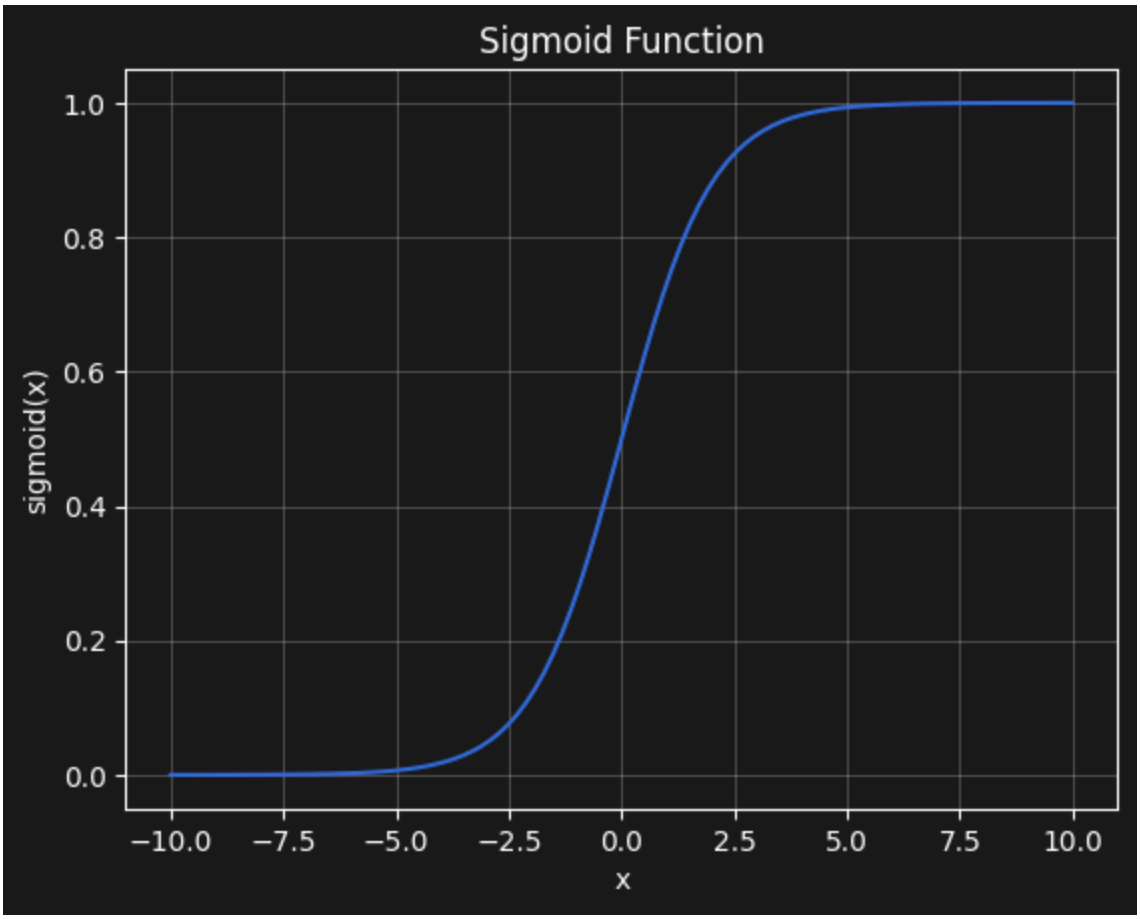
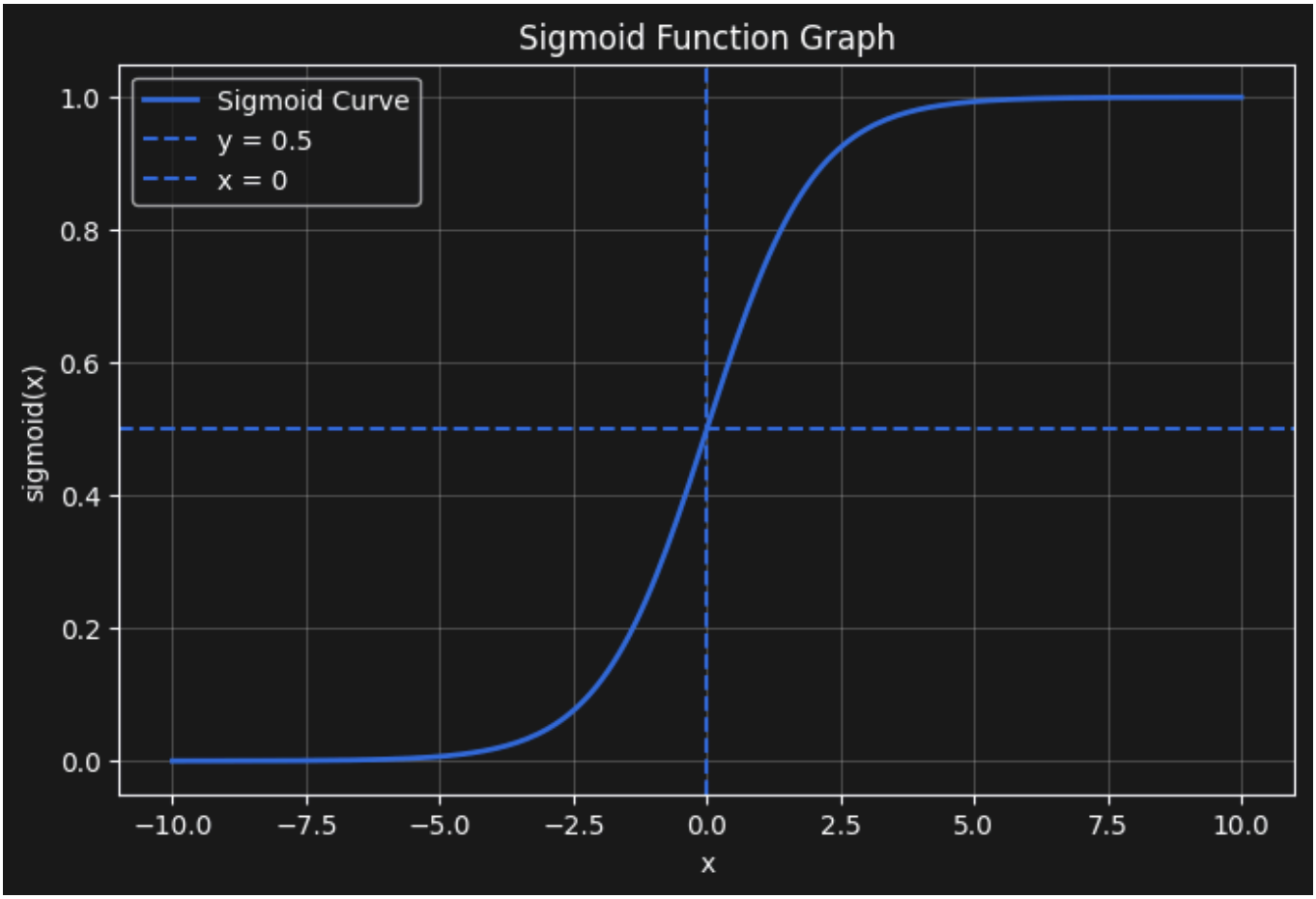
```
import numpy as np
import matplotlib.pyplot as plt

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Generate x values
x = np.linspace(-10, 10, 100)

# Generate y values using sigmoid function
y = sigmoid(x)

# Create the graph
plt.plot(x, y)

# Add title and labels
plt.title("Sigmoid Function")
plt.xlabel("x")
plt.ylabel("sigmoid(x)")

# Add grid
plt.grid(True)

# Show the graph
plt.show()
```



