Logistic Regression in Machine Learning

Type	@datasciencebrain

Introduction:

Logistic regression is a supervised learning algorithm predominantly used for binary classification problems, though it can be extended to multiclass classification. Despite its name, logistic regression is essentially a classification method, not a regression algorithm, and is widely employed in fields like medicine, finance, social sciences, and more.

Example:

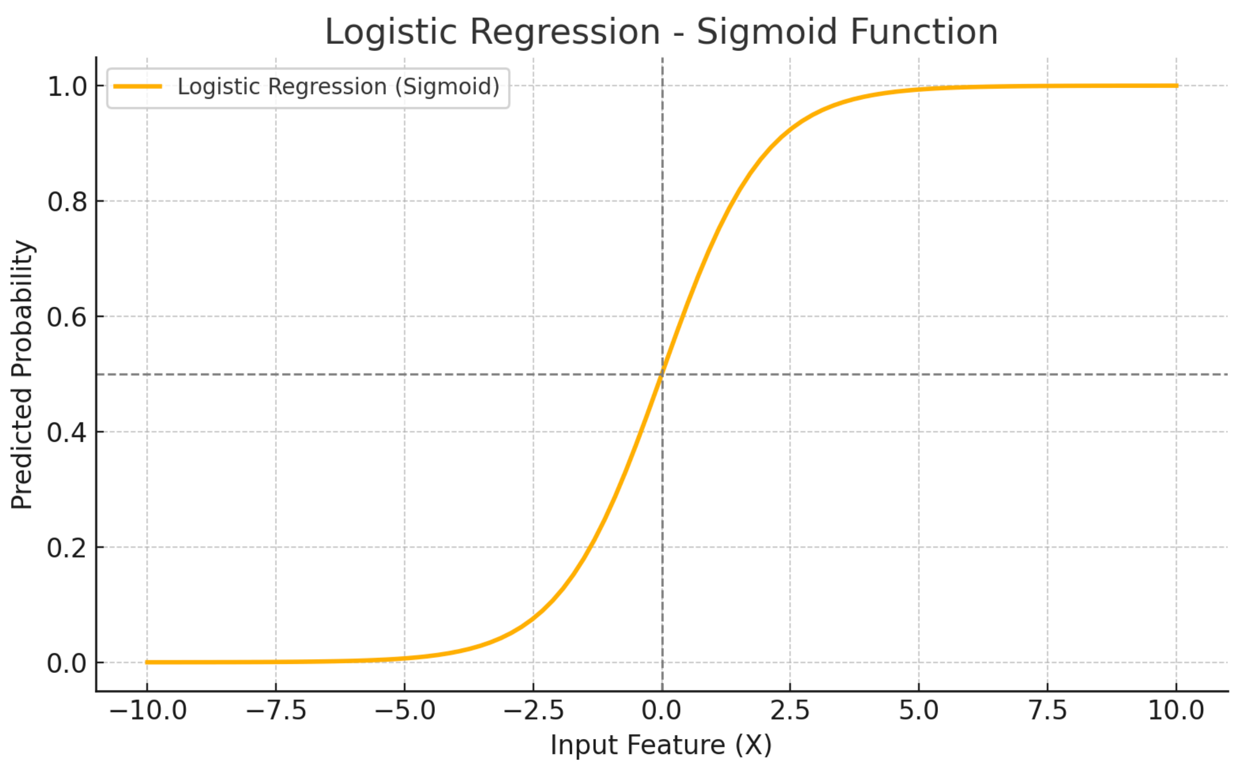
```
import numpy as np
import matplotlib.pyplot as plt

# Generate sample data
np.random.seed(42)
X = np.linspace(-10, 10, 100)
y = 1 / (1 + np.exp(-X))

# Plotting logistic regression (Sigmoid Curve)
plt.figure(figsize=(8, 5))
plt.plot(X, y, label="Logistic Regression (Sigmoid)", linewidth=2)
plt.axhline(0.5, color='grey', linestyle='--', linewidth=1)
plt.axvline(0, color='grey', linestyle='--', linewidth=1)

# Labels and Title
```

```
plt.title("Logistic Regression - Sigmoid Function")
plt.xlabel("Input Feature (X)")
plt.ylabel("Predicted Probability")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



2. Working Principle:

Logistic regression predicts the probability of a binary outcome (0 or 1) by fitting data to a logistic function (sigmoid curve). It models the relationship between a set of independent variables (features) and the probability of a particular outcome.

3. Mathematical Formulation:

The logistic function, also known as the sigmoid function, is represented as:

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

where z is a linear combination of input features:

$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$

- β0 is the intercept
- β1,β2,...,βn are coefficients for input variables x1,x2,...,xn
- e is Euler's number (approx. 2.71828)

4. Decision Boundary:

Logistic regression predicts probabilities, and classification is performed based on a threshold (typically 0.5). Values above the threshold are classified as class 1, and those below as class 0. The decision boundary is defined by setting σ(z)=0.5, implying z = 0.

5. Cost Function:
Logistic regression uses the log loss (cross-entropy) function as the cost function:

$J(\beta) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(h_\beta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\beta(x^{(i)}))]$

where:

- m is the number of training examples
- $y^{(i)}$ is the actual class (0 or 1)
- $h_\beta(x^{(i)})$ is the predicted probability

6. Optimization Method (Gradient Descent):

Coefficients β are optimized using gradient descent, an iterative optimization algorithm. The coefficients are updated iteratively to minimize the cost function:

$\beta_j := \beta_j - \alpha \frac{\partial}{\partial \beta_j} J(\beta)$

where:

- α is the learning rate
- Partial derivative is computed for each βj

7. Assumptions of Logistic Regression:

- Observations must be independent.
- Minimal or no multicollinearity among features.
- Linearity between the log odds (logit) and input variables.
- Requires a large sample size to produce stable results.

8. Evaluation Metrics:

- **Confusion Matrix**
- **Accuracy:**

  $\frac{TP+TN}{TP+TN+FP+FN}$
- **Precision:**

  $\frac{TP}{TP+FP}$
- **Recall (Sensitivity):**

  $\frac{TP}{TP+FN}$
- **F1 Score:** Harmonic mean of precision and recall
- **ROC-AUC:** Receiver Operating Characteristic Curve and Area Under the Curve for evaluating model performance across all thresholds.

9. Regularization:

Regularization techniques (L1 or L2) are employed to prevent overfitting:

- **L1 Regularization (Lasso):** Adds absolute values of the coefficients as penalty, driving some coefficients to zero.
- **L2 Regularization (Ridge):** Adds squared values of coefficients as penalty, shrinking coefficients without eliminating them completely.

10. Advantages:

- Simple and interpretable model.
- Computationally efficient.
- Effective for binary and multiclass problems (via One-vs-Rest or Multinomial logistic regression).
- Performs well with linearly separable data.

11. Limitations:

- Assumes linearity between log-odds and predictors.
- Sensitive to outliers.
- Struggles with highly correlated features (multicollinearity).
- Not suitable for capturing complex, non-linear relationships without feature transformations.

12. Extensions of Logistic Regression:

- **Multinomial Logistic Regression:** Handles classification tasks with more than two classes.
- **Ordinal Logistic Regression:** Used when the dependent variable has an ordered category.

13. Practical Applications:

- Medical diagnosis (e.g., cancer detection).
- Credit scoring (loan approval).
- Marketing (predicting customer churn).
- Fraud detection (financial transactions).

14. Real-World Project: Breast Cancer Classification using Logistic Regression:

Here's a complete real-world example of implementing Logistic Regression using a built-in dataset from scikit-learn (Breast Cancer Wisconsin dataset). The project includes proper preprocessing, pipelines, model evaluation, and interpretation of results:

Step 1: Importing Necessary Libraries and Dataset

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix, accuracy_
score, roc_auc_score, roc_curve
import matplotlib.pyplot as plt
```

