Fundamentals of Optimization for Supervised Machine Learning Models

Biynah Ayobami Dayok
LeMoyne-Owen College
Department of Computer Science

November 10, 2024

Contents

Abstract

This article provides an overview of supervised machine learning models, which are essential tools for discovering patterns and making decisions from unseen datasets. We examine the significance of these models in diverse applications. The article emphasizes the importance of high-quality datasets in the training phase, highlighting how they enable models to generalize effectively.

We detail the training process, which involves using various machine learning algorithms, such as linear regression, decision trees, and support vector machines to optimize model parameters and enhance predictive accuracy. Key components of this process, including data preprocessing, feature selection, and model evaluation, are discussed to illustrate their roles in improving model performance.

By exploring the unique characteristics of different machine learning models and providing practical guidance on their implementation, this article aims to equip readers with the knowledge to leverage machine learning solutions effectively in real-world scenarios.

Introduction

A machine learning model is a set of programs designed to discover patterns and make decisions based on unseen datasets. These models play a crucial role in various applications, including image recognition, which enables systems to identify objects such as boys, girls, mirrors, cars, dogs, and more. The significance of machine learning extends beyond image recognition; it also encompasses fields like natural language processing, fraud detection, recommendation systems, and autonomous vehicles, illustrating its versatility and impact on modern technology.

Each model relies on a dataset to perform its tasks effectively during the training phase. The quality and quantity of data are vital, as they directly influence the models performance. A well-structured dataset allows the model to generalize better when faced with new, unseen data, ensuring reliable predictions.

Training a machine learning model involves using a machine learning algorithm to optimize its parameters. This optimization process helps the model learn and find specific patterns or outputs from the dataset, tailored to the tasks at hand. Common algorithms include linear regression, decision trees, support vector machines, and neural networks, each with its own strengths and weaknesses depending on the nature of the data and the problem being addressed.

In addition to the algorithm, the training process typically includes steps like data preprocessing, feature selection, and model evaluation. Data preprocessing ensures that the data is clean and formatted correctly, while feature selection identifies the most relevant attributes that contribute to the model's predictive power. Model evaluation, often conducted through techniques like cross-validation, assesses the model's performance and helps in fine-tuning its parameters.

In this article, we will explore various machine learning models, delve into their unique characteristics, and provide guidance on how to utilize and work with them effectively. By understanding these models and their applications, readers will gain insights into how to implement machine learning solutions in real-world scenarios. [GeeksForGeeks(2024)].

Chapter 1

SUPERVISED VS UNSUPERVISED MACHINE LEARNING

What's a Label?

A label is simply a name or tag you give to something that tells the computer what it is. If you have a picture of an apple, the label is the word "apple." It's like writing the answer on a flashcard. The computer uses the labels to learn what things are and make better guesses next time.

Supervised Learning:

Supervised learning is like teaching a child to recognize different animals. You show pictures of animals and tell the child the name of each one. In this case, the label is the name you give for each picture, like "cat" or "dog." The computer uses these labels to learn what a cat looks like compared to a dog. The label tells the computer the correct answer for each example.

There are two main tasks in supervised learning:

1. Classification: Putting things into groups. For example, sorting fruits into "apple" or "banana" based on their features. Here, the labels are the words "apple" and "banana."

2. Regression: Predicting a number based on other information. For instance, guessing a person's age based on their height and weight. The label would be the actual age.

This article primarily focuses on supervised models, but let's also explore unsupervised models to gain a more comprehensive understanding.

Unsupervised Learning:

Unsupervised learning is when you let the computer find patterns by itself. It's like giving the computer a bunch of photos without telling it what's in them. The computer tries to sort them into groups on its own, even though it doesn't know what the pictures show. There are no labels here to guide it.

The main tasks are:

1. Clustering: Grouping things that seem similar. For example, a computer might put photos of flowers into different groups based on color or shape, even if it doesn't know their names.

2. Association: Finding things that often go together. For instance, noticing that people who buy milk often buy bread too.

3. Dimensionality Reduction: Simplifying the data by keeping only the important parts, like summarizing a long story to its main points.

Key Difference:

Supervised Learning has labels, meaning the computer is told what each example is. Unsupervised Learning has no labels, so the computer tries to figure out patterns on its own.

Choosing Which to Use:

Use supervised learning if you know the correct answers for your data, like predicting grades based on study hours. Use unsupervised learning if you don't have any answers and want to explore the data to find patterns.

Semi-Supervised Learning:

This uses a mix of labeled and unlabeled examples. Imagine a few pictures are labeled "cat," but many others aren't labeled. The computer can learn from the labeled ones to help identify

what the unlabeled pictures might be. [IBM(2021)]

Chapter 2

REGRESSION MODELS

A regression model provides a function that describes the relationship between one or more independent variables and a response, dependent, or target variable. [IMSL(2021)]

2.1 Types of Regression Models

Although numerous regression models are available, this introductory book will limit its scope to an examination of only three types:

2.1.1 Linear Regression

Linear regression is a foundational technique in statistics and machine learning for predicting the relationship between a dependent variable (often called the target) and one or more independent variables (called predictors). The model fits a line to the data points in such a way that it minimizes prediction errors, allowing us to make accurate predictions for new data. Here, we'll dive into why linear regression is useful, when it is best applied, and how it works with practical examples and code for demonstration.

Why Use Linear Regression? Linear regression is commonly used for its simplicity and interpretability. When we believe there is a linear relationship between variables, linear regression becomes useful. For example, in predicting housing prices based on features like square footage or number of bedrooms, linear regres-

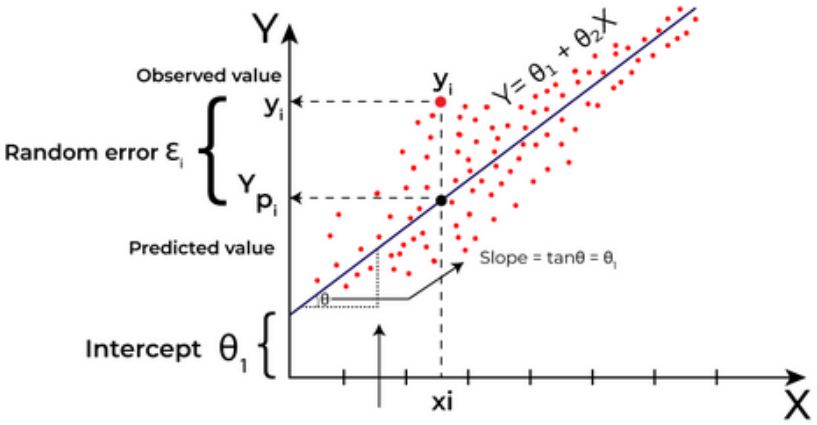

Figure 2.1: Visual representation of Hierarchical clustering. [GeeksForGeeks(2024)]

sion provides a straightforward way to quantify the relationship between these features and the price.

When to Use Linear Regression

Linear regression is a good choice when:

1. Linearity: There is a straight-line relationship between your input variables (predictors) and the output variable (target). This means that as one changes, the other tends to change in a consistent way.

2. Independence: Each data point is separate and does not influence any other data point. This ensures that the observations are independent of each other.

3. Normal Distribution: The differences between what your model predicts and the actual values (called residuals) should follow a normal distribution. This means most of the errors are small, with fewer large errors.

