0.96666667 0.96666667]
Mean score: 0.9733333333333334
```

[39]:
```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

np.random.seed(42)
```

[28]:
```
fastball_speed = np.random.randint(90,106, size=500)
tommy_john = np.where(fastball_speed > 96, np.random.choice([0,1], size=500,␣
 ↪p=[0.3, 0.7]), 0)

d = {
    'fastball_speed': fastball_speed,
    'tommy_john': tommy_john
}
```

```
df = pd.DataFrame(d)
df
```

```
[28]:      fastball_speed  tommy_john
      0                96           0
      1                93           0
      2               102           1
      3               104           1
      4               100           0
      ..              …          …
      495             104           1
      496              92           0
      497             101           1
      498              90           0
      499              93           0

      [500 rows x 2 columns]
```

```
[29]: X = df[['fastball_speed']]
      y = df[['tommy_john']]

      X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2,␣
       ↪random_state=11)
```

```
[30]: lr = LogisticRegression()

      lr.fit(X_train, y_train.values.ravel())
```

```
[30]: LogisticRegression()
```

```
[31]: lr.score(X_test, y_test)
```

```
[31]: 0.71
```

1.2 Test again

```
[32]: X2_train, X2_test, y2_train, y2_test = train_test_split(X,y, test_size=0.2,␣
       ↪random_state=25)

      lr = LogisticRegression()
      lr.fit(X2_train, y2_train.values.ravel())
```

```
[32]: LogisticRegression()
```

```
[33]: lr.score(X2_test, y2_test)
```

```
[33]: 0.8
```

[34]:
```
cvscore = cross_val_score(lr, X, y.values.ravel(), cv=10)
cvscore
```

[34]:
```
array([0.68, 0.78, 0.72, 0.72, 0.78, 0.82, 0.74, 0.74, 0.76, 0.82])
```

[35]:
```
print(np.average(cvscore))
print(np.std(cvscore))
```

```
0.756
0.04270831300812523
```

[36]:
```
kf = KFold(n_splits=15, shuffle=True, random_state=42)

kfscore = cross_val_score(lr, X, y.values.ravel(), cv=kf, scoring='f1')
kfscore2 = cross_val_score(lr, X, y.values.ravel(), cv=kf, scoring='accuracy')

print(kfscore)
print(np.average(kfscore))
print(np.std(kfscore))

print(kfscore2)
print(np.average(kfscore2))
print(np.std(kfscore2))
```

```
[0.66666667 0.82758621 0.72727273 0.7027027  0.6875     0.72727273
 0.74074074 0.72       0.66666667 0.60869565 0.66666667 0.63157895
 0.64516129 0.66666667 0.6       ]
0.6856785107611354
0.05620337068328653
[0.76470588 0.85294118 0.82352941 0.67647059 0.70588235 0.81818182
 0.78787879 0.78787879 0.78787879 0.72727273 0.72727273 0.78787879
 0.66666667 0.78787879 0.63636364]
0.7559120617944146
0.060970963505883366
```

[37]:
```
kf3 = StratifiedKFold(n_splits=10, shuffle=True, random_state=11)

kfscore3 = cross_val_score(lr, X, y.values.ravel(), cv=kf3)

print(kfscore3)
print(np.average(kfscore3))
print(np.std(kfscore3))
```

```
[0.66 0.76 0.76 0.8  0.8  0.74 0.76 0.74 0.78 0.76]
0.756
0.03773592452822642
```

1.3 Pipeline Sample

[40]:
```
scales = StandardScaler()
pipe1 = make_pipeline(scales, lr)
pipe1.fit(X_train, y_train.values.ravel())
```

[40]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('logisticregression', LogisticRegression())])
```

[42]:
```
scorespipe = cross_val_score(pipe1, X, y.values.ravel(), cv=10)

print(scorespipe)
print(np.average(scorespipe))
print(np.std(scorespipe))
```

```
[0.68 0.78 0.72 0.72 0.78 0.82 0.74 0.74 0.76 0.82]
0.756
0.04270831300812523
```

15-Hyperparameter_Tuning

October 20, 2024

1 Hyperparameter Tuning

Hyperparameter tuning is the process of finding the optimal values for the hyperparameters of a machine learning model. Unlike model parameters, which are learned from the training data (e.g., weights in a neural network, coefficients in a regression model), hyperparameters are set before the learning process and control how the model is trained or the structure of the model.

Examples of hyperparameters include:

- Learning rate in gradient-based algorithms.
- Number of trees in a Random Forest.
- Depth of a decision tree.
- C (regularization) and kernel in Support Vector Machines.
- k in k-nearest neighbors (KNN).
- Batch size or number of layers in neural networks.

Tuning these hyperparameters is crucial because they can have a significant impact on the model's performance. The goal is to find the best combination of hyperparameter values that yields the highest-performing model.

When to Use Hyperparameter Tuning? Hyperparameter tuning is used:

- **When building any machine learning model**: Most machine learning algorithms have hyperparameters that need to be fine-tuned.
- **When optimizing model performance**: After selecting a base model, hyperparameter tuning can help improve its accuracy, precision, recall, or other performance metrics.
- **To avoid overfitting/underfitting**: The right hyperparameters can prevent a model from overfitting (too complex) or underfitting (too simple) the data.
- **During model validation**: When performing cross-validation, tuning helps adjust the model for better generalization to unseen data.

Hyperparameter tuning is commonly used when:

- Building models for production systems where performance is critical.
- Participating in machine learning competitions.

- Deploying models with specific performance constraints, such as low latency in real-time systems.

1.0.1 How Does Hyperparameter Tuning Work?

The tuning process involves searching through a defined set of hyperparameter values and evaluating the model's performance for each combination of values. There are several methods to tune hyperparameters:

1. Grid Search:

A brute-force approach that evaluates all possible combinations of hyperparameters from a specified grid. You define a grid (list of values) for each hyperparameter, and the algorithm tries all possible combinations. Example: If you have two hyperparameters with 3 values each, Grid Search will evaluate all $3 \times 3 = 9$ combinations.

Pros:

- Exhaustive: Tests every combination.
- Simple to implement and understand.

Cons:

- Computationally expensive for large datasets or a large number of hyperparameters.
- May test irrelevant combinations of hyperparameters, leading to inefficiency..

2. Random Search:

- Instead of evaluating all combinations, Random Search randomly samples combinations of hyperparameters.
- You define a range of values for each hyperparameter, and the algorithm randomly selects combinations for evaluation.

Pros:

- More efficient than Grid Search as it focuses on randomly chosen combinations.
- Can still find good hyperparameter values without exhaustive search.

Cons:

- May miss the best combination if it isn't sampled.
- Still computationally expensive if many iterations are performed.

3. Bayesian Optimization:

- Bayesian optimization models the hyperparameter search as a probability problem and selects hyperparameter values based on the likelihood of improving the model's performance.
- This method focuses on exploring the hyperparameter space intelligently rather than randomly, making it more efficient than Grid or Random Search.

Pros:

- More efficient, as it focuses on the most promising hyperparameter areas.

- Can achieve high performance with fewer evaluations.

Cons:

- More complex to implement and requires specialized libraries (e.g., Hyperopt, Optuna).

4. Gradient-Based Optimization:

Similar to gradient descent used in model training, gradient-based methods adjust hyperparameters based on how changes impact performance.

Pros:

- Can quickly converge to optimal hyperparameters.

Cons:

- Limited to continuous hyperparameters (can't handle categorical ones like tree depth or number of estimators).

5. Manual Search:

A trial-and-error approach where hyperparameters are manually tuned based on intuition and experience.

Pros:

- Simple and effective when you have experience with the model or the dataset.

Cons:

- May miss optimal settings due to lack of systematic exploration.
- Time-consuming if tried exhaustively.

1.0.2 Who Should Use Hyperparameter Tuning?

Hyperparameter tuning is essential for:

- Machine learning practitioners and data scientists who want to maximize model performance.
- Researchers testing different hypotheses or models and trying to extract the best possible results from their experiments.
- Beginners who are learning machine learning and want to experience the difference in model behavior with tuned hyperparameters.
- Competitors in machine learning contests, such as Kaggle, where the smallest improvement can make a difference in ranking.

Advantages of Hyperparameter Tuning:

- **Improved performance**: Well-tuned hyperparameters can significantly improve model accuracy, precision, recall, and other metrics.
- **Prevention of overfitting or underfitting**: Tuning can help find the sweet spot between a model that's too complex and one that's too simple.

- **Customization**: Allows for deep customization of machine learning models to suit the specific characteristics of your dataset.

Disadvantages of Hyperparameter Tuning:

- **Computational expense**: For large datasets or complex models, tuning (especially Grid Search) can be time-consuming and resource-intensive.
- **Complexity**: Some tuning methods, such as Bayesian optimization, can be hard to implement and require specialized knowledge.
- **Local minima**: Certain tuning strategies, like gradient-based methods, may get stuck in local minima and miss the global optimum.

1.0.3 Real-World Applications of Hyperparameter Tuning:

- **Deep Learning**: Tuning hyperparameters like learning rate, batch size, and the number of layers to improve model performance on image, text, or speech data.
- **Kaggle Competitions**: Competitors frequently use hyperparameter tuning to optimize their models for the best performance on the leaderboard.
- **Predictive Modeling**: In fields like finance and healthcare, tuning hyperparameters to build accurate predictive models for stock prices, patient outcomes, etc.
- **Natural Language Processing (NLP)**: Tuning hyperparameters for algorithms like transformers or recurrent neural networks for better text classification, translation, or sentiment analysis.

[30]:
```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split, GridSearchCV,␣
 ↪RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
```

[31]:
```
mean1 = 55
std_dev1 = 10
num_samples = 500

column1_numbers = np.random.normal(mean1, std_dev1, num_samples)
column1_numbers = np.clip(column1_numbers, 30, 120)
column1_numbers = np.round(column1_numbers).astype(int)

mean2 =18
std_dev2 = 3

column2_numbers = np.random.normal(mean2, std_dev2, num_samples)
column2_numbers = np.clip(column2_numbers, 30, 120)
column2_numbers = np.round(column2_numbers).astype(int)
```

```
column3_numbers = np.random.randint(2, size=num_samples)
column3_numbers[column1_numbers > mean1] = 1

data = {'Miles_Per_Week': column1_numbers, 'Farthest_run': column2_numbers,␣
↪'Qualified_Boston_Marathon': column3_numbers}
df = pd.DataFrame(data)
df
```

[31]:
```
     Miles_Per_Week  Farthest_run  Qualified_Boston_Marathon
0                55            30                          0
1                59            30                          1
2                72            30                          1
3                72            30                          1
4                30            30                          1
..              …             …                         …
495              51            30                          0
496              51            30                          1
497              76            30                          1
498              47            30                          1
499              61            30                          1

[500 rows x 3 columns]
```

[32]:
```
X = df.iloc[:, 0:2]
y = df.iloc[:, 2]

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3,␣
↪random_state=26)

rf = RandomForestClassifier()
param_grid = [{
    'n_estimators': [500, 1000, 1500],
    'criterion': ['entropy', 'gini'],
    'min_samples_split': [5,10,15],
    'min_samples_leaf': [1,2,4],
    'max_depth': [10,20,30]
}]

grid_search = GridSearchCV(rf, param_grid, cv=2, scoring='accuracy', n_jobs=-1)

grid_search.fit(X_train, y_train)
```

[32]:
```
GridSearchCV(cv=2, estimator=RandomForestClassifier(), n_jobs=-1,
             param_grid=[{'criterion': ['entropy', 'gini'],
                          'max_depth': [10, 20, 30],
                          'min_samples_leaf': [1, 2, 4],
                          'min_samples_split': [5, 10, 15],
                          'n_estimators': [500, 1000, 1500]}],
             scoring='accuracy')
```

[33]:
```
grid_search.best_score_
```

[33]:
```
0.7885714285714286
```

[34]:
```
grid_search.best_params_
```

[34]:
```
{'criterion': 'gini',
 'max_depth': 20,
 'min_samples_leaf': 4,
 'min_samples_split': 15,
 'n_estimators': 1500}
```

[35]:
```
random_param_grid = [{
    'n_estimators': [500, 1000, 1500],
    'criterion': ['entropy', 'gini'],
    'min_samples_split': [5,10,15],
    'min_samples_leaf': [1,2,4],
    'max_depth': [10,20,30]
}]

random_grid_search = RandomizedSearchCV(rf, random_param_grid, cv=5,␣
 ↪scoring='accuracy', n_jobs=-1, random_state=26)
random_grid_search.fit(X_train, y_train)
```

[35]:
```
RandomizedSearchCV(cv=5, estimator=RandomForestClassifier(), n_jobs=-1,
                   param_distributions=[{'criterion': ['entropy', 'gini'],
                                         'max_depth': [10, 20, 30],
                                         'min_samples_leaf': [1, 2, 4],
                                         'min_samples_split': [5, 10, 15],
                                         'n_estimators': [500, 1000, 1500]}],
                   random_state=26, scoring='accuracy')
```

[36]:
```
random_grid_search.best_score_
```

[36]:
```
0.7457142857142858
```

[37]:
```
random_grid_search.best_params_
```

[37]:
```
{'n_estimators': 1000,
 'min_samples_split': 10,
 'min_samples_leaf': 1,
 'max_depth': 20,
 'criterion': 'entropy'}
```

16-PCA_Analysis

October 20, 2024

1 PCA Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique that is used to reduce the number of variables (features) in a dataset while retaining as much of the important information (variance) as possible. PCA does this by identifying directions, called principal components, along which the variance in the data is maximized.

PCA can be thought of as a way to transform a large set of possibly correlated variables into a smaller set of uncorrelated variables (the principal components). The first principal component captures the most variance in the data, the second captures the second most, and so on.

In simpler terms:

- Dimensionality reduction: PCA reduces the number of features by creating new ones (principal components) that explain most of the variability.
- Data simplification: By transforming data into fewer dimensions, PCA simplifies the data while keeping the most critical information.

1.0.1 When to Use PCA?

PCA is typically used in scenarios where:

- **The data has many features**: When dealing with high-dimensional data, PCA helps reduce complexity by creating a smaller set of meaningful features.
- **There is multicollinearity**: PCA is useful when features in the dataset are highly correlated. The principal components are uncorrelated, which helps in creating a better predictive model.
- **You need visualization of high-dimensional data**: PCA is commonly used for visualizing data in 2D or 3D, even if the original data has more dimensions.
- **To avoid overfitting**: By reducing the number of dimensions, PCA helps prevent overfitting in machine learning models.
- **Preprocessing**: Often used as a data preprocessing step before applying machine learning models like clustering (e.g., K-means), classification (e.g., logistic regression), or regression models.

PCA is used in applications like:

- **Image compression**: Reducing the number of pixels (features) while preserving image quality.

- **Genetics**: Analyzing large datasets of gene expression data.
- **Financial markets**: Reducing the number of stock price indicators while retaining information about market movement.
- **Natural language processing**: Reducing dimensionality of word embeddings or text data before applying machine learning.

1.0.2 How Does PCA Work?

PCA works through a series of mathematical steps that transform the data into a new set of coordinates, called principal components. These components are ordered such that the first one accounts for the most variance, the second for the next largest variance, and so on.

Here's how PCA works step by step:

1. **Standardize the Data:**
    - PCA works best when data is standardized. This is because PCA is affected by the scale of the variables. Standardization transforms the features so that they have a mean of 0 and a standard deviation of 1.
2. **Compute the Covariance Matrix:**
    - The covariance matrix shows how much the features vary from the mean with respect to each other. This helps identify patterns of correlation in the data.
    - The covariance matrix is an m × m matrix (where m is the number of features), with entries representing the covariance between each pair of features.
3. **Compute the Eigenvalues and Eigenvectors:**
    - Eigenvalues represent the amount of variance explained by each principal component, while eigenvectors represent the direction of these components.
    - Eigenvalues are used to rank the principal components in order of importance (from the largest to the smallest eigenvalue).
4. **Form the Principal Components:**
    - The principal components are linear combinations of the original features. Each component is a vector in the new feature space. The first component explains the largest amount of variance, the second explains the next largest, and so on.
5. **Select the Number of Principal Components (k):**
    - Usually, you select the number of principal components k that explain a certain threshold of variance (e.g., 90%, 95%). This helps to retain the most important information while reducing the number of dimensions.
6. **Project the Data:**
    - The original data is projected onto the new k-dimensional space (spanned by the selected principal components), creating a reduced-dimensional representation.

Advantages of PCA:

- **Dimensionality reduction**: Significantly reduces the number of features while retaining most of the variance, making the dataset simpler and more manageable.
- **Uncorrelated features**: The principal components are linearly uncorrelated, solving multicollinearity issues in the original features.
- **Data visualization**: PCA helps in visualizing high-dimensional data in two or three dimensions.
- **Speeds up model training**: Reducing the number of features can make machine learning models faster and less prone to overfitting.

Disadvantages of PCA:

- **Loss of interpretability**: The principal components are linear combinations of the original features, which makes them hard to interpret.
- **Sensitive to scaling**: PCA requires features to be on the same scale, so data standardization is often necessary.
- **Linear transformations only**: PCA only captures linear relationships, so it may not work well for datasets with complex, nonlinear structures.
- **Variance-focused**: PCA maximizes variance, but it does not directly optimize for predictive power, so it may not always yield the best features for prediction.

1.0.3 Real-World Applications of PCA:

- **Image Compression**: PCA can reduce the dimensionality of image data by retaining the most important pixel values, enabling image compression without significant loss of quality.
- **Genomics**: PCA is widely used in genomics to reduce the number of gene expression measurements while maintaining the most important patterns of variation across samples.
- **Finance**: In stock market analysis, PCA is used to reduce the number of correlated variables (e.g., stock prices) into principal components that explain overall market movements.
- **Natural Language Processing (NLP)**: PCA is often used to reduce the dimensionality of word vectors (embeddings) in text analysis.

[1]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

df = pd.read_csv('2022mlbteams.csv')
df
```

[1]:
```
                        Tm  #Bat  BatAge   R/G    G    PA    AB    R     H  \
0    Arizona Diamondbacks    57    26.5  4.33  162  6027  5351  702  1232
```

```
1          Atlanta Braves    53    27.5  4.87  162  6082  5509  789  1394
2       Baltimore Orioles    58    27.0  4.16  162  6049  5429  674  1281
3          Boston Red Sox    54    28.8  4.54  162  6144  5539  735  1427
4            Chicago Cubs    64    27.9  4.06  162  6072  5425  657  1293
5       Chicago White Sox    44    29.3  4.23  162  6123  5611  686  1435
6         Cincinnati Reds    66    29.4  4.00  162  5978  5380  648  1264
7     Cleveland Guardians    50    25.9  4.31  162  6163  5558  698  1410
8        Colorado Rockies    43    29.1  4.31  162  6105  5540  698  1408
9          Detroit Tigers    53    27.9  3.44  162  5870  5378  557  1240
10         Houston Astros    45    29.3  4.55  162  6054  5409  737  1341
11     Kansas City Royals    55    27.1  3.95  162  6010  5437  640  1327
12     Los Angeles Angels    66    27.9  3.85  162  5977  5423  623  1265
13    Los Angeles Dodgers    51    29.6  5.23  162  6247  5526  847  1418
14          Miami Marlins    56    28.9  3.62  162  5949  5395  586  1241
15      Milwaukee Brewers    51    29.1  4.48  162  6122  5417  725  1271
16        Minnesota Twins    61    26.9  4.30  162  6113  5476  696  1356
17          New York Mets    61    29.7  4.77  162  6176  5489  772  1422
18       New York Yankees    54    30.2  4.98  162  6172  5422  807  1308
19      Oakland Athletics    64    28.3  3.51  162  5863  5314  568  1147
20  Philadelphia Phillies    56    28.1  4.61  162  6077  5496  747  1392
21     Pittsburgh Pirates    68    26.3  3.65  162  5912  5331  591  1186
22       San Diego Padres    55    28.2  4.35  162  6175  5468  705  1317
23       Seattle Mariners    59    27.5  4.26  162  6117  5375  690  1236
24   San Francisco Giants    66    30.0  4.42  162  6117  5392  716  1261
25    St. Louis Cardinals    51    28.8  4.77  162  6165  5496  772  1386
26         Tampa Bay Rays    61    27.0  4.11  162  6008  5412  666  1294
27          Texas Rangers    55    28.0  4.36  162  6029  5478  707  1308
28      Toronto Blue Jays    51    27.1  4.78  162  6158  5555  775  1464
29   Washington Nationals    55    28.7  3.72  162  5998  5434  603  1351

     2B  …    OPS  OPS+    TB  GDP  HBP  SH  SF  IBB   LOB  Playoffs
0   262  …  0.689    95  2061   97   60  31  50   14  1039         0
1   298  …  0.761   109  2443  103   66   1  36   13  1030         1
2   275  …  0.695    99  2119   95   83  12  43   10  1095         0
3   352  …  0.731   102  2268  131   63  12  50   23  1133         0
4   265  …  0.698    94  2097  130   84  19  36   16  1100         0
5   272  …  0.698    97  2172  127   73  16  35    9  1117         1
6   235  …  0.676    85  2003  127   92  12  33    6  1020         0
7   273  …  0.699   102  2126  119   81  22  52   36  1156         1
8   280  …  0.713    91  2203  139   61  10  40   10  1113         1
9   235  …  0.632    82  1859  108   58  10  44    8  1015         0
10  284  …  0.743   111  2293  118   60   9  42   18  1068         1
11  247  …  0.686    93  2064  101   48  20  44    7  1091         0
12  219  …  0.687    93  2116   95   54  25  25   28  1050         0
13  325  …  0.775   115  2441   85   56   3  53   22  1159         1
14  248  …  0.658    85  1961  120   70   4  36    6  1045         0
15  251  …  0.724   103  2213  117   80  11  37   25  1102         1
```

```
16  269  …  0.718   105  2195  133   62  10  46   11  1126          0
17  272  …  0.744   113  2261  122  112  20  44   25  1158          1
18  225  …  0.751   112  2311  121   70  14  41   36  1093          1
19  249  …  0.626    84  1837  109   59  22  33    7   969          0
20  255  …  0.739   108  2320  116   52   6  44   15  1075          1
21  221  …  0.655    85  1939   96   54  19  32   14  1016          0
22  275  …  0.700   102  2087   95   65  17  46   24  1174          1
23  229  …  0.704   106  2094  120   89   9  45   17  1129          1
24  255  …  0.705   100  2101  109   95   6  53   14  1115          0
25  290  …  0.745   112  2309  112   80   5  45   11  1132          1
26  296  …  0.686    99  2041   93   57   7  31   13  1074          1
27  224  …  0.696    96  2166   82   47  10  38   12  1007          0
28  307  …  0.760   117  2395  136   55   8  33   13  1111          1
29  252  …  0.688    98  2051  141   60  20  37   12  1099          0

[30 rows x 30 columns]
```

[2]:
```
df.drop(columns=['Tm', '#Bat'], axis=1, inplace=True)
```

[3]:
```
X = df.iloc[:, 0:27]
y = df.iloc[:, 27]

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2,␣
 ↪random_state=21)
scaleStandard = StandardScaler()
X_train = scaleStandard.fit_transform(X_train)

df.columns
```

[3]:
```
Index(['BatAge', 'R/G', 'G', 'PA', 'AB', 'R', 'H', '2B', '3B', 'HR', 'RBI',
       'SB', 'CS', 'BB', 'SO', 'BA', 'OBP', 'SLG', 'OPS', 'OPS+', 'TB', 'GDP',
       'HBP', 'SH', 'SF', 'IBB', 'LOB', 'Playoffs'],
      dtype='object')
```

[4]:
```
X_train = pd.DataFrame(X_train, columns=['BatAge', 'R/G', 'G', 'PA', 'AB', 'R',␣
 ↪'H', '2B', '3B', 'HR', 'RBI',
       'SB', 'CS', 'BB', 'SO', 'BA', 'OBP', 'SLG', 'OPS', 'OPS+', 'TB', 'GDP',
       'HBP', 'SH', 'SF', 'IBB', 'LOB'])


X_train
```

[4]:
```
     BatAge       R/G    G        PA        AB         R         H        2B  \
0 -0.032987 -1.708056  0.0 -2.105939 -2.081256 -1.713743 -2.374828 -0.496051
1 -2.144152  0.099809  0.0  1.070839  1.875833  0.099373  1.267922  0.293245
2 -0.296883  0.212801  0.0 -0.348122  0.578427  0.224896 -0.144856 -1.318234
3 -1.088570 -0.713730  0.0 -0.549318 -0.086494 -0.709556  0.118309 -0.561825
```

```
4   1.638352  1.613896  0.0  1.166142 -0.329757  1.619600 -0.144856 -1.285346
5  -1.176535 -0.352157  0.0 -0.570496 -0.491933 -0.346933 -0.338766  1.049654
6  -1.616361  0.145006  0.0 -0.369300 -1.481206  0.155161 -1.197514 -0.068515
7  -0.208917  0.777758  0.0  0.160163  0.870343  0.782778  1.018608 -0.298727
8   0.934630 -0.600738  0.0 -0.888174 -1.010896 -0.597979 -0.754289 -0.956473
9   1.110561  2.178854  0.0  1.960337  1.356871  2.177482  1.378728  2.003386
10  0.318874 -1.233491  0.0 -0.676389 -0.135146 -1.225596  0.450727 -0.397389
11 -0.384848 -0.939713  0.0 -0.898763 -0.313540 -0.946656 -0.740438 -1.482670
12  0.494804 -1.459474  0.0 -1.195263 -0.767632 -1.462696 -1.072857 -0.528938
13  0.670735  0.099809  0.0  0.456662  1.583917  0.099373  1.240220  0.523457
14  0.846665  0.642169  0.0 -0.083390 -0.540586  0.643307  0.312219  0.655006
15  1.198526  1.139331  0.0  1.208499  0.756820  1.131454  1.434131  0.260358
16 -1.176535 -0.239165  0.0 -0.136337 -0.216234 -0.235356 -0.518826  0.359020
17  0.406839  0.619570  0.0  0.869643  1.567699  0.615413  1.503384  2.891344
18  0.406839  1.139331  0.0  1.092017  0.870343  1.131454  0.935503  0.852330
19 -1.264500  0.077211  0.0  0.541376  0.545992  0.071479  0.519981  0.161696
20 -0.384848 -0.465149  0.0  0.107216 -0.281105 -0.472456 -0.352617  0.030147
21  1.462422  0.348391  0.0  0.583733 -0.816285  0.350419 -0.795842 -0.298727
22  0.670735  0.483980  0.0  0.636679 -0.410845  0.475943 -0.657334 -0.430276
23 -0.384848 -1.866244  0.0 -2.031814 -1.043331 -1.867160 -1.086707 -0.956473

