Nithin S
221IT085

IT350 Lab Assignment 2 : Naive Bayes Classifier

Implement the Naive Bayes classifier using the IRIS and HEART(SPECT dataset) datasets. Implement k-fold cross-validation with k=10. Compute the correctly classified instances, incorrectly classified instances; root mean squared error, relative absolute error, True positive rate, False positive rate, Confusion matrix and Kappa score. Display the evaluation metrics for each fold separately and then print all folds' final average evaluation metrics.
Please do not use built-in functions.

IRIS Flower

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target

class GaussianNaiveBayes:
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.n_classes = len(self.classes)
        self.n_features = X.shape[1]

        # Calculate prior probabilities
        self.priors = np.zeros(self.n_classes)
        for i, c in enumerate(self.classes):
            self.priors[i] = np.mean(y == c)

        # Calculate mean and variance for each feature in each class
        self.means = np.zeros((self.n_classes, self.n_features))
        self.vars = np.zeros((self.n_classes, self.n_features))

        for i, c in enumerate(self.classes):
            X_c = X[y == c]
            self.means[i, :] = X_c.mean(axis=0)
            self.vars[i, :] = X_c.var(axis=0)

    def _calculate_likelihood(self, x, mean, var):
        eps = 1e-10  # To avoid division by zero
        coef = 1.0 / np.sqrt(2.0 * np.pi * var + eps)
        exponent = -0.5 * ((x - mean) ** 2) / (var + eps)
        return coef * np.exp(exponent)

    def predict(self, X):
        y_pred = np.zeros(X.shape[0])

        for i, x in enumerate(X):
            posteriors = []

            # Calculate posterior probability for each class
            for c in range(self.n_classes):
                prior = np.log(self.priors[c])
```

```python
            # Calculate posterior probability for each class
            for c in range(self.n_classes):
                prior = np.log(self.priors[c])
                likelihood = np.sum(np.log(self._calculate_likelihood(x, self.means[c, :], self.vars[c, :])))
                posterior = prior + likelihood
                posteriors.append(posterior)

            # Select class with highest posterior probability
            y_pred[i] = self.classes[np.argmax(posteriors)]

        return y_pred

def k_fold_split(X, y, k=10):
    fold_size = len(X) // k
    X_folds = []
    y_folds = []

    indices = np.random.permutation(len(X))
    for i in range(k):
        start_idx = i * fold_size
        end_idx = start_idx + fold_size if i < k-1 else len(X)

        fold_indices = indices[start_idx:end_idx]
        X_folds.append(X[fold_indices])
        y_folds.append(y[fold_indices])

    return X_folds, y_folds

def calculate_metrics(y_true, y_pred, n_classes=3):
    conf_matrix = np.zeros((n_classes, n_classes))
    for i in range(len(y_true)):
        conf_matrix[int(y_true[i])][int(y_pred[i])] += 1

    correct = np.sum(y_true == y_pred)
    incorrect = len(y_true) - correct
    accuracy = correct / len(y_true)

    tpr = np.zeros(n_classes)
    fpr = np.zeros(n_classes)

    for i in range(n_classes):
        tp = conf_matrix[i][i]
        fn = np.sum(conf_matrix[i]) - tp
```

```python
        conf_matrix[int(y_true[i])][int(y_pred[i])] += 1

    correct = np.sum(y_true == y_pred)
    incorrect = len(y_true) - correct
    accuracy = correct / len(y_true)

    tpr = np.zeros(n_classes)
    fpr = np.zeros(n_classes)

    for i in range(n_classes):
        tp = conf_matrix[i][i]
        fn = np.sum(conf_matrix[i]) - tp
        fp = np.sum(conf_matrix[:, i]) - tp
        tn = np.sum(conf_matrix) - tp - fp - fn

        tpr[i] = tp / (tp + fn) if (tp + fn) > 0 else 0
        fpr[i] = fp / (fp + tn) if (fp + tn) > 0 else 0


    rmse = np.sqrt(np.mean((y_true - y_pred) ** 2))
    mae = np.mean(np.abs(y_true - y_pred))
    baseline_mae = np.mean(np.abs(y_true - np.mean(y_true)))
    rae = mae / baseline_mae if baseline_mae != 0 else 0

    expected_accuracy = 0
    for i in range(n_classes):
        row_sum = np.sum(conf_matrix[i])
        col_sum = np.sum(conf_matrix[:, i])
        expected_accuracy += (row_sum * col_sum) / np.sum(conf_matrix)**2

    kappa = (accuracy - expected_accuracy) / (1 - expected_accuracy)

    return {
        'accuracy': accuracy,
        'correct': correct,
        'incorrect': incorrect,
        'rmse': rmse,
        'rae': rae,
        'tpr': np.mean(tpr),
        'fpr': np.mean(fpr),
        'kappa': kappa,
        'confusion_matrix': conf_matrix
    }
```

