Machine Learning and Deep Learning Notes

Tim Rößling

Invalid Date

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

Welcome to the neural grimoire of machine learning and deep learning—a living, breathing fusion of math, code, and curiosity. This isn't your typical textbook. Think of it as a digital spellbook, where Python is the wand, NumPy is your incantation library, and each code cell is a summon to the gods of computation.

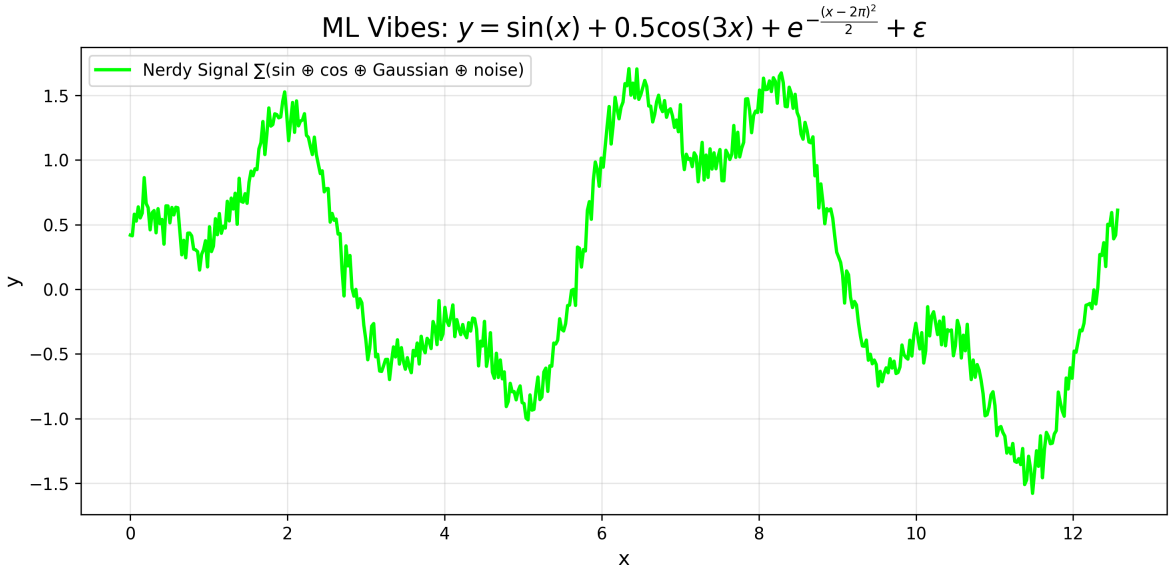
Here, we don't just learn machine learning—we tinker, plot, debug, and dive deep into the arcane mechanics behind models. Inspired by Donald Knuth's vision of literate programming (Knuth (1984)), this book marries prose with executable Python, so ideas are readable for humans and executable for machines—a poetic duet of code and concept.

Whether it's gradient descent (aka "foggy hill hiking"), activation functions ("threshold guardians"), or backpropagation ("the chain rule on steroids"), everything here is broken down with analogies, animations, and hands-on snippets that make abstract math feel tangible.

Let's begin with a nerdy Python warm-up—because no ML journey should start without at least one plot… of a Fourier-based sine wave mashup with a hint of randomness—just like data in the wild.

A Sine Wave… with a Nerdy Twist

```
import numpy as np
import matplotlib.pyplot as plt

# A nerdy sine-cosine fusion with a Gaussian bump and a dash of noise
x = np.linspace(0, 4 * np.pi, 500)
y = np.sin(x) + 0.5 * np.cos(3 * x) + np.exp(-0.5 * (x - 2*np.pi)**2) + 0.1 * np.random.randn

plt.figure(figsize=(10, 5))
plt.plot(x, y, color='lime', linewidth=2, label='Nerdy Signal (sin + cos + Gaussian + noise)'
plt.title(r'$\text{ML Vibes: } y = \sin(x) + 0.5\cos(3x) + e^{-\frac{(x-2\pi)^2}{2}} + \epsil
plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()
```



Why Start Here? Because this plot is a metaphor: order meets chaos, signal meets noise, theory meets messy real-world data—a perfect teaser for the ML landscape. From this wobbly waveform to deep neural architectures, everything builds upon fundamental functions, just like these.

So buckle up—this is ML for the curious, the meticulous, and the unapologetically nerdy. Whether you're deciphering a loss curve or fine-tuning a transformer, this book is your Pythonic companion in the quest to tame intelligent algorithms.

Let's dive into the matrix.

2 Fundamentals of Machine Learning

Machine Learning (ML) powers everything from Netflix suggestions to self-driving cars. But what is it? ML is teaching computers to learn from data and make decisions—think of it like training a dog with treats (data) to do tricks (predictions).

2.1 Types of Machine Learning Systems

ML systems are categorized by how they learn. Why? It determines what data they need and how they'll perform.

2.1.1 1. Supervised Learning

Supervised learning is like a teacher guiding a student with a textbook and answer key. We give the model inputs (features) and outputs (labels) to learn patterns.

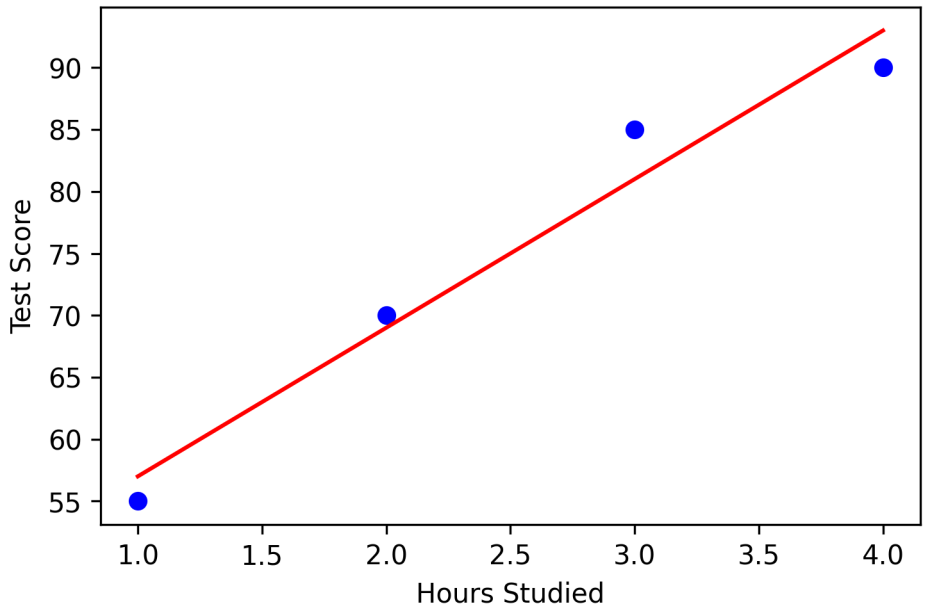
2.1.1.1 Key Techniques

- **Classification**: Sorting data into buckets—like labeling emails as "spam" or "not spam."
- **Regression**: Predicting numbers—like guessing tomorrow's temperature.

Example: Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Features (hours studied), Labels (test scores)
X = np.array([[1], [2], [3], [4]])
y = np.array([55, 70, 85, 90])

model = LinearRegression().fit(X, y)
predictions = model.predict(X)
```

```
plt.scatter(X, y, color="blue")
plt.plot(X, predictions, color="red")
plt.xlabel("Hours Studied")
plt.ylabel("Test Score")
plt.show()
```



Why? This shows how hours studied predict scores with a straight line.

2.1.2 2. Unsupervised Learning

No labels here—think of it like a librarian organizing books without titles. The system finds patterns on its own.

2.1.2.1 Key Techniques

– **Clustering: Grouping similar items—like sorting candies by color.** – **Dimensionality Reduction: Shrinking data while keeping the good stuff—like summarizing a book into key points.** Example: K-Means Clustering

```
from sklearn.cluster import KMeans
import numpy as np
```

```
# Random data points
X = np.array([[1, 2], [1, 4], [1, 0], [10, 2], [10, 4], [10, 0]])

kmeans = KMeans(n_clusters=2).fit(X)
labels = kmeans.labels_

print("Cluster labels:", labels)
```

```
Cluster labels: [1 1 1 0 0 0]
```

Why? This groups data into 2 clusters based on similarity.

2.1.3 3. Semi-Supervised Learning

A hybrid approach—imagine teaching with a few labeled examples and a pile of unlabeled ones. Why? It's efficient when labeling all data is too costly (e.g., speech recognition).

2.1.4 4. Reinforcement Learning

Think of training a puppy with treats and timeouts. An agent learns by trying actions in an environment, earning rewards or penalties. Why? Perfect for dynamic tasks like robotics.

2.1.4.1 Example Concept

A robot learning to walk gets a treat (reward) for each step forward.

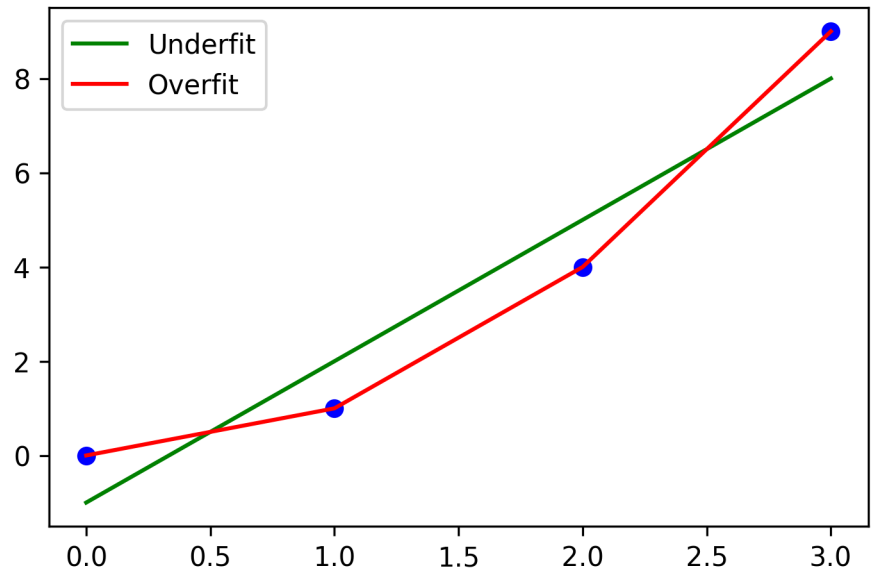
2.1.4.2 Main Challenges of Machine Learning

ML isn't perfect—here's why these challenges matter.

- **Insufficient Training Data**: Models need lots of data—like a chef needing ingredients to cook well.
- **Nonrepresentative Data**: Bad data = bad predictions—like using beach weather to predict mountain snow.
- **Poor Quality Data**: Noise or errors mess it up—like static in a phone call.
- **Irrelevant Features**: Extra junk confuses the model—like adding random spices to a recipe.
- **Overfitting**: Memorizing the textbook but failing the test—too specific to training data.

- **Underfitting**: Too simple, like using a straight line for a curvy pattern. Example: Overfitting vs. Underfitting

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Data
X = np.array([[0], [1], [2], [3]])
y = np.array([0, 1, 4, 9])

# Underfit (linear)
lin_model = LinearRegression().fit(X, y)
lin_pred = lin_model.predict(X)

# Overfit (high-degree polynomial)
poly = PolynomialFeatures(degree=3)
X_poly = poly.fit_transform(X)
poly_model = LinearRegression().fit(X_poly, y)
poly_pred = poly_model.predict(X_poly)

plt.scatter(X, y, color="blue")
plt.plot(X, lin_pred, color="green", label="Underfit")
plt.plot(X, poly_pred, color="red", label="Overfit")
plt.legend()
plt.show()
```



Why? Green underfits (misses the curve), red overfits (too wiggly).

3 Building an End-to-End Machine Learning Pipeline

3.1 Look at the Big Picture

Every machine learning project starts with a clear understanding of the problem.

Example scenario:
You are working at an e-commerce company, and your goal is to predict product demand using historical sales data. Features may include:

- Product category
- Price
- Seasonality
- Customer demographics

This enables the company to optimize inventory and reduce stockouts.

Analogy: Think of your model as a weather forecast — not perfect, but helpful in making decisions ahead of time.

3.2 Get the Data

Acquiring and preparing the data is the first technical step. You may:

- Query internal databases
- Use APIs to fetch real-time data
- Scrape websites
- Merge multiple sources and ensure data consistency

3.2.1 Python Example

import numpy as np import pandas as pd

```
# 1. Define the Problem and Get the Data
from sklearn.datasets import load_iris
import pandas as pd

# Load the Iris dataset
data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="species")

# Display the first few rows of the DataFrame
print(X.head())
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
```

3.2.2 Dataset Splitting

- **Training set**: Used to train the model
- **Validation set**: Used to tune hyperparameters and prevent overfitting
- **Test set**: Used to evaluate final performance

Analogy: The test set is like your final exam — it should not be used during preparation.

3.2.3 Python Example

```
# 2. Dataset Splitting
from sklearn.model_selection import train_test_split

# Split data into 70% training, 15% validation, 15% test
```

```
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_sta
```

3.3 Discover and Visualize the Data

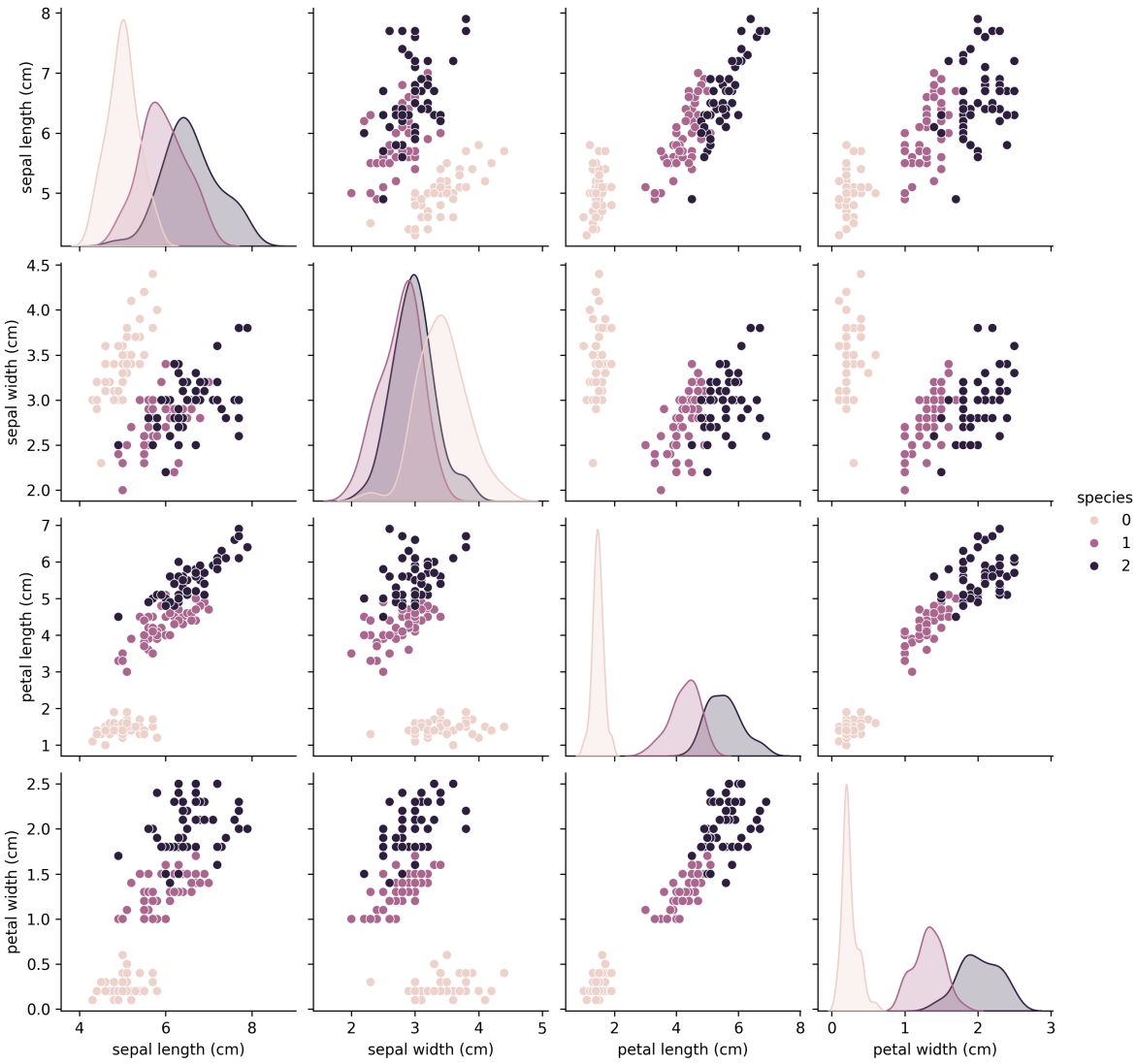
Understanding the data is critical before model building. Steps include:

- Identifying missing values
- Detecting outliers
- Checking skewness of distributions
- Visualizing feature relationships (scatter plots, histograms, correlation matrices)
- Understanding categorical vs. numerical features

Analogy: This step is like reading a map before planning your route.

3.3.1 Python Example

```
# 3. Data Exploration
import seaborn as sns
import matplotlib.pyplot as plt

# Visualize the feature relationships using a pairplot
sns.pairplot(pd.concat([X, y], axis=1), hue='species')
plt.show()
```



3.4 Feature Scaling and Normalization

Many algorithms are sensitive to the magnitude of features.

3.4.1 Common Methods

- **Min-Max Scaling (Normalization)**: Rescales features to a fixed range, typically [0, 1]
- **Standardization**: Centers features around zero mean with unit variance

Analogy: Think of it like converting all ingredients to the same unit before cooking a recipe.

3.4.2 Python Example

```
# 4. Feature Scaling
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
X_test_scaled = scaler.transform(X_test)
```

3.5 Prepare the Data for Machine Learning Algorithms

3.5.1 Common Preprocessing Steps

1. **Handle Missing Values**

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy='median')
X_filled = imputer.fit_transform(X)
```

2. **Encode Categorical Variables**

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse=False)
X_encoded = encoder.fit_transform(df[['category']])
```

3. **Feature Engineering**

```
df['year'] = pd.to_datetime(df['date']).dt.year
```

4. **Feature Scaling** (as shown earlier)

Analogy: These steps are like preparing clean and measured ingredients before starting to cook.

3.6 Using Scikit-Learn Pipelines

Pipelines help automate the preprocessing steps in sequence, ensuring consistency and reducing errors.

3.6.1 Example Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

X_prepared = pipeline.fit_transform(X)
```

Analogy: A pipeline is like an assembly line — each step handles part of the process automatically.

3.7 Select and Train a Model

Select a model based on your data type and problem type.

- **Linear Regression**: For continuous numeric targets
- **Decision Trees / Random Forests**: For structured tabular data
- **Gradient Boosting (e.g., XGBoost, LightGBM)**: For high performance
- **Neural Networks**: For complex/high-dimensional data

3.7.1 Training Example

```
# 5. Model Training (Logistic Regression as an example)
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=200)
model.fit(X_train_scaled, y_train)
```

```
LogisticRegression(max_iter=200)
```

3.8 Fine-Tune Your Model

After training, tune the model to improve performance.

3.8.1 Common Techniques

- **Grid Search**: Tries all combinations of parameters
- **Random Search**: Samples random combinations
- **Cross-Validation**: Ensures performance generalizes well

3.8.2 Example

```
# 6. Fine-Tuning (Optional)
from sklearn.model_selection import GridSearchCV

# Example of hyperparameter tuning using GridSearchCV
param_grid = {'C': [0.1, 1, 10], 'solver': ['liblinear', 'lbfgs']}
grid_search = GridSearchCV(LogisticRegression(max_iter=200), param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

# Best parameters and score after tuning
print("Best Parameters:", grid_search.best_params_)
print("Best Accuracy:", grid_search.best_score_)
```

```
Best Parameters: {'C': 1, 'solver': 'lbfgs'}
Best Accuracy: 0.9428571428571428
```

3.9 Model Evaluation Metrics

Different tasks need different evaluation metrics.

- **Accuracy**: Ratio of correct predictions (for balanced classification tasks)
- **Precision**: Proportion of predicted positives that are correct
- **Recall**: Proportion of actual positives that are correctly identified
- **F1 Score**: Harmonic mean of precision and recall (useful for imbalanced classes)
- **Confusion Matrix**: Table of true vs. predicted values

3.9.1 Python Example

```
# 7. Model Evaluation (Accuracy)
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Make predictions on the test set
y_pred = model.predict(X_test_scaled)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

```
Accuracy: 1.0000
Confusion Matrix:
[[ 6  0  0]
 [ 0 10  0]
 [ 0  0  7]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
```

```
           1       1.00      1.00      1.00        10
           2       1.00      1.00      1.00         7

    accuracy                           1.00        23
   macro avg       1.00      1.00      1.00        23
weighted avg       1.00      1.00      1.00        23
```

3.10 Present Your Solution

Communicate your work effectively:

- Prepare a concise report
- Include visualizations and metrics
- Share business recommendations
- Package the model (e.g., via API or script)

Analogy: A good model is only useful if others can understand and use it.

3.11 Launch, Monitor, and Maintain the System

Deployment involves more than just shipping the model:

- Expose it via an API or application
- Monitor predictions over time
- Detect and handle model drift
- Automate retraining as data evolves
- Ensure logging, scalability, and security

Analogy: Model deployment is like maintaining software — it needs updates, monitoring, and support.

3.12 Summary Checklist

- Define the problem
- Collect and clean the data
- Explore and visualize the data
- Prepare the data (encode, scale, engineer)
- Build pipelines
- Train models
- Fine-tune with validation
- Evaluate using appropriate metrics
- Present results clearly
- Deploy, monitor, and maintain the system

4 Unsupervised Learning: Clustering Techniques

4.1 What is Unsupervised Learning?

Unsupervised learning is a type of machine learning where the model identifies patterns or structures in data **without labels**. Unlike supervised learning, where the model is trained with input-output pairs, unsupervised learning works solely with input features to uncover structure.

4.1.1 Why Use It?

- No labels are required
- Ideal for exploratory analysis and large unlabeled datasets
- Helps uncover hidden groupings, relationships, and outliers

4.1.2 Analogy

Think of unsupervised learning like exploring a city without a map. You don't know what's where, but you start grouping areas based on what you see—residential zones, commercial zones, parks, etc.

4.2 K-Means Clustering

4.2.1 What is K-Means?

K-Means is a popular clustering algorithm that partitions data into **k distinct clusters**, each represented by a **centroid** (the mean of the cluster points).

4.2.2 Use Cases

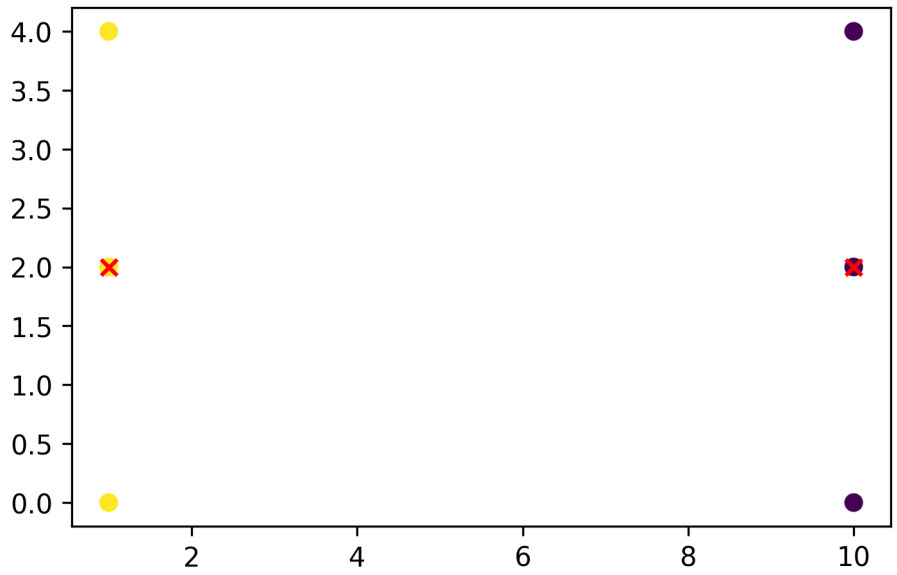
- Customer segmentation
- Fraud detection
- Market segmentation

4.2.3 How It Works

1. **Initialization**: Choose k initial centroids randomly.
2. **Assignment Step**: Assign each point to the nearest centroid.
3. **Update Step**: Recalculate centroids as the mean of assigned points.
4. **Repeat** until convergence (no or minimal change in centroids).

```
from sklearn.cluster import KMeans
import numpy as np
from matplotlib import pyplot as plt

X = np.array([[1, 2], [1, 4], [1, 0], [10, 2], [10, 4], [10, 0]])
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
print(kmeans.labels_)
print(kmeans.cluster_centers_)

# plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='red', marker='x
plt.show()
```

```
[1 1 1 0 0 0]
[[10.  2.]
 [ 1.  2.]]
```



4.2.4 Convergence & Efficiency

- Linear complexity: O(n * k * d)
- Always converges, but **not necessarily to the global optimum**
- Sensitive to **initial centroids**

4.2.5 Centroid Initialization Strategies

- Random initialization
- **K-means++** (recommended): smarter seeding for better clusters
- Manual initialization (if prior knowledge exists)

4.2.6 Example: Webstore Segmentation

Cluster customers into: - High spenders - Discount seekers - Infrequent shoppers

4.2.7 Pros

- Simple, fast, scalable
- Works well with large datasets

4.2.8 Cons

- Needs k beforehand
- Sensitive to outliers and initial placement
- Assumes spherical clusters

4.3 Mini-Batch K-Means

Mini-Batch K-Means is a faster, scalable variant of K-Means.

4.3.1 How It Works

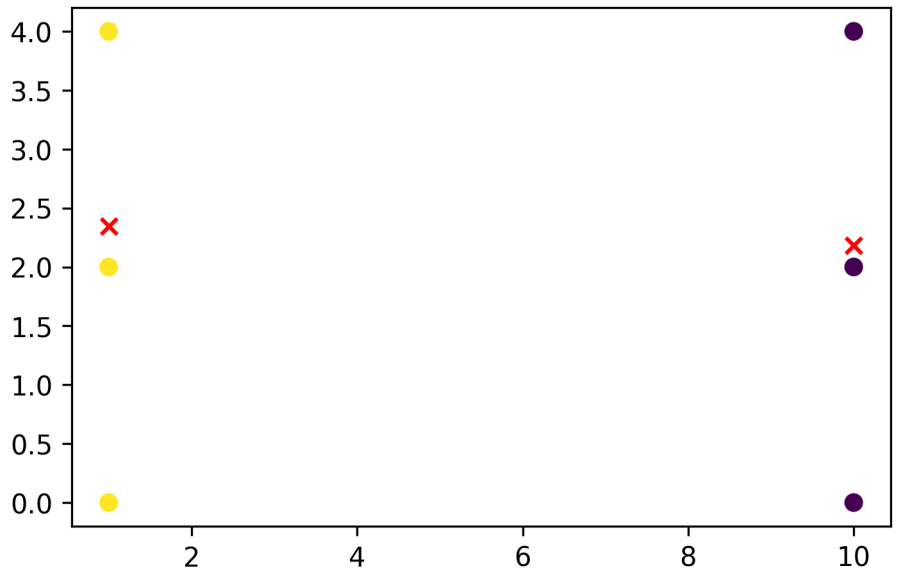
- Uses small random subsets (mini-batches) to update centroids.
- Trades off accuracy for speed.

4.3.2 Benefits

- Faster convergence (3-4x faster)
- Works better with large datasets

```
from sklearn.cluster import MiniBatchKMeans

mb_kmeans = MiniBatchKMeans(n_clusters=2, batch_size=10, random_state=0).fit(X)
print(mb_kmeans.labels_)

# plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=mb_kmeans.labels_, cmap='viridis')
plt.scatter(mb_kmeans.cluster_centers_[:, 0], mb_kmeans.cluster_centers_[:, 1], c='red', mar
plt.show()
```

```
[1 1 1 0 0 0]
```



4.4 Evaluating Clustering Quality

4.4.1 Silhouette Score

Measures how well a point fits its own cluster vs others.

4.4.1.1 Formula:

s = (b - a) / max(a, b)

- **a** = distance to points in the same cluster
- **b** = distance to points in nearest cluster

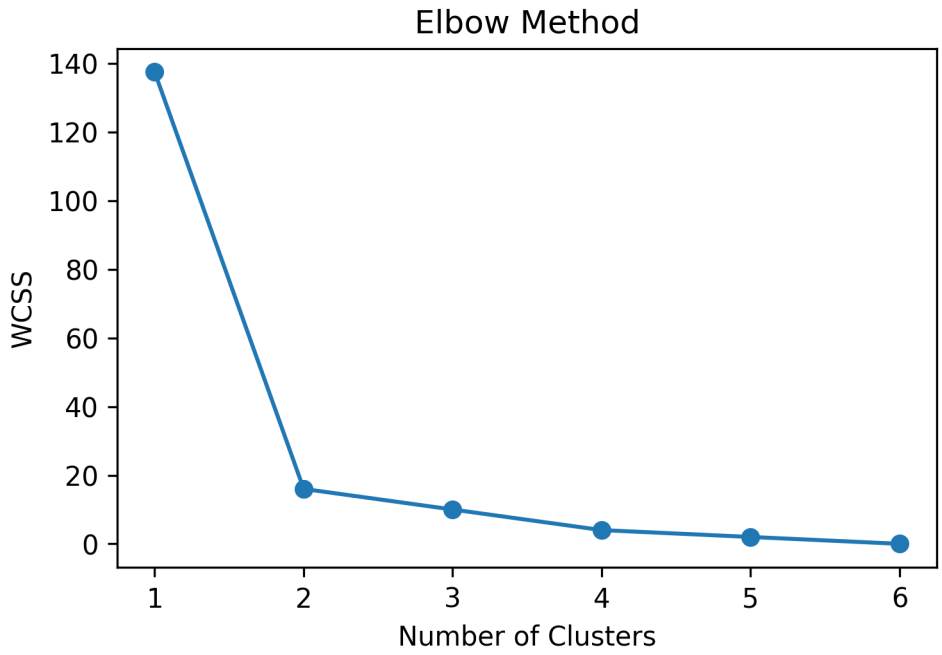
```
from sklearn.metrics import silhouette_score

score = silhouette_score(X, kmeans.labels_)
print(score)
```

- **+1**: well-clustered
- **0**: on boundary
- **-1**: likely misclassified

4.4.2 Elbow Method

Plots **WCSS vs. k**, looks for a point ("elbow") where further increase in k has diminishing returns.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans


wcss = []
max_clusters = min(10, X.shape[0])
for i in range(1, max_clusters + 1):
    kmeans = KMeans(n_clusters=i, random_state=0).fit(X)
    wcss.append(kmeans.inertia_)

plt.plot(range(1, max_clusters + 1), wcss, marker='o')
plt.xlabel("Number of Clusters")
plt.ylabel("WCSS")
plt.title("Elbow Method")
plt.show()
```



4.4.3 Elbow vs Silhouette

Metric	Use When
Elbow Method	Clear elbow point exists
Silhouette Score	When elbow is unclear or clusters are complex

---

4.5 DBSCAN: Density-Based Clustering

4.5.1 What is DBSCAN?

DBSCAN forms clusters based on **density** of data points, identifying core, border, and noise points.

4.5.2 Key Terms

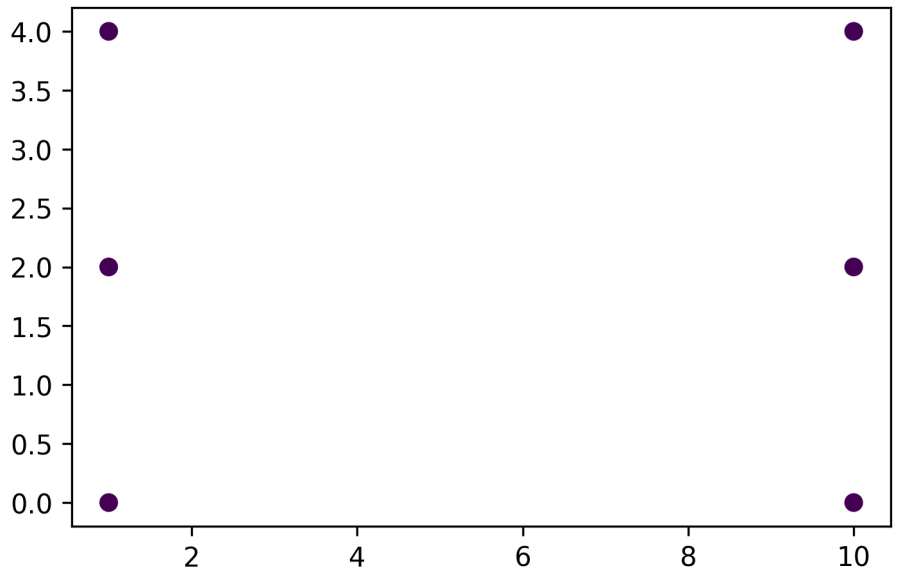
- **Eps ( )**: Neighborhood radius
- **MinPts**: Minimum points to form a dense region
- **Core Point**: MinPts in -neighborhood
- **Border Point**: < MinPts but within of a core point
- **Noise**: Not in any cluster

4.5.3 DBSCAN Steps

1. Identify core points using Eps and MinPts
2. Expand clusters from core points
3. Label non-core/non-border points as noise

```
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.5, min_samples=4).fit(X)
print(db.labels_)

# plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=db.labels_, cmap='viridis')
plt.scatter(db.components_[:, 0], db.components_[:, 1], c='red', marker='x')
plt.show()
```

```
[-1 -1 -1 -1 -1 -1]
```



4.5.4 Pros

- No need to specify k
- Handles **arbitrary shaped clusters**
- Handles **noise and outliers**

4.5.5 Cons

- Sensitive to Eps and MinPts
- Struggles with **varying densities**
- Less effective in **high-dimensional data**

4.5.6 Parameter Tuning

- Plot **k-distance graph** to choose Eps
- MinPts: at least D+1 (D = number of dimensions)

4.5.7 Applications

- Geospatial analysis
- Anomaly detection
- Image segmentation

- Genomic data clustering

4.6 Clustering with Hierarchical Clustering

4.6.1 What is Hierarchical Clustering?

Hierarchical clustering is a technique that builds a tree-like structure of clusters, known as a dendrogram. It doesn't require the number of clusters to be specified upfront, unlike K-means. You start with each data point as its own cluster and progressively merge the closest clusters (agglomerative) or split clusters (divisive) until you reach a stopping point.

4.6.2 Types of Hierarchical Clustering

- **Agglomerative (Bottom-Up)**: Start with individual points as clusters, then merge the closest ones.
- **Divisive (Top-Down)**: Start with all points in one cluster and split it progressively.

4.6.3 Distance Metrics

To merge or split clusters, we need a way to measure distance. Common ones include: - **Euclidean Distance**: Think of it as measuring the straight-line distance between two points, like measuring the shortest path between two cities on a map. - **Manhattan Distance**: The sum of the absolute differences of coordinates, like driving along streets in a grid (no diagonals). - **Cosine Similarity**: Measures how similar two vectors are based on their direction, not magnitude, often used in text.

4.6.4 Linkage Criteria

This defines how we calculate the distance between clusters: - **Single Linkage**: Distance between two clusters is the shortest distance between any two points. - **Complete Linkage**: Distance is the longest distance between points. - **Ward's Linkage**: Minimizes variance within clusters.

4.6.5 Example: Agglomerative Clustering

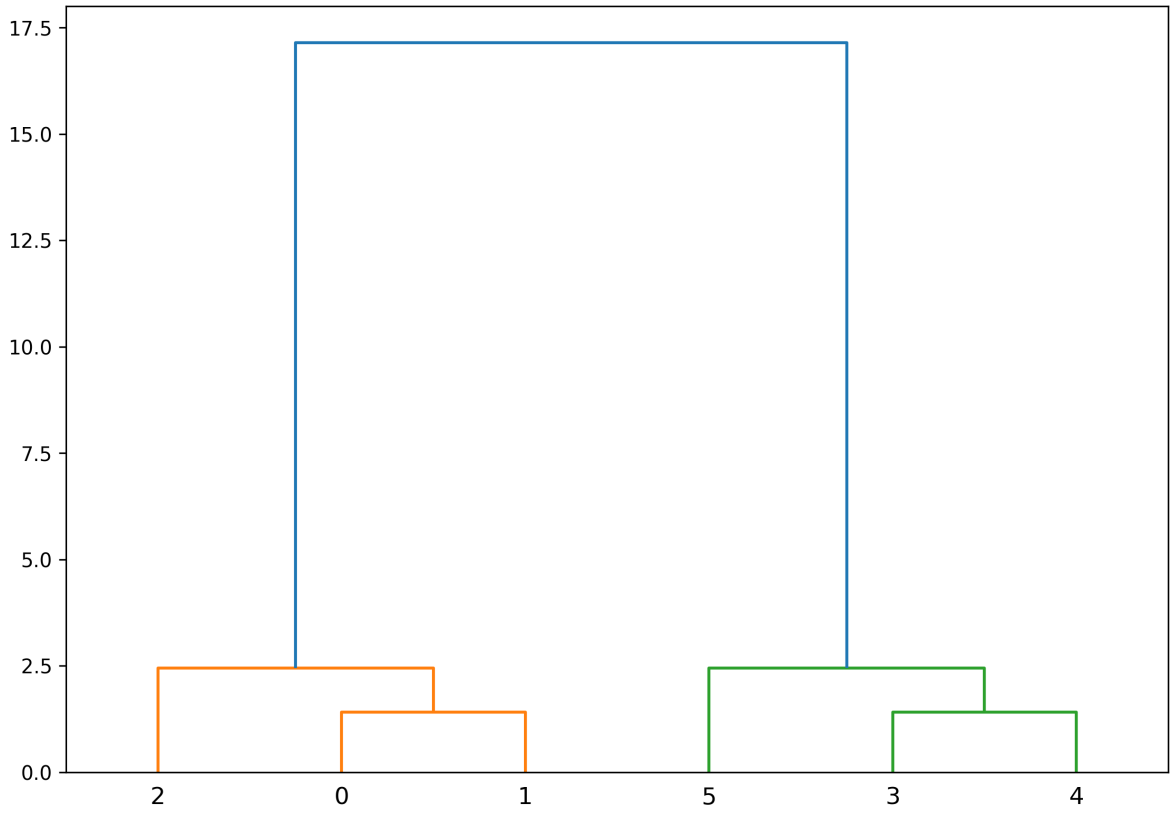
Imagine you have six points (A, B, C, D, E, F) and want to group them. Here's how agglomerative clustering works:

1. **Start with each point as a cluster**: A, B, C, D, E, F.
2. **Merge the closest clusters**: (D, F) at distance 0.50.
3. **Repeat**: Merge (A, B) at distance 0.71.
4. **Continue merging**: Eventually, you'll have one large cluster.

The merging process forms a **dendrogram**, which shows how clusters are joined at different distances.

4.6.6 Python Example

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

# Example data
data = np.array([[1, 2], [2, 3], [3, 4], [8, 9], [9, 10], [10, 11]])

# Perform hierarchical clustering
linked = linkage(data, 'ward')

# Plot dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linked)
plt.show()
```



This code generates a dendrogram that shows how points are merged.

4.6.7 Advantages

No need to predefine clusters: It can find the natural groupings in the data. Dendrogram visualization: Helps in deciding how many clusters to extract. Captures nested clusters: Useful for complex data structures. ### Disadvantages Computationally expensive: $O(n^3)$, so not ideal for large datasets. Sensitive to outliers: Especially with single linkage, outliers can cause chain-like clusters. ### Applications Gene Expression Analysis: Group genes with similar activity patterns. Customer Segmentation: Segment customers based on purchasing behavior. Document Clustering: Group similar text documents.

4.7 Final Notes

- **Always scale your features** before clustering (StandardScaler or MinMaxScaler).

- **Try multiple initializations** for K-means to avoid local optima.
- Use **Silhouette and Elbow methods** for evaluation.
- Use **DBSCAN** when clusters are non-spherical or you expect outliers.

5 Supervised Learning: Regression and Classification

```
# Import necessary libraries for examples
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris, make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, mean_squared_error, confusion_matrix, classificat
```

5.1 Introduction to Supervised Learning

Supervised learning is a type of machine learning where the model learns from labeled data. Each input is paired with the correct output, allowing the model to learn the mapping between them. Once trained, the model can predict outcomes for new, unseen data.

Supervised learning is divided into two main categories:

1. **Classification**: Predicting a categorical output (class label)
2. **Regression**: Predicting a continuous numerical value

5.2 K-Nearest Neighbors (KNN)

5.2.1 Overview

K-Nearest Neighbors (KNN) is a simple yet powerful non-parametric algorithm used for both classification and regression. Introduced by Fix and Hodges in 1951, it works on a fundamental principle: similar data points tend to have similar outputs.

Tip

Real-world analogy: KNN is like asking your closest friends for advice. If you want to know if you'll enjoy a movie, you might ask the opinions of friends whose taste in movies is similar to yours.

