110705067 洪薇欣 HW1

Methods

1. Classifiers:

Naive Bayes Classifier:

a probabilistic machine learning algorithm based on Bayes' Theorem with an assumption of **independence among predictors**. The most important limitation of Naive Bayes is its "naive" independence assumption - the **assumption that features are conditionally independent given the class**. Not suitable if dataset has strongly correlated features

How It Works: P(class|features) = P(features|class) × P(class) / P(features)

- P(class|features) is the posterior probability of the class given the features
- P(features|class) is the likelihood of the features given the class
- P(class) is the prior probability of the class
- P(features) is the prior probability of the features

Perceptron Classifier:

A perceptron takes multiple input signals, assigns weights to them, sums them up (along with a bias term), and then applies a step function to produce a binary output. It essentially creates a **linear decision boundary to separate two classes of data**. The most fundamental limitation is that perceptron **can only correctly classify linearly separable data**.

Key components: Input features ($x_1, x_2, \ldots, x_n$), Weights ($w_1, w_2, \ldots, w_n$), Bias term (b), Activation function

How it works:

1. Initialize weights and bias (often randomly or to zeros)
2. For each training example:
   - Calculate the weighted sum: $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$
   - Apply activation function: output = 1 if $z \geq 0$, else 0
   - Compare with actual label and update weights if prediction is wrong
3. Repeat until convergence or maximum iterations

Weight update rule: If prediction is correct: no change. If prediction is wrong: $w_i = w_i + \eta(y - \hat{y})x_i$ (η is the learning rate, y is the true label, and $\hat{y}$ is the prediction)

2. Data Splitting:

single train-test split:

Divide dataset into two parts. **Randomize the order** of your dataset before splitting to ensure both sets contain a representative distribution of the data. With only one test set, your evaluation might be sensitive to which data points happen to end up in the test set.

<u>K-fold Cross-validation:</u>

Every data point is used for both training and validation. It provides a more robust estimate of model performance than a single train-test split. Each data point is used for validation exactly once

How it works:

1. The original dataset is partitioned into k equally sized subsets (folds)
2. The model is trained and tested k times, where: **Each time, one-fold is used as the validation set .**The remaining **k-1 folds are used as the training set**
3. The k results are averaged to produce a single performance estimate

3. <u>Evaluations</u>

- **<u>Precision:</u>**

High precision means when your model predicts something as positive, it's usually **correct**.

Formula: Precision= TP/(TP+FP)

- **<u>Accuracy:</u>**

Good general measure but **can be misleading when classes are imbalanced**.

Formula: Accuracy=TP+TN/(FP+FN+TP+TN)

<u>Recall (Sensitivity, True Positive Rate):</u>

High recall means the model **catches most of the positive** cases.

Formula: Recall =TP /(FN+TP)

- **<u>Recall (Sensitivity, True Positive Rate):</u>**

A good single metric when you want to **balance precision and recall**, especially on imbalanced datasets.

Formula: F1 Score=2×Precision×Recall/(Precision + Recall)

- **<u>ROC (Receiver Operating Characteristic) Curve:</u>**

A model with a curve closer to the **top-left corner is better.**

X-axis: False Positive Rate = FP / (FP + TN), **Y-axis**: True Positive Rate = TP / (TP + FN)

- **<u>AUC (Area Under the Curve):</u>**

Higher AUC = better ability to distinguish between classes.

Range: 0 to 1 (0.5 = no better than random, 1.0 = perfect classifier)

4. **Normalization:**

- **Min-Max Scaling:**

Min-max scaling is a normalization technique used in data preprocessing to rescale features to a **fixed range—typically [0, 1]**. It's especially useful when you **want all features to have the same scale without distorting differences in the ranges of values.**

How it works: Xscaled=(X−Xmin)/(Xmax−Xmin) , where : X is the original value, Xmin, Xmax :are the min and max values of the feature

- **Z-Score Scaling (standardization) :**

it transforms features so they have: **Mean = 0, Standard deviation = 1.**

How it works: Xscaled= (X−μ)/σ

Experiments

1. **Datasets**

   Breast Cancer Coimbra: 116 samples, 9 features, **2 classes (small & imbalanced)**

