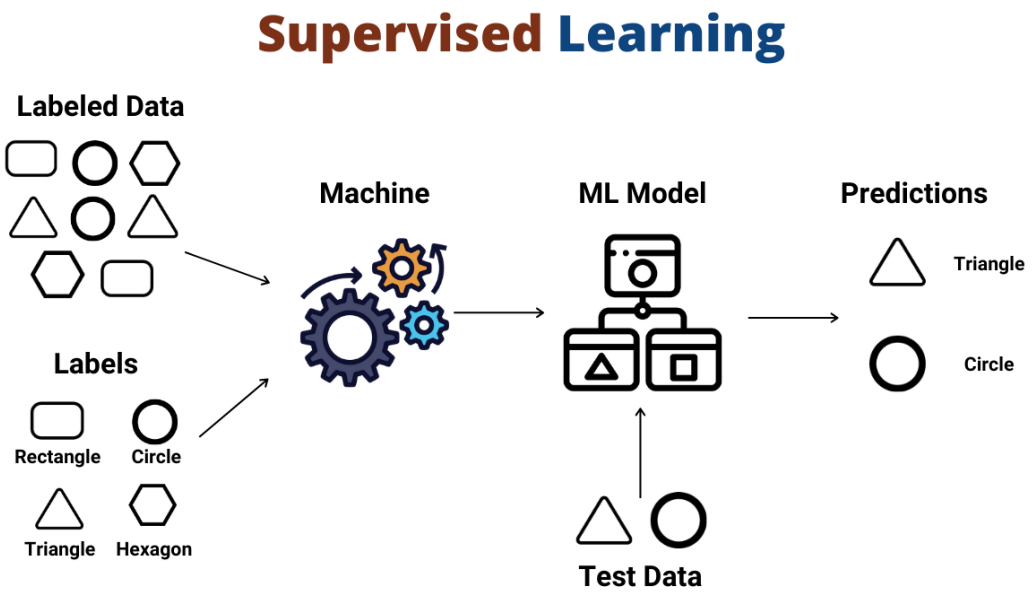


Week 6: Supervised Learning

Introduction to Supervised Learning

Understanding Supervised Learning

Definition: Supervised learning is a type of machine learning where the algorithm is trained on a labeled dataset, meaning the input data has corresponding output labels.

Key concept: The algorithm learns the mapping from input to output by using a set of training examples.

Purpose: Predicting outcomes for new, unseen data based on the patterns learned during training.

Key Terminology in Supervised Learning

Features (X):

Definition: The input variables or independent variables that the model uses to make predictions.

Example: In a housing price prediction model, features could include square footage, number of bedrooms, etc.

Target Variable (y):

Definition: The output variable or dependent variable that the model aims to predict.

Example: In the same housing price prediction model, the target variable is the price of the house.

Training Data:

Definition: The labeled dataset used to train the machine learning model.

Example: Pairs of (features, target) used for learning the patterns.

Types of Supervised Learning

- Two Main Types of Supervised Learning:

Regression:

Definition: Predicting a continuous output variable.

Example: Predicting house prices, stock prices.

Classification:

Definition: Predicting a categorical output variable.

Example: Classifying emails as spam or not spam, predicting whether a patient has a disease or not.

Regression

- Regression is a type of supervised learning where the goal is to predict a continuous or numerical output variable.

Example:

Imagine you are working on a real estate project, and you want to predict the price of houses based on various features such as square footage, number of bedrooms, and location. Here, the target variable (output) is a continuous value (the price).

- The algorithm learns a mapping from input features to a continuous output space.
- The predicted values are not limited to specific categories but can take any numeric value.

Use Cases: Predicting stock prices, house prices, temperature, sales revenue, etc.

Classification

- Classification is a type of supervised learning where the goal is to predict the category or class of a given input.

Example:

Consider an email filtering system. Given the features of an email, the task is to predict whether the email is spam or not spam. Here, the target variable (output) is a discrete category (spam or not spam).

- The algorithm classifies input data into predefined categories or classes.
- The output is a discrete label, and the goal is to assign each input to the correct category.

Use Cases: Email spam detection, image classification, sentiment analysis, medical diagnosis, etc.

Regression vs Classification

Distinguishing Characteristics:

The primary distinction between regression and classification lies in the nature of the output variable—continuous for regression and categorical for classification.