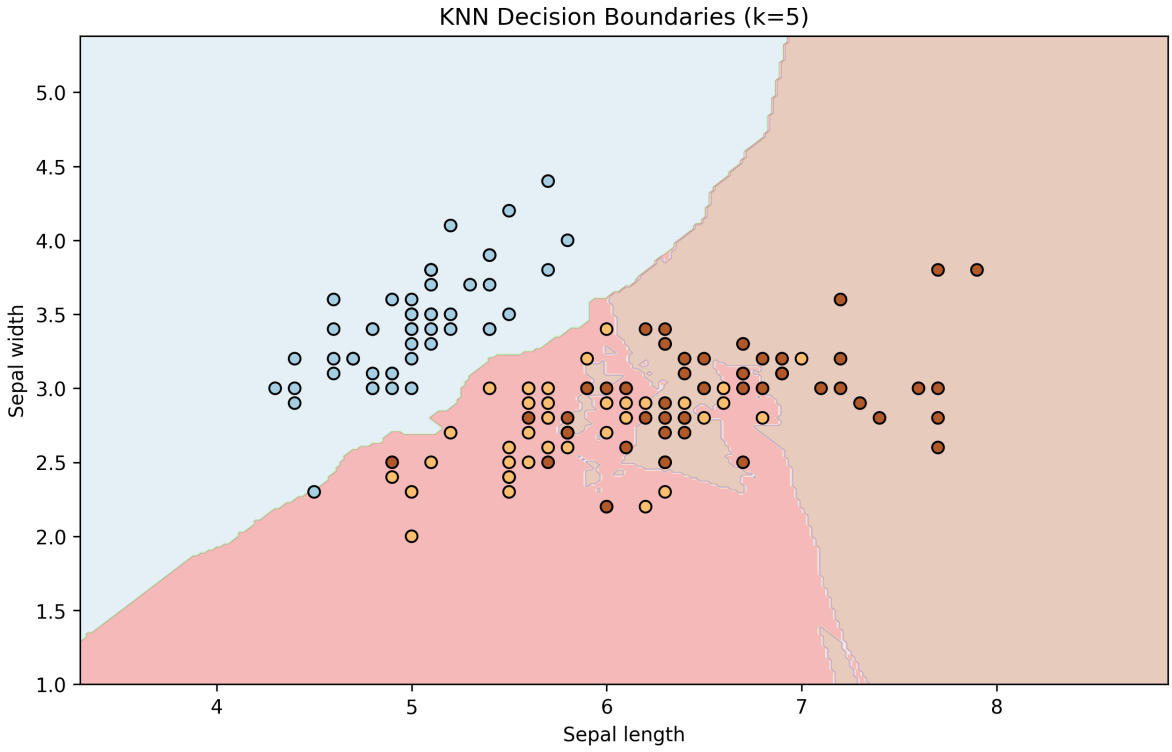
5.2.2 How KNN Works

1. **Distance Calculation**: For a new data point, calculate its distance to all points in the training set
2. **Neighbor Selection**: Select the K nearest neighbors based on those distances
3. **Output Determination**:
    - For classification: Use majority vote of the K neighbors
    - For regression: Take the average of the K neighbors' values

```
# Example: KNN Classification with Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train a KNN classifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Predict and evaluate
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"KNN Classification Accuracy: {accuracy:.4f}")

# Visualize decision boundaries for 2 features
plt.figure(figsize=(10, 6))
# Using only the first two features for visualization
X_vis = X[:, :2]
y_vis = y

# Create mesh grid
h = 0.02  # step size in the mesh
x_min, x_max = X_vis[:, 0].min() - 1, X_vis[:, 0].max() + 1
y_min, y_max = X_vis[:, 1].min() - 1, X_vis[:, 1].max() + 1
```

```
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Train and predict on the 2D data
knn_vis = KNeighborsClassifier(n_neighbors=5)
knn_vis.fit(X_vis, y_vis)
Z = knn_vis.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.Paired)
plt.scatter(X_vis[:, 0], X_vis[:, 1], c=y_vis, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title('KNN Decision Boundaries (k=5)')
plt.show()
```

```
KNN Classification Accuracy: 1.0000
```



5.2.3 Distance Metrics

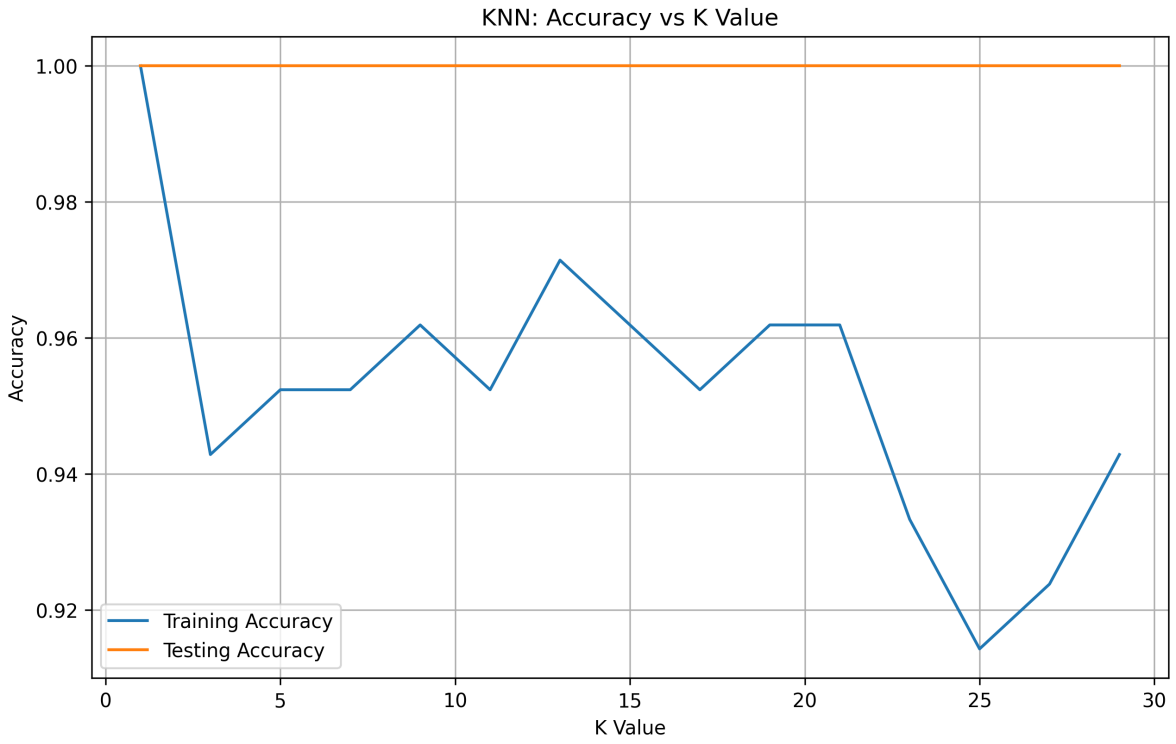
KNN relies on distance metrics to determine similarity:

1. **Euclidean Distance**: $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$ - Straight-line distance
2. **Manhattan Distance**: $\sum_{i=1}^{n}|x_i - y_i|$ - Sum of absolute differences
3. **Minkowski Distance**: A generalization of both Euclidean and Manhattan distances

5.2.4 Choosing K Value

The value of K is critical: - Small K (e.g., K=1): High flexibility, potential overfitting - Large K: Smoother decision boundary, potential underfitting - For classification problems, use an odd K to avoid ties

```
# Example: Effect of K value on KNN classifier performance
k_values = list(range(1, 30, 2))
train_accuracy = []
test_accuracy = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    # Training accuracy
    train_pred = knn.predict(X_train)
    train_acc = accuracy_score(y_train, train_pred)
    train_accuracy.append(train_acc)

    # Testing accuracy
    test_pred = knn.predict(X_test)
    test_acc = accuracy_score(y_test, test_pred)
    test_accuracy.append(test_acc)

plt.figure(figsize=(10, 6))
plt.plot(k_values, train_accuracy, label='Training Accuracy')
plt.plot(k_values, test_accuracy, label='Testing Accuracy')
plt.xlabel('K Value')
plt.ylabel('Accuracy')
plt.title('KNN: Accuracy vs K Value')
plt.legend()
plt.grid(True)
plt.show()
```



5.2.5 Preprocessing for KNN

- **Feature Scaling**: Essential since KNN uses distance calculations
- **Dimensionality Reduction**: Helpful for high-dimensional data
- **Missing Value Imputation**: KNN itself can be used for this purpose

5.2.6 Advantages and Disadvantages

Advantages: - Simple, intuitive algorithm - No training phase (lazy learning) - Non-parametric (no assumptions about data distribution) - Works well for complex decision boundaries

Disadvantages: - Computationally expensive for large datasets - Suffers from the curse of dimensionality - Sensitive to noisy data and outliers - Requires feature scaling

5.2.7 Applications

- Recommendation systems
- Pattern recognition

- Anomaly detection
- Medical diagnosis
- Financial predictions

5.3 Linear Regression

5.3.1 Overview

Linear regression is a fundamental algorithm that models the relationship between a dependent variable and one or more independent variables by fitting a linear equation.

> Tip
>
> **Real-world analogy**: Linear regression is like drawing a "line of best fit" through scattered points on a graph. If you plot house sizes vs. prices, linear regression finds the straight line that best represents the relationship.

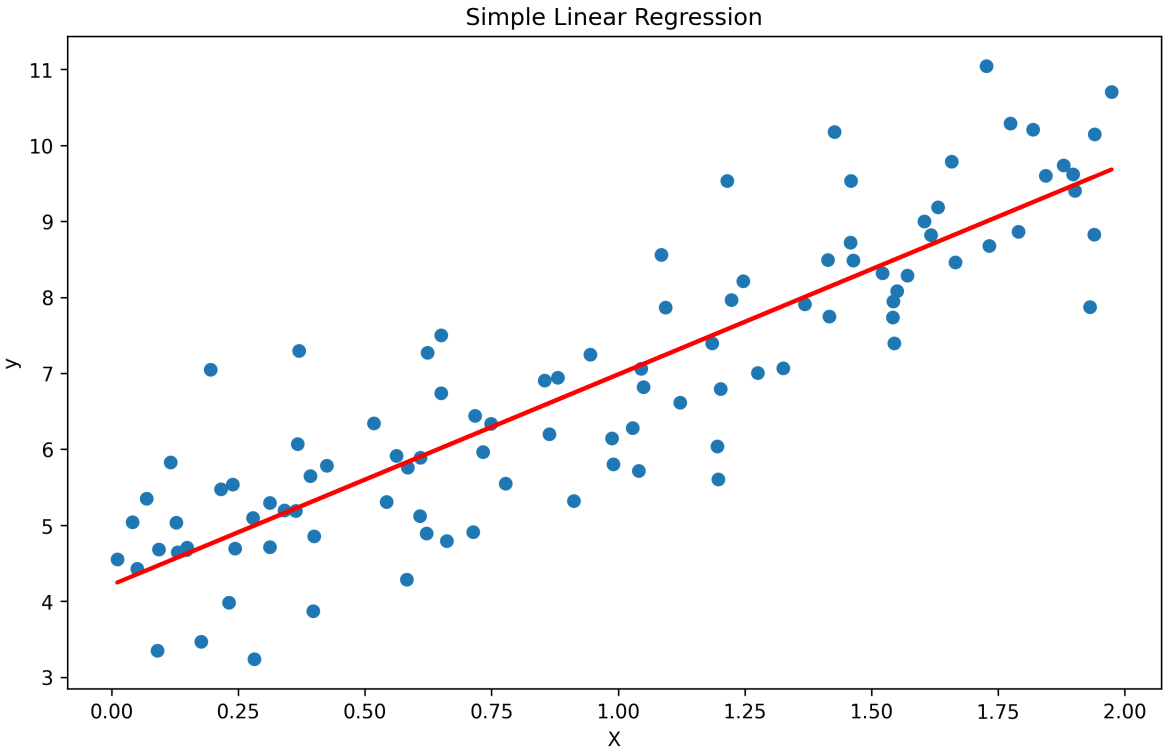
5.3.2 The Model

- **Simple Linear Regression**: $y = mx + b$
  - One independent variable
  - y is the predicted value
  - m is the slope
  - b is the y-intercept
- **Multiple Linear Regression**: $y = b + w_1x_1 + w_2x_2 + ... + w_nx_n$
  - Multiple independent variables
  - b is the bias (y-intercept)
  - w_i are the weights for each feature
  - x_i are the input features

```
# Example: Simple Linear Regression
# Generate synthetic data
np.random.seed(42)
X_simple = 2 * np.random.rand(100, 1)
y_simple = 4 + 3 * X_simple + np.random.randn(100, 1)

# Fit linear regression model
model = LinearRegression()
model.fit(X_simple, y_simple)
```

```
# Print coefficients
print(f"Intercept: {model.intercept_[0]:.4f}")
print(f"Slope: {model.coef_[0][0]:.4f}")

# Visualize
plt.figure(figsize=(10, 6))
plt.scatter(X_simple, y_simple)
plt.plot(X_simple, model.predict(X_simple), color='red', linewidth=2)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Simple Linear Regression')
plt.show()

# Example: Multiple Linear Regression
# Generate synthetic data
np.random.seed(42)
X_multi = np.random.rand(100, 3)
y_multi = 4 + X_multi[:, 0]*2 + X_multi[:, 1]*3 + X_multi[:, 2]*1.5 + np.random.randn(100)*0

# Fit multiple linear regression model
multi_model = LinearRegression()
multi_model.fit(X_multi, y_multi)

# Print coefficients
print(f"Intercept: {multi_model.intercept_:.4f}")
print(f"Coefficients: {multi_model.coef_}")

# Evaluate
y_pred = multi_model.predict(X_multi)
mse = mean_squared_error(y_multi, y_pred)
print(f"Mean Squared Error: {mse:.4f}")
```

```
Intercept: 4.2151
Slope: 2.7701
```




```
Intercept: 3.8675
Coefficients: [2.13900206 2.9211189  1.78520288]
Mean Squared Error: 0.2301
```

5.3.3 Loss Function and Optimization

The objective of linear regression is to find the line (or hyperplane) that minimizes the differences between predicted and actual values. This is typically done using:

- **Mean Squared Error (MSE)**: $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_i')^2$
- **Optimization Algorithm**: Usually Gradient Descent or Normal Equation

5.3.4 Assumptions

Linear regression makes several assumptions: 1. Linearity between independent and dependent variables 2. Independence of observations 3. Homoscedasticity (constant variance of errors) 4. Normal distribution of residuals 5. No or minimal multicollinearity

5.3.5 Advantages and Disadvantages

Advantages: - Simple and interpretable - Computationally efficient - Provides clear insight into feature importance - Good baseline for regression problems

Disadvantages: - Limited to linear relationships - Sensitive to outliers - Assumes independence of features - Restrictive assumptions

5.3.6 Applications

- Sales forecasting
- Price prediction (e.g., housing prices)
- Financial analysis
- Medical outcome prediction
- Resource allocation

5.4 Logistic Regression

5.4.1 Overview

Despite its name, logistic regression is a classification algorithm. It estimates the probability that an instance belongs to a particular class using the logistic function to transform a linear combination of features.

> Tip
>
> **Real-world analogy**: Logistic regression is like determining the probability of passing an exam based on hours studied. There's a threshold (e.g., studying 10 hours) where the probability of passing jumps dramatically.

5.4.2 The Logistic Function

The logistic (sigmoid) function transforms a linear equation into a probability between 0 and 1:

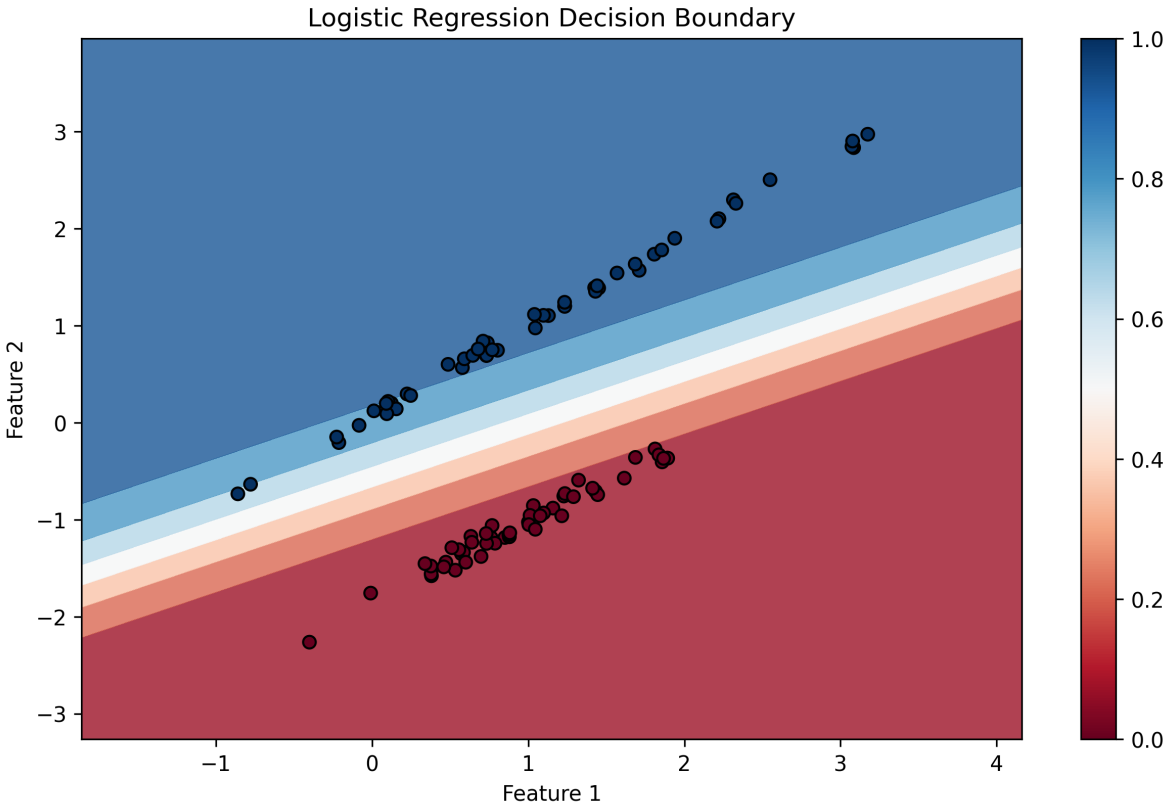
$$P(y = 1|X) = \frac{1}{1 + e^{-(b+w_1x_1+w_2x_2+...+w_nx_n)}}$$

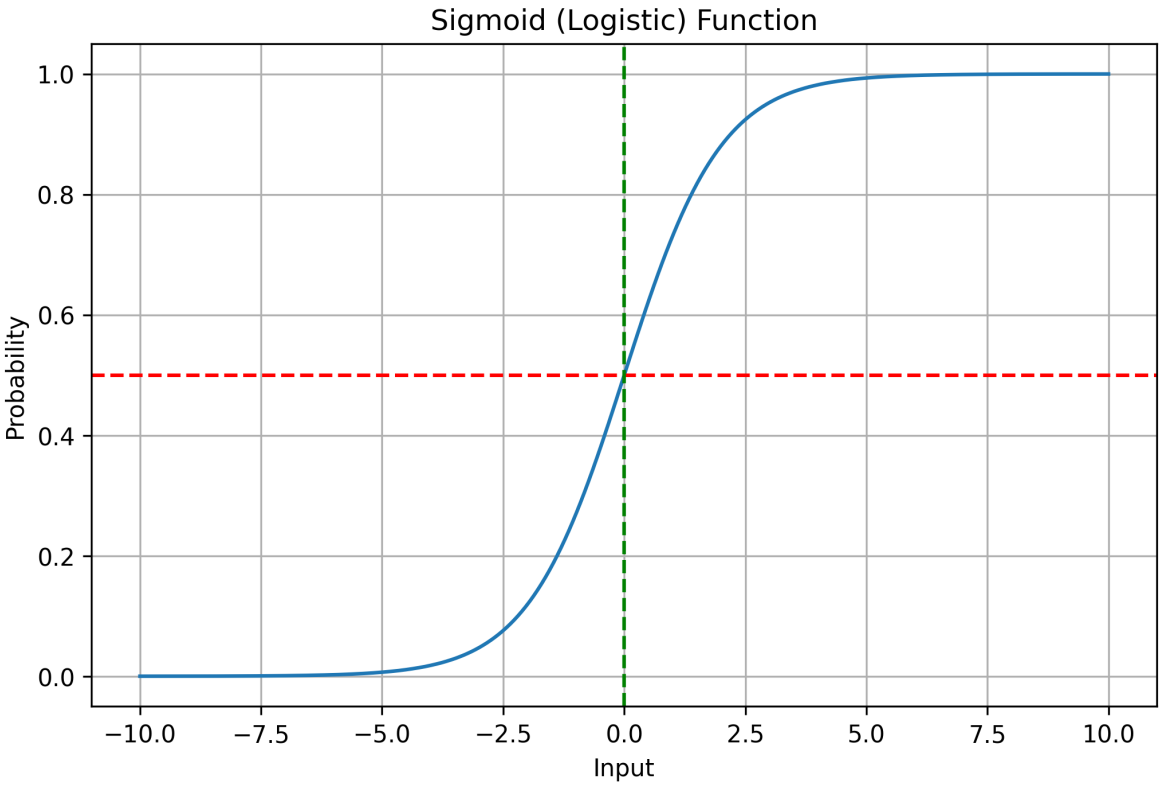
Where: - $P(y = 1|X)$ is the probability of the positive class - b is the bias term (intercept) - w_i are the weights for each feature - x_i are the input features

```
# Example: Logistic Regression
# Generate classification dataset
X, y = make_classification(n_samples=100, n_features=2, n_redundant=0,
                           n_informative=2, random_state=42, n_clusters_per_class=1)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train logistic regression model
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)

# Predict and evaluate
y_pred = log_reg.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Logistic Regression Accuracy: {accuracy:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Visualize decision boundary
plt.figure(figsize=(10, 6))
# Create mesh grid
h = 0.02  # step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Predict probabilities
Z = log_reg.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
Z = Z.reshape(xx.shape)

# Plot decision boundary and points
plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.RdBu)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Logistic Regression Decision Boundary')
plt.colorbar()
plt.show()

# Visualize sigmoid function
def sigmoid(x):
```

```
    return 1 / (1 + np.exp(-x))

x = np.linspace(-10, 10, 1000)
y = sigmoid(x)

plt.figure(figsize=(8, 5))
plt.plot(x, y)
plt.grid(True)
plt.title('Sigmoid (Logistic) Function')
plt.xlabel('Input')
plt.ylabel('Probability')
plt.axhline(y=0.5, color='r', linestyle='--')
plt.axvline(x=0, color='g', linestyle='--')
plt.show()
```

```
Logistic Regression Accuracy: 1.0000

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       1.00      1.00      1.00        15

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```


Logistic Regression Decision Boundary
Feature 1
Feature 2
3
2
1
0
−1
−2
−3
−1
0
1
2
3
4
1.0
0.8
0.6
0.4
0.2
0.0



5.4.3 Types of Logistic Regression

1. **Binary Logistic Regression**: Two possible outcomes (e.g., spam/not spam)
2. **Multinomial Logistic Regression**: Multiple classes without order (e.g., predicting fruit type)
3. **Ordinal Logistic Regression**: Multiple classes with order (e.g., movie ratings)

5.4.4 Maximum Likelihood Estimation

Logistic regression uses Maximum Likelihood Estimation (MLE) to find the optimal parameters. It seeks to maximize the probability of observing the data given the parameters.

5.4.5 Evaluation Metrics

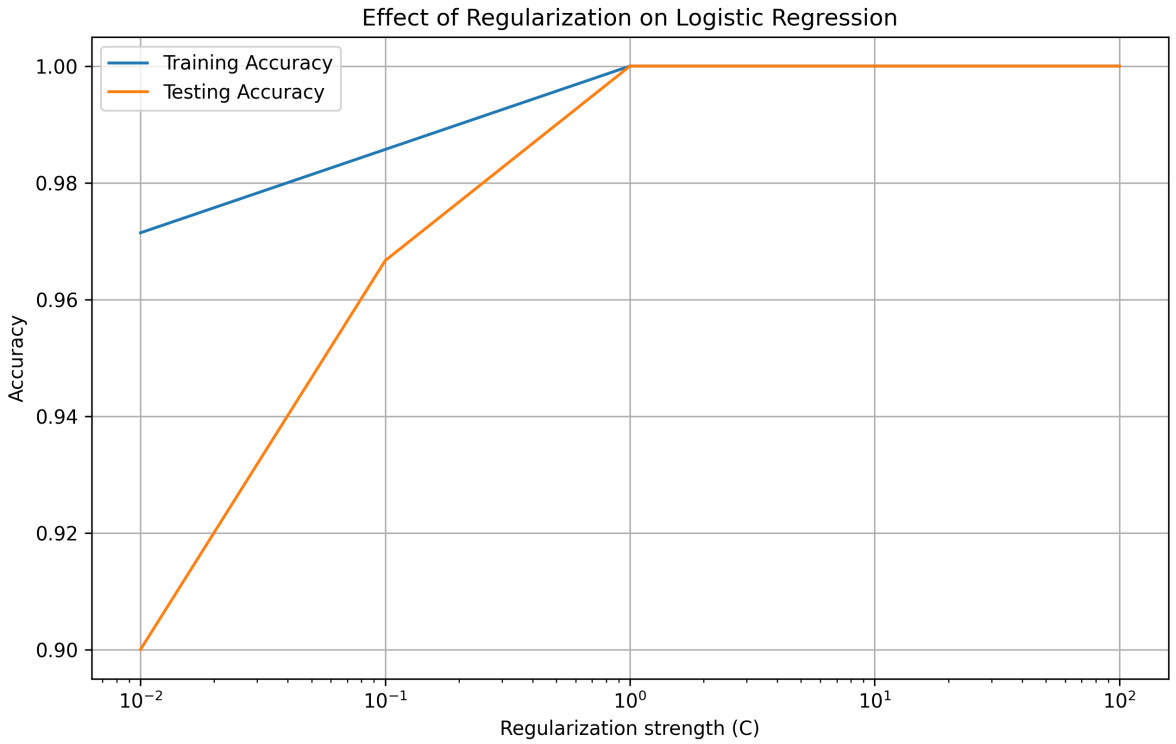
- **Accuracy**: Proportion of correct predictions
- **Precision**: True positives / (True positives + False positives)
- **Recall**: True positives / (True positives + False negatives)

- **F1 Score**: Harmonic mean of precision and recall
- **AUC-ROC**: Area under the Receiver Operating Characteristic curve

5.4.6 Regularization

Logistic regression can benefit from regularization to prevent overfitting: - **L1 Regularization (Lasso)**: Adds absolute value of coefficients to loss function - **L2 Regularization (Ridge)**: Adds squared value of coefficients to loss function

```
# Example: Logistic Regression with Regularization
# Comparing different regularization strengths
C_values = [0.01, 0.1, 1, 10, 100]
train_scores = []
test_scores = []

for C in C_values:
    # C is inverse of regularization strength
    log_reg = LogisticRegression(C=C, max_iter=1000)
    log_reg.fit(X_train, y_train)

    # Score on training data
    train_score = log_reg.score(X_train, y_train)
    train_scores.append(train_score)

    # Score on test data
    test_score = log_reg.score(X_test, y_test)
    test_scores.append(test_score)

plt.figure(figsize=(10, 6))
plt.semilogx(C_values, train_scores, label='Training Accuracy')
plt.semilogx(C_values, test_scores, label='Testing Accuracy')
plt.xlabel('Regularization strength (C)')
plt.ylabel('Accuracy')
plt.title('Effect of Regularization on Logistic Regression')
plt.legend()
plt.grid(True)
plt.show()
```



5.4.7 Advantages and Disadvantages

Advantages: - Outputs probabilities - Highly interpretable coefficients - Efficient training - Works well for linearly separable classes - Can be extended to multi-class problems

Disadvantages: - Limited to linear decision boundaries - Requires feature engineering for complex relationships - Can suffer from complete separation issues - Sensitive to outliers and imbalanced data

5.4.8 Applications

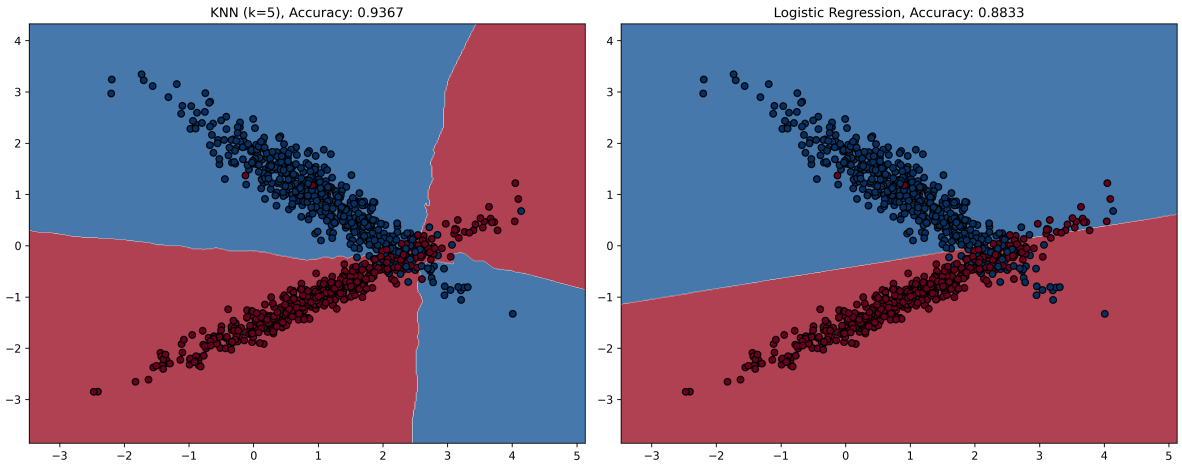
- Credit scoring
- Medical diagnosis
- Email spam detection
- Customer churn prediction
- Marketing campaign effectiveness

5.5 Comparison of Algorithms

```
# Generate a dataset for comparison
np.random.seed(42)
X, y = make_classification(n_samples=1000, n_features=2, n_redundant=0,
                           n_informative=2, random_state=42, n_clusters_per_class=1)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train models
knn_model = KNeighborsClassifier(n_neighbors=5)
log_reg_model = LogisticRegression()

knn_model.fit(X_train, y_train)
log_reg_model.fit(X_train, y_train)

# Predict
knn_pred = knn_model.predict(X_test)
log_reg_pred = log_reg_model.predict(X_test)

# Evaluate
knn_acc = accuracy_score(y_test, knn_pred)
log_reg_acc = accuracy_score(y_test, log_reg_pred)

print(f"KNN Accuracy: {knn_acc:.4f}")
print(f"Logistic Regression Accuracy: {log_reg_acc:.4f}")

# Visualize decision boundaries
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Create mesh grid
h = 0.02  # step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# KNN decision boundary
Z = knn_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
ax1.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)
```

```
ax1.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.RdBu)
ax1.set_title(f'KNN (k=5), Accuracy: {knn_acc:.4f}')

# Logistic Regression decision boundary
Z = log_reg_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
ax2.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)
ax2.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.RdBu)
ax2.set_title(f'Logistic Regression, Accuracy: {log_reg_acc:.4f}')

plt.tight_layout()
plt.show()
```

```
KNN Accuracy: 0.9367
Logistic Regression Accuracy: 0.8833
```



5.6 Summary

This lecture covered three fundamental supervised learning algorithms:

1. **K-Nearest Neighbors (KNN)**
    - Non-parametric instance-based learning
    - Makes predictions based on similarity (distance metrics)
    - Works for both classification and regression
    - Simple but powerful for many applications

2. **Linear Regression**

   - Models linear relationship between dependent and independent variables
   - Uses least squares method to minimize errors
   - Best for numeric predictions where relationships are linear
   - Provides interpretable coefficients

3. **Logistic Regression**

   - Despite its name, used for classification problems
   - Estimates probabilities using the sigmoid function
   - Outputs values between 0 and 1 (probabilities)
   - Works well for binary and multi-class classification

5.6.1 Choosing the Right Algorithm

- Use **KNN** when:
  - The decision boundary is complex
  - You need a simple, interpretable model
  - The dataset is small to medium sized
  - Preprocessing and feature scaling are properly applied
- Use **Linear Regression** when:
  - The relationship between variables is approximately linear
  - You need a continuous numerical prediction
  - Interpretability of coefficients is important
  - You need a baseline regression model
- Use **Logistic Regression** when:
  - You need class probabilities as output
  - The decision boundary is approximately linear
  - You need an interpretable model
  - Feature importance is of interest

6 Dimensionality Reduction Methods

Dimensionality reduction is a critical technique in data analysis and machine learning that reduces the number of input variables (features) while preserving essential information. High-dimensional datasets often contain redundancy or noise that can be eliminated through these methods.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris, fetch_openml
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA, KernelPCA, IncrementalPCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.manifold import TSNE
import plotly.express as px

# Set plotting styles
sns.set_style("whitegrid")
plt.rcParams['figure.figsize'] = (10, 6)
```

Figure 6.1

6.1 Why Reduce Dimensionality?

The primary goals of dimensionality reduction include:

1. **Reducing overfitting** by eliminating noise and redundant features
2. **Improving computational efficiency** for faster, less expensive algorithms
3. **Enabling data visualization** by mapping to 2D or 3D spaces
4. **Removing noise** to focus on meaningful patterns

6.2 Dataset Example: Iris

Let's load and examine the Iris dataset, which we'll use throughout this document:

```
# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

# Create a DataFrame for easier manipulation
iris_df = pd.DataFrame(X, columns=feature_names)
iris_df['species'] = [target_names[i] for i in y]

# Display dataset information
print(f"Dataset shape: {X.shape}")
print(f"Features: {feature_names}")
print(f"Number of samples per class: {np.bincount(y)}")

# Preview the dataset
iris_df.head()
```

```
Dataset shape: (150, 4)
Features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Number of samples per class: [50 50 50]
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

6.3 Approaches to Dimensionality Reduction

Dimensionality reduction methods fall into two main categories:

6.3.1 Unsupervised Methods

These techniques don't require labeled data and find lower-dimensional representations based solely on the intrinsic structure of features.

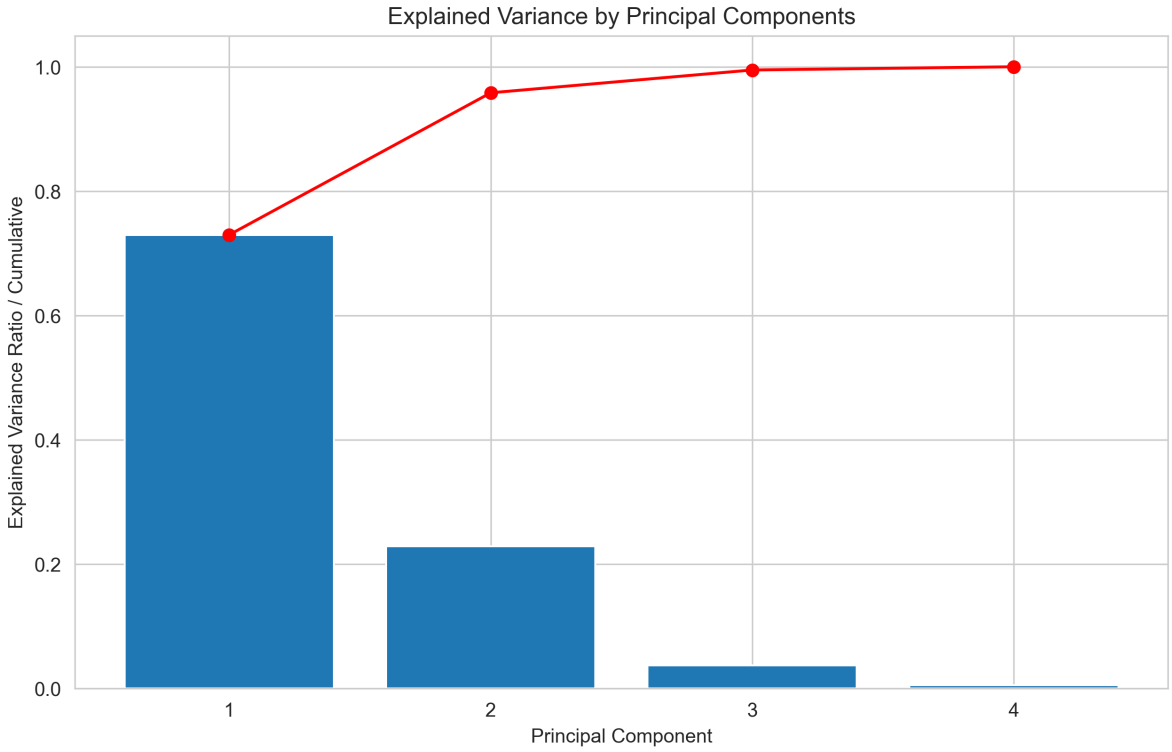
6.3.1.1 Principal Component Analysis (PCA)

PCA identifies directions (principal components) where data varies the most and projects data onto this lower-dimensional space.

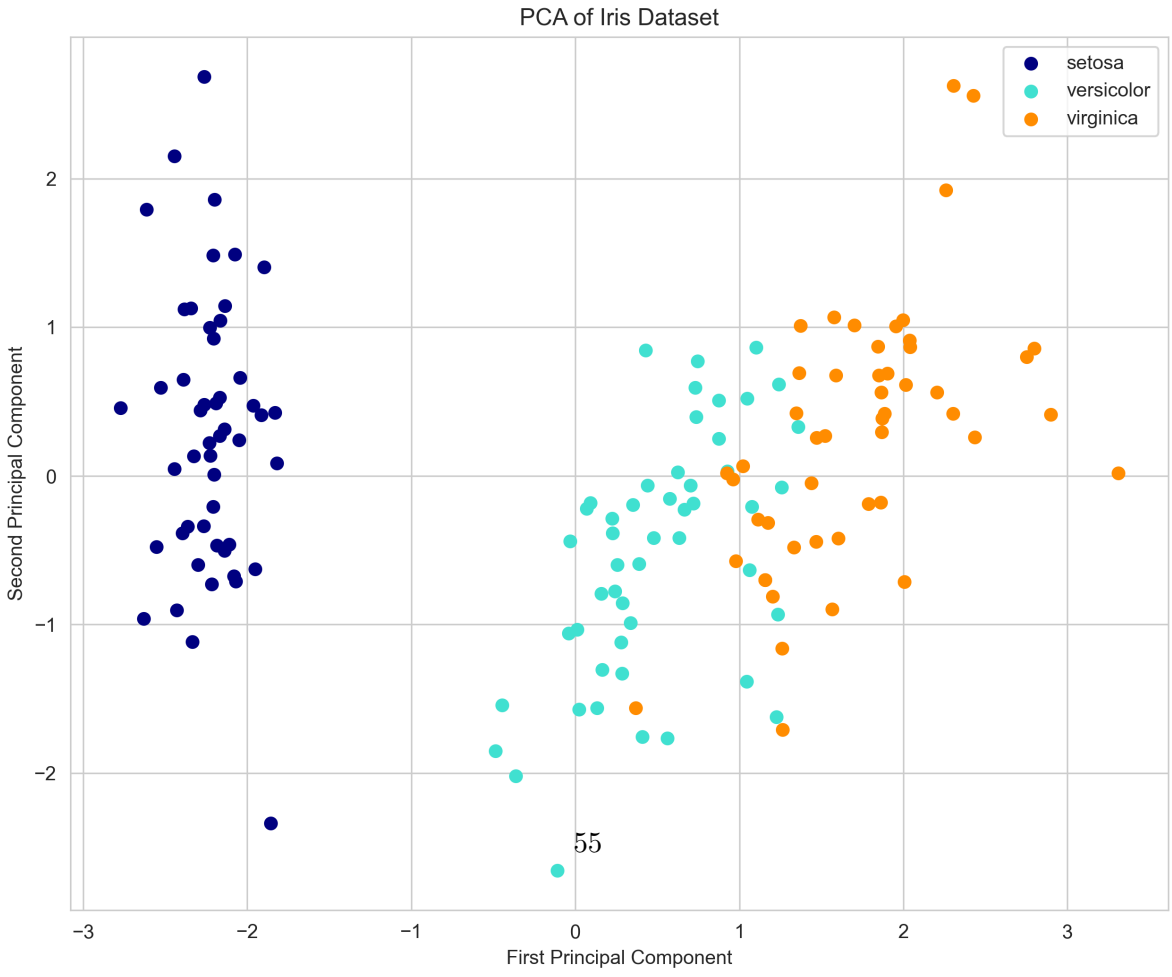
```
# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Plot explained variance
plt.figure(figsize=(10, 6))
plt.bar(range(1, len(pca.explained_variance_ratio_) + 1), pca.explained_variance_ratio_)
plt.plot(range(1, len(pca.explained_variance_ratio_) + 1),
         np.cumsum(pca.explained_variance_ratio_), 'r-o')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio / Cumulative')
plt.xticks(range(1, len(pca.explained_variance_ratio_) + 1))
plt.title('Explained Variance by Principal Components')
plt.grid(True)
plt.show()

# Print variance explained
print(f"Variance explained by each component: {pca.explained_variance_ratio_}")
print(f"Cumulative variance explained: {np.cumsum(pca.explained_variance_ratio_)}")

# Visualization in 2D
plt.figure(figsize=(10, 8))
colors = ['navy', 'turquoise', 'darkorange']
for i, c, label in zip(range(3), colors, target_names):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], c=c, label=label)
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title('PCA of Iris Dataset')
```

```
plt.legend()
plt.grid(True)
plt.show()
```

```
Variance explained by each component: [0.72962445 0.22850762 0.03668922 0.00517871]
Cumulative variance explained: [0.72962445 0.95813207 0.99482129 1.        ]
```



(a) PCA visualization of the Iris dataset



(b)

Figure 6.2

6.3.1.1.1 Key Concepts of PCA

1. **Variance Maximization**: Captures directions with maximum variance
2. **Linear Combinations**: Each PC is a weighted sum of original features
3. **Uncorrelated Components**: PCs are orthogonal to each other
4. **Coordinate Transformation**: Rotates data into a new coordinate system

Let's explore the relationship between original features and principal components:

```
# Display the component loadings
loadings = pd.DataFrame(
    pca.components_.T,
    columns=[f'PC{i+1}' for i in range(pca.components_.shape[0])],
    index=feature_names
)

# Visualize the loadings
plt.figure(figsize=(10, 6))
sns.heatmap(loadings, annot=True, cmap='coolwarm', fmt='.3f')
plt.title('PCA Component Loadings')
plt.tight_layout()
plt.show()

# Determine optimal number of components for 95% variance
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
print(f"Number of components needed for 95% variance: {d}")
```

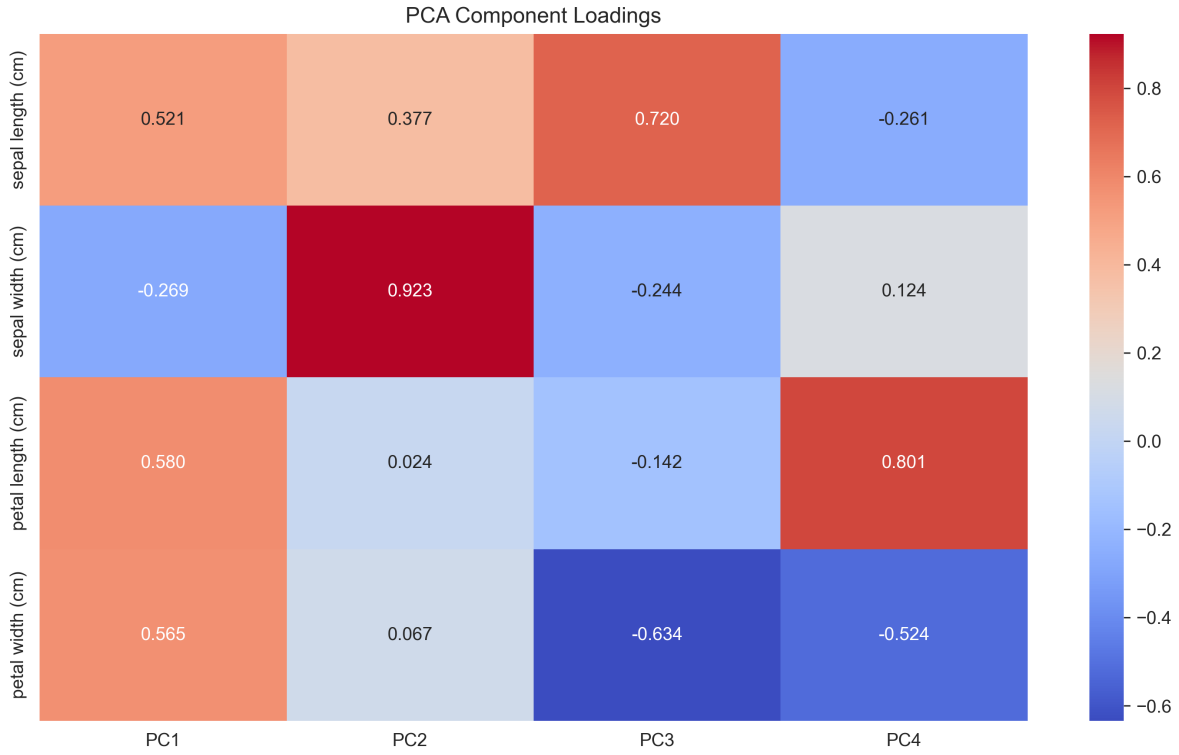


Figure 6.3: PCA components and their relationship to original features

```
Number of components needed for 95% variance: 2
```

6.3.1.2 Other Unsupervised Methods

- **Independent Component Analysis (ICA)**: Focuses on statistical independence of components
- **Non-negative Matrix Factorization (NMF)**: Factorizes data into non-negative matrices

6.3.2 Supervised Methods

These techniques consider class labels during dimensionality reduction to better preserve class separability.

6.3.2.1 Linear Discriminant Analysis (LDA)

LDA maximizes class separation by projecting data onto a lower-dimensional space:

```
# Apply LDA
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X_scaled, y)

# Visualization in 2D
plt.figure(figsize=(10, 8))
for i, c, label in zip(range(3), colors, target_names):
    plt.scatter(X_lda[y == i, 0], X_lda[y == i, 1], c=c, label=label)
plt.xlabel('First LDA Component')
plt.ylabel('Second LDA Component')
plt.title('LDA of Iris Dataset')
plt.legend()
plt.grid(True)
plt.show()

