Machine Learning Classification Models

Logistic Regression

What is Logistic Regression?

Logistic Regression is a supervised learning algorithm used for **classification problems**. It predicts the probability that an input belongs to a certain class (e.g., spam or not spam, sick or healthy).

Unlike Linear Regression, which predicts continuous values, **Logistic Regression predicts categorical values (0 or 1, Yes or No, True or False).**

Mathematical Formulation

Logistic Regression uses the **sigmoid function (also called the logistic function)** to convert outputs into probabilities:

P = 1 / (1+e−(b0+b1X1+b2X2+...+bnXn))

- p = Probability of class 1
- b0,b1,...,bn = Model coefficients
- X1,X2,...,XnX_1, X_2, ..., X_nX1,X2,...,Xn = Features

The model predicts:

- **Class 1 (Positive, Yes, Spam, etc.)** if ppp is **greater than 0.5**
- **Class 0 (Negative, No, Not Spam, etc.)** if ppp is **less than 0.5**
-

Example Code: Logistic Regression

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split


# Sample dataset

X = np.array([[20], [25], [30], [35], [40], [45], [50], [55], [60], [65]])  #
Age

y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])  # 0 = No Disease, 1 = Disease


# Train-test split

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


# Train model

model = LogisticRegression()

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Decision Tree: Regression & Classification

A **Decision Tree** is a supervised learning algorithm used for both **classification** and **regression** problems. It works by splitting data into smaller subsets using a tree-like structure, where each node represents a decision based on a feature.

1. Decision Tree for Classification

What is Decision Tree Classification?

- It is used to classify data into different categories (e.g., spam/not spam, pass/fail, disease/no disease).
- It splits the dataset based on feature values, forming a tree where each node represents a decision question.
- The final leaf nodes represent the predicted class (e.g., “Yes” or “No”).

🔹 **How It Works:**

1. Start from the root node (entire dataset).
2. Choose the best feature to split the data based on **Gini Impurity** or **Entropy & Information Gain**.
3. Continue splitting the data at each node until a stopping condition is met (e.g., pure class, max depth reached).
4. Use majority voting at leaf nodes to classify new data points.

Mathematical Formulation for Decision Tree Classifier

1. **Entropy** (Measure of impurity):
   H(S)=−∑pilog2(pi)H(S) = - \sum p_i \log_2(p_i)H(S)=−∑pilog2(pi)

- pip_ipi = Probability of each class in a node

Information Gain (To split the nodes):

IG=H(S)−∑i=1c∣Si∣∣S∣H(Si)IG = H(S) - \sum_{i=1}^{c} \frac{|S_i|}{|S|} H(S_i)IG=H(S)−i=1∑c∣S∣∣Si∣H(Si)

- **H(Si)** = Entropy of each subset after splitting
- **|S|** = Total instances in the current node
- **|Si|** = Instances in each split
- The split that **reduces entropy the most** is chosen.

Example Code: Decision Tree Classifier

```
from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import train_test_split

import numpy as np


# Sample dataset

X = np.array([[20], [25], [30], [35], [40], [45], [50], [55], [60], [65]])

y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])  # 0 = No Disease, 1 = Disease

# Train-test split

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

# Train model

model = DecisionTreeClassifier(max_depth=3)

model.fit(X_train, y_train)

# Predictions

y_pred = model.predict(X_test)
```

Random Forest: Regression & Classification

Random Forest is an **ensemble learning method** that combines multiple decision trees to improve accuracy and reduce overfitting. It is used for both **classification** and **regression** problems.

1. Random Forest for Classification

What is Random Forest Classification?

- It builds **multiple decision trees** using different subsets of data and features.
- Each tree makes a prediction, and the final classification is determined by **majority voting** (most common class).
- Helps to **reduce overfitting** and improves accuracy compared to a single decision tree.