Step 2: Loading and Understanding the Dataset

```
# Load dataset
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
```

```
y = pd.Series(data.target)

# Display dataset information
print(X.info())
print(X.head())
print("Target Distribution:\n", y.value_counts())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 30 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
```

```
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
dtypes: float64(30)
memory usage: 133.5 KB
None
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  ...  worst radius  worst texture  worst perimeter  \
0                 0.07871  ...         25.38          17.33           184.60
1                 0.05667  ...         24.99          23.41           158.80
2                 0.05999  ...         23.57          25.53           152.50
3                 0.09744  ...         14.91          26.50            98.87
4                 0.05883  ...         22.54          16.67           152.20

   worst area  worst smoothness  worst compactness  worst concavity  \
0      2019.0            0.1622             0.6656           0.7119
1      1956.0            0.1238             0.1866           0.2416
2      1709.0            0.1444             0.4245           0.4504
3       567.7            0.2098             0.8663           0.6869
```

```
4      1575.0          0.1374           0.2050          0.4000

   worst concave points  worst symmetry  worst fractal dimension
0                0.2654         0.4601                  0.11890
1                0.1860         0.2750                  0.08902
2                0.2430         0.3613                  0.08758
3                0.2575         0.6638                  0.17300
4                0.1625         0.2364                  0.07678

[5 rows x 30 columns]
Target Distribution:
 1    357
0    212
Name: count, dtype: int64
```

Step 3: Splitting Dataset into Training and Test Sets

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

Step 4: Creating a Pipeline (Data Scaling & Logistic Regression)

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('logreg', LogisticRegression(solver='liblinear', random_state=42))
])
```

Step 5: Hyperparameter Tuning using GridSearchCV

```
param_grid = {
    'logreg__C': [0.01, 0.1, 1, 10, 100],
    'logreg__penalty': ['l1', 'l2']
}
```

```
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

print("Best parameters:", grid_search.best_params_)
print("Best cross-validation accuracy:", grid_search.best_score_)
```

Step 6: Evaluating Model Performance on Test Set

```
# Best estimator from Grid Search
best_model = grid_search.best_estimator_

# Predictions
y_pred = best_model.predict(X_test)
y_prob = best_model.predict_proba(X_test)[:, 1]

# Accuracy and Classification Report
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy on test set: {accuracy:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:\n", cm)
```