# Check explained variance ratio
print(f"Explained variance ratio: {lda.explained_variance_ratio_}")
```

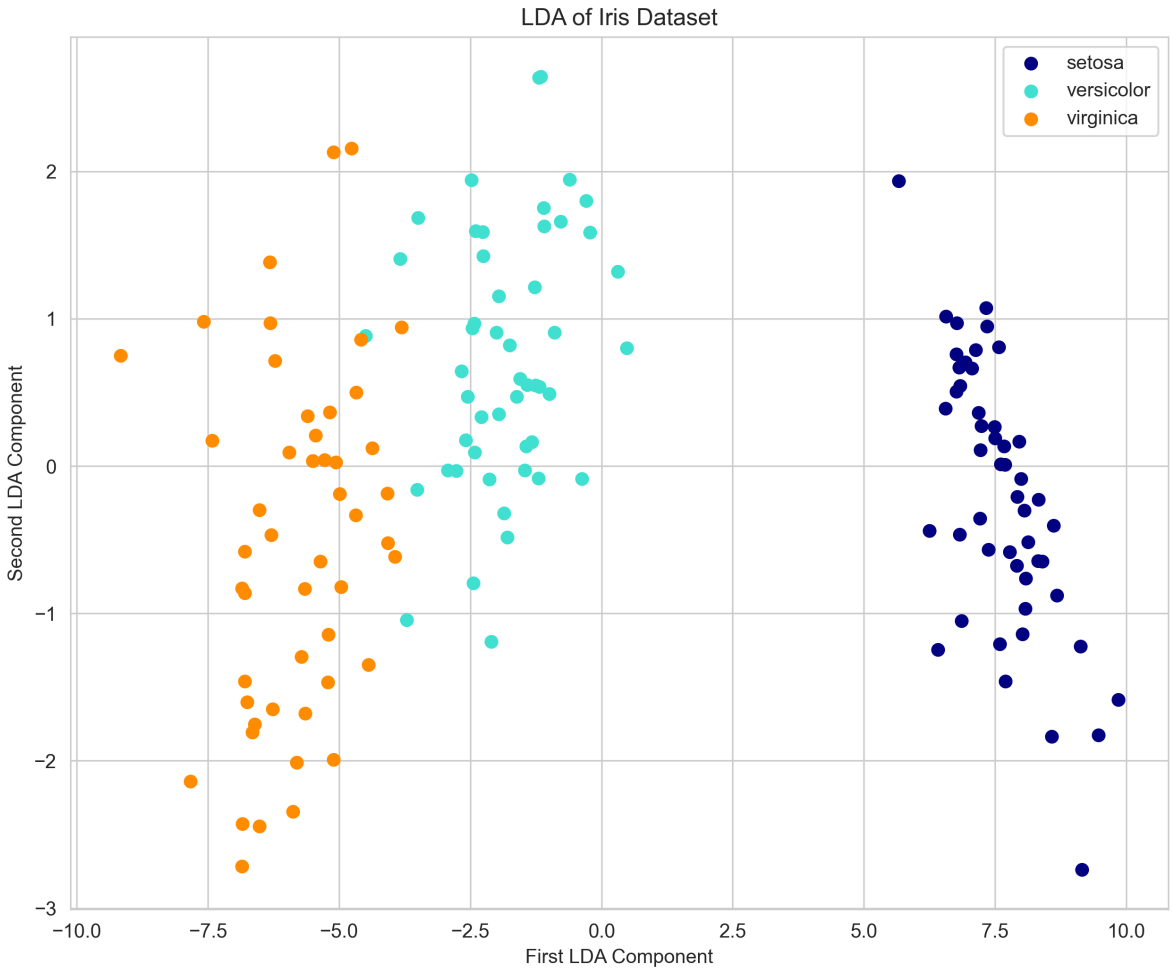


Figure 6.4: LDA visualization of the Iris dataset

```
Explained variance ratio: [0.9912126 0.0087874]
```

6.4 Advanced PCA Implementations

6.4.1 Kernel PCA

When data is not linearly separable, Kernel PCA can be more effective:

```
# Apply Kernel PCA with different kernels
kernels = ['linear', 'poly', 'rbf']
fig, axes = plt.subplots(1, len(kernels), figsize=(18, 5))
```

```
for i, kernel in enumerate(kernels):
    kpca = KernelPCA(n_components=2, kernel=kernel)
    X_kpca = kpca.fit_transform(X_scaled)

    for j, c, label in zip(range(3), colors, target_names):
        axes[i].scatter(X_kpca[y == j, 0], X_kpca[y == j, 1], c=c, label=label)

    axes[i].set_xlabel('First Component')
    axes[i].set_ylabel('Second Component')
    axes[i].set_title(f'Kernel PCA ({kernel})')
    axes[i].legend()
    axes[i].grid(True)

plt.tight_layout()
plt.show()
```

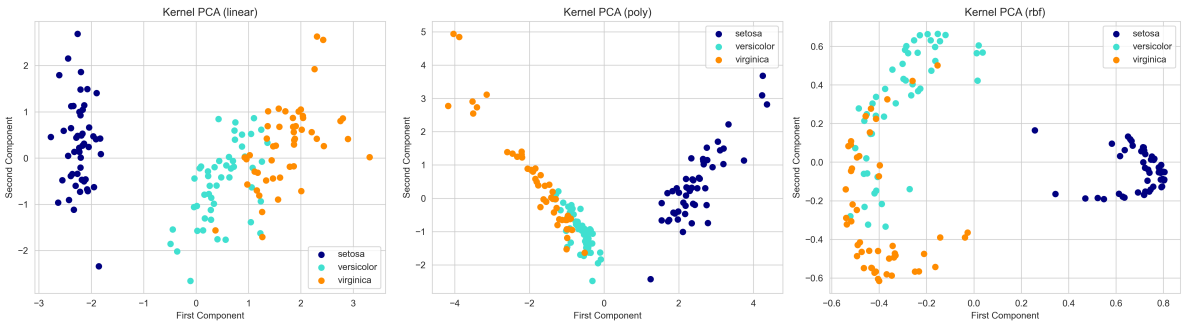


Figure 6.5: Comparison of PCA and Kernel PCA

6.4.2 Incremental PCA

For larger datasets, Incremental PCA processes data in batches:

```
# Simulate a larger dataset by repeating Iris
X_large = np.vstack([X_scaled] * 10)
y_large = np.hstack([y] * 10)

# Apply Incremental PCA
n_batches = 5
batch_size = X_large.shape[0] // n_batches
```

```
ipca = IncrementalPCA(n_components=2)
for i in range(n_batches):
    start = i * batch_size
    end = min((i + 1) * batch_size, X_large.shape[0])
    ipca.partial_fit(X_large[start:end])

X_ipca = ipca.transform(X_large)

# Compare with standard PCA
pca = PCA(n_components=2)
X_pca_large = pca.fit_transform(X_large)

# Visualize both
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Plot IPCA
for i, c, label in zip(range(3), colors, target_names):
    axes[0].scatter(X_ipca[y_large == i, 0], X_ipca[y_large == i, 1], c=c, label=label, alpha
axes[0].set_title('Incremental PCA')
axes[0].legend()
axes[0].grid(True)

# Plot standard PCA
for i, c, label in zip(range(3), colors, target_names):
    axes[1].scatter(X_pca_large[y_large == i, 0], X_pca_large[y_large == i, 1], c=c, label=la
axes[1].set_title('Standard PCA')
axes[1].legend()
axes[1].grid(True)

plt.tight_layout()
plt.show()

# Compare explained variance
print(f"IPCA explained variance: {ipca.explained_variance_ratio_}")
print(f"PCA explained variance: {pca.explained_variance_ratio_}")
```

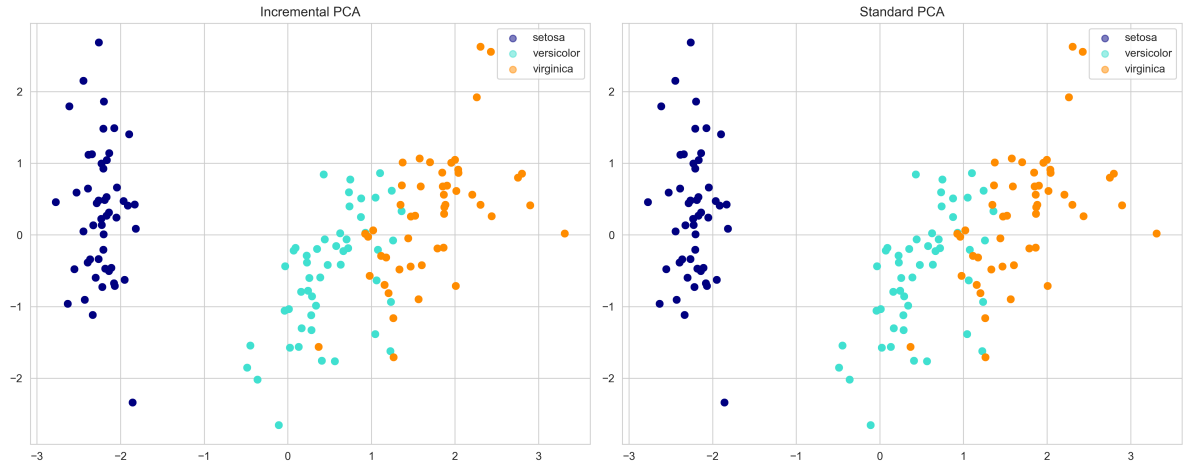


Figure 6.6

```
IPCA explained variance: [0.72962445 0.22850762]
PCA explained variance: [0.72962445 0.22850762]
```

6.5 Visualizing High-Dimensional Data with t-SNE

t-SNE is particularly effective for visualizing high-dimensional data:

```python
# Apply t-SNE
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_scaled)

# Create visualization
plt.figure(figsize=(10, 8))
for i, c, label in zip(range(3), colors, target_names):
    plt.scatter(X_tsne[y == i, 0], X_tsne[y == i, 1], c=c, label=label)
plt.title('t-SNE of Iris Dataset')
plt.legend()
plt.grid(True)
plt.show()
```

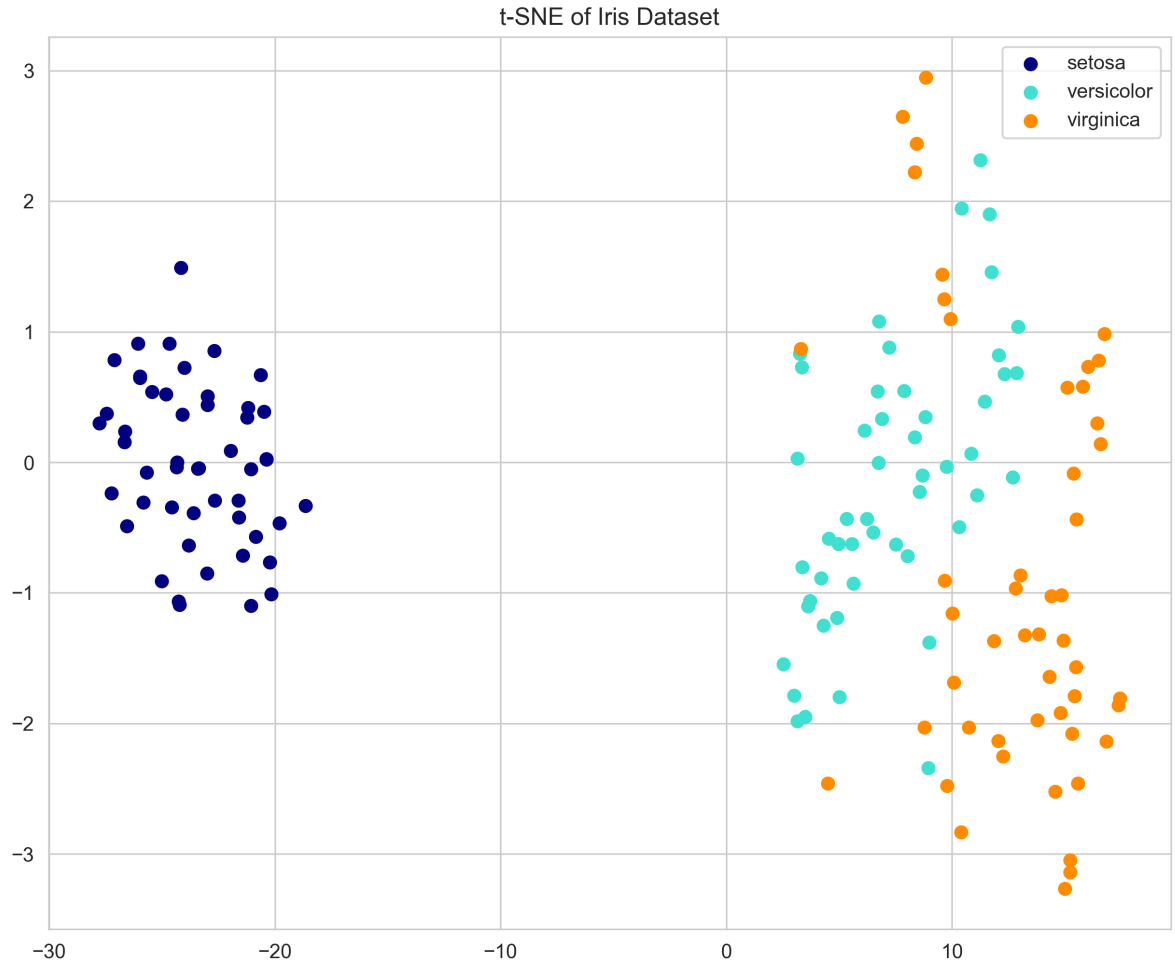


Figure 6.7: t-SNE visualization of the Iris dataset

6.6 Real-World Application: MNIST Dataset

Let's apply dimensionality reduction to a larger, more complex dataset:

6.7 Interactive 3D Visualization with Plotly

Plotly enables interactive exploration of dimensionality reduction results:

```
# Load a subset of MNIST for demonstration
mnist = fetch_openml('mnist_784', version=1, as_frame=False, parser='auto')
X_mnist = mnist.data[:2000]
y_mnist = mnist.target[:2000].astype(int)

# Standardize the data
scaler = StandardScaler()
X_mnist_scaled = scaler.fit_transform(X_mnist)

# Apply PCA
pca_mnist = PCA(n_components=50)  # Reduce from 784 to 50 dimensions
X_mnist_pca = pca_mnist.fit_transform(X_mnist_scaled)

# Plot explained variance
plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(pca_mnist.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance vs. Number of PCA Components (MNIST)')
plt.grid(True)
plt.show()

# Check how many components needed for 95% variance
cumsum = np.cumsum(pca_mnist.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
print(f"Number of components needed for 95% variance: {d}")

# Visualize first two components
plt.figure(figsize=(10, 8))
for i in range(10):
    plt.scatter(X_mnist_pca[y_mnist == i, 0], X_mnist_pca[y_mnist == i, 1], label=str(i), al
plt.legend()
plt.title('PCA of MNIST Dataset (First 2 Components)')
plt.grid(True)
plt.show()

# Visualize some original vs. reconstructed images
n_row, n_col = 2, 5
fig, axes = plt.subplots(n_row, n_col, figsize=(15, 6))

# Reconstruct images from PCA components
X_mnist_reconstructed = pca_mnist.inverse_transform(X_mnist_pca)
X_mnist_reconstructed = scaler.inverse_transform(X_mnist_reconstructed)

for i in range(n_row):
    for j in range(n_col):
        idx = i * n_col + j
        if i == 0:
            axes[i, j].imshow(X_mnist[idx].reshape(28, 28), cmap='gray')
            axes[i, j].set_title(f"Original: {y_mnist[idx]}")
        else:
            axes[i, j].imshow(X_mnist_reconstructed[idx].reshape(28, 28), cmap='gray')
            axes[i, j].set_title(f"Reconstructed: {y_mnist[idx]}")
```

```
# Apply PCA with 3 components
pca_3d = PCA(n_components=3)
components = pca_3d.fit_transform(X_scaled)

# Create a DataFrame for plotting
df = pd.DataFrame({
    'PC1': components[:, 0],
    'PC2': components[:, 1],
    'PC3': components[:, 2],
    'Species': [target_names[i] for i in y]
})

# Create 3D scatter plot
fig = px.scatter_3d(
    df, x='PC1', y='PC2', z='PC3',
    color='Species',
    title='3D PCA of Iris Dataset',
    labels={'PC1': 'Principal Component 1',
            'PC2': 'Principal Component 2',
            'PC3': 'Principal Component 3'}
)

fig.update_layout(
    legend_title_text='Species',
    scene=dict(
        xaxis_title='PC1',
        yaxis_title='PC2',
        zaxis_title='PC3'
    )
)

fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) 3D PCA visualization of the Iris dataset

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 6.9

6.8 Choosing the Right Dimensionality Reduction Technique

Technique	Strengths	Weaknesses	Best For
PCA	Fast, easy to interpret	Linear transformations only	Large datasets, initial exploration
Kernel PCA	Handles nonlinear relationships	More parameters to tune	Complex, nonlinear data
LDA	Maximizes class separation	Requires labeled data	Classification tasks
t-SNE	Excellent for visualization	Slow on large datasets	Visualizing high-dimensional data
UMAP	Preserves local and global structure	Complex implementation	Alternative to t-SNE for larger datasets

6.9 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a powerful linear algebra technique that decomposes a matrix into three component matrices, revealing the underlying structure of the data. SVD forms the mathematical foundation for many dimensionality reduction techniques, including PCA.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from scipy.linalg import svd
from sklearn.preprocessing import StandardScaler
from PIL import Image
```

Figure 6.10

6.9.1 Mathematical Foundation

SVD decomposes a matrix A (of size $m \times n$) into three matrices:

$$A = U\Sigma V^T$$

Where: - U is an $m \times m$ orthogonal matrix containing the left singular vectors - Σ is an $m \times n$ diagonal matrix containing the singular values - V^T is the transpose of an $n \times n$ orthogonal matrix containing the right singular vectors

The singular values in Σ are ordered in descending order, with the largest values representing the most important dimensions of the data.

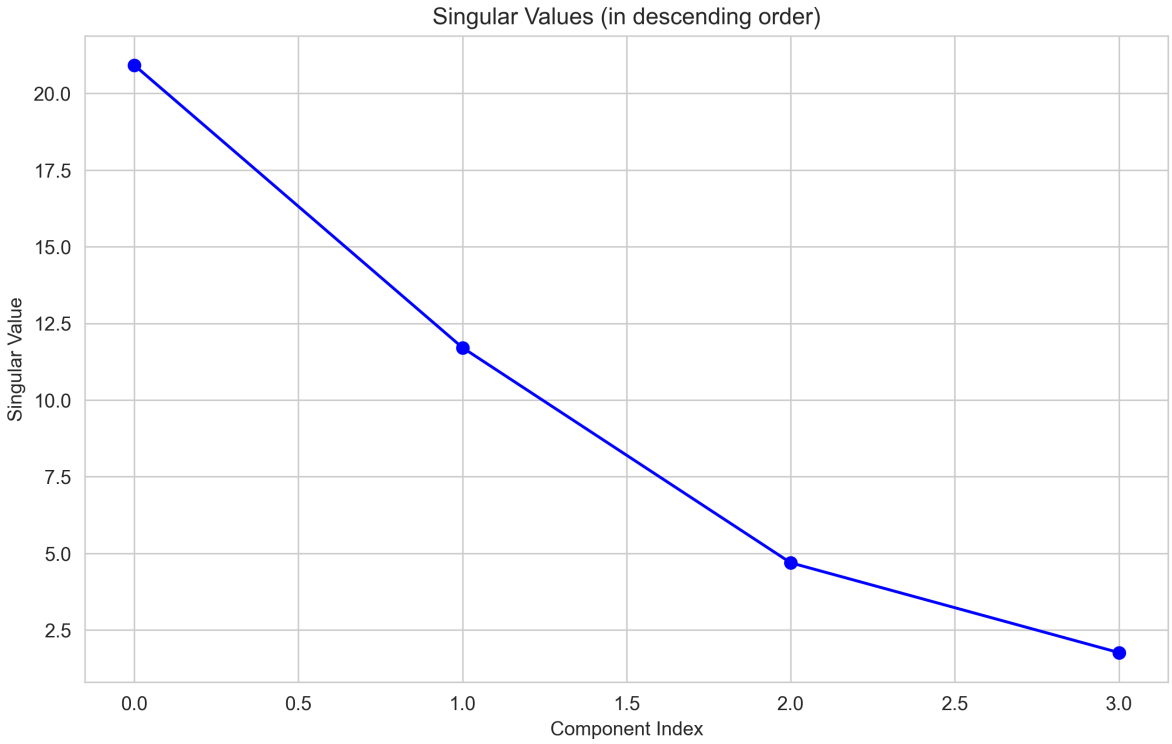
6.9.2 Basic SVD Example

Let's implement SVD on the Iris dataset to understand its mechanics:

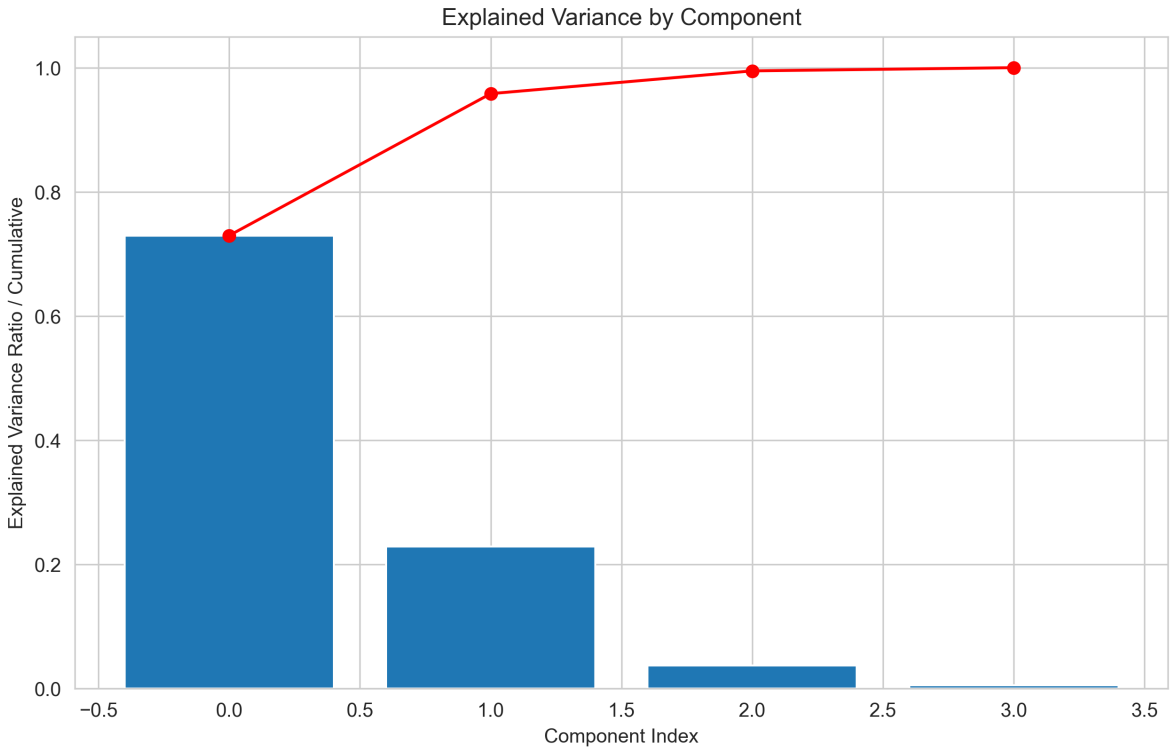
```
# Load and scale the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply SVD
U, sigma, Vt = svd(X_scaled)

# Print dimensions of decomposed matrices
print(f"Original matrix shape: {X_scaled.shape}")
print(f"U matrix shape: {U.shape}")
print(f"Sigma shape: {sigma.shape}")
print(f"V^T matrix shape: {Vt.shape}")

# Plot the singular values
plt.figure(figsize=(10, 6))
plt.plot(sigma, 'bo-')
plt.xlabel('Component Index')
plt.ylabel('Singular Value')
plt.title('Singular Values (in descending order)')
plt.grid(True)
plt.show()

# Calculate and plot the explained variance ratio
explained_variance = (sigma ** 2) / (len(X_scaled) - 1)
total_var = explained_variance.sum()
```

```
explained_variance_ratio = explained_variance / total_var

plt.figure(figsize=(10, 6))
plt.bar(range(len(explained_variance_ratio)), explained_variance_ratio)
plt.plot(range(len(explained_variance_ratio)),
         np.cumsum(explained_variance_ratio), 'r-o')
plt.xlabel('Component Index')
plt.ylabel('Explained Variance Ratio / Cumulative')
plt.title('Explained Variance by Component')
plt.grid(True)
plt.show()
```

```
Original matrix shape: (150, 4)
U matrix shape: (150, 150)
Sigma shape: (4,)
V^T matrix shape: (4, 4)
```



(a) SVD applied to the Iris dataset



(b)

Figure 6.11

6.9.3 Relationship Between SVD and PCA

PCA can be implemented using SVD, which is often more numerically stable. The principal components in PCA are equivalent to the right singular vectors in SVD.

```
# Project data onto first two singular vectors (equivalent to first two PCs)
svd_projection = X_scaled @ Vt.T[:, :2]

# Visualize the projection
plt.figure(figsize=(10, 8))
colors = ['navy', 'turquoise', 'darkorange']
target_names = iris.target_names

for i, c, label in zip(range(3), colors, target_names):
    plt.scatter(svd_projection[y == i, 0], svd_projection[y == i, 1],
                c=c, label=label)

plt.xlabel('First Component')
plt.ylabel('Second Component')
plt.title('SVD Projection of Iris Dataset')
plt.legend()
plt.grid(True)
plt.show()

# Compare first two singular values with corresponding eigenvectors
print(f"First two singular values: {sigma[:2]}")
print(f"First two singular values squared: {sigma[:2]**2}")
```

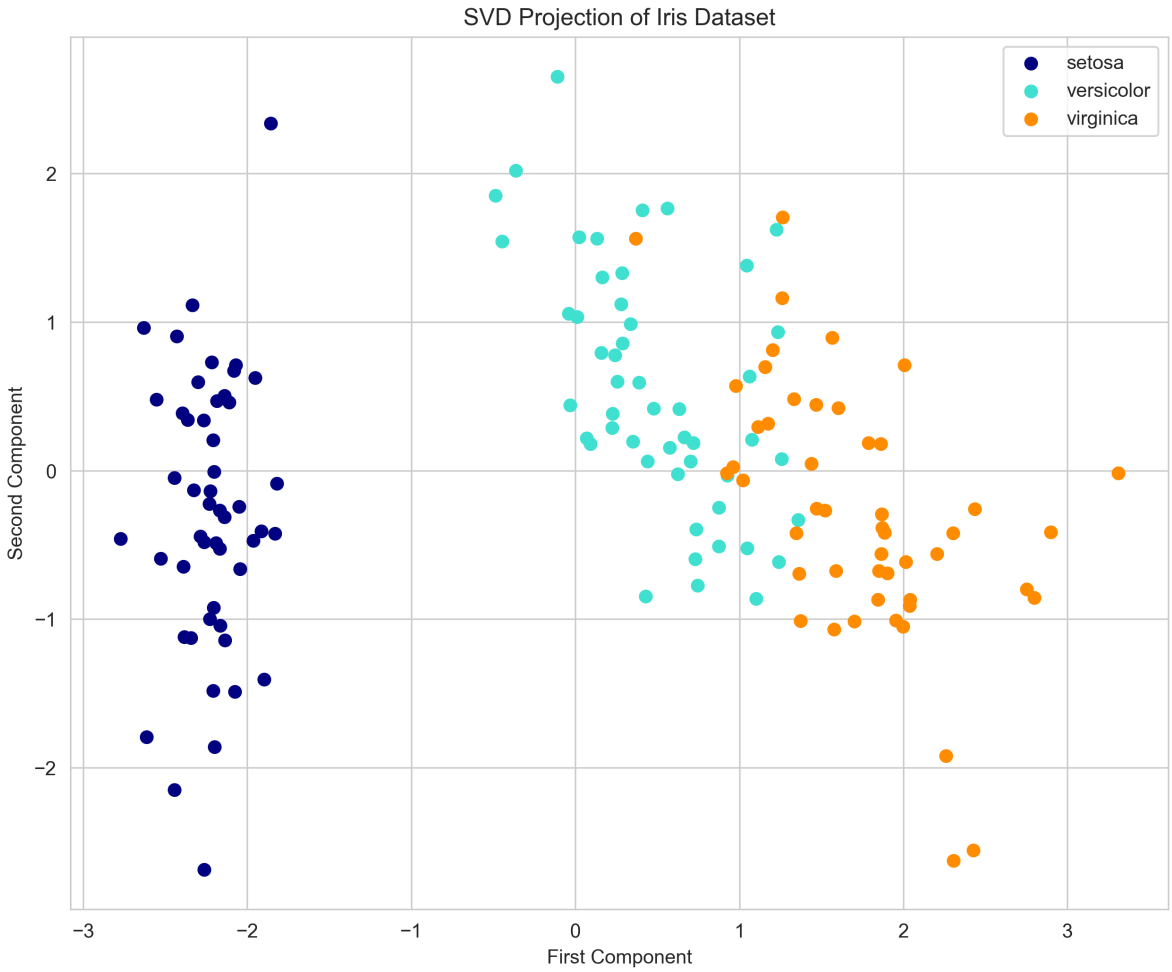


Figure 6.12: Comparison of SVD and PCA projections

```
First two singular values: [20.92306556 11.7091661 ]
First two singular values squared: [437.77467248 137.10457072]
```

6.9.4 Low-Rank Approximation

One of the key applications of SVD is low-rank matrix approximation, which enables data compression:

```
# Create a simple matrix for demonstration
A = np.array([
    [1, 2, 3, 4, 5],
```

```
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15]
])

# Apply SVD
U, sigma, Vt = svd(A)

# Create diagonal matrix Sigma
Sigma = np.zeros((A.shape[0], A.shape[1]))
for i in range(min(A.shape)):
    Sigma[i, i] = sigma[i]

# Function to reconstruct with k components
def reconstruct_svd(u, sigma, vt, k):
    # Create truncated sigma matrix
    sigma_k = np.zeros((u.shape[0], vt.shape[0]))
    for i in range(min(k, len(sigma))):
        sigma_k[i, i] = sigma[i]

    # Reconstruct
    return u @ sigma_k @ vt

# Reconstruct with different ranks
ranks = [1, 2, 3]
fig, axes = plt.subplots(1, len(ranks) + 1, figsize=(15, 4))

# Original matrix
axes[0].imshow(A, cmap='viridis')
axes[0].set_title('Original Matrix')
axes[0].axis('off')

# Reconstructions
for i, k in enumerate(ranks):
    A_k = reconstruct_svd(U, sigma, Vt, k)
    axes[i+1].imshow(A_k, cmap='viridis')
    axes[i+1].set_title(f'Rank {k} Approximation')
    axes[i+1].axis('off')

plt.tight_layout()
plt.show()

# Calculate and display approximation errors
```

```
for k in ranks:
    A_k = reconstruct_svd(U, sigma, Vt, k)
    error = np.linalg.norm(A - A_k, 'fro')
    print(f"Rank {k} approximation error: {error:.4f}")
```

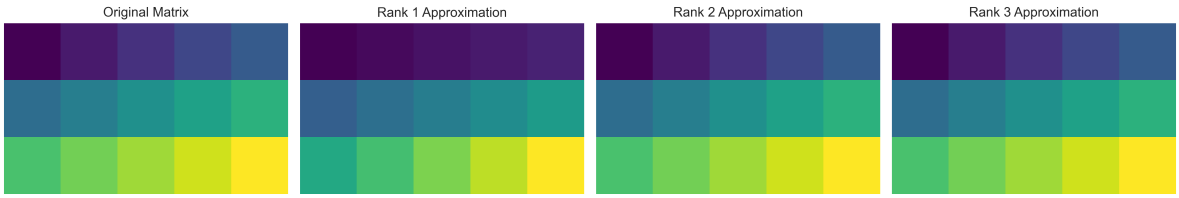


Figure 6.13: Low-rank approximation of a matrix

```
Rank 1 approximation error: 2.4654
Rank 2 approximation error: 0.0000
Rank 3 approximation error: 0.0000
```

6.9.5 SVD for Image Compression

A common application of SVD is image compression. Let's demonstrate this with a grayscale image:

6.9.6 Applications of SVD

SVD has numerous applications across various domains:

6.9.6.1 1. Recommendation Systems

6.9.6.2 2. Latent Semantic Analysis (LSA) for Text Mining

6.9.7 Truncated SVD vs. PCA

Truncated SVD can be applied directly to sparse matrices, while PCA typically requires dense matrices. This makes Truncated SVD particularly useful for text analysis and high-dimensional sparse data:

```
# Load a sample image
# For demonstration, let's create a simple gradient image
img_size = 512
img = np.zeros((img_size, img_size))
for i in range(img_size):
    for j in range(img_size):
        img[i, j] = (i + j) / (2 * img_size)

# Apply SVD
U, sigma, Vt = svd(img, full_matrices=False)

# Compress image with different numbers of singular values
k_values = [5, 20, 50, 100]
fig, axes = plt.subplots(1, len(k_values) + 1, figsize=(18, 4))

# Original image
axes[0].imshow(img, cmap='gray')
axes[0].set_title('Original Image')
axes[0].axis('off')

# Compressed images
for i, k in enumerate(k_values):
    # Reconstruct image with k singular values
    compressed_img = U[:, :k] @ np.diag(sigma[:k]) @ Vt[:k, :]

    # Display
    axes[i+1].imshow(compressed_img, cmap='gray')
    axes[i+1].set_title(f'k={k}, CR={img.size/(k*(img.shape[0] + img.shape[1] + 1)):.1f}')
    axes[i+1].axis('off')

    # Print compression ratio
    original_size = img.size * 8  # Assuming 8 bits per pixel
    compressed_size = k * (img.shape[0] + img.shape[1] + 1) * 8  # k(m+n+1) values stored
    compression_ratio = original_size / compressed_size
    print(f"k={k}, Compression ratio: {compression_ratio:.2f}")

plt.tight_layout()
plt.show()
```

Figure 6.14

```
# Create a user-item ratings matrix (movies example)
# Rows: users, Columns: movies, Values: ratings
ratings = np.array([
    [5, 4, 0, 0, 1],
    [4, 0, 0, 3, 1],
    [1, 1, 0, 5, 0],
    [0, 0, 4, 0, 3],
    [2, 0, 5, 0, 0]
])

# Apply SVD
U, sigma, Vt = svd(ratings)

# Use a low-rank approximation (k=2)
k = 2
ratings_approx = U[:, :k] @ np.diag(sigma[:k]) @ Vt[:k, :]

# Fill in missing ratings
print("Original ratings matrix:")
print(ratings)
print("\nReconstructed ratings matrix:")
print(np.round(ratings_approx, 1))

# Find recommendations for a user
user_id = 0
missing_ratings = np.where(ratings[user_id] == 0)[0]
recommendations = [(item, ratings_approx[user_id, item]) for item in missing_ratings]
recommendations.sort(key=lambda x: x[1], reverse=True)

print(f"\nTop recommendations for User {user_id}:")
for item, score in recommendations:
    print(f"Item {item}: Predicted rating {score:.1f}")
```

Figure 6.15

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

# Sample documents
documents = [
    "Machine learning is a field of artificial intelligence",
    "Deep learning is a subset of machine learning",
    "Neural networks are used in deep learning",
    "SVD is used for dimensionality reduction",
    "PCA and SVD are related techniques",
    "Dimensionality reduction helps with visualizing data"
]

# Create TF-IDF matrix
vectorizer = TfidfVectorizer(stop_words='english')
X = vectorizer.fit_transform(documents)

# Get feature names
feature_names = vectorizer.get_feature_names_out()

# Print the document-term matrix
print("Document-Term Matrix (TF-IDF):")
df = pd.DataFrame(X.toarray(), columns=feature_names)
print(df)

# Apply LSA (truncated SVD)
n_components = 2
lsa = TruncatedSVD(n_components=n_components)
X_lsa = lsa.fit_transform(X)

# Print explained variance
print(f"\nExplained variance ratio: {lsa.explained_variance_ratio_}")
print(f"Total explained variance: {sum(lsa.explained_variance_ratio_):.2f}")

# Plot documents in the reduced space
plt.figure(figsize=(10, 8))
plt.scatter(X_lsa[:, 0], X_lsa[:, 1], alpha=0.8)

# Label each point
for i, doc in enumerate(documents):
    plt.annotate(f"Doc {i+1}", (X_lsa[i, 0], X_lsa[i, 1]),
                 xytext=(5, 5), textcoords='offset points')

plt.xlabel("Component 1")
plt.ylabel("Component 2")
plt.title("Documents in LSA Space")
plt.grid(True)
plt.show()

# Examine term weights in components
component_terms = {}
for i, component in enumerate(lsa.components_):
    # Get the top terms for this component
```

```
from sklearn.decomposition import TruncatedSVD, PCA
from scipy.sparse import csr_matrix

# Create a sparse matrix
rows = np.random.randint(0, 100, 1000)
cols = np.random.randint(0, 50, 1000)
data = np.random.randn(1000)
sparse_matrix = csr_matrix((data, (rows, cols)), shape=(100, 50))

# Apply Truncated SVD
tsvd = TruncatedSVD(n_components=2)
tsvd_result = tsvd.fit_transform(sparse_matrix)

# Convert to dense for PCA
dense_matrix = sparse_matrix.toarray()

# Apply PCA
pca = PCA(n_components=2)
pca_result = pca.fit_transform(dense_matrix)

# Compare results
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Plot Truncated SVD result
axes[0].scatter(tsvd_result[:, 0], tsvd_result[:, 1], alpha=0.5)
axes[0].set_title('Truncated SVD')
axes[0].set_xlabel('Component 1')
axes[0].set_ylabel('Component 2')
axes[0].grid(True)

# Plot PCA result
axes[1].scatter(pca_result[:, 0], pca_result[:, 1], alpha=0.5)
axes[1].set_title('PCA')
axes[1].set_xlabel('Component 1')
axes[1].set_ylabel('Component 2')
axes[1].grid(True)

plt.tight_layout()
plt.show()

# Compare explained variance
print(f"Truncated SVD explained variance ratio: {tsvd.explained_variance_ratio_}")
```

```
print(f"PCA explained variance ratio: {pca.explained_variance_ratio_}")
```

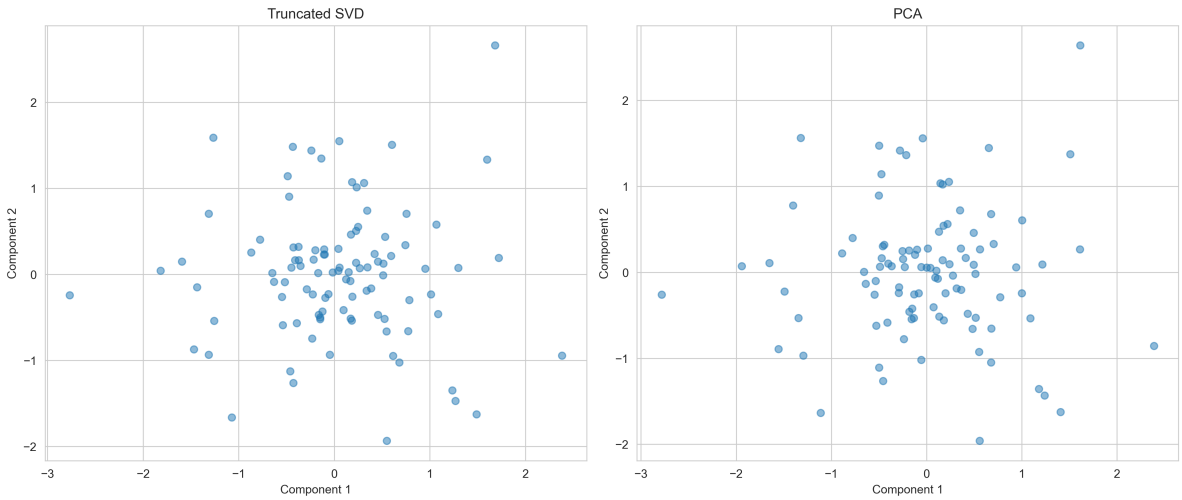


Figure 6.17: Comparison of Truncated SVD and PCA

```
Truncated SVD explained variance ratio: [0.06270912 0.05752154]
PCA explained variance ratio: [0.06275364 0.05756782]
```

6.10 No# Advantages and Limitations of SVD

6.10.0.1 Advantages

1. **Robust mathematical foundation**: Based on well-established linear algebra principles
2. **Numerical stability**: Often more stable than eigendecomposition-based methods
3. **Applicability to non-square matrices**: Can be applied to any rectangular matrix
4. **Optimal low-rank approximation**: Provides the best approximation in terms of Frobenius norm

6.10.0.2 Limitations

1. **Computational cost**: Full SVD is expensive for large matrices ($O(\min(mn^2, m^2n))$)
2. **Memory requirements**: Working with large matrices can be memory-intensive
3. **Interpretability**: The resulting components may be difficult to interpret in some domains
4. **Linearity**: As with PCA, SVD assumes linear relationships in the data

6.11 Conclusion

Dimensionality reduction techniques are essential tools in the data scientist's toolkit, enabling:

- More efficient model training
- Better visualization of complex datasets
- Improved performance through noise reduction
- Insights into feature importance and relationships

As with all techniques, the choice of dimensionality reduction method should be guided by:

1. The specific characteristics of your dataset
2. Your analysis goals
3. Computational constraints
4. Whether you need interpretable results

Singular Value Decomposition is a fundamental technique in linear algebra with powerful applications in dimensionality reduction, data compression, noise filtering, and recommendation systems. Its ability to decompose any matrix into meaningful components makes it an essential tool for data scientists and machine learning practitioners.

By understanding the mathematical principles behind SVD and its relationship to other dimensionality reduction techniques like PCA, we can effectively apply it to solve complex problems across various domains.

In practice, the choice between full SVD, truncated SVD, or randomized algorithms depends on the specific characteristics of the data and computational constraints. Modern implementations in libraries like SciPy and scikit-learn provide efficient algorithms that make SVD accessible for large-scale applications.

The Python implementations demonstrated in this document provide a starting point for applying these techniques to your own data analysis and machine learning projects.

6.12 References

1. Jolliffe, I. T., & Cadima, J. (2016). Principal component analysis: a review and recent developments. Philosophical Transactions of the Royal Society A, 374(2065).
2. Van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. Journal of Machine Learning Research, 9(11).
3. Pedregosa, F., et al. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825-2830.

7 Tree-Based Models and Ensemble Learning

7.1 Decision Trees

Decision trees are versatile machine learning models that can handle both classification and regression tasks. They're powerful tools for inductive inference and particularly useful for approximating discrete-valued target functions.

7.1.1 Key Features

- **Robust to Noisy Data**: Handle imperfections in data, including noise and missing values
- **Complex Datasets**: Capable of fitting complex datasets and representing disjunctive expressions
- **Interpretable**: Trees model decisions through a series of "if/else" questions, providing clear decision-making processes

7.1.2 Core Concepts

Decision trees operate by recursively partitioning the feature space based on the values of input features. At each node:

1. **Feature Selection**: Choose the most informative feature to split on
2. **Splitting Criterion**: Determine the optimal threshold or condition for the split
3. **Recursive Partitioning**: Continue splitting until stopping criteria are met

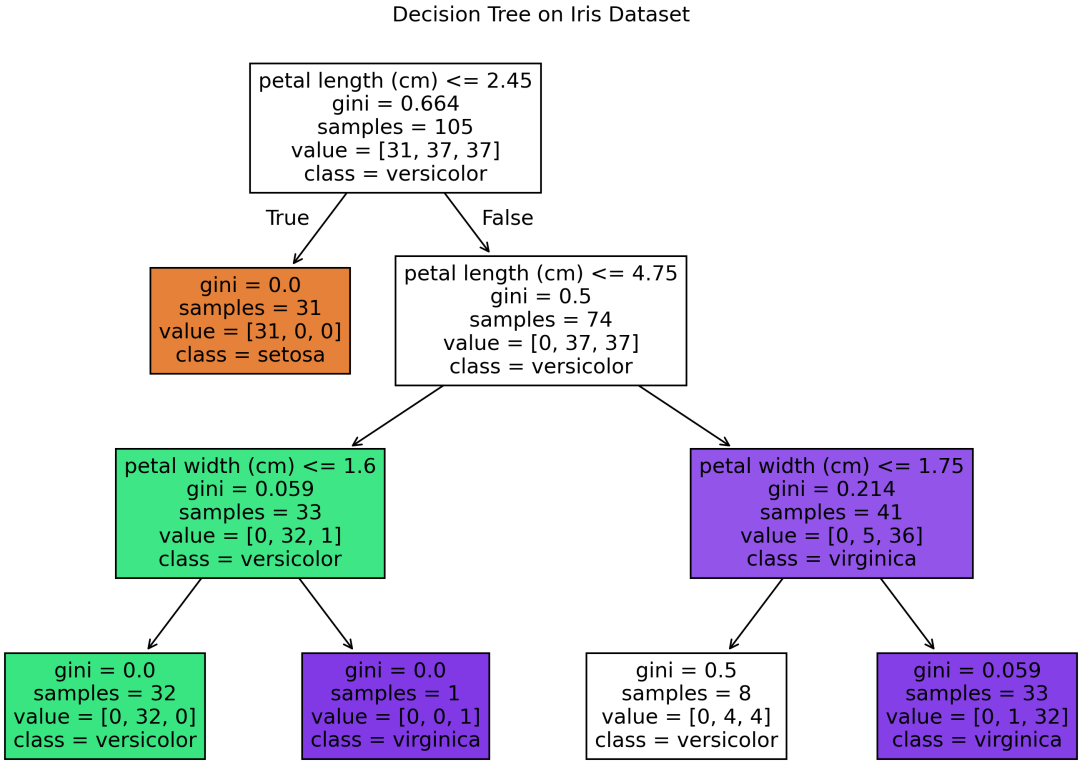
The algorithm aims to create homogeneous subsets with respect to the target variable, maximizing information gain at each step.