          3B        HR  …       SLG       OPS      OPS+        TB       GDP  \
0  -1.023632 -1.011987  … -1.991887 -2.198327 -1.578618 -2.106985 -0.269665
1   1.101096 -1.305671  … -0.440312 -0.149673  0.310476 -0.112710  0.352639
2  -0.359654  0.779487  …  0.062902 -0.233865 -0.319222  0.163314 -1.949889
3   2.030664 -0.982618  … -0.566115 -0.514502 -0.634071 -0.540547 -0.767509
4  -1.953200  2.424118  …  1.362870  1.309641  1.359973  1.163902  0.477100
5  -0.758041 -0.953250  … -0.691919 -0.514502 -0.004373 -0.699261 -1.265353
6   0.171527  0.045276  … -0.356443 -0.430311 -0.424172 -0.561249 -1.016431
7   0.835505  0.985066  …  1.195132  0.972877  0.940174  1.226008  0.165948
8  -0.625245 -0.453987  … -0.901591 -0.795139 -1.473669 -0.961484  0.850483
9   1.101096  1.190645  …  2.033822  1.983171  1.674822  2.060981 -1.763197
10 -0.359654 -1.041355  … -0.691919 -0.458375 -0.109323 -0.630255  1.721710
11  1.101096  0.544539  … -0.146771 -0.486438 -0.634071 -0.181716 -1.140892
12 -0.359654 -0.806408  … -1.279001 -1.300287 -1.473669 -1.251310  0.414870
13  1.499482 -0.659566  …  0.188705  0.243219 -0.843970  0.418637  1.597249
14 -1.289223  1.249381  …  1.279001  1.085132  1.255023  1.039691  0.290409
15  0.569914 -0.013461  …  0.775788  1.113195  1.464923  0.818872  0.539331
16  0.304323 -0.013461  … -0.146771 -0.261928 -0.004373 -0.161014 -1.140892
17 -1.422018 -0.483355  …  0.649984  0.748367  0.310476  0.867176  1.099405
18 -0.226859  0.750118  …  1.111263  1.141259  1.359973  1.150101 -0.082974
19 -0.625245  0.192118  …  0.314508  0.383538  0.625325  0.363432  1.223866
20  1.101096 -0.365882  … -0.272574 -0.177737 -0.529121 -0.312827  1.037175
21 -0.625245  0.338960  … -0.146771  0.018709  0.100577 -0.285225 -0.269665
22 -0.758041  1.396224  …  0.649984  0.551920  0.415426  0.487643  0.228178
23  0.569914 -1.804934  … -1.991887 -2.029944 -1.788518 -1.955172 -0.331896
```

```
         HBP        SH        SF       IBB       LOB
0  -0.590017  1.201209 -1.197071 -1.046090 -2.339528
1   0.776338  1.201209  1.465682  2.309111  1.440098
2  -1.335301 -0.462003 -0.496347 -0.467607 -1.571476
3  -1.273194  0.924007  0.344523 -1.046090  0.126324
4   0.093161  0.092401 -0.075912  2.309111  0.166748
5  -0.714231 -0.877807 -1.477361 -0.351910 -0.217278
6  -0.527910  2.448618  1.185393 -0.236214 -0.924695
7  -1.024766 -1.016408  0.344523 -0.120517 -0.197066
8   1.459515 -0.184801 -1.197071 -1.161786 -1.308721
9  -0.776338 -1.432211  1.605827  0.689359  1.500734
10 -0.527910  0.924007 -0.636492 -0.467607  0.288020
11 -0.900552  1.617012 -2.318231  1.383538 -0.702364
12  0.093161 -1.293610 -0.776637 -1.161786 -0.803423
13 -0.465803 -0.462003 -0.216057 -0.699000  0.570986
14 -0.527910 -0.600604  0.064233  0.226572 -0.338549
15  2.701655  0.924007  0.344523  1.036449  1.480522
16  0.900552 -0.184801  0.204378 -0.699000  0.207172
17 -0.341589 -0.184801  1.185393  0.805055  0.975224
18  0.714231 -1.155009  0.484668 -0.583304  0.955012
19 -0.403696 -0.462003  0.624813 -0.583304  0.833741
20  0.962659  0.785406 -0.776637 -0.004821  0.308232
21  1.645836 -1.016408  1.605827 -0.236214  0.611410
22  0.714231 -0.323402 -0.636492  1.036449  0.348655
23 -0.652124 -0.462003  0.344523 -0.930393 -1.409780

[24 rows x 27 columns]
```

```
[5]: X_train.describe().round(3)
```

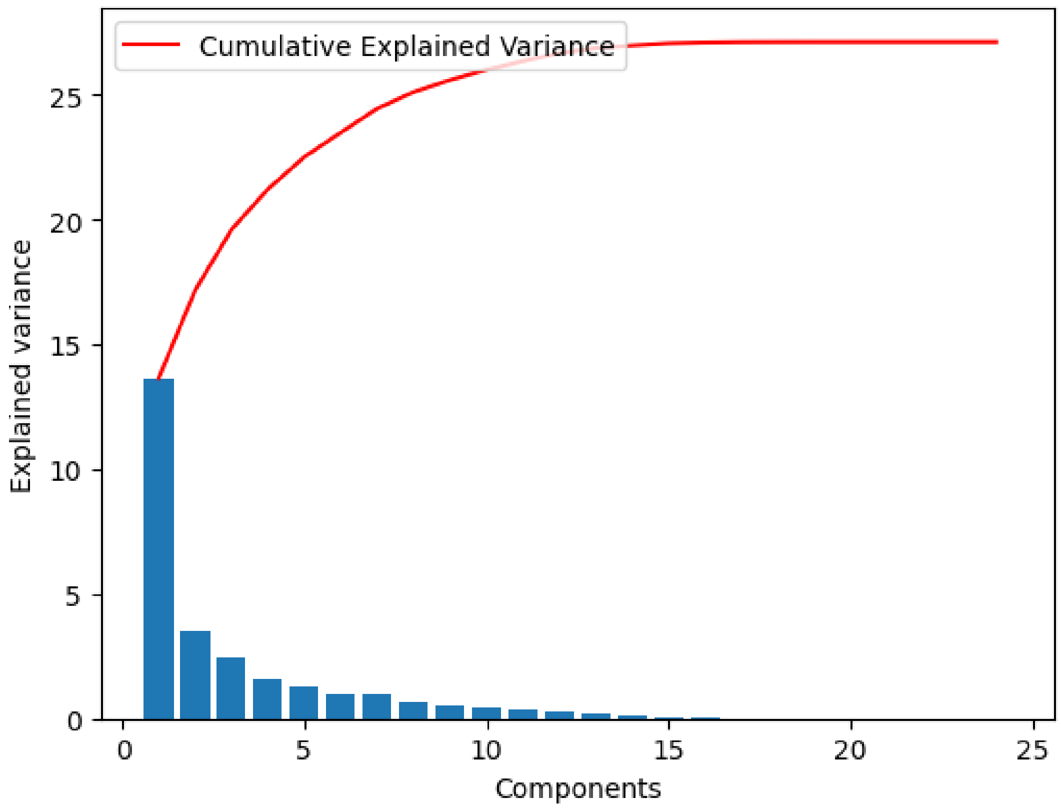
```
[5]:       BatAge     R/G     G      PA      AB       R       H      2B      3B  \
count  24.000  24.000  24.0  24.000  24.000  24.000  24.000  24.000  24.000
mean    0.000   0.000   0.0  -0.000   0.000   0.000   0.000   0.000   0.000
std     1.022   1.022   0.0   1.022   1.022   1.022   1.022   1.022   1.022
min    -2.144  -1.866   0.0  -2.106  -2.081  -1.867  -2.375  -1.483  -1.953
25%    -0.561  -0.629   0.0  -0.597  -0.597  -0.626  -0.744  -0.537  -0.658
50%     0.143   0.100   0.0   0.012  -0.249   0.099  -0.145  -0.184  -0.293
75%     0.715   0.625   0.0   0.695   0.785   0.622   0.956   0.400   0.902
max     1.638   2.179   0.0   1.960   1.876   2.177   1.503   2.891   2.031

           HR  …     SLG     OPS    OPS+      TB     GDP     HBP      SH  \
count  24.000  …  24.000  24.000  24.000  24.000  24.000  24.000  24.000
mean   -0.000  …   0.000  -0.000  -0.000  -0.000   0.000   0.000  -0.000
std     1.022  …   1.022   1.022   1.022   1.022   1.022   1.022   1.022
min    -1.805  …  -1.992  -2.198  -1.789  -2.107  -1.950  -1.335  -1.432
25%    -0.843  …  -0.598  -0.493  -0.634  -0.579  -0.830  -0.668  -0.670
50%    -0.013  …  -0.147  -0.164  -0.004  -0.137   0.197  -0.435  -0.254
```

```
75%      0.757  …   0.681   0.804   0.704   0.831   0.617   0.730   0.924
max      2.424  …   2.034   1.983   1.675   2.061   1.722   2.702   2.449

             SF     IBB     LOB
count   24.000  24.000  24.000
mean     0.000  -0.000   0.000
std      1.022   1.022   1.022
min     -2.318  -1.162  -2.340
25%     -0.672  -0.699  -0.728
50%      0.134  -0.294   0.187
75%      0.520   0.718   0.667
max      1.606   2.309   1.501

[8 rows x 27 columns]
```

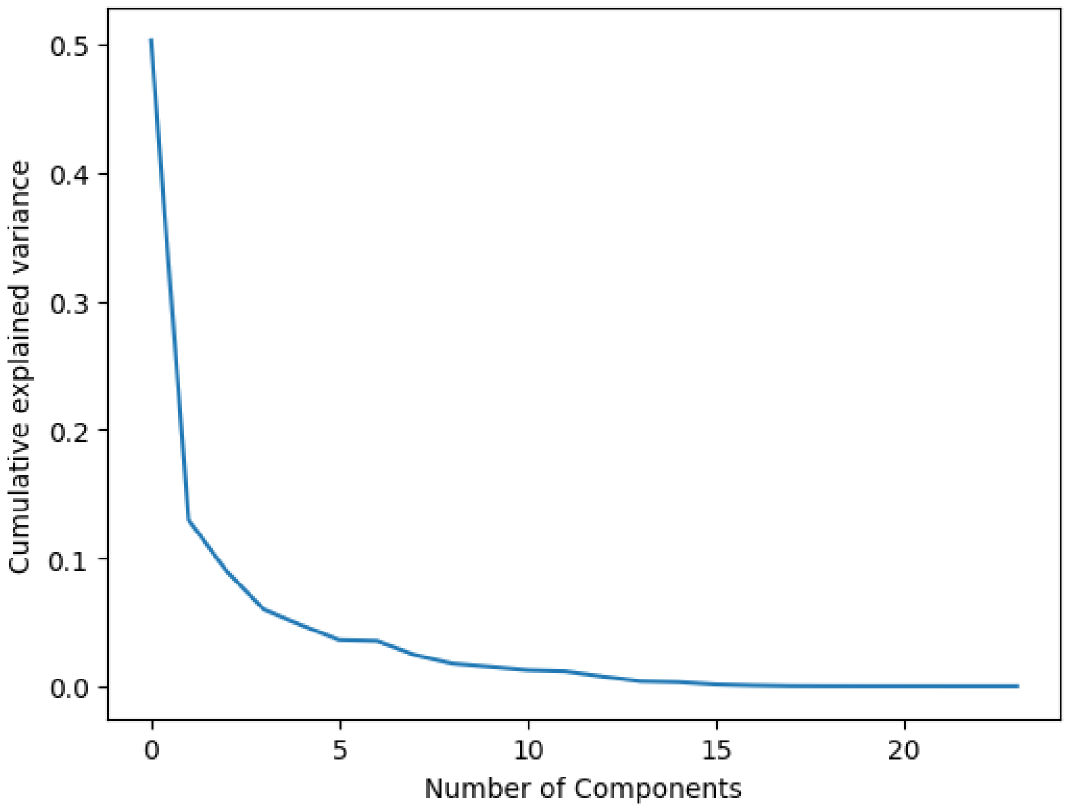
[6]:
```
pca1 = PCA()
X_pca1 = pca1.fit_transform(X_train)
pca1.explained_variance_ratio_
```

[6]:
```
array([5.03143586e-01, 1.29565631e-01, 8.99453783e-02, 5.97627178e-02,
       4.75204409e-02, 3.60767505e-02, 3.53674555e-02, 2.44602746e-02,
       1.77297976e-02, 1.52080776e-02, 1.26222548e-02, 1.16581886e-02,
       7.54086691e-03, 3.93586244e-03, 3.24827865e-03, 1.37155962e-03,
       6.27774032e-04, 1.54100200e-04, 4.62620159e-05, 9.33531667e-06,
       4.48882814e-06, 6.43748708e-07, 2.74352526e-07, 3.89906609e-34])
```

[7]:
```
plt.bar(range(1, len(pca1.explained_variance_) + 1), pca1.explained_variance_)
plt.ylabel("Explained variance")
plt.xlabel('Components')
plt.plot(range(1, len(pca1.explained_variance_) + 1), np.cumsum(pca1.
  ↪explained_variance_), c='red', label="Cumulative Explained Variance")
plt.legend(loc="upper left")
plt.show()
```



[8]:
```
plt.plot(pca1.explained_variance_ratio_)
plt.xlabel('Number of Components')
plt.ylabel('Cumulative explained variance')
plt.show()
```



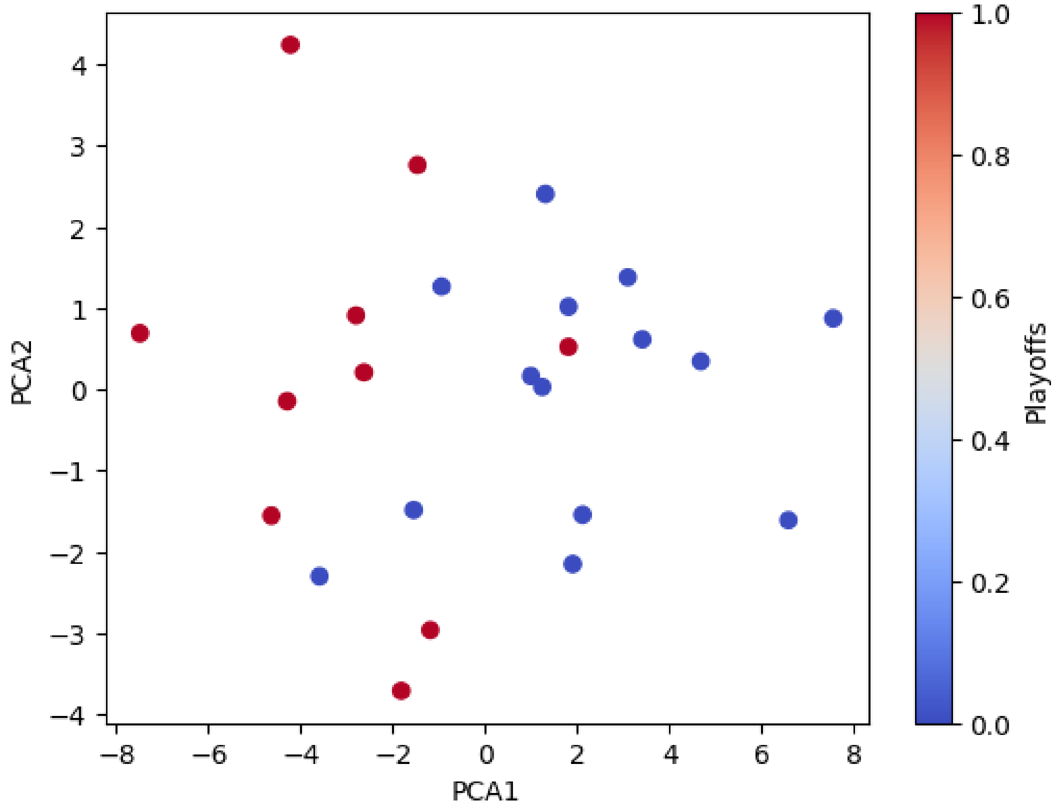
```
[9]: pca2 = PCA(0.95)
     X_pca2 = pca2.fit_transform(X_train)
     X_pca2.shape
```

```
[9]: (24, 10)
```

```
[10]: pca2.explained_variance_ratio_
```

```
[10]: array([0.50314359, 0.12956563, 0.08994538, 0.05976272, 0.04752044,
             0.03607675, 0.03536746, 0.02446027, 0.0177298 , 0.01520808])
```

```
[11]: pca2c = PCA(n_components=2)
      X_pca2c = pca2c.fit_transform(X_train)

      colormap = plt.get_cmap('coolwarm')
      plt.figure()
      scatter = plt.scatter(X_pca2c[:, 0], X_pca2c[:, 1], c=y_train, cmap=colormap)
      plt.xlabel('PCA1')
      plt.ylabel('PCA2')
      plt.colorbar(scatter, label="Playoffs")
      plt.show()
```


PCA1
PCA2
Playoffs

17-AdaBoost

October 20, 2024

1 Adaboost

AdaBoost (Adaptive Boosting) is an ensemble learning technique that combines multiple weak classifiers to form a strong classifier. It works by training weak classifiers (often decision stumps or shallow trees) sequentially, where each classifier focuses more on the mistakes made by the previous ones.

The key idea behind AdaBoost is to assign more weight to the misclassified instances from the previous classifier, so that the next classifier can focus on these harder-to-classify examples. By combining the predictions of these weak learners, AdaBoost can create a highly accurate model.

AdaBoost = Adaptive + Boosting

- **Boosting**: Combining weak classifiers to create a strong one.
- **Adaptive**: Adjusting the importance (weights) of misclassified examples.

When to Use AdaBoost? AdaBoost is particularly useful when:

- You want to improve the performance of a weak learner (such as a decision stump).
- You are dealing with binary classification problems, although it can be extended to multi-class classification as well.
- You need an efficient model that can provide high accuracy without too much computational cost.
- Your dataset has moderate noise. However, AdaBoost can be sensitive to outliers, so noisy datasets should be preprocessed carefully.

How Does AdaBoost Work? AdaBoost trains classifiers sequentially, each one focusing more on the mistakes of the previous one. Here's the step-by-step breakdown:

1. Initialize Weights:

- All instances in the training set are given equal weights initially. For N data points, the weight for each data point is 1/N.

2. Train the First Weak Classifier:

A weak classifier (usually a decision stump) is trained on the weighted data. It predicts the class for each example, and the weighted error is calculated.

3. Update Weights:

- Increase the weights of the misclassified examples so that the next classifier pays more attention to these points.
- Decrease the weights of correctly classified examples.

4. **Repeat:**
   - Train a new weak classifier using the updated weights.
   - Adjust the weights based on the errors of the new classifier.
   - This process continues for a specified number of iterations or until a stopping criterion is met.
5. **Combine Weak Learners:**
   - After multiple iterations, the weak classifiers are combined to make the final prediction.
   - Each classifier is given a weight based on its accuracy, with more accurate classifiers contributing more to the final decision.

1.0.1 Who Should Use AdaBoost?

- **Data scientists and machine learning engineers**: Who want to improve the performance of simple models in tasks like binary classification.
- **Beginners and intermediate learners**: AdaBoost is relatively simple to understand, making it suitable for learners who are exploring boosting techniques.
- **Industries that rely on classification tasks**: AdaBoost can be used in fraud detection, text classification, image recognition, and customer churn prediction.

Advantages of AdaBoost:

- **Improves weak learners**: Transforms weak classifiers into a strong one.
- **No parameter tuning**: Unlike many machine learning algorithms, AdaBoost often works well with minimal tuning.
- **Efficient and scalable**: It can handle large datasets efficiently.
- **Feature selection**: AdaBoost implicitly performs feature selection by giving more importance to features that are useful for classification.

Disadvantages of AdaBoost:

- **Sensitive to noisy data**: Outliers can disrupt the performance of AdaBoost because they get higher weights, causing overfitting.
- **Computation cost**: For very large datasets, the sequential training of weak learners can become computationally expensive.
- **Works best with weak learners**: Using a strong learner as the base model can lead to overfitting.

[43]:
```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.datasets import make_classification
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,␣
↪classification_report

from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
```

[44]:
```
X, y = make_classification(n_samples=2000, n_features=10, n_informative=8,␣
↪n_redundant=2, random_state=11)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
↪2,random_state=11)
abc = AdaBoostClassifier()
abc.fit(X_train, y_train)
```

```
c:\Users\ikiga\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\ensemble\_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
```

[44]:
```
AdaBoostClassifier()
```

[45]:
```
y_pred = abc.predict(X_test)
y_pred
```

[45]:
```
array([0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
       1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0,
       0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
       1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0,
       1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
       0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0,
       0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0,
       0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1,
       0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
       0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0,
       1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,
       1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
       1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
       1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1,
       1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0,
       1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0,
       1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1,
       0, 1, 0, 0])
```

[46]:
```
print(accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.815
[[171  34]
 [ 40 155]]
              precision    recall  f1-score   support

           0       0.81      0.83      0.82       205
           1       0.82      0.79      0.81       195

    accuracy                           0.81       400
   macro avg       0.82      0.81      0.81       400
weighted avg       0.82      0.81      0.81       400
```

1.1 Logistic Regression

[47]:
```
abclog = AdaBoostClassifier(estimator=LogisticRegression())
abclog.fit(X_train, y_train)
```

```
c:\Users\ikiga\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\ensemble\_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
```

[47]:
```
AdaBoostClassifier(estimator=LogisticRegression())
```

[48]:
```
y_pred2 = abclog.predict(X_test)
y_pred2
```

[48]:
```
array([0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
       0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0,
       0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1,
       1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0,
       1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1,
       0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0,
       0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1,
       1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0,
       1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1,
       0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1,
       0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
       1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1,
       1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1,
       1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0,
       1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
       1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1,
       0, 0, 0, 0])
```

```
[49]: print(accuracy_score(y_test, y_pred2))
      print(confusion_matrix(y_test, y_pred2))
      print(classification_report(y_test, y_pred2))
```

```
0.785
[[167  38]
 [ 48 147]]
              precision    recall  f1-score   support

           0       0.78      0.81      0.80       205
           1       0.79      0.75      0.77       195

    accuracy                           0.79       400
   macro avg       0.79      0.78      0.78       400
weighted avg       0.79      0.79      0.78       400
```

1.2 Support Vector Machine

```
[50]: svc = SVC(kernel='linear', probability=True)

      abcsvm = AdaBoostClassifier(estimator=svc, n_estimators=25, learning_rate=0.1)
      abcsvm.fit(X_train, y_train)
```

```
c:\Users\ikiga\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\ensemble\_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
```

```
[50]: AdaBoostClassifier(estimator=SVC(kernel='linear', probability=True),
                         learning_rate=0.1, n_estimators=25)
```

```
[51]: y_pred3 = abcsvm.predict(X_test)
      y_pred3
```

```
[51]: array([0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
             0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0,
             0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1,
             1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0,
             1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1,
             0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0,
             0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1,
             1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0,
             1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1,
             0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1,
             0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
             0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
             1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1,
             1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1,
             0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1,
             1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0,
             1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
             1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1,
             0, 0, 0, 0])
```

[52]:
```
print(accuracy_score(y_test, y_pred3))
print(confusion_matrix(y_test, y_pred3))
print(classification_report(y_test, y_pred3))
```

```
0.7925
[[168  37]
 [ 46 149]]
              precision    recall  f1-score   support

           0       0.79      0.82      0.80       205
           1       0.80      0.76      0.78       195

    accuracy                           0.79       400
   macro avg       0.79      0.79      0.79       400
weighted avg       0.79      0.79      0.79       400
```

1.3 Hyperparameter Tuning

[54]:
```
param_grid = {
    'n_estimators': [1,5,10,25,50,100,500],
    'learning_rate': [0.00001, 0.0001, 0.001, 0.1, 0.5, 1.0]
}

abc_grid = GridSearchCV(abc, param_grid, cv=3, n_jobs=-1)
abc_grid.fit(X_train, y_train)
```

```
c:\Users\ikiga\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\ensemble\_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
```

[54]:
```
GridSearchCV(cv=3, estimator=AdaBoostClassifier(), n_jobs=-1,
             param_grid={'learning_rate': [1e-05, 0.0001, 0.001, 0.1, 0.5, 1.0],
                         'n_estimators': [1, 5, 10, 25, 50, 100, 500]})
```

[55]:
```
abc_grid.best_params_
```

[55]:
```
{'learning_rate': 0.1, 'n_estimators': 500}
```

[56]:
```
abc2 = AdaBoostClassifier(learning_rate=0.2, n_estimators=500)
abc2.fit(X_train, y_train)
```

```
c:\Users\ikiga\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\ensemble\_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
```

[56]:
```
AdaBoostClassifier(learning_rate=0.2, n_estimators=500)
```

[58]:
```
y_pred4 = abc2.predict(X_test)
y_pred4
```

[58]:
```
array([0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1,
       1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0,
       0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1,
       1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0,
       1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
       0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0,
       0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0,
       0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1,
       0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
       0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
       1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
       1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1,
       1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1,
       0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1,
       1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0,
       1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0,
       0, 1, 0, 0])
```

[59]:
```
print(accuracy_score(y_test, y_pred4))
print(confusion_matrix(y_test, y_pred4))
print(classification_report(y_test, y_pred4))
```

```
0.8275
[[173  32]
 [ 37 158]]
              precision    recall  f1-score   support