   Ionosphere: 351 samples, 34 features, **2 classes (high dimensional)**

   Iris: 150 samples, 4 features, **3 classes (3 balanced classes)**

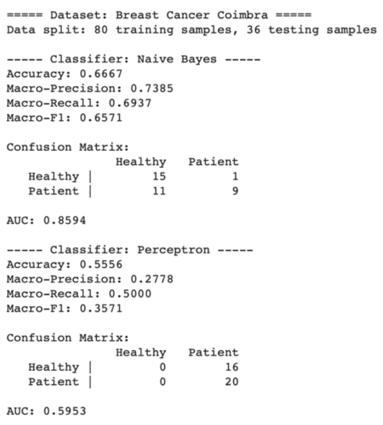
   Wine: 178 samples, 13 features, **3 classes (more complex)**

2. **Results and analysis**

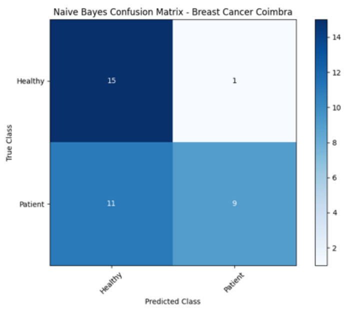
- **Naïve Bayes vs. Perceptron (single tarin-test split ratio=0.7)**

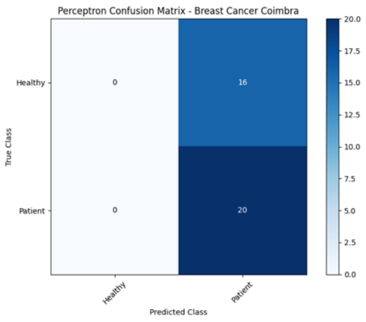
- **Breast Cancer Coimbra:**

Better performance on naïve bayes (higher AUC, precision). Naïve Bayes performs reasonably well with good AUC, even if some patients are misclassified. Accuracy and AUC are pretty solid, especially for a small dataset. Precision and Recall are balanced. Confusion Matrix shows healthy class is well identified (15 out of 16 correctly predicted), misses 11 patients (lower sensitivity for patient class). **Perceptron classifier perform less reliably** on Breast Cancer Coimbra might be due to small dataset and possibly **non-linear boundaries**. Because perceptron algorithm can only correctly classify linearly separable data. And naïve bayes only assumes features are independent. Also, naïve bayes works well with small data and probabilistic interpretations. Perceptron's accuracy and AUC is not good, close to random guessing. Precision is extremely low. Model predicted everyone as a patient. Confusion Matrix completely fails to predict the Healthy class (0 true positives).



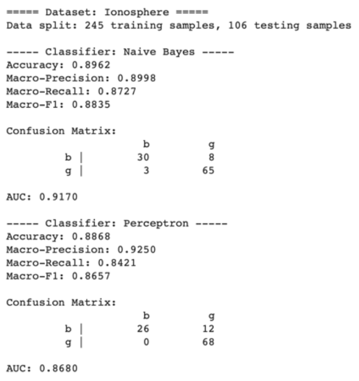
```
===== Dataset: Breast Cancer Coimbra =====
Data split: 80 training samples, 36 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.6667
Macro-Precision: 0.7385
Macro-Recall: 0.6937
Macro-F1: 0.6571

Confusion Matrix:
               Healthy   Patient
   Healthy |        15         1
   Patient |        11         9

AUC: 0.8594

----- Classifier: Perceptron -----
Accuracy: 0.5556
Macro-Precision: 0.2778
Macro-Recall: 0.5000
Macro-F1: 0.3571

Confusion Matrix:
               Healthy   Patient
   Healthy |         0        16
   Patient |         0        20

AUC: 0.5953
```