7.1.3 Decision Boundaries

Decision trees create piecewise constant decision boundaries that are parallel to the feature axes. This characteristic leads to:

- **Rectangular Partitioning**: Each leaf represents a rectangular region in the feature space
- **Orthogonal Boundaries**: Decision boundaries are always perpendicular to feature axes
- **Staircase Effect**: Complex functions are approximated using axis-parallel rectangles

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split

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

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create and train decision tree
clf = DecisionTreeClassifier(max_depth=3, random_state=42)
clf.fit(X_train, y_train)

# Visualize decision tree
plt.figure(figsize=(12, 8))
plot_tree(clf, filled=True, feature_names=iris.feature_names, class_names=iris.target_names)
plt.title("Decision Tree on Iris Dataset")
plt.show()

# Evaluate performance
train_score = clf.score(X_train, y_train)
test_score = clf.score(X_test, y_test)
print(f"Training accuracy: {train_score:.3f}")
print(f"Testing accuracy: {test_score:.3f}")
```




```
Training accuracy: 0.952
Testing accuracy: 1.000
```

7.1.4 Representation

Decision trees represent a disjunction (OR) of conjunctions (AND) of constraints on attribute values:

- Each path from root to leaf represents a conjunction (AND) of tests on instance attributes
- The entire tree is a disjunction (OR) of these conjunctions
- Each leaf node represents a classification outcome

7.1.5 When to Use Decision Trees

- **Attribute-Value Pair Representation**: Instances are represented as attribute-value pairs
- **Discrete Output**: Target function has discrete output values (classification tasks)

- **Disjunctive Descriptions**: Need to represent logical ORs
- **Noisy Data**: Training data contains errors or missing values

7.1.6 Classification Process

Trees classify instances by sorting them from root to leaf: - Each node tests an attribute - Each branch corresponds to a possible attribute value - Each leaf assigns a class label

7.1.7 Algorithmic Approaches

Several algorithms exist for constructing decision trees:

1. **ID3 (Iterative Dichotomiser 3)**: Uses entropy and information gain
2. **C4.5**: Extends ID3 by handling continuous attributes and missing values
3. **CART (Classification and Regression Trees)**: Uses Gini impurity for classification
4. **CHAID (Chi-square Automatic Interaction Detector)**: Uses chi-square tests for categorical outputs

7.1.8 The CART Algorithm

CART (Classification and Regression Trees) is one of the most popular decision tree algorithms:

1. Start with all data at the root node
2. For each feature, find the best split that minimizes impurity
3. Split the data based on the best feature and threshold
4. Recursively apply steps 2-3 to the child nodes
5. Stop when a stopping criterion is met (e.g., max depth, min samples)

7.1.9 Training a Decision Tree

The training process involves finding the best set of questions (splits) to divide the data:

```
class Leaf:
    def __init__(self, value):
        self.value = value

class Node:
    def __init__(self, attribute):
        self.attribute = attribute
```

```
        self.branches = {}

    def add_branch(self, value, subtree):
        self.branches[value] = subtree

def basic_decision_tree_algorithm(examples, target_attribute, attributes):
    """
    Basic implementation of a decision tree algorithm
    """
    # If all examples have the same value for the target attribute, return a leaf node
    if len(set(ex[target_attribute] for ex in examples)) == 1:
        return Leaf(examples[0][target_attribute])

    # If no attributes are left, return a leaf node with the majority value
    if not attributes:
        majority_value = max(set(ex[target_attribute] for ex in examples), key=lambda val: su
        return Leaf(majority_value)

    # Choose the best attribute to split on (placeholder for actual selection logic)
    best_attribute = attributes[0]  # Replace with actual logic to select the best attribute

    # Create a new decision tree node
    tree = Node(best_attribute)

    # Get unique values for the best attribute
    unique_values = set(ex[best_attribute] for ex in examples)

    for value in unique_values:
        # Create a subset of examples where the best attribute equals the current value
        subset = [ex for ex in examples if ex[best_attribute] == value]

        if not subset:
            # If the subset is empty, add a leaf with the majority value
            majority_value = max(set(ex[target_attribute] for ex in examples), key=lambda val
            tree.add_branch(value, Leaf(majority_value))
        else:
            # Recursively build the subtree
            subtree = basic_decision_tree_algorithm(subset, target_attribute, [attr for attr
            tree.add_branch(value, subtree)

    return tree
```

7.1.9.1 Impurity Measures

To decide the best feature to split on, decision trees use impurity measures:

- **Gini Index**: Measures the likelihood of misclassifying a randomly selected instance
- **Entropy**: Measures disorder or uncertainty in the dataset
- **Misclassification Error**: Proportion of misclassified instances

7.1.9.1.1 Entropy

Entropy quantifies the uncertainty or randomness in a set of examples:

$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$

Where: - S is the dataset - c is the number of classes - p_i is the proportion of examples in class i

7.1.9.1.2 Information Gain

Information gain measures the reduction in entropy after splitting on an attribute:

$IG(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v)$

Where: - A is the attribute - S_v is the subset of S for which attribute A has value v

7.1.9.1.3 Gini Impurity

Gini impurity measures the probability of incorrectly classifying a randomly chosen element if it were randomly labeled according to the class distribution in the subset:

$Gini(S) = 1 - \sum_{i=1}^{c} (p_i)^2$

```
def calculate_entropy(y):
    """Calculate entropy of a label set"""
    classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / counts.sum()
    entropy = -np.sum(probabilities * np.log2(probabilities))
    return entropy

def calculate_gini(y):
    """Calculate Gini impurity of a label set"""
    classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / counts.sum()
    gini = 1 - np.sum(probabilities**2)
    return gini
```

```
# Example calculation
sample_labels = np.array([0, 0, 1, 1, 0, 1, 0, 1])
print(f"Entropy: {calculate_entropy(sample_labels):.4f}")
print(f"Gini: {calculate_gini(sample_labels):.4f}")
```

```
Entropy: 1.0000
Gini: 0.5000
```

7.1.10 Pruning Techniques

Decision trees are prone to overfitting, especially when they grow too deep. Pruning helps mitigate this:

7.1.10.1 Pre-pruning (Early Stopping)

Stops the tree from growing before it perfectly fits the training data: - Maximum depth limit - Minimum samples per leaf - Minimum impurity decrease

7.1.10.2 Post-pruning

Builds the full tree, then removes branches that don't improve generalization: - Cost-complexity pruning (Minimal Cost-Complexity Pruning) - Reduced Error Pruning - Pessimistic Error Pruning

7.1.11 Handling Categorical and Continuous Features

Decision trees can handle both categorical and continuous features:

- **Categorical Features**: Create branches for each category or group similar categories
- **Continuous Features**: Find the optimal threshold that maximizes information gain

7.1.12 Advantages and Limitations

7.1.12.1 Advantages

- Intuitive and easy to explain
- Require little data preprocessing
- Handle numerical and categorical data
- Non-parametric (no assumptions about data distribution)
- Handle missing values and outliers effectively

7.1.12.2 Limitations

- Prone to overfitting
- Biased toward features with more levels
- Unstable (small changes in data can result in very different trees)
- Struggle with diagonal decision boundaries
- May create biased trees if classes are imbalanced

7.2 Ensemble Methods

Ensemble methods combine multiple predictors to improve accuracy by reducing variance and bias.

7.2.1 Core Principles

Ensemble methods work based on the "wisdom of crowds" principle:

1. **Diversity**: Individual models make different errors
2. **Independence**: Errors are uncorrelated
3. **Aggregation**: Combining predictions reduces overall error

7.2.2 The Bias-Variance Tradeoff

Ensemble methods address the fundamental bias-variance tradeoff:

- **Bias**: Error from incorrect assumptions in the learning algorithm
- **Variance**: Error from sensitivity to small fluctuations in the training set
- **Total Error** = Bias^2 + Variance + Irreducible Error

Different ensemble techniques target different components of this error: - Bagging primarily reduces variance - Boosting reduces both bias and variance

```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import accuracy_score
import pandas as pd

# Create ensemble models
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
gb_model = GradientBoostingClassifier(n_estimators=100, random_state=42)

# Train models
rf_model.fit(X_train, y_train)
gb_model.fit(X_train, y_train)

# Make predictions
rf_preds = rf_model.predict(X_test)
gb_preds = gb_model.predict(X_test)

# Compare accuracies
results = pd.DataFrame({
    'Model': ['Decision Tree', 'Random Forest', 'Gradient Boosting'],
    'Test Accuracy': [
        clf.score(X_test, y_test),
        accuracy_score(y_test, rf_preds),
        accuracy_score(y_test, gb_preds)
    ]
})

print(results)
```

```
               Model  Test Accuracy
0      Decision Tree            1.0
1      Random Forest            1.0
2  Gradient Boosting            1.0
```

7.2.3 Types of Ensemble Learning

1. **Bagging (Bootstrap Aggregating)**:
    - Builds independent predictors and combines them
    - Models trained on bootstrapped datasets (random samples with replacement)

- Reduces variance, effective against overfitting
- Example: Random Forest

2. **Boosting**:
    - Builds predictors sequentially, each correcting errors of previous models
    - Assigns higher weights to misclassified data points
    - Reduces both bias and variance
    - Examples: AdaBoost, Gradient Boosting, XGBoost
3. **Stacking**:
    - Combines multiple models using another model (meta-learner)
    - Base models make predictions independently
    - Meta-learner learns how to combine these predictions optimally
    - Examples: Stacked Generalization, Blending
4. **Voting**:
    - Simple aggregation of predictions from multiple models
    - Hard Voting: Majority vote for classification
    - Soft Voting: Weighted average of probabilities
    - Works best with diverse, uncorrelated models

7.2.4 Theoretical Foundations

The power of ensemble methods is backed by mathematical proofs:

- **Condorcet's Jury Theorem**: As the number of independent, better-than-random models increases, the probability of a correct majority vote approaches 1
- **Bias-Variance Decomposition**: Ensembles can reduce variance without increasing bias
- **No Free Lunch Theorem**: No single model is optimal for all problems, but ensembles can adapt to different problem structures

7.3 Random Forests

Random Forest is an ensemble method combining multiple decision trees through bagging.

7.3.1 Core Concepts

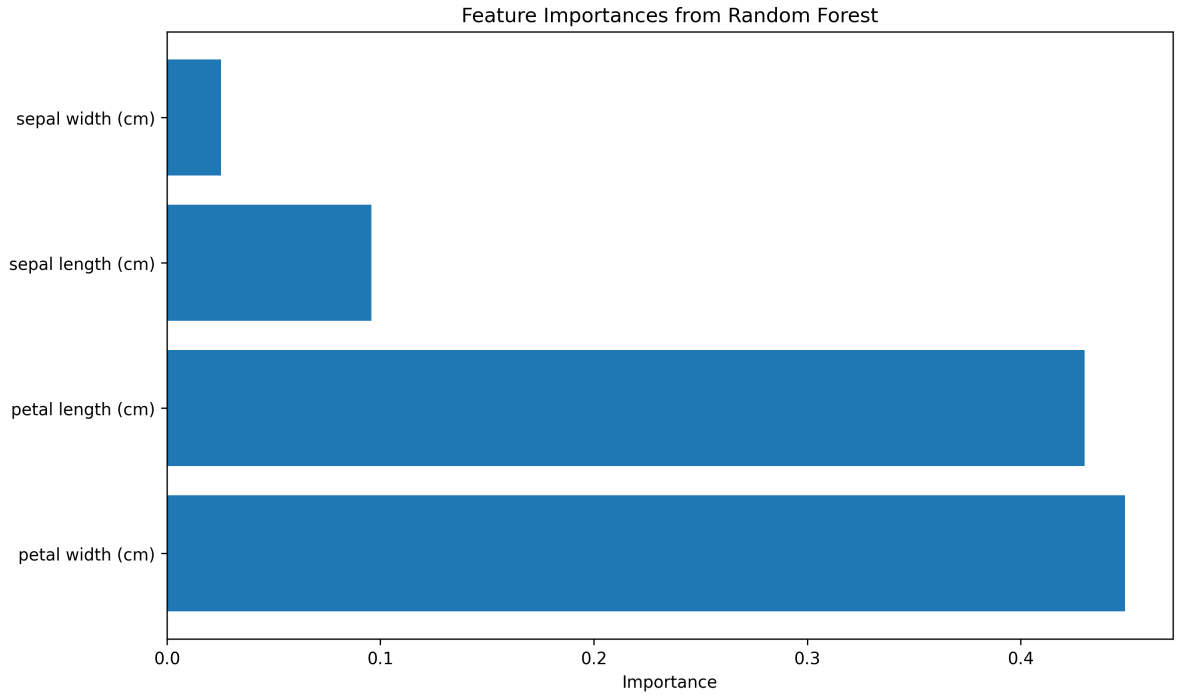
Random Forests extend the bagging idea with additional randomness:

1. **Bootstrap Sampling**: Each tree is trained on a random subset of data
2. **Feature Randomization**: At each node, consider only a random subset of features
3. **Ensemble Aggregation**: Combine predictions through voting (classification) or averaging (regression)

7.3.2 Random Forest Algorithm

1. Create n_estimators bootstrap samples from the original dataset
2. For each sample, grow a decision tree with the following modification:
    - At each node, randomly select m features (typically m sqrt(total features))
    - Split on the best feature among the m features
3. Predict new data by aggregating predictions from all trees

```
# Train a Random Forest with different parameters
rf = RandomForestClassifier(
    n_estimators=100,
    max_depth=4,
    min_samples_split=10,
    random_state=42
)
rf.fit(X_train, y_train)

# Get feature importances
importances = pd.DataFrame({
    'Feature': iris.feature_names,
    'Importance': rf.feature_importances_
}).sort_values('Importance', ascending=False)

# Plot feature importances
plt.figure(figsize=(10, 6))
plt.barh(importances['Feature'], importances['Importance'])
plt.xlabel('Importance')
plt.title('Feature Importances from Random Forest')
plt.tight_layout()
plt.show()

print(importances)
```



	Feature	Importance
3	petal width (cm)	0.448932
2	petal length (cm)	0.429927
0	sepal length (cm)	0.095760
1	sepal width (cm)	0.025381

7.3.3 How Random Forest Works

- Each tree is trained independently on a random subset of the data
- Additional randomness is introduced by selecting a random subset of features at each node
- Final output is determined by aggregating results from all trees (majority voting for classification, averaging for regression)

7.3.4 Key Features of Random Forests

- Controls the growth of individual trees
- Controls the ensemble as a whole through bagging parameters
- Randomness at each decision tree's growth stage increases tree diversity

7.3.5 Mathematical Intuition

The error rate of a random forest depends on:

1. **Correlation between trees**: Lower correlation improves performance
2. **Strength of individual trees**: Stronger trees improve performance

The feature randomization helps reduce correlation between trees while maintaining their strength.

7.3.6 Out-of-Bag (OOB) Error Estimation

A unique advantage of random forests is built-in validation:

- Each bootstrap sample leaves out approximately 1/3 of the data (out-of-bag samples)
- These OOB samples can be used to estimate model performance without a separate validation set
- OOB error is an unbiased estimate of the generalization error

7.3.7 Feature Importance

Random forests provide a natural way to measure feature importance:

1. **Mean Decrease Impurity (MDI)**: Average reduction in impurity across all trees
2. **Mean Decrease Accuracy (MDA)**: Decrease in model accuracy when a feature is permuted
3. **Permutation Importance**: Randomize one feature at a time and measure the drop in performance

7.3.8 Proximity Analysis

Random forests can measure the similarity between data points:

- Two points are "close" if they often end up in the same leaf nodes
- This proximity measure can be used for clustering, outlier detection, and missing value imputation

7.3.9 Hyperparameters

Key parameters affecting random forest performance:

- **n_estimators**: Number of trees (more is usually better)
- **max_features**: Number of features to consider at each split
- **max_depth**: Maximum depth of each tree
- **min_samples_split**: Minimum samples required to split a node
- **min_samples_leaf**: Minimum samples required in a leaf node
- **bootstrap**: Whether to use bootstrap sampling

7.3.10 Advantages and Limitations

7.3.10.1 Advantages

- Reduced overfitting compared to decision trees
- Robust to outliers and noise
- Handles high-dimensional data well
- Provides feature importance measures
- Built-in cross-validation through OOB samples

7.3.10.2 Limitations

- Less interpretable than single decision trees
- Computationally more intensive
- Biased for categorical features with different numbers of levels
- May overfit on noisy datasets with many features

7.4 Gradient Boosting

Gradient Boosting combines Gradient Descent with boosting principles.

7.4.1 Core Concepts

Gradient Boosting frames the ensemble learning process as an optimization problem:

1. **Loss Function**: Define a differentiable loss function to minimize
2. **Weak Learners**: Use simple models (typically shallow decision trees)
3. **Additive Training**: Build models sequentially to minimize the loss function
4. **Gradient Descent**: Each new model fits the negative gradient of the loss function

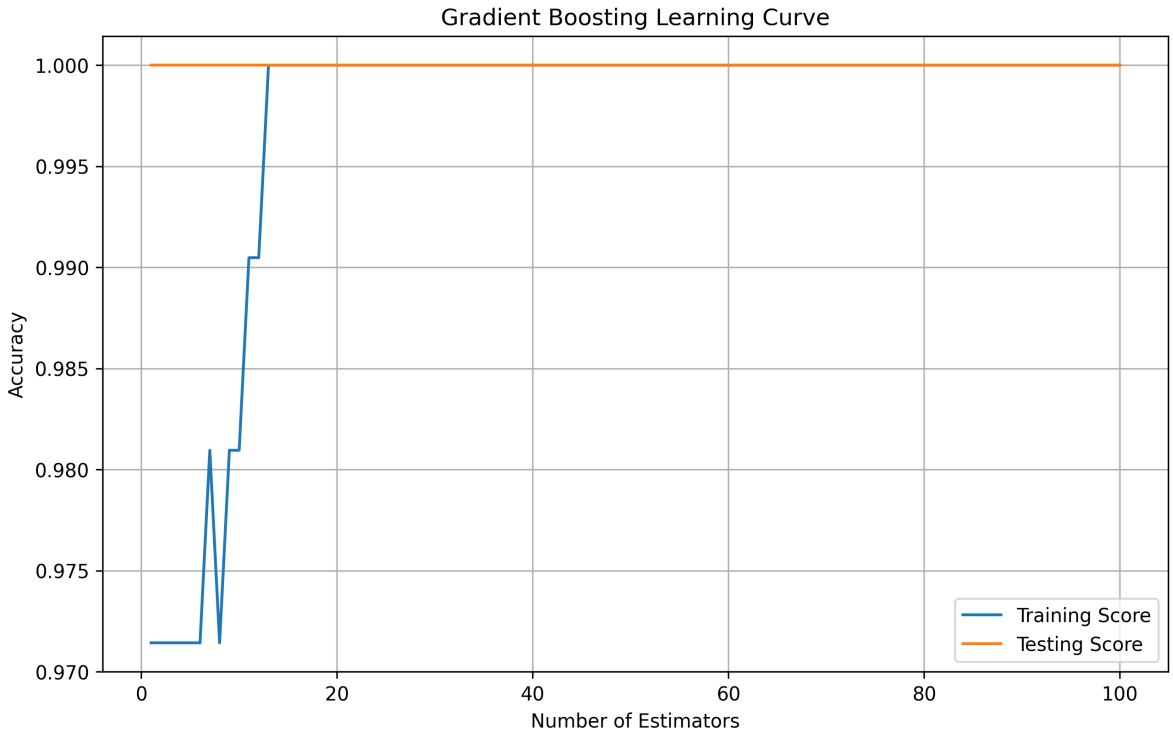
7.4.2 Gradient Boosting Algorithm

1. Initialize model with a constant value
2. For m = 1 to M (number of boosting rounds):
    - Compute negative gradient (residual) of the loss function
    - Fit a base learner (decision tree) to the negative gradient
    - Calculate optimal leaf values
    - Update the model by adding the new tree (scaled by learning rate)
3. Return the final model (sum of all trees)

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import confusion_matrix, classification_report

# Create and train Gradient Boosting model
gb = GradientBoostingClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    random_state=42
)
gb.fit(X_train, y_train)

# Predict and evaluate
y_pred = gb.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred, target_names=iris.target_names))

# Learning curve visualization
train_scores = []
test_scores = []
estimator_range = range(1, 101)

# Train models with different numbers of estimators
for n_estimators in estimator_range:
    gb = GradientBoostingClassifier(
        n_estimators=n_estimators,
        learning_rate=0.1,
        max_depth=3,
        random_state=42
    )
    gb.fit(X_train, y_train)
```

```
    train_scores.append(gb.score(X_train, y_train))
    test_scores.append(gb.score(X_test, y_test))

# Plot learning curve
plt.figure(figsize=(10, 6))
plt.plot(estimator_range, train_scores, label='Training Score')
plt.plot(estimator_range, test_scores, label='Testing Score')
plt.xlabel('Number of Estimators')
plt.ylabel('Accuracy')
plt.title('Gradient Boosting Learning Curve')
plt.legend()
plt.grid(True)
plt.show()
```

```
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```



7.4.3 How Gradient Boosting Works

1. Initial Model: Start with one model and its predictions
2. Training on Residuals: Train new model on residuals of first model
3. Iterative Correction: Each new model predicts residuals from ensemble of previous models
4. Final Prediction: Sum of predictions from all trees

7.4.4 Mathematical Formulation

Gradient Boosting minimizes a loss function $L(y, F(x))$ by iteratively adding weak learners:

$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x)$

Where: - $F_m(x)$ is the model after m iterations - $h_m(x)$ is the weak learner (decision tree) - α_m is the step size (learning rate)

The weak learner h_m is trained to approximate the negative gradient of the loss function:

$h_m(x) \approx -\left[\frac{\partial L(y,F(x))}{\partial F(x)}\right]_{F(x)=F_{m-1}(x)}$

7.4.5 Loss Functions

Different loss functions can be used depending on the task:

- **Regression**:
  - L2 loss (mean squared error)
  - L1 loss (mean absolute error)
  - Huber loss (robust to outliers)
- **Classification**:
  - Binomial deviance (logistic loss)
  - Multinomial deviance
  - Exponential loss (AdaBoost)

7.4.6 Types of Gradient Boosting

- **AdaBoost**: Each new model focuses on mistakes of previous models by weighting misclassified instances
- **XGBoost**: Highly efficient implementation with additional optimizations like regularization
- **LightGBM**: Uses gradient-based one-side sampling and exclusive feature bundling for faster training
- **CatBoost**: Handles categorical features automatically and uses ordered boosting

7.4.7 Regularization Techniques

Gradient Boosting can overfit easily. Common regularization techniques include:

1. **Shrinkage (Learning Rate)**: Scale contribution of each tree by a factor < 1
2. **Subsampling**: Train each tree on a random subset of data
3. **Early Stopping**: Stop training when validation error stops improving
4. **Tree Constraints**: Limit tree depth, minimum samples per leaf, etc.
5. **L1/L2 Regularization**: Penalize large leaf weights

7.4.8 Key Hyperparameters

- **n_estimators**: Number of boosting stages (trees)
- **learning_rate**: Controls how much each tree influences predictions
- **max_depth**: Limits nodes in each regression estimator (tree)
- **subsample**: Fraction of samples to use for fitting each tree
- **loss**: Loss function to be optimized

7.4.9 Advantages and Limitations

7.4.9.1 Advantages

- Often provides best predictive accuracy
- Flexible - works with various loss functions
- Handles mixed data types well
- Robust to outliers with robust loss functions
- Automatically handles feature interactions

7.4.9.2 Limitations

- Prone to overfitting without careful tuning
- Sensitive to noisy data and outliers with some loss functions
- Computationally intensive
- Less interpretable than single decision trees
- Sequential nature limits parallelization

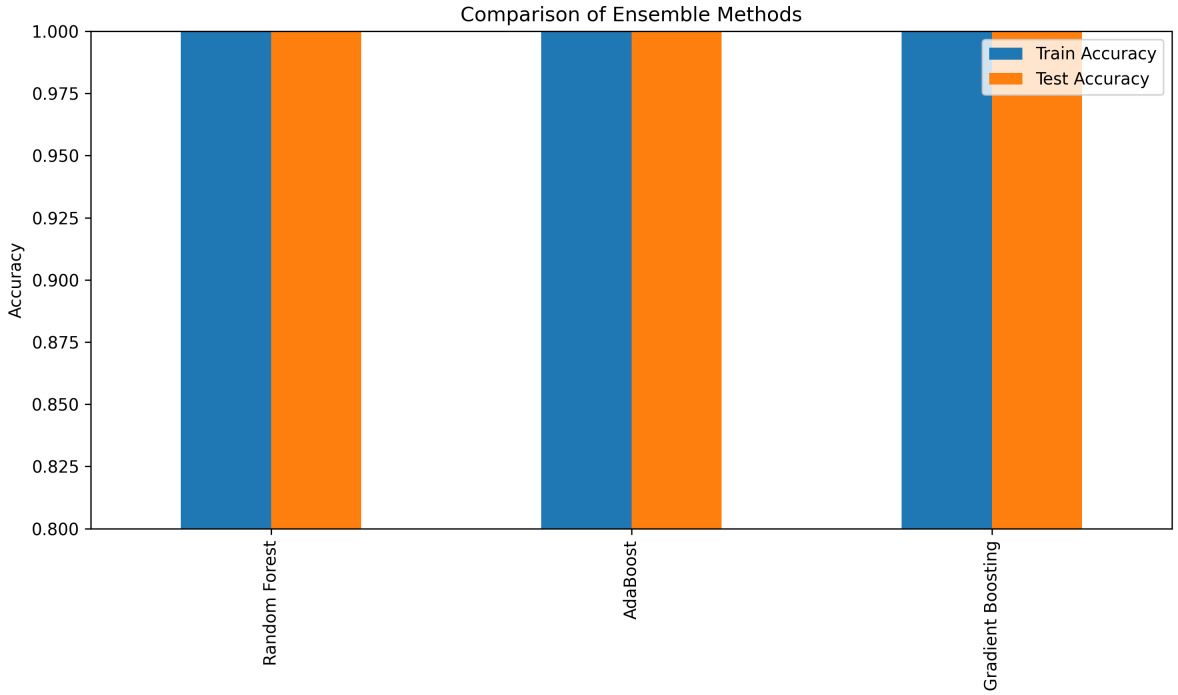
7.5 Comparing Ensemble Methods

```
# Compare different ensemble methods
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier, GradientBoostingCla
from sklearn.metrics import accuracy_score

# Initialize models
models = {
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
    'AdaBoost': AdaBoostClassifier(n_estimators=100, random_state=42),
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100, random_state=42)
}

# Train and evaluate each model
results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    train_acc = accuracy_score(y_train, model.predict(X_train))
    test_acc = accuracy_score(y_test, model.predict(X_test))
    results[name] = {'Train Accuracy': train_acc, 'Test Accuracy': test_acc}
```

```python
# Display results
results_df = pd.DataFrame(results).T
print(results_df)

# Plot results
plt.figure(figsize=(10, 6))
results_df.plot(kind='bar', figsize=(10, 6))
plt.title('Comparison of Ensemble Methods')
plt.ylabel('Accuracy')
plt.ylim(0.8, 1.0)  # Adjust if needed
plt.tight_layout()
plt.show()
```

```
                  Train Accuracy  Test Accuracy
Random Forest                1.0            1.0
AdaBoost                     1.0            1.0
Gradient Boosting            1.0            1.0
```

```
<Figure size 3000x1800 with 0 Axes>
```



7.5.1 Bagging vs. Boosting

7.5.1.1 Bagging (Random Forest)

- **Goal**: Reduce variance (overfitting)
- **Training**: Parallel (trees are independent)
- **Weighting**: Equal weight for all models
- **Bias-Variance**: Primarily reduces variance
- **Robustness**: Less prone to overfitting
- **Speed**: Can be parallelized easily

7.5.1.2 Boosting (Gradient Boosting)

- **Goal**: Reduce bias and variance
- **Training**: Sequential (each tree depends on previous trees)
- **Weighting**: Different weights for different models
- **Bias-Variance**: Reduces both bias and variance
- **Robustness**: More prone to overfitting
- **Speed**: Generally slower due to sequential nature

7.5.2 Stacking

Stacking combines multiple models using a meta-learner:

1. Train base models on the original dataset
2. Generate predictions from each base model
3. Use these predictions as features to train a meta-model
4. Final prediction is given by the meta-model

7.5.3 Choosing the Right Ensemble Method

The choice depends on the problem characteristics:

- **Random Forest**: Good default for most problems, especially with limited data
- **Gradient Boosting**: When maximum performance is needed and you can tune hyperparameters
- **AdaBoost**: Simple boosting algorithm, good for weak learners
- **Stacking**: When you have diverse models and computational resources
- **Voting**: Simple ensemble when you already have several good models

7.5.4 Practical Considerations

When implementing ensemble methods:

1. **Computational Resources**: Boosting methods are generally more resource-intensive
2. **Model Complexity**: Simpler models may be preferred for production
3. **Interpretability Requirements**: Random forests offer better interpretability than boosting
4. **Dataset Size**: For small datasets, random forests may be more appropriate
5. **Noise Level**: For noisy data, bagging methods are more robust

7.6 Conclusion

- Decision Trees provide interpretable models for both classification and regression
- Ensemble methods like Random Forest and Gradient Boosting improve upon Decision Trees by combining multiple models
- Random Forest reduces variance through bagging and random feature selection
- Gradient Boosting reduces both bias and variance through sequential model building
- Each method has strengths and weaknesses depending on the specific problem and dataset

7.6.1 Key Takeaways

1. **No Free Lunch**: No single algorithm is best for all problems
2. **Bias-Variance Tradeoff**: Different ensemble methods address different aspects of model error
3. **Hyperparameter Tuning**: Proper tuning is crucial for optimal performance
4. **Interpretability vs. Performance**: More complex ensembles usually offer better performance at the cost of interpretability
5. **Computational Considerations**: Training time and resource requirements vary significantly between methods

```
# Final comprehensive example: train models on different datasets and compare

from sklearn.datasets import load_breast_cancer, load_wine
from sklearn.preprocessing import StandardScaler

datasets = {
    'Iris': load_iris(),
    'Breast Cancer': load_breast_cancer(),
    'Wine': load_wine()
```

```
}

results = []

for name, dataset in datasets.items():
    X, y = dataset.data, dataset.target

    # Scale features
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

    # Train models
    dt = DecisionTreeClassifier(max_depth=4, random_state=42)
    rf = RandomForestClassifier(n_estimators=100, max_depth=4, random_state=42)
    gb = GradientBoostingClassifier(n_estimators=100, max_depth=3, random_state=42)

    models = {'Decision Tree': dt, 'Random Forest': rf, 'Gradient Boosting': gb}

    # Evaluate
    for model_name, model in models.items():
        model.fit(X_train, y_train)
        train_acc = model.score(X_train, y_train)
        test_acc = model.score(X_test, y_test)

        results.append({
            'Dataset': name,
            'Model': model_name,
            'Train Accuracy': train_acc,
            'Test Accuracy': test_acc
        })

# Create DataFrame with results
final_results = pd.DataFrame(results)
print(final_results.pivot_table(index='Dataset', columns='Model', values='Test Accuracy'))
```

```
Model          Decision Tree  Gradient Boosting  Random Forest
Dataset
Breast Cancer       0.953216           0.959064        0.97076
Iris                1.000000           1.000000        1.00000
```

```
Wine                0.962963          0.907407        1.00000
```

7.6.2 Further Research Directions

- **Explainable AI**: Methods to interpret complex ensemble models
- **Automatic Machine Learning (AutoML)**: Automating the selection and tuning of ensemble methods
- **Deep Forest**: Combining deep learning concepts with random forests
- **Online Learning**: Adapting ensemble methods for streaming data
- **Imbalanced Learning**: Specialized ensemble techniques for imbalanced datasets

8 Support Vector Machines and Model Evaluation

8.1 Support Vector Machine (SVM)

Support Vector Machines (SVMs) are powerful supervised learning algorithms used for both classification and regression tasks. Their primary goal is to find the optimal hyperplane that maximizes the margin between different classes.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

8.1.1 Key Concepts

- **Maximizes the margin** between classes to improve generalization
- **Support vectors** are the data points closest to the decision boundary
- **Decision boundary (hyperplane)** separates the classes
- Based on optimization theory with learning bias derived from statistical learning theory

8.1.1.1 Conceptual Understanding

SVMs work by finding the hyperplane that creates the largest margin between the two classes in the training data. This margin is defined as the perpendicular distance between the decision boundary and the closest data points from each class (the support vectors).

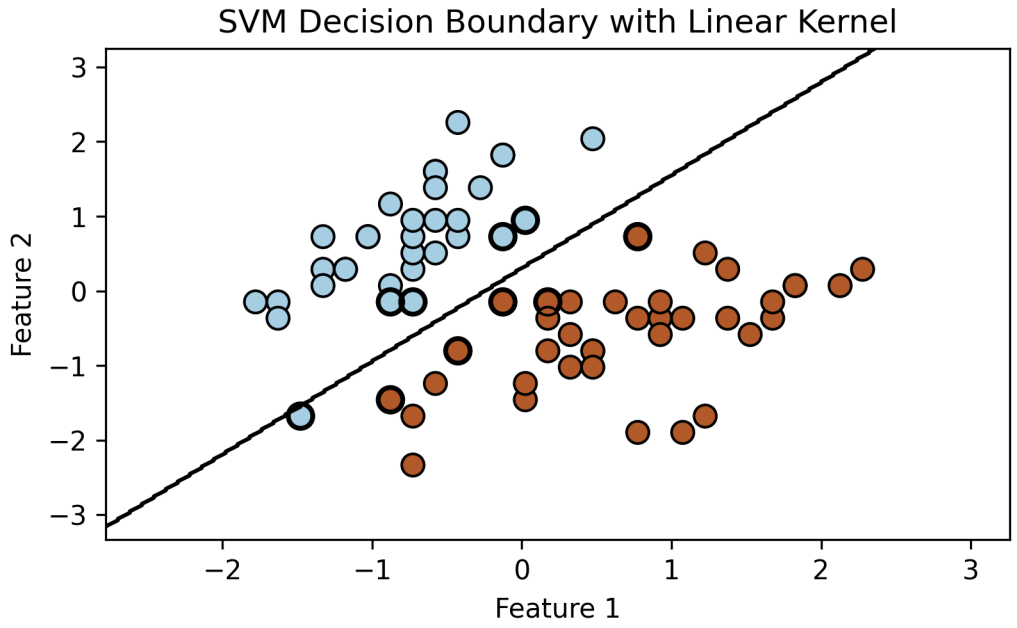
The mathematical objective of an SVM can be expressed as: - Maximize the margin width (2/||w||) - Subject to constraints that ensure no data points fall within the margin

The optimization problem becomes: - Minimize (1/2)||w||² subject to y_i(w · x_i + b) 1 for all training points (x_i, y_i)

For non-linearly separable data, SVMs introduce "slack variables" ( ) that allow some points to violate the margin: - Minimize (1/2)||w||² + C · Σ _i subject to y_i(w · x_i + b) 1 - _i and _i 0

```
# Load and prepare sample data
iris = datasets.load_iris()
X = iris.data[:, :2]  # Using first two features for visualization
y = iris.target

# Only use two classes for binary classification example
X = X[y != 2]
y = y[y != 2]

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train SVM classifier with linear kernel
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X_train, y_train)

# Create a mesh to plot in
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
h = 0.02  # Fixed step size for the mesh grid
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# Plot the decision boundary
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], linestyles=['--', '-', '--'])

# Plot the training points
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired,
            edgecolors='black', s=70)

# Highlight the support vectors
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
```

```
            linewidth=1, facecolors='none', edgecolors='k')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('SVM Decision Boundary with Linear Kernel')
plt.tight_layout()
plt.show()

# Print model accuracy
print(f"Training accuracy: {clf.score(X_train, y_train):.3f}")
print(f"Testing accuracy: {clf.score(X_test, y_test):.3f}")
```




```
Training accuracy: 0.986
Testing accuracy: 1.000
```

8.1.2 The Dual Problem and Support Vectors

SVMs are often solved in their dual form using Lagrange multipliers, which makes the kernel trick possible. The dual optimization problem becomes:

- Maximize Σ _i - (1/2)ΣΣ _i · _j · y_i · y_j · K(x_i, x_j)
- Subject to 0 _i C and Σ _i · y_i = 0

In this formulation: - Data points with _i > 0 are the support vectors - Support vectors are the critical elements that define the decision boundary - Only a subset of training points become support vectors, making SVM memory-efficient

8.1.3 SVM for Classification and Regression

SVMs can be used for both classification and regression tasks:

8.1.3.1 SVM Classification (SVC)

- **Binary Classification**: Finds the optimal hyperplane to separate two classes
- **Multiclass Classification**: Uses strategies like one-vs-rest or one-vs-one
- **Probabilistic Outputs**: Can be calibrated to provide probability estimates

8.1.3.2 SVM Regression (SVR)

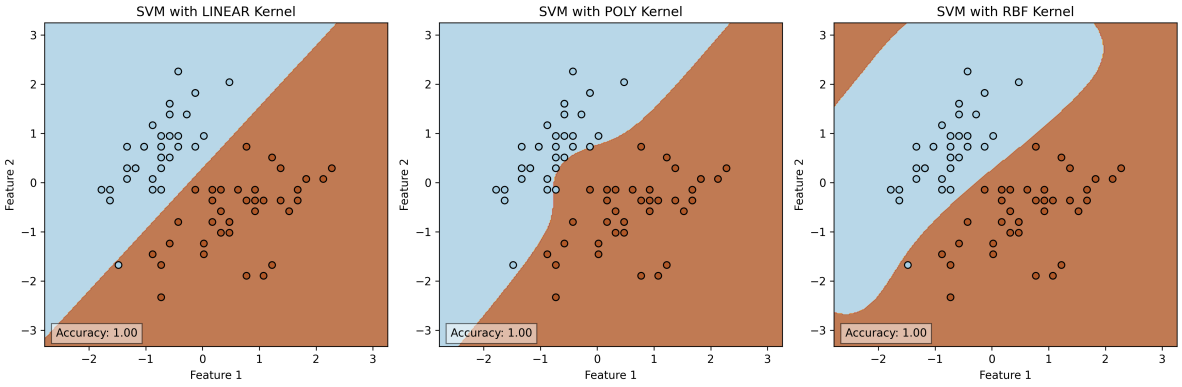
- Predicts continuous values by finding a function that has at most deviation from the targets
- Uses an -insensitive loss function: only errors greater than are penalized
- Applies the same principles of margin maximization but for regression

8.1.4 Non-linear Classification with Kernels

When data isn't linearly separable in the original feature space, SVMs use the **kernel trick** to implicitly map data into a higher-dimensional space where linear separation becomes possible:

```
# Compare different SVM kernels
kernels = ['linear', 'poly', 'rbf']
plt.figure(figsize=(15, 5))

for i, kernel in enumerate(kernels):
    clf = svm.SVC(kernel=kernel, gamma='scale')
    clf.fit(X_train, y_train)

    # Plot the decision boundary
    plt.subplot(1, 3, i+1)

    # Create a mesh to plot in
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
```

```
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Plot the decision boundary
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.Paired)

    # Plot the training points
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired,
                edgecolors='black')

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(f'SVM with {kernel.upper()} Kernel')

    accuracy = clf.score(X_test, y_test)
    plt.text(x_min + 0.2, y_min + 0.2, f'Accuracy: {accuracy:.2f}',
             bbox=dict(facecolor='white', alpha=0.5))

plt.tight_layout()
plt.show()
```



8.1.4.1 Understanding the Kernel Trick

The kernel trick works by computing the dot product between the transformed data points without explicitly calculating the transformation:

$K(x, y) = (x) \cdot (y)$

Where: - K is the kernel function - is the transformation function to a higher-dimensional space - x and y are data points in the original space

This allows SVM to operate in the transformed space without the computational burden of explicitly computing the transformation.

8.1.4.2 Common Kernel Types

1. **Linear kernel**: K(x, y) = x · y
   - Used when data is linearly separable
   - Simplest kernel with fewest parameters
   - Decision boundary is a straight line (2D) or hyperplane (higher dimensions)
2. **Radial Basis Function (RBF) kernel**: K(x, y) = exp(- ||x-y||²)
   - Effective for non-linear boundaries
   - Creates decision regions that can be distinct islands
   - controls the influence radius of each support vector
3. **Polynomial kernel**: K(x, y) = (x · y + r)^d
   - Creates complex curved decision boundaries
   - d is the polynomial degree
   - Higher degrees create more complex boundaries but risk overfitting
4. **Sigmoid kernel**: K(x, y) = tanh(x · y + r)
   - Inspired by neural networks
   - Creates decision boundaries similar to those of neural networks

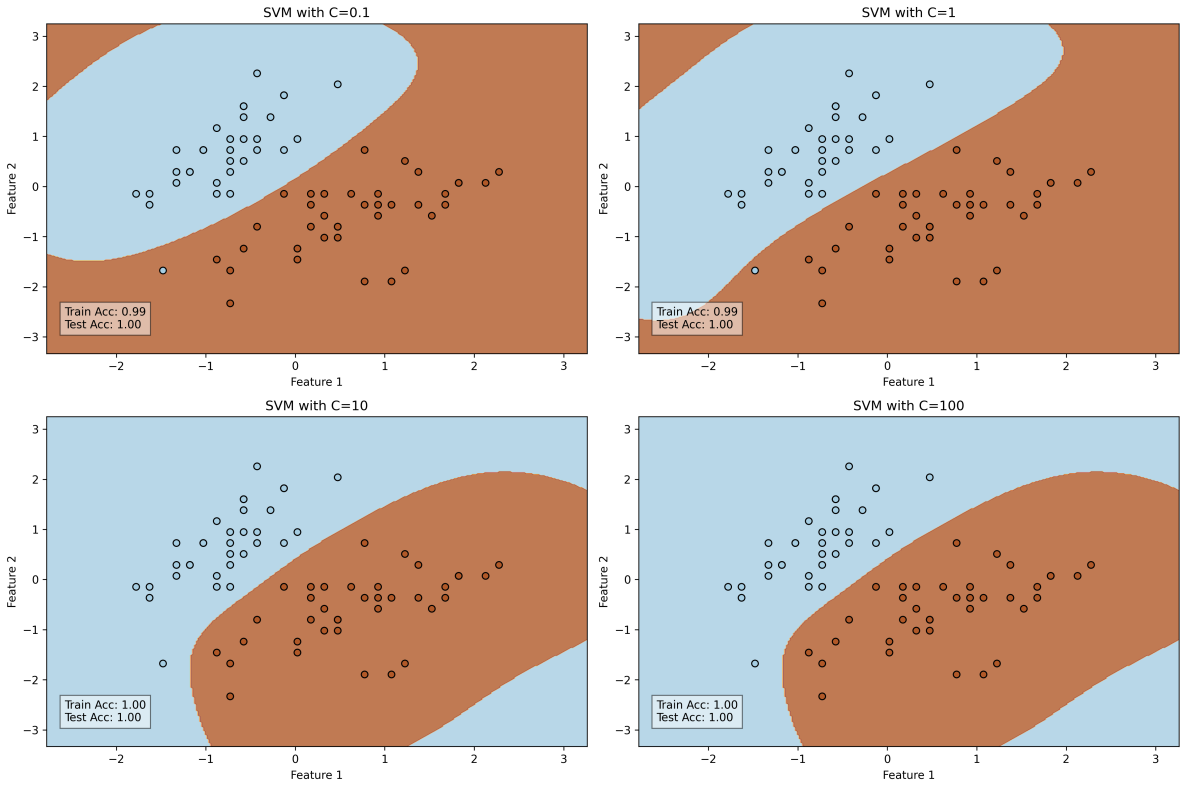
8.1.5 SVM Parameters

8.1.5.1 The Role of C (Regularization Parameter)

The `C` parameter represents the trade-off between model complexity and training error:

```
# Explore the effect of C parameter
C_values = [0.1, 1, 10, 100]
plt.figure(figsize=(15, 10))

for i, C in enumerate(C_values):
    clf = svm.SVC(kernel='rbf', C=C, gamma='scale')
    clf.fit(X_train, y_train)
```

```
    # Plot the decision boundary
    plt.subplot(2, 2, i+1)

    # Create a mesh to plot in
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Plot the decision boundary
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.Paired)

    # Plot the training points
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired,
                edgecolors='black')

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(f'SVM with C={C}')

    train_accuracy = clf.score(X_train, y_train)
    test_accuracy = clf.score(X_test, y_test)
    plt.text(x_min + 0.2, y_min + 0.5,
             f'Train Acc: {train_accuracy:.2f}\nTest Acc: {test_accuracy:.2f}',
             bbox=dict(facecolor='white', alpha=0.5))

plt.tight_layout()
plt.show()
```



- **Large C**: Penalizes misclassifications heavily
  - Creates a smaller-margin hyperplane that attempts to classify all training examples correctly
  - May lead to overfitting by creating complex decision boundaries
  - More support vectors are typically used
- **Small C**: Allows more misclassifications
  - Creates a wider margin that may misclassify more training points
  - Produces simpler, more generalized models
  - Fewer support vectors are typically used

8.1.5.2 The Role of Gamma (Kernel Coefficient)

For RBF, polynomial, and sigmoid kernels, the `gamma` parameter defines how far the influence of a single training example reaches:

- **Large gamma**: Results in a decision boundary that closely follows individual training examples
  - May lead to overfitting as the model becomes too specialized to training data

  - Creates more complex, tightly curved decision boundaries

- **Small gamma**: Gives points far away from the decision boundary more influence
  - Creates smoother, simpler decision boundaries
  - May lead to underfitting if too small

8.1.6 Strengths and Weaknesses of SVM

8.1.6.1 Strengths

- **No Local Minima**: The optimization problem is convex, ensuring a globally optimized solution
- **Memory Efficiency**: Only support vectors are needed to define the decision boundary
- **Versatility**: Effective with both linear and non-linear data via different kernels
- **Robustness**: Less prone to overfitting in high-dimensional spaces
- **Theoretical Guarantees**: Based on statistical learning theory with solid mathematical foundations

8.1.6.2 Weaknesses

- **Computationally Intensive**: For large datasets, especially with non-linear kernels ($O(n^2)$ to $O(n^3)$ complexity)
- **Sensitive to Parameters**: Performance depends on appropriate kernel and parameter selection
- **Black Box Nature**: Limited interpretability compared to simpler models
- **Not Directly Probabilistic**: Requires additional calibration for probability outputs
- **Struggles with Highly Imbalanced Data**: Without adjustment, tends to favor the majority class

8.1.7 Practical Example: Full Iris Dataset

```
# Use all features and classes from the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```python
# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train SVM classifier
clf = svm.SVC(kernel='rbf', C=1.0, gamma='scale')
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Print performance metrics
from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

```
Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

8.2 Performance Metrics in Machine Learning

Performance metrics are crucial for evaluating how well machine learning models perform. The choice of metric depends on the type of task (classification or regression) and the specific problem requirements.

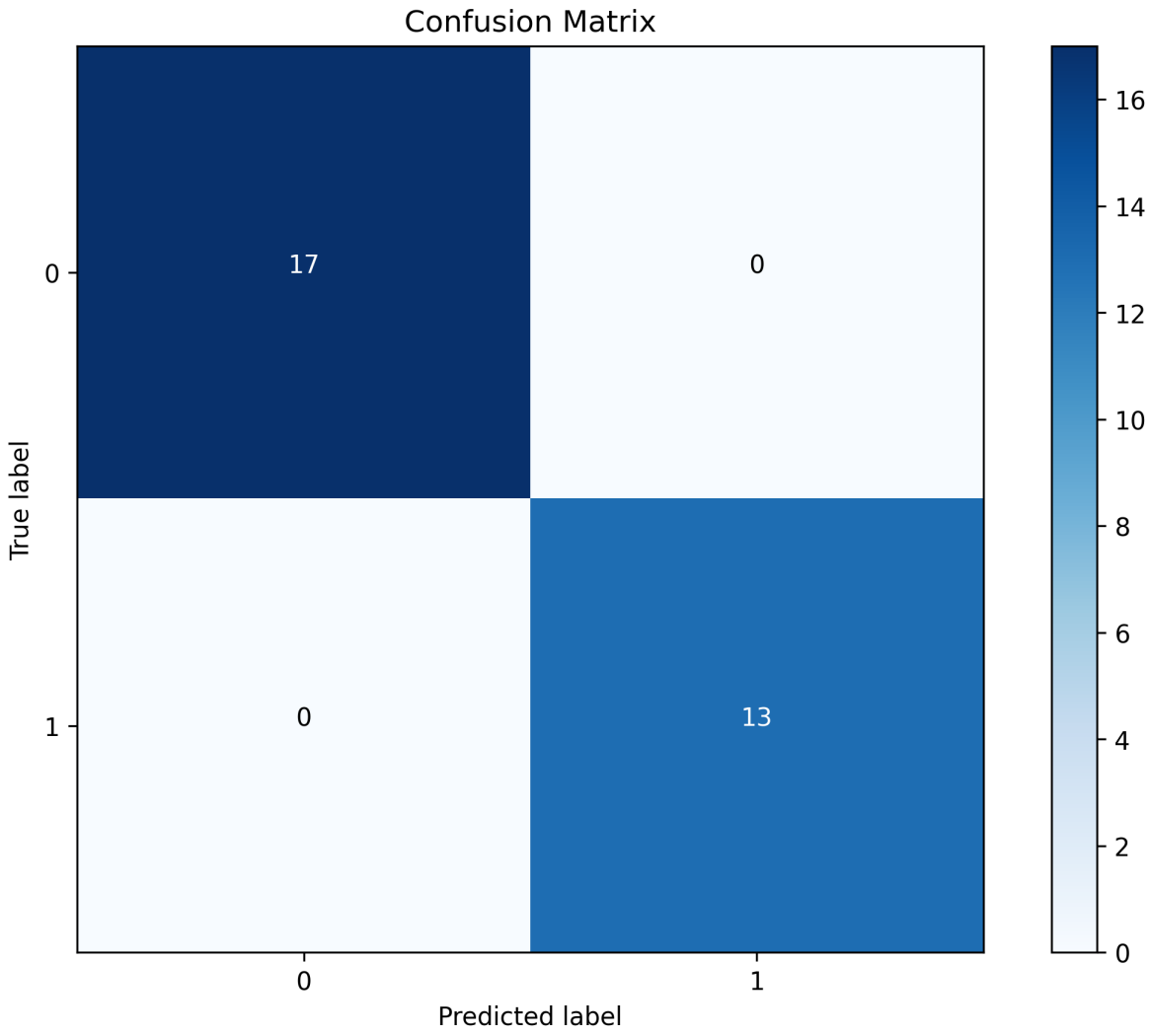
8.2.1 Classification Metrics

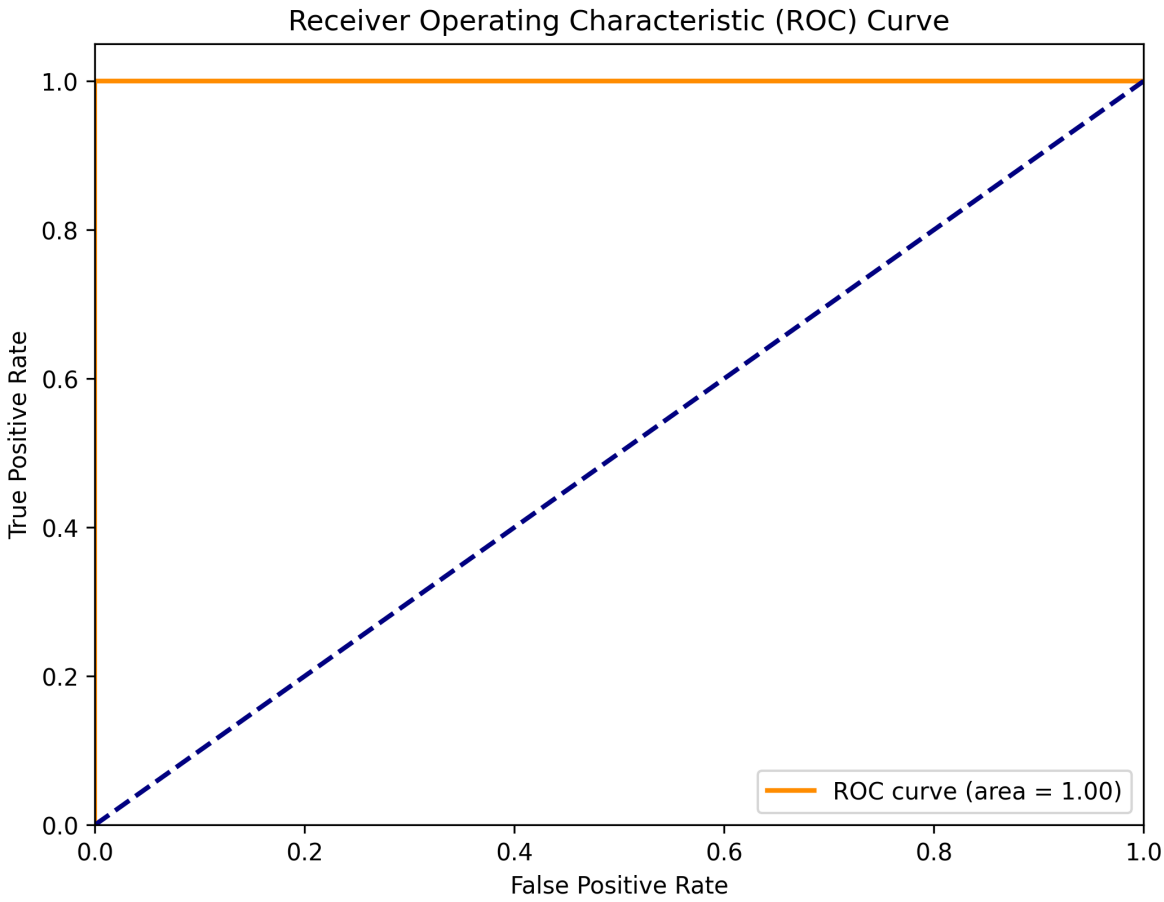
```
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, roc_curve, auc, precision_recall_curve

# Function to calculate and display all classification metrics
def evaluate_classification(y_true, y_pred, y_scores=None):
    """
    Calculate classification metrics

    Parameters:
    y_true: True labels
    y_pred: Predicted labels
    y_scores: Predicted probabilities (for ROC and PR curves)
    """
    # Basic metrics
    accuracy = accuracy_score(y_true, y_pred)

    # For multi-class, we use 'macro' average
    precision = precision_score(y_true, y_pred, average='macro')
    recall = recall_score(y_true, y_pred, average='macro')
    f1 = f1_score(y_true, y_pred, average='macro')

    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")

    # Confusion Matrix
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Confusion Matrix')
    plt.colorbar()

    classes = np.unique(y_true)
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    # Add text annotations
```

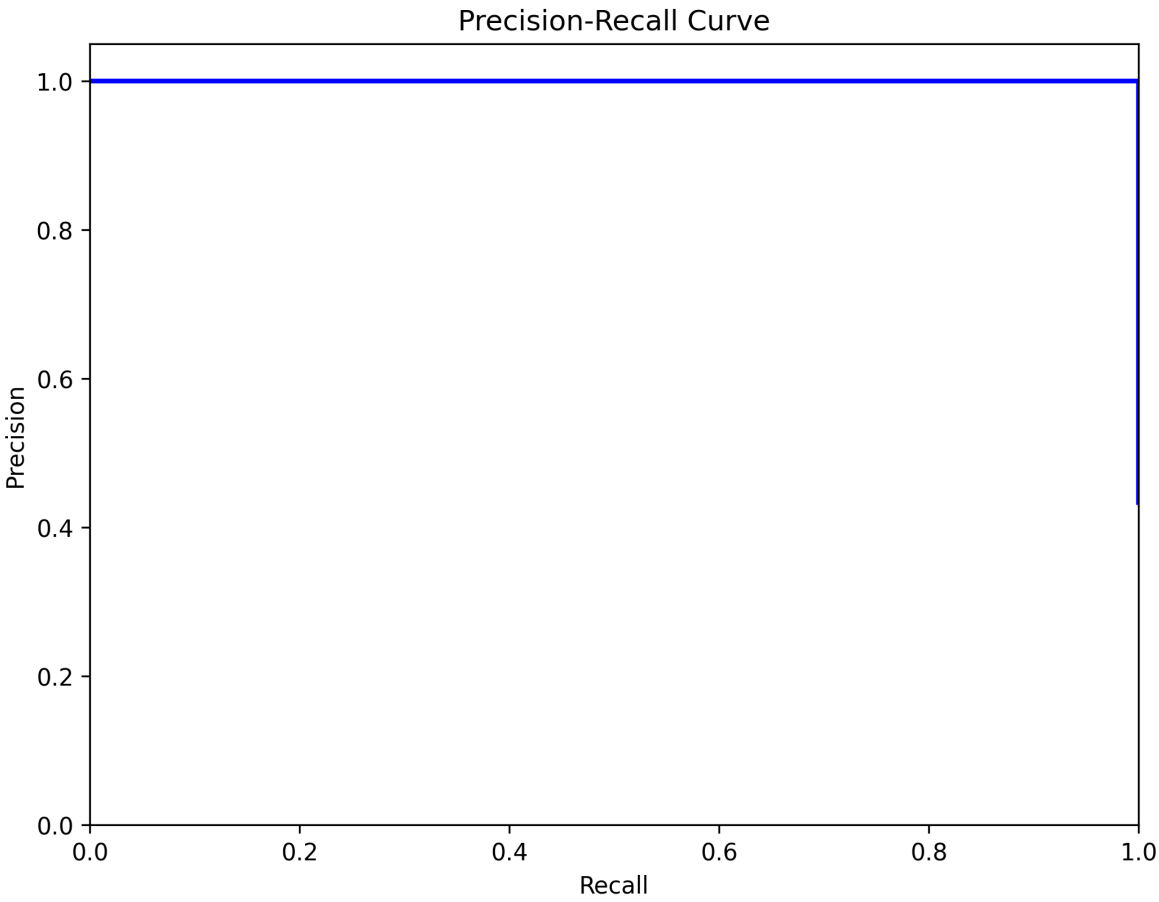
```
    thresh = cm.max() / 2
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, format(cm[i, j], 'd'),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()

    # If we have probability scores, calculate ROC curve (for binary classification)
    if y_scores is not None and len(np.unique(y_true)) == 2:
        # ROC Curve
        fpr, tpr, _ = roc_curve(y_true, y_scores)
        roc_auc = auc(fpr, tpr)

        plt.figure(figsize=(8, 6))
        plt.plot(fpr, tpr, color='darkorange', lw=2,
                 label=f'ROC curve (area = {roc_auc:.2f})')
        plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('Receiver Operating Characteristic (ROC) Curve')
        plt.legend(loc="lower right")
        plt.show()

        # Precision-Recall Curve
        precision, recall, _ = precision_recall_curve(y_true, y_scores)

        plt.figure(figsize=(8, 6))
        plt.plot(recall, precision, color='blue', lw=2)
        plt.xlabel('Recall')
        plt.ylabel('Precision')
        plt.title('Precision-Recall Curve')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.show()
```

```
# Example using our SVM model from earlier
# For binary classification demo
from sklearn.svm import SVC

# Create binary classification problem
binary_X = iris.data[iris.target != 2]
binary_y = iris.target[iris.target != 2]

X_train, X_test, y_train, y_test = train_test_split(binary_X, binary_y,
                                                    test_size=0.3, random_state=42)

# Create SVM model with probability output
binary_clf = SVC(kernel='rbf', probability=True)
binary_clf.fit(X_train, y_train)

# Get predictions and probability scores
y_pred = binary_clf.predict(X_test)
y_scores = binary_clf.predict_proba(X_test)[:, 1]  # Probability of class 1

print("Binary Classification Metrics:")
evaluate_classification(y_test, y_pred, y_scores)
```

```
Binary Classification Metrics:
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
```


Confusion Matrix
17
0
0
13
0
1
0
1
True label
Predicted label
16
14
12
10
8
6
4
2
0


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 1.00)



8.2.1.1 1. Accuracy

Measures the proportion of correctly classified instances: - Formula: Accuracy = (TP + TN) / (TP + TN + FP + FN) - Simple and intuitive, but can be misleading for imbalanced datasets - Example: In a dataset with 95% negative samples, a model that always predicts "negative" would have 95% accuracy despite being useless

8.2.1.2 2. Precision

Measures how many predicted positives were actually positive: - Formula: Precision = TP / (TP + FP) - Answers: "Of all instances predicted as positive, how many were actually positive?" - Important when false positives are costly (e.g., spam detection, medical diagnosis) - High precision indicates low false positive rate

8.2.1.3 3. Recall (Sensitivity or True Positive Rate)

Measures how many actual positives were correctly predicted: - Formula: Recall = TP / (TP + FN) - Answers: "Of all actual positive instances, how many did we correctly identify?" - Important when false negatives are costly (e.g., disease screening, fraud detection) - High recall indicates low false negative rate

8.2.1.4 4. F1 Score

Harmonic mean of precision and recall: - Formula: F1 Score = 2 × (Precision × Recall) / (Precision + Recall) - Balances precision and recall into a single metric - Particularly useful for imbalanced datasets - F1 Score is low if either precision or recall is low

8.2.1.5 5. Confusion Matrix

Tabular representation of actual vs. predicted values: - True Positives (TP): Correctly predicted positives - True Negatives (TN): Correctly predicted negatives - False Positives (FP): Incorrectly predicted positives (Type I error) - False Negatives (FN): Incorrectly predicted negatives (Type II error)

The confusion matrix provides a comprehensive view of model performance and serves as the basis for calculating most classification metrics.

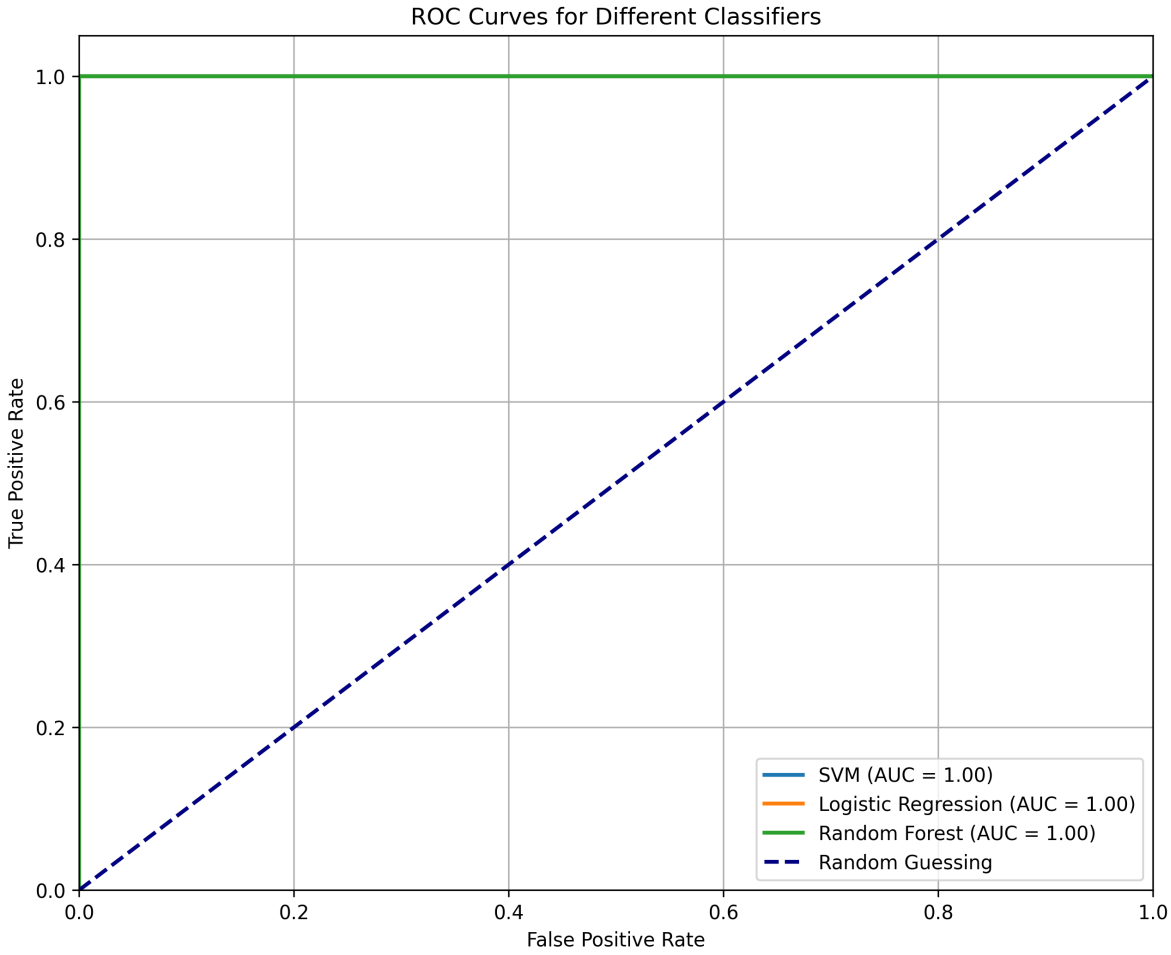
8.2.2 Understanding ROC and Precision-Recall Curves

8.2.2.1 ROC Curve (Receiver Operating Characteristic)

The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various classification thresholds:

```
# ROC Curve across different classification thresholds
from sklearn.metrics import roc_curve, auc
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

# Train multiple classifiers on the same binary dataset
classifiers = {
    'SVM': SVC(kernel='rbf', probability=True, random_state=42),
    'Logistic Regression': LogisticRegression(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42)
}
```

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

# Plot ROC for each classifier
for name, clf in classifiers.items():
    clf.fit(X_train, y_train)
    y_scores = clf.predict_proba(X_test)[:, 1]

    fpr, tpr, _ = roc_curve(y_test, y_scores)
    roc_auc = auc(fpr, tpr)

    plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC = {roc_auc:.2f})')

# Plot the random guessing line
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guessing')

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Different Classifiers')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```



8.2.2.2 ROC Curve Components

- **True Positive Rate (TPR)**: TP / (TP + FN) - y-axis
  - Also known as Recall or Sensitivity
  - Measures the proportion of actual positives correctly identified
- **False Positive Rate (FPR)**: FP / (FP + TN) - x-axis
  - Also known as (1 - Specificity)
  - Measures the proportion of actual negatives incorrectly classified as positive
- **Classification Threshold**: Each point on the curve represents a different threshold
  - Moving along the curve represents changing the threshold for classifying a sample as positive

  - Lower thresholds yield higher TPR but also higher FPR (upper right)
  - Higher thresholds yield lower TPR but also lower FPR (lower left)

8.2.2.3 AUC (Area Under the ROC Curve)

The AUC metric quantifies the overall performance of a classifier:

- **AUC = 1.0**: Perfect classification (100% sensitivity, 100% specificity)
- **AUC = 0.5**: No better than random guessing (shown as diagonal line)
- **AUC < 0.5**: Worse than random guessing (rare, usually indicates an error)

AUC has an important probabilistic interpretation: if you randomly select a positive and a negative example, the AUC represents the probability that the classifier will rank the positive example higher than the negative one.

8.2.2.4 Precision-Recall Curve

The Precision-Recall curve plots Precision against Recall at various thresholds: - Particularly useful for imbalanced datasets where ROC curves may be overly optimistic - The higher the curve (toward the upper-right corner), the better the model - The area under the PR curve (AUPRC) is another useful aggregate measure

8.2.2.5 When to Use ROC vs. PR Curves

- **ROC Curves**: Better when working with balanced datasets or when both classes are equally important
- **Precision-Recall Curves**: Better for imbalanced datasets or when the positive class is of particular interest

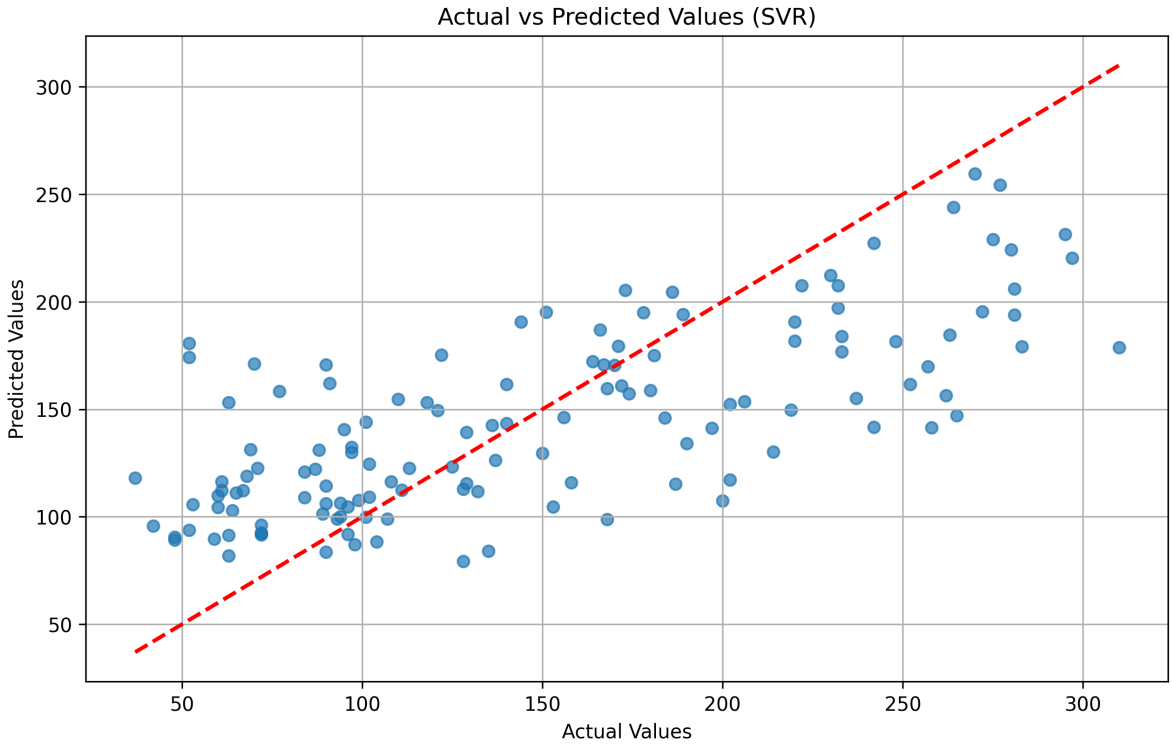
8.2.3 Regression Metrics

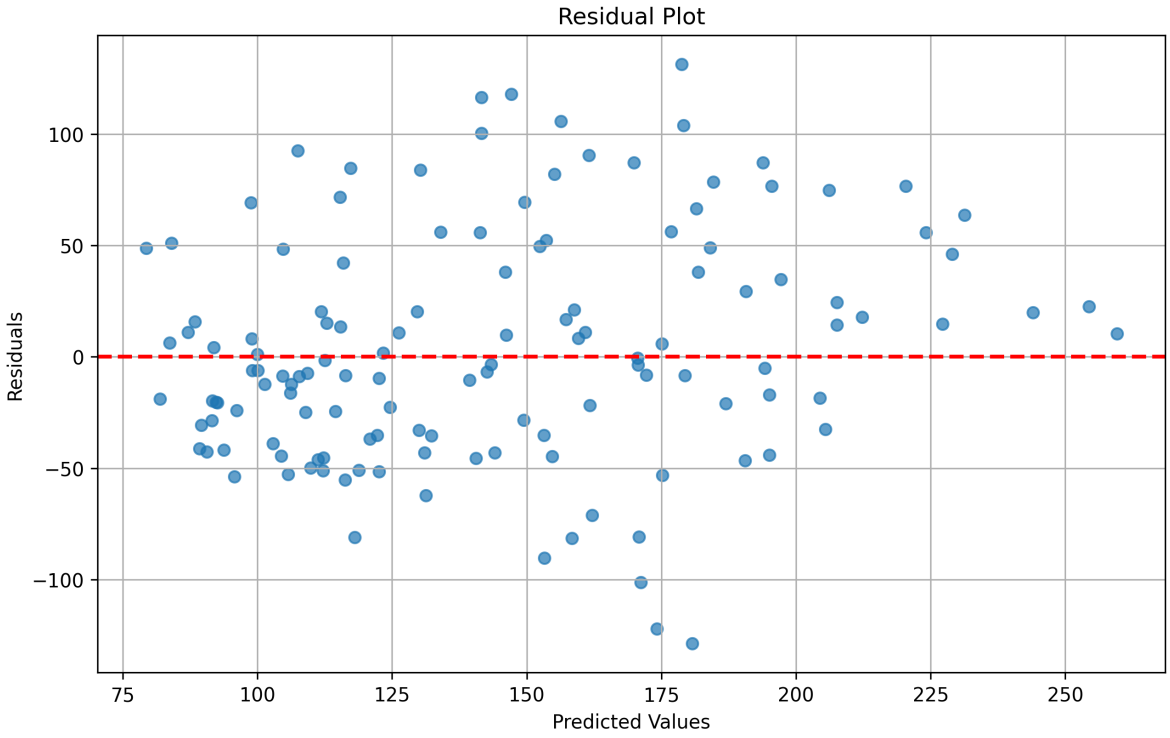
```
# Regression metrics example
from sklearn.datasets import load_diabetes
from sklearn.svm import SVR
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Load diabetes dataset
diabetes = load_diabetes()
X, y = diabetes.data, diabetes.target
```

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train SVR model
svr = SVR(kernel='rbf', C=10, gamma='scale')
svr.fit(X_train, y_train)

# Make predictions
y_pred = svr.predict(X_test)

# Calculate regression metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R-squared (R²): {r2:.2f}")

# Visualize actual vs predicted values
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs Predicted Values (SVR)')
plt.grid(True)
plt.show()

# Plot residuals
residuals = y_test - y_pred
plt.figure(figsize=(10, 6))
plt.scatter(y_pred, residuals, alpha=0.7)
plt.axhline(y=0, color='r', linestyle='--', lw=2)
plt.xlabel('Predicted Values')
```

```
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.grid(True)
plt.show()
```

```
Mean Absolute Error (MAE): 41.38
Mean Squared Error (MSE): 2703.38
Root Mean Squared Error (RMSE): 51.99
R-squared (R²): 0.50
```





8.2.3.1 1. Mean Absolute Error (MAE)

Average absolute difference between predictions and actual values: - Formula: MAE = (1/n) × Σ|yi - ŷi| - Units match the target variable - Robust to outliers since it doesn't square the errors - All errors are weighted equally regardless of magnitude

8.2.3.2 2. Mean Squared Error (MSE)

Average squared difference between predictions and actual values: - Formula: MSE = (1/n) × Σ(yi - ŷi)² - Penalizes larger errors more due to squaring - More sensitive to outliers than MAE - Units are squared (not directly interpretable in terms of the original target)

8.2.3.3 3. Root Mean Squared Error (RMSE)

Square root of MSE, providing error in the same units as the target variable: - Formula: RMSE = √MSE - More interpretable than MSE - Still penalizes large errors more than small ones - Often used as the primary metric for regression models

8.2.3.4 4. R-Squared (R^2)

Proportion of variance explained by the model: - Formula: $R^2 = 1 - (\Sigma(yi - \hat{y}i)^2) / (\Sigma(yi - \bar{y})^2)$ - Range: 0 to 1 (higher is better) - $R^2 = 0$ means the model is no better than predicting the mean - $R^2 = 1$ means perfect prediction - Can be negative if model performs worse than just predicting the mean - Scale-free, allowing comparison across different target scales

8.2.3.5 5. Adjusted R-Squared

Modified version of R^2 that adjusts for the number of predictors: - Formula: Adjusted $R^2 = 1 - [(1 - R^2)(n - 1)/(n - p - 1)]$ - Helps prevent overfitting by penalizing excessive features - Increases only if new features improve the model more than would be expected by chance

8.2.4 Understanding Residual Plots

Residual plots (predicted values vs. residuals) help diagnose model performance:

- **Random scatter around zero**: Indicates a good fit
- **Funnel shape**: Indicates heteroscedasticity (non-constant variance)
- **Curved pattern**: Suggests non-linear relationships weren't captured
- **Clusters**: May indicate model misspecification or segmented data

8.3 Model Fit Issues

8.3.1 Underfitting vs. Overfitting

9 Demonstrate underfitting vs. overfitting with polynomial regression

from sklearn.preprocessing import PolynomialFeatures from sklearn.linear_model import LinearRegression from sklearn.pipeline import Pipeline

10 Generate synthetic data

np.random.seed(42) X = np.sort(5 * np.random.rand(80, 1), axis=0) y = np.sin(X).ravel() + 0.1 * np.random.randn(80)

11 Split into train and test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

12 Create polynomials of different degrees

degrees = [1, 3, 15] # Underfitting, good fit, overfitting plt.figure(figsize=(16, 5))

for i, degree in enumerate(degrees): # Create pipeline with polynomial features and linear regression model = Pipeline([('poly', PolynomialFeatures(degree=degree)), ('linear', LinearRegression())])

```
# Fit model
model.fit(X_train, y_train)

# Get predictions
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Calculate scores
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Plot results
plt.subplot(1, 3, i+1)

# Sort for smooth curve plotting
X_plot = np.linspace(0, 5, 100).reshape(-1, 1)
y_plot = model.predict(X_plot)

# Plot data and model
plt.scatter(X_train, y_train, color='blue', s=30, alpha=0.4, label='Training data')
plt.scatter(X_test, y_test, color='red', s=30, alpha=0.4, label='Testing data')
plt.plot(X_plot, y_plot, color='green', label=f'Degree {degree} polynomial')

plt.title(f'Polynomial Degree {degree}\nTrain R²: {train_r2:.2f}, Test R²: {test_r2:.2f}')
```

13 Outlier Detection and Recommendation Systems

Outlier detection is a fundamental aspect of data analysis, helping to identify data points that significantly deviate from the overall pattern. These anomalies can indicate errors, rare events, or interesting insights that merit further investigation.

Why Outlier Detection Matters

Outliers can significantly impact statistical analyses, model performance, and business decisions. Detecting them is crucial for:

- Data cleaning and preprocessing
- Fraud detection
- Network intrusion detection
- Medical diagnosis (detecting abnormal test results)
- Manufacturing quality control

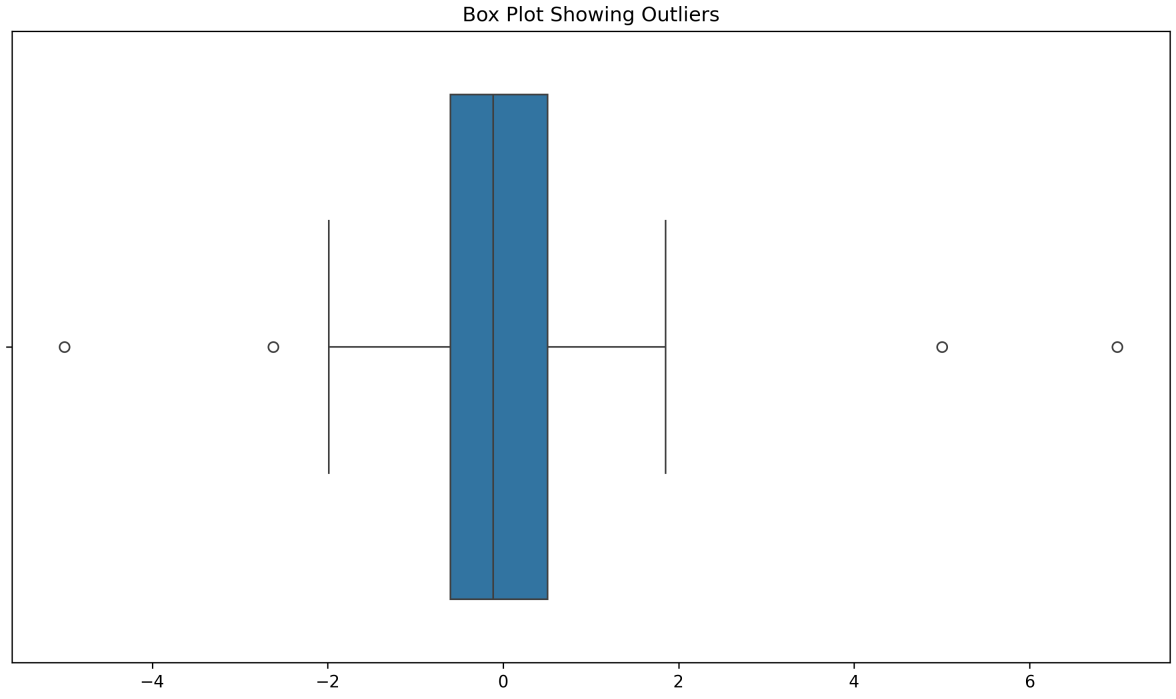
13.1 Graphical Outlier Detection

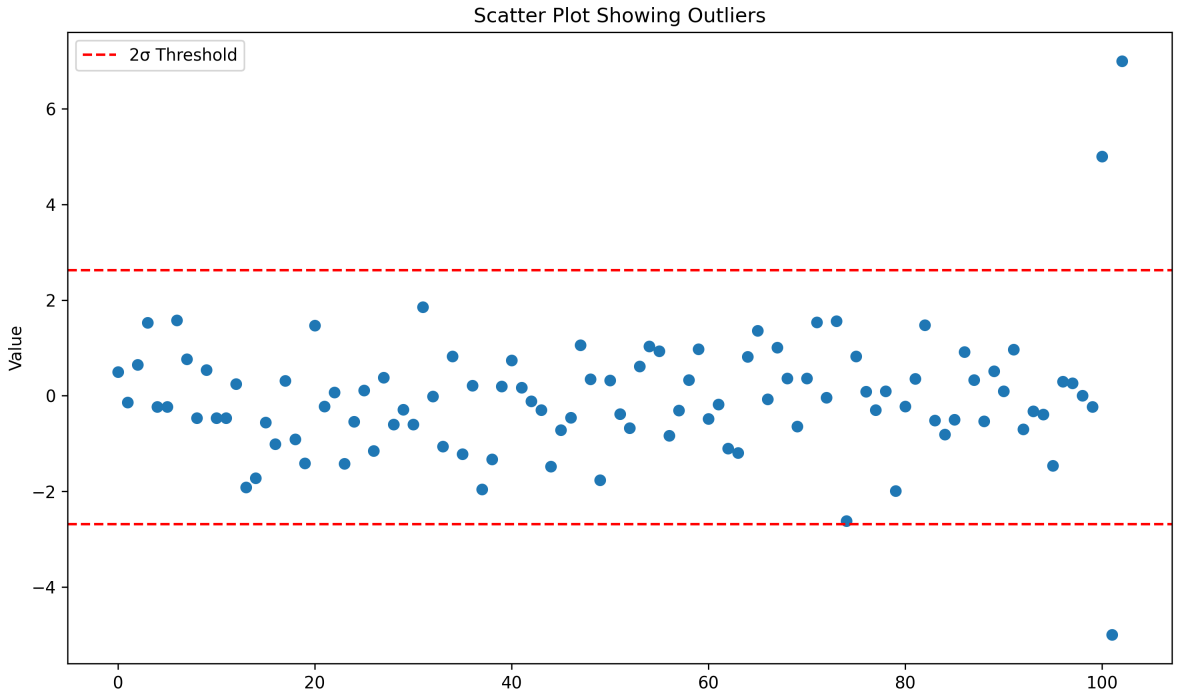
One of the simplest ways to detect outliers is through visualization. By plotting the data, human intuition can be leveraged to identify unusual points. Common graphical methods include:

- **Boxplots**: Provide a summary of the data distribution, highlighting potential outliers
- **Scatterplots**: Useful for detecting complex patterns in two-variable datasets
- **Histograms**: Help identify values that fall outside the typical distribution

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Generate sample data with outliers
np.random.seed(42)
```

```
data = np.random.normal(0, 1, 100)
data = np.append(data, [5, -5, 7])  # Add outliers

# Create a box plot
plt.figure(figsize=(10, 6))
sns.boxplot(x=data)
plt.title('Box Plot Showing Outliers')
plt.tight_layout()
plt.show()

# Create a scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(range(len(data)), data)
plt.title('Scatter Plot Showing Outliers')
plt.ylabel('Value')
plt.axhline(y=np.mean(data) + 2*np.std(data), color='r', linestyle='--', label='2 Threshold'
plt.axhline(y=np.mean(data) - 2*np.std(data), color='r', linestyle='--')
plt.legend()
plt.tight_layout()
plt.show()
```





13.1.1 Quartiles and the Boxplot Method

A boxplot divides data into quartiles, summarizing five key statistics:

- **Minimum**: The smallest value excluding outliers
- **First quartile (Q1)**: The median of the lower half (25% of data below Q1)
- **Median**: The middle value of the dataset
- **Third quartile (Q3)**: The median of the upper half (75% of data below Q3)
- **Maximum**: The largest value excluding outliers

A common rule for identifying outliers in boxplots is the 1.5 IQR rule:

- IQR (Interquartile Range) = Q3 - Q1
- Any value above Q3 + 1.5 × IQR or below Q1 - 1.5 × IQR is considered an outlier.

This method is robust to extreme values and doesn't assume a specific distribution, making it widely applicable.