           0       0.82      0.84      0.83       205
           1       0.83      0.81      0.82       195

    accuracy                           0.83       400
   macro avg       0.83      0.83      0.83       400
weighted avg       0.83      0.83      0.83       400
```

18-Gradient_Boosring

October 20, 2024

1 Gradient Boosting

Gradient Boosting is an ensemble learning technique that builds models in a stage-wise fashion. It creates a strong predictive model by combining the predictions of several weaker models (often decision trees). Unlike AdaBoost, which focuses on misclassified instances, Gradient Boosting minimizes a loss function using the gradient descent optimization algorithm, hence the name "Gradient Boosting."

1.0.1 When to Use Gradient Boosting?

Gradient Boosting is particularly useful when:

- You have structured or tabular data.
- You want to achieve high accuracy and predictive performance.
- Your data contains complex patterns that simpler models struggle to capture.
- You are interested in both regression and classification tasks.

Gradient Boosting can handle a variety of loss functions and is often employed in Kaggle competitions due to its robustness.

1.0.2 How Does Gradient Boosting Work?

Gradient Boosting works by sequentially adding weak learners (typically shallow decision trees) to the ensemble, each one correcting the errors made by the previous learners. Here's a step-by-step breakdown:

1. **Initialize the Model:**
   - Start with a base prediction (often the mean of the target values for regression tasks).
2. **Calculate Residuals:**
   - Calculate the residuals (the difference between actual values and predicted values) of the current model.
3. **Fit a Weak Learner:**
   - Train a new weak learner (e.g., decision tree) on the residuals to predict these errors. The idea is to learn the error of the previous model.
4. **Update the Model:**

- Update the model by adding the predictions of the new weak learner to the current model predictions, scaled by a learning rate (denoted as).

5. **Repeat:**

- Repeat steps 2 to 4 for a specified number of iterations or until a stopping criterion is met.

6. **Final Prediction:**

- The final prediction is a weighted sum of the weak learners' predictions.

1.0.3 Who Should Use Gradient Boosting?

- **Data scientists and machine learning practitioners**: Who want to build predictive models that leverage the power of ensemble learning.
- **Kaggle competitors**: Gradient Boosting is popular in data science competitions for its high predictive accuracy and flexibility.
- **Industries that require predictive analytics**: Applicable in finance, marketing, healthcare, and other fields that deal with regression or classification problems.

Advantages of Gradient Boosting:

- **High predictive performance**: Often achieves state-of-the-art results on various tasks.
- **Flexibility: Can optimize various loss functions and handle different types of data (e.g., continuous, categorical). Robustness to overfitting**: When used with techniques like early stopping or regularization (e.g., limiting tree depth).
- **Feature importance**: Provides insight into the importance of features in making predictions.

Disadvantages of Gradient Boosting:

- **Sensitive to noise and outliers**: Can overfit if not properly regularized.
- **Longer training times**: Sequential training can be slower compared to parallelized models like Random Forest.
- **Complexity**: Requires careful tuning of hyperparameters (e.g., number of trees, learning rate, tree depth) for optimal performance.

1.0.4 Real-World Applications of Gradient Boosting:

- **Finance**: Credit scoring and risk assessment, where high accuracy is crucial.
- **Customer segmentation**: In marketing analytics, to understand customer behavior.
- **Healthcare**: Predictive models for patient outcomes and disease risk.
- **Recommendation systems**: To personalize user experiences based on historical data.

[19]:
```
import pandas as pd

from sklearn import datasets
from sklearn.model_selection import train_test_split, cross_val_score,␣
 ↪GridSearchCV
```

```
from sklearn.ensemble import GradientBoostingClassifier


wine = datasets.load_wine(as_frame=True)
wine
```

```
[19]: {'data':      alcohol  malic_acid   ash  alcalinity_of_ash  magnesium
total_phenols  \
 0      14.23        1.71  2.43               15.6      127.0           2.80
 1      13.20        1.78  2.14               11.2      100.0           2.65
 2      13.16        2.36  2.67               18.6      101.0           2.80
 3      14.37        1.95  2.50               16.8      113.0           3.85
 4      13.24        2.59  2.87               21.0      118.0           2.80
 ..       …          …    …                 …         …             …
 173    13.71        5.65  2.45               20.5       95.0           1.68
 174    13.40        3.91  2.48               23.0      102.0           1.80
 175    13.27        4.28  2.26               20.0      120.0           1.59
 176    13.17        2.59  2.37               20.0      120.0           1.65
 177    14.13        4.10  2.74               24.5       96.0           2.05

      flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue
\
 0          3.06                  0.28             2.29             5.64  1.04
 1          2.76                  0.26             1.28             4.38  1.05
 2          3.24                  0.30             2.81             5.68  1.03
 3          3.49                  0.24             2.18             7.80  0.86
 4          2.69                  0.39             1.82             4.32  1.04
 ..          …                   …                …                …    …
 173        0.61                  0.52             1.06             7.70  0.64
 174        0.75                  0.43             1.41             7.30  0.70
 175        0.69                  0.43             1.35            10.20  0.59
 176        0.68                  0.53             1.46             9.30  0.60
 177        0.76                  0.56             1.35             9.20  0.61

      od280/od315_of_diluted_wines  proline
 0                            3.92   1065.0
 1                            3.40   1050.0
 2                            3.17   1185.0
 3                            3.45   1480.0
 4                            2.93    735.0
 ..                            …       …
 173                          1.74    740.0
 174                          1.56    750.0
 175                          1.56    835.0
 176                          1.62    840.0
 177                          1.60    560.0
 [178 rows x 13 columns],
 'target': 0      0
 1      0
 2      0
 3      0
 4      0
       ..
 173    2
 174    2
 175    2
 176    2
 177    2
 Name: target, Length: 178, dtype: int32,
 'frame':      alcohol  malic_acid   ash  alcalinity_of_ash  magnesium
total_phenols  \
 0      14.23        1.71  2.43               15.6      127.0           2.80
 1      13.20        1.78  2.14               11.2      100.0           2.65
 2      13.16        2.36  2.67               18.6      101.0           2.80
 3      14.37        1.95  2.50               16.8      113.0           3.85
 4      13.24        2.59  2.87               21.0      118.0           2.80
 ..       …          …   …                 …         …             …
 173    13.71        5.65  2.45               20.5       95.0           1.68
 174    13.40        3.91  2.48               23.0      102.0           1.80
 175    13.27        4.28  2.26               20.0      120.0           1.59
 176    13.17        2.59  2.37               20.0      120.0           1.65
 177    14.13        4.10  2.74               24.5       96.0           2.05

      flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue
\
 0          3.06                  0.28             2.29             5.64  1.04
 1          2.76                  0.26             1.28             4.38  1.05
 2          3.24                  0.30             2.81             5.68  1.03
 3          3.49                  0.24             2.18             7.80  0.86
 4          2.69                  0.39             1.82             4.32  1.04
 ..          …                   …                …                …    …
 173        0.61                  0.52             1.06             7.70  0.64
 174        0.75                  0.43             1.41             7.30  0.70
 175        0.69                  0.43             1.35            10.20  0.59
 176        0.68                  0.53             1.46             9.30  0.60
 177        0.76                  0.56             1.35             9.20  0.61

      od280/od315_of_diluted_wines  proline  target
 0                            3.92   1065.0       0
 1                            3.40   1050.0       0
 2                            3.17   1185.0       0
 3                            3.45   1480.0       0
 4                            2.93    735.0       0
```

```
..                           ...      ...     ...
173                          1.74    740.0       2
174                          1.56    750.0       2
175                          1.56    835.0       2
176                          1.62    840.0       2
177                          1.60    560.0       2

[178 rows x 14 columns],
'target_names': array(['class_0', 'class_1', 'class_2'], dtype='<U7'),
'DESCR': '.. _wine_dataset:\n\nWine recognition
dataset\n------------------------\n\n**Data Set Characteristics:**\n\n:Number of
Instances: 178\n:Number of Attributes: 13 numeric, predictive attributes and the
class\n:Attribute Information:\n    - Alcohol\n    - Malic acid\n    - Ash\n
- Alcalinity of ash\n    - Magnesium\n    - Total phenols\n    - Flavanoids\n
- Nonflavanoid phenols\n    - Proanthocyanins\n    - Color intensity\n    -
Hue\n    - OD280/OD315 of diluted wines\n    - Proline\n    - class:\n        -
class_0\n        - class_1\n        - class_2\n\n:Summary
Statistics:\n\n============================= ==== ===== ======= =====\n
Min   Max   Mean     SD\n============================= ==== ===== =======
=====\nAlcohol:                      11.0  14.8    13.0   0.8\nMalic Acid:
0.74  5.80    2.34  1.12\nAsh:                          1.36  3.23    2.36
0.27\nAlcalinity of Ash:            10.6  30.0    19.5   3.3\nMagnesium:
70.0 162.0    99.7  14.3\nTotal Phenols:                0.98  3.88    2.29
0.63\nFlavanoids:                   0.34  5.08    2.03  1.00\nNonflavanoid
Phenols:          0.13  0.66    0.36  0.12\nProanthocyanins:              0.41
3.58    1.59  0.57\nColour Intensity:             1.3  13.0     5.1   2.3\nHue:
0.48  1.71    0.96  0.23\nOD280/OD315 of diluted wines: 1.27  4.00    2.61
0.71\nProline:                      278  1680     746
315\n============================= ==== ===== ======= =====\n\n:Missing
Attribute Values: None\n:Class Distribution: class_0 (59), class_1 (71), class_2
(48)\n:Creator: R.A. Fisher\n:Donor: Michael Marshall
(MARSHALL%PLU@io.arc.nasa.gov)\n:Date: July, 1988\n\nThis is a copy of UCI ML
Wine recognition datasets.\nhttps://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data\n\nThe data is the results of a chemical analysis of
wines grown in the same\nregion in Italy by three different cultivators. There
are thirteen different\nmeasurements taken for different constituents found in
the three types of\nwine.\n\nOriginal Owners:\n\nForina, M. et al, PARVUS -\nAn
Extendible Package for Data Exploration, Classification and
Correlation.\nInstitute of Pharmaceutical and Food Analysis and
Technologies,\nVia Brigata Salerno, 16147 Genoa, Italy.\n\nCitation:\n\nLichman,
M. (2013). UCI Machine Learning Repository\n[https://archive.ics.uci.edu/ml].
Irvine, CA: University of California,\nSchool of Information and Computer
Science.\n\n|details-start|\n**References**\n|details-split|\n\n(1) S.
Aeberhard, D. Coomans and O. de Vel,\nComparison of Classifiers in High
Dimensional Settings,\nTech. Rep. no. 92-02, (1992), Dept. of Computer Science
and Dept. of\nMathematics and Statistics, James Cook University of North
Queensland.\n(Also submitted to Technometrics).\n\nThe data was used with many
```

```
others for comparing various\nclassifiers. The classes are separable, though
only RDA\nhas achieved 100% correct classification.\n(RDA : 100%, QDA 99.4%, LDA
98.9%, 1NN 96.1% (z-transformed data))\n(All results using the leave-one-out
technique)\n\n(2) S. Aeberhard, D. Coomans and O. de Vel,\n"THE CLASSIFICATION
PERFORMANCE OF RDA"\nTech. Rep. no. 92-01, (1992), Dept. of Computer Science and
Dept. of\nMathematics and Statistics, James Cook University of North
Queensland.\n(Also submitted to Journal of Chemometrics).\n\n|details-end|\n',
 'feature_names': ['alcohol',
  'malic_acid',
  'ash',
  'alcalinity_of_ash',
  'magnesium',
  'total_phenols',
  'flavanoids',
  'nonflavanoid_phenols',
  'proanthocyanins',
  'color_intensity',
  'hue',
  'od280/od315_of_diluted_wines',
  'proline']}
```

[20]:
```
X = wine['data']
y = wine['target']
```

[21]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=17)

gbr = GradientBoostingClassifier()
gbr.fit(X_train, y_train)
```

[21]:
```
GradientBoostingClassifier()
```

[22]:
```
cross_val_score(gbr, X_train, y_train, cv=3, n_jobs=-1).mean()
```

[22]:
```
0.9221335697399526
```

[23]:
```
param_grid = {
    'n_estimators': [10,50,100, 500],
    'learning_rate': [0.0001, 0.001, 0.01, 0.1, 1],
    'max_depth': [3,5,7,9]
}

gbr2 = GridSearchCV(gbr, param_grid, cv=3, n_jobs=-1)
gbr2.fit(X_train, y_train)
```

[23]:
```
GridSearchCV(cv=3, estimator=GradientBoostingClassifier(), n_jobs=-1,
             param_grid={'learning_rate': [0.0001, 0.001, 0.01, 0.1, 1],
                                    'max_depth': [3, 5, 7, 9],
                                    'n_estimators': [10, 50, 100, 500]})
```

[24]:
```
gbr2.best_params_
```

[24]:
```
{'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
```

[25]:
```
gbr2.best_score_
```

[25]:
```
0.9434101654846335
```

19-Extra_Trees_Classifier

October 20, 2024

1 Extra Tress Classifier

Extra Trees, short for Extremely Randomized Trees, is an ensemble learning technique that builds a collection of decision trees in a random manner. Like Random Forest, it constructs multiple decision trees, but it introduces additional randomness into the tree-building process. This results in an ensemble method that typically provides better performance and is less prone to overfitting than traditional decision trees.

Extra Trees can be used for both classification and regression tasks. The key idea is to leverage the randomness in both the selection of features and the thresholds for splitting nodes to create a diverse set of trees.

Extra Trees Classifier is a powerful and efficient ensemble method that excels in classification tasks. Its ability to introduce randomness while leveraging the strengths of decision trees makes it a valuable tool in the machine learning toolkit.

1.0.1 When to Use Extra Trees Classifier?

Extra Trees Classifier is particularly useful when:

- You want to improve the performance of a single decision tree or even Random Forest by introducing more randomness.
- Your dataset is large and complex, and you are looking for a robust model that can capture intricate patterns.
- You are interested in reducing overfitting compared to other tree-based models.
- You need an efficient model for classification or regression tasks that can handle high-dimensional data.

1.0.2 How Does Extra Trees Classifier Work?

The Extra Trees Classifier builds decision trees using the following steps:

1. **Bootstrapping**:
   - Unlike Random Forest, which samples the training data with replacement, Extra Trees uses the entire dataset for each tree, ensuring all data points are considered.
2. **Random Feature Selection:**
   - For each node in a tree, a random subset of features is selected. Instead of considering all features, the model randomly selects a subset of features to determine the best split.

3. **Threshold Selection**:
    - Instead of using the best split point based on the optimization criterion (like Gini impurity or information gain), Extra Trees selects the split threshold randomly from the possible values for the selected features. This means that the trees are built using more randomness, hence the name "Extremely Randomized."
4. **Tree Construction**:
    - The process of splitting nodes continues recursively until a stopping criterion is reached (e.g., maximum depth, minimum samples per leaf).
5. **Prediction**:
    - For classification tasks, predictions are made based on the majority vote of all the trees in the ensemble. For regression tasks, the average of the predictions from all trees is used.

1.0.3 Who Should Use Extra Trees Classifier?

- **Data scientists and machine learning engineers**: Who want to build robust classification models that leverage tree ensembles.
- **Kaggle competitors**: Extra Trees is often used in competitions for its high accuracy and efficiency.
- **Industries requiring fast and accurate models**: Suitable for applications in finance, healthcare, and customer analytics, where quick and reliable predictions are needed.

Advantages of Extra Trees Classifier:

- **High accuracy**: Typically provides better accuracy than single decision trees and sometimes outperforms Random Forest due to increased randomness.
- **Less prone to overfitting**: The additional randomness in feature selection and threshold determination helps reduce overfitting.
- **Fast training time**: Since all data points are used without bootstrapping and splits are chosen randomly, Extra Trees can train faster than other tree-based methods.
- **Feature importance**: Like other tree-based models, Extra Trees can provide insights into feature importance.

Disadvantages of Extra Trees Classifier:

- **Interpretability**: While feature importance is available, the ensemble nature makes it harder to interpret compared to a single decision tree.
- **Sensitivity to noise**: Although less than traditional decision trees, Extra Trees can still be sensitive to noise and outliers in the data.
- **Limited extrapolation**: The predictions can be less reliable for unseen data outside the range of the training data.

1.0.4 Real-World Applications of Extra Trees Classifier:

- **Credit scoring**: To assess risk based on various financial indicators.

- **Image classification**: Used in computer vision tasks to classify images based on features extracted from pixel data.
- **Customer segmentation**: For marketing strategies and customer behavior analysis.
- **Medical diagnosis**: In predicting patient outcomes based on clinical data.

[7]:
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, cross_val_score,␣
↪GridSearchCV
from sklearn.ensemble import ExtraTreesClassifier

X, y = make_classification(n_features=11, random_state=21)

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,␣
↪random_state=16)

etc = ExtraTreesClassifier(random_state=21)
etc.fit(X_train, y_train)
```

[7]:
```
ExtraTreesClassifier(random_state=21)
```

[5]:
```
cross_val_score(etc, X_train, y_train, scoring='accuracy', cv=5, n_jobs=-1).
↪mean()
```

[5]:
```
0.9375
```

[8]:
```
param_grid = {
    'criterion': ['gini', 'entropy'],
    'n_estimators': [100, 250, 500],
    'min_samples_leaf': [5,15,25],
    'max_features': [3,5,7,9,11]
}

etc2 = GridSearchCV(etc,param_grid, cv=3, n_jobs=-1)

etc2.fit(X_train, y_train)
```

[8]:
```
GridSearchCV(cv=3, estimator=ExtraTreesClassifier(random_state=21), n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_features': [3, 5, 7, 9, 11],
                         'min_samples_leaf': [5, 15, 25],
                         'n_estimators': [100, 250, 500]})
```

[10]:
```
etc2.best_params_
```

[10]:
```
{'criterion': 'gini',
 'max_features': 5,
 'min_samples_leaf': 5,
 'n_estimators': 500}
```

```
[12]: etc2.best_score_
```

```
[12]: 0.9377967711301044
```

20-Voting_Classifier

October 20, 2024

1 Voting Classifier

A **Voting Classifier** is an ensemble learning technique that combines multiple machine learning models (often referred to as "base learners" or "weak learners") to make a single, more robust prediction. The idea is to leverage the strengths of different algorithms and reduce the risk of overfitting while improving the overall accuracy of predictions.

The Voting Classifier is a powerful ensemble method that improves prediction accuracy and robustness by aggregating multiple machine learning models. Its flexibility and effectiveness make it a valuable tool in many predictive modeling tasks.

There are two main types of voting:

- **Hard Voting**: The predicted class is the one that receives the majority of votes from the base models.
- **Soft Voting**: The predicted class is based on the average predicted probabilities of each class, and the class with the highest average probability is chosen.

1.0.1 When to Use Voting Classifier?

Voting Classifier is particularly useful when:

- You want to improve prediction accuracy by combining the strengths of different models.
- You have multiple algorithms that perform well individually, and you want to harness their collective performance.
- You want to reduce variance and increase robustness, especially in scenarios where different models may have different error patterns.

1.0.2 How Does Voting Classifier Work?

The Voting Classifier operates as follows:

1. **Model Selection**:
   - Choose a diverse set of models (e.g., Logistic Regression, Decision Trees, Support Vector Machines, etc.) that will serve as base learners.
2. **Training**:
   - Each model is trained independently on the same training dataset.
3. **Prediction**:

- For a new data point:
  - Hard Voting: Each model predicts a class label, and the class with the most votes is selected as the final prediction.
  - Soft Voting: Each model outputs the predicted probabilities for each class. The probabilities are averaged across all models, and the class with the highest average probability is selected.

4. Final Output:

- The Voting Classifier provides the final class label for the data point based on the voting mechanism.

1.0.3 Who Should Use Voting Classifier?

- **Data scientists and machine learning practitioners**: Who aim to build robust models by combining the strengths of multiple algorithms.
- **Kaggle competitors**: The Voting Classifier is a common strategy in competitions to boost accuracy by leveraging multiple models.
- **Industries requiring high accuracy**: Useful in fields like finance, healthcare, and marketing, where precise predictions are critical.

Advantages of Voting Classifier:

- **Improved accuracy**: Combining models often leads to better performance than any individual model.
- **Robustness**: Reduces the risk of overfitting and is less sensitive to noise in the data.
- **Flexibility**: Can incorporate different types of models, allowing the use of various algorithms tailored to specific aspects of the data.
- **Easy implementation**: Simple to implement using libraries like scikit-learn.

Disadvantages of Voting Classifier:

- **Complexity**: More complex than using a single model, as it requires managing multiple algorithms.
- **Computationally expensive**: Training multiple models can be time-consuming and resource-intensive, especially with large datasets.
- **Interpretability**: The ensemble nature makes it harder to interpret the model compared to a single algorithm.

1.0.4 Real-World Applications of Voting Classifier:

1. **Sentiment analysis**: Combining different models to classify sentiments from text data.
2. **Fraud detection**: In finance, where multiple algorithms can identify fraudulent patterns.
3. **Medical diagnosis**: Using various models to improve the accuracy of predicting patient outcomes.

4. **Customer churn prediction**: In marketing, to predict which customers are likely to stop using a service.

```
[34]: import pandas as pd

      from sklearn.datasets import make_classification
      from sklearn.model_selection import train_test_split, cross_val_score,␣
       ↪GridSearchCV
      from sklearn.naive_bayes import GaussianNB
      from sklearn.linear_model import LogisticRegression
      from sklearn.ensemble import RandomForestClassifier, VotingClassifier

      X, y = make_classification(n_samples=2500, n_features=15, n_informative=8,␣
       ↪n_redundant=2, random_state=11)
      X_train, X_test, Y_train, Y_test = train_test_split(X,y, test_size=0.2,␣
       ↪random_state=11)
```

```
[35]: gnb = GaussianNB()
      gnb.fit(X_train, Y_train)
```

```
[35]: GaussianNB()
```

```
[36]: cross_val_score(gnb,X_train, Y_train, cv=3).mean()
```

```
[36]: 0.7884931408169789
```

```
[37]: lr = LogisticRegression()
      lr.fit(X_train, Y_train)
```

```
[37]: LogisticRegression()
```

```
[38]: cross_val_score(lr,X_train, Y_train, cv=3).mean()
```

```
[38]: 0.7455041248144697
```

```
[39]: rfc = RandomForestClassifier()
      rfc.fit(X_train, Y_train)
```

```
[39]: RandomForestClassifier()
```

```
[40]: cross_val_score(rfc,X_train, Y_train, cv=3).mean()
```

```
[40]: 0.9044929487208347
```

```
[41]: vc = VotingClassifier([
          ('NaiveBayes', gnb),
          ('LogisticRegression', lr),
          ('RandomForestClassifier', rfc)
])
cross_val_score(vc,X_train, Y_train, cv=3).mean()
```

[41]:
```
0.8315016665841254
```

[42]:
```
param_grid = {
    'voting': ['hard', 'soft'],
    'weights': [(1,1,1), (2,1,1), (1,2,1), (1,1,2)]
}

vc2 = GridSearchCV(vc, param_grid, cv=5, n_jobs=-1)
vc2.fit(X_train, Y_train)
```

[42]:
```
GridSearchCV(cv=5,
             estimator=VotingClassifier(estimators=[('NaiveBayes',
                                                      GaussianNB()),
                                                     ('LogisticRegression',
                                                      LogisticRegression()),
                                                     ('RandomForestClassifier',
RandomForestClassifier())]),
             n_jobs=-1,
             param_grid={'voting': ['hard', 'soft'],
                         'weights': [(1, 1, 1), (2, 1, 1), (1, 2, 1),
                                     (1, 1, 2)]})
```

[43]:
```
vc2.best_params_
```

[43]:
```
{'voting': 'hard', 'weights': (1, 1, 2)}
```

[44]:
```
vc2.best_score_
```

[44]:
```
0.875
```

21-Linear_Regression

October 20, 2024

1 Linear Regression

Linear Regression is a statistical method used to model the relationship between a dependent variable (target) and one or more independent variables (features). It assumes that this relationship can be described by a straight line (in the case of one independent variable) or a hyperplane (in the case of multiple independent variables). The goal of linear regression is to predict the dependent variable by fitting the best line or plane to the given data points.

Linear Regression is one of the simplest and most widely used algorithms in machine learning. Its strengths lie in its interpretability and efficiency, making it a go-to choice for many regression tasks. However, it's important to ensure that the relationship between variables is linear, as the model can fail in more complex scenarios.

1.0.1 When to Use Linear Regression?

Linear Regression is commonly used when:

- You want to model and understand the linear relationship between one or more independent variables and a dependent variable.
- You are looking for a simple and interpretable model for regression tasks.
- The relationship between the variables is approximately linear (i.e., the change in the target is proportional to the change in the feature).
- You need a baseline model for regression tasks to compare with more complex algorithms.

1.0.2 How Does Linear Regression Work?

Linear Regression works by fitting a line (or hyperplane in the case of multiple variables) to minimize the difference between the predicted values and the actual target values. This difference is captured by the residual sum of squares (RSS):

1.0.3 Who Should Use Linear Regression?

- **Data scientists, statisticians, and analysts**: Who want to model and quantify relationships between variables.
- **Business analysts**: To predict trends like sales forecasting, market demand, etc.
- **Researchers**: In social sciences or economics, where interpretability is important and linear models are commonly used.

1.0.4 Advantages of Linear Regression:

- **Simplicity**: Easy to understand, implement, and interpret.
- **Efficiency**: Computationally fast, making it suitable for large datasets.
- **Interpretability**: The coefficients provide insights into how much each independent variable affects the dependent variable.
- **Baselines for comparison**: Often used as a baseline to compare with more complex models.

1.0.5 Disadvantages of Linear Regression:

- **Linearity assumption**: It assumes a linear relationship between the independent and dependent variables, which may not always hold true.
- **Sensitivity to outliers**: Linear regression can be easily influenced by outliers in the data.
- **Multicollinearity**: When independent variables are highly correlated, it can cause issues with coefficient estimation and model interpretation.
- **Underfitting**: Linear regression might not capture complex patterns in the data, leading to underfitting.

1.0.6 Real-World Applications of Linear Regression:

- **Predicting house prices**: Based on factors like square footage, number of rooms, and location.
- **Sales forecasting**: Using historical sales data and other relevant features to predict future sales.
- **Stock market prediction**: To estimate stock prices based on various financial indicators.
- **Healthcare**: Estimating patient outcomes like the length of hospital stay based on medical data.

[1]:
```python
import pandas as pd
import numpy as np
import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
  ↪r2_score

np.random.seed(42)
```

[2]:
```python
num_samples = 500
years_of_experience = np.random.randint(2,21, size=num_samples)
slope = (200_000 - 60_000) / 18
```

```
intercept = 60_000

salaries = slope * years_of_experience + intercept + np.random.normal(0,␣
↪10_000, size=num_samples)

data = {'Years_of_Experience': years_of_experience, 'Salary': salaries}
df = pd.DataFrame(data)

df
```

[2]:

```
     Years_of_Experience         Salary
0                      8  115037.780010
1                     16  182309.972927
2                     12  156442.408989
3                      9  144753.562169
4                      8  130798.818454
..                   …              …
495                   18  203824.097462
496                    8  123886.744304
497                   14  173813.401529
498                    5  101780.575328
499                    5  123441.890288

[500 rows x 2 columns]
```

[3]:

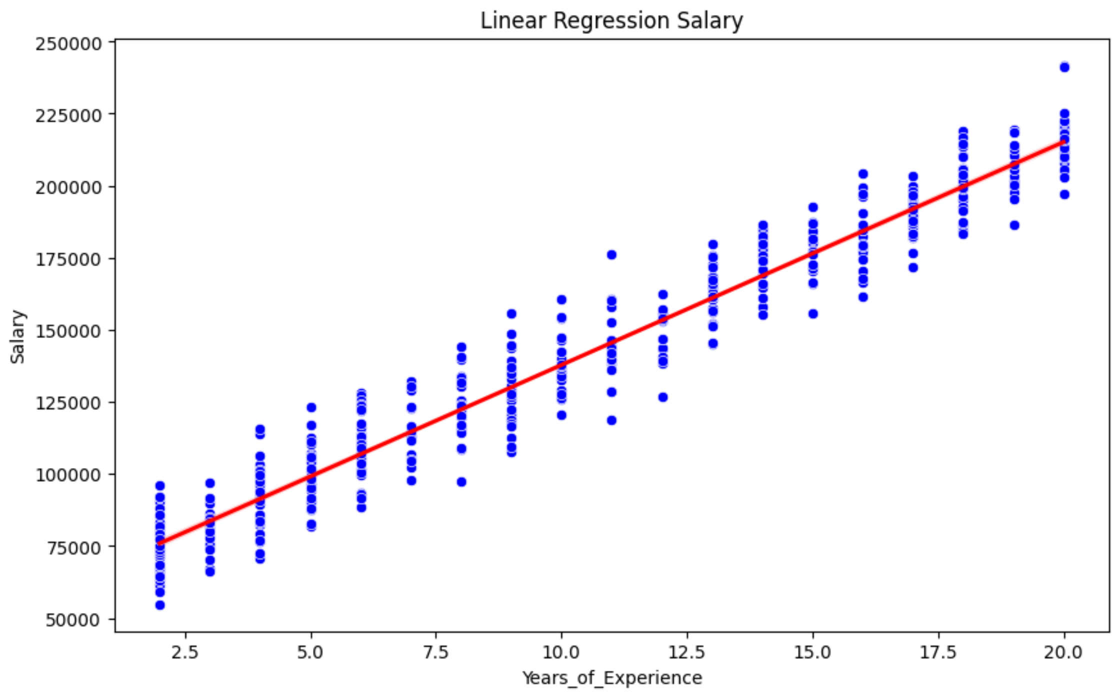
```
df.describe()
```

[3]:

```
       Years_of_Experience         Salary
count           500.000000     500.000000
mean             10.616000  142570.011096
std               5.662922   44935.263058
min               2.000000   54881.134555
25%               5.750000  104426.300731
50%              10.000000  139865.032545
75%              16.000000  182341.125962
max              20.000000  241879.376204
```

[4]:

```
plt.figure(figsize=(10,6))
sns.scatterplot(x="Years_of_Experience", y='Salary', data=df, color="blue")
sns.regplot(x="Years_of_Experience", y='Salary', data=df, scatter=False,␣
↪color='red')

plt.xlabel("Years_of_Experience")
plt.ylabel('Salary')
plt.title('Linear Regression Salary')
plt.show()
```



```
[5]: X = df[['Years_of_Experience']]
     y = df[['Salary']]

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=42)

     lr = LinearRegression()
     lr.fit(X_train, y_train)
```

```
[5]: LinearRegression()
```

```
[6]: lr.score(X_train, y_train)
```

```
[6]: 0.9488372177019488
```

```
[7]: lr.score(X_test, y_test)
```

```
[7]: 0.9573704756930143
```

```
[8]: y_pred = lr.predict(X_test)
     y_pred
```

```
[8]: array([[107141.01841859],
            [122534.2442647 ],
            [107141.01841859],
            [161017.30887996],
            [184107.14764911],
            [107141.01841859],
            [145624.08303385],
            [153320.69595691],
            [168713.92180301],
            [161017.30887996],
            [130230.85718775],
            [ 76354.56672638],
            [ 84051.17964944],
            [ 76354.56672638],
            [ 76354.56672638],
            [207196.98641827],
            [199500.37349522],
            [122534.2442647 ],
            [199500.37349522],
            [207196.98641827],
            [214893.59934132],
            [130230.85718775],
            [ 91747.79257249],
            [161017.30887996],
            [161017.30887996],
            [191803.76057217],
            [168713.92180301],
            [161017.30887996],
            [ 76354.56672638],
            [191803.76057217],
            [145624.08303385],
            [207196.98641827],
            [199500.37349522],
            [ 99444.40549554],
            [153320.69595691],
            [130230.85718775],
            [ 91747.79257249],
            [184107.14764911],
            [ 76354.56672638],
            [130230.85718775],
            [191803.76057217],
            [ 84051.17964944],
            [207196.98641827],
            [114837.63134164],
            [176410.53472606],
            [130230.85718775],
            [122534.2442647 ],
            [ 84051.17964944],
            [137927.4701108 ],
            [184107.14764911],
            [ 76354.56672638],
            [191803.76057217],
            [168713.92180301],
            [199500.37349522],
            [137927.4701108 ],
            [ 84051.17964944],
            [214893.59934132],
            [199500.37349522],
            [122534.2442647 ],
            [153320.69595691],
            [ 76354.56672638],
            [107141.01841859],
            [130230.85718775],
            [145624.08303385],
            [214893.59934132],
            [161017.30887996],
            [137927.4701108 ],
            [207196.98641827],
            [ 99444.40549554],
            [ 76354.56672638],
            [191803.76057217],
            [107141.01841859],
            [ 91747.79257249],
            [ 84051.17964944],
            [145624.08303385],
            [168713.92180301],
            [199500.37349522],
            [137927.4701108 ],
            [184107.14764911],
            [122534.2442647 ],
            [199500.37349522],
            [ 91747.79257249],
            [199500.37349522],
            [114837.63134164],
            [107141.01841859],
            [137927.4701108 ],
            [153320.69595691],
            [214893.59934132],
            [184107.14764911],
            [ 76354.56672638],
            [168713.92180301],
            [184107.14764911],
            [184107.14764911],
            [161017.30887996],
            [207196.98641827],
            [ 84051.17964944],
            [145624.08303385],
                   [ 76354.56672638],
                   [ 91747.79257249],
                   [ 76354.56672638]])
```

[9]:
```
mean_absolute_error(y_test, y_pred)
```

[9]:
```
7905.114728709234
```

[10]:
```
root_mean_squared_error(y_test, y_pred)
```

[10]:
```
9691.918147584149
```

[11]:
```
r2_score(y_test, y_pred)
```

[11]:
```
0.9573704756930143
```

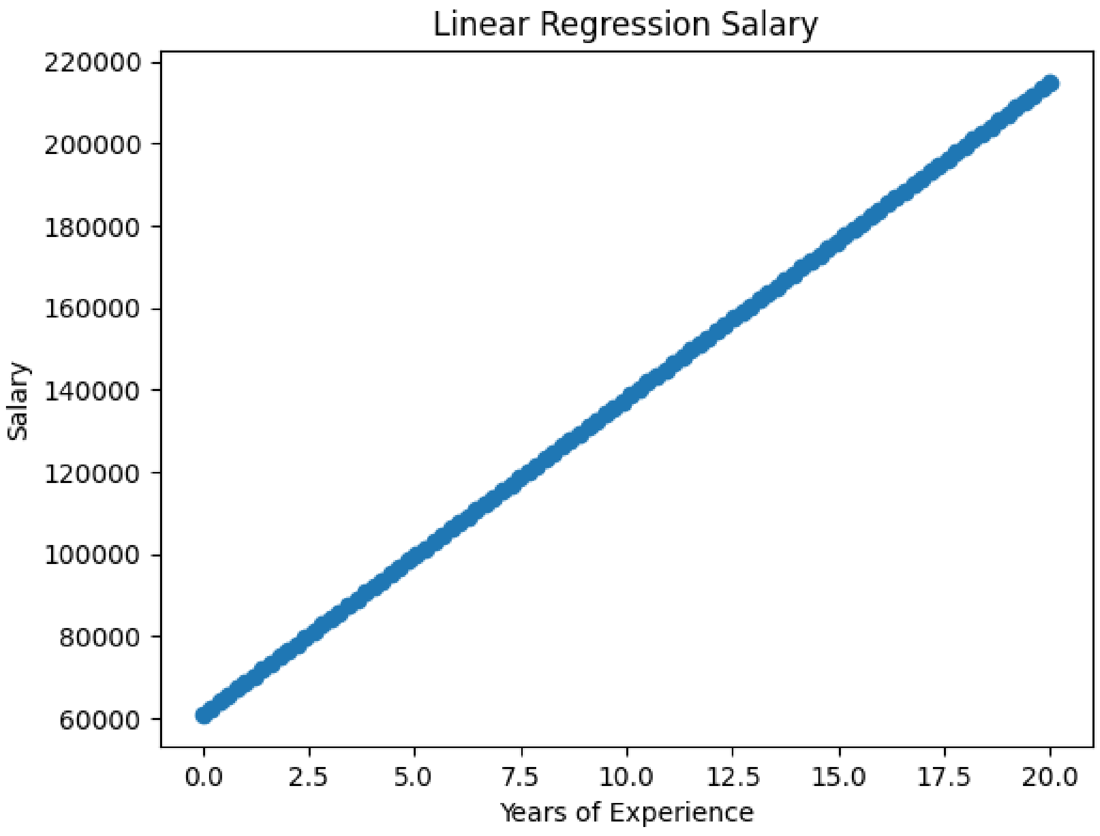
[12]:
```
lr.coef_
```

[12]:
```
array([[7696.61292305]])
```

[13]:
```
lr.intercept_
```

[13]:
```
array([60961.34088028])
```

[14]:
```
coefficients = lr.coef_
intercept = lr.intercept_
x = np.linspace(0, 20, 100)
y = coefficients*x + intercept

plt.scatter(x,y,label=f'y - {coefficients[0]} x + {intercept}')

plt.xlabel('Years of Experience')
plt.ylabel("Salary")
plt.title("Linear Regression Salary")

plt.show()
```


Linear Regression Salary
Salary
Years of Experience
220000
200000
180000
160000
140000
120000
100000
80000
60000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

22-Multiple_Linear_Regression

October 20, 2024

1 Multiple Linear Regression

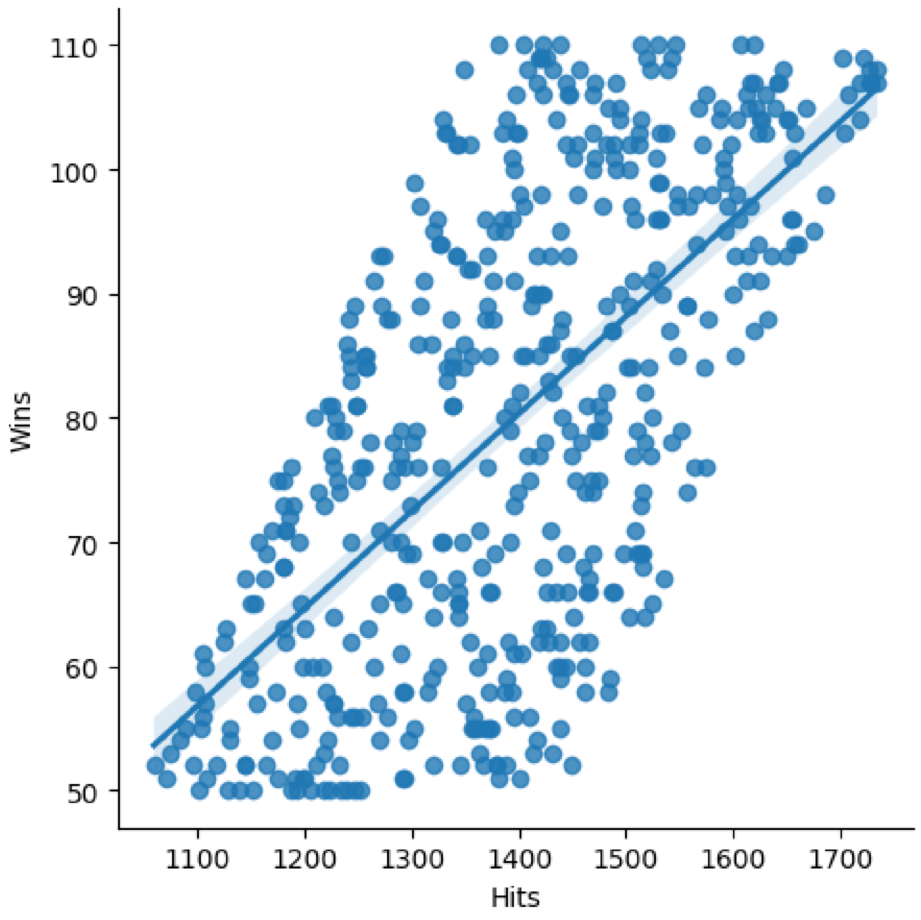
[1]:
```python
import random
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
 ↪r2_score

data = []

for _ in range(500):
    team_name = f"Team {chr(random.randint(65,90))}"
    season = random.randint(2010, 2023)
    wins = random.randint(50, 110)
    losses = 162 - wins
    hits = random.randint(1200,1600)
    doubles = random.randint(200,350)
    triples = random.randint(10,40)
    home_runs = random.randint(100,250)
    strikeouts = random.randint(1000,1500)

    hits_adjusted = hits + (wins-80) * 5
    doubles_adjusted = doubles + (wins-80) * 2
    triples_adjusted = triples + (wins-80) * 3
    home_runs_adjusted = home_runs + (wins-80) * 3
    strikeouts_adjusted = strikeouts - (wins - 80) * 10

    data.append([team_name, season, wins, losses, hits_adjusted,␣
 ↪doubles_adjusted, triples_adjusted, home_runs_adjusted, strikeouts_adjusted])
```

```
[2]: columns = ["Team", 'Season', 'Wins',"Losses", 'Hits', 'Doubles', "Tripples",␣
       ↪'HomeRuns', 'Strikesouts']

     df = pd.DataFrame(data, columns=columns)
     df
```

```
[2]:        Team  Season  Wins  Losses  Hits  Doubles  Tripples  HomeRuns  \
     0    Team R    2018    68      94  1516      197       -17       172
     1    Team Y    2023    65      97  1344      275        -6       153
     2    Team U    2017    62     100  1457      197       -26       152
     3    Team U    2011    62     100  1182      233       -28        68
     4    Team N    2016    75      87  1248      233         0       194
     ..      …       …    …      …    …       …        …
     495  Team D    2010    56     106  1106      152       -40       173
     496  Team O    2022   104      58  1587      310        93       307
     497  Team F    2019   106      56  1612      402       104       285
     498  Team Y    2010    64      98  1320      178       -30       126
     499  Team F    2019    55     107  1090      299       -41        87

          Strikesouts
     0           1582
     1           1495
     2           1454
     3           1271
     4           1058
     ..            …
     495         1341
     496         1130
     497         1224
     498         1204
     499         1573

     [500 rows x 9 columns]
```

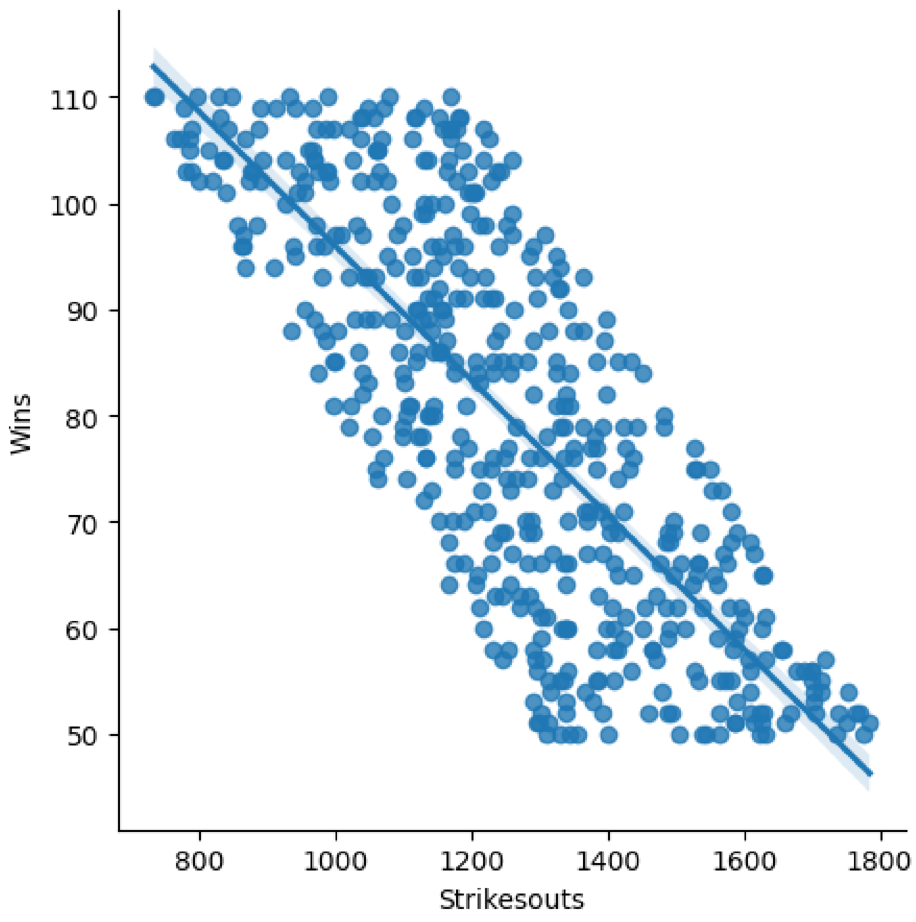
```
[3]: sns.lmplot(x="Hits", y="Wins", data=df)
```

```
[3]: <seaborn.axisgrid.FacetGrid at 0x143f5ca4bd0>
```



```
[4]: sns.lmplot(x="Strikesouts", y="Wins", data=df)
```

```
[4]: <seaborn.axisgrid.FacetGrid at 0x143c0b5b450>
```



[5]:
```
df2 = df.drop(columns=['Team', 'Season', 'Losses'], axis=1)
df2
```

[5]:
```
     Wins  Hits  Doubles  Tripples  HomeRuns  Strikesouts
0      68  1516      197       -17       172         1582
1      65  1344      275        -6       153         1495
2      62  1457      197       -26       152         1454
3      62  1182      233       -28        68         1271
4      75  1248      233         0       194         1058
..    ...   ...      ...       ...       ...          ...
495    56  1106      152       -40       173         1341
496   104  1587      310        93       307         1130
497   106  1612      402       104       285         1224
498    64  1320      178       -30       126         1204
499    55  1090      299       -41        87         1573

[500 rows x 6 columns]
```

```
[6]: df2.columns
```

```
[6]: Index(['Wins', 'Hits', 'Doubles', 'Tripples', 'HomeRuns', 'Strikesouts'],
     dtype='object')
```

```
[7]: X = df[['Hits', 'Doubles', 'Tripples', 'HomeRuns', 'Strikesouts']]
     y = df['Wins']

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=24)

     lr = LinearRegression()
     lr.fit(X_train, y_train)
```

```
[7]: LinearRegression()
```

```
[8]: lr.score(X_test, y_test)
```

```
[8]: 0.9766876951321011
```

```
[9]: lr.score(X_train, y_train)
```

```
[9]: 0.9779524643162276
```

```
[10]: y_pred = lr.predict(X_test)
      y_pred
```

```
[10]: array([ 79.43521812,  84.98752505,  50.06532199,  86.3822643 ,
              70.2017775 , 105.95777075, 112.95922339,  63.40753436,
              69.69225294,  61.45407411, 109.65817756,  91.24687093,
              53.51602938,  93.85145009,  52.19256777, 103.42618223,
              93.19739631,  49.75559028,  69.68435674,  66.0810152 ,
              55.59211481,  55.51175199,  83.75514524,  87.95757516,
              80.68253472,  89.01173975,  66.41065628,  51.2704802 ,
             114.21228759, 100.30212302,  88.33733575,  78.18776603,
              84.57104209,  55.42473415,  68.36036516,  77.72199593,
              63.81868048,  78.23857521,  95.88171008,  46.68880483,
             106.20061061,  99.65684269, 102.86836133,  76.59226093,
              97.43890582,  76.61837152,  67.35962044,  96.54023581,
              98.99463936,  56.07128436,  73.84535217,  66.34627102,
             100.5049504 ,  74.42990629,  66.88490465, 103.15534167,
              97.23129157,  64.7906051 ,  69.02154043,  67.47991565,
              62.22324859,  57.34789107, 103.81785868,  97.64835468,
              77.99833509,  98.76947005, 106.69044356,  56.70169242,
             111.37678841,  69.6311106 ,  99.53710032,  61.06668516,
             104.96695125,  90.96773068,  96.64482364,  49.51410132,
              67.4316584 ,  56.55871945, 101.21202627, 103.06747698,
              88.72028133, 100.99416768,  90.87960032,  95.11780101,
             105.01142639,  77.77341573,  57.13396359,  75.36364628,
              62.00493217,  66.91415242,  72.84128309,  97.89586341,
             109.90864271, 112.59300656,  76.58139883,  89.71476103,
              97.45697597,  80.89370933, 110.05559584,  56.32579865])
```

```
[11]: mean_absolute_error(y_test, y_pred)
```

```
[11]: 2.394583294287179
```

```
[12]: root_mean_squared_error(y_test, y_pred)
```

```
[12]: 2.7715279457881903
```

```
[13]: r2_score(y_test, y_pred)
```

```
[13]: 0.9766876951321011
```

```
[14]: lr.coef_
```

```
[14]: array([ 0.00362517,  0.00364042,  0.29551377,  0.01535465, -0.00312123])
```

```
[15]: lr.intercept_
```

```
[15]: 67.51826690267674
```

23-Lasso_Regression

October 20, 2024

1 Lasso Regression

Lasso (Least Absolute Shrinkage and Selection Operator) Regression is a type of linear regression that introduces a penalty equal to the absolute value of the magnitude of the coefficients. This penalty forces some of the coefficients to become exactly zero, effectively performing feature selection. It's a type of regularization technique used to prevent overfitting and simplify models by reducing the number of predictors.

1.0.1 Why is Lasso Regression Important?

- **Feature Selection**: One of the unique advantages of Lasso is that it can shrink some coefficients to zero, essentially performing automatic feature selection. This is especially useful when dealing with datasets that have many irrelevant or redundant features.
- **Prevents Overfitting**: In models with too many features, Lasso reduces overfitting by shrinking coefficients, improving the model's ability to generalize on unseen data.
- **Sparse Models**: By reducing coefficients to zero, Lasso simplifies the model, creating a sparse solution. Sparse models are easier to interpret and often have better prediction accuracy.
- **Works well with high-dimensional data**: In cases where the number of predictors exceeds the number of observations, Lasso is particularly useful because it selects only the most important predictors.

1.0.2 How does Lasso Regression work?

The Lasso regression process follows these steps:

1. Initialization: Start by defining a linear regression model with a penalty on the magnitude of the coefficients. This penalty is controlled by a hyperparameter .

2. Regularization: Introduce the L1 penalty, which is the sum of the absolute values of the regression coefficients. Unlike traditional linear regression, which only minimizes the sum of squared errors, Lasso also minimizes the total size of the coefficients.

3. Optimization: Use optimization algorithms like coordinate descent to solve the Lasso objective function. The solver adjusts the coefficients to find the balance between fitting the data and shrinking some coefficients to zero.

4. Coefficient Shrinkage: Depending on the value of , some of the coefficients may be shrunk to exactly zero. The larger the value of , the stronger the regularization effect, meaning more features will be set to zero.

5.Prediction: After training, the resulting model can be used to predict outcomes based on the reduced set of features.

1.0.3 When should you use Lasso Regression?

- **High-dimensional datasets**: When you have more features than observations, Lasso is a good choice because it selects a subset of the most important features, reducing the complexity of the model.
- **Feature Selection**: If you suspect that many of the features in your dataset are irrelevant, Lasso can help automatically select the relevant ones.
- **Overfitting prevention**: When you notice that your model is performing well on training data but poorly on validation or test data, Lasso can be a good solution as it reduces the chance of overfitting by regularizing the coefficients.
- **Sparse solutions**: If you need an interpretable model with fewer predictors, Lasso is suitable because it produces sparse models where some coefficients are zero.

1.0.4 Who uses Lasso Regression?

- **Data Scientists and Machine Learning Engineers**: Especially when working on predictive modeling tasks that involve many features, Lasso is a popular tool for improving model generalizability and interpretability.
- **Researchers**: In fields like genomics, where the number of predictors can be vast, Lasso is frequently used to select important variables while disregarding the rest.
- **Statisticians**: Those dealing with linear regression models that have multicollinearity (high correlation between features) use Lasso to shrink the coefficients and combat the effects of collinearity.

1.0.5 Key Points to Remember:

- The strength of the regularization is controlled by the hyperparameter   (or alpha in Python's scikit-learn). The larger the value, the stronger the regularization.
- It's a biased estimator, meaning it sacrifices some bias to reduce variance, leading to a better performance on test data (especially when overfitting is a concern).
- Lasso selects features and shrinks coefficients, making it a good choice for interpretable models.

[1]:
```
import pandas as pd
import numpy as np

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
 ↪r2_score
```

```
ca_housing = fetch_california_housing()

X = ca_housing.data
y = ca_housing.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=19)

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

```
[2]: lasso = Lasso()
     lasso.fit(X_train, y_train)
```

```
[2]: Lasso()
```

```
[3]: y_pred = lasso.predict(X_test)
     y_pred
```

```
[3]: array([2.06708162, 2.06708162, 2.06708162, …, 2.06708162, 2.06708162,
            2.06708162])
```

```
[4]: mean_absolute_error(y_test, y_pred)
```

```
[4]: 0.9119430573559844
```

```
[5]: root_mean_squared_error(y_test, y_pred)
```

```
[5]: 1.1517022831067947
```

```
[6]: r2_score(y_test, y_pred)
```

```
[6]: -4.109353628090062e-05
```

```
[7]: param_grid = {
         'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
     }

     lasso_cv = GridSearchCV(lasso, param_grid, cv=3, n_jobs=-1)
     lasso_cv.fit(X_train, y_train)
```

```
[7]: GridSearchCV(cv=3, estimator=Lasso(), n_jobs=-1,
                  param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]})
```

```
[8]: y_pred2 = lasso_cv.predict(X_test)
     y_pred2

[8]: array([2.72485002, 0.66011565, 2.12095357, …, 1.92427048, 1.88073015,
            2.3150062 ])

[9]: mean_absolute_error(y_test, y_pred2)

[9]: 0.5353500594399226

[10]: root_mean_squared_error(y_test, y_pred2)

[10]: 0.7218861196454859

[11]: r2_score(y_test, y_pred2)

[11]: 0.60710656378633

[12]: lasso_cv.best_estimator_

[12]: Lasso(alpha=0.001)

[13]: lasso3 = Lasso(alpha=0.001)
      lasso3.fit(X_train, y_train)

[13]: Lasso(alpha=0.001)

[14]: lasso3.intercept_

[14]: 2.0670816194279977

[15]: lasso3.coef_

[15]: array([ 0.83673788,  0.12126534, -0.26089701,  0.30370697, -0.00173652,
             -0.02849403, -0.8865986 , -0.86020295])

[16]: feature_names =␣
       ↪['MedInc','HouseAge','AveRooms','AveBedrms','Population','AveOccup','Latitude','Longitude']

      df = pd.DataFrame({'Feature_Names': feature_names, 'Coef': lasso3.coef_})
      df

[16]:   Feature_Names      Coef
      0        MedInc  0.836738
      1      HouseAge  0.121265
      2      AveRooms -0.260897
      3     AveBedrms  0.303707
      4    Population -0.001737
```

```
5      AveOccup -0.028494
6      Latitude -0.886599
7     Longitude -0.860203
```

24-Ridge_Regression

October 20, 2024

1 Ridge Regression

Ridge Regression is a type of linear regression that adds a regularization term to the model to penalize large coefficients. Unlike ordinary linear regression, which only minimizes the sum of squared errors, Ridge Regression minimizes both the squared errors and a penalty term proportional to the square of the magnitude of the coefficients. This penalty helps reduce model complexity and prevent overfitting.