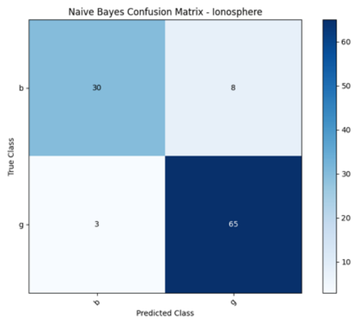


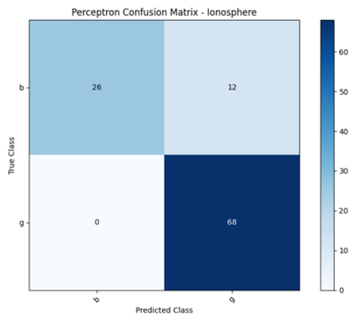
- **Ionosphere:**

Naïve bayes handles high-dimensional input well. Accuracy and AUC are very strong. Confusion Matrix shows good detection of both classes. High F1 score and very little class imbalance in prediction. Perceptron has strong performance, but model favors one class more. **Naïve bayes still edges it out slightly due to better balance.** Perceptron's accuracy and AUC are slightly worse than naïve bayes but still solid. Perfectly classified the "g" class but missed 12 in "b". Precision was very high (0.925), but recall for b dropped a bit (**class bias**).



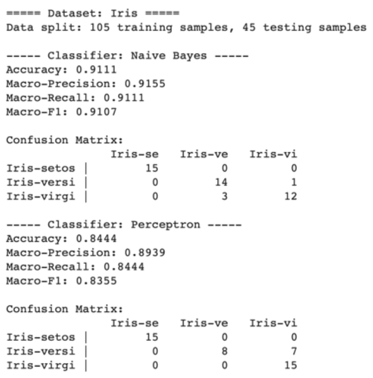
```
===== Dataset: Ionosphere =====
Data split: 245 training samples, 106 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.8962
Macro-Precision: 0.8998
Macro-Recall: 0.8727
Macro-F1: 0.8835

Confusion Matrix:
               b         g
        b |   30         8
        g |    3        65

AUC: 0.9170

----- Classifier: Perceptron -----
Accuracy: 0.8868
Macro-Precision: 0.9250
Macro-Recall: 0.8421
Macro-F1: 0.8657

Confusion Matrix:
               b         g
        b |   26        12
        g |    0        68

AUC: 0.8680
```



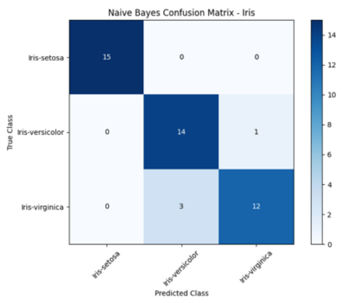


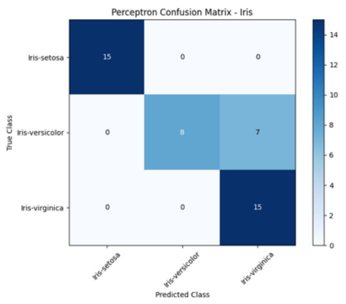
- **Iris:**

Naïve Bayes does great, as Iris is nicely separated and simple. Accuracy is strong. Confusion Matrix: perfect on Setosa, small confusion between Versicolor & Virginica.F1, Precision, recall all near 0.91, which is very balanced. **Perceptron is more sensitive to feature overlap (Versicolor vs. Virginica). Not as stable here as naïve bayes.** Perceptron has accuracy decent but noticeably worse. Misclassifies 7 Versicolor as Virginica. Perfect on Setosa and Virginica.