🔹 **How It Works:**

1. Randomly selects subsets of the dataset using **Bootstrap Sampling** (bagging).
2. Trains multiple **Decision Tree classifiers** on different subsets.
3. Each tree gives a prediction, and the **majority class** is chosen as the final output.

Example Code: Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split

import numpy as np
```

```
# Sample dataset

X = np.array([[20], [25], [30], [35], [40], [45], [50], [55], [60], [65]])

y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])  # 0 = No Disease, 1 = Disease

# Train-test split

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

# Train model

model = RandomForestClassifier(n_estimators=100, max_depth=3, random_state=42)

model.fit(X_train, y_train)

# Predictions

y_pred = model.predict(X_test)
```

Key Differences: Decision Tree vs Random Forest

Feature	**Decision Tree**	**Random Forest**
Overfitting	High (prone to overfitting)	Low (reduces overfitting)
Performance	Less accurate	More accurate
Training Speed	Faster	Slower (trains multiple trees)
Interpretability	Easy to interpret	Hard to interpret
Use Case	When you need a quick decision-making model	When accuracy and robustness are more important

Support Vector Machine (SVM): Regression & Classification

Support Vector Machine (SVM) is a powerful algorithm used for **both classification and regression** tasks. It works by finding the best decision boundary (hyperplane) that **maximizes the margin** between different classes in classification and **minimizes error** in regression.

1. SVM for Classification

What is SVM Classification?

- It **separates classes** using a hyperplane in **high-dimensional space**.
- Tries to **maximize the margin** between data points of different classes.
- Uses **support vectors**, which are the critical data points that influence the decision boundary.
- Works well even for **non-linearly separable data** using **kernel tricks** (like RBF, polynomial).

Mathematical Expression

For binary classification:

w⋅x+b=0w \cdot x + b = 0w⋅x+b=0

Where:

- www = weight vector
- xxx = input feature vector
- bbb = bias
- The decision boundary is chosen to maximize the margin between classes.

Example Code: SVM Classifier

```
from sklearn.svm import SVC

from sklearn.model_selection import train_test_split

import numpy as np


# Sample dataset

X = np.array([[20], [25], [30], [35], [40], [45], [50], [55], [60], [65]])

y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])  # 0 = No Disease, 1 = Disease


# Train-test split

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


# Train SVM model

model = SVC(kernel='linear', C=1.0)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Key Difference between: SVM vs SVR :

Features	SVM (Classification)	SVR (Regression)
Output Type	Discrete classes (e.g., 0 or 1)	Continuous values (e.g., 0.5, 3.7)
Decision Boundary	Separates classes	Fits a margin around the data
Kernel Trick	Used to separate nonlinear data	Used to capture complex patterns
Optimization Goal	Maximizes Margin	Minimizes error within ϵ margin
Use Case	Image recognition, fraud detection, disease classification	House price prediction, stock forecasting, salary estimation

XGBoost: Regression & Classification

XGBoost (Extreme Gradient Boosting) is an advanced machine learning algorithm that builds **decision trees sequentially**, improving the model at each step. It is **fast, scalable, and powerful**, often winning machine learning competitions.

1. XGBoost for Classification

What is XGBoost Classification?

- Used for **categorical outputs** (e.g., spam vs. not spam, fraud vs. not fraud).
- Works by **building multiple decision trees** sequentially, improving errors at each step.

- Uses **gradient boosting**, which adjusts weak models into a strong one.
- Supports **regularization** (L1 & L2) to prevent overfitting.