4. Homoscedasticity: The spread of the residuals should be consistent across all values of the predictors. In simpler terms, the size of the errors should not change based on the value of the predictors.

Using linear regression under these conditions helps ensure that your model is accurate and reliable.

How Linear Regression Works The idea behind linear regression is to fit a line (or hyperplane, in higher dimensions) that minimizes the prediction error for each data point. This is achieved by minimizing the cost function, usually Mean Squared Error (MSE), which measures the average squared difference between the observed and predicted values. For simple linear regression with one predictor, the formula is:

$$y = mx + c$$

where:
y is the predicted value,
m is the slope (change in yy per unit change in xx),
x is the input feature,
c is the intercept.

Example: Linear Regression Model with Python Code

We'll demonstrate a simple linear regression model that predicts house prices based on square footage. We'll split the data into training and testing sets, fit the model, make predictions, and visualize the results.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Sample data: square footage and house prices
data = pd.DataFrame({
        'SquareFootage': [1000, 1500, 2000, 2500, 3000],
        'Price': [200000, 250000, 300000, 350000, 400000]
})

# Defining predictor (X) and target (y)
X = data['SquareFootage'].values.reshape(-1, 1)
```

```
y = data['Price']

# Splitting into train and test sets with a smaller test size
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
    random_state=42)

# Fitting the model
lr = LinearRegression()
lr.fit(X_train, y_train)

# Making predictions on the test data
y_pred = lr.predict(X_test)

# Displaying results
print("Intercept:", lr.intercept_)
print("Slope:", lr.coef_[0])
print("RMSE:", np.sqrt(mean_squared_error(y_test, y_pred)))
print("R-squared:", r2_score(y_test, y_pred))

# Plotting the data points and the regression line
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', label="Actual Data") # Scatter plot of actual
    data
plt.plot(X, lr.predict(X), color='red', label="Regression Line") # Regression
    line
plt.xlabel("Square Footage")
plt.ylabel("Price")
plt.title("Linear Regression: Square Footage vs. Price")
plt.legend()
plt.show()
```

Explanation of the Code:

1. Data Definition: We create a simple dataset of house prices and their corresponding square footage.

2. Splitting the Data: The data is split into training and testing sets to evaluate the model's performance on unseen data.

3. Fitting the Model: We use LinearRegression from sklearn to fit the model on the training data.

4. Predictions and Evaluation: Predictions are made on the test data, and we calculate metrics like RMSE (Root Mean Squared Error) and R-squared to gauge accuracy.

5. Visualization: The model is visualized by plotting the regression line along with actual data points, helping to assess the fit visually.

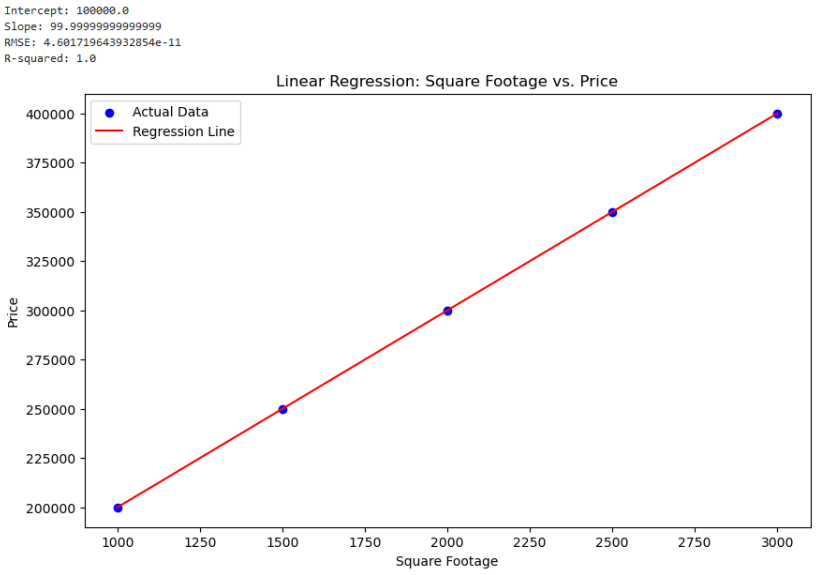

Figure 2.2: Visual representation of the Linear Regression Model

Interpreting Evaluation Metrics:

1. RMSE: This metric indicates the average error of predictions in the same units as the target variable, giving insight into prediction accuracy.

2. R-squared: This score indicates how much of the variance in the target variable is explained by the model; values closer to 1 implies a better fit.

Reducing Errors and Optimizing the Model

1. Feature Engineering: Selecting relevant features or creating new ones can improve model accuracy.

2. Regularization: You can use either Ridge or Lasso regression. Both are similar to linear regression but with an added penalty:

(a) Ridge: Adds a penalty for large squared coefficients, which shrinks them but doesn't remove any completely.

(b) Lasso: Adds a penalty based on the absolute value of coefficients, which can shrink some coefficients to zero, effectively removing those features.

```
from sklearn.linear_model import Ridge, Lasso

# Ridge Regression
ridge = Ridge(alpha=1.0) # Try values like 0.1, 1, or 10
ridge.fit(X_train, y_train)
y_pred_ridge = ridge.predict(X_test)

# Lasso Regression
lasso = Lasso(alpha=1.0) # Try different alpha values too
lasso.fit(X_train, y_train)
y_pred_lasso = lasso.predict(X_test)

# Evaluate results
print("Ridge RMSE:", np.sqrt(mean_squared_error(y_test, y_pred_ridge)))
print("Lasso RMSE:", np.sqrt(mean_squared_error(y_test, y_pred_lasso)))
```

3. Cross-validation: Cross-validation tests the model on different subsets of the data to see how it performs across various parts of your dataset. Instead of splitting data just once, it splits it multiple times (often 5 or 10 splits) and takes the average of these results. This helps get a more accurate view of the model's performance.

   How to do it:

   Use cross_val_score in scikit-learn to calculate scores over multiple splits. Here, we'll use negative mean squared error (MSE) to evaluate performance.

   Code Example:

```
from sklearn.model_selection import cross_val_score

# Perform cross-validation
scores = cross_val_score(lr, X, y, cv=5, scoring='neg_mean_squared_error'
    ) # 5-fold CV
print("Cross-Validated RMSE:", np.sqrt(-scores.mean())) # Convert
    negative MSE to positive RMSE
```

4. Hyperparameter Tuning: Hyperparameters are settings you can adjust to improve your model's performance. For Ridge and Lasso, the main hyperparameter is alpha (the penalty strength). Testing different values of alpha can help find the best setting for your model.

   How to do it:

   Use GridSearchCV to try multiple values of alpha automatically. It will test each value and tell you which one works best based on cross-validation.