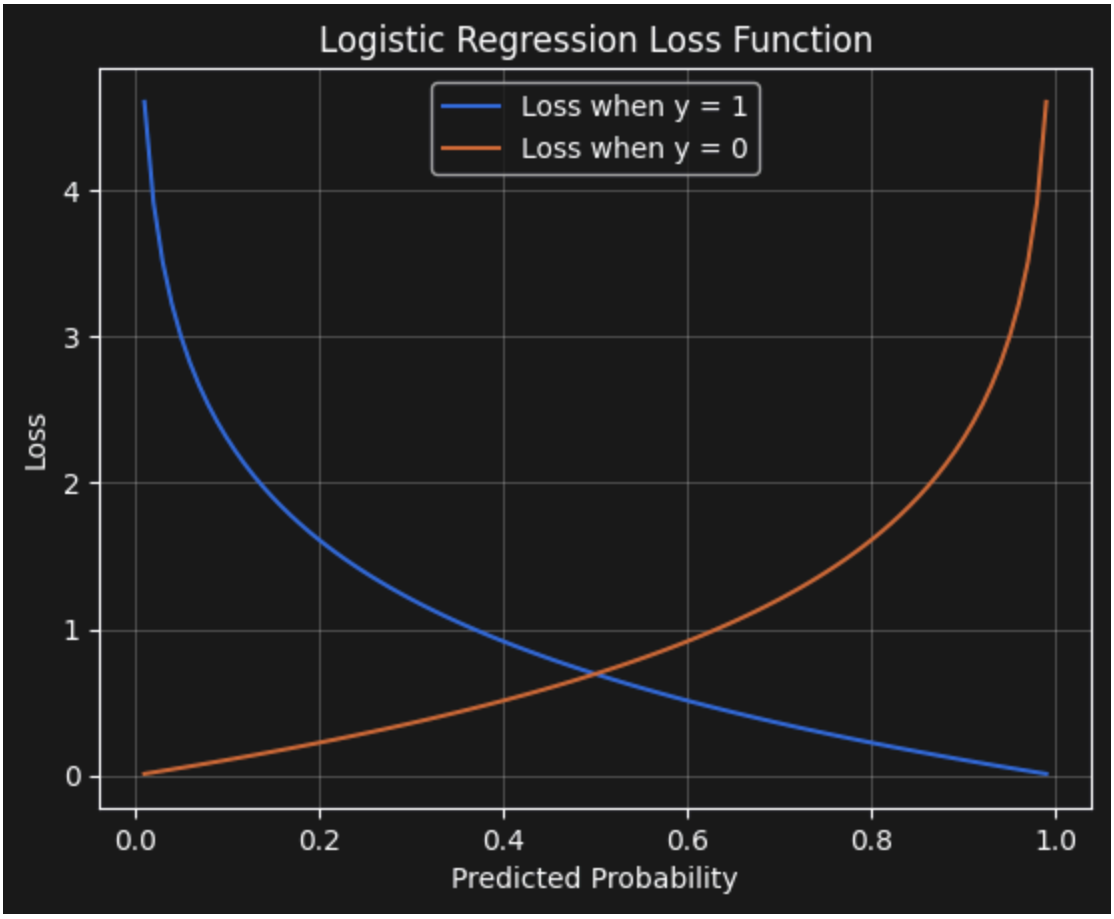

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.linspace(-10, 10, 200)
y = sigmoid(x)

plt.figure(figsize=(8, 5))

plt.plot(x, y, linewidth=2, label="Sigmoid Curve")

plt.axhline(y=0.5, linestyle="--", label="y = 0.5")
plt.axvline(x=0, linestyle="--", label="x = 0")

plt.title("Sigmoid Function Graph")
plt.xlabel("x")
plt.ylabel("sigmoid(x)")

