Content (Code in Python)

1. Linear regression (Chapter 3)
2. Logistic Regression (Chapter 4)
3. Decision Tree (Chapter 5)
4. Random Forest (Chapter 6)
5. Rule base classifier (Chapter 7)
6. Naïve Bayes Classifier (Chapter 8)
7. K-NN (Chapter 9)
8. Support Vector Machine (Chapter 10)
9. K-means Clustering (Chapter 11)
10. PCA (Principal Component Analysis) (Chapter 12)
11. Dimensionality Reduction with t-SNE (Chapter 12)
12. Association Rule Mining (Chapter 13)
13. FP growth (Chapter 13)

Linear regression (Chapter 3)

# Import the necessary libraries

import pandas as pd

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

# Load the data

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

# Select the features and target variable

X = data[['feature1', 'feature2',...]] # features

y = data['target'] # target variable

# Split the 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 a linear regression model

model = LinearRegression()

# Train the model on the training data

model.fit(X\_train, y\_train)

# Make predictions on the testing data

y\_pred = model.predict(X\_test)

# Evaluate the model using mean squared error

mse = model.score(X\_test, y\_test)

print(f'Mean squared error: {mse:.2f}')

# Get the coefficients

coefficients = model.coef\_

print(f'Coefficients: {coefficients}')

# Get the intercept

intercept = model.intercept\_

print(f'Intercept: {intercept}')

Logistic Regression (Chapter 4)

# Import the necessary libraries

import pandas as pd

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

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

# Load the data

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

# Select the features and target variable

X = data[['feature1', 'feature2',...]] # features

y = data['target'] # target variable

# Split the 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 a logistic regression model

model = LogisticRegression()

# Train the model on the training data

model.fit(X\_train, y\_train)

# Make predictions on the testing data

y\_pred = model.predict(X\_test)

# Evaluate the model using accuracy score

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

print(f'Accuracy: {accuracy:.2f}')

# Evaluate the model using classification report

report = classification\_report(y\_test, y\_pred)

print(f'Classification Report:\n{report}')

# Evaluate the model using confusion matrix

matrix = confusion\_matrix(y\_test, y\_pred)

print(f'Confusion Matrix:\n{matrix}')

# Get the coefficients

coefficients = model.coef\_

print(f'Coefficients: {coefficients}')

# Get the intercept

intercept = model.intercept\_

print(f'Intercept: {intercept}')

Decision Tree (Chapter 5)

# Import the necessary libraries

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split

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

# Load the data

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

# Select the features and target variable

X = data[['feature1', 'feature2',...]] # features

y = data['target'] # target variable

# Split the 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 a decision tree classifier

model = DecisionTreeClassifier(random\_state=42)

# Train the model on the training data

model.fit(X\_train, y\_train)

# Make predictions on the testing data

y\_pred = model.predict(X\_test)

# Evaluate the model using accuracy score

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

print(f'Accuracy: {accuracy:.2f}')

# Evaluate the model using classification report

report = classification\_report(y\_test, y\_pred)

print(f'Classification Report:\n{report}')

# Evaluate the model using confusion matrix

matrix = confusion\_matrix(y\_test, y\_pred)

print(f'Confusion Matrix:\n{matrix}')

# Visualize the decision tree

from sklearn.tree import plot\_tree

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

plot\_tree(model, filled=True)

plt.show()

Random Forest (Chapter 6)

# Import the necessary libraries

import pandas as pd

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

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

# Load the data

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

# Select the features and target variable

X = data[['feature1', 'feature2',...]] # features

y = data['target'] # target variable

# Split the 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 a random forest classifier

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

# Train the model on the training data

model.fit(X\_train, y\_train)

# Make predictions on the testing data

y\_pred = model.predict(X\_test)

# Evaluate the model using accuracy score

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

print(f'Accuracy: {accuracy:.2f}')

# Evaluate the model using classification report

report = classification\_report(y\_test, y\_pred)

print(f'Classification Report:\n{report}')

# Evaluate the model using confusion matrix

matrix = confusion\_matrix(y\_test, y\_pred)

print(f'Confusion Matrix:\n{matrix}')

# Get the feature importances

importances = model.feature\_importances\_

print(f'Feature Importances: {importances}')

# Plot the feature importances

import matplotlib.pyplot as plt

plt.barh(range(X.shape[1]), importances)

plt.xlabel('Importance')

plt.ylabel('Feature')

plt.show()

Rule base classifier (Chapter 7)

# Define the rules

rules = [

{"if": {"feature1": ">=", "value": 5}, "then": "class1"},

{"if": {"feature2": "<", "value": 3}, "then": "class2"},

{"if": {"feature1": "<", "value": 3} and {"feature2": ">=", "value": 4}, "then": "class3"},

# Add more rules as needed

]