In Ridge Regression, the penalty term is based on the L2 norm (the square of the coefficients), which encourages small but non-zero coefficients. Unlike Lasso, Ridge does not shrink coefficients to zero; rather, it shrinks them towards zero to reduce variance.

1.0.1 Why is Ridge Regression Important?

- **Multicollinearity Solution**: Ridge Regression is particularly useful when there is multicollinearity (high correlation between independent variables). In such cases, ordinary least squares (OLS) can give unreliable estimates. Ridge stabilizes the regression by adding a penalty that discourages large coefficient values.
- **Overfitting Prevention**: By introducing a regularization term, Ridge Regression controls overfitting, ensuring the model generalizes better to unseen data.
- **Handling High-Dimensional Data**: Ridge works well when the number of features (predictors) is large, especially when the number of features exceeds the number of observations.
- **Interpretability**: While Ridge does not perform feature selection like Lasso, it can still simplify models by shrinking coefficients to smaller values, making the model more interpretable.

1.0.2 How does Ridge Regression work?

- **Define the objective**: Ridge minimizes the sum of squared errors while adding an L2 penalty (the sum of squared coefficients).
- **Choose regularization parameter ( )**: The controls the amount of regularization. If =0, Ridge behaves like standard linear regression. As increases, the regularization effect becomes stronger, shrinking the coefficients.
- **Fit the model**: Solving the Ridge Regression objective involves minimizing the cost function, which can be efficiently done using optimization algorithms like gradient descent or closed-form solutions. For Ridge, there is an exact solution:

- **Shrinkage of coefficients**: As increases, the model becomes more conservative, shrinking the coefficients to smaller values, but never exactly zero as in Lasso.
- **Prediction**: Once the coefficients are trained, the model can predict outcomes for new data points using the regularized coefficients.

1.0.3 When should you use Ridge Regression?

- **Multicollinearity**: Ridge is ideal when independent variables are highly correlated. It helps by shrinking coefficients, which reduces the variance that can result from multicollinearity.
- **High-dimensional data**: When the dataset has many predictors (especially when the number of predictors exceeds the number of observations), Ridge is more stable than ordinary least squares regression.
- **When all features are important**: If you believe all the features contribute to the target variable (even to a small degree) and you don't want to discard any, Ridge is preferred over Lasso, which might zero out some coefficients.
- **Balancing bias and variance**: Ridge strikes a balance between variance (how much your model changes with different training data) and bias (error introduced by approximating a complex model). It controls variance by shrinking coefficients without introducing significant bias.

1.0.4 Who uses Ridge Regression?

- **Data Scientists and Machine Learning Engineers**: Ridge is a go-to algorithm when building predictive models that need to handle multicollinearity or high-dimensional datasets.
- **Economists and Statisticians**: In fields where datasets often have correlated variables (like economic indicators), Ridge is commonly used to produce more stable and interpretable models.
- **Genomics and Bioinformatics**: Ridge is useful in genomics, where datasets often have a large number of highly correlated predictors (genes), and overfitting is a concern.

1.0.5 Key Differences Between Ridge and Lasso:

- Ridge shrinks coefficients, but it does not set them to exactly zero, meaning all features remain in the model (though their contributions may be small).
- Lasso, on the other hand, can shrink some coefficients to zero, effectively removing features from the model.

1.0.6 Key Points to Remember:

- **Regularization**: The key feature of Ridge is that it adds an L2 penalty (squared coefficients) to the loss function. This penalty reduces the model's complexity by shrinking the coefficients.
- **Bias-Variance Tradeoff**: Ridge Regression helps reduce variance (thus improving generalization) at the cost of introducing a small bias. No Feature Selection: Unlike Lasso, Ridge does not perform feature selection. All features are kept in the model but are shrunk towards zero.

```
[1]: import pandas as pd
     import numpy as np

     from sklearn.model_selection import train_test_split, GridSearchCV
     from sklearn.linear_model import Ridge
     from sklearn.datasets import make_regression
     from sklearn.preprocessing import StandardScaler
     from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
      ↪r2_score

     X, y = make_regression(n_samples=100, n_features=4, noise=1, random_state=42 ,␣
      ↪effective_rank=2)

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=19)

     scaler = StandardScaler()
     X_train = scaler.fit_transform(X_train)
     X_test = scaler.fit_transform(X_test)
```

```
[2]: ridge = Ridge()
     ridge.fit(X_train, y_train)
```

```
[2]: Ridge()
```

```
[3]: y_pred = ridge.predict(X_test)
     y_pred
```

```
[3]: array([  7.13539375,  -8.8002492 ,   8.78912513,   6.09927205,
              5.99994735, -12.30572261, -10.03631065,  -1.04649374,
             23.52913328,   2.55203747,  11.48889191,  -7.61522827,
              0.46017582,  -8.62564635, -29.50846326,  -7.76254477,
              8.06258548,  -7.44014532,  -3.84252421,  25.8910913 ])
```

```
[4]: mean_absolute_error(y_test, y_pred)
```

```
[4]: 3.0816824025813956
```

```
[5]: root_mean_squared_error(y_test, y_pred)
```

```
[5]: 11.330220034384492
```

```
[6]: r2_score(y_test, y_pred)
```

```
[6]: 0.9276234847540757
```

```
[7]: param_grid = {
         'alpha': [ 0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
     }

     ridge_cv = GridSearchCV(ridge, param_grid, cv=3, n_jobs=-1)
     ridge_cv.fit(X_train, y_train)
```

```
[7]: GridSearchCV(cv=3, estimator=Ridge(), n_jobs=-1,
                  param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0]})
```

```
[8]: y_pred2 = ridge_cv.predict(X_test)
     y_pred2
```

```
[8]: array([  7.17119326,  -8.89540639,   8.83622445,   6.17650173,
              6.08043617, -12.40702852, -10.10162896,  -1.07770549,
             23.78518149,   2.57122983,  11.59362725,  -7.71459096,
              0.4722029 ,  -8.69228577, -29.79550622,  -7.85527431,
              8.12657042,  -7.53080381,  -3.87581506,  26.15720314])
```

```
[9]: mean_absolute_error(y_test, y_pred2)
```

```
[9]: 3.0816824025813956
```

```
[10]: root_mean_squared_error(y_test, y_pred2)
```

```
[10]: 11.349061639991897
```

```
[11]: r2_score(y_test, y_pred2)
```

```
[11]: 0.9275031261245554
```

```
[12]: ridge_cv.best_estimator_
```

```
[12]: Ridge(alpha=0.0001)
```

```
[13]: ridge_cv.best_estimator_.intercept_
```

```
[13]: 0.15121625800403157
```

```
[14]: ridge_cv.best_estimator_.coef_
```

```
[14]: array([4.3844567 , 5.0806461 , 7.6664195 , 2.89485579])
```

25-Elastic_Net_Regression

October 20, 2024

1 Elastic Net Regression

Elastic Net Regression is a regularization technique that combines both Lasso Regression and Ridge Regression. It introduces penalties to the model, balancing the L1 norm (from Lasso) and the L2 norm (from Ridge) to create a more flexible regularization method. This helps improve predictive accuracy, deal with multicollinearity, and perform automatic feature selection.

Elastic Net allows for the benefits of both Lasso and Ridge:

- **L1 penalty (Lasso)** encourages sparse solutions by shrinking some coefficients to exactly zero, thus selecting features.
- **L2 penalty (Ridge)** shrinks coefficients but keeps them non-zero, thus stabilizing the model when predictors are highly correlated.

1.0.1 Why is Elastic Net Regression Important?

- **Combining Lasso and Ridge**: In situations where Lasso struggles (e.g., highly correlated features), Ridge can compensate. Elastic Net creates a balance between the two, offering a robust model that benefits from both feature selection and coefficient shrinkage.
- **Handles Multicollinearity**: Elastic Net is particularly useful when there are correlated predictors. It keeps Ridge's ability to handle multicollinearity while also keeping Lasso's feature selection capability.
- **Feature Selection**: Like Lasso, Elastic Net can shrink coefficients to zero, effectively eliminating irrelevant features from the model, which makes it interpretable and prevents overfitting.
- **Flexible Regularization**: It introduces two penalties (L1 and L2), giving you the flexibility to fine-tune the degree of regularization needed for the data, based on the dataset's complexity and feature correlation.

1.0.2 How does Elastic Net Regression work?

1. Initialization: Start with the linear regression objective but add two penalty terms: one for Lasso (L1 norm) and one for Ridge (L2 norm).

2. Blending penalties: The model combines the advantages of both regularization techniques. The L1 penalty encourages sparsity by setting some coefficients to zero, while the L2 penalty helps when multicollinearity exists by shrinking the coefficients but keeping them non-zero.

3. Hyperparameter tuning: Elastic Net has two hyperparameters, (Lasso penalty strength) and (Ridge penalty strength). These are often tuned using cross-validation to find the optimal values that prevent overfitting while improving model performance.

4. Optimization: Elastic Net uses algorithms like coordinate descent to solve the objective function, iteratively adjusting the coefficients until an optimal solution is reached.

5. Prediction: After training, the model can predict outcomes based on the selected features and shrunk coefficients.

1.0.3 When should you use Elastic Net Regression?

- **When features are highly correlated**: Lasso struggles with groups of highly correlated features by selecting only one and ignoring the others. Elastic Net, by blending Ridge, can distribute the effect across correlated features.
- **High-dimensional data**: In datasets where the number of features is much larger than the number of observations (e.g., in genomics or text classification), Elastic Net is helpful because it performs both feature selection (like Lasso) and shrinks the coefficients (like Ridge).
- **Sparse models**: If you expect only a subset of your predictors to be relevant, Elastic Net can shrink irrelevant predictors to zero, like Lasso.
- **Handling multicollinearity**: When predictors are highly correlated, Elastic Net is better suited than Lasso alone, since Ridge helps stabilize coefficient estimates in the presence of collinearity.

1.0.4 Who uses Elastic Net Regression?

- **Data Scientists and Machine Learning Practitioners**: Elastic Net is a go-to choice when modeling datasets with many features or when there is multicollinearity, especially in high-stakes predictive tasks.
- **Bioinformaticians and Geneticists**: In fields like genomics, where there are many correlated features (genes), Elastic Net helps in selecting key predictors while accounting for correlations.
- **Economists and Statisticians**: Elastic Net is useful in domains where multicollinearity and high-dimensional datasets are common, as it balances the trade-off between bias and variance.

1.0.5 Key Points to Remember:

- **L1 and L2 regularization**: Elastic Net combines L1 (Lasso) and L2 (Ridge) regularization, balancing feature selection and coefficient shrinkage.
- **Tuning and l1_ratio**: The mix between Lasso and Ridge penalties is controlled by the l1_ratio parameter. An value helps control the overall strength of the regularization.
- **No perfect separation**: While Lasso is more aggressive in shrinking coefficients to zero, Elastic Net allows coefficients to stay small but non-zero when dealing with correlated features.

1.0.6 Differences Between Elastic Net, Lasso, and Ridge:

- **Lasso (only L1 penalty)**: Performs automatic feature selection by shrinking some coefficients to zero, but struggles with correlated features.
- **Ridge (only L2 penalty)**: Shrinks coefficients uniformly without zeroing them out, handling multicollinearity but not performing feature selection.
- **Elastic Net**: Balances the two by performing feature selection like Lasso but also shrinking correlated features together like Ridge.

[1]:
```
import pandas as pd
import seaborn as sns

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
 ↪r2_score

tips = sns.load_dataset('tips')
tips
```

[1]:
```
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          …     …      …     …    …      …   …
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

[2]:
```
tips = pd.get_dummies(tips)
tips
```

[2]:
```
     total_bill   tip  size  sex_Male  sex_Female  smoker_Yes  smoker_No  \
0         16.99  1.01     2     False        True       False       True
1         10.34  1.66     3      True       False       False       True
2         21.01  3.50     3      True       False       False       True
3         23.68  3.31     2      True       False       False       True
4         24.59  3.61     4     False        True       False       True
..          …     …     …        …          …          …         …
239       29.03  5.92     3      True       False       False       True
```

```
240        27.18  2.00     2     False        True        True       False
241        22.67  2.00     2      True       False        True       False
242        17.82  1.75     2      True       False       False        True
243        18.78  3.00     2     False        True       False        True

     day_Thur  day_Fri  day_Sat  day_Sun  time_Lunch  time_Dinner
0       False    False    False     True       False         True
1       False    False    False     True       False         True
2       False    False    False     True       False         True
3       False    False    False     True       False         True
4       False    False    False     True       False         True
..        …        …       …       …          …          …
239     False    False     True    False       False         True
240     False    False     True    False       False         True
241     False    False     True    False       False         True
242     False    False     True    False       False         True
243      True    False    False    False       False         True

[244 rows x 13 columns]
```

[3]:
```
X = tips.drop('tip', axis=1)
y = tips['tip']

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2,␣
 ↪random_state=19)

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

[4]:
```
elastic_net = ElasticNet()
elastic_net.fit(X_train, y_train)
```

[4]:
```
ElasticNet()
```

[5]:
```
y_pred = elastic_net.predict(X_test)
y_pred
```

[5]:
```
array([3.22787603, 2.75553242, 2.98583549, 2.93657376, 2.81653161,
       2.85814622, 3.02869499, 3.33226823, 2.85601214, 2.97872189,
       2.80728392, 3.14873713, 3.52149022, 2.91701133, 2.92928231,
       2.84907637, 2.89904947, 2.80177087, 2.74681825, 3.23410043,
       3.47738585, 2.85654566, 2.79696919, 2.72014221, 2.85138829,
       2.94262032, 2.94564361, 2.86348143, 2.97498724, 2.942976  ,
       3.1030322 , 2.94422088, 2.80159303, 2.79732487, 2.91896757,
       2.95062313, 3.06248463, 2.93408399, 2.79696919, 2.93070503,
       2.89691539, 2.80159303, 3.20173352, 2.76922611, 2.90313979,
       3.01784673, 2.94937825, 2.86810527, 3.47454041])
```

[6]:
```
mean_absolute_error(y_test, y_pred)
```

[6]:
```
1.214329254741493
```

[7]:
```
root_mean_squared_error(y_test, y_pred)
```

[7]:
```
2.899097972303773
```

[8]:
```
r2_score(y_test, y_pred)
```

[8]:
```
0.14011647876429623
```

[9]:
```
param_grid = {
    'alpha': [ 0.1,0.3,0.5,0.7,0.9,1.0 ],
    'l1_ratio': [ 0.1,0.3,0.5,0.7,0.9,1.0 ]
}

elastic_net_cv = GridSearchCV(elastic_net, param_grid, cv=3,␣
 ↪scoring='neg_root_mean_squared_error', n_jobs=-1)
elastic_net_cv.fit(X_train, y_train)
```

[9]:
```
GridSearchCV(cv=3, estimator=ElasticNet(), n_jobs=-1,
             param_grid={'alpha': [0.1, 0.3, 0.5, 0.7, 0.9, 1.0],
                         'l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9, 1.0]},
             scoring='neg_root_mean_squared_error')
```

[10]:
```
y_pred2 = elastic_net_cv.predict(X_test)
y_pred2
```

[10]:
```
array([4.36045835, 2.01577541, 3.49350917, 2.6632068 , 2.43904349,
       2.39559597, 3.21389761, 4.37446046, 2.65737098, 2.7356982 ,
       2.40921402, 4.13318475, 4.6636574 , 2.67645063, 2.83025002,
       2.48055813, 2.47870584, 2.54539045, 2.12747365, 4.0536548 ,
       4.76205469, 2.48556614, 2.21289653, 2.15979944, 2.52823827,
       2.84575684, 3.18666694, 2.56724603, 2.9643126 , 2.73268614,
       3.28063625, 3.04227102, 2.65903479, 2.40876079, 2.89783283,
       2.87108944, 3.6777688 , 2.65517579, 2.19874392, 2.82147533,
       2.61162889, 2.39085742, 3.45504734, 2.28645428, 2.73256056,
       3.06932308, 3.04475407, 2.58216077, 5.2470807 ])
```

[11]:
```
mean_absolute_error(y_test, y_pred2)
```

[11]:
```
0.9467399418855579
```

[12]:
```
root_mean_squared_error(y_test, y_pred2)
```

```
[12]: 1.9034103248341647
```

```
[13]: r2_score(y_test, y_pred2)
```

```
[13]: 0.43544123444224914
```

```
[14]: elastic_net_cv.best_params_
```

```
[14]: {'alpha': 0.1, 'l1_ratio': 0.1}
```

```
[15]: elastic_net_cv.best_estimator_
```

```
[15]: ElasticNet(alpha=0.1, l1_ratio=0.1)
```

```
[16]: elastic_net_cv.best_score_
```

```
[16]: -0.9574652456524534
```

26-Stacking_Regressor

October 20, 2024

1 Stacking Regression

A **Stacking Regressor** is an ensemble machine learning technique that combines the predictions of multiple regression models (base models) by training a meta-model (stacker) to make final predictions based on the outputs of those base models. Stacking, also known as stacked generalization, aims to leverage the strengths of several models to improve predictive performance compared to using any single model.

The key idea of stacking is that different models may excel in capturing different aspects of the data. By training a second-level model (meta-model) on the outputs of base models, stacking allows the meta-model to learn how to best combine the predictions of the base models.

1.0.1 Why is Stacking Regressor Important?

- **Boosts model performance**: Since stacking combines the predictive power of several models, it often yields better performance than individual models. It uses the strengths of each model while compensating for their weaknesses.
- **Improves generalization**: Stacking helps in reducing overfitting by leveraging a blend of multiple models. The meta-model is trained to understand which base models perform best under different circumstances.
- **Flexibility**: You can use a wide range of base learners (e.g., decision trees, linear regression, support vector machines, etc.) and combine their predictions in a meaningful way, allowing for diverse model architectures.

1.0.2 How does Stacking Regressor work?

Stacking involves two layers:

1. **Base Models (Level 0)**: These are the individual regression models that make predictions on the dataset. Each base model is trained on the original features of the data.
2. **Meta-model (Level 1)**: The meta-model (also called the blender or stacker) is trained on the predictions of the base models. It learns to combine these predictions to generate the final output.

Here's a step-by-step breakdown of how stacking works:

1. **Train base models**: Each base model is trained on the training data. These models could be decision trees, random forests, gradient boosting, linear regression, etc.

2. **Generate base model predictions**: Once trained, the base models are used to make predictions on both the training data (for cross-validation) and the test data. These predictions are then used as inputs to the meta-model.
3. **Train the meta-model**: A second model (meta-model), often a simple linear regression or another powerful algorithm, is trained using the predictions from the base models as features. This model learns to combine the outputs of the base models to improve predictive performance.
4. **Final prediction**: During the prediction phase, the base models make predictions on new data, and these predictions are fed into the meta-model, which then produces the final prediction.

1.0.3 When should you use Stacking Regressor?

- **When base models have different strengths**: If your base models have different inductive biases and capture different patterns in the data (e.g., decision trees might capture non-linear relationships, while linear models capture linear trends), stacking can help by learning how to best combine them.
- **High complexity datasets**: Stacking can be useful when the dataset is complex and no single model is able to fully capture the data's underlying patterns. By combining multiple models, stacking often performs better in these cases.
- **To reduce overfitting**: Stacking helps in preventing overfitting by blending multiple models. If individual models tend to overfit, the meta-model can act as a stabilizing factor by learning to trust the better-performing models.
- **When you want to combine models with different architectures**: Stacking gives the flexibility to combine a variety of different models (e.g., tree-based models, linear models, neural networks, etc.) in one powerful ensemble.

1.0.4 Who uses Stacking Regressor?

- **Data Scientists and Machine Learning Engineers**: Stacking is a common technique used in competitions like Kaggle and real-world applications where predictive performance is critical. It's widely adopted when optimizing models for predictive accuracy.
- **Financial Analysts and Economists**: Stacking is used in scenarios such as stock market predictions or economic forecasting, where combining models that capture different signals (e.g., time-series patterns, economic indicators, etc.) can enhance the overall predictive power.
- **Researchers and AI Developers**: Stacking is often used in research when building complex models that need to integrate multiple types of predictions (e.g., biological data analysis, medical imaging, etc.).

1.0.5 Key Points to Remember:

- **Base Models and Meta-model**: The base models can be diverse (linear models, tree-based models, SVM, etc.), and the meta-model is trained on their predictions to combine them effectively.

- **Cross-Validation**: Stacking usually involves cross-validation to avoid overfitting. The meta-model should be trained on out-of-sample predictions to ensure it generalizes well.
- **Customizability**: You can choose any combination of models for the base learners and meta-model depending on the problem and data structure.

1.0.6 How does Stacking differ from Bagging and Boosting?

- **Bagging**: Bagging (e.g., Random Forest) trains multiple models in parallel on different subsets of the data and averages their predictions to reduce variance.
- **Boosting**: Boosting (e.g., Gradient Boosting) trains models sequentially, where each subsequent model attempts to correct the mistakes of the previous one.
- **Stacking**: Stacking trains multiple models in parallel (like Bagging) but uses another model to learn how to best combine the outputs of the base models.
- Lasso Regression captures important features by shrinking irrelevant ones.
- Ridge Regression handles multicollinearity and smooths out feature effects.
- Support Vector Regression captures complex, non-linear relationships.

[1]:
```
import pandas as pd
import seaborn as sns

from sklearn.model_selection import train_test_split, GridSearchCV,␣
 ↪RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import StackingRegressor, RandomForestRegressor,␣
 ↪GradientBoostingRegressor, VotingRegressor
from sklearn.linear_model import Ridge, Lasso, LinearRegression
from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
 ↪r2_score
from sklearn.svm import SVR

mpg = sns.load_dataset('mpg')
mpg
```

[1]:
```
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
0    18.0          8         307.0       130.0    3504          12.0
1    15.0          8         350.0       165.0    3693          11.5
2    18.0          8         318.0       150.0    3436          11.0
3    16.0          8         304.0       150.0    3433          12.0
4    17.0          8         302.0       140.0    3449          10.5
..    …          …            …          …       …             …
393  27.0          4         140.0        86.0    2790          15.6
394  44.0          4          97.0        52.0    2130          24.6
395  32.0          4         135.0        84.0    2295          11.6
396  28.0          4         120.0        79.0    2625          18.6
397  31.0          4         119.0        82.0    2720          19.4
```

```
     model_year  origin                       name
0            70     usa  chevrolet chevelle malibu
1            70     usa          buick skylark 320
2            70     usa         plymouth satellite
3            70     usa              amc rebel sst
4            70     usa                ford torino
..          …       …                        …
393          82     usa            ford mustang gl
394          82  europe                  vw pickup
395          82     usa              dodge rampage
396          82     usa                ford ranger
397          82     usa                 chevy s-10

[398 rows x 9 columns]
```

[2]:
```
mpg = mpg.drop('name', axis=1)
mpg = pd.get_dummies(mpg)
mpg
```

[2]:
```
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
0    18.0          8         307.0       130.0    3504          12.0
1    15.0          8         350.0       165.0    3693          11.5
2    18.0          8         318.0       150.0    3436          11.0
3    16.0          8         304.0       150.0    3433          12.0
4    17.0          8         302.0       140.0    3449          10.5
..    …        …            …          …       …           …
393  27.0          4         140.0        86.0    2790          15.6
394  44.0          4          97.0        52.0    2130          24.6
395  32.0          4         135.0        84.0    2295          11.6
396  28.0          4         120.0        79.0    2625          18.6
397  31.0          4         119.0        82.0    2720          19.4

     model_year  origin_europe  origin_japan  origin_usa
0            70          False         False        True
1            70          False         False        True
2            70          False         False        True
3            70          False         False        True
4            70          False         False        True
..          …            …           …          …
393          82          False         False        True
394          82           True         False       False
395          82          False         False        True
396          82          False         False        True
397          82          False         False        True

[398 rows x 10 columns]
```

```
[3]: pd.DataFrame(mpg.isnull().sum().sort_values(ascending=False))
```

```
[3]:                0
     horsepower     6
     mpg            0
     cylinders      0
     displacement   0
     weight         0
     acceleration   0
     model_year     0
     origin_europe  0
     origin_japan   0
     origin_usa     0
```

```
[4]: mpg['horsepower'].fillna(mpg['horsepower'].mean(), inplace=True)
```

```
C:\Users\ikiga\AppData\Local\Temp\ipykernel_26612\370553803.py:1: FutureWarning:
A value is trying to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  mpg['horsepower'].fillna(mpg['horsepower'].mean(), inplace=True)
```

```
[5]: mpg
```

```
[5]:       mpg  cylinders  displacement  horsepower  weight  acceleration  \
     0    18.0          8         307.0       130.0    3504          12.0
     1    15.0          8         350.0       165.0    3693          11.5
     2    18.0          8         318.0       150.0    3436          11.0
     3    16.0          8         304.0       150.0    3433          12.0
     4    17.0          8         302.0       140.0    3449          10.5
     ..    …         …            …          …      …           …
     393  27.0          4         140.0        86.0    2790          15.6
     394  44.0          4          97.0        52.0    2130          24.6
     395  32.0          4         135.0        84.0    2295          11.6
     396  28.0          4         120.0        79.0    2625          18.6
     397  31.0          4         119.0        82.0    2720          19.4

          model_year  origin_europe  origin_japan  origin_usa
     0            70          False         False        True
     1            70          False         False        True
```

```
2              70          False          False         True
3              70          False          False         True
4              70          False          False         True
..            …            …           …         …
393            82          False          False         True
394            82           True          False        False
395            82          False          False         True
396            82          False          False         True
397            82          False          False         True

[398 rows x 10 columns]
```

[6]:
```
X = mpg.drop('mpg', axis=1)
y = mpg['mpg']

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,␣
 ↪random_state=19)
```

1.1 Linear Regression

[7]:
```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

[7]:
```
LinearRegression()
```

[8]:
```
y_pred = lr.predict(X_test)
y_pred
```

[8]:
```
array([24.30115472, 24.73892314, 24.96215822, 17.38881632, 15.86675499,
       27.76426172, 17.05954093, 10.49344079, 20.347386  , 25.11254933,
       19.36068674, 28.56363082, 26.2720292 , 17.73036858, 20.66471276,
       28.7669855 , 28.98129852, 29.25304272, 16.8393709 ,  8.76273604,
       16.24941413, 13.29303025, 27.03902874, 23.9878045 ,  8.31184254,
       31.52160011, 13.42789179, 35.09654981, 11.22026997, 23.39755039,
       33.81737584, 29.59696005, 15.57771451, 23.05695715, 31.56588099,
       36.11360599, 27.33151931, 12.33155235, 28.54718257, 15.3977963 ,
       31.21498729, 23.50190767, 14.44084076, 27.66645708, 26.78275774,
       25.05523353, 24.20075891, 16.30434811, 15.63453382, 12.61013012,
       17.36880625, 20.27285977, 30.10153781, 16.80607781, 33.29417373,
       11.53912186, 32.05759396, 21.35355061, 18.08124065, 30.24876403,
       20.89162341, 20.33755243, 10.89210702, 35.17824516, 32.51214434,
       15.01604073, 25.98942786, 15.12905345, 32.04280232, 24.17805781,
       29.82643316, 22.55608296, 14.09889069, 14.59105924, 26.25541638,
       23.19242234, 12.07920811, 33.66219498, 29.56124407, 20.90794551])
```

[9]:
```
mean_absolute_error(y_test, y_pred)
```