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

```
plt.show()
```



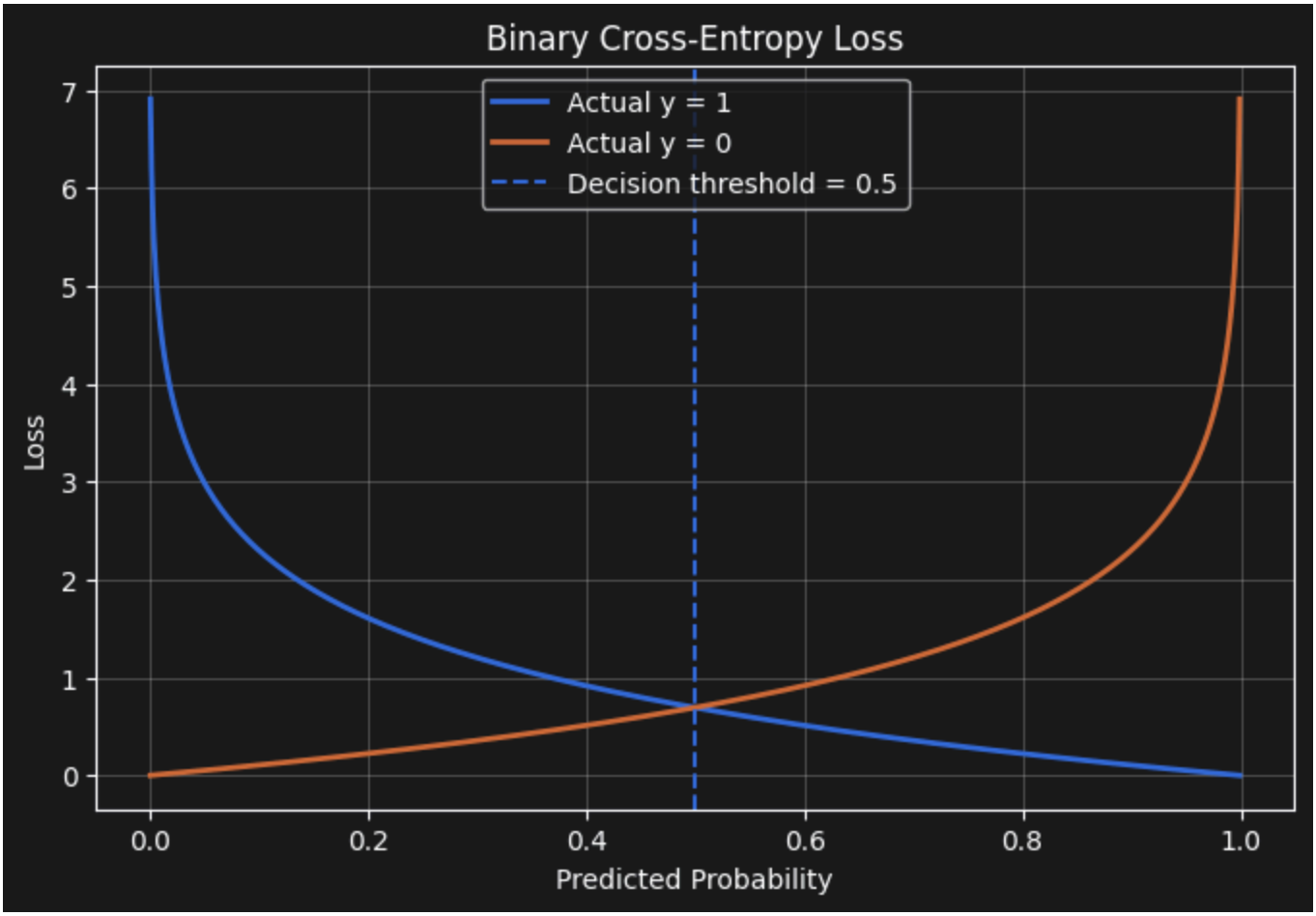

```
import numpy as np
import matplotlib.pyplot as plt

# Predicted probabilities from 0.01 to 0.99
y_pred = np.linspace(0.01, 0.99, 100)

# Loss when actual class y = 1
loss_y1 = -np.log(y_pred)

# Loss when actual class y = 0
loss_y0 = -np.log(1 - y_pred)

# Plot both losses
plt.plot(y_pred, loss_y1, label="Loss when y = 1")
plt.plot(y_pred, loss_y0, label="Loss when y = 0")

# Add title and labels
plt.title("Logistic Regression Loss Function")
plt.xlabel("Predicted Probability")
plt.ylabel("Loss")
```

```
# Add grid and legend
plt.grid(True)
plt.legend()

# Show graph
plt.show()
```