Real-world Analogies:

Think of real-world examples for both regression and classification to solidify their understanding.

Flexibility of Algorithms:

Some algorithms can be used for both regression and classification tasks, such as decision trees and support vector machines.

Common Supervised Learning Algorithms

1. **Linear Regression**:

Purpose: Predicting a continuous variable.

Key Feature: Assumes a linear relationship between features and target variable.

2. **Logistic Regression**:

Purpose: Binary classification.

Key Feature: Sigmoid function to map predictions between 0 and 1.

3. **Decision Trees**:

Purpose: Both classification and regression.

Key Feature: Tree-like structure of decision nodes.

Common Supervised Learning Algorithms

4. **Random Forest**:

Purpose: Ensemble method for improved performance.

Key Feature: Combines multiple decision trees.

5. **Support Vector Machines (SVM)**:

Purpose: Classification and regression.

Key Feature: Finds the hyperplane that best separates classes.

6. **K-Nearest Neighbors (KNN)**:

Purpose: Classification and regression.

Key Feature: Predicts based on the majority class among k-nearest neighbors.

Model Training Process Recap

Data Splitting :

- Divide the dataset into training and testing sets.
- Ensures the model's performance on unseen data.

Model Training :

- The algorithm learns the patterns in the training data.
- This is where the model understands the relationships between features and target.

Model Evaluation :

- Assess the model's performance on the testing set.
- Accuracy, precision, recall, F1 score, etc.

Hyperparameter Tuning :

- Adjust model settings to optimize performance.
- Fine-tuning for better predictions.

Hyperparameter Tuning

- Hyperparameters are settings external to the model that must be specified before training.

- Tuning involves adjusting these hyperparameters to optimize the model's performance.

Common Hyperparameters:

- Number of Trees (in Random Forest): Adjusts the number of decision trees in the ensemble.
- Kernel Type (in SVM): Specifies the type of mathematical function used for transforming input data.

Linear Regression

- Linear regression is used for predicting a continuous output variable based on one or more input features.
- It assumes a linear relationship between the input features and the target variable.

Example:

Consider predicting house prices based on features like square footage and the number of bedrooms. Linear regression would model the relationship as a straight line.

Linear Regression

How it Works:

The algorithm fits a line to the data, minimizing the difference between predicted and actual values using a mathematical technique called least squares.

Real-world Applications:

Predicting sales, estimating GDP growth, forecasting temperature, Estimating the relationship between age and blood pressure, etc.

Visualization:

Visual representation often involves a scatter plot of data points with a line fitted through them.

Example

Load the “USA_Housing.csv” dataset.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