```
[9]: 2.3241334520650594
```

```
[10]: root_mean_squared_error(y_test, y_pred)
```

```
[10]: 8.30859663913005
```

```
[11]: r2_score(y_test, y_pred)
```

```
[11]: 0.8325409560733986
```

1.2 Random Forest Regressor

```
[12]: rfr = RandomForestRegressor(random_state=13)
      rfr.fit(X_train, y_train)
```

```
[12]: RandomForestRegressor(random_state=13)
```

```
[13]: y_pred_rfr = rfr.predict(X_test)
      y_pred_rfr
```

```
[13]: array([24.91 , 26.604, 24.747, 15.894, 15.851, 25.264, 18.296, 13.44 ,
             18.43 , 25.158, 17.164, 32.449, 26.397, 16.445, 19.475, 26.181,
             28.641, 33.474, 15.54 , 12.64 , 15.985, 14.75 , 24.137, 22.409,
             12.15 , 30.689, 13.895, 37.527, 13.52 , 21.971, 35.155, 27.763,
             15.234, 26.289, 33.153, 36.126, 23.992, 13.88 , 35.015, 14.225,
             31.179, 25.104, 15.482, 31.032, 25.78 , 26.102, 21.299, 16.106,
             14.604, 14.18 , 16.227, 18.722, 35.506, 16.804, 34.797, 13.82 ,
             32.936, 18.86 , 18.847, 36.382, 15.164, 21.129, 13.38 , 36.05 ,
             33.419, 16.248, 24.139, 16.082, 32.642, 23.887, 30.994, 20.967,
             14.33 , 15.03 , 24.663, 21.33 , 14.09 , 38.027, 31.285, 19.26 ])
```

```
[14]: mean_absolute_error(y_test, y_pred_rfr)
```

```
[14]: 1.6326000000000005
```

```
[15]: root_mean_squared_error(y_test, y_pred_rfr)
```

```
[15]: 5.647237075000005
```

```
[16]: r2_score(y_test, y_pred_rfr)
```

```
[16]: 0.8861804270347423
```

1.3 Ridge Regression

[17]:
```
ridge = Ridge()
param_grid = {
    'alpha': [0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30 ,50,75]
}

ridge_cv = GridSearchCV(ridge, param_grid, cv=5, n_jobs=-1)
ridge_cv.fit(X_train, y_train)
```

[17]:
```
GridSearchCV(cv=5, estimator=Ridge(), n_jobs=-1,
             param_grid={'alpha': [0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 50,
                                   75]})
```

[18]:
```
ridge_cv.best_estimator_
```

[18]:
```
Ridge(alpha=10)
```

[19]:
```
y_pred_ridge = ridge_cv.predict(X_test)
y_pred_ridge
```

[19]:
```
array([24.50784329, 25.1453431 , 24.53292975, 17.28315713, 15.76458563,
       27.56386969, 17.08360929, 10.4757291 , 20.27551626, 25.32880598,
       19.37647572, 28.86733796, 26.62154492, 17.73752567, 20.67788725,
       28.61052096, 29.25178742, 29.5422219 , 16.87281192,  8.53318336,
       16.27511627, 13.2104424 , 26.88340032, 24.009771  ,  8.54411519,
       31.26528627, 13.51952031, 34.90097167, 10.98124741, 23.16980535,
       33.58843945, 29.44372653, 15.49893771, 23.46053976, 31.4803678 ,
       35.94195329, 27.1668034 , 12.13433213, 28.84816147, 15.45947453,
       30.97278809, 23.81928018, 14.51031251, 27.2552616 , 26.63822582,
       25.25480835, 24.33315835, 16.31946596, 15.52974001, 12.67976987,
       17.40627283, 20.26742196, 30.05946769, 16.89475666, 33.16874796,
       11.53698617, 31.9061312 , 21.40546978, 18.10722736, 30.55085517,
       20.62056802, 20.53397226, 10.65600453, 35.01686624, 32.35642928,
       15.05582844, 25.86631482, 15.21926104, 31.87832831, 24.08914757,
       29.75506395, 22.43481848, 14.10345592, 14.38519282, 26.3607398 ,
       23.43599278, 12.1534485 , 33.50094882, 29.46266978, 20.95613101])
```

[20]:
```
mean_absolute_error(y_test, y_pred_ridge)
```

[20]:
```
2.298757602037677
```

[21]:
```
root_mean_squared_error(y_test, y_pred_ridge)
```

[21]:
```
8.045449532074198
```

[22]:
```
r2_score(y_test, y_pred_ridge)
```

```
[22]: 0.8378446631702253
```

1.4 Gradient Boosting Regressor

```
[23]: gbr = GradientBoostingRegressor()
      gbr.fit(X_train, y_train)
```

```
[23]: GradientBoostingRegressor()
```

```
[24]: y_pred_gbr = gbr.predict(X_test)
      y_pred_gbr
```

```
[24]: array([24.81614709, 26.92370305, 24.05991901, 17.30771176, 16.19740785,
             25.28778472, 18.47327473, 13.4293537 , 17.87010961, 25.32283075,
             17.3784293 , 32.80854588, 26.43231189, 16.71346622, 19.89542764,
             26.93072095, 29.09128776, 34.5437197 , 16.2287741 , 12.9976502 ,
             16.15260843, 14.0218489 , 23.430844  , 23.554742  , 12.15018212,
             29.3384603 , 13.78280808, 37.46136379, 13.83317576, 21.92720442,
             35.35308293, 27.59229457, 15.38298242, 25.77062188, 32.41174551,
             36.29002888, 22.94838594, 13.1043108 , 35.75852911, 14.27426203,
             31.12414823, 24.07273405, 15.53471207, 33.44820815, 26.03408155,
             25.8403547 , 20.59553436, 16.25817053, 15.70183123, 13.21764247,
             16.88855731, 18.84642478, 31.8104251 , 16.10117025, 36.1970594 ,
             13.4293537 , 33.0030821 , 18.74768476, 18.59036535, 35.9761289 ,
             19.47705059, 20.92402884, 14.42364406, 36.40972479, 32.92852343,
             16.49541674, 24.5359267 , 14.67871467, 31.81852565, 23.62370365,
             29.16526417, 20.5589327 , 14.29479101, 15.56229952, 23.83009838,
             21.90942175, 13.86561836, 34.69275483, 29.33933   , 19.03518476])
```

```
[25]: mean_absolute_error(y_test, y_pred_gbr)
```

```
[25]: 1.7513204620962235
```

```
[26]: root_mean_squared_error(y_test, y_pred_gbr)
```

```
[26]: 5.225603103128156
```

```
[27]: r2_score(y_test, y_pred_gbr)
```

```
[27]: 0.8946784231324356
```

1.5 Stacking Regression

```
[28]: estimators = [
          ('rfr', rfr),
          ('ridge', ridge_cv.best_estimator_),
          ('lr', lr)
      ]
```

```
sr = StackingRegressor(
    estimators=estimators,
    final_estimator=gbr
)

sr.fit(X_train, y_train)
```

```
[28]: StackingRegressor(estimators=[('rfr', RandomForestRegressor(random_state=13)),
                                    ('ridge', Ridge(alpha=10)),
                                    ('lr', LinearRegression())],
                      final_estimator=GradientBoostingRegressor())
```

```
[29]: y_pred_sr = sr.predict(X_test)
      y_pred_sr
```

```
[29]: array([23.53378874, 25.71455216, 24.09668026, 15.40149474, 15.40615392,
             27.36438654, 17.98058559, 13.43407508, 18.52080362, 24.24343196,
             16.93869117, 30.90969533, 26.90396953, 16.73109235, 18.98478987,
             26.63577215, 25.55108187, 29.9583349 , 15.33757724, 12.2164736 ,
             15.40615392, 15.17557021, 24.92925791, 22.1997565 , 12.13294436,
             33.13916835, 13.79983485, 34.80103745, 13.56035943, 20.45624966,
             36.40332172, 27.01318555, 15.40615392, 23.05829459, 31.64665183,
             36.72001283, 27.15296778, 13.79983485, 32.15462099, 14.16801969,
             34.11696062, 23.53378874, 15.87656151, 32.40313431, 25.59016802,
             25.45964114, 20.68403814, 15.81340739, 15.40615392, 14.3074484 ,
             15.9730475 , 18.43689213, 31.85386936, 16.50287556, 39.40124297,
             13.56035943, 33.96965856, 17.56922769, 18.4320498 , 36.70733073,
             16.3237033 , 20.25150774, 13.56035943, 37.78515789, 34.28042863,
             15.87161528, 25.40213556, 15.87161528, 32.20626472, 23.6224426 ,
             30.78302652, 20.5972072 , 15.00843971, 15.46436181, 25.59016802,
             20.68403814, 14.3074484 , 37.76093249, 30.96411227, 18.98478987])
```

```
[30]: mean_absolute_error(y_test, y_pred_sr)
```

```
[30]: 1.9308715384123967
```

```
[31]: root_mean_squared_error(y_test, y_pred_sr)
```

```
[31]: 6.580558650525768
```

```
[32]: r2_score(y_test, y_pred_sr)
```

```
[32]: 0.8673694117090569
```

1.6 Voting Regressor

```
[33]: vr = VotingRegressor([
          ('rfr', rfr),
          ('gbr', gbr),
          ('lr', lr)
      ], weights=[2,3,1])

      vr.fit(X_train, y_train)
```

```
[33]: VotingRegressor(estimators=[('rfr', RandomForestRegressor(random_state=13)),
                                  ('gbr', GradientBoostingRegressor()),
                                  ('lr', LinearRegression())],
                      weights=[2, 3, 1])
```

```
[34]: y_pred_vc = vr.predict(X_test)
      y_pred_vc
```

```
[34]: array([24.76159933, 26.45300538, 24.43931921, 16.84999193, 16.02682976,
             25.69260264, 18.17856085, 12.94358365, 18.46961914, 25.23284026,
             17.63732911, 31.98121141, 26.39382748, 16.79346121, 19.88349928,
             26.98685806, 28.9228603 , 33.30536697, 16.10094887, 12.17261444,
             16.11287324, 14.14309616, 24.26759346, 23.24500508, 11.51039815,
             30.15249683, 13.76105267, 37.08910687, 13.29329954, 22.18686061,
             35.0311041 , 27.98330729, 15.36577696, 25.49113713, 32.49028417,
             36.20594877, 24.02677952, 13.23408079, 34.30879498, 14.44509707,
             31.157572  , 24.32135164, 15.3348295 , 31.67918025, 26.07416706,
             25.79671627, 21.43089366, 16.21514328, 15.32467125, 13.43717625,
             16.7480797 , 19.04268902, 32.84100406, 16.45293143, 35.24655866,
             13.24453049, 32.82314004, 19.21943415, 18.59105612, 35.15685845,
             18.27006274, 20.89460649, 13.48210674, 36.08456992, 33.0226191 ,
             16.16638182, 24.64586799, 15.22153291, 32.13039654, 23.51922136,
             29.75720817, 21.02781351, 14.27387728, 15.22299296, 24.51195192,
             21.9301146 , 13.6426772 , 35.63240991, 30.02487235, 19.42224996])
```

```
[35]: root_mean_squared_error(y_test, y_pred_vc)
```

```
[35]: 5.0571779581339475
```

```
[36]: r2_score(y_test, y_pred_vc)
```

```
[36]: 0.8980730173074735
```

1.7 Final Estimator

[37]:
```
estimators2 = [
    ('rfr', rfr),
    ('ridge', ridge_cv.best_estimator_),
    ('gbr', gbr)
]



sr2 = StackingRegressor(
    estimators=estimators2,
    final_estimator=vr
)
sr2.fit(X_train, y_train)
```

[37]:
```
StackingRegressor(estimators=[('rfr', RandomForestRegressor(random_state=13)),
                              ('ridge', Ridge(alpha=10)),
                              ('gbr', GradientBoostingRegressor())],
                  final_estimator=VotingRegressor(estimators=[('rfr',
RandomForestRegressor(random_state=13)),
                                                              ('gbr',
GradientBoostingRegressor()),
                                                              ('lr',
LinearRegression())],
                                                  weights=[2, 3, 1]))
```

[38]:
```
y_pred_sr2 = sr2.predict(X_test)
y_pred_sr2
```

[38]:
```
array([24.3491363 , 27.11568059, 23.39061009, 17.05006586, 15.40976879,
       25.52060588, 18.04750935, 13.39328651, 18.64617906, 24.98338014,
       17.1792968 , 30.97858195, 28.10750468, 15.68440833, 19.64604119,
       27.63643889, 26.05606814, 30.67613107, 15.67300316, 12.33635145,
       15.40526841, 14.087848  , 25.02953828, 22.15264521, 10.96638522,
       33.17943801, 13.99024305, 35.98606902, 13.3395992 , 21.31222876,
       36.64232503, 28.26352216, 15.0655988 , 24.74528493, 32.06494113,
       37.00372476, 25.66198544, 13.83742911, 32.91868345, 13.83196459,
       33.0385216 , 23.3559947 , 15.62901099, 30.6175818 , 27.05611273,
       25.57012826, 21.10867599, 15.50163291, 15.26869649, 13.67442293,
       15.7415394 , 18.49101327, 34.04061254, 16.02565044, 38.78773653,
       13.52816628, 33.1378723 , 17.90159211, 18.37792041, 35.9752136 ,
       18.70332375, 20.73245054, 14.26120238, 37.42491304, 33.21332751,
       15.71597323, 25.43355876, 15.75085825, 32.73059787, 23.50424327,
       30.53905967, 20.94953693, 14.23724779, 15.62513511, 24.19958196,
       20.9596028 , 13.96885838, 37.38444735, 30.55691623, 19.2151501 ])
```

[39]:
```
root_mean_squared_error(y_test, y_pred_sr2)
```

[39]: 6.414412415181024

[40]:
```
r2_score(y_test, y_pred_sr2)
```

[40]: 0.8707180746579599

[41]:
```
estimators3 = [
    ('rfr', rfr),
    ('ridge', ridge_cv.best_estimator_),
    ('svr', SVR(C=1.0, kernel='linear')),
    ('random_forest',RandomForestRegressor())
]

sr3 = StackingRegressor(
    estimators=estimators3,
    final_estimator=Ridge(alpha=1.0)
)

param_grid_sr = {
    'random_forest__n_estimators':[50,100,250],
    'svr__C': [0.1, 1.0, 10.0],
    'final_estimator__alpha': [0.1,1.0,10.0]
}

sr_cv = RandomizedSearchCV(sr3, param_grid_sr, n_iter=5, cv=3,␣
 ↪scoring='neg_root_mean_squared_error', n_jobs=-1)
sr_cv.fit(X_train, y_train)
```

[41]:
```
RandomizedSearchCV(cv=3,
                   estimator=StackingRegressor(estimators=[('rfr',
RandomForestRegressor(random_state=13)),
                                                           ('ridge',
                                                            Ridge(alpha=10)),
                                                           ('svr',
SVR(kernel='linear')),
                                                           ('random_forest',
RandomForestRegressor())],
                                               final_estimator=Ridge()),
                   n_iter=5, n_jobs=-1,
                   param_distributions={'final_estimator__alpha': [0.1, 1.0,
                                                                   10.0],
                                        'random_forest__n_estimators': [50, 100,
                                                                        250],
                                        'svr__C': [0.1, 1.0, 10.0]},
                   scoring='neg_mean_squared_error')
```

[42]:
```
y_pred_8 = sr_cv.predict(X_test)
y_pred_8
```

[42]:
```
array([24.45505141, 26.08064778, 25.377915  , 15.93300187, 15.24263626,
       25.45247833, 17.30640013, 12.365641  , 18.28087359, 25.39590536,
       17.09670375, 30.33733978, 26.29055207, 16.56544171, 19.26286061,
       27.02489251, 29.40600026, 32.61210673, 15.07380717, 11.0614695 ,
       15.61011049, 13.4696746 , 24.67148938, 23.53833227, 11.43036111,
       31.19528332, 13.48103115, 37.06619408, 12.66684219, 21.79112221,
       34.90552177, 28.92212115, 14.95776505, 25.75974317, 32.5812645 ,
       36.34849374, 24.79244634, 12.77105095, 32.95633545, 14.24149635,
       31.54489825, 24.29351523, 14.88776827, 30.01059792, 26.08003928,
       25.61700765, 21.68020634, 15.65541277, 14.90706114, 13.46447027,
       15.61558932, 18.61179576, 34.77888606, 16.10115523, 34.68386065,
       13.07459897, 31.41977482, 19.17668167, 17.95115591, 34.99862448,
       16.6652014 , 20.35133345, 12.5105491 , 35.28992113, 33.31986247,
       15.64511174, 24.53707771, 15.51719415, 31.25388316, 23.41547812,
       30.63741646, 21.42362772, 14.42679083, 14.19939658, 25.45697562,
       22.11579612, 13.25555966, 37.29781596, 30.58671278, 19.00212692])
```

[43]:
```
root_mean_squared_error(y_test, y_pred_8)
```

[43]:
```
5.580300867802961
```

[44]:
```
r2_score(y_test, y_pred_8)
```

[44]:
```
0.8875295204795366
```

[45]:
```
sr_cv.best_estimator_
```

[45]:
```
StackingRegressor(estimators=[('rfr', RandomForestRegressor(random_state=13)),
                              ('ridge', Ridge(alpha=10)),
                              ('svr', SVR(C=10.0, kernel='linear')),
                              ('random_forest', RandomForestRegressor())],
                  final_estimator=Ridge(alpha=0.1))
```

27-Random_Forest_Regressor

October 20, 2024

1 Random Forest Regressor

A **Random Forest Regressor** is an ensemble learning method that combines the predictions of multiple decision trees to improve predictive performance for regression tasks. Unlike a single decision tree, which might overfit or have high variance, a random forest aggregates the outputs of many trees to make the final prediction, thus reducing overfitting and variance while improving generalization.

It works by constructing several decision trees during training and averaging their predictions. This process of averaging multiple trees helps to smooth out the predictions and gives more robust results.

1.0.1 Why is Random Forest Regressor Important?

- **Reduces Overfitting**: Individual decision trees tend to overfit, especially when the trees are deep. By averaging the predictions of many trees, Random Forest mitigates overfitting, making the model more generalizable to unseen data.
- **Handles Non-linear Relationships**: Decision trees naturally handle non-linear relationships. A random forest regressor benefits from this, enabling it to capture complex patterns in the data.
- **Robustness to Noise and Outliers**: Because Random Forest averages the predictions of many trees, it is less sensitive to outliers and noise than individual decision trees.
- **Feature Importance**: Random Forest provides feature importance scores, which can be useful in understanding which features have the most predictive power.

1.0.2 How does Random Forest Regressor work?

Random Forest Regressor is based on an ensemble of decision trees. Here's a step-by-step explanation of how it works:

1. Bagging (Bootstrap Aggregating): The first key idea behind Random Forest is bagging. Random subsets (bootstrapped samples) of the original training data are created. Each decision tree in the forest is trained on a different random sample of the data. This technique helps in reducing the variance and overfitting that can occur with individual trees.

2. Random Feature Selection**: During the construction of each tree, Random Forest also selects a random subset of features to consider at each split. This random selection of features ensures that the trees in the forest are more diverse, further reducing correlation between the trees.

3. Tree Construction: Each tree is built independently by splitting the data at different nodes based on feature values (as in any decision tree). The process is recursive, and the tree grows until a stopping criterion is met (e.g., a minimum number of samples per leaf or a maximum tree depth).

4. Prediction: After all the trees are trained, predictions are made by each tree, and the final prediction for regression is the average of the predictions from all the trees.

1.0.3 When should you use Random Forest Regressor?

- **Non-linear and complex datasets**: Random Forest works well when your dataset has complex, non-linear relationships between features and target values.
- **High-dimensional data**: It handles high-dimensional datasets with many features and can reduce the risk of overfitting by selecting random subsets of features.
- **When interpretability is not the primary goal**: Random Forest is less interpretable than simpler models like linear regression, so it's best used when predictive accuracy is more important than understanding the model's inner workings.
- **When there are many categorical and continuous variables**: It can handle mixed-type data without much preprocessing.

1.0.4 Who uses Random Forest Regressor?

- **Data Scientists**: Random Forest is widely used in industries like finance, healthcare, and marketing to solve regression problems where complex patterns need to be captured.
- **Business Analysts**: It is often used to predict continuous outcomes such as sales forecasts, customer retention rates, and other business metrics.
- **Researchers**: In research fields like genomics or environmental science, Random Forest is applied to make predictions based on numerous features, and it's useful for feature selection.

1.0.5 Key Points to Remember:

- **Ensemble of Decision Trees**: Random Forest is a collection of decision trees, each trained on a different bootstrap sample of the data.
- **Random Feature Selection**: At each split, a random subset of features is chosen, making the trees less correlated with one another.
- **Averaging Predictions**: For regression tasks, the final prediction is the average of the predictions made by all the trees.
- **Feature Importance**: Random Forest naturally provides feature importance scores, which can help identify the most relevant features.

1.0.6 Advantages of Random Forest Regressor:

- **Reduced Overfitting**: By averaging multiple trees, Random Forest smooths out the predictions and reduces overfitting compared to a single decision tree.
- **Handles Missing Data**: Random Forest can handle missing values relatively well by filling in missing data based on correlations.

- **Robust to Outliers**: Due to the averaging process, Random Forest is less sensitive to outliers in the data.
- **No need for feature scaling**: Unlike models like SVM or linear regression, Random Forest doesn't require feature scaling (e.g., normalization or standardization).

[1]:
```
import pandas as pd
import seaborn as sns

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
 ↪r2_score

healthexp = sns.load_dataset('healthexp')
healthexp
```

[1]:
```
     Year        Country  Spending_USD  Life_Expectancy
0    1970        Germany       252.311             70.6
1    1970         France       192.143             72.2
2    1970  Great Britain       123.993             71.9
3    1970          Japan       150.437             72.0
4    1970            USA       326.961             70.9
..    …              …            …                …
269  2020        Germany      6938.983             81.1
270  2020         France      5468.418             82.3
271  2020  Great Britain      5018.700             80.4
272  2020          Japan      4665.641             84.7
273  2020            USA     11859.179             77.0

[274 rows x 4 columns]
```

[2]:
```
healthexp = pd.get_dummies(healthexp)
healthexp
```

[2]:
```
     Year  Spending_USD  Life_Expectancy  Country_Canada  Country_France  \
0    1970       252.311             70.6           False           False
1    1970       192.143             72.2           False            True
2    1970       123.993             71.9           False           False
3    1970       150.437             72.0           False           False
4    1970       326.961             70.9           False           False
..    …            …                …               …               …
269  2020      6938.983             81.1           False           False
270  2020      5468.418             82.3           False            True
271  2020      5018.700             80.4           False           False
272  2020      4665.641             84.7           False           False
273  2020     11859.179             77.0           False           False
```

```
     Country_Germany  Country_Great Britain  Country_Japan  Country_USA
0               True                  False          False        False
1              False                  False          False        False
2              False                   True          False        False
3              False                  False           True        False
4              False                  False          False         True
..               …                      …             …          …
269             True                  False          False        False
270            False                  False          False        False
271            False                   True          False        False
272            False                  False           True        False
273            False                  False          False         True

[274 rows x 9 columns]
```

[3]:
```
X = healthexp.drop(['Life_Expectancy'], axis=1)
y = healthexp['Life_Expectancy']