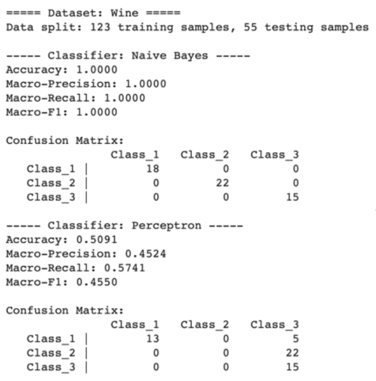
```
===== Dataset: Iris =====
Data split: 105 training samples, 45 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9111
Macro-Precision: 0.9155
Macro-Recall: 0.9111
Macro-F1: 0.9107

Confusion Matrix:
               Iris-se   Iris-ve   Iris-vi
Iris-setos |        15         0         0
Iris-versi |         0        14         1
Iris-virgi |         0         3        12

----- Classifier: Perceptron -----
Accuracy: 0.8444
Macro-Precision: 0.8939
Macro-Recall: 0.8444
Macro-F1: 0.8355

Confusion Matrix:
               Iris-se   Iris-ve   Iris-vi
Iris-setos |        15         0         0
Iris-versi |         0         8         7
Iris-virgi |         0         0        15
```



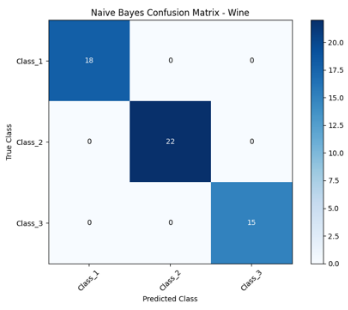


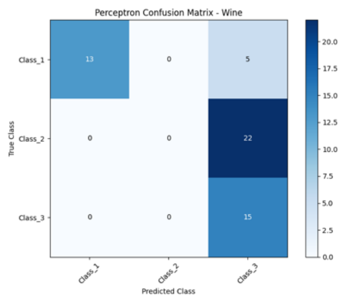
- **Wine:**

For naïve bayes, accuracy is 100% on test set. No misclassifications at all. Very strong performance. This might be due to this split was particularly favorable, or naïve bayes aligns extremely well with class distributions. performance. For perceptron, accuracy: 50.9%, poor performance. Misclassifies entire Class 2 as Class 3. This might be due to **can't separate classes linearly with the current features. Naïve bayes wins by a large margin**.



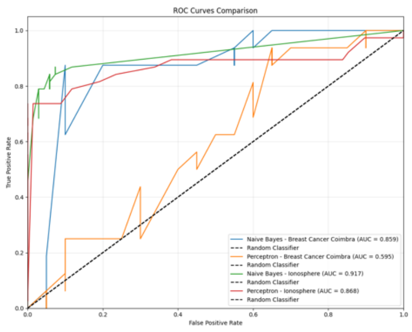
```
===== Dataset: Wine =====
Data split: 123 training samples, 55 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 1.0000
Macro-Precision: 1.0000
Macro-Recall: 1.0000
Macro-F1: 1.0000

Confusion Matrix:
              Class_1   Class_2   Class_3
   Class_1 |       18         0         0
   Class_2 |        0        22         0
   Class_3 |        0         0        15

----- Classifier: Perceptron -----
Accuracy: 0.5091
Macro-Precision: 0.4524
Macro-Recall: 0.5741
Macro-F1: 0.4550

Confusion Matrix:
              Class_1   Class_2   Class_3
   Class_1 |       13         0         5
   Class_2 |        0         0        22
   Class_3 |        0         0        15
```





- **ROC:**

Naïve Bayes is consistently more stable and accurate across datasets — especially for smaller or imbalanced ones. **Perceptron needs linearly separable data** and is more fragile with smaller or complex distributions.

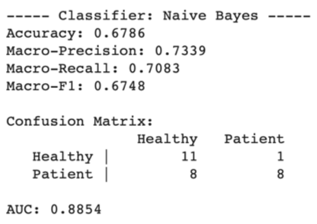


■ <u>**Naïve Bayes vs. Perceptron (K-fold cross validation, K=5)**</u>

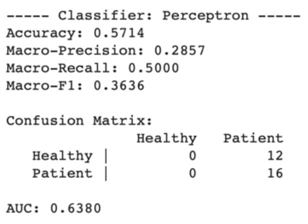
Single 70/30 split may create bias, so expected cross-validation would give more realistic generalization performance. 70/30 split might have high variance—results may differ significantly depending on how the split is done. And, sensitive to class imbalance in the split. **5-Fold Cross-Validation has lower variance**, since every data point is used for both training and validation. Expecting **5-Fold Cross-Validation to give a better estimate of generalization**. Naïve Bayes might performance improves slightly if some classes were underrepresented in a simple split. Perceptron benefits more using cross-validation, since it can learn from more comprehensive samples over multiple folds.