# Define the data

data = [

{"feature1": 4, "feature2": 2, "target": "class1"},

{"feature1": 6, "feature2": 1, "target": "class2"},

{"feature1": 2, "feature2": 5, "target": "class3"},

# Add more data points as needed

]

# Define the classifier

class RuleBasedClassifier:

def \_\_init\_\_(self, rules):

self.rules = rules

def classify(self, instance):

for rule in self.rules:

if self.evaluate\_rule(rule["if"], instance):

return rule["then"]

return "unknown"

def evaluate\_rule(self, condition, instance):

for feature, op, value in condition.items():

if op == ">=":

if instance[feature] >= value:

continue

else:

return False

elif op == "<":

if instance[feature] < value:

continue

else:

return False

# Add more operators as needed

return True

# Create an instance of the classifier

classifier = RuleBasedClassifier(rules)

# Test the classifier

for instance in data:

predicted\_class = classifier.classify(instance)

print(f"Instance: {instance}, Predicted class: {predicted\_class}")

Naïve Bayes Classifier (Chapter 8)

import numpy as np

class NaiveBayes:

def fit(self, X, y):

n\_samples, n\_features = X.shape

self.\_classes = np.unique(y)

n\_classes = len(self.\_classes)

# calculate mean, var, and prior for each class

self.\_mean = np.zeros((n\_classes, n\_features), dtype=np.float64)

self.\_var = np.zeros((n\_classes, n\_features), dtype=np.float64)

self.\_priors = np.zeros(n\_classes, dtype=np.float64)

for idx, c in enumerate(self.\_classes):

X\_c = X[y==c]

self.\_mean[idx, :] = X\_c.mean(axis=0)

self.\_var[idx, :] = X\_c.var(axis=0)

self.\_priors[idx] = X\_c.shape[0] / float(n\_samples)

def predict(self, X):

y\_pred = []

for x in X:

posteriors = []

for idx, c in enumerate(self.\_classes):

prior = np.log(self.\_priors[idx])

posterior = np.sum(np.log(self.\_pdf(idx, x)))

posterior = prior + posterior

posteriors.append(posterior)

y\_pred.append(self.\_classes[np.argmax(posteriors)])

return y\_pred

def \_pdf(self, class\_idx, x):

mean = self.\_mean[class\_idx]

var = self.\_var[class\_idx]

numerator = np.exp(-((x-mean)\*\*2) / (2 \* var))

denominator = np.sqrt(2 \* np.pi \* var)

return numerator / denominator

# Example usage

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

iris = load\_iris()

X, y = iris.data, iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

nb = NaiveBayes()

nb.fit(X\_train, y\_train)

y\_pred = nb.predict(X\_test)

print("Naive Bayes classification accuracy:", np.mean(y\_pred == y\_test))

K-NN (Chapter 9)

import numpy as np

from collections import Counter

class KNN:

def \_\_init\_\_(self, k=3):

self.k = k

def fit(self, X, y):

self.X\_train = X

self.y\_train = y

def predict(self, X):

predictions = [self.\_predict(x) for x in X]

return np.array(predictions)

def \_predict(self, x):

# Compute distances

distances = [np.sqrt(np.sum((x - x\_train) \*\* 2)) for x\_train in self.X\_train]

# Get k nearest samples, labels

k\_indices = np.argsort(distances)[:self.k]

k\_nearest\_labels = [self.y\_train[i] for i in k\_indices]

# Majority vote, most common class label

most\_common = Counter(k\_nearest\_labels).most\_common(1)

return most\_common[0][0]

# Example usage

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

iris = load\_iris()

X, y = iris.data, iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

knn = KNN(k=5)

knn.fit(X\_train, y\_train)

y\_pred = knn.predict(X\_test)

print("kNN classification accuracy:", np.mean(y\_pred == y\_test))

Support Vector Machine (Chapter 10)

from sklearn import datasets

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Load the iris dataset

iris = datasets.load\_iris()

X = iris.data

y = iris.target

# Split the 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 an SVM classifier with a linear kernel

svm = SVC(kernel='linear', C=1.0, random\_state=42)

# Train the SVM classifier

svm.fit(X\_train, y\_train)

# Make predictions on the testing set

y\_pred = svm.predict(X\_test)

# Evaluate the accuracy of the SVM classifier

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

print("SVM classification accuracy:", accuracy)

K-means Clustering (Chapter 11)

import numpy as np

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

# Generate some sample data

np.random.seed(0)

X = np.random.rand(100, 2)

# Create a KMeans object with 3 clusters

kmeans = KMeans(n\_clusters=3, random\_state=0)

# Fit the KMeans object to the data

kmeans.fit(X)

# Get the cluster labels

labels = kmeans.labels\_

# Get the cluster centers

centers = kmeans.cluster\_centers\_

# Plot the data and the cluster centers

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')

plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5)

plt.show()

PCA (Principal Component Analysis) (Chapter 12)

import numpy as np

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

# Generate some sample data

np.random.seed(0)