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2,␣
↪random_state=19)

rfr = RandomForestRegressor(random_state=13)
rfr.fit(X_train, y_train)
```

[3]:
```
RandomForestRegressor(random_state=13)
```

[4]:
```
y_pred = rfr.predict(X_test)
y_pred
```

[4]:
```
array([79.705, 81.089, 74.833, 79.357, 71.152, 79.186, 82.417, 81.7  ,
       78.368, 71.48 , 76.446, 75.79 , 82.694, 78.601, 76.848, 77.614,
       71.838, 78.91 , 72.414, 82.033, 71.816, 78.415, 79.937, 78.513,
       78.265, 72.639, 75.759, 81.337, 81.408, 82.269, 74.003, 74.395,
       78.677, 74.98 , 71.777, 74.347, 74.581, 76.581, 81.142, 80.895,
       74.979, 81.68 , 77.489, 78.219, 80.888, 83.572, 78.392, 73.976,
       77.236, 80.114, 74.728, 76.782, 78.031, 78.778, 77.519])
```

[5]:
```
mean_absolute_error(y_test, y_pred)
```

[5]:
```
0.25916363636361917
```

[6]:
```
root_mean_squared_error(y_test, y_pred)
```

[6]:
```
0.31970520512155615
```

[7]:
```
r2_score(y_test, y_pred)
```

[7]:
```
0.9910457602615238
```

[8]:
```
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10,20,30],
    'min_samples_split': [2,5,10],
    'min_samples_leaf': [1,2,4]
}

rfr_cv = GridSearchCV(rfr, param_grid, cv=3, n_jobs=-1)
rfr_cv.fit(X_train, y_train)
```

[8]:
```
GridSearchCV(cv=3, estimator=RandomForestRegressor(random_state=13), n_jobs=-1,
             param_grid={'max_depth': [10, 20, 30],
                         'min_samples_leaf': [1, 2, 4],
                         'min_samples_split': [2, 5, 10],
                         'n_estimators': [100, 200, 300]})
```

[9]:
```
y_pred_cv = rfr_cv.predict(X_test)
y_pred_cv
```

[9]:
```
array([79.71933333, 81.09833333, 74.83733333, 79.40433333, 71.159     ,
       79.232     , 82.41166667, 81.686     , 78.43533333, 71.404     ,
       76.47266667, 75.904     , 82.69633333, 78.64833333, 76.843     ,
       77.58633333, 71.715     , 78.93433333, 72.42533333, 82.016     ,
       71.731     , 78.45666667, 80.        , 78.529     , 78.31966667,
       72.51466667, 75.78433333, 81.29033333, 81.392     , 82.281     ,
       73.972     , 74.404     , 78.67233333, 74.99466667, 71.68766667,
       74.34333333, 74.56433333, 76.604     , 81.14366667, 80.91     ,
       74.97633333, 81.66533333, 77.59366667, 78.17166667, 80.87033333,
       83.52633333, 78.43366667, 73.93433333, 77.149     , 80.104     ,
       74.718     , 76.741     , 78.136     , 78.70566667, 77.53133333])
```

[10]:
```
mean_absolute_error(y_test, y_pred_cv)
```

[10]:
```
0.25089696969702713
```

[11]:
```
root_mean_squared_error(y_test, y_pred_cv)
```

[11]:
```
0.31042865768991584
```

[12]:
```
r2_score(y_test, y_pred_cv)
```

[12]:
```
0.9915578528520343
```

28-Gradient_Boosting_Regressor

October 20, 2024

1 Gradient Boosting Regressor

A **Gradient Boosting Regressor** is a machine learning algorithm that builds an ensemble of decision trees sequentially to minimize the loss function (or error) for regression tasks. Unlike Random Forest, which builds trees independently, Gradient Boosting builds trees sequentially, where each new tree aims to correct the errors made by the previous trees. It uses the gradient descent technique to optimize the model by reducing the difference between the predicted and actual values.

It combines the predictions of many "weak learners" (typically shallow decision trees) to create a more accurate final prediction. The key idea is to reduce bias and variance by focusing on improving areas where the model previously performed poorly.

1.0.1 Why is Gradient Boosting Regressor Important?

- **Accurate Predictions**: Gradient Boosting often produces very accurate models that outperform simpler models like linear regression, and even Random Forest, in certain cases.
- **Focuses on Hard-to-Predict Instances**: By iteratively correcting errors, Gradient Boosting effectively captures complex patterns in the data that may be missed by other models.
- **Balances Bias-Variance Tradeoff**: Gradient Boosting reduces both bias (error due to underfitting) and variance (error due to overfitting), providing a powerful model in various situations.

1.0.2 How does Gradient Boosting Regressor work?

The core idea behind Gradient Boosting is to add new models that improve the residual errors (i.e., the difference between predicted and true values) of previous models. Here's the step-by-step process:

1. Initial Model: Start with an initial model, typically predicting the mean of the target variable.

2. Calculate Residuals*: For each data point, compute the residual (the difference between the true value and the model's prediction).

3. Fit a Weak Learner: A decision tree is fitted to these residuals. This new tree learns to predict the errors (or residuals) made by the previous model.

4. Update the Model: The new predictions are added to the previous predictions. This step is scaled by a learning rate to control the contribution of the new tree.

5. Repeat: This process is repeated iteratively, each time fitting a new tree to the residuals and updating the model.

6. Final Prediction: After a certain number of iterations (trees), the final model is the sum of the predictions from all trees.

The key difference between Gradient Boosting and Random Forest is that Gradient Boosting builds trees sequentially, with each new tree correcting the errors made by the previous trees, whereas Random Forest builds trees independently in parallel.

1.0.3 When should you use Gradient Boosting Regressor?

- **When high accuracy is needed**: Gradient Boosting is particularly powerful when you need highly accurate predictions and are willing to trade off some interpretability.
- **Imbalanced or Noisy Data**: Gradient Boosting handles imbalanced and noisy data well, making it suitable for real-world applications where clean, well-balanced datasets are rare.
- **When dealing with complex relationships**: If the data exhibits complex relationships between features and target variables, Gradient Boosting can model these effectively.
- **When feature importance is important**: Like Random Forest, Gradient Boosting also provides feature importance, which can help in understanding which features contribute the most to the predictions.

1.0.4 Who uses Gradient Boosting Regressor?

- **Data Scientists and Machine Learning Engineers**: Gradient Boosting is widely used in industries like finance, healthcare, and marketing for predictive modeling tasks where high accuracy is crucial.
- **Kaggle Competitors**: Gradient Boosting models, especially variants like XGBoost, LightGBM, and CatBoost, are commonly used in data science competitions because of their ability to achieve state-of-the-art performance.
- **Researchers**: In research fields like genetics or climate modeling, Gradient Boosting is used for regression tasks that require capturing non-linear patterns and interactions among features.

1.0.5 Key Points to Remember:

- **Sequential Trees**: Gradient Boosting builds trees one at a time, with each tree focusing on correcting the residual errors of the previous trees.
- **Learning Rate**: The learning rate controls how much the new tree's predictions influence the overall model. A lower learning rate requires more trees to be effective but helps prevent overfitting.
- **Combining Weak Learners**: It combines many "weak learners" (typically small decision trees) to create a strong predictive model.
- **Loss Function**: Gradient Boosting minimizes a loss function (e.g., Mean Squared Error for regression) by performing gradient descent in function space.

1.0.6 Advantages of Gradient Boosting Regressor:

1. High Predictive Accuracy: Gradient Boosting tends to produce highly accurate models that often outperform other models on many datasets.

2. Flexibility: You can specify different loss functions (e.g., least squares for regression) and customize the model's hyperparameters.

3. Handles Missing Data: Gradient Boosting can handle missing data quite well and does not require complex imputation strategies.

4. Feature Importance: It provides feature importance, which can be useful for understanding which features have the most predictive power.

5. Robust to Outliers: It focuses on reducing residual errors, making it more robust to outliers in some cases.

1.0.7 Limitations of Gradient Boosting Regressor:

- **Longer Training Time**: Since Gradient Boosting builds trees sequentially, training can take significantly longer than other algorithms like Random Forest, especially on large datasets.
- **Sensitive to Hyperparameters**: Gradient Boosting can be sensitive to the choice of hyperparameters, such as the number of trees, learning rate, and tree depth. It often requires careful tuning to perform optimally.
- **Can Overfit**: Although Gradient Boosting generally performs well, it can overfit the training data if the number of trees is too high or if the trees are too deep.
- **Not Interpretable**: Like other ensemble methods, Gradient Boosting produces complex models that are difficult to interpret compared to simpler models like linear regression.

[1]:
```
import pandas as pd

from sklearn import datasets
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, root_mean_squared_error,␣
 ↪r2_score

diabetes = datasets.load_diabetes()
diabetes.data
```

[1]:
```
array([[ 0.03807591,  0.05068012,  0.06169621, …, -0.00259226,
         0.01990749, -0.01764613],
       [-0.00188202, -0.04464164, -0.05147406, …, -0.03949338,
        -0.06833155, -0.09220405],
       [ 0.08529891,  0.05068012,  0.04445121, …, -0.00259226,
         0.00286131, -0.02593034],
       …,
       [ 0.04170844,  0.05068012, -0.01590626, …, -0.01107952,
        -0.04688253,  0.01549073],
       [-0.04547248, -0.04464164,  0.03906215, …,  0.02655962,
         0.04452873, -0.02593034],
       [-0.04547248, -0.04464164, -0.0730303 , …, -0.03949338,
        -0.00422151,  0.00306441]])
```

[2]:
```
diabetes.target
```

[2]:
```
array([151.,  75., 141., 206., 135.,  97., 138.,  63., 110., 310., 101.,
        69., 179., 185., 118., 171., 166., 144.,  97., 168.,  68.,  49.,
        68., 245., 184., 202., 137.,  85., 131., 283., 129.,  59., 341.,
        87.,  65., 102., 265., 276., 252.,  90., 100.,  55.,  61.,  92.,
       259.,  53., 190., 142.,  75., 142., 155., 225.,  59., 104., 182.,
       128.,  52.,  37., 170., 170.,  61., 144.,  52., 128.,  71., 163.,
       150.,  97., 160., 178.,  48., 270., 202., 111.,  85.,  42., 170.,
       200., 252., 113., 143.,  51.,  52., 210.,  65., 141.,  55., 134.,
        42., 111.,  98., 164.,  48.,  96.,  90., 162., 150., 279.,  92.,
        83., 128., 102., 302., 198.,  95.,  53., 134., 144., 232.,  81.,
       104.,  59., 246., 297., 258., 229., 275., 281., 179., 200., 200.,
       173., 180.,  84., 121., 161.,  99., 109., 115., 268., 274., 158.,
       107.,  83., 103., 272.,  85., 280., 336., 281., 118., 317., 235.,
        60., 174., 259., 178., 128.,  96., 126., 288.,  88., 292.,  71.,
       197., 186.,  25.,  84.,  96., 195.,  53., 217., 172., 131., 214.,
        59.,  70., 220., 268., 152.,  47.,  74., 295., 101., 151., 127.,
       237., 225.,  81., 151., 107.,  64., 138., 185., 265., 101., 137.,
       143., 141.,  79., 292., 178.,  91., 116.,  86., 122.,  72., 129.,
       142.,  90., 158.,  39., 196., 222., 277.,  99., 196., 202., 155.,
        77., 191.,  70.,  73.,  49.,  65., 263., 248., 296., 214., 185.,
        78.,  93., 252., 150.,  77., 208.,  77., 108., 160.,  53., 220.,
       154., 259.,  90., 246., 124.,  67.,  72., 257., 262., 275., 177.,
        71.,  47., 187., 125.,  78.,  51., 258., 215., 303., 243.,  91.,
       150., 310., 153., 346.,  63.,  89.,  50.,  39., 103., 308., 116.,
       145.,  74.,  45., 115., 264.,  87., 202., 127., 182., 241.,  66.,
        94., 283.,  64., 102., 200., 265.,  94., 230., 181., 156., 233.,
        60., 219.,  80.,  68., 332., 248.,  84., 200.,  55.,  85.,  89.,
        31., 129.,  83., 275.,  65., 198., 236., 253., 124.,  44., 172.,
       114., 142., 109., 180., 144., 163., 147.,  97., 220., 190., 109.,
       191., 122., 230., 242., 248., 249., 192., 131., 237.,  78., 135.,
       244., 199., 270., 164.,  72.,  96., 306.,  91., 214.,  95., 216.,
       263., 178., 113., 200., 139., 139.,  88., 148.,  88., 243.,  71.,
        77., 109., 272.,  60.,  54., 221.,  90., 311., 281., 182., 321.,
        58., 262., 206., 233., 242., 123., 167.,  63., 197.,  71., 168.,
       140., 217., 121., 235., 245.,  40.,  52., 104., 132.,  88.,  69.,
       219.,  72., 201., 110.,  51., 277.,  63., 118.,  69., 273., 258.,
        43., 198., 242., 232., 175.,  93., 168., 275., 293., 281.,  72.,
       140., 189., 181., 209., 136., 261., 113., 131., 174., 257.,  55.,
        84.,  42., 146., 212., 233.,  91., 111., 152., 120.,  67., 310.,
        94., 183.,  66., 173.,  72.,  49.,  64.,  48., 178., 104., 132.,
       220.,  57.])
```

[3]:
```
X = diabetes.data
y = diabetes.target

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2,␣
 ↪random_state=17)

gbr = GradientBoostingRegressor()
gbr.fit(X_train, y_train)
```

[3]:
```
GradientBoostingRegressor()
```

[4]:
```
y_pred = gbr.predict(X_test)
y_pred
```

[4]:
```
array([192.51225586, 200.5750764 ,  82.78231894, 101.58510513,
       125.14414245,  81.39853646,  93.84628461, 102.45928951,
       121.90945671,  74.25891122, 255.91425718, 175.42143255,
        93.81164289,  99.4616262 , 217.43499676, 217.43197684,
       190.62007333, 172.98697169, 197.2837651 , 175.80982432,
       182.09395792, 269.70763529, 196.07382183,  82.25148771,
       114.47110984, 185.87258222, 109.04630496, 165.3091288 ,
        84.08079017, 241.11508989,  89.77664274,  69.54907641,
       108.03496427, 288.56040447, 170.41296018, 164.48100696,
        82.93345568, 179.53609042,  91.32177689, 245.58054393,
       112.63272196,  90.77107175, 179.75704585, 306.48853493,
       154.11016096,  91.65230129, 200.49974318, 107.20195482,
       190.62028854, 167.72815926, 191.98143713, 186.27531144,
       200.12776418, 156.00397936, 180.02678304, 138.30178657,
        96.52611334, 300.64236743, 138.20868823, 162.7712336 ,
       176.10010144, 231.10141163,  91.57716704, 143.7471825 ,
        87.83391334,  91.52052802, 216.22481281,  68.19976982,
       167.75688514, 157.00162733, 103.12604888, 283.27021778,
       267.45306696, 196.29244184,  96.48618333, 124.81762663,
       199.10839318, 149.25733184, 145.53020157, 178.10088311,
       258.17118587,  92.95097597, 149.38905097,  71.33146438,
       272.96959869, 311.01641671,  95.577347  , 104.14250789,
        83.75796396])
```

[5]:
```
mean_absolute_error(y_test, y_pred)
```

[5]:
```
48.792591465108316
```

[6]:
```
root_mean_squared_error(y_test, y_pred)
```

[6]:
```
60.90137361126108
```

```
[7]: r2_score(y_test, y_pred)
```

```
[7]: 0.37191596197265364
```

```
[8]: param_grid = {
         'n_estimators': [100, 200, 300],
         'learning_rate': [0.01, 0.1, 0.2],
         'max_depth': [3,4,5],
         'min_samples_split': [2,3,4],
         'min_samples_leaf': [1,2,3]
     }

     gbr_cv = GridSearchCV(gbr, param_grid, cv=5,␣
      ↪n_jobs=-1,scoring='neg_root_mean_squared_error')
     gbr_cv.fit(X_test, y_test)
```

```
[8]: GridSearchCV(cv=5, estimator=GradientBoostingRegressor(), n_jobs=-1,
                  param_grid={'learning_rate': [0.01, 0.1, 0.2],
                              'max_depth': [3, 4, 5], 'min_samples_leaf': [1, 2, 3],
                              'min_samples_split': [2, 3, 4],
                              'n_estimators': [100, 200, 300]},
                  scoring='neg_root_mean_squared_error')
```

```
[9]: y_pred_gbr_cv = gbr_cv.predict(X_test)
     y_pred_gbr_cv
```

```
[9]: array([167.54547384, 196.45689046, 131.99856228, 128.81980861,
            155.38736053, 104.78343157, 104.46878826, 103.86864233,
            172.16237464,  92.877811  , 226.28290431, 151.96615636,
             99.36099017, 184.74380459, 209.7874883 , 169.38985178,
            240.03918274, 210.69910838, 247.98095965, 113.01953415,
            110.64514541, 232.66480304, 109.31434018,  91.94434266,
             96.51531206, 215.20968459, 105.09234396, 138.79638199,
             82.32851947, 215.84795964, 119.4373777 , 116.85111785,
            109.52830044, 199.97027303, 126.65963299, 231.83342139,
             79.28364229, 253.17948242, 107.99860644, 248.12009523,
             95.73091409, 134.91997317, 165.55760765, 241.81722869,
            107.02535237,  78.75843641, 211.98096742,  99.36099017,
            137.05366889, 163.12544504, 164.34451529, 178.11208294,
            182.44224851, 114.1775484 , 201.79323332, 104.77956892,
            115.67104533, 233.66677325, 163.98975366, 221.99947933,
            198.83299795, 172.19115755,  90.89484407, 123.40323468,
            131.31453483, 191.68730783, 167.43521912, 105.88265868,
            187.91161719, 150.97652913, 119.96678341, 241.33644356,
            257.04655152, 164.34451529,  99.89637165, 143.87516951,
            101.74985603, 243.13993649, 152.49661722, 231.19421556,
            233.42395538, 120.44390831, 140.12835633,  81.63035609,
       236.03992088, 239.51517181, 117.57491008, 104.18911766,
       104.31100263])
```

```
[10]: mean_absolute_error(y_test, y_pred_gbr_cv)
```

```
[10]: 27.560044315482546
```

```
[11]: root_mean_squared_error(y_test, y_pred_gbr_cv)
```

```
[11]: 32.99791585313554
```

```
[12]: r2_score(y_test, y_pred_gbr_cv)
```

```
[12]: 0.8156103237854524
```

```
[13]: gbr_cv.best_params_
```

```
[13]: {'learning_rate': 0.01,
       'max_depth': 3,
       'min_samples_leaf': 3,
       'min_samples_split': 3,
       'n_estimators': 200}
```

29-Optuna_Hypertuning

October 20, 2024

1 Hypertuning with Optuna

Optuna is an open-source, hyperparameter optimization framework designed for machine learning and deep learning models. It automates the search for optimal hyperparameters, helping data scientists and machine learning engineers tune models more efficiently. Optuna is flexible, scalable, and supports both sequential and parallel optimization, allowing it to adapt to different scales of problems.

Optuna's key strength is its define-by-run approach, where hyperparameter configurations are dynamically constructed during each trial, making it more efficient and flexible than traditional grid or random search methods.

1.0.1 Why is Optuna Important?

Tuning hyperparameters is a critical part of improving machine learning models' performance, but it can be extremely time-consuming, especially for complex models like deep neural networks or ensembles. Optuna helps by:

- **Automating hyperparameter search**: Reducing the manual effort required to find optimal settings.
- **Efficient search**: Optuna uses sophisticated algorithms like Tree-structured Parzen Estimator (TPE) and CMA-ES (Covariance Matrix Adaptation Evolution Strategy) to explore the hyperparameter space more intelligently than grid or random search.
- **Early stopping**: It includes pruning to stop trials that are not promising early, saving time and computational resources.
- **Scalability**: Optuna can be easily scaled to large clusters or cloud environments, making it practical for tuning complex models.

1.0.2 How does Optuna Hypertuning Work?

Optuna performs hyperparameter optimization through an iterative process called trials, where each trial represents a single set of hyperparameters and their corresponding evaluation. Optuna tries to minimize (or maximize) the objective function defined by the user. Here's how it works:

1. Define the Objective Function: The objective function evaluates the model's performance using a given set of hyperparameters. It returns a metric (like accuracy, loss, etc.) to minimize (or maximize).

2. Sampling Hyperparameters: Optuna uses advanced optimization algorithms like TPE or CMA-ES to select hyperparameter values for each trial. It doesn't just randomly choose values but uses past trial results to guide the search toward more promising regions of the hyperparameter space.

3. Run Trials: Optuna evaluates the model with the selected hyperparameters in each trial and records the objective value (e.g., validation accuracy or loss).

4. Pruning: If a trial is not performing well, Optuna can prune (i.e., stop) it early to save resources and move on to more promising hyperparameter combinations.

5. Repeat: Optuna repeats the trial process multiple times (as defined by the user) until it finds the best set of hyperparameters or the predefined number of trials is completed.

1.0.3 How does Optuna Compare to Other Hyperparameter Optimization Methods?

Method	Pros	Cons
Grid Search	Exhaustively searches through predefined values.	Computationally expensive. Limited by pre-defined grid.
Random Search	Explores the search space randomly and is simpler than grid.	Inefficient for large search spaces. May miss optimal values.
Bayesian Search	More efficient by modeling the function being optimized.	Requires more complex setup. Slower for high-dimensional spaces.
Optuna	Define-by-run, efficient search with pruning and parallelism.	More complex to set up than simple random or grid search.

1.0.4 When should you use Optuna?

- **Hyperparameter Optimization for Complex Models**: If you are working with complex machine learning models like Gradient Boosting, Deep Neural Networks, or ensembles, where manual tuning is impractical.
- **Large Hyperparameter Search Space**: When the number of hyperparameters and possible values is large, Optuna's efficiency makes it superior to grid or random search.
- **Need for Resource Efficiency**: Optuna's ability to prune underperforming trials helps save computational time and resources, making it ideal for long-running tasks or expensive-to-evaluate models.

1.0.5 Who uses Optuna?

- **Machine Learning Engineers and Data Scientists**: Optuna is widely adopted by professionals working on machine learning projects where hyperparameter optimization is crucial for achieving high performance.

- **Deep Learning Researchers**: Researchers often use Optuna for neural network hyperparameter tuning, as it efficiently searches through learning rates, optimizers, and architectures.
- **Kaggle Competitors**: Competitors in machine learning competitions use Optuna to gain an edge by finding the best possible hyperparameters.

1.0.6 Key Features of Optuna:

1. Define-by-Run: Hyperparameter space is defined dynamically during the execution of the trials, allowing for flexibility in how you construct and explore the space.

2. Pruning: Automatically stops unpromising trials to save computation time and focus on better-performing hyperparameter combinations.

3. Parallelism: Optuna supports running multiple trials in parallel across different CPU or GPU resources, speeding up the search process.

4. Visualization: Provides built-in tools for visualizing optimization history, parameter importance, and more, making it easier to analyze the optimization process.

1.0.7 Advantages of Optuna:

- **Efficient Optimization**: Optuna uses state-of-the-art algorithms like TPE for efficient hyperparameter search, making it faster than random or grid search.
- **Dynamic Construction**: Unlike grid search, Optuna builds the hyperparameter space dynamically, making it more flexible.
- **Automatic Pruning**: Unpromising trials are stopped early, saving computation time. Parallelization: Optuna can parallelize trials to speed up the optimization process.
- **Visualization Tools**: Optuna includes tools to visualize the optimization process, which helps in understanding how hyperparameters impact model performance.

[34]:
```
import pandas as pd
import seaborn as sns
import optuna
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, root_root_mean_squared_error,␣
 ↪r2_score

healthexp = sns.load_dataset('healthexp')
healthexp = pd.get_dummies(healthexp)
healthexp
```

[34]:
```
   Year  Spending_USD  Life_Expectancy  Country_Canada  Country_France  \
0  1970       252.311             70.6           False           False
1  1970       192.143             72.2           False            True
2  1970       123.993             71.9           False           False
```

```
3    1970       150.437             72.0           False           False
4    1970       326.961             70.9           False           False
..    …            …            …             …             …
269  2020      6938.983             81.1           False           False
270  2020      5468.418             82.3           False            True
271  2020      5018.700             80.4           False           False
272  2020      4665.641             84.7           False           False
273  2020     11859.179             77.0           False           False

     Country_Germany  Country_Great Britain  Country_Japan  Country_USA
0               True                  False          False        False
1              False                  False          False        False
2              False                   True          False        False
3              False                  False           True        False
4              False                  False          False         True
..               …                      …             …           …
269             True                  False          False        False
270            False                  False          False        False
271            False                   True          False        False
272            False                  False           True        False
273            False                  False          False         True

[274 rows x 9 columns]
```

[27]:
```
X = healthexp.drop(['Life_Expectancy'], axis=1)
y= healthexp['Life_Expectancy']

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,␣
 ↪random_state=54)
rfr = GradientBoostingRegressor(random_state=34)
rfr.fit(X_train, y_train)
```

[27]:
```
GradientBoostingRegressor(random_state=34)
```

[28]:
```
y_pred = rfr.predict(X_test)
y_pred
```

[28]:
```
array([78.39157822, 79.56433878, 77.29122733, 79.10343937, 80.96445194,
       77.29122733, 81.03357969, 81.0478605 , 73.89333767, 77.36302545,
       81.88363916, 77.31394029, 71.47882512, 74.82462518, 82.95075741,
       78.07791025, 71.37073548, 81.34544275, 73.48856096, 80.50574084,
       75.02644369, 80.19365241, 73.22967666, 78.54255466, 77.87168267,
       76.14733657, 73.14057728, 82.16757383, 74.42529019, 76.12931714,
       76.30302906, 81.41008928, 76.11675626, 78.73694658, 78.5494218 ,
       82.64753618, 81.09723669, 76.11675626, 77.89083117, 76.6816684 ,
       79.75257164, 84.16434938, 74.41614983, 78.72096389, 82.16529143,
       84.03429896, 79.14171978, 80.44485511, 81.71684243, 81.2197126 ,
        80.93125484, 79.89895388, 78.76478282, 78.28311191, 79.3784097 ])
```

[29]:
```
mean_absolute_error(y_test, y_pred)
```

[29]:
```
0.2709170644443816
```

[30]:
```
root_root_mean_squared_error(y_test, y_pred)
```

[30]:
```
0.355971309502974
```

[31]:
```
r2_score(y_test, y_pred)
```

[31]:
```
0.9866397423796623
```

[32]:
```
def objective(trial):
    n_estimators = trial.suggest_int('n_estimators', 50, 300)
    learning_rate = trial.suggest_float('learning_rate', 0.01, 0.3)
    max_depth = trial.suggest_int('max_depth', 2, 10)
    subsample = trial.suggest_float('subsample', 0.5, 1.0)

    # Define the model
    model = GradientBoostingRegressor(
        n_estimators=n_estimators,
        learning_rate=learning_rate,
        max_depth=max_depth,
        subsample=subsample,
        random_state=42
    )

    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    mse = root_root_mean_squared_error(y_test, y_pred)

    return mse

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=100)

print(f"Best hyperparameters: {study.best_params}")
```

```
[I 2024-10-20 11:55:27,700] A new study created in memory with name: no-
name-96fbbaf2-3f29-44b6-9cc6-00f7be9dc2b3
[I 2024-10-20 11:55:27,950] Trial 0 finished with value: 0.32519704881303185 and
parameters: {'n_estimators': 297, 'learning_rate': 0.267593117461795,
'max_depth': 9, 'subsample': 0.7706360903802824}. Best is trial 0 with value:
0.32519704881303185.
[I 2024-10-20 11:55:28,018] Trial 1 finished with value: 0.32933339451026583 and
```

```
parameters: {'n_estimators': 95, 'learning_rate': 0.2589727113442577,
'max_depth': 5, 'subsample': 0.8863003094336444}. Best is trial 0 with value:
0.32519704881303185.
[I 2024-10-20 11:55:28,338] Trial 2 finished with value: 0.34532580713847005 and
parameters: {'n_estimators': 279, 'learning_rate': 0.21427980221020418,
'max_depth': 9, 'subsample': 0.9493276445660934}. Best is trial 0 with value:
0.32519704881303185.
[I 2024-10-20 11:55:28,597] Trial 3 finished with value: 0.337141751498068 and
parameters: {'n_estimators': 288, 'learning_rate': 0.13182683300401327,
'max_depth': 8, 'subsample': 0.8868435801052132}. Best is trial 0 with value:
0.32519704881303185.
[I 2024-10-20 11:55:28,797] Trial 4 finished with value: 0.3124102730275322 and
parameters: {'n_estimators': 274, 'learning_rate': 0.20260187408469205,
'max_depth': 5, 'subsample': 0.9062239337295626}. Best is trial 4 with value:
0.3124102730275322.
[I 2024-10-20 11:55:28,974] Trial 5 finished with value: 0.33315273771793247 and
parameters: {'n_estimators': 218, 'learning_rate': 0.028634699851232688,
'max_depth': 7, 'subsample': 0.9546279936392901}. Best is trial 4 with value:
0.3124102730275322.
[I 2024-10-20 11:55:29,152] Trial 6 finished with value: 0.3531645162647396 and
parameters: {'n_estimators': 256, 'learning_rate': 0.2844066049483992,
'max_depth': 6, 'subsample': 0.6782931822279332}. Best is trial 4 with value:
0.3124102730275322.
[I 2024-10-20 11:55:29,264] Trial 7 finished with value: 0.3257614028015756 and
parameters: {'n_estimators': 151, 'learning_rate': 0.19935313467618743,
'max_depth': 6, 'subsample': 0.7203121769063907}. Best is trial 4 with value:
0.3124102730275322.
[I 2024-10-20 11:55:29,363] Trial 8 finished with value: 0.5949002296375471 and
parameters: {'n_estimators': 184, 'learning_rate': 0.023014559915282794,
'max_depth': 2, 'subsample': 0.7777546955416367}. Best is trial 4 with value:
0.3124102730275322.
[I 2024-10-20 11:55:29,414] Trial 9 finished with value: 0.44110445545375904 and
parameters: {'n_estimators': 70, 'learning_rate': 0.04678877022641999,
'max_depth': 4, 'subsample': 0.642435911988464}. Best is trial 4 with value:
0.3124102730275322.
[I 2024-10-20 11:55:29,559] Trial 10 finished with value: 0.283432180253704 and
parameters: {'n_estimators': 233, 'learning_rate': 0.1213112051937812,
'max_depth': 3, 'subsample': 0.5246775038673241}. Best is trial 10 with value:
0.283432180253704.
[I 2024-10-20 11:55:29,727] Trial 11 finished with value: 0.2733841950906861 and
parameters: {'n_estimators': 229, 'learning_rate': 0.11446729346660696,
'max_depth': 3, 'subsample': 0.5082849682777049}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:29,882] Trial 12 finished with value: 0.33738549688910224
and parameters: {'n_estimators': 221, 'learning_rate': 0.1142162775589909,
'max_depth': 2, 'subsample': 0.512223578308002}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,016] Trial 13 finished with value: 0.2821326667996498 and
```