- **Breast Cancer Coimbra:**

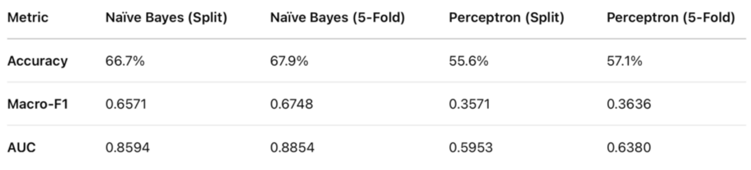
  88 training samples, 28 testing samples



```
----- Classifier: Naive Bayes -----
Accuracy: 0.6786
Macro-Precision: 0.7339
Macro-Recall: 0.7083
Macro-F1: 0.6748

Confusion Matrix:
              Healthy   Patient
   Healthy |       11         1
   Patient |        8         8

AUC: 0.8854
```



```
----- Classifier: Perceptron -----
Accuracy: 0.5714
Macro-Precision: 0.2857
Macro-Recall: 0.5000
Macro-F1: 0.3636

Confusion Matrix:
              Healthy   Patient
   Healthy |        0        12
   Patient |        0        16

AUC: 0.6380
```

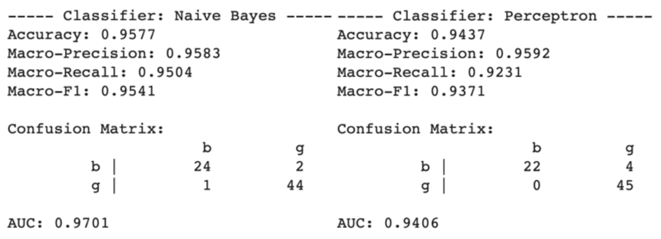


Metric	Naïve Bayes (Split)	Naïve Bayes (5-Fold)	Perceptron (Split)	Perceptron (5-Fold)
Accuracy	66.7%	67.9%	55.6%	57.1%
Macro-F1	0.6571	0.6748	0.3571	0.3636
AUC	0.8594	0.8854	0.5953	0.6380

Naïve Bayes is consistent across both splits, **slightly better with 5-fold**, especially in AUC. Perceptron is underperforming again but **gets a small improvement with cross-validation**.

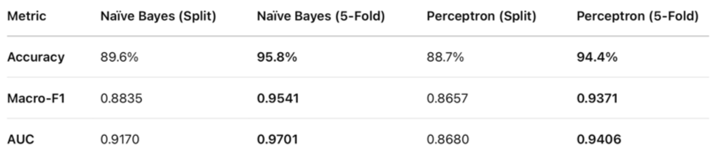
- **Ionosphere:**

  280 training samples, 71 testing samples



```
----- Classifier: Naive Bayes -----
Accuracy: 0.9577
Macro-Precision: 0.9583
Macro-Recall: 0.9504
Macro-F1: 0.9541

Confusion Matrix:
                  b         g
        b |      24         2
        g |       1        44

AUC: 0.9701
```

```
----- Classifier: Perceptron -----
Accuracy: 0.9437
Macro-Precision: 0.9592
Macro-Recall: 0.9231
Macro-F1: 0.9371

Confusion Matrix:
                  b         g
        b |      22         4
        g |       0        45

AUC: 0.9406
```