X = np.random.rand(100, 5)

# Create a PCA object with 2 components

pca = PCA(n\_components=2)

# Fit the PCA object to the data

pca.fit(X)

# Transform the data into the new coordinate system

X\_pca = pca.transform(X)

# Print the explained variance ratio for each component

print("Explained variance ratio:", pca.explained\_variance\_ratio\_)

# Plot the original data

plt.scatter(X[:, 0], X[:, 1], c=X[:, 2], cmap='viridis')

plt.title("Original Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

# Plot the transformed data

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=X[:, 2], cmap='viridis')

plt.title("PCA-Transformed Data")

plt.xlabel("Principal Component 1")

plt.ylabel("Principal Component 2")

plt.show()

Dimensionality Reduction with t-SNE (Chapter 12)

import numpy as np

import matplotlib.pyplot as plt

from sklearn.manifold import TSNE

# Generate some sample data

np.random.seed(0)

X = np.random.rand(100, 10)

# Create a t-SNE object with 2 components

tsne = TSNE(n\_components=2, random\_state=0)

# Fit the t-SNE object to the data

X\_tsne = tsne.fit\_transform(X)

# Plot the original data

plt.scatter(X[:, 0], X[:, 1], c=X[:, 2], cmap='viridis')

plt.title("Original Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

# Plot the t-SNE-transformed data

plt.scatter(X\_tsne[:, 0], X\_tsne[:, 1], c=X[:, 2], cmap='viridis')

plt.title("t-SNE-Transformed Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

Association Rule Mining (Chapter 13)

import pandas as pd

from mlxtend.frequent\_patterns import apriori

from mlxtend.frequent\_patterns import association\_rules

# Load the dataset

data = pd.read\_csv('market\_basket.csv')

# Convert the data into a binary matrix

binary\_data = pd.get\_dummies(data.stack()).sum(level=0)

# Apply the apriori algorithm to find frequent itemsets

frequent\_itemsets = apriori(binary\_data, min\_support=0.05, use\_colnames=True)

# Generate association rules from the frequent itemsets

rules = association\_rules(frequent\_itemsets, metric='confidence', min\_threshold=0.7)

# Print the association rules

print(rules)

FP growth (Chapter 13)

from collections import defaultdict

class TreeNode:

def \_\_init\_\_(self, name, count, parent\_node):

self.name = name

self.count = count

self.parent\_node = parent\_node

self.children = {}

self.next\_sibling = None

class FPGrowth:

def \_\_init\_\_(self, transactions, min\_support):

self.transactions = transactions

self.min\_support = min\_support

self.item\_counts = defaultdict(int)

self.fp\_tree = None

self.frequent\_items = None

def build\_fp\_tree(self):

# Create the root node of the FP-tree

root\_node = TreeNode('root', 0, None)

# Iterate over the transactions and build the FP-tree

for transaction in self.transactions:

current\_node = root\_node

for item in transaction:

if item not in current\_node.children:

current\_node.children[item] = TreeNode(item, 1, current\_node)

else:

current\_node.children[item].count += 1

current\_node = current\_node.children[item]

# Update the item counts

for node in self.get\_nodes(root\_node):

self.item\_counts[node.name] += node.count

# Prune the FP-tree to remove items with low support

self.fp\_tree = self.prune\_fp\_tree(root\_node)

def prune\_fp\_tree(self, node):

if node.count < self.min\_support:

return None

for child in node.children.values():

child = self.prune\_fp\_tree(child)

if child is not None:

node.children[child.name] = child

else:

del node.children[child.name]

return node

def get\_nodes(self, node):

nodes = [node]

for child in node.children.values():

nodes.extend(self.get\_nodes(child))

return nodes

def mine\_fp\_tree(self):

# Find the frequent items

self.frequent\_items = [item for item, count in self.item\_counts.items() if count >= self.min\_support]

# Mine the FP-tree to find the frequent patterns

patterns = []

for item in self.frequent\_items:

pattern = [item]

self.mine\_fp\_tree\_recursive(self.fp\_tree, pattern, patterns)

return patterns

def mine\_fp\_tree\_recursive(self, node, pattern, patterns):

if node.count >= self.min\_support:

patterns.append(pattern[:])

for child in node.children.values():

pattern.append(child.name)

self.mine\_fp\_tree\_recursive(child, pattern, patterns)

pattern.pop()

def run(self):

self.build\_fp\_tree()

patterns = self.mine\_fp\_tree()

return patterns

# Example usage

transactions = [

['A', 'B', 'C'],

['A', 'B', 'D'],

['A', 'C', 'D'],

['B', 'C', 'D'],

['A', 'B', 'C', 'D']

]

min\_support = 2

fp\_growth = FPGrowth(transactions, min\_support)

patterns = fp\_growth.run()

print("Frequent patterns:")

for pattern in patterns:

print(pattern)