```
parameters: {'n_estimators': 225, 'learning_rate': 0.08900454928810904,
'max_depth': 3, 'subsample': 0.5104345068793334}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,118] Trial 14 finished with value: 0.2936606123578851 and
parameters: {'n_estimators': 153, 'learning_rate': 0.07506187156406356,
'max_depth': 4, 'subsample': 0.582380490793184}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,236] Trial 15 finished with value: 0.2971268863998212 and
parameters: {'n_estimators': 186, 'learning_rate': 0.08185626842879058,
'max_depth': 3, 'subsample': 0.5830011523959723}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,386] Trial 16 finished with value: 0.2784443164517285 and
parameters: {'n_estimators': 244, 'learning_rate': 0.16865472566121278,
'max_depth': 3, 'subsample': 0.5847662153259866}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,549] Trial 17 finished with value: 0.32945547373396866
and parameters: {'n_estimators': 251, 'learning_rate': 0.16414146586098646,
'max_depth': 4, 'subsample': 0.5904421244601817}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,662] Trial 18 finished with value: 0.3243825595212696 and
parameters: {'n_estimators': 195, 'learning_rate': 0.15154369262583414,
'max_depth': 2, 'subsample': 0.6578053194559447}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,797] Trial 19 finished with value: 0.352405750294523 and
parameters: {'n_estimators': 124, 'learning_rate': 0.17254249831711935,
'max_depth': 10, 'subsample': 0.5615022984441036}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:30,950] Trial 20 finished with value: 0.36472528222929507
and parameters: {'n_estimators': 206, 'learning_rate': 0.22891639396218252,
'max_depth': 5, 'subsample': 0.8154315863946258}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:31,100] Trial 21 finished with value: 0.3073020016778538 and
parameters: {'n_estimators': 250, 'learning_rate': 0.08462566454055234,
'max_depth': 3, 'subsample': 0.5022795336584275}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:31,240] Trial 22 finished with value: 0.28500727995486796
and parameters: {'n_estimators': 238, 'learning_rate': 0.09996116056277207,
'max_depth': 3, 'subsample': 0.5454805945154205}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:31,359] Trial 23 finished with value: 0.3241893163539905 and
parameters: {'n_estimators': 162, 'learning_rate': 0.06051969627768748,
'max_depth': 4, 'subsample': 0.6320900822024877}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:31,508] Trial 24 finished with value: 0.3340743120889516 and
parameters: {'n_estimators': 269, 'learning_rate': 0.14404392207024466,
'max_depth': 2, 'subsample': 0.6182634714905018}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:31,646] Trial 25 finished with value: 0.28898722310627706
```

```
and parameters: {'n_estimators': 215, 'learning_rate': 0.18325206586845688,
'max_depth': 3, 'subsample': 0.6968492329500976}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:31,805] Trial 26 finished with value: 0.2859771349934609 and
parameters: {'n_estimators': 233, 'learning_rate': 0.11248707230834971,
'max_depth': 5, 'subsample': 0.5452022752235064}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:31,942] Trial 27 finished with value: 0.31601034658811294
and parameters: {'n_estimators': 199, 'learning_rate': 0.09525980881503929,
'max_depth': 4, 'subsample': 0.6028687123419023}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:32,049] Trial 28 finished with value: 0.3411566621028202 and
parameters: {'n_estimators': 170, 'learning_rate': 0.1377780919990122,
'max_depth': 2, 'subsample': 0.5441643781159113}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:32,266] Trial 29 finished with value: 0.3058034089127453 and
parameters: {'n_estimators': 293, 'learning_rate': 0.06232785721282394,
'max_depth': 7, 'subsample': 0.5007527369849425}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:32,354] Trial 30 finished with value: 0.34296501275487 and
parameters: {'n_estimators': 127, 'learning_rate': 0.239036660636862,
'max_depth': 3, 'subsample': 0.750523188775213}. Best is trial 11 with value:
0.2733841950906861.
[I 2024-10-20 11:55:32,498] Trial 31 finished with value: 0.2687702975135457 and
parameters: {'n_estimators': 233, 'learning_rate': 0.12255598931314905,
'max_depth': 3, 'subsample': 0.5309087953054625}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:32,650] Trial 32 finished with value: 0.2911019907959925 and
parameters: {'n_estimators': 255, 'learning_rate': 0.10341750383000117,
'max_depth': 3, 'subsample': 0.5610610466342697}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:32,813] Trial 33 finished with value: 0.2939611647216501 and
parameters: {'n_estimators': 235, 'learning_rate': 0.16223175960762215,
'max_depth': 4, 'subsample': 0.999010386111054}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:32,964] Trial 34 finished with value: 0.344139250148219 and
parameters: {'n_estimators': 269, 'learning_rate': 0.13409401544100277,
'max_depth': 2, 'subsample': 0.521813772821312}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:33,109] Trial 35 finished with value: 0.3082259462703164 and
parameters: {'n_estimators': 212, 'learning_rate': 0.04311457745659346,
'max_depth': 5, 'subsample': 0.5707021871807052}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:33,287] Trial 36 finished with value: 0.2845700970493313 and
parameters: {'n_estimators': 224, 'learning_rate': 0.1307828538799112,
'max_depth': 3, 'subsample': 0.6103923191274128}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:33,487] Trial 37 finished with value: 0.31574363020947704
```

```
and parameters: {'n_estimators': 284, 'learning_rate': 0.18707784889203227,
'max_depth': 5, 'subsample': 0.5353396695041222}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:33,724] Trial 38 finished with value: 0.35073747410206124
and parameters: {'n_estimators': 264, 'learning_rate': 0.15184723264528946,
'max_depth': 9, 'subsample': 0.8315364287992428}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:33,877] Trial 39 finished with value: 0.3071835616924356 and
parameters: {'n_estimators': 238, 'learning_rate': 0.12409019896753956,
'max_depth': 4, 'subsample': 0.6574995767812493}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,029] Trial 40 finished with value: 0.4677086522347409 and
parameters: {'n_estimators': 194, 'learning_rate': 0.012764854978969356,
'max_depth': 6, 'subsample': 0.7053418368479953}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,173] Trial 41 finished with value: 0.28897035646844416
and parameters: {'n_estimators': 243, 'learning_rate': 0.1181888403545856,
'max_depth': 3, 'subsample': 0.5221660145614726}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,301] Trial 42 finished with value: 0.32736784804216373
and parameters: {'n_estimators': 222, 'learning_rate': 0.10679544639401044,
'max_depth': 2, 'subsample': 0.5305061638806614}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,463] Trial 43 finished with value: 0.29762694215456287
and parameters: {'n_estimators': 279, 'learning_rate': 0.06518549601844656,
'max_depth': 3, 'subsample': 0.5031059382923468}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,610] Trial 44 finished with value: 0.26878256364376213
and parameters: {'n_estimators': 228, 'learning_rate': 0.09139112662408969,
'max_depth': 4, 'subsample': 0.5477457735008036}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,751] Trial 45 finished with value: 0.29143930744339114
and parameters: {'n_estimators': 205, 'learning_rate': 0.08965053455365322,
'max_depth': 4, 'subsample': 0.5685473731521724}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,885] Trial 46 finished with value: 0.39313205257246503
and parameters: {'n_estimators': 225, 'learning_rate': 0.046345326038113074,
'max_depth': 2, 'subsample': 0.5938485089230439}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:34,947] Trial 47 finished with value: 0.3396645631610075 and
parameters: {'n_estimators': 56, 'learning_rate': 0.0731028791459466,
'max_depth': 6, 'subsample': 0.5526624348425062}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:35,071] Trial 48 finished with value: 0.312967918752511 and
parameters: {'n_estimators': 178, 'learning_rate': 0.20356881669983035,
'max_depth': 4, 'subsample': 0.6369500029479468}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:35,231] Trial 49 finished with value: 0.2833206390868708 and
```

```
parameters: {'n_estimators': 262, 'learning_rate': 0.14482048416305823,
'max_depth': 3, 'subsample': 0.5772907570360203}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:35,394] Trial 50 finished with value: 0.3254244477042977 and
parameters: {'n_estimators': 244, 'learning_rate': 0.29049585313894283,
'max_depth': 5, 'subsample': 0.5184295143589501}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:35,554] Trial 51 finished with value: 0.28673224445533974
and parameters: {'n_estimators': 262, 'learning_rate': 0.17198550686886205,
'max_depth': 3, 'subsample': 0.5617611851623718}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:35,709] Trial 52 finished with value: 0.2910215237132724 and
parameters: {'n_estimators': 256, 'learning_rate': 0.14777145647614381,
'max_depth': 3, 'subsample': 0.5884989110350857}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:35,841] Trial 53 finished with value: 0.33663834445763846
and parameters: {'n_estimators': 227, 'learning_rate': 0.12351040479543017,
'max_depth': 2, 'subsample': 0.5352066137495815}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:36,044] Trial 54 finished with value: 0.29805083664401705
and parameters: {'n_estimators': 297, 'learning_rate': 0.09354034571522202,
'max_depth': 4, 'subsample': 0.6228478739475506}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:36,176] Trial 55 finished with value: 0.3186550196558229 and
parameters: {'n_estimators': 212, 'learning_rate': 0.16147731103371912,
'max_depth': 3, 'subsample': 0.5820879890308186}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:36,323] Trial 56 finished with value: 0.33128018709998036
and parameters: {'n_estimators': 247, 'learning_rate': 0.14010660706178468,
'max_depth': 2, 'subsample': 0.6544211258426351}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:36,537] Trial 57 finished with value: 0.28831754387250763
and parameters: {'n_estimators': 275, 'learning_rate': 0.07626085785849052,
'max_depth': 4, 'subsample': 0.6799987666663414}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:36,702] Trial 58 finished with value: 0.320019796192551 and
parameters: {'n_estimators': 190, 'learning_rate': 0.10540528737668065,
'max_depth': 8, 'subsample': 0.5130929258940896}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:36,884] Trial 59 finished with value: 0.3431264075744974 and
parameters: {'n_estimators': 260, 'learning_rate': 0.1846341094850861,
'max_depth': 3, 'subsample': 0.6059995367785499}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:37,009] Trial 60 finished with value: 0.36218200880182144
and parameters: {'n_estimators': 206, 'learning_rate': 0.2706652855831479,
'max_depth': 2, 'subsample': 0.5489548662054436}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:37,151] Trial 61 finished with value: 0.27663318682392724
```

```
and parameters: {'n_estimators': 232, 'learning_rate': 0.11744139207969508,
'max_depth': 3, 'subsample': 0.5258295173126227}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:37,291] Trial 62 finished with value: 0.282327356204379 and
parameters: {'n_estimators': 232, 'learning_rate': 0.11441097886689146,
'max_depth': 3, 'subsample': 0.5802111287236595}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:37,434] Trial 63 finished with value: 0.2846683813869978 and
parameters: {'n_estimators': 230, 'learning_rate': 0.10964627012259157,
'max_depth': 3, 'subsample': 0.5343856984399099}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:37,581] Trial 64 finished with value: 0.29354989451107105
and parameters: {'n_estimators': 216, 'learning_rate': 0.12765983353767738,
'max_depth': 4, 'subsample': 0.5021350240116306}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:37,720] Trial 65 finished with value: 0.35948926495929134
and parameters: {'n_estimators': 240, 'learning_rate': 0.08687325387700343,
'max_depth': 2, 'subsample': 0.5508074120453469}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:37,876] Trial 66 finished with value: 0.2980607734880508 and
parameters: {'n_estimators': 249, 'learning_rate': 0.0971489217202378,
'max_depth': 3, 'subsample': 0.5213821830496556}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:38,027] Trial 67 finished with value: 0.2757023560730843 and
parameters: {'n_estimators': 199, 'learning_rate': 0.11547805321778493,
'max_depth': 4, 'subsample': 0.872277198440613}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:38,166] Trial 68 finished with value: 0.3212843996999328 and
parameters: {'n_estimators': 181, 'learning_rate': 0.08147559495991766,
'max_depth': 5, 'subsample': 0.9404810225597366}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:38,339] Trial 69 finished with value: 0.2998468920089925 and
parameters: {'n_estimators': 199, 'learning_rate': 0.13222337419778468,
'max_depth': 4, 'subsample': 0.8114060461385173}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:38,511] Trial 70 finished with value: 0.2936321850898209 and
parameters: {'n_estimators': 217, 'learning_rate': 0.09959855855692458,
'max_depth': 4, 'subsample': 0.8906174319977933}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:38,663] Trial 71 finished with value: 0.3031109360473692 and
parameters: {'n_estimators': 229, 'learning_rate': 0.11676696430986833,
'max_depth': 3, 'subsample': 0.8394740356203501}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:38,812] Trial 72 finished with value: 0.30506979109723376
and parameters: {'n_estimators': 235, 'learning_rate': 0.11429279951338023,
'max_depth': 3, 'subsample': 0.7812357657877741}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:38,955] Trial 73 finished with value: 0.299335059932847 and
```

```
parameters: {'n_estimators': 204, 'learning_rate': 0.06796442040304482,
'max_depth': 4, 'subsample': 0.5623236135973059}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:39,103] Trial 74 finished with value: 0.27726456031107777
and parameters: {'n_estimators': 221, 'learning_rate': 0.12350693793514392,
'max_depth': 3, 'subsample': 0.5372381389445163}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:39,259] Trial 75 finished with value: 0.3197026616693903 and
parameters: {'n_estimators': 168, 'learning_rate': 0.15526663355775777,
'max_depth': 7, 'subsample': 0.9211213691341871}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:39,393] Trial 76 finished with value: 0.3610721016867259 and
parameters: {'n_estimators': 221, 'learning_rate': 0.05720600204864207,
'max_depth': 2, 'subsample': 0.8570779651846874}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:39,529] Trial 77 finished with value: 0.3017856322375178 and
parameters: {'n_estimators': 212, 'learning_rate': 0.1742764965575541,
'max_depth': 3, 'subsample': 0.5374811319067782}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:39,603] Trial 78 finished with value: 0.28771354036692975
and parameters: {'n_estimators': 89, 'learning_rate': 0.1369162710195497,
'max_depth': 4, 'subsample': 0.5089338457782696}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:39,758] Trial 79 finished with value: 0.36001397984272665
and parameters: {'n_estimators': 251, 'learning_rate': 0.12366063549935136,
'max_depth': 2, 'subsample': 0.7398744705249802}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:39,909] Trial 80 finished with value: 0.28039318485554693
and parameters: {'n_estimators': 241, 'learning_rate': 0.09222678665030536,
'max_depth': 3, 'subsample': 0.5174340187345003}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:40,055] Trial 81 finished with value: 0.2949761769037094 and
parameters: {'n_estimators': 243, 'learning_rate': 0.08205921520156348,
'max_depth': 3, 'subsample': 0.520242399964311}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:40,195] Trial 82 finished with value: 0.28748086601963063
and parameters: {'n_estimators': 221, 'learning_rate': 0.10383294564680376,
'max_depth': 3, 'subsample': 0.5508761620389389}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:40,380] Trial 83 finished with value: 0.28988642449517504
and parameters: {'n_estimators': 238, 'learning_rate': 0.09184546850695231,
'max_depth': 4, 'subsample': 0.5319059506207203}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:40,510] Trial 84 finished with value: 0.330846000770609 and
parameters: {'n_estimators': 197, 'learning_rate': 0.11003247880130926,
'max_depth': 3, 'subsample': 0.9959906722867444}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:40,638] Trial 85 finished with value: 0.2755188597168557 and
```

```
parameters: {'n_estimators': 209, 'learning_rate': 0.12065534816826647,
'max_depth': 3, 'subsample': 0.5137576808894949}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:40,784] Trial 86 finished with value: 0.3151901181133613 and
parameters: {'n_estimators': 208, 'learning_rate': 0.12034314215245714,
'max_depth': 5, 'subsample': 0.5105430357013842}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:40,904] Trial 87 finished with value: 0.3394515609516515 and
parameters: {'n_estimators': 186, 'learning_rate': 0.13250523083945367,
'max_depth': 2, 'subsample': 0.5689450129169772}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:41,042] Trial 88 finished with value: 0.279441527189532 and
parameters: {'n_estimators': 226, 'learning_rate': 0.15520164978259704,
'max_depth': 3, 'subsample': 0.5388918894107891}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:41,247] Trial 89 finished with value: 0.35259678053966315
and parameters: {'n_estimators': 227, 'learning_rate': 0.15566931509182913,
'max_depth': 10, 'subsample': 0.5940160321261414}. Best is trial 31 with value:
0.2687702975135457.
[I 2024-10-20 11:55:41,392] Trial 90 finished with value: 0.26756036736615935
and parameters: {'n_estimators': 216, 'learning_rate': 0.14322018406580803,
'max_depth': 4, 'subsample': 0.5406516005689751}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:41,535] Trial 91 finished with value: 0.2839898731388246 and
parameters: {'n_estimators': 218, 'learning_rate': 0.16538656442887342,
'max_depth': 4, 'subsample': 0.5557724344694196}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:41,669] Trial 92 finished with value: 0.27470824373319225
and parameters: {'n_estimators': 202, 'learning_rate': 0.14311660190415712,
'max_depth': 4, 'subsample': 0.5418480086606893}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:41,803] Trial 93 finished with value: 0.29370744684684014
and parameters: {'n_estimators': 202, 'learning_rate': 0.14440344299472405,
'max_depth': 4, 'subsample': 0.5284729481912918}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:41,950] Trial 94 finished with value: 0.28045739215083354
and parameters: {'n_estimators': 209, 'learning_rate': 0.1275406340731873,
'max_depth': 5, 'subsample': 0.5477745710819973}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:42,075] Trial 95 finished with value: 0.29977071364481417
and parameters: {'n_estimators': 191, 'learning_rate': 0.1382706441772703,
'max_depth': 4, 'subsample': 0.5019503850286139}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:42,225] Trial 96 finished with value: 0.29074368761970637
and parameters: {'n_estimators': 212, 'learning_rate': 0.14685314197335717,
'max_depth': 4, 'subsample': 0.5280517474257275}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:42,409] Trial 97 finished with value: 0.319604857634209 and
```

```
parameters: {'n_estimators': 255, 'learning_rate': 0.11867606754400264,
'max_depth': 5, 'subsample': 0.5733044545999941}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:42,536] Trial 98 finished with value: 0.2857077027347594 and
parameters: {'n_estimators': 174, 'learning_rate': 0.12736296874721426,
'max_depth': 4, 'subsample': 0.5602314712383062}. Best is trial 90 with value:
0.26756036736615935.
[I 2024-10-20 11:55:42,691] Trial 99 finished with value: 0.2878060321930935 and
parameters: {'n_estimators': 234, 'learning_rate': 0.17614425422064806,
'max_depth': 3, 'subsample': 0.5402212509921593}. Best is trial 90 with value:
0.26756036736615935.

Best hyperparameters: {'n_estimators': 216, 'learning_rate':
0.14322018406580803, 'max_depth': 4, 'subsample': 0.5406516005689751}
```

[33]:
```python
study.best_params
```

[33]:
```
{'n_estimators': 216,
 'learning_rate': 0.14322018406580803,
 'max_depth': 4,
 'subsample': 0.5406516005689751}
```

[35]:
```python
optuna.visualization.plot_optimization_history(study)
```

[36]:
```python
optuna.visualization.plot_parallel_coordinate(study)
```

[38]:
```python
optuna.visualization.plot_slice(study, params=['n_estimators', 'learning_rate',␣
↪'max_depth', 'subsample'])
```

[40]:
```python
optuna.visualization.plot_param_importances(study)
```

30-K_Means_Clustering

October 20, 2024

1 K-Means Clustering

K-Means Clustering is one of the simplest and most widely used unsupervised learning algorithms for partitioning a dataset into distinct groups or clusters. In K-Means, each cluster is represented by a centroid, and each data point is assigned to the cluster with the closest centroid.

The key idea behind K-Means is to partition data into k clusters, where k is a predefined number. The algorithm works iteratively to assign data points to one of the k clusters based on the similarity (often measured by Euclidean distance).

1.0.1 Why do we use K-Means Clustering?

K-Means Clustering is highly popular due to:

- **Simplicity**: Easy to understand and implement.
- **Speed**: Efficient for clustering large datasets.
- **Applicability**: Useful for a variety of tasks such as customer segmentation, image compression, and document clustering.
- **Interpretability**: Results are easy to interpret, as each data point belongs to exactly one cluster.

1.0.2 How does K-Means Clustering work?

The algorithm works in the following steps:

Initialization:

Choose the number of clusters k. Randomly initialize k centroids (cluster centers) in the feature space. Assignment (Step 1):

For each data point in the dataset, calculate its distance to each of the k centroids. Assign each data point to the nearest centroid (based on minimum distance). Update Centroids (Step 2):

After the assignment step, recompute the centroids by calculating the mean of all data points assigned to each cluster. The new centroid will be the new center of the cluster. Repeat:

Repeat the Assignment and Update steps iteratively until the centroids do not change significantly, or until a predefined number of iterations is reached. This is called convergence. Output:

The algorithm produces k clusters, each with a centroid and a set of assigned data points.

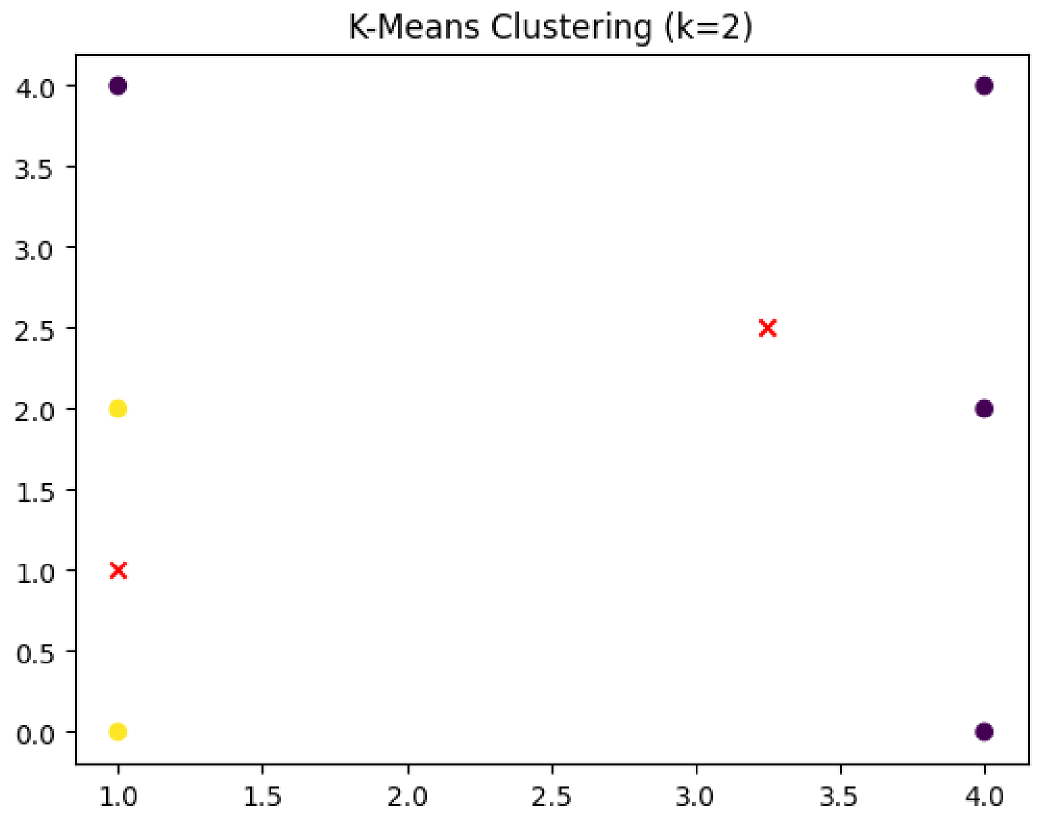
1.0.3 When to use K-Means Clustering?

K-Means is best suited for problems where:

You know the number of clusters (k): K-Means requires you to specify k in advance. Clusters are spherical and equally sized: K-Means assumes clusters are compact, and all of approximately equal size. There are no significant outliers: K-Means is sensitive to outliers, as they can heavily skew the centroid calculation. Common use cases include:

Customer Segmentation: Group customers with similar buying behaviors. Image Compression: Reduce image size by clustering similar pixels. Document Clustering: Group similar documents based on their content.

[1]:
```python
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt

X = np.array([[1, 2], [1, 4], [1, 0],
              [4, 2], [4, 4], [4, 0]])

kmeans = KMeans(n_clusters=2, random_state=0)

kmeans.fit(X)
labels = kmeans.predict(X)

centroids = kmeans.cluster_centers_

# Visualize the clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x')
plt.title("K-Means Clustering (k=2)")
plt.show()
```



1.0.4 Advantages of K-Means Clustering:

Scalability: Can handle large datasets efficiently. Easy to implement: Straightforward in both understanding and coding. Fast convergence: The iterative approach usually converges quickly.

1.0.5 Disadvantages of K-Means Clustering:

Choosing k: The number of clusters (k) must be pre-specified, and determining the optimal k can be challenging. Sensitive to initialization: Different random initializations can lead to different clustering results (though you can mitigate this by running the algorithm multiple times or using the k-means++ initialization method). Sensitive to outliers: Outliers can significantly affect cluster centroids. Assumption of spherical clusters: K-Means works best when clusters are of equal size and spherical in shape.

1.0.6 Who uses K-Means Clustering?

- **Marketers**: To segment customers based on purchasing patterns.
- **Biologists**: To classify genes with similar characteristics.
- **Image Analysts**: To compress images or classify regions in an image.
- **Text Miners**: To group similar documents or articles.