```python
X_folds, y_folds = k_fold_split(X, y, k)
metrics_per_fold = []

for fold in range(k):
    X_test = X_folds[fold]
    y_test = y_folds[fold]

    X_train = np.concatenate([X_folds[i] for i in range(k) if i != fold])
    y_train = np.concatenate([y_folds[i] for i in range(k) if i != fold])

    nb = GaussianNaiveBayes()
    nb.fit(X_train, y_train)
    y_pred = nb.predict(X_test)

    fold_metrics = calculate_metrics(y_test, y_pred)
    metrics_per_fold.append(fold_metrics)

    print(f"\nFold {fold + 1} Results:")
    print(f"Correctly Classified Instances: {fold_metrics['correct']}")
    print(f"Incorrectly Classified Instances: {fold_metrics['incorrect']}")
    print(f"Accuracy: {fold_metrics['accuracy']:.4f}")
    print(f"Root Mean Squared Error: {fold_metrics['rmse']:.4f}")
    print(f"Relative Absolute Error: {fold_metrics['rae']:.4f}")
    print(f"True Positive Rate: {fold_metrics['tpr']:.4f}")
    print(f"False Positive Rate: {fold_metrics['fpr']:.4f}")
    print(f"Kappa Score: {fold_metrics['kappa']:.4f}")
    print("\nConfusion Matrix:")
    print(fold_metrics['confusion_matrix'])

print("\nAverage Metrics Across All Folds:")
avg_metrics = {
    'accuracy': np.mean([m['accuracy'] for m in metrics_per_fold]),
    'rmse': np.mean([m['rmse'] for m in metrics_per_fold]),
    'rae': np.mean([m['rae'] for m in metrics_per_fold]),
    'tpr': np.mean([m['tpr'] for m in metrics_per_fold]),
    'fpr': np.mean([m['fpr'] for m in metrics_per_fold]),
    'kappa': np.mean([m['kappa'] for m in metrics_per_fold])
}

print(f"Average Accuracy: {avg_metrics['accuracy']:.4f}")
print(f"Average RMSE: {avg_metrics['rmse']:.4f}")
print(f"Average RAE: {avg_metrics['rae']:.4f}")
print(f"Average TPR: {avg_metrics['tpr']:.4f}")
```

```
Fold 1 Results:
Correctly Classified Instances: 14
Incorrectly Classified Instances: 1
Accuracy: 0.9333
Root Mean Squared Error: 0.2582
Relative Absolute Error: 0.0893
True Positive Rate: 0.9167
False Positive Rate: 0.0370
Kappa Score: 0.8973

Confusion Matrix:
[[5. 0. 0.]
 [0. 3. 1.]
 [0. 0. 6.]]

Fold 2 Results:
Correctly Classified Instances: 15
Incorrectly Classified Instances: 0
Accuracy: 1.0000
Root Mean Squared Error: 0.0000
Relative Absolute Error: 0.0000
True Positive Rate: 1.0000
False Positive Rate: 0.0000
Kappa Score: 1.0000

Confusion Matrix:
[[6. 0. 0.]
 [0. 5. 0.]
 [0. 0. 4.]]

Fold 3 Results:
Correctly Classified Instances: 15
Incorrectly Classified Instances: 0
Accuracy: 1.0000
Root Mean Squared Error: 0.0000
Relative Absolute Error: 0.0000
True Positive Rate: 1.0000
False Positive Rate: 0.0000
Kappa Score: 1.0000

Confusion Matrix:
[[2. 0. 0.]
 [0. 7. 0.]
 [0. 0. 6.]]

Fold 4 Results:
```

```
Fold 4 Results:
Correctly Classified Instances: 13
Incorrectly Classified Instances: 2
Accuracy: 0.8667
Root Mean Squared Error: 0.3651
Relative Absolute Error: 0.1923
True Positive Rate: 0.8667
False Positive Rate: 0.0741
Kappa Score: 0.7945

Confusion Matrix:
[[4. 0. 0.]
 [0. 3. 2.]
 [0. 0. 6.]]