df = pd.read_csv('USA_Housing.csv')
```

```
df.head()
```

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	Address
0	79545.458574	5.682861	7.009188	4.09	23086.800503	1.059034e+06	208 Michael Ferry Apt. 674\nLaurabury, NE 3701...
1	79248.642455	6.002900	6.730821	3.09	40173.072174	1.505891e+06	188 Johnson Views Suite 079\nLake Kathleen, CA...
2	61287.067179	5.865890	8.512727	5.13	36882.159400	1.058988e+06	9127 Elizabeth Stravenue\nDanieltown, WI 06482...
3	63345.240046	7.188236	5.586729	3.26	34310.242831	1.260617e+06	USS Barnett\nFPO AP 44820
4	59982.197226	5.040555	7.839388	4.23	26354.109472	6.309435e+05	USNS Raymond\nFPO AE 09386

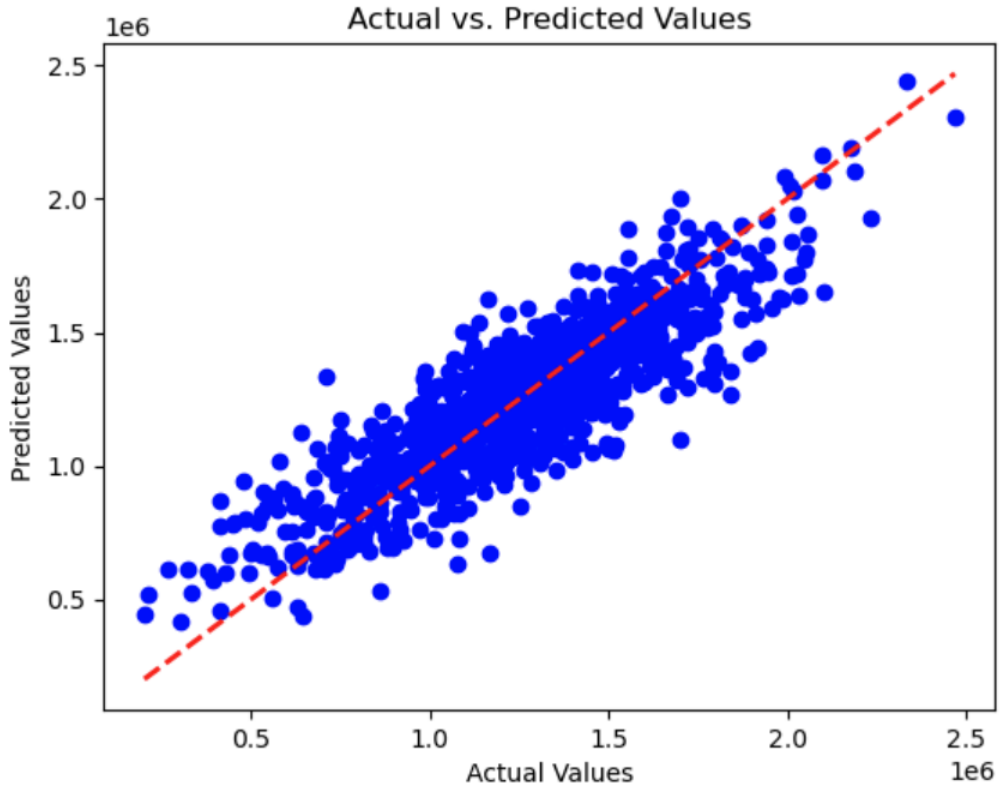
Example

Select predictive variables and train a Linear Regression model:

```
X = df[['Avg. Area Number of Rooms', 'Avg. Area Income', 'Avg. Area House Age']] # your predictor variables
y = df['Price']  # target variable

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

# Create a linear regression model
model = LinearRegression()

# Fit the model on the training data
model.fit(X_train, y_train)

# Predict values on the test set
y_pred = model.predict(X_test)

# Visualize the actual vs. predicted values
plt.scatter(y_test, y_pred, color='blue')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], linestyle='--', color='red', linewidth=2)
plt.title('Actual vs. Predicted Values')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.show()
```

Example

Plotting predicted versus actual values:



Logistic Regression

- Logistic regression is used for binary classification tasks, predicting the probability of an instance belonging to a particular class.
- It uses the logistic (sigmoid) function to map predictions between 0 and 1.

Example:

Classifying emails as spam or not spam. Logistic regression would provide a probability that an email is spam.

How it Works:

The algorithm models the relationship between the input features and the probability of belonging to a certain class using the logistic function.

Logistic Regression

Real-world Applications:

Predicting the likelihood of a patient developing a specific disease, Assessing the probability of a customer defaulting on a loan, Spam detection, credit scoring, disease diagnosis, etc.

Visualization:

Visual representation often involves an S-shaped curve representing the logistic function.

Linear Regression Versus Logistic Regression:

- Linear regression predicts continuous values, while logistic regression is specifically designed for binary classification.
- Logistic regression is suitable for classification tasks, providing probabilities that can be thresholded to make binary predictions.

Example

Training a model with Logistic Regression algorithm:

```python
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the Titanic dataset from a CSV file
titanic_data = pd.read_csv('titanic_clean.csv')

selected_features = ['Age', 'Sex', 'Fare']
X = titanic_data[selected_features]
y = titanic_data['Survived']