```
# Find outliers using the IQR method
def find_outliers_iqr(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
```

```
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr

    outliers = [x for x in data if x < lower_bound or x > upper_bound]
    outlier_indices = [i for i, x in enumerate(data) if x < lower_bound or x > upper_bound]

    return outliers, outlier_indices, (lower_bound, upper_bound)

outliers, outlier_indices, bounds = find_outliers_iqr(data)
print(f"Outliers: {outliers}")
print(f"Outlier indices: {outlier_indices}")
print(f"Bounds (lower, upper): {bounds}")
```

```
Outliers: [np.float64(-2.6197451040897444), np.float64(5.0), np.float64(-5.0), np.float64(7.0
Outlier indices: [74, 100, 101, 102]
Bounds (lower, upper): (np.float64(-2.260417817278694), np.float64(2.164235959266887))
```

13.2 Cluster-Based Outlier Detection

This method involves clustering data points and identifying those that do not belong to any cluster or form small, isolated clusters. The fundamental assumption is that normal data points belong to large, dense clusters, while outliers either:

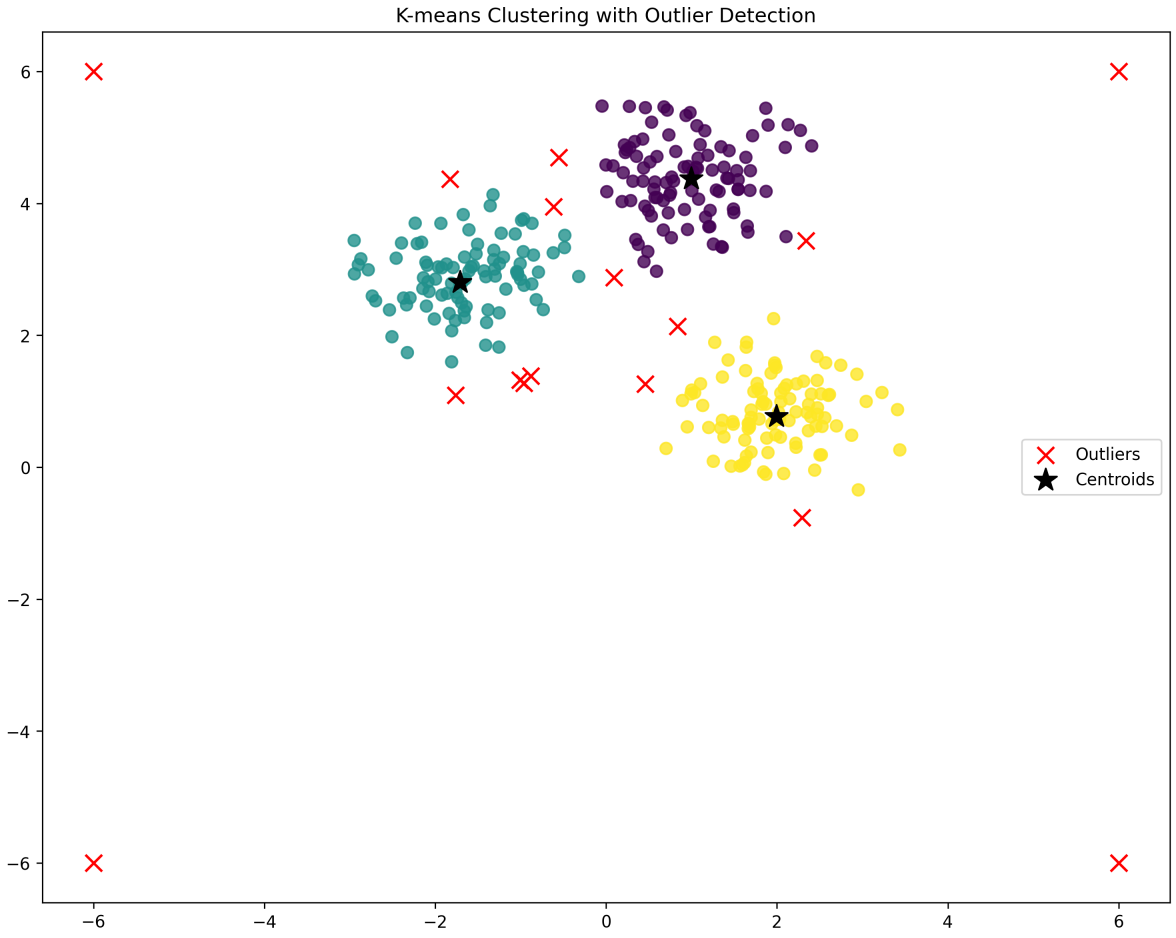
1. Form small clusters far from the main clusters
2. Do not belong to any cluster
3. Are assigned to a cluster but are far from the cluster center

Common clustering algorithms used for outlier detection include:

- **K-means Clustering**: Outliers are points that are far from any cluster mean or belong to a small cluster
- **Density-Based Clustering** (e.g., DBSCAN): Outliers are data points that remain unassigned to clusters
- **Hierarchical Clustering**: Outliers take longer to merge with other groups, making them distinguishable

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generate sample data with clusters and outliers
```

```
X, _ = make_blobs(n_samples=300, centers=3, cluster_std=0.60, random_state=0)
# Add some outliers
X = np.vstack([X, np.array([[6, 6], [-6, -6], [6, -6], [-6, 6]])])

# Apply K-means clustering
kmeans = KMeans(n_clusters=3, random_state=0)
cluster_labels = kmeans.fit_predict(X)
cluster_centers = kmeans.cluster_centers_

# Calculate distance of each point to its cluster center
distances = np.zeros(X.shape[0])
for i in range(X.shape[0]):
    cluster_idx = cluster_labels[i]
    distances[i] = np.linalg.norm(X[i] - cluster_centers[cluster_idx])

# Identify potential outliers (points with largest distances)
threshold = np.percentile(distances, 95)  # Top 5% as outliers
outlier_mask = distances > threshold

# Visualize the clusters and outliers
plt.figure(figsize=(10, 8))
plt.scatter(X[~outlier_mask, 0], X[~outlier_mask, 1], c=cluster_labels[~outlier_mask],
            cmap='viridis', marker='o', s=50, alpha=0.8)
plt.scatter(X[outlier_mask, 0], X[outlier_mask, 1], c='red', marker='x', s=100, label='Outli
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1], c='black', marker='*', s=200, label
plt.legend()
plt.title('K-means Clustering with Outlier Detection')
plt.tight_layout()
plt.show()
```



13.3 Distance-Based Outlier Detection

Rather than relying on visualization or clustering, distance-based methods use spatial relationships to detect anomalies. These approaches are particularly useful for high-dimensional data where visualization becomes challenging.

13.3.1 Global Distance-Based Detection (KNN)

The K-Nearest Neighbors (KNN) approach for outlier detection follows these steps:

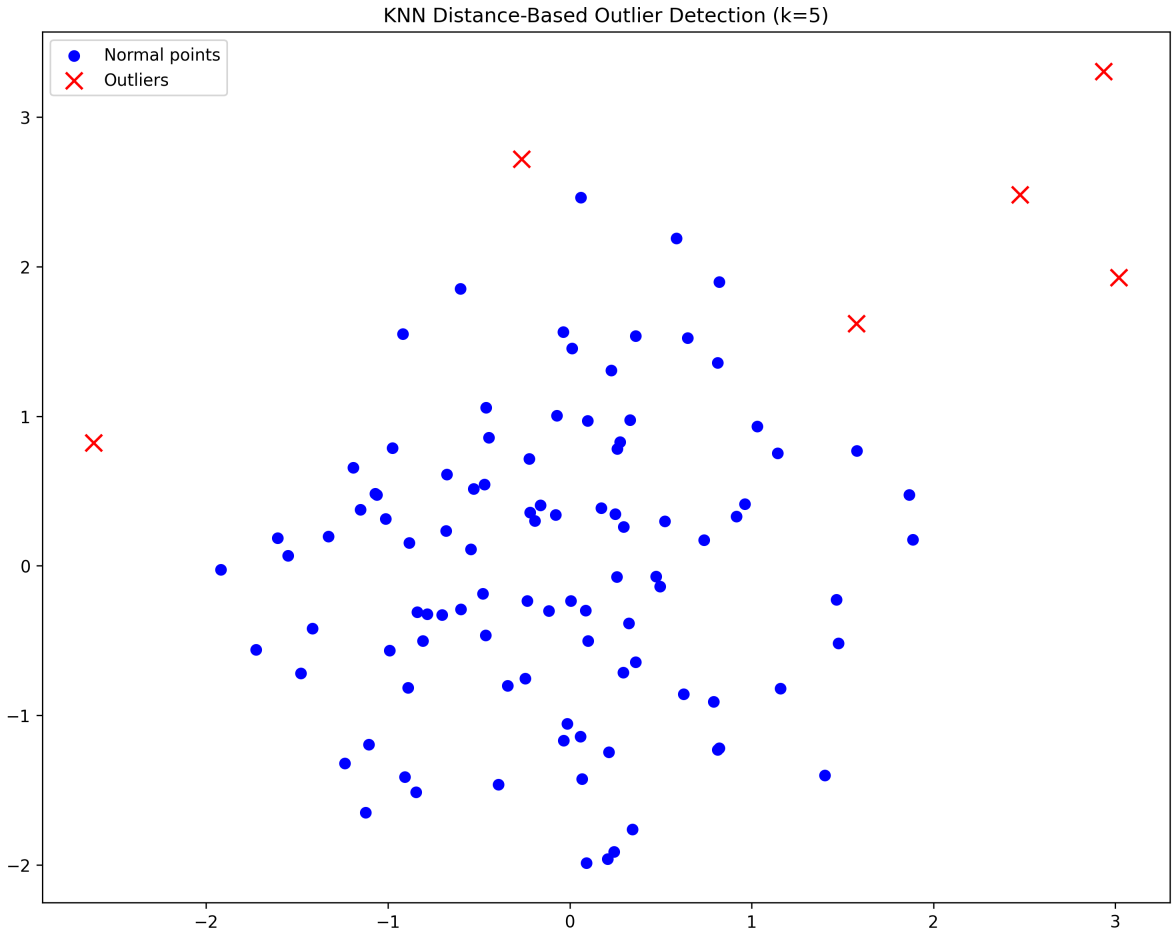
1. Compute the average distance of each point to its K-nearest neighbors
2. Sort these distances and flag the largest ones as outliers
3. This is useful for identifying global outliers that deviate from the overall data distribution

13.3.2 Local Distance-Based Detection

Local distance-based methods account for varying data densities by considering the locality of each point:

1. An outlier's 'outlierness' is determined by comparing its distance to neighbors relative to how far those neighbors are from their own neighbors
2. If the ratio exceeds 1, the point is flagged as an outlier
3. This approach can detect local outliers in datasets with varying densities

```
from sklearn.neighbors import NearestNeighbors

# Generate sample 2D data
np.random.seed(42)
X_normal = np.random.normal(0, 1, (100, 2))
X_outliers = np.random.uniform(-4, 4, (5, 2))
X = np.vstack([X_normal, X_outliers])

# Find k-nearest neighbors
k = 5
nbrs = NearestNeighbors(n_neighbors=k).fit(X)
distances, indices = nbrs.kneighbors(X)

# Calculate average distance to k-nearest neighbors
avg_knn_distance = distances[:, 1:].mean(axis=1)  # Exclude self (distance=0)

# Identify outliers
threshold = np.percentile(avg_knn_distance, 95)
outlier_mask = avg_knn_distance > threshold

# Visualize results
plt.figure(figsize=(10, 8))
plt.scatter(X[~outlier_mask, 0], X[~outlier_mask, 1], c='blue', label='Normal points')
plt.scatter(X[outlier_mask, 0], X[outlier_mask, 1], c='red', marker='x', s=100, label='Outli
plt.title(f'KNN Distance-Based Outlier Detection (k={k})')
plt.legend()
plt.tight_layout()
plt.show()
```



13.4 Tree-Based Outlier Detection: Isolation Forests

Isolation Forests provide a tree-based approach to anomaly detection, making them highly efficient for large and high-dimensional datasets. This method partitions data randomly to isolate anomalies based on the principle that outliers are "few and different" and therefore should be easier to isolate.

Key Features:

- Uses multiple decision trees to calculate anomaly scores
- Has linear time complexity, making it scalable for large datasets
- Does not require assumptions about feature distributions
- Works best with large datasets but performs poorly on small datasets
- Can detect anomalies without prior knowledge but does not explain why a point is anomalous

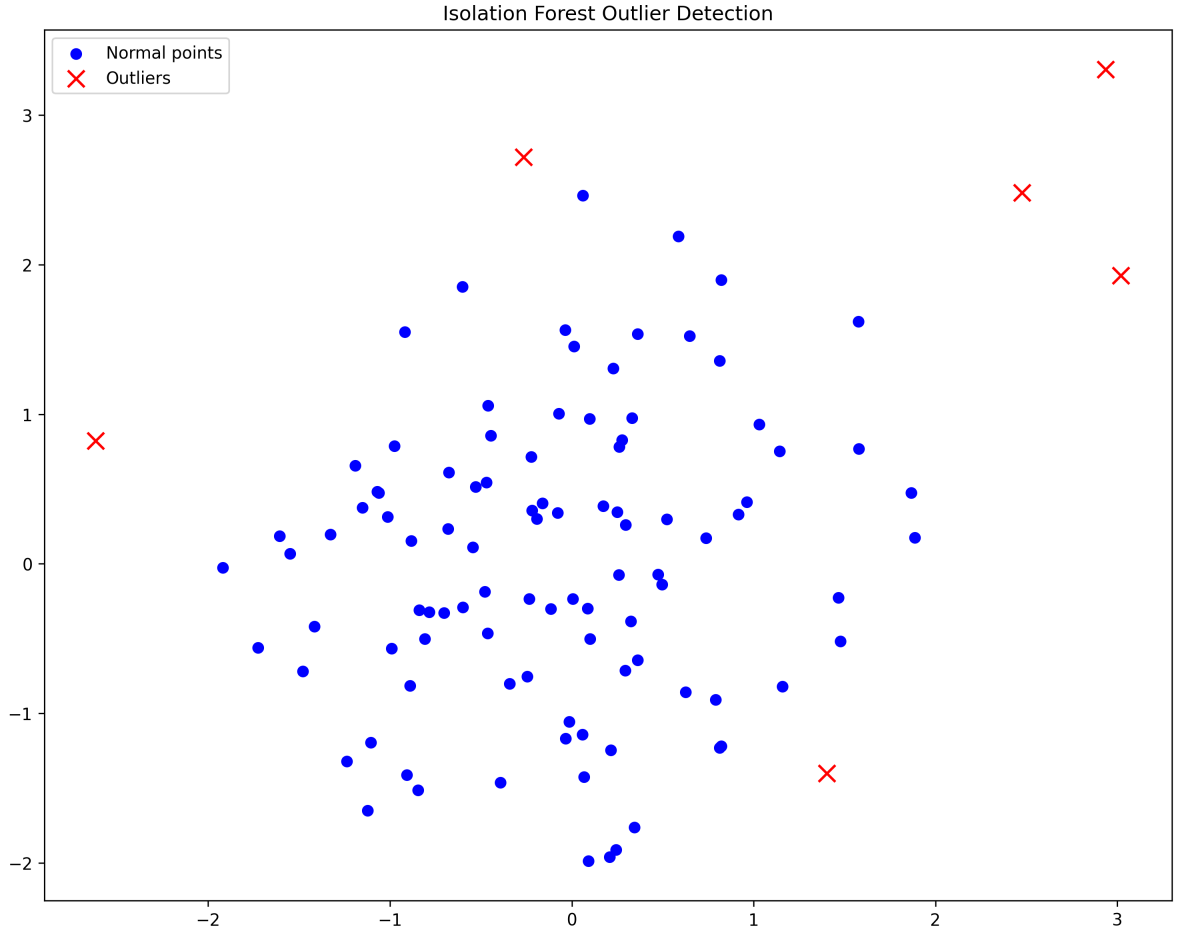
Steps of Isolation Forest Algorithm:

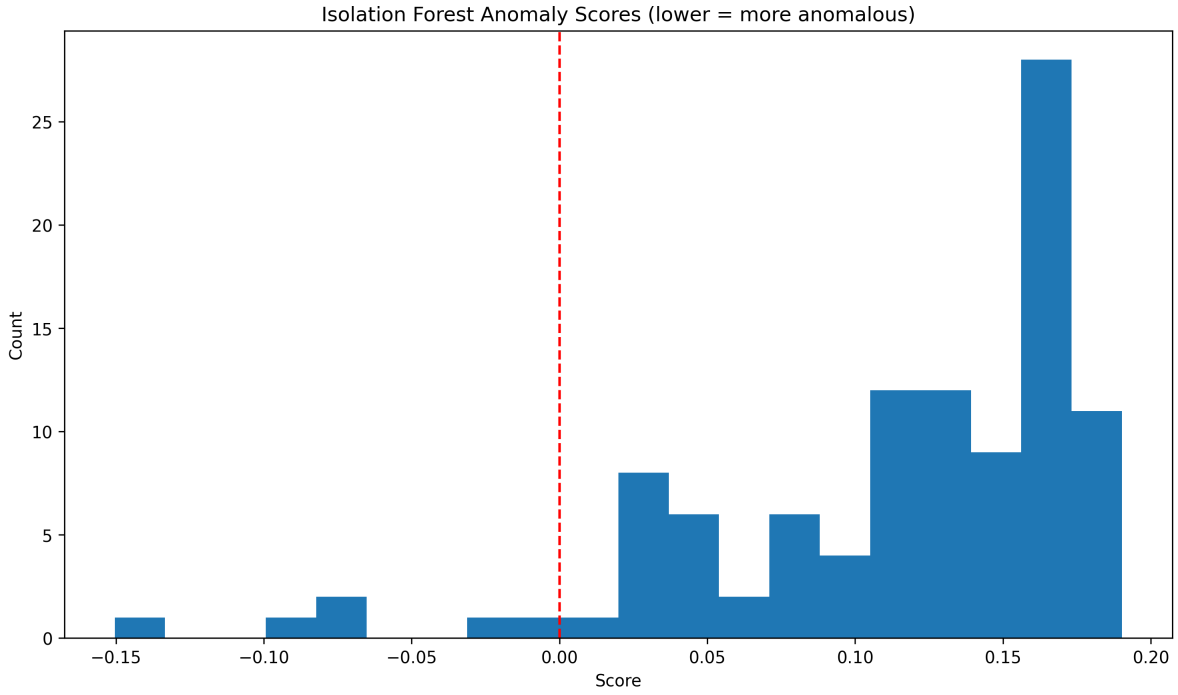
1. Randomly select a feature
2. Randomly choose a split value within the feature's range
3. Partition the data into two child nodes
4. Recursively repeat the process until:
    - Each leaf node has only one instance
    - A predefined maximum depth is reached

The anomaly score is calculated based on the path length to isolate a point. Outliers typically have shorter path lengths.

```
from sklearn.ensemble import IsolationForest

# Generate sample data with outliers
np.random.seed(42)
X_normal = np.random.normal(0, 1, (100, 2))
X_outliers = np.random.uniform(-4, 4, (5, 2))
X = np.vstack([X_normal, X_outliers])

# Apply Isolation Forest
iso_forest = IsolationForest(contamination=0.05, random_state=42)
predictions = iso_forest.fit_predict(X)
outlier_mask = predictions == -1  # -1 for outliers, 1 for inliers

# Visualize results
plt.figure(figsize=(10, 8))
plt.scatter(X[~outlier_mask, 0], X[~outlier_mask, 1], c='blue', label='Normal points')
plt.scatter(X[outlier_mask, 0], X[outlier_mask, 1], c='red', marker='x', s=100, label='Outli
plt.title('Isolation Forest Outlier Detection')
plt.legend()
plt.tight_layout()
plt.show()

# Show anomaly scores
anomaly_scores = iso_forest.decision_function(X)
plt.figure(figsize=(10, 6))
plt.hist(anomaly_scores, bins=20)
plt.axvline(x=0, color='r', linestyle='--')
plt.title('Isolation Forest Anomaly Scores (lower = more anomalous)')
plt.xlabel('Score')
plt.ylabel('Count')
```

```
plt.tight_layout()
plt.show()
```





13.5 Challenges in Unsupervised Outlier Detection

While unsupervised methods are powerful, they come with challenges:

- **False positives**: Legitimate data points may be flagged as outliers
- **Domain-specific outliers**: What constitutes an outlier varies by domain
- **Parameter sensitivity**: Results depend on parameter choices (k in KNN, contamination in Isolation Forest)
- **Dismissal of true anomalies**: A notable example is the delayed discovery of the ozone hole, which remained undetected for years because the anomaly was disregarded by automated systems

Striking a balance between reporting genuine outliers and avoiding excessive false positives is crucial in data-driven decision-making.

14 Recommender Systems

Recommender systems play a crucial role in online retail, content platforms, and various digital services by helping businesses suggest relevant products to customers. By analyzing user behavior, purchase history, and product similarities, recommendation algorithms improve user experience and increase sales.

Business Impact of Recommender Systems

- 35% of Amazon's revenue comes from recommendations
- 75% of Netflix views are driven by recommendations
- Spotify's Discover Weekly has a 55% click-through rate

14.1 Recommendation Scenarios

Recommender systems operate in different contexts:

- **Item-based recommendation**: Suggest items similar to a given item (e.g., Amazon's "Customers who bought this also bought")
- **User-based recommendation**: Suggest items to a user based on their past behavior (e.g., Netflix homepage)
- **Hybrid recommendation**: Combines both item-based and user-based approaches for personalized recommendations

A key challenge is that users rate only a small fraction of available items, leading to a sparse user-item matrix. The system must predict missing ratings to provide effective recommendations.

14.2 Types of Recommender Systems

14.2.1 1. Content-Based Filtering

Content-based filtering recommends items similar to those a user has liked in the past based on item features:

- **Assumptions**: Access to side information about items (e.g., genre, keywords, descriptions)
- **Approach**: Uses supervised learning to extract item and user features, then builds a model to predict ratings
- **Advantages**: Can make recommendations for new users/items without requiring previous interactions
- **Real-world examples**:
  - Pandora (music recommendations based on song attributes)
  - Gmail's important messages (predicting which emails are important based on content)

14.2.2 2. Collaborative Filtering

Collaborative filtering recommends items based on similarity patterns between users and/or items:

- **Assumptions**: Does not require side information about items
- **Core idea**: Personal tastes are correlated. If Alice and Bob both like X, and Alice likes Y, then Bob is more likely to like Y
- **Approach**: Uses an unsupervised learning approach. Have labels (ratings) but no explicit feature vectors
- **Limitations**: Struggles with the cold start problem (poor predictions for new users or items)

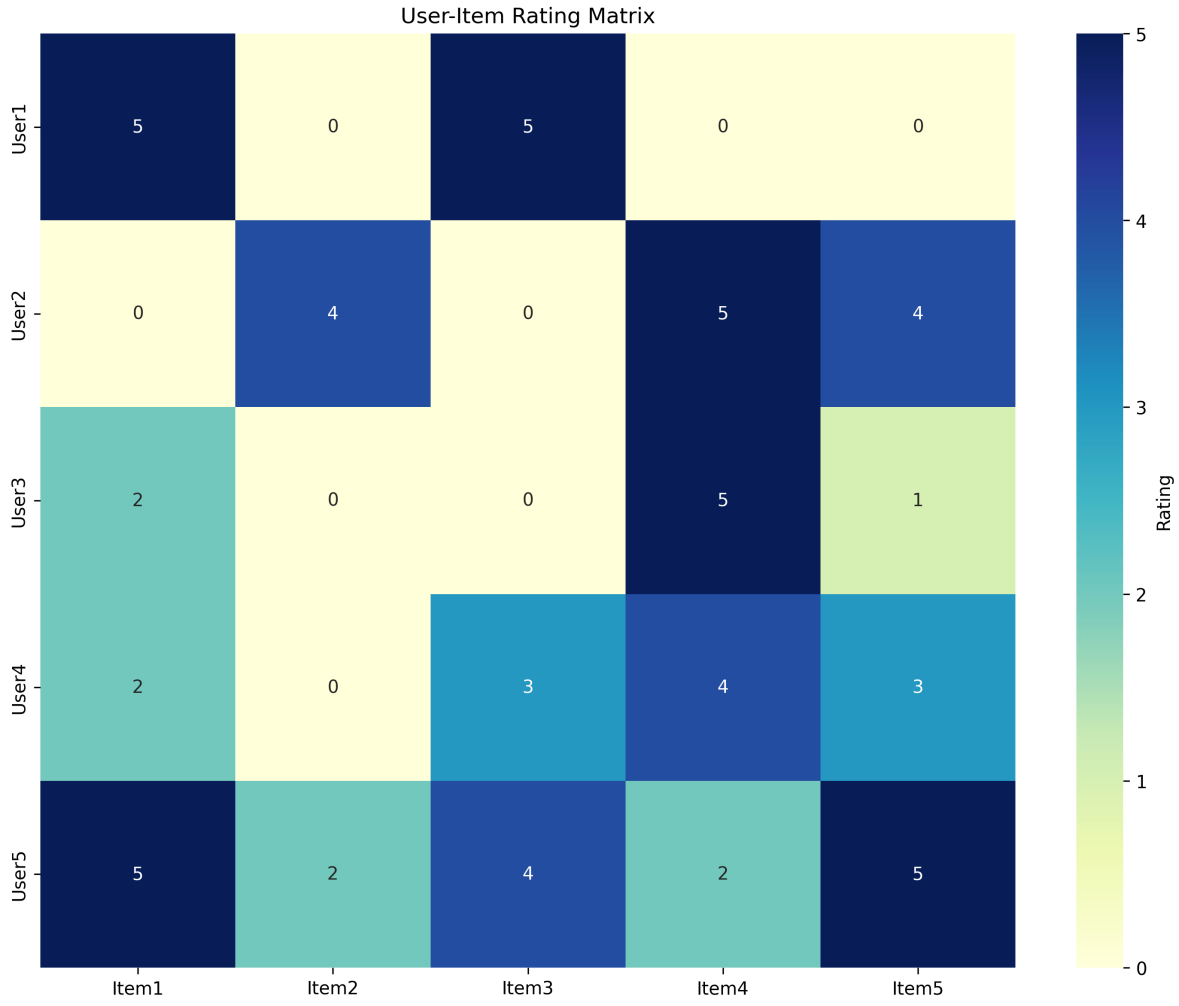
14.3 User-Product Matrix

The user-product matrix represents users as rows and products as columns, with entries indicating purchases or ratings. This matrix is the foundation of many recommendation algorithms.

```
# Create a sample user-item matrix
users = ['User1', 'User2', 'User3', 'User4', 'User5']
items = ['Item1', 'Item2', 'Item3', 'Item4', 'Item5']
np.random.seed(42)
ratings = np.zeros((len(users), len(items)))
# Fill with some ratings (1-5), 0 means no rating
for i in range(len(users)):
    for j in range(len(items)):
        if np.random.random() > 0.3:  # 70% chance of having a rating
            ratings[i, j] = np.random.randint(1, 6)
```

```
# Create a DataFrame for better visualization
ratings_df = pd.DataFrame(ratings, index=users, columns=items)
print("User-Item Rating Matrix:")
print(ratings_df)

# Visualize the matrix
plt.figure(figsize=(10, 8))
sns.heatmap(ratings_df, annot=True, cmap='YlGnBu', cbar_kws={'label': 'Rating'})
plt.title('User-Item Rating Matrix')
plt.tight_layout()
plt.show()
```

```
User-Item Rating Matrix:
       Item1  Item2  Item3  Item4  Item5
User1    5.0    0.0    5.0    0.0    0.0
User2    0.0    4.0    0.0    5.0    4.0
User3    2.0    0.0    0.0    5.0    1.0
User4    2.0    0.0    3.0    4.0    3.0
User5    5.0    2.0    4.0    2.0    5.0
```



14.4 Collaborative Filtering Methods

14.4.1 1. Neighborhood Methods

Neighborhood methods find users or items with similar preferences:

- **User-based**: If a group of users liked the same set of movies, recommend those movies to others in the group
- **Item-based**: If two items have similar rating patterns, recommend one to users who liked the other

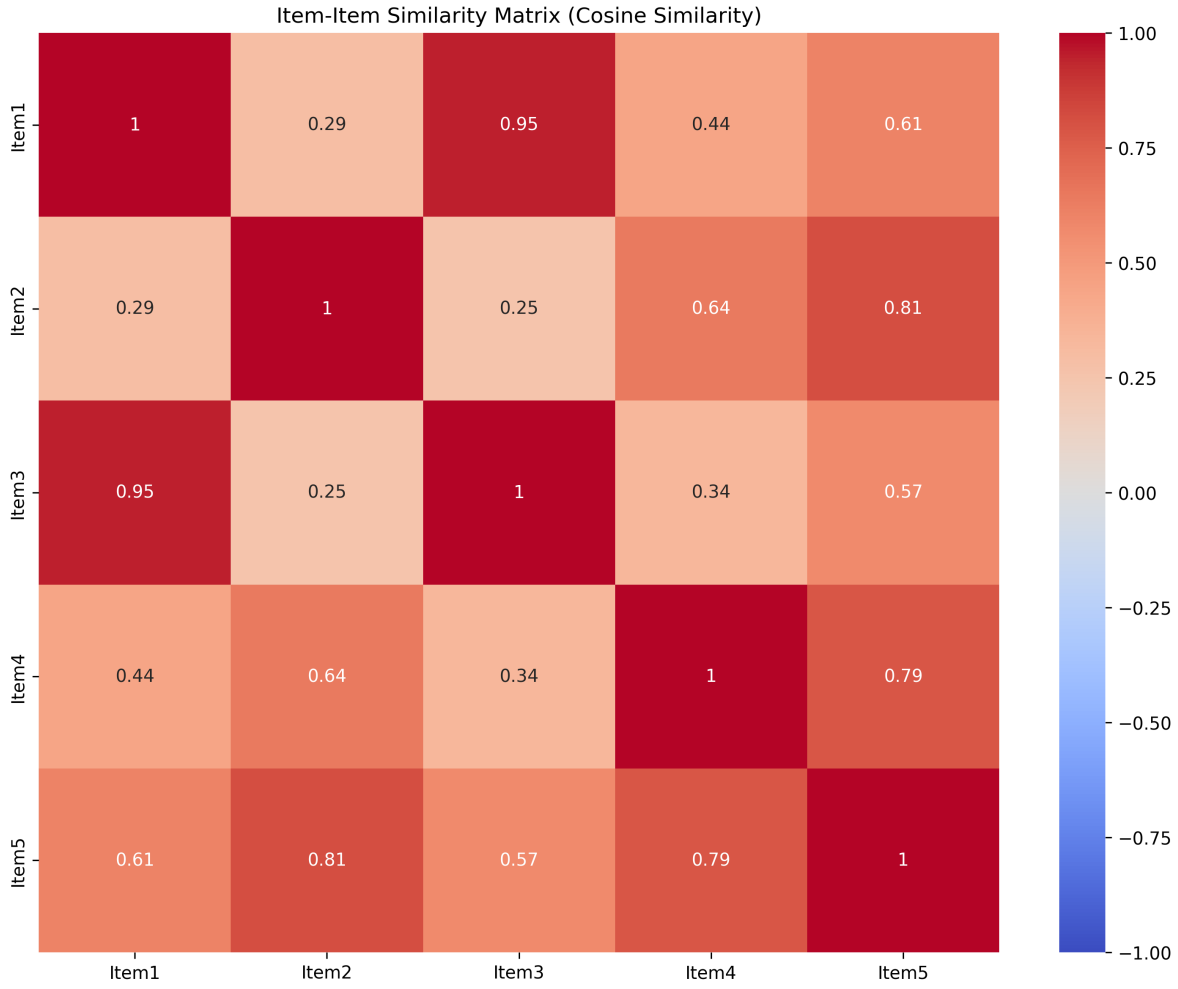
Algorithm: 1. Identify similar users/movies based on rating patterns 2. Recommend movies watched by similar users

Amazon's Product Recommendation Method uses nearest neighbor (KNN) searches across product columns to determine similarity. The goal is to find products that minimize the difference between them:

- Normalize each column by dividing by its norm: $\hat{X}_j = \frac{X_j}{\|X_j\|}$
- This ensures that recommendations reflect the relative popularity of a product rather than absolute purchase counts
- Products bought by similar users are considered more alike

```
from sklearn.metrics.pairwise import cosine_similarity

# Compute item-item similarity matrix
item_similarity = cosine_similarity(ratings_df.T)
item_sim_df = pd.DataFrame(item_similarity, index=items, columns=items)
print("Item-Item Similarity Matrix:")
print(item_sim_df)

# Visualize item similarity
plt.figure(figsize=(10, 8))
sns.heatmap(item_sim_df, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Item-Item Similarity Matrix (Cosine Similarity)')
plt.tight_layout()
plt.show()

# Function to get top N similar items
def get_similar_items(item_name, item_sim_df, n=2):
    similar_items = item_sim_df[item_name].sort_values(ascending=False)
    # Exclude the item itself
    similar_items = similar_items.drop(item_name)
    return similar_items.head(n)

# Example: Get items similar to Item1
similar_to_item1 = get_similar_items('Item1', item_sim_df)
print("\nItems similar to Item1:")
print(similar_to_item1)
```

```
Item-Item Similarity Matrix:
          Item1     Item2     Item3     Item4     Item5
Item1  1.000000  0.293610  0.947046  0.439435  0.606757
Item2  0.293610  1.000000  0.252982  0.641427  0.814092
Item3  0.947046  0.252982  1.000000  0.338062  0.574286
Item4  0.439435  0.641427  0.338062  1.000000  0.786618
Item5  0.606757  0.814092  0.574286  0.786618  1.000000
```




```
Items similar to Item1:
Item3    0.947046
Item5    0.606757
Name: Item1, dtype: float64
```

14.4.2 2. Latent Factor Methods

Instead of looking at raw ratings, latent factor models assume that both users and movies exist in a lower-dimensional feature space representing hidden properties.

- Each movie and user is mapped to a vector in this space
- Recommendations are made based on proximity in this latent space

Example: A user interested in action movies might have a high latent factor score for "intensity," leading to recommendations for high-action films.

14.5 Matrix Factorization (MF)

Matrix Factorization is a powerful approach to collaborative filtering, decomposing the user-item matrix into lower-dimensional factors:

- Defines a model with an objective function
- Optimized using stochastic gradient descent

Types of Matrix Factorization: - Unconstrained Matrix Factorization - Singular Value Decomposition (SVD) - Non-negative Matrix Factorization (NMF)

Mathematical Formulation: For a user-item matrix R with users u and items i, matrix factorization finds matrices P and Q such that:

$R \approx P \times Q^T$

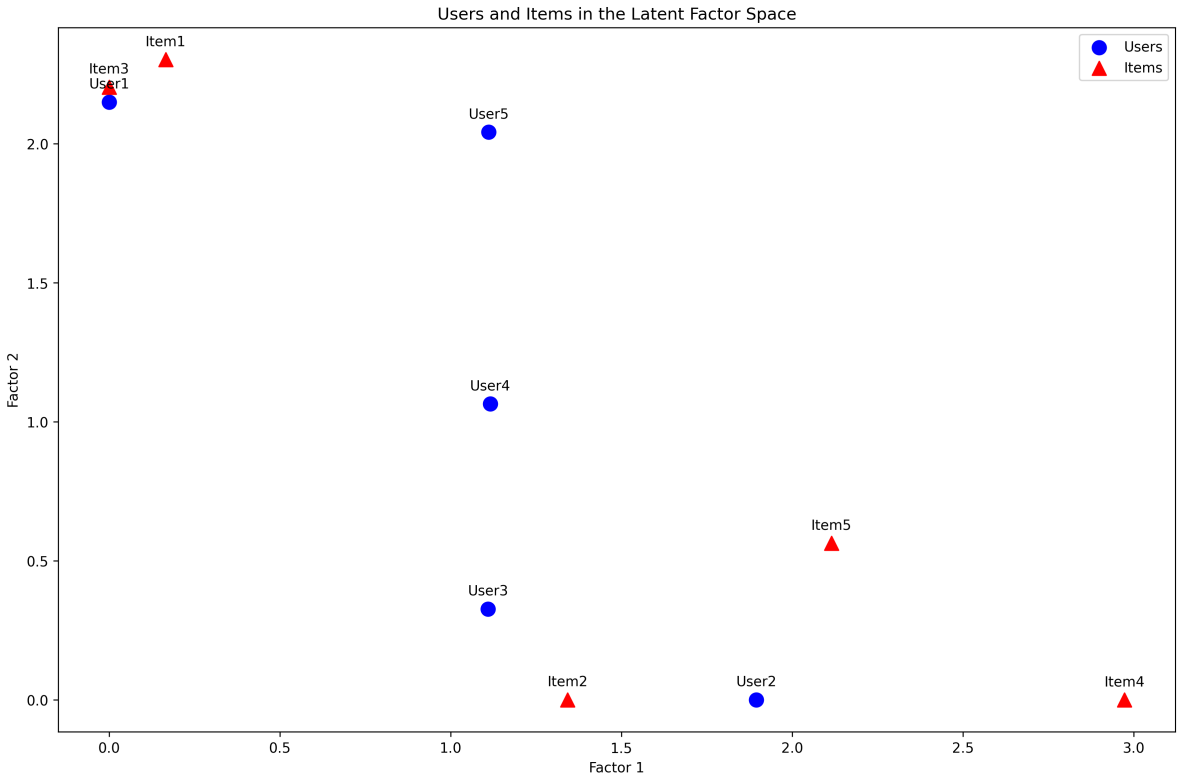
Where P represents user vectors and Q represents item vectors in the latent space.

```
from sklearn.decomposition import NMF

# Fill missing values with zeros for demonstration
# In practice, you might want to use mean imputation or more sophisticated methods
ratings_matrix = ratings_df.values

# Non-negative Matrix Factorization
n_components = 2  # Number of latent factors
model = NMF(n_components=n_components, init='random', random_state=0)
user_features = model.fit_transform(ratings_matrix)
item_features = model.components_

# Display latent factors
print("User Latent Factors:")
user_factors_df = pd.DataFrame(user_features, index=users,
                               columns=[f'Factor {i+1}' for i in range(n_components)])
print(user_factors_df)

print("\nItem Latent Factors:")
item_factors_df = pd.DataFrame(item_features.T, index=items,
                               columns=[f'Factor {i+1}' for i in range(n_components)])
print(item_factors_df)
```

```
# Visualize user and item factors in the latent space
plt.figure(figsize=(12, 8))
plt.scatter(user_features[:, 0], user_features[:, 1], c='blue', marker='o', s=100, label='Use
plt.scatter(item_features.T[:, 0], item_features.T[:, 1], c='red', marker='^', s=100, label=

# Add labels
for i, user in enumerate(users):
    plt.annotate(user, (user_features[i, 0], user_features[i, 1]), textcoords="offset points'
                 xytext=(0,10), ha='center')
for i, item in enumerate(items):
    plt.annotate(item, (item_features.T[i, 0], item_features.T[i, 1]), textcoords="offset poi
                 xytext=(0,10), ha='center')

plt.title('Users and Items in the Latent Factor Space')
plt.xlabel('Factor 1')
plt.ylabel('Factor 2')
plt.legend()
plt.tight_layout()
plt.show()

# Reconstruct the ratings matrix and compute the predicted ratings
reconstructed_ratings = np.dot(user_features, item_features)
predicted_ratings_df = pd.DataFrame(reconstructed_ratings, index=users, columns=items)

print("\nPredicted Ratings:")
print(predicted_ratings_df.round(1))
```

```
User Latent Factors:
       Factor 1  Factor 2
User1  0.000000  2.150735
User2  1.895195  0.000000
User3  1.109400  0.326875
User4  1.115771  1.064441
User5  1.110953  2.042516

Item Latent Factors:
       Factor 1  Factor 2
Item1  0.165554  2.303040
Item2  1.342522  0.000000
Item3  0.000000  2.203462
Item4  2.973004  0.000000
```

```
Item5  2.114626  0.563500
```




```
Predicted Ratings:
       Item1  Item2  Item3  Item4  Item5
User1    5.0    0.0    4.7    0.0    1.2
User2    0.3    2.5    0.0    5.6    4.0
User3    0.9    1.5    0.7    3.3    2.5
User4    2.6    1.5    2.3    3.3    3.0
User5    4.9    1.5    4.5    3.3    3.5
```

14.6 Computational Challenges

Finding KNNs in a dataset with n users and d products has a computational cost of O(nd), which becomes infeasible at scale. However, optimizations include:

- Leveraging sparse matrices to reduce complexity
- Using approximate nearest neighbor search to speed up calculations
- Applying clustering techniques to limit the search space

14.7 Beyond Accuracy in Recommender Systems

While accuracy is crucial, other factors influence a recommender system's effectiveness:

- **Diversity**: How different are the recommendations? (Avoid showing only similar items)
- **Serendipity**: How surprising and useful are the recommendations?
- **Persistence**: How long should recommendations stay relevant?
- **Trust**: Providing explanations for recommendations increases user trust
  - Example: Quora explains why certain answers are recommended
- **Social Recommendation**: What did your friends watch or buy?
- **Freshness**: Users often prefer recent and surprising recommendations

Recommender systems continue to evolve, incorporating hybrid models, deep learning, and reinforcement learning to enhance personalization and engagement.

15 Class Imbalance in Machine Learning

Class imbalance occurs when one class in a dataset has significantly more samples than another. This imbalance can impact the performance of machine learning models, particularly classification algorithms.

15.1 Categorization of Class Imbalance

A class imbalance problem arises when the classes in a dataset are not equally represented. Common examples include:

- Fraud detection (few fraudulent transactions among many legitimate ones)
- Medical diagnosis (rare diseases)
- Network intrusion detection (few attacks among normal traffic)

The **imbalance ratio** is calculated as: Imbalance Ratio $= \frac{\text{Number of Majority Class Samples}}{\text{Number of Minority Class Samples}}$

A high imbalance ratio indicates a severely skewed dataset.

15.2 Sampling Techniques

Sampling is a statistical process where a predetermined number of observations are taken from a larger population. It helps adjust the class distribution in a dataset to improve model performance.

15.2.1 Oversampling

Oversampling increases the number of instances in the minority class. Two sophisticated techniques include:

15.2.1.1 Synthetic Minority Oversampling Technique (SMOTE)

SMOTE generates synthetic examples for the minority class by interpolating existing instances:

1. Identifies the k-nearest neighbors of a minority class instance
2. Randomly selects one of the k-nearest neighbors
3. Generates a new synthetic instance along the line segment connecting the two points

SMOTE avoids overfitting and helps balance datasets while maintaining diversity.