```
from sklearn.model_selection import GridSearchCV

# Set up parameter grid
param_grid = {'alpha': [0.1, 1, 10]} # Try different alpha values

# Use GridSearchCV for Ridge
ridge_cv = GridSearchCV(Ridge(), param_grid, cv=5, scoring='
    neg_mean_squared_error')
ridge_cv.fit(X, y)
print("Best alpha for Ridge:", ridge_cv.best_params_['alpha'])

# Use GridSearchCV for Lasso
lasso_cv = GridSearchCV(Lasso(), param_grid, cv=5, scoring='
    neg_mean_squared_error')
lasso_cv.fit(X, y)
print("Best alpha for Lasso:", lasso_cv.best_params_['alpha'])
```

Conclusion

Linear regression is highly interpretable and useful for identifying simple linear relationships. By tuning features, optimizing the model, and validating with test data, linear regression can offer valuable insights for predictive analysis across various fields, from finance to healthcare. [AnalyticsVidhya(2024)]

2.1.2 Decision Trees

A Decision Tree is a graphical representation used for making decisions based on certain conditions. It consists of nodes, branches,

and leaves, where each node represents a feature, each branch indicates a decision rule, and each leaf signifies an outcome. Decision Trees can be used for both classification and regression tasks, allowing them to predict categorical and continuous outcomes.

Why Use Decision Trees?

Decision Trees are popular for several reasons:

1. Interpretability: Their intuitive structure allows for easy understanding and visualization of decision-making processes, making them accessible to non-experts.
2. Flexibility: They can handle both numerical and categorical data without requiring extensive preprocessing.
3. Non-parametric Nature: Decision Trees do not assume a specific distribution for the data, making them suitable for various applications.
4. Feature Importance: They can provide insights into the significance of different features in the decision-making process.

When to Use Decision Trees

Decision Trees are particularly beneficial in the following situations:

1. Complex Relationships: When the relationship between features and the target variable is non-linear or involves interactions.
2. Mixed Data Types: When working with datasets that include both categorical and numerical features.
3. Need for Clear Explanations: When stakeholders require transparent and understandable models for decision-making.

How Decision Trees Work

A Decision Tree works by repeatedly asking questions about the features in the data to split it into smaller, more organized groups. At each step, it finds the best question that divides the data so that data points with similar outcomes are grouped together. This process continues until the data is neatly separated, and the final groups are used to make predictions.

Example of Decision Trees

For example, consider a situation where we want to predict whether a student will pass or fail based on their hours of study and attendance. A Decision Tree might first split students based on attendance levels and then further divide them based on hours of study, ultimately leading to a prediction of pass or fail.

Example Code for Decision Trees

Here's a simple code snippet to create and visualize a Decision Tree model using Python:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

# Sample data
data = {
       'Hours_Study': [5, 10, 15, 20, 25],
       'Attendance': [80, 90, 95, 85, 70],
       'Pass_Fail': ['Fail', 'Pass', 'Pass', 'Pass', 'Fail']
}

# Create DataFrame
df = pd.DataFrame(data)

# Features and target variable
X = df[['Hours_Study', 'Attendance']]
y = df['Pass_Fail']

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Create Decision Tree model
```

```
model = DecisionTreeClassifier(max_depth=3)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))

# Visualize the Decision Tree
plt.figure(figsize=(10, 6))
plot_tree(model, filled=True, feature_names=X.columns, class_names=y.unique())
plt.show()
```

```
Accuracy: 1.0
              precision    recall  f1-score   support

        Pass       1.00      1.00      1.00         1

    accuracy                           1.00         1
   macro avg       1.00      1.00      1.00         1
weighted avg       1.00      1.00      1.00         1
```

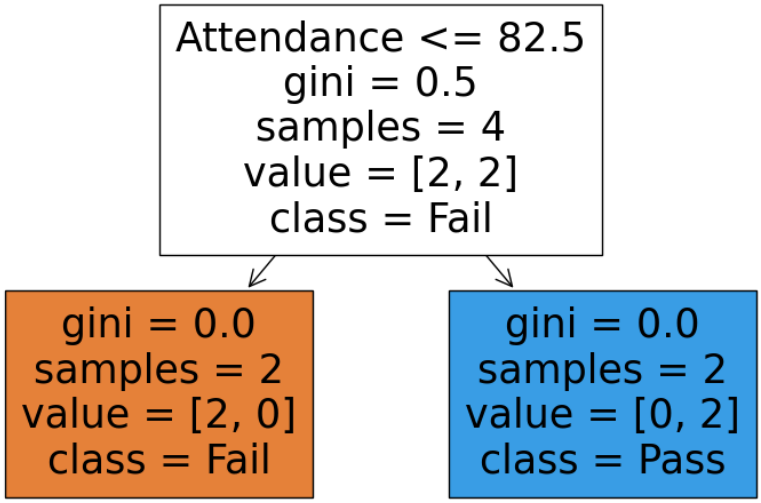

Figure 2.3: Visual representation of Hierarchical clustering.

Explanation of the Code

This code demonstrates how to use a Decision Tree Classifier in Python using the sklearn library. Let's walk through each part step-by-step:

1. Import Libraries:
    - pandas is used to handle and manipulate data in a tabular form.

- sklearn.model_selection provides tools for splitting the data into training and testing sets.
- sklearn.tree contains the Decision Tree Classifier and functions to visualize the tree.
- sklearn.metrics provides tools to evaluate the performance of the model.
- matplotlib.pyplot is used to visualize the Decision Tree.

2. Sample Data:
   - The data dictionary contains three features: Hours_Study, Attendance, and Pass_Fail.
   - Pass_Fail is the target variable that indicates whether a student passes or fails based on their study hours and attendance.
3. Create DataFrame:
   - The sample data is converted into a DataFrame called df using pandas.
4. Feature Selection:
   - X contains the features: Hours_Study and Attendance.
   - y is the target variable: Pass_Fail.
5. Split Data:
   - train_test_split is used to split the data into training (80
   - random_state=42 ensures that the split is reproducible.
6. Create Decision Tree Model:
   - DecisionTreeClassifier(max_depth=3) creates a Decision Tree with a maximum depth of 3, meaning the tree will not have more than 3 levels.

- model.fit(X_train, y_train) trains the model on the training data.

7. Make Predictions:
   - model.predict(X_test) generates predictions for the test data.
8. Evaluate the Model:
   - accuracy_score(y_test, y_pred) computes the accuracy of the model.
   - classification_report(y_test, y_pred) provides a detailed performance report, including precision, recall, and F1-score.
9. Visualize the Decision Tree:
   - plot_tree creates a visualization of the Decision Tree, showing how the features split to make predictions.
   - filled =True colors the nodes based on the class they represent.

Optimizing Decision Trees

1. Pruning: Pruning involves trimming branches of the tree that have little significance to improve model performance and reduce overfitting. It simplifies the model by preventing it from becoming too complex.

   Use min_samples_split: The minimum number of samples required to split an internal node. Increasing this value can prevent the model from creating very small splits that lead to overfitting.

```
model = DecisionTreeClassifier(min_samples_split=4)
```

Use min_samples_leaf: The minimum number of samples required to be at a leaf node. This can help reduce overfitting by ensuring leaves have enough samples.

```
model = DecisionTreeClassifier(min_samples_leaf=2)
```

2. Feature Selection: Choose relevant features that significantly impact the target variable while eliminating unnecessary ones to reduce complexity. Handling Missing Values: Address missing data through imputation or removal to ensure the integrity of the dataset.