# Convert 'Sex' to numerical values (0 for female, 1 for male)
X['Sex'] = X['Sex'].map({'female': 0, 'male': 1})

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

# Create a logistic regression model
model = LogisticRegression()

# Fit the model on the training data
model.fit(X_train, y_train)

# Predict survival
y_pred = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.776536312849162
```

Decision Trees

- Decision trees can be used for both classification and regression tasks by breaking down a dataset into smaller subsets while recursively making decisions.

Example:

Classifying whether a passenger survived or not on the Titanic based on features like age, gender, and ticket class.

How it Works:

The algorithm builds a tree structure by making decisions based on features, optimizing for the most significant split at each node.

Decision Trees

Real-world Applications:

Predicting whether a customer will churn based on their interactions, Quality control by classifying defective and non-defective products, fraud detection, predicting housing prices, etc.

Visualization:

Visual representation often involves a tree diagram with nodes representing decisions and leaves representing outcomes.

Example

Training a model with Decision Trees algorithm:

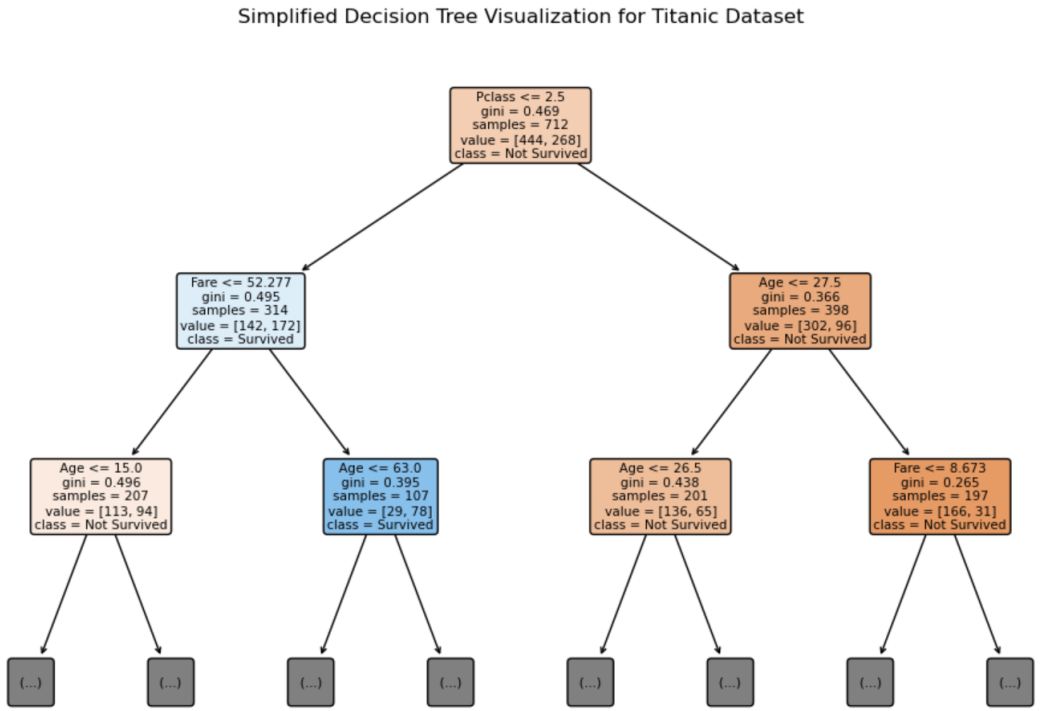
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score

# Load the Titanic dataset from a CSV file
titanic_data = pd.read_csv('titanic_clean.csv')

selected_features = ['Age', 'Pclass', 'Fare']
X = titanic_data[selected_features]
y = titanic_data['Survived']

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

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Fit the model on the training data
clf.fit(X_train, y_train)

# Predict survival
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.7206703910614525
```

Example

Decision Tree Visualization:

```
# Visualize a simplified version of the decision tree with max_depth=2
plt.figure(figsize=(12, 8))
plot_tree(clf, feature_names=selected_features, class_names=['Not Survived', 'Survived'],
          filled=True, rounded=True, max_depth=2)  # Adjust max_depth as needed
plt.title('Simplified Decision Tree Visualization for Titanic Dataset')
plt.show()
```