15.2.1.2 ADASYN (Adaptive Synthetic Sampling)

ADASYN extends SMOTE by focusing on difficult-to-classify instances:

1. Calculates the ratio of majority class instances in the k-nearest neighbors of each minority instance
2. Generates synthetic samples in proportion to this ratio
3. Adjusts the decision boundary to improve classification performance

15.2.2 Undersampling

Reduces the number of instances in the majority class, either randomly or using techniques like:

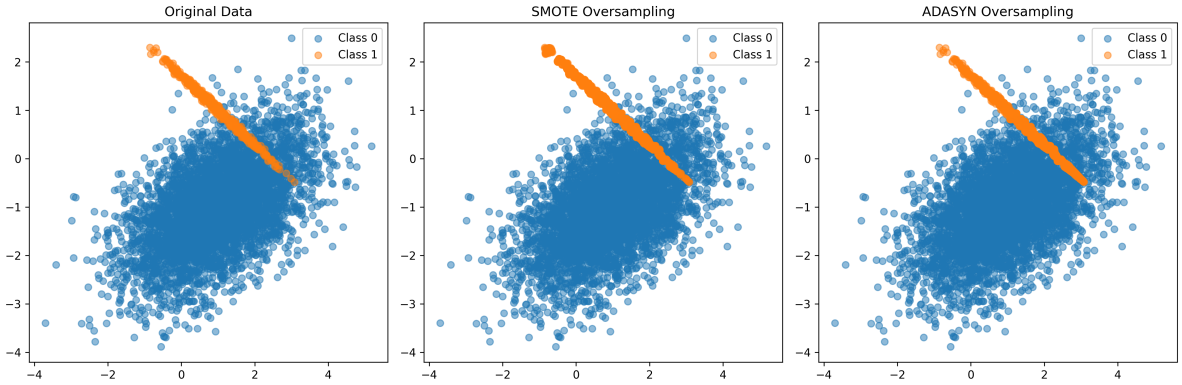
- Cluster-based undersampling
- Tomek links removal
- Near-miss algorithm

```
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE, ADASYN
from collections import Counter

# Generate imbalanced dataset
X, y = make_classification(n_samples=5000, n_features=2, n_informative=2,
                           n_redundant=0, n_repeated=0, n_classes=2,
                           n_clusters_per_class=1,
                           weights=[0.9, 0.1], flip_y=0, random_state=42)

# Check class distribution
print("Original class distribution:", Counter(y))
```

```
# Calculate imbalance ratio
imbalance_ratio = sum(y == 0) / sum(y == 1)
print(f"Imbalance ratio: {imbalance_ratio:.2f}")

# Apply SMOTE
smote = SMOTE(random_state=42)
X_smote, y_smote = smote.fit_resample(X, y)
print("SMOTE class distribution:", Counter(y_smote))

# Apply ADASYN
adasyn = ADASYN(random_state=42)
X_adasyn, y_adasyn = adasyn.fit_resample(X, y)
print("ADASYN class distribution:", Counter(y_adasyn))

# Visualize original and resampled data
plt.figure(figsize=(15, 5))

# Original data
plt.subplot(1, 3, 1)
plt.scatter(X[y == 0, 0], X[y == 0, 1], label='Class 0', alpha=0.5)
plt.scatter(X[y == 1, 0], X[y == 1, 1], label='Class 1', alpha=0.5)
plt.title('Original Data')
plt.legend()

# SMOTE
plt.subplot(1, 3, 2)
plt.scatter(X_smote[y_smote == 0, 0], X_smote[y_smote == 0, 1], label='Class 0', alpha=0.5)
plt.scatter(X_smote[y_smote == 1, 0], X_smote[y_smote == 1, 1], label='Class 1', alpha=0.5)
plt.title('SMOTE Oversampling')
plt.legend()

# ADASYN
plt.subplot(1, 3, 3)
plt.scatter(X_adasyn[y_adasyn == 0, 0], X_adasyn[y_adasyn == 0, 1], label='Class 0', alpha=0
plt.scatter(X_adasyn[y_adasyn == 1, 0], X_adasyn[y_adasyn == 1, 1], label='Class 1', alpha=0
plt.title('ADASYN Oversampling')
plt.legend()

plt.tight_layout()
plt.show()
```

```
Original class distribution: Counter({np.int64(0): 4500, np.int64(1): 500})
```

```
Imbalance ratio: 9.00
SMOTE class distribution: Counter({np.int64(0): 4500, np.int64(1): 4500})
ADASYN class distribution: Counter({np.int64(1): 4519, np.int64(0): 4500})
```



15.3 Comparison: SMOTE vs. ADASYN

- **SMOTE** generates synthetic samples uniformly, without distinguishing between easy and hard-to-classify instances
- **ADASYN** focuses more on samples near decision boundaries, enhancing model performance for difficult cases

15.3.1 Disadvantages of Oversampling

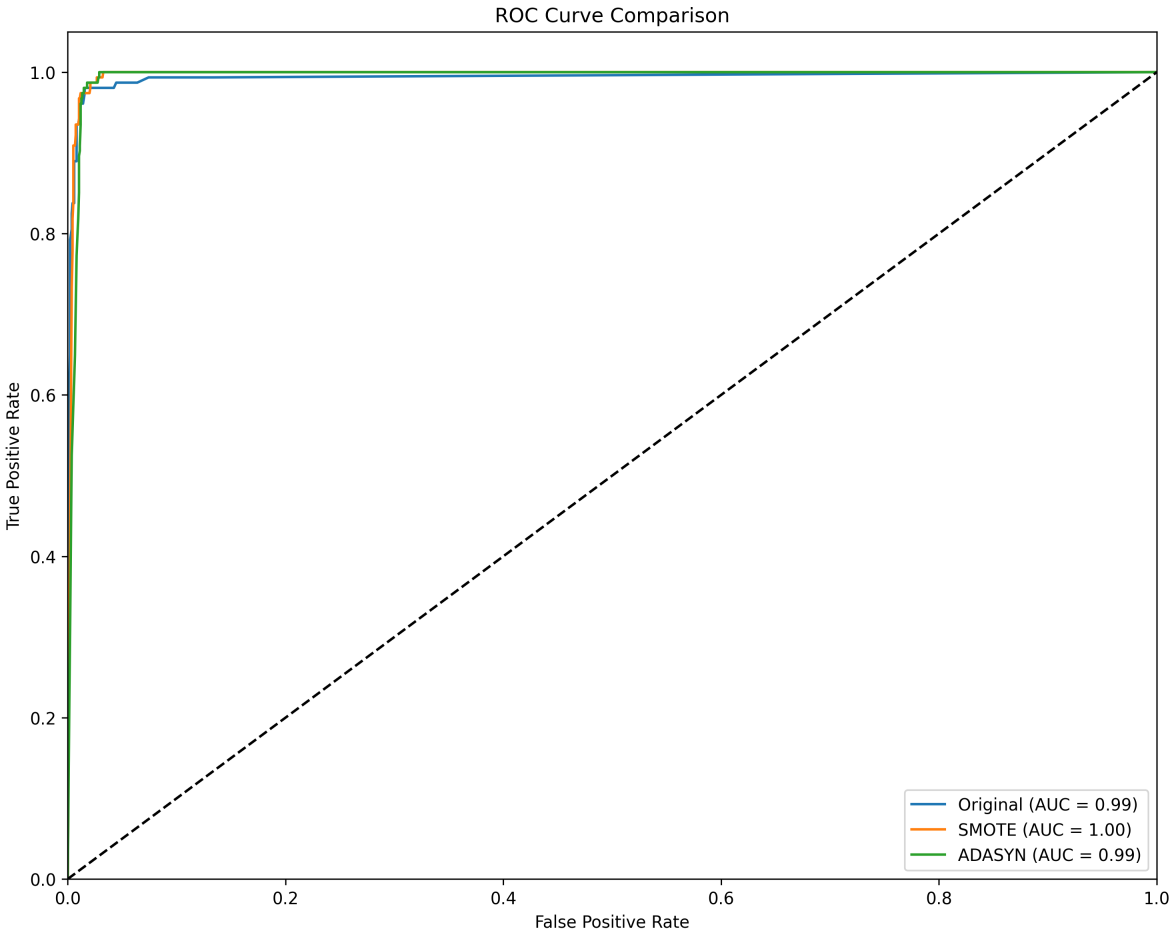
- Assumes that the space between any two minority class samples belongs to the minority class, which may not be true for non-linearly separable data
- Can introduce noise if not carefully applied
- May exacerbate the problem of overlapping class distributions

```python
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc

# Split the original data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train and evaluate model on original data
clf_orig = RandomForestClassifier(random_state=42)
clf_orig.fit(X_train, y_train)
```

```
y_pred_orig = clf_orig.predict(X_test)
print("Original data results:")
print(classification_report(y_test, y_pred_orig))

# Train and evaluate model on SMOTE-resampled data
X_train_smote, y_train_smote = SMOTE(random_state=42).fit_resample(X_train, y_train)
clf_smote = RandomForestClassifier(random_state=42)
clf_smote.fit(X_train_smote, y_train_smote)
y_pred_smote = clf_smote.predict(X_test)
print("\nSMOTE results:")
print(classification_report(y_test, y_pred_smote))

# Train and evaluate model on ADASYN-resampled data
X_train_adasyn, y_train_adasyn = ADASYN(random_state=42).fit_resample(X_train, y_train)
clf_adasyn = RandomForestClassifier(random_state=42)
clf_adasyn.fit(X_train_adasyn, y_train_adasyn)
y_pred_adasyn = clf_adasyn.predict(X_test)
print("\nADASYN results:")
print(classification_report(y_test, y_pred_adasyn))

# ROC curves
plt.figure(figsize=(10, 8))

# Original data
y_scores_orig = clf_orig.predict_proba(X_test)[:, 1]
fpr_orig, tpr_orig, _ = roc_curve(y_test, y_scores_orig)
roc_auc_orig = auc(fpr_orig, tpr_orig)
plt.plot(fpr_orig, tpr_orig, label=f'Original (AUC = {roc_auc_orig:.2f})')

# SMOTE
y_scores_smote = clf_smote.predict_proba(X_test)[:, 1]
fpr_smote, tpr_smote, _ = roc_curve(y_test, y_scores_smote)
roc_auc_smote = auc(fpr_smote, tpr_smote)
plt.plot(fpr_smote, tpr_smote, label=f'SMOTE (AUC = {roc_auc_smote:.2f})')

# ADASYN
y_scores_adasyn = clf_adasyn.predict_proba(X_test)[:, 1]
fpr_adasyn, tpr_adasyn, _ = roc_curve(y_test, y_scores_adasyn)
roc_auc_adasyn = auc(fpr_adasyn, tpr_adasyn)
plt.plot(fpr_adasyn, tpr_adasyn, label=f'ADASYN (AUC = {roc_auc_adasyn:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
```

```
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend(loc="lower right")
plt.tight_layout()
plt.show()
```

```
Original data results:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1346
           1       0.93      0.94      0.93       154

    accuracy                           0.99      1500
   macro avg       0.96      0.96      0.96      1500
weighted avg       0.99      0.99      0.99      1500


SMOTE results:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99      1346
           1       0.87      0.97      0.92       154

    accuracy                           0.98      1500
   macro avg       0.93      0.98      0.95      1500
weighted avg       0.98      0.98      0.98      1500


ADASYN results:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99      1346
           1       0.84      0.99      0.91       154

    accuracy                           0.98      1500
   macro avg       0.92      0.98      0.95      1500
weighted avg       0.98      0.98      0.98      1500
```



15.4 Evaluation of Classifiers with Imbalanced Data

When evaluating classifiers on imbalanced datasets, standard accuracy can be misleading. More appropriate metrics include:

- **Precision & Recall**: Measures how well the model identifies the minority class
- **F1-score**: Harmonic mean of precision and recall
- **ROC-AUC**: Evaluates the ability to distinguish between classes across thresholds
- **Precision-Recall AUC**: Often more informative than ROC-AUC for imbalanced datasets
- **Geometric Mean**: Balance between sensitivity and specificity

:

16 Gradient Descent: Optimization in Machine Learning

Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize a cost function by iteratively adjusting model parameters. It's the engine that powers many machine learning algorithms, from simple linear regression to complex neural networks.

At its core, gradient descent is based on a simple yet powerful idea: to find the minimum of a function, we can follow the direction of the steepest descent. Mathematically, this direction is given by the negative gradient of the function.

16.1 The Intuition Behind Gradient Descent

Think of gradient descent as descending a mountain in foggy conditions where you can only see a small area around you. To reach the bottom (the minimum), you:

1. Feel the ground around you to determine the steepest downward direction (calculate the gradient)
2. Take a step in that direction (update parameters)
3. Repeat until you reach a point where all directions lead upward (convergence)

The cost function represents the "mountain landscape" in this analogy, with its valleys corresponding to low error and its peaks to high error.

16.2 Mathematical Foundation

For a function J( ), where represents our model parameters, the gradient J( ) points in the direction of the steepest increase. Therefore, to minimize J( ), we update as follows:

= - · J( )

Where: - is the learning rate (step size) - J( ) is the gradient of the cost function with respect to parameters

For a linear regression problem with mean squared error (MSE) cost function, this translates to:

J( ) = (1/m) · X^T · (X · - y)

Where: - m is the number of training examples - X is the feature matrix - y is the target vector - is the parameter vector

16.3 How Gradient Descent Works

Gradient Descent operates through a systematic process:

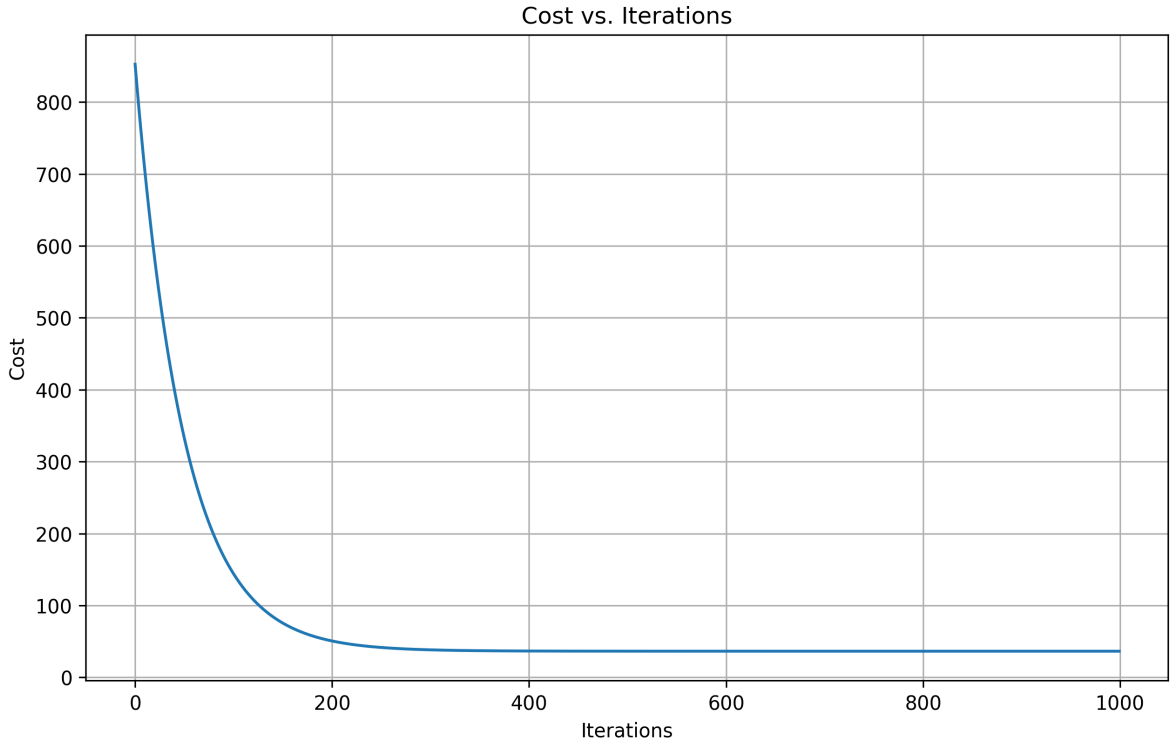
1. **Random Initialization**: Begin with arbitrary parameter values. This represents our starting point on the "mountain."
2. **Calculate the Gradient**: Compute the gradient (slope) of the cost function with respect to each parameter. This tells us which direction leads downhill most steeply.
3. **Update Parameters**: Adjust parameters in the opposite direction of the gradient. The size of this adjustment is controlled by the learning rate.
4. **Repeat Until Convergence**: Continue until the gradient approaches zero, indicating a minimum.

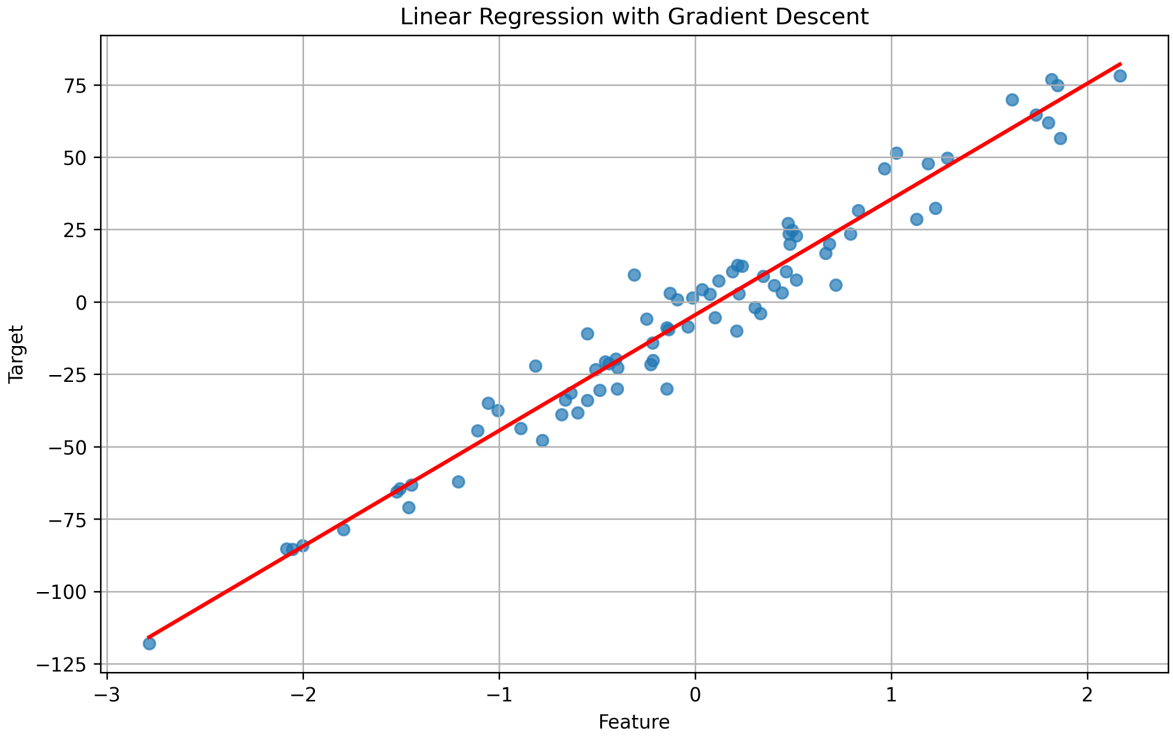
The algorithm's success hinges on the learning rate, which controls the step size in each iteration.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Generate a simple regression dataset
X, y = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)
X = StandardScaler().fit_transform(X)
y = y.reshape(-1, 1)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Implement basic gradient descent for linear regression
def gradient_descent(X, y, learning_rate=0.01, iterations=1000):
    # Initialize parameters (weights and bias)
    m = X.shape[0]
    theta = np.zeros((2, 1))  # [w, b]
    X_b = np.c_[np.ones((m, 1)), X]  # Add intercept term
```

```
    # To store cost history
    cost_history = []

    for i in range(iterations):
        # Calculate predictions
        predictions = X_b.dot(theta)

        # Calculate error
        error = predictions - y

        # Calculate gradients
        gradients = (1/m) * X_b.T.dot(error)

        # Update parameters
        theta = theta - learning_rate * gradients

        # Calculate cost (MSE)
        cost = (1/(2*m)) * np.sum(np.square(error))
        cost_history.append(cost)

    return theta, cost_history

# Run gradient descent
theta, cost_history = gradient_descent(X_train, y_train)

# Visualize cost over iterations
plt.figure(figsize=(10, 6))
plt.plot(cost_history)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Cost vs. Iterations')
plt.grid(True)
plt.show()

# Visualize the regression line
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, alpha=0.7)
X_new = np.array([[np.min(X_train)], [np.max(X_train)]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta)
plt.plot(X_new, y_predict, "r-", linewidth=2)
plt.xlabel('Feature')
```

```
plt.ylabel('Target')
plt.title('Linear Regression with Gradient Descent')
plt.grid(True)
plt.show()

print(f"Optimized parameters: w = {theta[1][0]:.4f}, b = {theta[0][0]:.4f}")
```






```
Optimized parameters: w = 39.9781, b = -4.4957
```

16.4 The Learning Rate: A Delicate Balance

The learning rate ( ) is a hyperparameter that determines how large each step is during optimization. Its selection involves a fundamental trade-off:

- **Too Small**: Convergence is extremely slow, requiring many iterations. This is computationally inefficient but less likely to miss the minimum.
- **Too Large**: The algorithm overshoots minima, potentially diverging instead of converging. This manifests as increasing cost values over iterations.

An optimal rate ensures steady progress toward the minimum without overshooting. In practice, finding the right learning rate often involves experimentation.

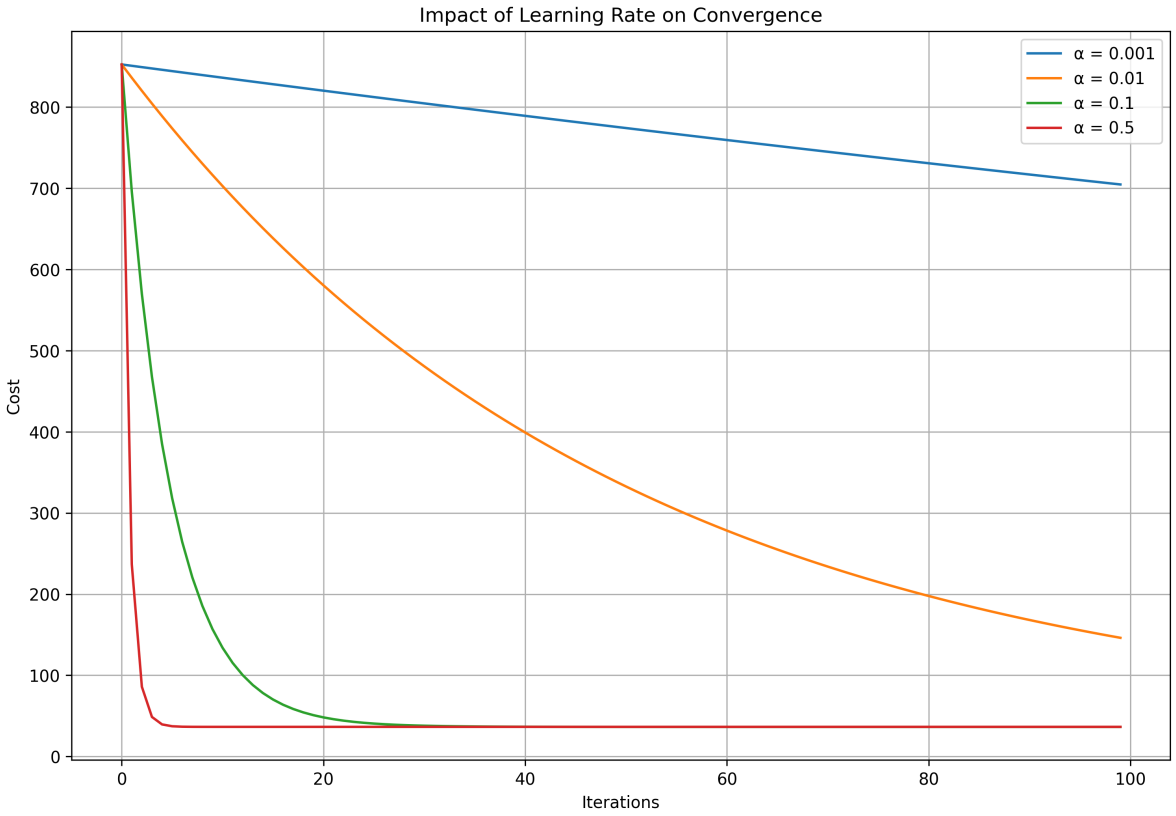
16.4.1 The Goldilocks Principle in Learning Rates

Think of learning rate selection as the "Goldilocks principle" – not too hot, not too cold, but just right:

1. **"Too cold" (small)**: The algorithm moves very cautiously, taking tiny steps. While it's unlikely to overshoot, it might take an impractically long time to reach the minimum.
2. **"Too hot" (large)**: The algorithm takes aggressive steps and might bounce around or completely miss the minimum, potentially even diverging (costs increase).
3. **"Just right" (optimal)**: The algorithm makes steady progress, converging to the minimum efficiently.

Learning rate scheduling techniques (discussed later) attempt to get the best of both worlds by starting with larger steps and gradually reducing them.

```
def compare_learning_rates(X, y, learning_rates=[0.001, 0.01, 0.1, 0.5], iterations=100):
    plt.figure(figsize=(12, 8))
    m = X.shape[0]
    X_b = np.c_[np.ones((m, 1)), X]

    for lr in learning_rates:
        theta = np.zeros((2, 1))
        cost_history = []

        for i in range(iterations):
            predictions = X_b.dot(theta)
            error = predictions - y
            gradients = (1/m) * X_b.T.dot(error)
            theta = theta - lr * gradients
            cost = (1/(2*m)) * np.sum(np.square(error))
            cost_history.append(cost)

        plt.plot(cost_history, label=f'  = {lr}')

    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.title('Impact of Learning Rate on Convergence')
    plt.legend()
    plt.grid(True)
    plt.show()

# Compare different learning rates
compare_learning_rates(X_train, y_train)
```



16.5 Challenges in Gradient Descent

Gradient Descent faces several challenges that arise from the nature of optimization landscapes:

16.5.1 1. Non-Convex Cost Functions

In many machine learning models, especially neural networks, the cost function isn't convex—meaning it has multiple local minima, saddle points, and plateaus. This creates several challenges:

- **Local Minima**: The algorithm might settle in a suboptimal local minimum rather than finding the global minimum. This is particularly problematic in deep learning.
- **Saddle Points**: Points where the gradient is zero in all directions, but it's neither a maximum nor a minimum. These can slow down convergence significantly.
- **Plateaus**: Flat regions in the cost function where the gradient is close to zero, slowing progress despite being far from a minimum.

16.5.2 2. The Ravine Problem

In some cost functions, the surface has elongated valley-like structures (ravines). In these cases, the gradient often points across the ravine rather than along it toward the minimum. This causes the algorithm to oscillate from one side to another, making slow progress.

16.5.3 3. Feature Scaling Issues

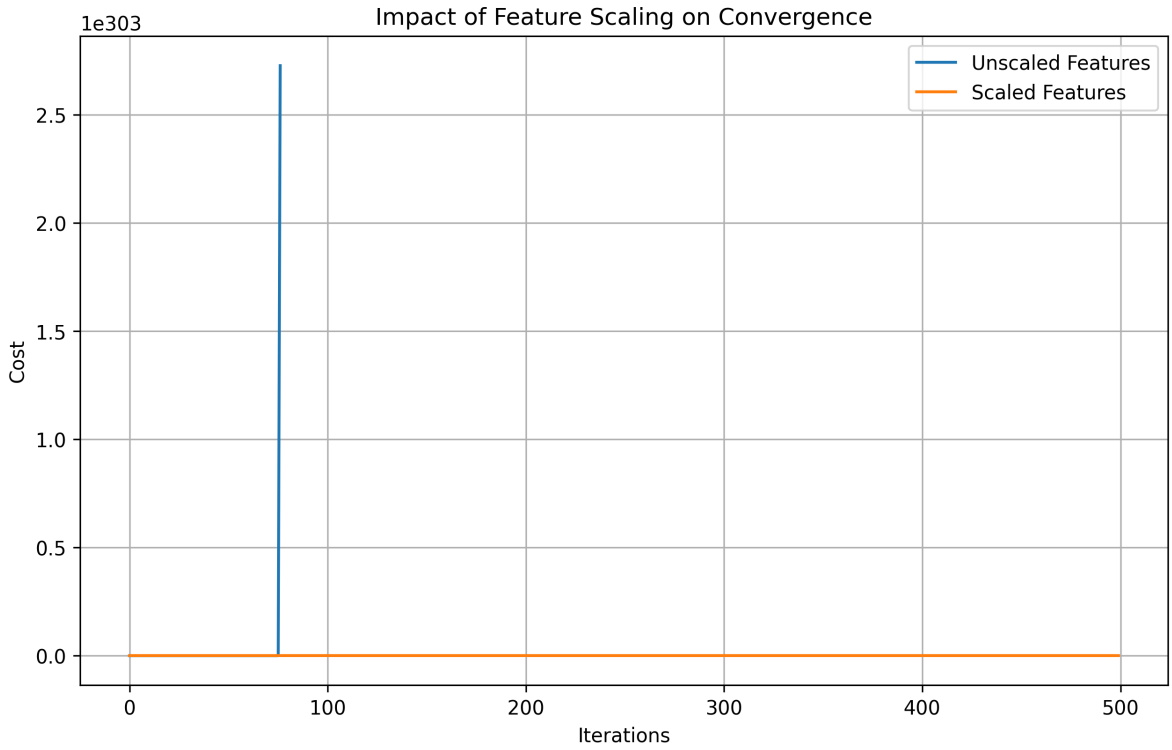
When features have different scales, the cost function contours become elongated ellipses rather than circles. This creates a similar ravine problem, where the gradient doesn't point directly toward the minimum.

To address this issue, feature scaling techniques like standardization are essential:

```
# Demonstrating the importance of feature scaling
X_multi, y_multi = make_regression(n_samples=100, n_features=2, noise=10, random_state=42)

# Scale one feature to be much larger
X_multi[:, 1] = X_multi[:, 1] * 100

# Split data
X_train_unscaled, X_test_unscaled, y_train_multi, y_test_multi = train_test_split(
    X_multi, y_multi.reshape(-1, 1), test_size=0.2, random_state=42
)

# Create scaled version
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_unscaled)

# Run gradient descent on both scaled and unscaled data
def run_gd_experiment(X, y, title, learning_rate=0.01, iterations=500):
    m = X.shape[0]
    n_features = X.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_b = np.c_[np.ones((m, 1)), X]
    cost_history = []

    for i in range(iterations):
        predictions = X_b.dot(theta)
        error = predictions - y
        gradients = (1/m) * X_b.T.dot(error)
        theta = theta - learning_rate * gradients
```

```
        cost = (1/(2*m)) * np.sum(np.square(error))
        cost_history.append(cost)

    return cost_history

# Compare convergence with and without scaling
cost_unscaled = run_gd_experiment(X_train_unscaled, y_train_multi, "Unscaled")
cost_scaled = run_gd_experiment(X_train_scaled, y_train_multi, "Scaled")

plt.figure(figsize=(10, 6))
plt.plot(cost_unscaled, label='Unscaled Features')
plt.plot(cost_scaled, label='Scaled Features')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Impact of Feature Scaling on Convergence')
plt.legend()
plt.grid(True)
plt.show()
```

```
C:\Users\roess\AppData\Local\Temp\ipykernel_17136\433023471.py:29: RuntimeWarning:

overflow encountered in square

C:\Users\roess\Documents\repos\Notes\myvenv312\Lib\site-packages\numpy\_core\fromnumeric.py:8

overflow encountered in reduce

C:\Users\roess\AppData\Local\Temp\ipykernel_17136\433023471.py:28: RuntimeWarning:

invalid value encountered in subtract
```



16.5.4 4. Vanishing and Exploding Gradients

In deep neural networks, gradients can either: - Become extremely small (vanish) as they're propagated backward through many layers - Become extremely large (explode) in certain network architectures

Both problems hamper effective training. Modern techniques like batch normalization, residual connections, and careful weight initialization help address these issues.

17 Types of Gradient Descent

There are three main variants of Gradient Descent, each with its own strengths and use cases. The primary difference is how much data they use to compute each parameter update.

17.1 1. Batch Gradient Descent

Batch Gradient Descent calculates the gradient using the entire dataset before updating parameters. This ensures consistent steps toward the minimum but can be computationally expensive for large datasets.

17.1.1 Conceptual Understanding

Think of Batch GD as carefully surveying the entire mountain before taking each step. This provides the most accurate direction but requires significant effort before making any progress.

17.1.2 Key Properties

- **Deterministic**: Always takes the same path for the same starting point
- **Memory-intensive**: Must process the entire dataset in memory
- **Smooth convergence**: Progress is steady with minimal fluctuations
- **Computationally expensive**: Especially for large datasets

```
def batch_gradient_descent(X, y, learning_rate=0.01, iterations=1000):
    m = X.shape[0]
    n_features = X.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_b = np.c_[np.ones((m, 1)), X]
    cost_history = []

    for i in range(iterations):
        # Use entire dataset for each update
        predictions = X_b.dot(theta)
```

```
        error = predictions - y
        gradients = (1/m) * X_b.T.dot(error)
        theta = theta - learning_rate * gradients
        cost = (1/(2*m)) * np.sum(np.square(error))
        cost_history.append(cost)

        # Early stopping condition (optional)
        if i > 0 and abs(cost_history[i] - cost_history[i-1]) < 1e-10:
            break

    return theta, cost_history
```

17.2 2. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent updates parameters using a single randomly selected training instance at each step. This makes it faster and more memory-efficient but introduces more randomness.

17.2.1 Conceptual Understanding

SGD is like taking quick steps based on whatever small part of the mountain you can immediately see. The direction might not be perfect, but you can take many more steps in the same amount of time.

17.2.2 Key Properties

- **Randomized**: Takes a different path each time
- **Memory-efficient**: Processes one example at a time
- **Noisy convergence**: Path includes fluctuations and bounces
- **Ability to escape local minima**: Randomness helps jump out of suboptimal valleys
- **Can work with streaming data**: Doesn't need the entire dataset at once

```
def stochastic_gradient_descent(X, y, learning_rate=0.01, epochs=10):
    m = X.shape[0]
    n_features = X.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_b = np.c_[np.ones((m, 1)), X]
    cost_history = []
```

```
    for epoch in range(epochs):
        # Shuffle data at the beginning of each epoch
        indices = np.random.permutation(m)
        X_shuffled = X_b[indices]
        y_shuffled = y[indices]

        for i in range(m):
            # Use a single instance for each update
            xi = X_shuffled[i:i+1]
            yi = y_shuffled[i:i+1]

            prediction = xi.dot(theta)
            error = prediction - yi
            gradients = xi.T.dot(error)

            # Learning rate scheduling (optional)
            lr = learning_rate / (epoch * m + i + 1)**0.5

            theta = theta - lr * gradients

        # Calculate cost after each epoch (using full dataset)
        predictions = X_b.dot(theta)
        cost = (1/(2*m)) * np.sum(np.square(predictions - y))
        cost_history.append(cost)

    return theta, cost_history
```

17.2.3 Learning Rate Scheduling in SGD

Learning rate scheduling gradually reduces the learning rate over time, helping SGD settle at a minimum. This combines the benefits of larger initial steps (faster progress) with smaller later steps (precision).

Common scheduling strategies include:

1. **Time-based decay**: = / (1 + kt)
2. **Step decay**: = × 0.5^(t/)
3. **Exponential decay**: = × e^(-kt)

Where: - is the initial learning rate - t is the iteration number or epoch - k and are hyperparameters

```
def learning_rate_schedule(initial_lr, epoch, decay_rate=0.01):
    return initial_lr / (1 + decay_rate * epoch)
```

17.3 3. Mini-Batch Gradient Descent

Mini-Batch Gradient Descent combines the best of both worlds: it updates parameters using small random subsets (mini-batches) of the training data.

17.3.1 Conceptual Understanding

This is like surveying a representative sample of the mountain before each step. The direction isn't as precise as Batch GD but is more accurate than SGD, while maintaining computational efficiency.

17.3.2 Key Properties

- **Semi-randomized**: Less variance than SGD, more than Batch GD
- **Good memory efficiency**: Only needs to process a batch at a time
- **Vectorization-friendly**: Can leverage parallel processing
- **Balance of stability and speed**: Smoother convergence than SGD, faster than Batch GD
- **Industry standard**: Most practical implementations use this approach

```
def mini_batch_gradient_descent(X, y, batch_size=32, learning_rate=0.01, epochs=10):
    m = X.shape[0]
    n_features = X.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_b = np.c_[np.ones((m, 1)), X]
    cost_history = []

    n_batches = int(np.ceil(m / batch_size))

    for epoch in range(epochs):
        # Shuffle data at the beginning of each epoch
        indices = np.random.permutation(m)
        X_shuffled = X_b[indices]
        y_shuffled = y[indices]

        for batch in range(n_batches):
```

```
            # Calculate batch indices
            start_idx = batch * batch_size
            end_idx = min((batch + 1) * batch_size, m)

            # Extract mini-batch
            X_mini = X_shuffled[start_idx:end_idx]
            y_mini = y_shuffled[start_idx:end_idx]

            # Update parameters using mini-batch
            predictions = X_mini.dot(theta)
            error = predictions - y_mini
            gradients = (1/len(X_mini)) * X_mini.T.dot(error)
            theta = theta - learning_rate * gradients

        # Calculate cost after each epoch (using full dataset)
        predictions = X_b.dot(theta)
        cost = (1/(2*m)) * np.sum(np.square(predictions - y))
        cost_history.append(cost)

    return theta, cost_history
```

17.4 Theoretical Understanding of Batch Size Impact

The batch size directly affects:

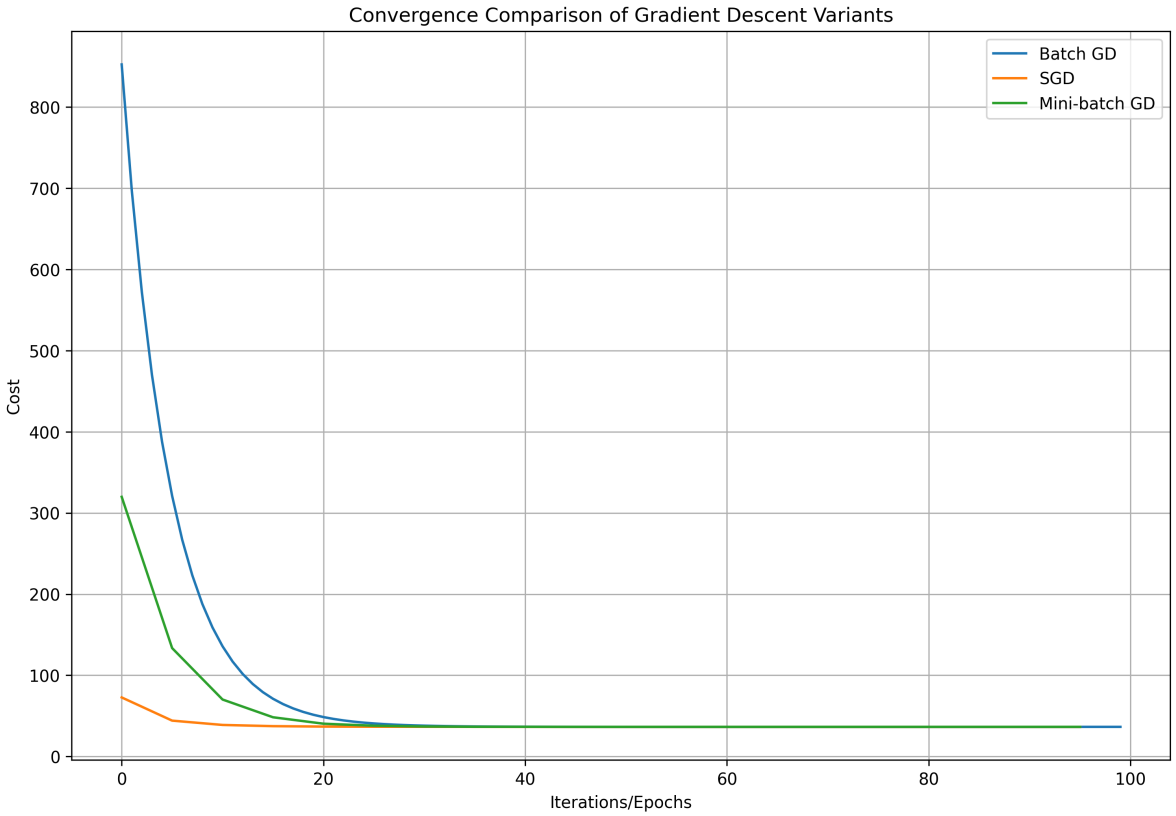
1. **Gradient estimation quality**: Larger batches provide more accurate gradient estimates
2. **Update frequency**: Smaller batches allow more frequent updates
3. **Generalization performance**: Research suggests very large batches may lead to poorer generalization
4. **Hardware utilization**: Optimal batch sizes leverage hardware parallelism

17.5 Comparison of Gradient Descent Variants

Let's compare all three approaches:

```
def compare_gd_variants(X, y):
    # Prepare data
    m = X.shape[0]
```

```
    X_scaled = StandardScaler().fit_transform(X)

    # Run Batch GD
    theta_bgd, cost_bgd = batch_gradient_descent(X_scaled, y, learning_rate=0.1, iterations=

    # Run SGD
    theta_sgd, cost_sgd = stochastic_gradient_descent(X_scaled, y, learning_rate=0.1, epochs=

    # Run Mini-batch GD
    theta_mbgd, cost_mbgd = mini_batch_gradient_descent(X_scaled, y, batch_size=16, learning_

    # Visualize convergence
    plt.figure(figsize=(12, 8))
    plt.plot(cost_bgd, label='Batch GD')
    plt.plot(range(0, len(cost_sgd) * 5, 5), cost_sgd, label='SGD')  # Adjusted for epoch cou
    plt.plot(range(0, len(cost_mbgd) * 5, 5), cost_mbgd, label='Mini-batch GD')  # Adjusted f
    plt.xlabel('Iterations/Epochs')
    plt.ylabel('Cost')
    plt.title('Convergence Comparison of Gradient Descent Variants')
    plt.legend()
    plt.grid(True)
    plt.show()

    return theta_bgd, theta_sgd, theta_mbgd

# Compare all three variants
theta_bgd, theta_sgd, theta_mbgd = compare_gd_variants(X_train, y_train)
```



17.6 When to Use Each Variant

Algorithm	Large Datasets	Memory Efficiency	Convergence Stability	Use Case
Batch GD	Slow	High memory usage	Very stable	Small datasets, need for deterministic solution
SGD	Fast	Very efficient	Noisy, may not settle exactly	Very large datasets, online learning

Algorithm	Large Datasets	Memory Efficiency	Convergence Stability	Use Case
Mini-batch GD	Fast	Efficient	Good balance	Most practical applications, especially deep learning

18 Advanced Optimization Techniques

18.1 Beyond Vanilla Gradient Descent

While gradient descent is powerful, several advanced optimization algorithms build upon it to address its limitations:

18.1.1 1. Momentum

Momentum adds a "velocity" term that accumulates past gradients, helping overcome: - Small local variations (noise) - Plateaus where gradients are small - The ravine problem by dampening oscillations

The update rule becomes: v = v - J( ) = + v

Where: - v is the velocity vector (initialized to zeros) - is the momentum coefficient (typically 0.9) - is the learning rate

Conceptually, momentum is like rolling a ball down the hill – it builds up velocity in consistent directions and dampens oscillations perpendicular to the main direction.

18.1.2 2. RMSprop (Root Mean Square Propagation)

RMSprop adapts the learning rate for each parameter based on the historical gradient magnitudes:

s = s + (1-)(J( ))² = - /√(s+) * J( )

Where: - s tracks the exponentially weighted average of squared gradients - is typically 0.9 - is a small value to prevent division by zero

This allows different parameters to have different effective learning rates based on their gradient histories.

18.1.3 3. Adam (Adaptive Moment Estimation)

Adam combines ideas from both momentum and RMSprop:

m = m + (1-) J( ) // First moment (momentum) v = v + (1-)(J( ))² // Second moment (RMSprop) m̂ = m/(1-) // Bias correction v̂ = v/(1-) // Bias correction = - * m̂/√(v̂+)

Where typical values are =0.9, =0.999, and =10 .

Adam is currently one of the most popular optimization algorithms for deep learning due to its excellent performance across a wide range of problems.

18.2 Regularization

Regularization prevents overfitting by adding a penalty to the cost function based on the size of model parameters. This creates a balance between fitting the training data well and maintaining simpler models.

18.2.1 L1 Regularization (Lasso)

L1 regularization adds a penalty proportional to the absolute value of weights, often resulting in sparse models with some coefficients set to zero (feature selection).

J_L1( ) = J( ) + | |

The gradient includes an additional term: sign( ) for each parameter.

```
def lasso_gradient_descent(X, y, learning_rate=0.01, iterations=1000, alpha=0.1):
    m = X.shape[0]
    n_features = X.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_b = np.c_[np.ones((m, 1)), X]
    cost_history = []

    for i in range(iterations):
        predictions = X_b.dot(theta)
        error = predictions - y

        # Gradient for the intercept (no regularization)
        gradient_intercept = (1/m) * X_b[:, 0:1].T.dot(error)

        # Gradient for the weights (with L1 regularization)
```

```
        gradients_weights = (1/m) * X_b[:, 1:].T.dot(error) + (alpha/m) * np.sign(theta[1:])

        # Update parameters
        theta[0] = theta[0] - learning_rate * gradient_intercept
        theta[1:] = theta[1:] - learning_rate * gradients_weights

        # Calculate cost with L1 regularization
        mse = (1/(2*m)) * np.sum(np.square(error))
        l1_penalty = (alpha/m) * np.sum(np.abs(theta[1:]))
        cost = mse + l1_penalty
        cost_history.append(cost)

    return theta, cost_history
```

18.2.2 L2 Regularization (Ridge)

L2 regularization adds a penalty proportional to the square of weights, shrinking all coefficients but rarely setting them to exactly zero.

J_L2( ) = J( ) + $^2/2$

The gradient includes an additional term: for each parameter.

```
def ridge_gradient_descent(X, y, learning_rate=0.01, iterations=1000, alpha=0.1):
    m = X.shape[0]
    n_features = X.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_b = np.c_[np.ones((m, 1)), X]
    cost_history = []

    for i in range(iterations):
        predictions = X_b.dot(theta)
        error = predictions - y

        # Gradient for the intercept (no regularization)
        gradient_intercept = (1/m) * X_b[:, 0:1].T.dot(error)

        # Gradient for the weights (with L2 regularization)
        gradients_weights = (1/m) * X_b[:, 1:].T.dot(error) + (alpha/m) * 2 * theta[1:]

        # Update parameters
        theta[0] = theta[0] - learning_rate * gradient_intercept
```

```
        theta[1:] = theta[1:] - learning_rate * gradients_weights

        # Calculate cost with L2 regularization
        mse = (1/(2*m)) * np.sum(np.square(error))
        l2_penalty = (alpha/(2*m)) * np.sum(np.square(theta[1:]))
        cost = mse + l2_penalty
        cost_history.append(cost)

    return theta, cost_history
```

18.2.3 Elastic Net

Elastic Net combines both L1 and L2 regularization, providing a balance between the feature selection properties of L1 and the parameter shrinking of L2:

J_EN( ) = J( ) + | | + (1-) ^{2}/2

Where controls the balance between L1 and L2 penalties.

```
def elastic_net_gradient_descent(X, y, learning_rate=0.01, iterations=1000, alpha=0.1, l1_ra
    m = X.shape[0]
    n_features = X.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_b = np.c_[np.ones((m, 1)), X]
    cost_history = []

    for i in range(iterations):
        predictions = X_b.dot(theta)
        error = predictions - y

        # Gradient for the intercept (no regularization)
        gradient_intercept = (1/m) * X_b[:, 0:1].T.dot(error)

        # Gradient for the weights (with Elastic Net regularization)
        l1_grad = (alpha/m) * l1_ratio * np.sign(theta[1:])
        l2_grad = (alpha/m) * (1 - l1_ratio) * 2 * theta[1:]
        gradients_weights = (1/m) * X_b[:, 1:].T.dot(error) + l1_grad + l2_grad

        # Update parameters
        theta[0] = theta[0] - learning_rate * gradient_intercept
        theta[1:] = theta[1:] - learning_rate * gradients_weights
```

```
        # Calculate cost with Elastic Net regularization
        mse = (1/(2*m)) * np.sum(np.square(error))
        l1_penalty = (alpha * l1_ratio/m) * np.sum(np.abs(theta[1:]))
        l2_penalty = (alpha * (1 - l1_ratio)/(2*m)) * np.sum(np.square(theta[1:]))
        cost = mse + l1_penalty + l2_penalty
        cost_history.append(cost)

    return theta, cost_history
```

18.3 Early Stopping

Early stopping prevents overfitting by monitoring validation error and stopping training when it starts to increase. This technique leverages the observation that models typically fit the training data progressively better over time while their performance on unseen data eventually starts to degrade.

18.3.1 Conceptual Understanding

Early stopping can be viewed as a form of implicit regularization: - During initial training, the model learns general patterns that apply to both training and validation data - Later in training, the model starts to memorize training examples (overfit), harming generalization

By stopping "early" when validation performance starts to deteriorate, we capture the model at its optimal generalization point.

18.4 Early Stopping

Early stopping prevents overfitting by monitoring validation error and stopping training when it starts to increase.