3. Use Cross-Validation: Use cross-validation to tune hyperparameters and select the best model configuration. This involves splitting the data into multiple subsets and training/testing the model on these subsets.

```
from sklearn.model_selection import GridSearchCV

params = {
    'max_depth': [2, 3, 4, 5],
    'min_samples_split': [2, 4, 6],
    'min_samples_leaf': [1, 2, 3]
}
grid_search = GridSearchCV(DecisionTreeClassifier(), params, cv=5)
grid_search.fit(X_train, y_train)
```

Conclusion

Decision Trees are powerful tools that simplify complex decision-making processes through clear visualization and intuitive understanding. They are versatile in handling various data types and relationships, making them an essential part of the machine learning toolkit.

Key Takeaways

1. Decision Trees provide clear interpretability and flexibility for different data types.

2. Techniques such as pruning and hyperparameter tuning are crucial for optimizing performance and mitigating overfitting. [GeeksForGeeks(2024)]

2.1.3 Random Forest

What is a Random Forest?

Random Forest is a versatile and powerful machine learning algorithm designed to improve the accuracy and robustness of predictions, especially with complex or noisy data. It works well for both classification and regression problems and is particularly effective when data has numerous features or when there's a risk of overfitting in simpler models like Decision Trees.

For instance, consider a health dataset predicting heart disease risk based on factors like age, cholesterol level, and exercise frequency. A single Decision Tree could become biased or overly tailored to the training data. Random Forest, by creating a "forest" of multiple decision trees, reduces the chances of this bias, making the predictions more accurate and generalizable.

When to Use Random Forest?

Random Forest is best used when:

Data has high dimensionality (many features) and could be prone to overfitting. Complex interactions between features need capturing without much tuning. Interpretability is not the highest priority since it's more of a "black box" model compared to a single Decision Tree. Classification tasks with balanced classes, such as predicting user sentiment based on text analysis.

For instance, in predicting customer churn in a subscription service, Random Forest can effectively identify complex feature interactions without overfitting, helping to isolate which factors most influence a customer's decision to leave.

How Does Random Forest Work?

Random Forest works by:

Bootstrapping: It randomly samples data from the training set with replacement, creating multiple subsets. Training multiple Decision Trees: Each subset trains a Decision Tree on a random sample of features.

Ensembling predictions: The algorithm combines the results from each Decision Tree to make a final prediction.

Consider an e-commerce recommendation system. Using bootstrapping, Random Forest builds different decision trees based on user demographics, browsing history, and purchasing behavior. Each tree "votes" on whether a user will purchase an item, and the final recommendation is determined by the majority vote or average prediction from all trees.

Random Forest Example

Below is sample code implementing a Random Forest Classifier in Python:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
import pandas as pd

# Generate a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5,
    random_state=42)
feature_names = [f"Feature {i+1}" for i in range(X.shape[1])]

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)

# Initialize and train the Random Forest model
model = RandomForestClassifier(n_estimators=100, max_depth=7, random_state=42)
model.fit(X_train, y_train)

# Make predictions and calculate accuracy
predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print("Model Accuracy:", accuracy)
```

```
# Display feature importances
importances = model.feature_importances_
importance_df = pd.DataFrame({'Feature': feature_names, 'Importance':
    importances}).sort_values(by='Importance', ascending=False)

# Plotting feature importances
plt.figure(figsize=(10, 6))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='skyblue
    ')
plt.xlabel('Importance Score')
plt.title('Feature Importances in Random Forest Model')
plt.gca().invert_yaxis() # To have the most important feature at the top
plt.show()
```

Explanation:

1. Import Libraries

   - `RandomForestClassifier` from `sklearn.ensemble` is used to build the Random Forest model.
   - `train_test_split` from `sklearn.model_selection` is used to split the data into training and testing sets.
   - `accuracy_score` from `sklearn.metrics` is used to evaluate the model's performance.
   - `make_classification` from `sklearn.datasets` generates a synthetic dataset for demonstration purposes.
   - `matplotlib.pyplot` is used to visualize the feature importances.
   - `pandas` is used to manage the feature importance data in a structured way.

2. Generate Synthetic Dataset

   - `make_classification` creates a dataset with 1,000 samples and 10 features, with 5 features being informative. The `random_state=42` ensures consistent data generation every time the code is run.

- `feature_names` is a list that names each feature as "Feature 1", "Feature 2", etc., for readability.

3. Split the Data

   - `train_test_split` divides the dataset into a training set (70%) and a testing set (30%). The `random_state=42` ensures reproducibility.

4. Initialize and Train the Model

   - `RandomForestClassifier` creates a Random Forest model with 100 trees (`n_estimators=100`) and a maximum depth of 7 (`max_depth=7`). `random_state=42` ensures consistent behavior.
   - `model.fit(X_train, y_train)` trains the model using the training data.

5. Make Predictions and Calculate Accuracy

   - `model.predict(X_test)` makes predictions on the testing data.
   - `accuracy_score(y_test, predictions)` calculates the accuracy by comparing predictions to the actual labels.

6. Display Feature Importances

   - `model.feature_importances_` returns the importance of each feature, showing its contribution to the model.
   - `importance_df` creates a DataFrame to display features and their importance scores in descending order.

7. Plotting Feature Importances

   - A horizontal bar plot is created to visualize the importance scores using `matplotlib`.

- `plt.gca().invert_yaxis()` inverts the y-axis, showing the most important feature at the top.
- `plt.show()` displays the plot.

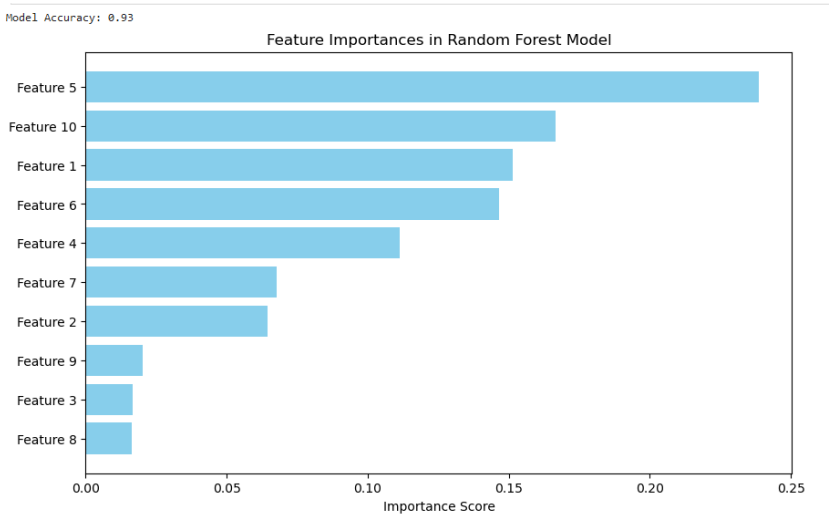

Figure 2.4: RANDOM FOREST.

Model Accuracy: This displays how well the model performs on the test set.

Feature Importances: The bar chart shows which features the Random Forest considered most valuable for making predictions. This visualization helps interpret the model and understand which factors are more influential.

With this code, you'll get both the accuracy output in the console and a visual display of feature importances, making the model's behavior and performance clearer. Let me know if you want to dive deeper into any part of the model analysis.

Optimizing Random Forest Models

To improve the performance of a Random Forest model like the one in your code, here are some steps to consider, along with code examples to illustrate each method.

1. Tune Hyperparameters

The default hyperparameters often do not yield the best results. Here's how you can tune key parameters in the RandomForestClassifier to improve performance:

- n_estimators: Increase the number of trees to improve accuracy but keep in mind that computation time will also increase.
- max_depth: Experiment with different maximum depths for the trees. Deeper trees capture more complexity, but overfitting can become an issue.
- max_features: Limit the number of features considered at each split, which can reduce overfitting and improve generalization.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
        'n_estimators': [100, 200, 300],
        'max_depth': [5, 7, 10, None],
        'max_features': ['auto', 'sqrt', 'log2']
}
grid_search = GridSearchCV(RandomForestClassifier(
    random_state=42), param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
model = grid_search.best_estimator_
```