Example Code: XGBoost Classifier

```
import xgboost as xgb

from sklearn.model_selection import train_test_split

from sklearn.datasets import load_breast_cancer


# Load dataset (binary classification example)

data = load_breast_cancer()

X, y = data.data, data.target


# Split dataset

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


# Train XGBoost classifier

model = xgb.XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=3)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Parameters of XGBoost Classifier :

Parameter	**Description**
n_estimators	Number of trees in the model. More trees = better learning but higher computation.
learning_rate	Step size for updating weights. Smaller values prevent overfitting.
max_depth	Depth of each decision tree. Higher depth captures more patterns but may overfit.
objective	Defines the problem type (`binary:logistic` for classification).
subsample	Percentage of data used for training each tree (used for regularization).

Key Differences: XGBoost Classifier vs. Regressor

Feature	**XGBoost Classifier**	**XGBoost Regressor**
Output Type	Categorical (0, 1, 2...)	Continuous (real numbers)
Loss Function	Log Loss, Cross-Entropy	Mean Squared Error (MSE)
Evaluation Metric	Accuracy, F1-score	RMSE, R-squared
Use Case	Fraud detection, disease classification	Stock price prediction, sales forecasting

LightGBM: Regression & Classification

LightGBM (Light Gradient Boosting Machine) is an **optimized gradient boosting framework** designed for **high efficiency, speed, and scalability**. It is **faster than XGBoost** because it uses **leaf-wise splitting** instead of level-wise splitting.

1. LightGBM for Classification

What is LightGBM Classification?

- Used for **categorical outputs** (e.g., spam detection, fraud detection).
- Works similarly to XGBoost but **trains much faster**.
- Handles large datasets well **without much tuning**.
- Supports **categorical feature handling** directly.

Example Code: LightGBM Classifier

```
import lightgbm as lgb

from sklearn.model_selection import train_test_split

from sklearn.datasets import load_breast_cancer


# Load dataset (binary classification example)

data = load_breast_cancer()

X, y = data.data, data.target

# Split dataset

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

```
# Train LightGBM classifier

model = lgb.LGBMClassifier(n_estimators=100, learning_rate=0.1, max_depth=-1)

model.fit(X_train, y_train)

# Predictions

y_pred = model.predict(X_test)
```

Parameters of LightGBM Classifier

Parameter	Description
`n_estimators`	Number of trees in the model. More trees = better learning but higher computation.
`learning_rate`	Step size for updating weights. Smaller values prevent overfitting.
`max_depth`	Maximum depth of each decision tree (`-1` means no limit).
`num_leaves`	Number of leaves per tree (higher values capture more patterns but may overfit).
`boosting_type`	Type of boosting (`gbdt` for standard gradient boosting, `dart` for dropout boosting).

Key Differences: LightGBM vs. XGBoost

Feature	LightGBM	XGBoost
Speed	Faster	Slower
Memory Usage	Lower	Higher
Splitting Method	Leaf-wise	Level-wise
Performance on Large Datasets	Better	Good but slower
Categorical Features	Can handle directly	Needs encoding

K-Nearest Neighbors (KNN)

KNN is a **non-parametric, instance-based learning algorithm** that classifies data points based on the majority vote of their nearest neighbors. It can be used for both **classification** and **regression**.

KNN for Classification

- Assigns a class based on the **majority vote** of the `k` nearest neighbors.
- Works well for small datasets but **slows down** on large ones.

Example Code: KNN Classifier

```
from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

from sklearn.datasets import load_iris


# Load dataset

data = load_iris()

X, y = data.data, data.target


# Split dataset

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


# Train KNN Classifier

model = KNeighborsClassifier(n_neighbors=5)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Parameters of KNN Classifier

Parameter	Description
`n_neighbors`	Number of nearest neighbors to consider.
`weights`	How neighbors influence predictions (`uniform` = equal, `distance` = closer points matter more).
`metric`	Distance metric (`euclidean, manhattan, minkowski`).

Naïve Bayes

Naïve Bayes (NB) is a **probabilistic machine learning algorithm** based on **Bayes' Theorem**. It is used for **classification tasks** and assumes that features are **independent**

There are **three types** of Naïve Bayes classifiers:

1. **Gaussian Naïve Bayes** – Used for **continuous numerical data** (assumes normal distribution).
2. **Multinomial Naïve Bayes** – Used for **text classification** and **word frequency data**.
3. **Bernoulli Naïve Bayes** – Used for **binary feature data** (0 or 1 values).

1. Gaussian Naïve Bayes (GNB)

Gaussian Naïve Bayes assumes that **continuous features** follow a **normal (Gaussian) distribution**.

Example Code: Gaussian Naïve Bayes for Classification