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

# Avoid 0 and 1 because log(0) is undefined
y_pred = np.linspace(0.001, 0.999, 1000)

# Loss formulas
loss_y1 = -np.log(y_pred)
loss_y0 = -np.log(1 - y_pred)

plt.figure(figsize=(8, 5))

plt.plot(y_pred, loss_y1, linewidth=2, label="Actual y = 1")
plt.plot(y_pred, loss_y0, linewidth=2, label="Actual y = 0")
```

```
plt.axvline(x=0.5, linestyle="--", label="Decision threshold = 0.5")

plt.title("Binary Cross-Entropy Loss")
plt.xlabel("Predicted Probability")
plt.ylabel("Loss")

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




```
import numpy as np

# Example predictions
predictions = [0.01, 0.1, 0.5, 0.9, 0.99]

print("When actual y = 1")
for p in predictions:
    loss = -np.log(p)
    print(f"Prediction: {p}, Loss: {loss:.4f}")

print("\nWhen actual y = 0")
for p in predictions:
```

```
    loss = -np.log(1 - p)
    print(f"Prediction: {p}, Loss: {loss:.4f}")
```

```
import numpy as np


class LogisticRegressionScratch:
    def __init__(self, learning_rate=0.01, iterations=1000):
        self.learning_rate = learning_rate
        self.iterations = iterations
        self.weights = None
        self.bias = None
        self.cost_history = []

    def sigmoid(self, z):
        """
        Applies the sigmoid function.

        Parameters:
        z: Linear combination of inputs and weights

        Returns:
        Probability values between 0 and 1
        """
        return 1 / (1 + np.exp(-z))

    def compute_cost(self, y, y_pred):
        """
        Computes binary cross-entropy loss.

        Parameters:
        y: Actual labels
        y_pred: Predicted probabilities

        Returns:
        Average cost
        """
        m = len(y)

        # Small value added to avoid log(0)
        epsilon = 1e-9

        cost = -(1 / m) * np.sum(
            y * np.log(y_pred + epsilon) +
            (1 - y) * np.log(1 - y_pred + epsilon)
```

```
        )

        return cost

    def fit(self, X, y):
        """
        Trains the logistic regression model.

        Parameters:
        X: Training feature matrix
        y: Training labels
        """
        n_samples, n_features = X.shape

        # Initialize parameters
        self.weights = np.zeros(n_features)
        self.bias = 0

        for i in range(self.iterations):
            # Step 1: Linear model
            linear_output = np.dot(X, self.weights) + self.bias

            # Step 2: Apply sigmoid
            y_pred = self.sigmoid(linear_output)

            # Step 3: Compute cost
            cost = self.compute_cost(y, y_pred)
            self.cost_history.append(cost)

            # Step 4: Compute gradients
            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y))
            db = (1 / n_samples) * np.sum(y_pred - y)

            # Step 5: Update parameters
            self.weights = self.weights - self.learning_rate * dw
            self.bias = self.bias - self.learning_rate * db

    def predict_proba(self, X):
        """
        Predicts probabilities for input data.

        Parameters:
        X: Feature matrix

        Returns:
        Predicted probabilities
```

```
        """
        linear_output = np.dot(X, self.weights) + self.bias
        return self.sigmoid(linear_output)

    def predict(self, X):
        """
        Converts probabilities into class labels.

        Parameters:
        X: Feature matrix

        Returns:
        Predicted class labels
        """
        probabilities = self.predict_proba(X)
        return np.array([1 if p >= 0.5 else 0 for p in probabilities])
```