2. Use Cross-Validation for Reliable Evaluation Rather than a single train-test split, using cross-validation helps in obtaining a more stable measure of model performance and reduces variance in results.

```
from sklearn.model_selection import cross_val_score

# Cross-validate the model
cv_scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
print("Cross-Validation Scores:", cv_scores)
print("Mean CV Accuracy:", cv_scores.mean())
```

3. Reduce Overfitting with Pruning Techniques

Random forests are inherently resistant to overfitting, but restricting tree depth or adjusting min_samples_split and min_samples_leaf can help in cases with high variance.

```
# Limit tree complexity by requiring more samples for a
    split
model_pruned = RandomForestClassifier(n_estimators=100,
    max_depth=7, min_samples_split=10, min_samples_leaf=5,
    random_state=42)
model_pruned.fit(X_train, y_train)
predictions = model_pruned.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print("Model Accuracy with Pruned Trees:", accuracy)
```

[AnalyticsVidhya(2024)]

Chapter 3

CLASSIFICATION MODELS

Classification is a supervised machine learning method where the model tries to predict the correct label of a given input data. In classification, the model is fully trained using the training data, and then it is evaluated on test data before being used to perform prediction on new unseen data.

For instance, an algorithm can learn to predict whether a given email is spam or ham (no spam), as illustrated below. [datacamp(2024)]

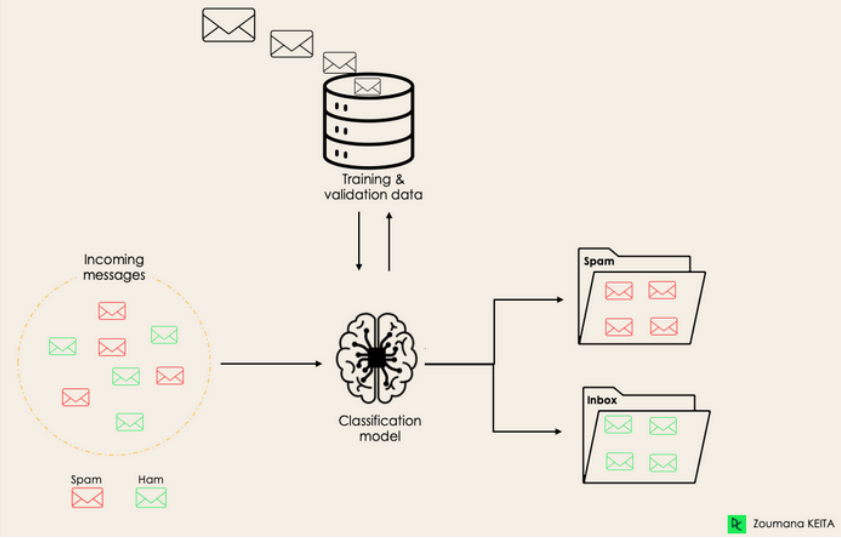

Figure 3.1: CLASSIFICATION MODELING.
[datacamp(2024)]

3.1 Types of Classification Models

While there are many types of classification models, this book focuses on two: logistic regression and support vector machines

3.1.1 Logistic Regression

What is Logistic Regression?

Logistic regression is a supervised machine learning algorithm used primarily for binary classification, predicting the probability that an instance belongs to a particular class. Although it builds on the principles of linear regression, logistic regression is suited for categorical outcomes, producing probability values between 0 and 1.

When to Use Logistic Regression

Logistic regression is ideal when:

1. Target variable is binary: For instance, it answers yes/no or 0/1 type questions.
2. Relationships are probabilistic: It outputs probabilities, making it useful for applications like fraud detection, spam filtering, and medical diagnoses.

How Logistic Regression Works

Logistic regression uses the sigmoid function (also known as the logistic function) to model probabilities:

$$z(\sigma) = \frac{1}{1 + e^{-z}}$$

where zz is a linear combination of input variables. This sigmoid function outputs values between 0 and 1, which can be interpreted as probabilities. By setting a threshold (e.g., 0.5), we can classify instances into different classes.

Types of Logistic Regression

1. **Binary Logistic Regression:** Two possible outcomes (0 or 1).
2. **Multinomial Logistic Regression:** Multiple, unordered categories (e.g., predicting animal types).
3. **Ordinal Logistic Regression:** Multiple, ordered categories (e.g., low, medium, high).

Logistic Regression Assumptions

1. **Type of Outcome:** Logistic regression works best when the outcome you're trying to predict is either binary (two possible results, like "yes" or "no") or ordinal (ordered categories, like "low," "medium," "high").
2. **Relationship Type:** The method assumes that there's a linear (straight-line) connection between your predictor variables (the inputs) and the "log-odds" of your outcome. This is just a way of saying that the predictors have a consistent influence on the probability of the outcome.
3. **Independence:** Each observation (data point) should be independent, meaning one does not affect another. For example, each survey response is its own separate input.
4. **Predictor Variables:** Your predictors should not be too closely related to each other. If two predictors are very similar, it's hard for the model to tell them apart, which can confuse the predictions.
5. **Sample Size:** You need enough data points for the model to make accurate predictions. A small sample may lead to unreliable results.

Implementing Logistic Regression with a Sample Dataset Step 1: Import Libraries and Load Data

To keep things simple, let's use the famous Iris dataset from sklearn. While this dataset has three classes, we'll modify it to create a binary classification problem by focusing on two specific flower types.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = (iris.target == 0).astype(int) # Convert to binary classification:
    Iris Setosa vs. Not Setosa

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)
```

Step 2: Train the Logistic Regression Model

```
# Create and fit the Logistic Regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Check accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy * 100)
```

Answer $\rightarrow$

Accuracy : 100.0

This basic model gives us an accuracy score, which is a good start, but let's go deeper with evaluation metrics.

Evaluating a Logistic Regression Model For a complete evaluation, we'll use additional metrics such as precision, recall, F1 score, and the confusion matrix.

Confusion Matrix The confusion matrix shows the breakdown of correct and incorrect predictions:

True Positives (TP): Correctly predicted positive cases. True Negatives (TN): Correctly predicted negative cases. False Positives (FP): Incorrectly predicted positive cases. False Negatives (FN): Incorrectly predicted negative cases.

```
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)
```

Answer →

Confusion Matrix:

[[200]

[010]]

Precision, Recall, and F1 Score

1. Precision measures the accuracy of positive predictions: Precision = $\frac{TP}{TP+FP}$
2. Recall measures how well the model captures all actual positives: Recall = $\frac{TP}{TP+FN}$
3. F1 Score is the harmonic mean of precision and recall:

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

```
print("Classification Report:\n", classification_report(y_test, y_pred)
    )
```

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	20
1	1.00	1.00	1.00	10
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

Figure 3.2: Classification Report.
[datacamp(2023)]

This output shows the precision, recall, and F1 score for each class, helping us understand the model's strengths and weaknesses.

ROC Curve and AUC Score

The ROC curve plots the true positive rate (recall) against the false positive rate. The area under this curve (AUC) provides an aggregate measure of performance across all classification thresholds.