Fold 5 Results:
Correctly Classified Instances: 15
Incorrectly Classified Instances: 0
Accuracy: 1.0000
Root Mean Squared Error: 0.0000
Relative Absolute Error: 0.0000
True Positive Rate: 1.0000
False Positive Rate: 0.0000
Kappa Score: 1.0000

Confusion Matrix:
[[3. 0. 0.]
 [0. 9. 0.]
 [0. 0. 3.]]

Fold 6 Results:
Correctly Classified Instances: 14
Incorrectly Classified Instances: 1
Accuracy: 0.9333
Root Mean Squared Error: 0.2582
Relative Absolute Error: 0.0833
True Positive Rate: 0.9444
False Positive Rate: 0.0278
Kappa Score: 0.8980

Confusion Matrix:
[[6. 0. 0.]
 [0. 3. 0.]
 [0. 1. 5.]]

Fold 7 Results:
```

```
Fold 7 Results:
Correctly Classified Instances: 15
Incorrectly Classified Instances: 0
Accuracy: 1.0000
Root Mean Squared Error: 0.0000
Relative Absolute Error: 0.0000
True Positive Rate: 1.0000
False Positive Rate: 0.0000
Kappa Score: 1.0000

Confusion Matrix:
[[5. 0. 0.]
 [0. 5. 0.]
 [0. 0. 5.]]

Fold 8 Results:
Correctly Classified Instances: 14
Incorrectly Classified Instances: 1
Accuracy: 0.9333
Root Mean Squared Error: 0.2582
Relative Absolute Error: 0.0974
True Positive Rate: 0.9333
False Positive Rate: 0.0278
Kappa Score: 0.8958

Confusion Matrix:
[[7. 0. 0.]
 [0. 4. 1.]
 [0. 0. 3.]]

Fold 9 Results:
Correctly Classified Instances: 13
Incorrectly Classified Instances: 2
Accuracy: 0.8667
Root Mean Squared Error: 0.3651
Relative Absolute Error: 0.2000
True Positive Rate: 0.8667
False Positive Rate: 0.0667
Kappa Score: 0.8000

Confusion Matrix:
[[5. 0. 0.]
 [0. 5. 0.]
 [0. 2. 3.]]

Fold 10 Results:
```

```
Fold 10 Results:
Correctly Classified Instances: 14
Incorrectly Classified Instances: 1
Accuracy: 0.9333
Root Mean Squared Error: 0.2582
Relative Absolute Error: 0.0765
True Positive Rate: 0.9444
False Positive Rate: 0.0256
Kappa Score: 0.8929

Confusion Matrix:
[[7. 0. 0.]
 [0. 2. 0.]
 [0. 1. 5.]]

Average Metrics Across All Folds:
Average Accuracy: 0.9467
Average RMSE: 0.1763
Average RAE: 0.0739
Average TPR: 0.9472
Average FPR: 0.0259
Average Kappa Score: 0.9178
```

Heart Spec

```
import numpy as np
import pandas as pd
from ucimlrepo import fetch_ucirepo

# Fetch SPECT Heart dataset
spect_heart = fetch_ucirepo(id=95)
X = spect_heart.data.features.to_numpy()
y = spect_heart.data.targets.to_numpy().ravel()

class BernoulliNaiveBayes:
    def fit(self, X, y):
        """
        Train the Bernoulli Naive Bayes classifier
        Specifically adapted for binary features in SPECT dataset
        """
        self.classes = np.unique(y)
        self.n_classes = len(self.classes)
        self.n_features = X.shape[1]

        # Calculate prior probabilities
        self.class_priors = {}
        for c in self.classes:
            self.class_priors[c] = np.mean(y == c)

        # Calculate feature probabilities for each class
        self.feature_probs = {}
        for c in self.classes:
            # Get samples for this class
            X_c = X[y == c]

            # Calculate probability of feature being 1 in this class
            # Add smoothing to avoid zero probabilities
            alpha = 1.0  # Laplace smoothing parameter
            feature_counts = np.sum(X_c == 1, axis=0) + alpha
            total_samples = len(X_c) + 2 * alpha
            self.feature_probs[c] = feature_counts / total_samples

    def predict(self, X):
        """
        Predict class labels for samples in X
        Using log probabilities for numerical stability
        """
        y_pred = np.zeros(X.shape[0])
```

```
            self.feature_probs[c] = feature_counts / total_samples

    def predict(self, X):
        """
        Predict class labels for samples in X
        Using log probabilities for numerical stability
        """
        y_pred = np.zeros(X.shape[0])

        for i, x in enumerate(X):
            # Calculate log probability for each class
            log_probs = {}
            for c in self.classes:
                # Start with log prior
                log_prob = np.log(self.class_priors[c])