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification

# Create synthetic classification data
X, y = make_classification(
    n_samples=1000,
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    random_state=48
)

plt.figure(figsize=(8, 6))

plt.scatter(
    X[:, 0],
    X[:, 1],
    c=y,
    cmap="coolwarm",
    alpha=0.7,
    edgecolor="k"
)

plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("Synthetic Classification Dataset")
plt.colorbar(label="Class")
```

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

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
# Split into training and testing data
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42
)

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

# Create model
model = LogisticRegressionScratch(
    learning_rate=0.01,
    iterations=2000
)

# Train model
model.fit(X_train, y_train)

# Predict test data
y_pred = model.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
print("Weights:", model.weights)
print("Bias:", model.bias)
```

```
import numpy as np

X = np.array([
    [10, 100],
    [20, 200],
```

```
    [30, 300],
    [40, 400],
    [50, 500]
])

mean = np.mean(X, axis=0)
std = np.std(X, axis=0)

X_scaled = (X - mean) / std

print("Mean:")
print(mean)

print("\nStandard Deviation:")
print(std)

print("\nScaled Data:")
print(X_scaled)
```

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

X = np.array([
    [10, 100],
    [20, 200],
    [30, 300],
    [40, 400],
    [50, 500]
])

mean = np.mean(X, axis=0)
std = np.std(X, axis=0)

X_scaled = (X - mean) / std

# Graph original data
plt.figure(figsize=(8, 4))
plt.plot(X[:, 0], label="Feature 1: 10 to 50", marker="o")
plt.plot(X[:, 1], label="Feature 2: 100 to 500", marker="o")
plt.title("Original Data")
plt.xlabel("Row index")
plt.ylabel("Value")
plt.legend()
plt.grid(True)
plt.show()
```

```
# Graph scaled data
plt.figure(figsize=(8, 4))
plt.plot(X_scaled[:, 0], label="Scaled Feature 1", marker="o")
plt.plot(X_scaled[:, 1], label="Scaled Feature 2", marker="o")
plt.title("Scaled Data")
plt.xlabel("Row index")
plt.ylabel("Standardized value")
plt.legend()
plt.grid(True)
plt.show()
```

```
import pandas as pd

results = pd.DataFrame({
    "Actual y": y_test,
    "Predicted y": y_pred
})

results["Result"] = results.apply(
    lambda row: "Correct" if row["Actual y"] == row["Predicted y"] else
"Wrong",
    axis=1
)

print(results.head(20))
```

```
# Add 4 condition labels
def condition(row):
    if row["Actual y"] == 0 and row["Predicted y"] == 0:
        return "TN: Actual 0, Predicted 0"
    elif row["Actual y"] == 0 and row["Predicted y"] == 1:
        return "FP: Actual 0, Predicted 1"
    elif row["Actual y"] == 1 and row["Predicted y"] == 0:
        return "FN: Actual 1, Predicted 0"
    elif row["Actual y"] == 1 and row["Predicted y"] == 1:
        return "TP: Actual 1, Predicted 1"

results["Condition"] = results.apply(condition, axis=1)

print(results.head(20))
```

```
condition_counts = results["Condition"].value_counts()
```

```
print("\n4 Conditions Count:")
print(condition_counts)
```

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test,y_pred)

cm_table = pd.DataFrame(
    cm,
    index=["Actual 0", "Actual 1"],
    columns=["Predicted 0", "Predicted 1"]
)

print(cm_table)
```

```
TN, FP, FN, TP = cm.ravel()

print(f"Actual 0 and Predicted 0: {TN} correct negative predictions")
print(f"Actual 0 but Predicted 1: {FP} false positive predictions")
print(f"Actual 1 but Predicted 0: {FN} false negative predictions")
print(f"Actual 1 and Predicted 1: {TP} correct positive predictions")
```

```
TP/(TP+FP)
TP/(TP+FN)
```

```
from sklearn.metrics import precision_score, recall_score

precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

print("Precision:", precision)
print("Recall:", recall)
```