```
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# Get model probabilities for the positive class
y_prob = model.predict_proba(X_test)[:, 1]

# Calculate ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
auc = roc_auc_score(y_test, y_prob)
```

```
print("AUC Score:", auc)

# Plot ROC curve
plt.plot(fpr, tpr, label="AUC = {:.2f}".format(auc))
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(loc="lower right")
plt.show()
```

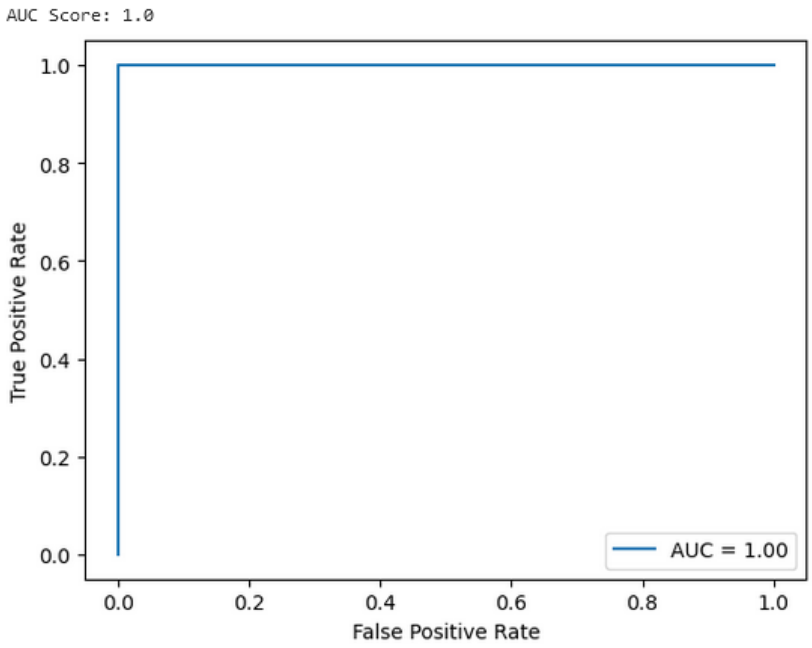

Figure 3.3: ROC Curve.
[datacamp(2023)]

A higher AUC score indicates a better model. Generally, an AUC score above 0.7 is good, with 1.0 representing a perfect classifier.

Improving the Logistic Regression Model

1. Hyperparameter Tuning: Parameters like C (regularization strength) and solver (optimization algorithm) affect the model's performance. We can tune them using Grid Search or Randomized Search.

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
        'C': [0.1, 1, 10, 100],
        'solver': ['liblinear', 'lbfgs']
}

# Initialize GridSearchCV
grid_search = GridSearchCV(LogisticRegression(), param_grid
    , cv=5)
```

```
grid_search.fit(X_train, y_train)

# Output the best parameters
print("Best parameters:", grid_search.best_params_)
```

Output → **Best parameters: 'C': 0.1, 'solver': 'liblinear'**

2. Feature Scaling:

   For better performance, especially if features vary in scale, standardize the dataset. Logistic regression models benefit from scaled features to optimize the gradient descent process.

```
from sklearn.preprocessing import StandardScaler

# Scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train and evaluate the model with scaled data
model = LogisticRegression()
model.fit(X_train_scaled, y_train)
y_pred_scaled = model.predict(X_test_scaled)
accuracy_scaled = accuracy_score(y_test, y_pred_scaled)
print("Scaled Data Accuracy:", accuracy_scaled * 100)
```

Output → **Scaled Data Accuracy: 100.0**

3. Adjusting the Decision Threshold:

   Instead of using the default 0.5 threshold, we can adjust it to maximize precision or recall based on the use case.

```
# Custom threshold
custom_threshold = 0.6
y_pred_custom = (y_prob >= custom_threshold).astype(int)
print("Custom Threshold Accuracy:", accuracy_score(y_test,
    y_pred_custom))
```

Output → **Custom Threshold Accuracy: 1.0**

[geeksforgeeks(2024)]

3.1.2 Support Vector Machine

An SVM is a machine learning model used to classify data into different groups. It's like a sorting tool for data and works especially well when you have two clear groups to separate. For example:

1. Determining if an email is spam or not spam.
2. Deciding if a mushroom is edible or poisonous.
3. Classifying sounds as normal or abnormal (like detecting machine faults from noise).

How SVM Works: Separating Data with a Line (or Hyperplane)

Imagine you're sorting candies by color into two baskets. The candies are laid out on a table, and each type has its own shape and color (e.g., red circles and blue squares). You want to draw a line across the table that divides the two types of candies as clearly as possible.

In SVM:

1. This line is called a hyperplane.
2. The goal is to place the line so that it leaves as much space as possible between the red and blue candies on either side.

The candies closest to the line are called support vectors, as they help define exactly where the line goes.

Example: If there's only one way to separate the candies (such as a straight line across the middle of the table), SVM will find this "best" line with the maximum gap between the closest candies of each type.

What About When Data Isn't Linearly Separable?

Sometimes, data is not so simple. Imagine now that you're sorting candies with swirls and dots that overlap in one section of the table, making it hard to separate them with a single straight line.

SVM has a solution for this: it can transform (or "stretch") the data into a higher dimension, where a simple line (or a flat surface) can separate the candies.

This transformation is done through something called a kernel, which allows SVM to find a line in a new space.

Absolutely, let's go over Support Vector Machines (SVM) using new examples that make it easier to understand. What is a Support Vector Machine (SVM)?

An SVM is a machine learning model used to classify data into different groups. It's like a sorting tool for data and works especially well when you have two clear groups to separate. For example:

Determining if an email is spam or not spam Deciding if a mushroom is edible or poisonous Classifying sounds as normal or abnormal (like detecting machine faults from noise)

How SVM Works: Separating Data with a Line (or Hyperplane)

Imagine you're sorting candies by color into two baskets. The candies are laid out on a table, and each type has its own shape and color (e.g., red circles and blue squares). You want to draw a line across the table that divides the two types of candies as clearly as possible.

In SVM:

This line is called a hyperplane. The goal is to place the line so that it leaves as much space as possible between the red and blue candies on either side.

The candies closest to the line are called support vectors, as they help define exactly where the line goes.

Example: If there's only one way to separate the candies (such as a straight line across the middle of the table), SVM will find this "best" line with the maximum gap between the closest candies of each type. What About When Data Isn't Linearly Separable?

Sometimes, data is not so simple. Imagine now that you're sorting candies with swirls and dots that overlap in one section of the table, making it hard to separate them with a single straight line.

SVM has a solution for this: it can transform (or "stretch") the data into a higher dimension, where a simple line (or a flat surface) can separate the candies.

This transformation is done through something called a kernel, which allows SVM to find a line in a new space.

Types of Kernels for SVM

Kernels are like tools that help us separate complex data. Here's how they work:

1. **Linear Kernel:** If the candies are already mostly separated, a simple line will do.

2. **Polynomial Kernel:** Useful when the boundary between types is curved, like in a spiral or cluster. If our candy groups are in small clusters, the polynomial kernel creates curved boundaries that fit them.