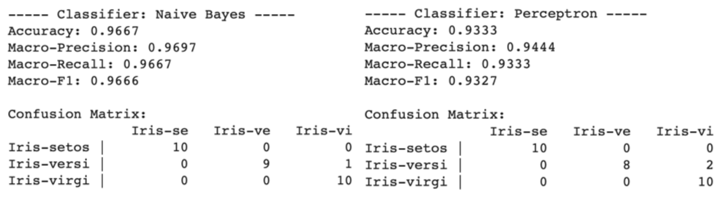


Metric	Naïve Bayes (Split)	Naïve Bayes (5-Fold)	Perceptron (Split)	Perceptron (5-Fold)
Accuracy	89.6%	**95.8%**	88.7%	**94.4%**
Macro-F1	0.8835	**0.9541**	0.8657	**0.9371**
AUC	0.9170	**0.9701**	0.8680	**0.9406**

Both classifiers show **notable improvements using 5-fold** (more training data = better learning). High-dimensional datasets benefit heavily from cross-validation.

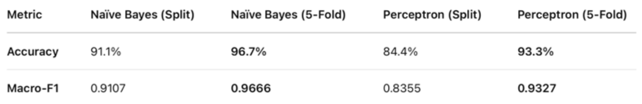
- **Iris:**

  120 training samples, 30 testing samples



```
----- Classifier: Naive Bayes -----
Accuracy: 0.9667
Macro-Precision: 0.9697
Macro-Recall: 0.9667
Macro-F1: 0.9666

Confusion Matrix:
              Iris-se   Iris-ve   Iris-vi
Iris-setos |       10         0         0
Iris-versi |        0         9         1
Iris-virgi |        0         0        10
```

```
----- Classifier: Perceptron -----
Accuracy: 0.9333
Macro-Precision: 0.9444
Macro-Recall: 0.9333
Macro-F1: 0.9327

Confusion Matrix:
              Iris-se   Iris-ve   Iris-vi
Iris-setos |       10         0         0
Iris-versi |        0         8         2
Iris-virgi |        0         0        10
```

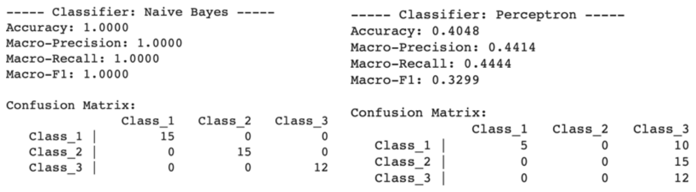


Metric	Naïve Bayes (Split)	Naïve Bayes (5-Fold)	Perceptron (Split)	Perceptron (5-Fold)
Accuracy	91.1%	**96.7%**	84.4%	**93.3%**
Macro-F1	0.9107	**0.9666**	0.8355	**0.9327**

Better performance for both models with cross-validation. Perceptron was weaker in the single split but caught up well in the folds.

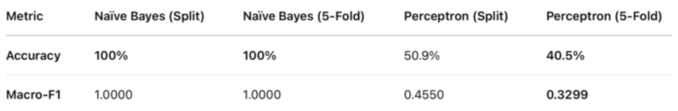
- **Wine:**

  136 training samples, 42 testing samples



```
----- Classifier: Naive Bayes -----
Accuracy: 1.0000
Macro-Precision: 1.0000
Macro-Recall: 1.0000
Macro-F1: 1.0000

Confusion Matrix:
             Class_1   Class_2   Class_3
   Class_1 |      15         0         0
   Class_2 |       0        15         0
   Class_3 |       0         0        12
```

```
----- Classifier: Perceptron -----
Accuracy: 0.4048
Macro-Precision: 0.4414
Macro-Recall: 0.4444
Macro-F1: 0.3299

Confusion Matrix:
             Class_1   Class_2   Class_3
   Class_1 |       5         0        10
   Class_2 |       0         0        15
   Class_3 |       0         0        12
```

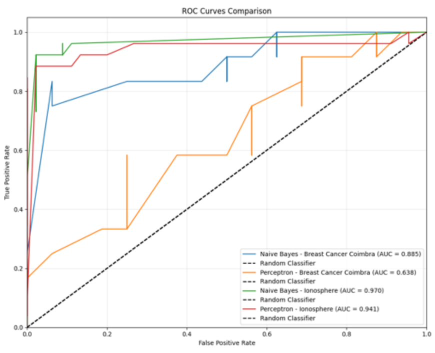


Metric	Naïve Bayes (Split)	Naïve Bayes (5-Fold)	Perceptron (Split)	Perceptron (5-Fold)
Accuracy	**100%**	**100%**	50.9%	**40.5%**
Macro-F1	1.0000	1.0000	0.4550	**0.3299**

Naïve bayes remains perfect, highly suited for this dataset, likely due to Gaussian assumptions matching real distributions. **Perceptron does worse with cross-validation**, possibly due to training folds that include tougher class boundaries.