```
from sklearn.naive_bayes import GaussianNB

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

from sklearn.datasets import load_wine


# Load dataset

data = load_wine()

X, y = data.data, data.target


# Split dataset

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


# Train Gaussian Naïve Bayes model

model = GaussianNB()

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

When to Use Gaussian Naïve Bayes?

✅ When features are **continuous numerical values**.
✅ When data is **normally distributed**.

2. Multinomial Naïve Bayes (MNB)

Multinomial Naïve Bayes is used for **discrete count data**, especially in **text classification** (e.g., spam detection, sentiment analysis).

Example Code: Multinomial Naïve Bayes for Text Classification

```
from sklearn.naive_bayes import MultinomialNB

from sklearn.feature_extraction.text import CountVectorizer

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Sample dataset (spam detection)

text_data = [

    "Win a free iPhone now",  # Spam

    "Congratulations! You won a lottery",  # Spam

    "Meeting at 10 AM",  # Not spam

    "Lunch at office today?",  # Not spam

    "You have been selected for a prize",  # Spam

    "Project deadline extended to next week",  # Not spam]

labels = [1, 1, 0, 0, 1, 0]  # 1 = Spam, 0 = Not Spam
```

```
# Convert text to numerical feature vectors

vectorizer = CountVectorizer()

X = vectorizer.fit_transform(text_data)


# Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2,
random_state=42)


# Train Multinomial Naïve Bayes model

model = MultinomialNB()

model.fit(X_train, y_train)

# Predictions

y_pred = model.predict(X_test)
```

When to Use Multinomial Naïve Bayes?

✅ **Text classification** (spam detection, sentiment analysis).
✅ **Word frequency data** (bag-of-words representation).
✅ Features represent **counts of occurrences**.

3. Bernoulli Naïve Bayes (BNB)

Bernoulli Naïve Bayes is used for **binary feature data** (0s and 1s), such as **word presence vs. absence** in text classification.

Example Code: Bernoulli Naïve Bayes for Text Classification

```python
from sklearn.naive_bayes import BernoulliNB

from sklearn.feature_extraction.text import CountVectorizer

from sklearn.model_selection import train_test_split


# Sample dataset (spam detection)

text_data = [

    "Win a free iPhone now",

    "Congratulations! You won a lottery",

    "Meeting at 10 AM",

    "Lunch at office today?",

    "You have been selected for a prize",

    "Project deadline extended to next week",  ]

labels = [1, 1, 0, 0, 1, 0]  # 1 = Spam, 0 = Not Spam

# Convert text to binary features (word presence)

vectorizer = CountVectorizer(binary=True)

X = vectorizer.fit_transform(text_data)

# Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2,
random_state=42)

# Train Bernoulli Naïve Bayes model

model = BernoulliNB()

model.fit(X_train, y_train)
```

```
# Predictions

y_pred = model.predict(X_test)
```

When to Use Bernoulli Naïve Bayes?

✅ **Binary feature data** (0 or 1, word presence or absence).
✅ **Text classification** where features are **binary** instead of count-based.

Comparison of Naïve Bayes Types

Type	Use Case	Works Best When
Gaussian Naïve Bayes	Numerical data classification	Features are **continuous & normally distributed**
Multinomial Naïve Bayes	Text classification	Features are **word frequencies (count data)**
Bernoulli Naïve Bayes	Binary feature classification	Features are **binary (0 or 1, word presence)**

Final Thoughts

- **Use Gaussian NB** if your data is numerical and continuous.
- **Use Multinomial NB** for text data with word frequencies.
- **Use Bernoulli NB** if text features are binary (presence/absence of words).