```
Best parameters: {'logreg__C': 0.1, 'logreg__penalty': 'l2'}
Best cross-validation accuracy: 0.9802197802197803
Accuracy on test set: 0.9825

Classification Report:
            precision    recall  f1-score   support

        0      0.98      0.98      0.98        42
        1      0.99      0.99      0.99        72
```

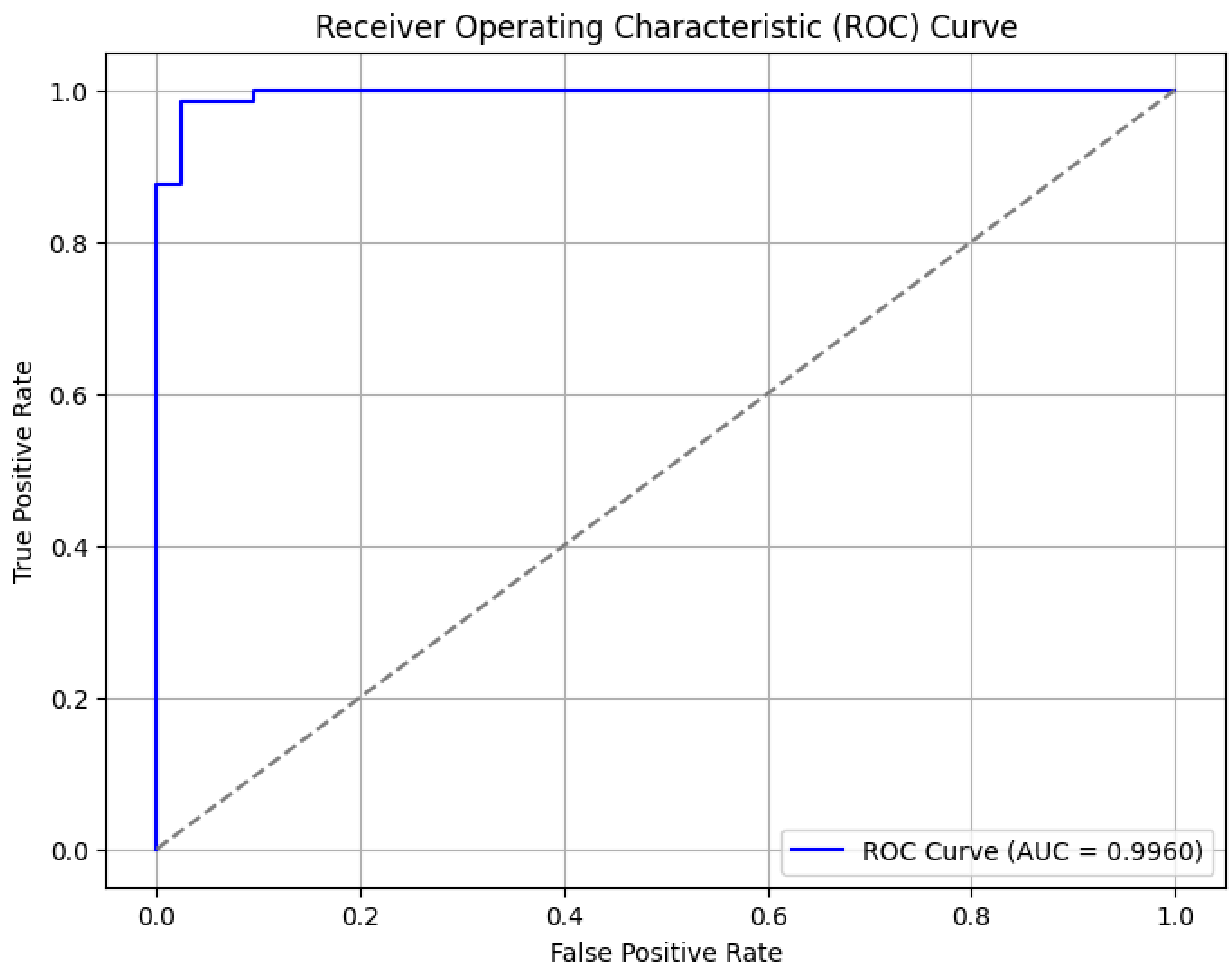
```
    accuracy                           0.98       114
   macro avg       0.98      0.98      0.98       114
weighted avg       0.98      0.98      0.98       114


Confusion Matrix:
 [[41  1]
 [ 1 71]]
```

Step 7: ROC Curve and AUC

```
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = roc_auc_score(y_test, y_prob)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC = {roc_auc:.4f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```



Step 8: Interpretation of Results

- **Accuracy**: Represents the proportion of correctly classified observations.
- **Precision**: Indicates the proportion of positive identifications that were actually correct.
- **Recall (Sensitivity)**: Shows the proportion of actual positives correctly identified.
- **F1-score**: Harmonic mean of precision and recall; useful metric for imbalanced datasets.
- **ROC-AUC**: High ROC-AUC indicates excellent performance.