Example

Decision Tree text visualization:

```python
from sklearn.tree import export_text

# Visualize a simplified version of the decision tree with text-based rules
tree_rules = export_text(clf, max_depth=2, feature_names=selected_features)
print("Decision Tree Rules:")
print(tree_rules)
```

```
Decision Tree Rules:
|--- Pclass <= 2.50
|   |--- Fare <= 52.28
|   |   |--- Age <= 15.00
|   |   |   |--- class: 1.0
|   |   |--- Age >  15.00
|   |   |   |--- truncated branch of depth 21
|   |--- Fare >  52.28
|   |   |--- Age <= 63.00
|   |   |   |--- truncated branch of depth 10
|   |   |--- Age >  63.00
|   |   |   |--- class: 0.0
|--- Pclass >  2.50
|   |--- Age <= 27.50
|   |   |--- Age <= 26.50
|   |   |   |--- truncated branch of depth 16
|   |   |--- Age >  26.50
|   |   |   |--- class: 1.0
|   |--- Age >  27.50
|   |   |--- Fare <= 8.67
|   |   |   |--- truncated branch of depth 11
|   |   |--- Fare >  8.67
|   |   |   |--- truncated branch of depth 10
```

Random Forest

- Random Forest is an ensemble method that builds multiple decision trees and combines their predictions for improved accuracy and robustness.

Example:

Predicting the genre of a movie based on various features. Random Forest aggregates predictions from multiple decision trees.

How it Works:

The algorithm creates multiple decision trees with different subsets of the data and combines their predictions through averaging or voting.

Random Forest

Real-world Applications:

Predicting species distribution based on environmental factors, Identifying potential customers for a new product, Image recognition, credit scoring, predicting stock prices, etc.

Visualization:

Visual representation involves a forest of decision trees, each contributing to the final prediction.

Decision Trees Versus Random Forest:

- Decision trees are the building blocks of random forests, and a single decision tree might be less accurate but more interpretable.
- Decision trees can be visualized, making them easy to interpret and explain.
- Random forests aggregate predictions from multiple decision trees, providing more robust results.
- Random forests are less prone to overfitting compared to individual decision trees.

Example

Training a model with Random Forest Algorithm:

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load the Titanic dataset from a CSV file
titanic_data = pd.read_csv('titanic_clean.csv')

selected_features = ['Age', 'Pclass', 'Fare']
X = titanic_data[selected_features]
y = titanic_data['Survived']

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

# Create a Random Forest classifier with 100 trees (you can adjust the number of trees as needed)
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the model on the training data
rf_classifier.fit(X_train, y_train)

# Predict survival
y_pred = rf_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.7653631284916201
```

Support Vector Machines (SVM)

- SVM is used for both classification and regression tasks, finding the hyperplane that best separates data points of different classes.

Example:

Classifying whether an email is spam or not based on features like word frequency. SVM finds the optimal hyperplane.

How it Works:

The algorithm aims to find the hyperplane that maximally separates classes in the feature space.

Support Vector Machines (SVM)

Real-world Applications:

Classifying objects in images, Predicting whether a customer will default on a loan, Image classification, text classification, handwriting recognition, etc.

Visualization:

Visual representation involves finding the best-fitting hyperplane that maximizes the margin between classes.

Example

Training a model with SVC algorithm:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the Titanic dataset from a CSV file
titanic_data = pd.read_csv('titanic_clean.csv')

selected_features = ['Age', 'Sex', 'Fare']
X = titanic_data[selected_features]
y = titanic_data['Survived']

# Convert 'Sex' to numerical values (0 for female, 1 for male)
X['Sex'] = X['Sex'].map({'female': 0, 'male': 1})

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

# Create an SVM classifier
clf = SVC()

# Fit the model on the training data
clf.fit(X_train, y_train)

# Predict survival
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.6480446927374302
```

K-Nearest Neighbors (KNN)

- KNN is used for both classification and regression tasks, predicting the label or value based on the majority of its k-nearest neighbors.

Example:

Predicting the genre of a song based on its audio features. KNN considers the genres of the k most similar songs.

How it Works:

The algorithm calculates the distance between data points and predicts based on the majority label or average value of the k-nearest neighbors.

K-Nearest Neighbors (KNN)

Real-world Applications:

Recommending products based on the preferences of similar customers, Predicting patient outcomes based on similar medical histories, Handwriting recognition, Anomaly detection, etc.

Visualization:

Visual representation involves mapping data points in the feature space and determining the majority class among the k-nearest neighbors.

SVM Versus K-Nearest Neighbors:

- SVM aims to find a hyperplane that separates classes, whereas KNN relies on the proximity of data points.
- SVM can handle non-linear relationships through the use of kernel functions.
- KNN relies on the majority class among the k-nearest neighbors, providing a more localized decision.
- KNN can be computationally expensive, especially with large datasets.