```
def gradient_descent_with_early_stopping(X_train, y_train, X_val, y_val, learning_rate=0.01,
    m_train = X_train.shape[0]
    n_features = X_train.shape[1]
    theta = np.zeros((n_features + 1, 1))
    X_train_b = np.c_[np.ones((m_train, 1)), X_train]
    X_val_b = np.c_[np.ones((X_val.shape[0], 1)), X_val]

    train_cost_history = []
    val_cost_history = []
```

```
best_val_cost = float('inf')
best_theta = None
patience_counter = 0

for i in range(max_iterations):
    # Train step
    predictions = X_train_b.dot(theta)
    error = predictions - y_train
    gradients = (1/m_train) * X_train_b.T.dot(error)
    theta = theta - learning_rate * gradients

    # Calculate training cost
    train_cost = (1/(2*m_train)) * np.sum(np.square(error))
    train_cost_history.append(train_cost)

    # Calculate validation cost
    val_predictions = X_val_b.dot(theta)
    val_error = val_predictions - y_val
    val_cost = (1/(2*X_val.shape[0])) * np.sum(np.square(val_error))
    val_cost_history.append(val_cost)

    # Early stopping logic
    if val_cost < best_val_cost:
        best_val_cost = val_cost
        best_theta = theta.copy()
        patience_counter = 0
    else:
        patience_counter += 1

    if patience_counter >= patience:
        print(f"Early stopping at iteration {i}")
        break

# Visualize training and validation costs
plt.figure(figsize=(10, 6))
plt.plot(train_cost_history, label='Training Cost')
plt.plot(val_cost_history, label='Validation Cost')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Training and Validation Costs with Early Stopping')
plt.legend()
plt.grid(True)
```

```
    plt.show()

    return best_theta, train_cost_history, val_cost_history
```

18.5 Batch Normalization

Batch Normalization improves training by normalizing layer inputs, addressing issues like internal covariate shift and vanishing gradients.

```
def batch_normalize(X, epsilon=1e-8):
    """Simple implementation of batch normalization for a mini-batch"""
    # Calculate mean and variance along the batch dimension
    batch_mean = np.mean(X, axis=0)
    batch_var = np.var(X, axis=0)

    # Normalize
    X_normalized = (X - batch_mean) / np.sqrt(batch_var + epsilon)

    # Parameters for scale and shift (in practice, these would be learned)
    gamma = np.ones(X.shape[1])
    beta = np.zeros(X.shape[1])

    # Scale and shift
    X_bn = gamma * X_normalized + beta

    return X_bn
```

19 Implementing Gradient Descent in Popular Libraries

19.1 Using Scikit-Learn

```
from sklearn.linear_model import SGDRegressor, SGDClassifier

# For regression
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty='l2', alpha=0.01, learning_rate='opti
sgd_reg.fit(X_train, y_train.ravel())

# For classification
from sklearn.datasets import make_classification
X_class, y_class = make_classification(n_samples=1000, n_features=20, random_state=42)
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(X_class, y_class

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, penalty='l2', alpha=0.01, learning_rate='opt
sgd_clf.fit(X_train_class, y_train_class)
```

```
SGDClassifier(alpha=0.01)
```

19.2 Using TensorFlow/Keras

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD

# Create a simple model
model = Sequential([
    Dense(10, activation='relu', input_shape=(X_train.shape[1],)),
```

```
    Dense(1)
])

# Configure the optimizer
optimizer = SGD(learning_rate=0.01)

# Compile the model
model.compile(optimizer=optimizer, loss='mse')

# Train the model
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    verbose=0
)

# Plot the learning curves
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Progress with SGD in TensorFlow/Keras')
plt.legend()
plt.grid(True)
plt.show()
```

20 Conclusion

Gradient Descent is a cornerstone optimization algorithm in machine learning with several variants optimized for different scenarios. The choice between Batch GD, SGD, and Mini-batch GD depends on your specific needs regarding speed, memory efficiency, and stability.

Advanced techniques like regularization, early stopping, and batch normalization can significantly enhance the performance of gradient-based optimization, leading to better models with improved generalization capabilities.

When implementing Gradient Descent, remember these key considerations: - Choose an appropriate learning rate - Scale your features - Select the right variant for your dataset size - Consider applying regularization to prevent overfitting - Monitor validation performance for early stopping

With a solid understanding of these concepts and techniques, you can effectively leverage Gradient Descent to train high-performing machine learning models across a wide range of applications.

20.1 References

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
2. Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.
3. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ICML.

21 Neural Networks and Deep Learning Foundations

21.1 ML and Deep Learning History

21.1.1 1950s-1960s: Early Developments

Initial excitement about machine learning began in the 1950s.

Perceptron was one of the first models: a linear classifier, trained using a method similar to stochastic gradient descent.

Limitation: Perceptrons cannot solve non-linearly separable problems (e.g., XOR), which led to a decline in popularity.

Analogy: Think of a perceptron like a straight knife trying to slice a circular cake with internal patterns—it can't reach the inner parts effectively.

```
# Simple Perceptron Example in Python
import numpy as np

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])  # AND operation

weights = np.zeros(2)
bias = 0
lr = 0.1

for epoch in range(10):
    for xi, target in zip(X, y):
        output = np.dot(xi, weights) + bias
        prediction = 1 if output > 0 else 0
        error = target - prediction
        weights += lr * error * xi
        bias += lr * error

print("Trained weights:", weights)
```

```
Trained weights: [0.2 0.1]
```

21.1.2 1970s-1980s: Rise of Connectionism

Interest shifted to connectionism — models inspired by how the brain works.

Hidden layers were introduced, increasing model expressiveness.

Achieved success in tasks like optical character recognition.

Analogy: Adding hidden layers is like giving a child a set of crayons instead of just a pencil—it allows more flexibility and richer representations.

21.1.3 1990s–2000s: Decline & Rise of Alternatives

Interest declined again due to slow training and lack of data.

Logistic Regression and SVMs with regularization and kernel tricks became dominant.

21.2 Overview of Machine Learning

21.2.1 Core Components

Data: The foundation of learning. Inputs (features) and Outputs (labels/targets).

Represented using a design matrix: rows are examples, columns are features.

```python
# Design matrix representation
import pandas as pd

df = pd.DataFrame({
    'feature1': [1, 2, 3],
    'feature2': [4, 5, 6],
    'output': [7, 8, 9]
})
print(df)
```

```
   feature1  feature2  output
0         1         4       7
1         2         5       8
2         3         6       9
```

Model: Maps input to output.

- Parametric (e.g., Linear Regression): fixed number of parameters.
- Non-parametric (e.g., k-NN): model complexity grows with data.

Objective Function: Measures how well the model is performing.

- Mean Squared Error (MSE) for regression.
- Cross-Entropy Loss for classification.
- Often derived from Maximum Likelihood Estimation (MLE).

```
from sklearn.metrics import mean_squared_error, log_loss

y_true = [0, 1, 1, 0]
y_pred_prob = [0.1, 0.9, 0.8, 0.3]

print("Cross-entropy loss:", log_loss(y_true, y_pred_prob))
```

```
Cross-entropy loss: 0.19763488164214868
```

Optimization Algorithm: Minimizes the objective function.

Gradient Descent: Move in the direction opposite to the gradient.

```
# Simple Gradient Descent
def f(x): return (x - 3) ** 2
def grad_f(x): return 2 * (x - 3)

x = 0
lr = 0.1
for _ in range(20):
    x -= lr * grad_f(x)

print("Minimum at:", x)
```

```
Minimum at: 2.9654123548617948
```

Analogy: Optimization is like hiking down a mountain by always stepping downhill (gradient direction).

21.2.2 Key Concepts: Overfitting and Underfitting

Underfitting: Model is too simple and fails to capture the underlying pattern in data.

Overfitting: Model is too complex and memorizes the training data, performing poorly on unseen data.

The goal is to balance complexity to minimize both training and test error.

21.3 Neural Networks: From Biological Inspiration to Deep Learning

21.3.1 Why Are They Called Neural Networks?

Neural networks draw inspiration from the structure and function of the human brain, hence the name. Let's explore this analogy:

Biological Neurons: In our brains, neurons receive signals through dendrites, process them in the cell body, and transmit output signals through axons to other neurons.

Artificial Neurons: In artificial neural networks (ANNs), we have mathematical "neurons" that:

1. Receive inputs (like dendrites receiving signals)
2. Apply weights to these inputs (representing synaptic strengths)
3. Sum the weighted inputs and apply an activation function (like the neuron firing)
4. Produce an output that connects to other neurons

```
import numpy as np

# Simple artificial neuron
def neuron(inputs, weights, bias, activation_function):
    # Calculate weighted sum of inputs (like dendrites and cell body function)
    weighted_sum = np.dot(inputs, weights) + bias

    # Apply activation function (like neuron firing)
    output = activation_function(weighted_sum)

    return output
```

While actual brains are infinitely more complex, this simplified model has proven remarkably effective for machine learning tasks.

21.4 Linearity and Non-Linearity in Neural Networks

21.4.1 The Concept of Linearity

A linear function has two key properties: 1. Additivity: $f(x+y) = f(x)+f(y)$ 2. Homogeneity: $f(\alpha x) = \alpha f(x)$

In neural networks, a linear transformation of inputs can be represented as:

```
def linear_layer(x, W, b):
    # Linear transformation: z = Wx + b
    z = np.dot(W, x) + b
    return z
```

21.4.2 Why We Need Non-Linearity

If we were to stack multiple linear layers together:

```
# Example of stacking linear layers
import numpy as np
import matplotlib.pyplot as plt

# Sample data
x = np.array([1, 2, 3, 4])

# First layer parameters
W1 = np.array([[0.1, 0.2, 0.3, 0.4],
               [0.5, 0.6, 0.7, 0.8]])
b1 = np.array([0.1, 0.2])

# Second layer parameters
W2 = np.array([[0.3, 0.5],
               [0.7, 0.9]])
b2 = np.array([0.2, 0.4])

# First linear layer
z1 = np.dot(W1, x) + b1
print("Output of first layer:", z1)

# Second linear layer
z2 = np.dot(W2, z1) + b2
print("Output of second layer:", z2)
```

```
# Equivalent single layer
W_combined = np.dot(W2, W1)
b_combined = np.dot(W2, b1) + b2
z_combined = np.dot(W_combined, x) + b_combined
print("Output of combined layer:", z_combined)
```

```
Output of first layer: [3.1 7.2]
Output of second layer: [4.73 9.05]
Output of combined layer: [4.73 9.05]
```

Mathematically, this is equivalent to a single linear transformation: $z2 = W2(W1x+b1)+b2 = (W2W1)x + (W2b1 + b2)$

This means that without non-linearity, a deep neural network would be no more powerful than a single layer! This is where non-linear activation functions become crucial.

21.4.3 Non-Linearity in Neural Networks

Non-linear functions enable neural networks to learn complex patterns:

```
def non_linear_layer(x, W, b, activation_function):
    # Linear transformation
    z = np.dot(W, x) + b
    # Non-linear activation
    a = activation_function(z)
    return a
```

This non-linearity allows neural networks to: - Approximate any continuous function (universal approximation theorem) - Learn hierarchical features - Solve complex, non-linear problems like image recognition, natural language processing, etc.

21.5 Activation Functions: Adding Non-Linearity

21.6 Activation Functions: Understanding the Neural Network's Decision-Making Process

Activation functions are the critical components that give neural networks their power to learn complex patterns. Without them, neural networks would be nothing more than linear regression models, unable to solve interesting problems.

21.6.1 The Conceptual Role of Activation Functions

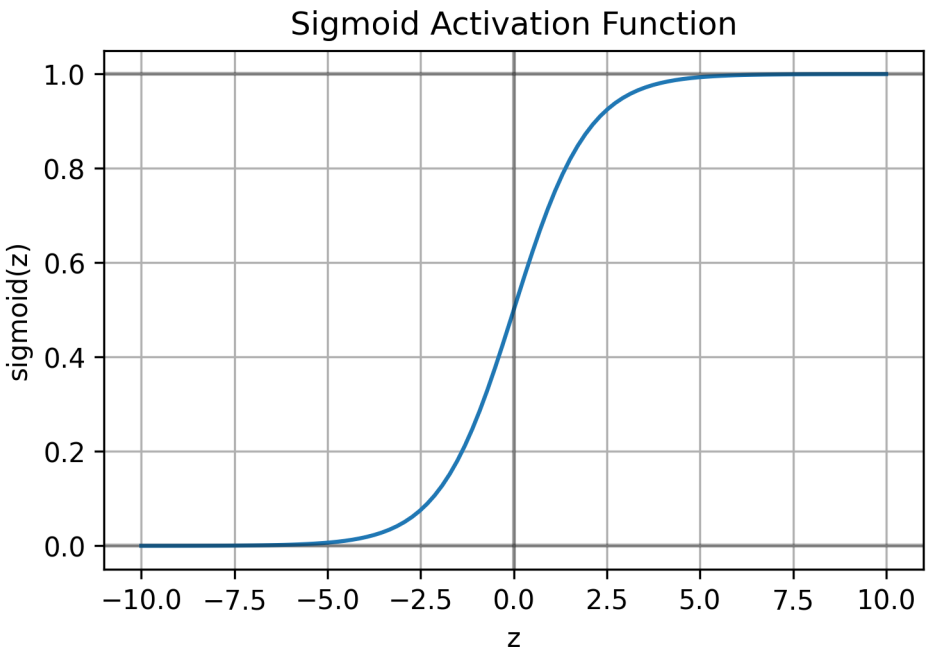
Activation functions serve several crucial purposes:

1. **Introducing Non-Linearity**: The real world is rarely linear. Activation functions allow networks to model complex, non-linear relationships.
2. **Feature Transformation**: They transform input signals into more useful representations.
3. **Decision Boundaries**: They help create complex decision boundaries that separate different classes of data.
4. **Biological Inspiration**: They mimic the "firing" behavior of biological neurons, which activate only when inputs reach certain thresholds.

21.6.2 Sigmoid Function: The S-Shaped Decision Maker

The sigmoid function maps any input to a value between 0 and 1, creating a smooth S-shaped curve:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Visualization
import matplotlib.pyplot as plt

z = np.linspace(-10, 10, 100)
plt.plot(z, sigmoid(z))
plt.title("Sigmoid Activation Function")
plt.xlabel("z")
plt.ylabel("sigmoid(z)")
plt.grid(True)
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.axhline(y=1, color='k', linestyle='-', alpha=0.3)
plt.axvline(x=0, color='k', linestyle='-', alpha=0.3)
```



Conceptual Understanding: - Acts like a "probability estimator" - values close to 1 represent high confidence, values close to 0 represent low confidence. - Creates a smooth transition between inactive (0) and active (1) states. - Historically important but less used in hidden layers today.

Mental Model: Imagine a thermostat that gradually turns on heating as the temperature drops, with a smooth transition rather than an abrupt on/off switch.

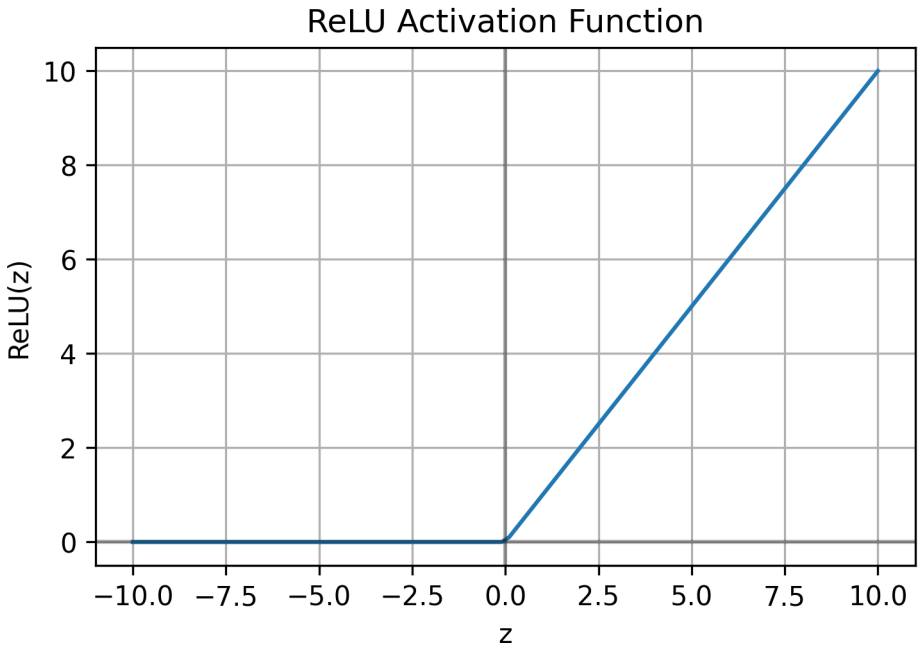
Mathematical Intuition: The sigmoid compresses extreme values while being most sensitive to changes around z=0.

21.6.3 ReLU: The Modern Workhorse

ReLU (Rectified Linear Unit) is elegantly simple: it returns the input if positive, otherwise zero.

```
import numpy as np
import matplotlib.pyplot as plt

def relu(z):
    return np.maximum(0, z)

# Visualization
```

```
z = np.linspace(-10, 10, 100)
plt.plot(z, relu(z))
plt.title("ReLU Activation Function")
plt.xlabel("z")
plt.ylabel("ReLU(z)")
plt.grid(True)
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.axvline(x=0, color='k', linestyle='-', alpha=0.3)
plt.show()
```



Conceptual Understanding: - Acts as a feature selector - it highlights useful signals (positive values) while suppressing noise (negative values). - Creates sparsity in the network, as many neurons will output zero for any given input. - Computationally efficient and helps signals flow more effectively through deep networks.

Mental Model: Think of ReLU as a filter that only allows positive information to pass through, completely blocking negative information.

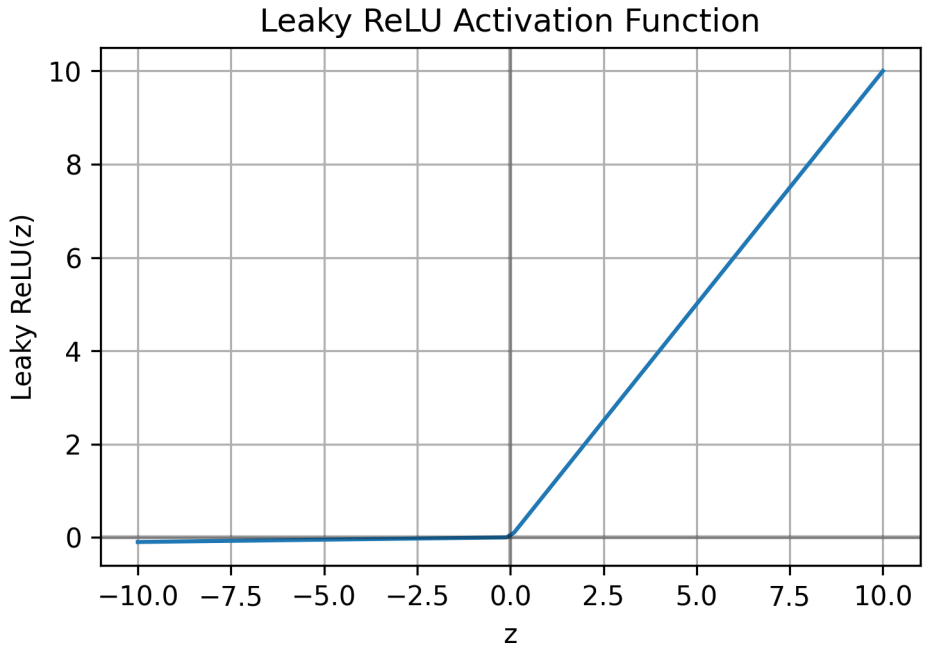
Biological Intuition: Similar to biological neurons that fire only when stimulation exceeds a threshold, ReLU neurons are only "activated" by positive inputs.

21.6.4 Variants: Leaky ReLU, ELU, and GELU

These ReLU variants address some of its limitations:

Leaky ReLU: Allows a small gradient when the unit is not active:

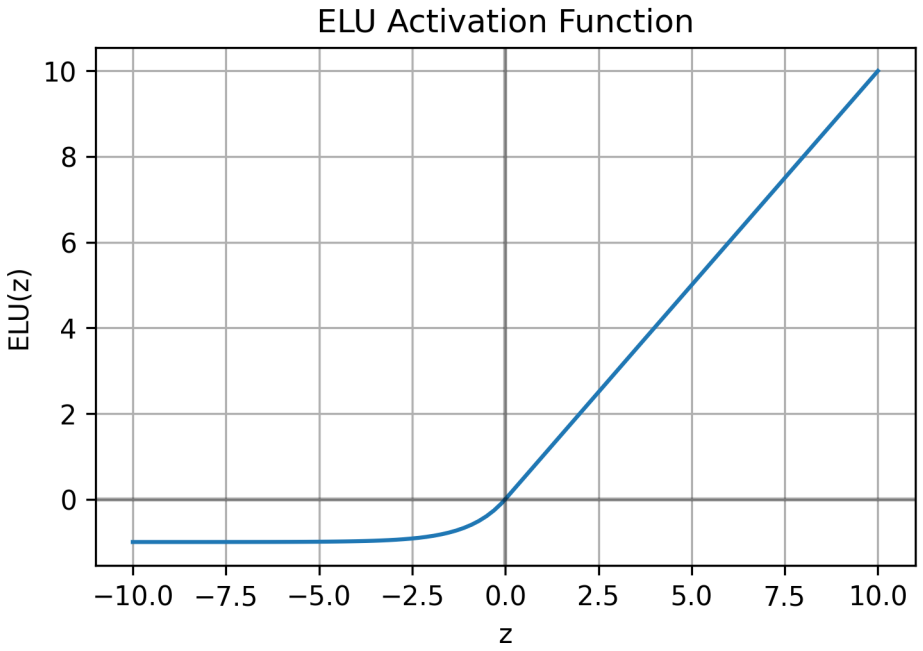
```
def leaky_relu(z, alpha=0.01):
    return np.maximum(alpha * z, z)

# Visualization
z = np.linspace(-10, 10, 100)
plt.plot(z, leaky_relu(z))
plt.title("Leaky ReLU Activation Function")
plt.xlabel("z")
plt.ylabel("Leaky ReLU(z)")
plt.grid(True)
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.axvline(x=0, color='k', linestyle='-', alpha=0.3)
plt.show()
```



Conceptual Understanding: Gives "dying" neurons a chance to recover by allowing a small gradient when inactive.

ELU (Exponential Linear Unit): Smoothes the transition at zero:

```python
import numpy as np
import matplotlib.pyplot as plt

def elu(z, alpha=1.0):
    # Use NumPy's where function for element-wise conditional operations
    return np.where(z > 0, z, alpha * (np.exp(z) - 1))

# Visualization
z = np.linspace(-10, 10, 100)
plt.plot(z, elu(z))
plt.title("ELU Activation Function")
plt.xlabel("z")
plt.ylabel("ELU(z)")
plt.grid(True)
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.axvline(x=0, color='k', linestyle='-', alpha=0.3)
plt.show()
```



Conceptual Understanding: Combines ReLU's benefits with a smoother transition, helping gradients flow better.

GELU (Gaussian Error Linear Unit): Used in many transformer models like BERT:

```
def gelu(z):
    return 0.5 * z * (1 + np.tanh(np.sqrt(2 / np.pi) * (z + 0.044715 * z**3)))
```

Conceptual Understanding: Weighs inputs by their value, approximating a smooth step function modulated by the input's magnitude.

21.6.5 Softmax: The Probability Distributor

Softmax converts a vector of values into a probability distribution:

```
def softmax(z):
    exp_z = np.exp(z - np.max(z))
    return exp_z / np.sum(exp_z)

# Visualization
z = np.array([2.0, 1.0, 0.1])
print("Softmax probabilities:", softmax(z))
```

```
Softmax probabilities: [0.65900114 0.24243297 0.09856589]
```

Conceptual Understanding: - Acts as a "decision maker" for multi-class problems. - Emphasizes the largest values while suppressing smaller ones. - Creates competition between outputs - increasing one probability must decrease others.

Mental Model: Imagine softmax as a committee voting system where members with stronger opinions (higher values) get more voting power, but the total votes must add up to 100%.

21.6.6 Choosing the Right Activation Function

The choice of activation function depends on:

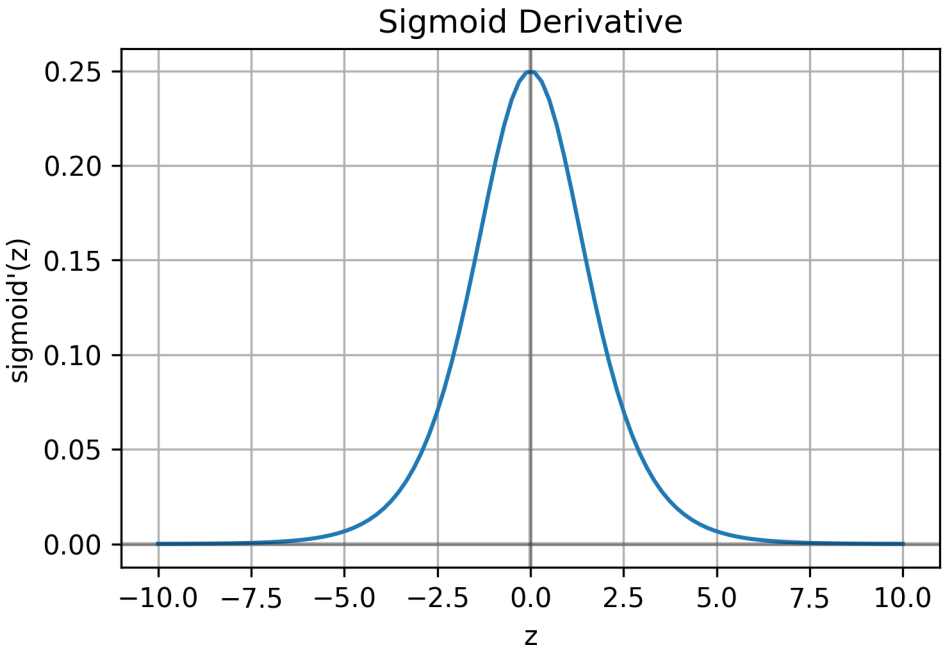
1. **Layer Type**:
    - Hidden layers: ReLU and variants work well
    - Output layer: Sigmoid for binary classification, Softmax for multi-class, Linear for regression
2. **Problem Domain**:
    - Image recognition: ReLU family performs well
    - Time series and sequence modeling: GELU and Swish often excel
    - Generative models: Leaky ReLU can help

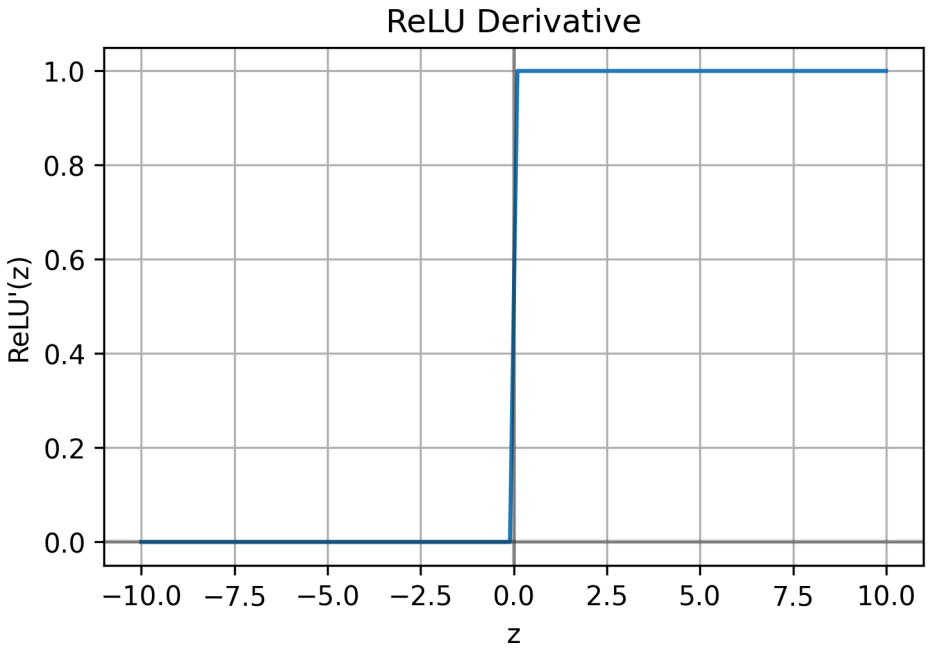
3. **Network Depth**:
    - Deeper networks often benefit from ReLU variants that help mitigate vanishing gradients

21.6.7 The Mathematics Behind Learning

Activation functions critically affect how networks learn through their derivatives:

```
# Derivatives
def sigmoid_derivative(z):
    sig = sigmoid(z)
    return sig * (1 - sig)

# Visualization
z = np.linspace(-10, 10, 100)
plt.plot(z, sigmoid_derivative(z))
plt.title("Sigmoid Derivative")
plt.xlabel("z")
plt.ylabel("sigmoid'(z)")
plt.grid(True)
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.axvline(x=0, color='k', linestyle='-', alpha=0.3)
plt.show()

def relu_derivative(z):
    return 1 if z > 0 else 0

# Visualization
z = np.linspace(-10, 10, 100)
plt.plot(z, [relu_derivative(zi) for zi in z])
plt.title("ReLU Derivative")
plt.xlabel("z")
plt.ylabel("ReLU'(z)")
plt.grid(True)
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.axvline(x=0, color='k', linestyle='-', alpha=0.3)
plt.show()
```





Conceptual Understanding: - The derivative determines how much a neuron's weights update during learning. - Functions with stronger derivatives in useful ranges learn faster. - Functions with derivatives that approach zero (like sigmoid at extremes) can slow or stop learning.

21.6.8 Visualizing Decision Boundaries

Different activation functions create different decision boundaries:

- Linear: Can only create linear boundaries (lines, planes)
- Sigmoid/Tanh: Can approximate curved boundaries with enough neurons
- ReLU: Creates piecewise linear boundaries, which can approximate any shape with enough neurons

This ability to create complex boundaries is what gives neural networks their remarkable flexibility.

21.7 The Role of Bias in Neural Networks

21.7.1 Understanding Bias in the Statistical Sense

In statistics and machine learning, bias refers to the model's tendency to consistently over or underestimate the true values. It's one component of the generalization error along with variance and irreducible error.

- **High Bias**: Model is too simple to capture the underlying pattern (underfitting)
- **High Variance**: Model is too complex and captures noise in the training data (overfitting)

21.7.2 Bias Nodes in Neural Networks

In neural networks, "bias" has a specific technical meaning: it's an additional parameter added to each layer that allows the activation function to shift:

```
# Example showing the effect of bias
import numpy as np

# Without bias
def no_bias_output(inputs, weights, activation_function):
    return activation_function(np.dot(weights, inputs))

# With bias
def with_bias_output(inputs, weights, bias, activation_function):
    return activation_function(np.dot(weights, inputs) + bias)

# Define a simple sigmoid function
```

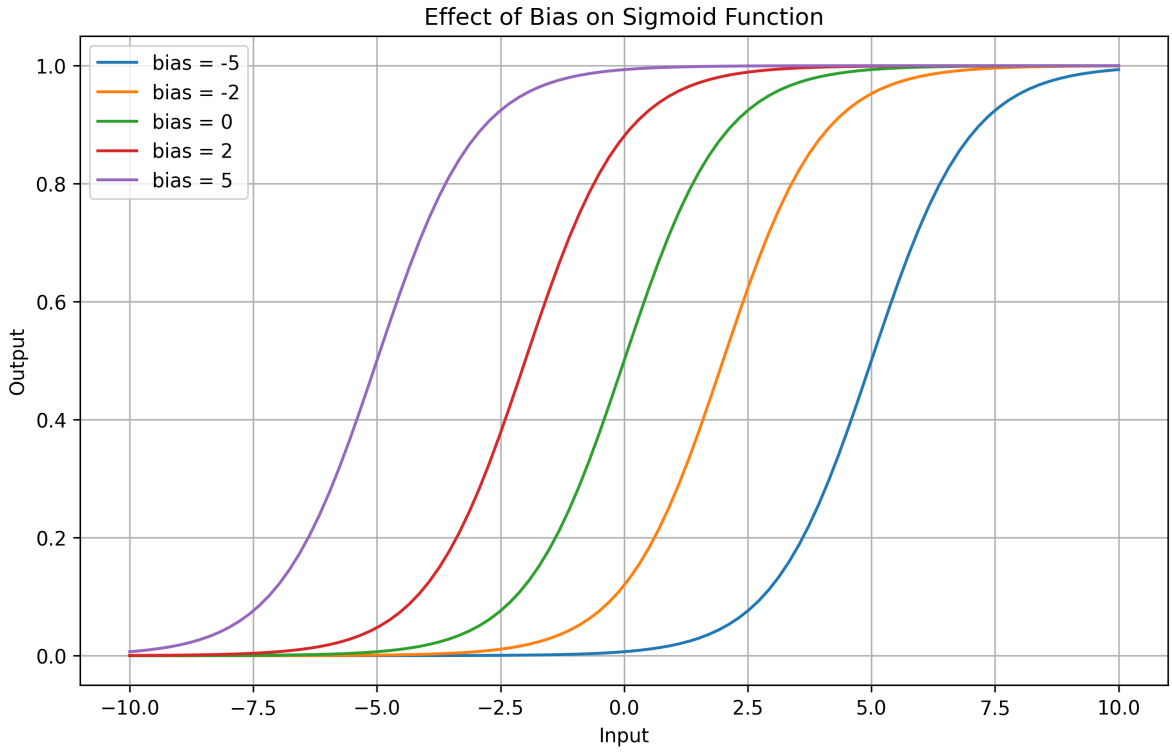
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Example data
inputs = np.array([0, 0])  # Both inputs are zero
weights = np.array([0.5, 0.5])

# Compare outputs
print("Without bias:", no_bias_output(inputs, weights, sigmoid))
print("With bias (b=1):", with_bias_output(inputs, weights, 1, sigmoid))
print("With bias (b=2):", with_bias_output(inputs, weights, 2, sigmoid))
print("With bias (b=-2):", with_bias_output(inputs, weights, -2, sigmoid))
```

```
Without bias: 0.5
With bias (b=1): 0.7310585786300049
With bias (b=2): 0.8807970779778823
With bias (b=-2): 0.11920292202211755
```

Analogy: Think of bias as the "y-intercept" in a linear equation y = mx + b. It shifts the entire activation function up or down, allowing the neuron to fire even when all inputs are zero.

Let's visualize the effect of bias on a sigmoid neuron:

```
def sigmoid_with_bias(x, bias):
    return 1 / (1 + np.exp(-(x + bias)))

x = np.linspace(-10, 10, 100)
plt.figure(figsize=(10, 6))
biases = [-5, -2, 0, 2, 5]
for b in biases:
    plt.plot(x, sigmoid_with_bias(x, b), label=f"bias = {b}")

plt.title("Effect of Bias on Sigmoid Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.legend()
plt.grid(True)
```



21.7.3 Practical Implementation in Python

Here's how bias is typically implemented in a neural network layer using NumPy:

```
class NeuralNetworkLayer:
    def __init__(self, input_size, output_size, activation_function):
        # Initialize weights and biases
        self.weights = np.random.randn(output_size, input_size) * 0.01
        self.bias = np.zeros((output_size, 1))
        self.activation_function = activation_function

    def forward(self, inputs):
        # Calculate the weighted sum including bias
        self.z = np.dot(self.weights, inputs) + self.bias
        # Apply activation function
        self.a = self.activation_function(self.z)
        return self.a
```

And here's how it looks in a modern deep learning framework like TensorFlow/Keras:

```
#| eval: false
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Create a model with two hidden layers
model = Sequential([
    # Input layer to first hidden layer
    Dense(128, activation='relu', input_shape=(784,), use_bias=True),
    # Second hidden layer
    Dense(64, activation='relu', use_bias=True),
    # Output layer
    Dense(10, activation='softmax', use_bias=True)
])

# Note: use_bias=True is actually the default in Keras
```

21.8 Putting It All Together: Building a Neural Network

Let's build a simple neural network that demonstrates these concepts:

```
import numpy as np

class SimpleNeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases
        self.W1 = np.random.randn(hidden_size, input_size) * 0.01
        self.b1 = np.zeros((hidden_size, 1))
        self.W2 = np.random.randn(output_size, hidden_size) * 0.01
        self.b2 = np.zeros((output_size, 1))

    def relu(self, Z):
        return np.maximum(0, Z)

    def softmax(self, Z):
        exp_Z = np.exp(Z - np.max(Z, axis=0, keepdims=True))
        return exp_Z / np.sum(exp_Z, axis=0, keepdims=True)

    def forward(self, X):
        # First layer linear transformation
```

```
        Z1 = np.dot(self.W1, X) + self.b1
        # Apply non-linear activation
        A1 = self.relu(Z1)
        # Second layer linear transformation
        Z2 = np.dot(self.W2, A1) + self.b2
        # Apply softmax activation for output layer
        A2 = self.softmax(Z2)

        return A2

    def predict(self, X):
        # Get probabilities
        probs = self.forward(X)
        # Return class with highest probability
        return np.argmax(probs, axis=0)

# Example usage
if __name__ == "__main__":
    # Create dummy data: 3 samples with 4 features each
    X = np.random.randn(4, 3)

    # Create a neural network with 4 input neurons, 5 hidden neurons, and 2 output classes
    nn = SimpleNeuralNetwork(4, 5, 2)

    # Make predictions
    predictions = nn.predict(X)
    print("Predictions:", predictions)
```

```
Predictions: [1 0 0]
```

21.9 Connection to Deep Learning

Deep learning refers to neural networks with many layers (hence "deep"). The key insight of deep learning is that:

1. Multiple layers allow the network to learn hierarchical representations
2. Lower layers learn simple features
3. Higher layers combine these features into more complex patterns

For example, in a deep convolutional network for image recognition: - First layer might detect edges and simple textures - Middle layers might detect patterns like corners and curves - Higher layers might detect entire objects like faces or cars

The non-linearity introduced by activation functions is what makes this hierarchical learning possible.

21.10 Real-World Analogy: Building a House

Let's use the analogy of building a house to understand neural networks:

1. **Inputs (X)**: These are the raw materials (bricks, wood, cement, etc.)
2. **Weights (W)**: These represent the importance of each material for different parts of the house
3. **Bias (b)**: This represents the architectural style or design preferences
4. **Activation function**: This represents the construction techniques that transform materials into structures
5. **Hidden layers**: These are the intermediate structures (walls, roof frames, plumbing)
6. **Output**: The completed house

Without non-linearity (activation functions), we could only build simple, linear structures. The non-linear activation functions allow us to create complex, curved, and intricate designs.

21.11 Conclusion

Neural networks derive their power from:

1. **Biological inspiration**: Learning from how our brains process information
2. **Non-linearity**: Enabling the approximation of complex functions
3. **Bias parameters**: Allowing flexibility in neuron activation
4. **Hierarchical representation**: Building complex concepts from simpler ones

Together, these elements have revolutionized machine learning and enabled breakthroughs in various fields including computer vision, natural language processing, and many other domains.

22 Neural Networks Foundations: TLUs, Perceptrons, and MLPs

22.1 Threshold Logic Units (TLUs)

22.1.1 The Building Blocks of Neural Networks

A Threshold Logic Unit (TLU), also known as a Linear Threshold Unit (LTU), is the foundational building block of early neural networks. Think of a TLU as a simplified model of a biological neuron - it receives multiple inputs, processes them, and produces a single output based on whether the combined input exceeds a certain threshold.

22.1.2 How TLUs Work

A TLU can be visualized as a decision-making unit with the following components:

- **Inputs**: A set of numerical values $x_1, x_2, ..., x_n$
- **Weights**: Each input is assigned a weight $w_1, w_2, ..., w_n$
- **Threshold**: A value θ that determines when the unit activates
- **Output**: A single value y (typically 0 or 1)

The TLU computes its output using this simple formula:

$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0, & \text{otherwise} \end{cases}$$

> **Note**
>
> This is often rewritten by moving the threshold to the other side, creating what we call a "bias" term $b = -\theta$:
>
> $$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i + b \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

22.1.3 Analogy: The Voting Committee

Imagine a committee where each member (input) has a different level of influence (weight) on the final decision. The committee is voting on whether to approve a proposal (output 1) or reject it (output 0). Each member casts their vote, which is then weighted by their influence. If the weighted sum of votes exceeds a certain threshold, the proposal is approved; otherwise, it's rejected.

22.1.4 TLUs as Logic Gates

TLUs can implement basic logical operations. Let's look at two examples:

22.1.4.1 Example 1: Logical AND Operation (x x)

For the logical AND of two binary inputs, we want: - Output 1 only when both inputs are 1 - Output 0 otherwise

We can achieve this with the following weights and threshold: - $w_1 = 1$ - $w_2 = 1$ - $\theta = 1.5$

With these parameters: - When $x_1 = 0, x_2 = 0$: $0 \cdot 1 + 0 \cdot 1 = 0 < 1.5$, so output is 0 - When $x_1 = 1, x_2 = 0$: $1 \cdot 1 + 0 \cdot 1 = 1 < 1.5$, so output is 0 - When $x_1 = 0, x_2 = 1$: $0 \cdot 1 + 1 \cdot 1 = 1 < 1.5$, so output is 0 - When $x_1 = 1, x_2 = 1$: $1 \cdot 1 + 1 \cdot 1 = 2 > 1.5$, so output is 1

22.1.4.2 Example 2: Logical Implication (x → x)

For the logical implication "if x then x ", we want: - Output 0 only when x is 1 and x is 0 (the only case where the implication fails) - Output 1 otherwise

We can achieve this with: - $w_1 = 1$ - $w_2 = -1$ - $\theta = -0.5$

Let's verify: - When $x_1 = 0, x_2 = 0$: $0 \cdot 1 + 0 \cdot (-1) = 0 > -0.5$, so output is 1 - When $x_1 = 1, x_2 = 0$: $1 \cdot 1 + 0 \cdot (-1) = 1 > -0.5$, so output is 1 - When $x_1 = 0, x_2 = 1$: $0 \cdot 1 + 1 \cdot (-1) = -1 < -0.5$, so output is 0 - When $x_1 = 1, x_2 = 1$: $1 \cdot 1 + 1 \cdot (-1) = 0 > -0.5$, so output is 1

22.1.5 Python Implementation of a TLU

```python
import numpy as np

class ThresholdLogicUnit:
    def __init__(self, weights, threshold):
        self.weights = np.array(weights)
        self.threshold = threshold

    def activate(self, inputs):
        # Ensure inputs is a numpy array
        inputs = np.array(inputs)

        # Calculate weighted sum
        weighted_sum = np.dot(inputs, self.weights)

        # Apply threshold
        return 1 if weighted_sum >= self.threshold else 0

# Example: TLU implementing logical AND
and_tlu = ThresholdLogicUnit([1, 1], 1.5)

# Test all possible inputs
inputs = [[0, 0], [0, 1], [1, 0], [1, 1]]
for input_pair in inputs:
    output = and_tlu.activate(input_pair)
    print(f"Inputs: {input_pair}, Output: {output}")

# Example: TLU implementing logical implication (x → x )
implication_tlu = ThresholdLogicUnit([1, -1], -0.5)

# Test all possible inputs
for input_pair in inputs:
    output = implication_tlu.activate(input_pair)
    print(f"Inputs: {input_pair}, Implication Output: {output}")
```

22.2 Perceptron

22.2.1 The First Learning Neural Network

The Perceptron, invented by Frank Rosenblatt in 1957, was one of the first algorithmic implementations of a learning neural network. It builds upon the TLU by adding a mechanism to automatically learn the weights and threshold.