                # Add log likelihood for each feature
                for j, x_j in enumerate(x):
                    if x_j == 1:
                        log_prob += np.log(self.feature_probs[c][j])
                    else:
                        log_prob += np.log(1 - self.feature_probs[c][j])

                log_probs[c] = log_prob

            # Select class with highest probability
            y_pred[i] = max(log_probs.items(), key=lambda x: x[1])[0]

        return y_pred

# Keep the same k_fold_split and calculate_metrics functions

# Perform k-fold cross validation with stratification
k = 10
# Stratify the folds to maintain class distribution
unique_classes = np.unique(y)
class_indices = {c: np.where(y == c)[0] for c in unique_classes}

# Create stratified folds
X_folds = []
y_folds = []
for c in unique_classes:
    indices = class_indices[c]
    np.random.shuffle(indices)
```

```
k = 10
# Stratify the folds to maintain class distribution
unique_classes = np.unique(y)
class_indices = {c: np.where(y == c)[0] for c in unique_classes}

# Create stratified folds
X_folds = []
y_folds = []
for c in unique_classes:
    indices = class_indices[c]
    np.random.shuffle(indices)
    fold_size = len(indices) // k
    for i in range(k):
        if i == 0:
            if len(X_folds) < k:
                X_folds.extend([[] for _ in range(k)])
                y_folds.extend([[] for _ in range(k)])
        start_idx = i * fold_size
        end_idx = start_idx + fold_size if i < k-1 else len(indices)
        fold_indices = indices[start_idx:end_idx]
        X_folds[i].extend(X[fold_indices])
        y_folds[i].extend(y[fold_indices])

# Convert to numpy arrays
X_folds = [np.array(fold) for fold in X_folds]
y_folds = [np.array(fold) for fold in y_folds]

metrics_per_fold = []

for fold in range(k):
    # Prepare training and testing data
    X_test = X_folds[fold]
    y_test = y_folds[fold]

    X_train = np.vstack([X_folds[i] for i in range(k) if i != fold])
    y_train = np.concatenate([y_folds[i] for i in range(k) if i != fold])

    # Train and predict
    nb = BernoulliNaiveBayes()
    nb.fit(X_train, y_train)
    y_pred = nb.predict(X_test)

    # Calculate metrics
    fold_metrics = calculate_metrics(y_test, y_pred)
```

```python
    y_pred = nb.predict(X_test)

    # Calculate metrics
    fold_metrics = calculate_metrics(y_test, y_pred)
    metrics_per_fold.append(fold_metrics)

    print(f"\nFold {fold + 1} Results:")
    print(f"Correctly Classified Instances: {fold_metrics['correct']}")
    print(f"Incorrectly Classified Instances: {fold_metrics['incorrect']}")
    print(f"Accuracy: {fold_metrics['accuracy']:.4f}")
    print(f"Precision: {fold_metrics['precision']:.4f}")
    print(f"Recall: {fold_metrics['recall']:.4f}")
    print(f"F1 Score: {fold_metrics['f1_score']:.4f}")
    print(f"True Positive Rate: {fold_metrics['tpr']:.4f}")
    print(f"False Positive Rate: {fold_metrics['fpr']:.4f}")
    print(f"Kappa Score: {fold_metrics['kappa']:.4f}")
    print("\nConfusion Matrix:")
    print(fold_metrics['confusion_matrix'])

# Calculate and print average metrics across all folds
print("\nAverage Metrics Across All Folds:")
avg_metrics = {
    'accuracy': np.mean([m['accuracy'] for m in metrics_per_fold]),
    'precision': np.mean([m['precision'] for m in metrics_per_fold]),
    'recall': np.mean([m['recall'] for m in metrics_per_fold]),
    'f1_score': np.mean([m['f1_score'] for m in metrics_per_fold]),
    'tpr': np.mean([m['tpr'] for m in metrics_per_fold]),
    'fpr': np.mean([m['fpr'] for m in metrics_per_fold]),
    'kappa': np.mean([m['kappa'] for m in metrics_per_fold])
}

print(f"Average Accuracy: {avg_metrics['accuracy']:.4f}")
print(f"Average Precision: {avg_metrics['precision']:.4f}")
print(f"Average Recall: {avg_metrics['recall']:.4f}")
print(f"Average F1 Score: {avg_metrics['f1_score']:.4f}")
print(f"Average TPR: {avg_metrics['tpr']:.4f}")
print(f"Average FPR: {avg_metrics['fpr']:.4f}")
print(f"Average Kappa Score: {avg_metrics['kappa']:.4f}")
```