Example

Training a model with K-Nearest Neighbors algorithm:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load the Titanic dataset from a CSV file
titanic_data = pd.read_csv('titanic_clean.csv')

selected_features = ['Age', 'Sex', 'Fare']
X = titanic_data[selected_features]
y = titanic_data['Survived']

# Convert 'Sex' to numerical values (0 for female, 1 for male)
X['Sex'] = X['Sex'].map({'female': 0, 'male': 1})

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

# Create a K-Nearest Neighbors classifier with k=3 (you can adjust k as needed)
knn = KNeighborsClassifier(n_neighbors=3)

# Fit the model on the training data
knn.fit(X_train, y_train)

# Predict survival
y_pred = knn.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.7094972067039106
```

Why do we convert categorical variables to a numeric value?

- In machine learning models, it's common to convert categorical variables into numerical values.
- The reason for this conversion is that many machine learning algorithms, including SVM work with numerical data and mathematical operations. They involve mathematical computations like matrix operations, and numerical representations facilitate these operations.
- For binary categorical variables (two unique values, like 'female' and 'male'), it's common to use 0 and 1 for encoding.
- Converting categorical variables to numerical format ensures consistency and uniformity in data representation. It allows for a standardized input format across different features.

Choosing the Right Algorithm

Consider Data Type:

- Choose algorithms based on whether the problem is regression or classification.
- Regression algorithms predict continuous outputs, while classification algorithms predict discrete classes.

Data Size:

- Some algorithms perform better with larger datasets.

Interpretability:

- Consider the interpretability of the model for the given problem.
- Some algorithms, like decision trees, are inherently more interpretable than complex models like neural networks.

Computational Efficiency:

- Assess the computational resources required for training and prediction.
- Certain algorithms, especially deep learning models, can be computationally intensive.

Imbalanced Data

Issue:

- Imbalanced datasets have unequal distribution of classes, where one class significantly outnumbers the others.

Challenges:

- Models may become biased towards the majority class.
- Poor performance on the minority class, impacting real-world applications.

Handling Techniques:

1. Resampling:

- Oversampling: Increase instances of the minority class.
- Undersampling: Decrease instances of the majority class.

2. Synthetic Data Generation:

- Create artificial samples for the minority class (e.g., SMOTE).

3. Weighted Models:

- Assign higher weights to minority class instances during model training.

4. Ensemble Methods:

- Combine predictions from multiple models to improve overall performance.

Real-world imbalanced dataset

- One commonly used real-world imbalanced dataset is the Credit Card Fraud Detection dataset available on Kaggle. This dataset is often used for binary classification tasks to detect fraudulent credit card transactions.
- You can download the dataset directly from Kaggle using the link provided. The dataset contains a mix of numerical and anonymized features, and the target variable is binary, indicating whether a transaction is fraudulent (class 1) or not (class 0).

```
import pandas as pd

# Load Credit Card Fraud Detection dataset
df = pd.read_csv('creditcard.csv')

# Display class distribution
class_distribution = df['Class'].value_counts()
print(class_distribution)
```

```
Class
0    284315
1       492
Name: count, dtype: int64
```

Real-world imbalanced dataset

- For the Credit Card Fraud Detection dataset, one effective technique is to use the SMOTE (Synthetic Minority Over-sampling Technique) for oversampling the minority class.
- SMOTE generates synthetic samples for the minority class, addressing the imbalance and providing the model with a more balanced training set.

SMOTE (Synthetic Minority Over-sampling Technique)

Problem:

- Imbalanced Dataset: In the Credit Card Fraud Detection dataset, fraudulent transactions (minority class) are significantly fewer than non-fraudulent transactions (majority class).

Real-world imbalanced dataset

Solution:

- SMOTE: Synthetically generate new instances for the minority class to balance the dataset.

How it Works:

- For each minority class instance, SMOTE creates synthetic examples by interpolating between that instance and its nearest neighbors.
- These synthetic instances are added to the dataset, making the minority class more prominent.

Benefits:

- Balanced Training Set: SMOTE helps in training the model on a more balanced dataset.
- Improved Model Performance: The model becomes more adept at recognizing patterns in the minority class.

Real-world imbalanced dataset

Implementation with Scikit-learn:

```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

# Assuming X and y are your feature matrix and target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply SMOTE only on the training set
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```