3. **Radial Basis Function (RBF) Kernel:** Great for really mixed or circular clusters. Imagine the candies are scattered in a circular way around one or two center points. The RBF kernel can handle these

circular shapes well, transforming the data so that a line can separate them in higher dimensions.

Example in Python: Classifying Fruit Ripeness Here's how SVM might work if we're classifying fruits based on ripeness (ripe vs. unripe) using features like color intensity and softness. This code finds the boundary between ripe and unripe fruits.

```
from sklearn.svm import SVC
import numpy as np
import matplotlib.pyplot as plt

# Example data for fruit ripeness (color, softness)
X = np.array([[5, 2], [3, 1], [2, 3], [8, 8], [7, 7], [6, 6]]) # Color
    intensity and softness
y = [0, 0, 0, 1, 1, 1] # 0 = unripe, 1 = ripe

# Build and train the SVM model
model = SVC(kernel="rbf", gamma=0.5, C=1.0)
model.fit(X, y)

# Visualize the decision boundary
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors="k")
plt.xlabel('Color Intensity')
plt.ylabel('Softness')
plt.title('SVM Classifying Fruit Ripeness')
plt.show()
```

This SVM model is trained to find the boundary line between unripe and ripe fruits based on color intensity and softness.

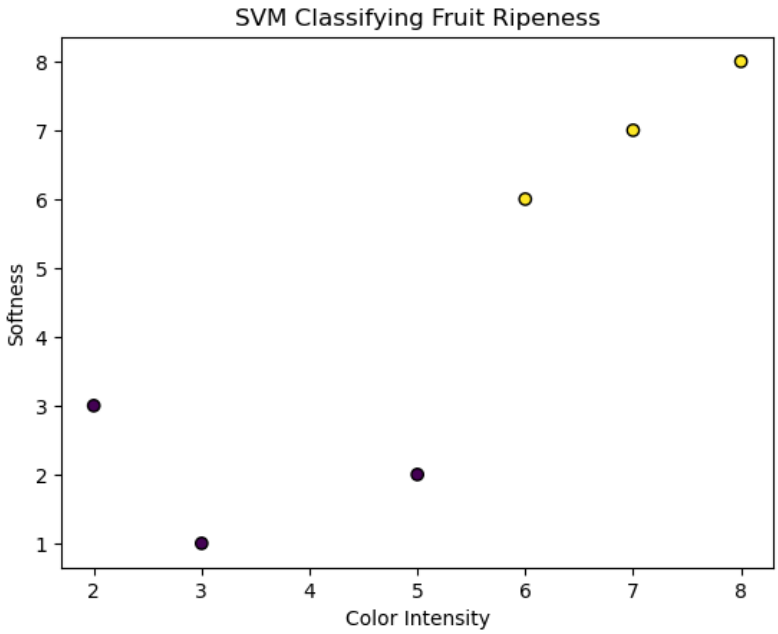

Figure 3.4: Visual representation of SVM Model.

Optimizing SVM Model

1. Hyperparameter Tuning The parameters C and gamma control the regularization and the width of the Gaussian (RBF) kernel, respectively. Optimizing these values with techniques like GridSearchCV can yield a more accurate model.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.1, 0.5, 1, 5, 10]
}
grid_search = GridSearchCV(SVC(kernel="rbf"), param_grid,
    cv=5, scoring='accuracy')
grid_search.fit(X, y)
model = grid_search.best_estimator_
print("Best Parameters:", grid_search.best_params_)
```

2. Cross-Validation for More Reliable Performance Evaluation Using cross-validation provides a more reliable estimate of model performance, especially with small datasets.

```
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(model, X_scaled, y, cv=5,
    scoring='accuracy')
print("Mean CV Accuracy:", cv_scores.mean())
```

3. Experiment with Different Kernels Depending on the data, alternative kernels like linear, poly (polynomial), or sigmoid may improve performance. Testing multiple kernels can reveal which one best separates the classes.

```
                kernels = ['linear', 'poly', 'rbf', 'sigmoid']
for kernel in kernels:
model = SVC(kernel=kernel, C=grid_search.best_params_['C'],
    gamma=grid_search.best_params_['gamma'])
model.fit(X_scaled, y)
print(f"Kernel: {kernel}, Accuracy: {model.score(X_scaled,
   y)}")
```

Summary

SVM is like a smart divider for data, finding the best line or surface to separate two classes. It's ideal for data where boundaries can be clear but may need some stretching (using kernels) to make them separable. While it's powerful, it also requires tuning, and it's not the best for every type of data. [geeksforgeeks(2024)]

Chapter 4

UNSUPERVISED MACHINE LEARNING MODELS

Unsupervised machine learning is a type of machine learning where the model finds patterns, structures, or groups in data without any labels or specific categories. It's like giving the model a mixed bag of items and asking it to sort them into groups or find common features, without telling it what each item is. This type of learning is useful for discovering natural patterns in data, like grouping customers with similar preferences or spotting unusual activity. While this writing focuses on supervised models, it only provides a brief overview of hierarchical clustering.

4.1 Clustering Models

Cluster analysis aims to identify patterns in data by grouping observations (a single data point or a row) into clusters, where objects within the same cluster are similar, and those in different clusters are dissimilar. It is a type of multivariate analysis, using multiple features (or vectors) to reveal latent structures in the data. The technique helps discover potential natural groupings, even if these clusters may not exist in the population.

Euclidean distance d,

$$(\mathbf{X}_i, \mathbf{X}_j) = \sqrt{(\mathbf{X}_i - \mathbf{X}_j)'(\mathbf{X}_i - \mathbf{X}_j)} = \sqrt{\sum_{k=1}^{p}(X_{ki} - X_{kj}^2)}$$

This gives the distance between the Xi and Xj observation. Using p number of features(variables). This means calculating the distance between two objects using their difference or similarities which is determined by p features/characteristics of each object.

Types of Clustering include:

1. **Hierarchical Clustering**

Hierarchical clustering is a data analysis method used to group observations based on their features. The hierarchy arranges data points into groups or clusters at different levels. Smaller clusters merge into larger ones, forming a tree-like structure that shows how closely related the data points are.

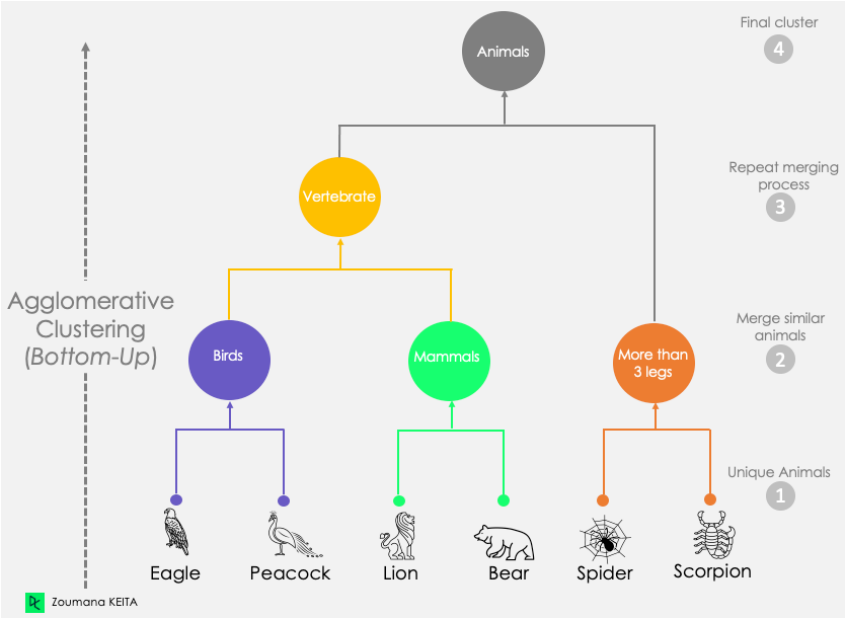

Figure 4.1: Visual representation of Hierarchical clustering. [datacamp(2023)]

It can be performed in two main ways:

(a) **Agglomerative clustering:** It starts by treating each observation as its own cluster, with n individual clusters (one per observation). At each step, the algorithm calculates the distance or similarity between all clusters and merges the two closest ones. This process continues step by step until all observations are combined into a single cluster, moving from individual clusters up to one large group.

(b) **Divisive clustering:** starts with all observations in a single large cluster. At each step, the algorithm splits the cluster into smaller groups based on how different or dissimilar the observations are, this process continues, dividing clusters step by step, until each observation is in its own cluster or until the desired number of clusters is reached, moving from one large group down to many smaller clusters.

The results can be visualized using a dendrogram, a tree-like diagram that shows the order in which clusters are combined or split. By cutting the dendrogram at a certain level, the optimal number of clusters can be determined. [Dale Bowman, Tasha Sahr, Andrew M. Olney. (2020)]

Listing 4.1: Hierarchical Clustering Model in Python

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage

# Sample data
X = np.array([[5, 3],
[10, 15],
[15, 12],
[24, 10],
[30, 30],
[85, 70],
[71, 80],
[60, 78],
[70, 55],
[80, 91]])

# Perform hierarchical clustering
linked = linkage(X, 'single')

# Create and plot dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linked,
orientation='top',
distance_sort='descending',
show_leaf_counts=True)
plt.show()
```