```
Fold 1 Results:
Correctly Classified Instances: 21
Incorrectly Classified Instances: 5
Accuracy: 0.8077
Precision: 1.0000
Recall: 0.7619
F1 Score: 0.8649
True Positive Rate: 0.7619
False Positive Rate: 0.0000
Kappa Score: 0.5517

Confusion Matrix:
[[16  0]
 [ 5  5]]

Fold 2 Results:
Correctly Classified Instances: 23
Incorrectly Classified Instances: 3
Accuracy: 0.8846
Precision: 0.9500
Recall: 0.9048
F1 Score: 0.9268
True Positive Rate: 0.9048
False Positive Rate: 0.2000
Kappa Score: 0.6549

Confusion Matrix:
[[19  1]
 [ 2  4]]

Fold 3 Results:
Correctly Classified Instances: 20
Incorrectly Classified Instances: 6
Accuracy: 0.7692
Precision: 0.9412
Recall: 0.7619
F1 Score: 0.8421
True Positive Rate: 0.7619
False Positive Rate: 0.2000
Kappa Score: 0.4307

Confusion Matrix:
[[16  1]
 [ 5  4]]
```

```
Fold 4 Results:
Correctly Classified Instances: 19
Incorrectly Classified Instances: 7
Accuracy: 0.7308
Precision: 0.8500
Recall: 0.8095
F1 Score: 0.8293
True Positive Rate: 0.8095
False Positive Rate: 0.6000
Kappa Score: 0.1947

Confusion Matrix:
[[17  3]
 [ 4  2]]

Fold 5 Results:
Correctly Classified Instances: 19
Incorrectly Classified Instances: 7
Accuracy: 0.7308
Precision: 0.9375
Recall: 0.7143
F1 Score: 0.8108
True Positive Rate: 0.7143
False Positive Rate: 0.2000
Kappa Score: 0.3724

Confusion Matrix:
[[15  1]
 [ 6  4]]

Fold 6 Results:
Correctly Classified Instances: 19
Incorrectly Classified Instances: 7
Accuracy: 0.7308
Precision: 0.9375
Recall: 0.7143
F1 Score: 0.8108
True Positive Rate: 0.7143
False Positive Rate: 0.2000
Kappa Score: 0.3724

Confusion Matrix:
[[15  1]
 [ 6  4]]
```

```
[[15  1]
 [ 6  4]]

Fold 7 Results:
Correctly Classified Instances: 22
Incorrectly Classified Instances: 4
Accuracy: 0.8462
Precision: 0.9474
Recall: 0.8571
F1 Score: 0.9000
True Positive Rate: 0.8571
False Positive Rate: 0.2000
Kappa Score: 0.5702

Confusion Matrix:
[[18  1]
 [ 3  4]]

Fold 8 Results:
Correctly Classified Instances: 21
Incorrectly Classified Instances: 5
Accuracy: 0.8077
Precision: 1.0000
Recall: 0.7619
F1 Score: 0.8649
True Positive Rate: 0.7619
False Positive Rate: 0.0000
Kappa Score: 0.5517

Confusion Matrix:
[[16  0]
 [ 5  5]]

Fold 9 Results:
Correctly Classified Instances: 20
Incorrectly Classified Instances: 6
Accuracy: 0.7692
Precision: 0.8571
Recall: 0.8571
F1 Score: 0.8571
True Positive Rate: 0.8571
False Positive Rate: 0.6000
Kappa Score: 0.2571

Confusion Matrix:
[[18  3]
 [ 3  2]]
```

```
Incorrectly Classified Instances: 6
Accuracy: 0.7692
Precision: 0.8571
Recall: 0.8571
F1 Score: 0.8571
True Positive Rate: 0.8571
False Positive Rate: 0.6000
Kappa Score: 0.2571

Confusion Matrix:
[[18  3]
 [ 3  2]]

Fold 10 Results:
Correctly Classified Instances: 28
Incorrectly Classified Instances: 5
Accuracy: 0.8485
Precision: 0.9091
Recall: 0.8696
F1 Score: 0.8889
True Positive Rate: 0.8696
False Positive Rate: 0.2000
Kappa Score: 0.6512

Confusion Matrix:
[[20  2]
 [ 3  8]]

Average Metrics Across All Folds:
Average Accuracy: 0.7925
Average Precision: 0.9330
Average Recall: 0.8012
Average F1 Score: 0.8596
Average TPR: 0.8012
Average FPR: 0.2400
Average Kappa Score: 0.4607
```