This Python code performs hierarchical clustering and visualizes it through a dendrogram:

(a) Libraries:

i. numpy: Creates a sample 2D dataset.

ii. matplotlib.pyplot: Plots the dendrogram to visualize the clustering.

iii. scipy.cluster.hierarchy: Contains the linkage function for clustering and dendrogram function for visualization.

(b) Data:

A 2D array X is used as the sample data for clustering.

(c) Clustering (linkage):

The linkage function performs hierarchical clustering using the single linkage method, calculating the distances between clusters and returning a linkage matrix.

(d) Dendrogram Visualization:

The dendrogram function generates a tree-like structure showing how individual data points merge into clusters based on distance. The plot includes cluster branch lengths representing the merging distances and leaf nodes showing data points.

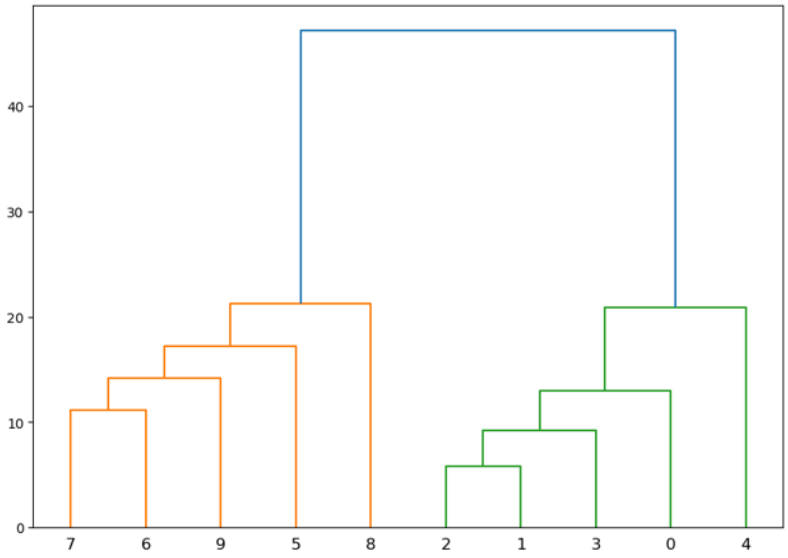

Figure 4.2: Visual representation of the Hierarchical Cluster Model

2. **Nonhierarchical Clustering**

Non-hierarchical clustering uses partitioning methods, also known as optimization methods. In this approach, the observations or data points are grouped into a set number of g clusters, and the best clustering arrangement is chosen based on specific criteria. These criteria typically focus on minimizing the variance between data points or observations in a cluster, which measures how similar the data points within each cluster are to one another. The goal is to ensure that points within the same cluster are as close as possible, while maximizing the separation between different clusters to enhance distinctiveness. A common example of a partitioning method is k-means clustering.

Bibliography

[GeeksForGeeks(2024)] GeeksForGeeks. *Machine Learning Models*. GeeksForGeeks, 8 Aug, 2024. URL: http://example.com

[GeeksForGeeks(2024)] GeeksForGeeks. *Decision Tree*. GeeksForGeeks, 17 May, 2024. URL: https://www.geeksforgeeks.org/decision-tree/

[AnalyticsVidhya(2024)] AnalyticsVidhya. *Linear Regression*. AnalyticsVidhya, 30 Sep, 2024. URL: https://www.analyticsvidhya.com/blog/2021/10/everything-you-need-to-know-about-linear-regression/#h-what-is-linear-regression

[AnalyticsVidhya(2024)] AnalyticsVidhya. *Random Forest*. AnalyticsVidhya, 8 Oct, 2024. URL: https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

[datacamp(2024)] Classification in Machine Learning. *Random Forest*. datacamp, 8 Aug, 2024. URL: https://www.datacamp.com/blog/classification-machine-learning

[geeksforgeeks(2024)] Logistic Regression. *Logistic Regression*. datacamp, 20 June, 2024. URL: https://www.geeksforgeeks.org/understanding-logistic-regression/

[geeksforgeeks(2024)] Support Vector Machine. *Support Vector Machine*. datacamp, 10 Oct, 2024. URL: https://www.geeksforgeeks.org/support-vector-machine-algorithm/

[IBM(2021)] IBM. *Supervised versus unsupervised learning:* IBM, 12 March, 2021. URL: https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning

[IMSL(2021)] IMSL. *What is a Regression Model:* IMSL, 16 June, 2021. URL: https://www.imsl.com/blog/what-is-regression-model

[datacamp(2023)] datacamp. *An Introduction to Hierarchical Clustering in Python*. datacamp, 19 Jan, 2023. URL: https://www.datacamp.com/tutorial/introduction-hierarchical-clustering-python

[GeeksForGeeks(2024)] GeeksForGeeks. *Linear Regression in Machine Learning*. GeeksForGeeks, 23 Oct, 2024. URL: https://www.geeksforgeeks.org/ml-linear-regression/

[Dale Bowman, Tasha Sahr, Andrew M. Olney. (2020)] Dale Bowman, Tasha Sahr, Andrew M. Olney.

[Cormen et al.(2009)] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[Sebesta(2012)] Sebesta, Robert W. *Concepts of Programming Languages*. Pearson, 2012.