- **ROC:**

Cross-validation gives a clearer and more generalizable picture, especially for small or sensitive datasets like Coimbra or Wine. Naïve Bayes is consistently reliable, especially when you can't be sure about linearity or when data is limited. Perceptron is sensitive to data distribution and class overlap. It **benefits more from larger, balanced training sets, which cross-validation provides**.



■ **Naïve Bayes vs. Perceptron (K-fold, with K=5~10, Breast Cancer Coimbra)**

As K increases, the more reliable estimate. As K increases, each model is trained on a larger portion of the data. This often leads to lower bias in the performance estimate. You get a smoother, more stable evaluation of your model. With high K (e.g., K=10 or K=20), more data is used for training in each fold. That can slightly improve how well your model generalizes compared to small K. But, as K increases (smaller validation sets) might not capture all the diversity of the data. So while bias decreases, the variance of the validation score might increase slightly.

```
Cross-Validation (5-fold):
88 training samples, 28 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.6786
Macro-Precision: 0.7339
Macro-Recall: 0.7083
Macro-F1: 0.6748

Confusion Matrix:
               Healthy   Patient
   Healthy |        11         1
   Patient |         8         8

----- Classifier: Perceptron -----
Accuracy: 0.5714
Macro-Precision: 0.2857
Macro-Recall: 0.5000
Macro-F1: 0.3636

Confusion Matrix:
               Healthy   Patient
   Healthy |         0        12
   Patient |         0        16
```

```
Cross-Validation (6-fold):
90 training samples, 26 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.6538
Macro-Precision: 0.7180
Macro-Recall: 0.6726
Macro-F1: 0.6406

Confusion Matrix:
               Healthy   Patient
   Healthy |        11         1
   Patient |         8         6

----- Classifier: Perceptron -----
Accuracy: 0.5385
Macro-Precision: 0.2692
Macro-Recall: 0.5000
Macro-F1: 0.3500

Confusion Matrix:
               Healthy   Patient
   Healthy |         0        12
   Patient |         0        14
```

```
Cross-Validation (7-fold):
96 training samples, 20 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.6500
Macro-Precision: 0.7000
Macro-Recall: 0.6500
Macro-F1: 0.6267

Confusion Matrix:
               Healthy   Patient
   Healthy |         9         1
   Patient |         6         4

----- Classifier: Perceptron -----
Accuracy: 0.6000
Macro-Precision: 0.6562
Macro-Recall: 0.6000
Macro-F1: 0.5604

Confusion Matrix:
               Healthy   Patient
   Healthy |         3         7
   Patient |         1         9
```

```
Cross-Validation (8-fold):
98 training samples, 18 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.6667
Macro-Precision: 0.6964
Macro-Recall: 0.6375
Macro-F1: 0.6250

Confusion Matrix:
               Healthy   Patient
   Healthy |         9         1
   Patient |         5         3

----- Classifier: Perceptron -----
Accuracy: 0.4444
Macro-Precision: 0.4688
Macro-Recall: 0.4875
Macro-F1: 0.3750

Confusion Matrix:
               Healthy   Patient
   Healthy |         1         9
   Patient |         1         7
```

```
Cross-Validation (9-fold):
96 training samples, 20 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.7000
Macro-Precision: 0.7188
Macro-Recall: 0.6458
Macro-F1: 0.6429

Confusion Matrix:
               Healthy   Patient
   Healthy |        11         1
   Patient |         5         3

----- Classifier: Perceptron -----
Accuracy: 0.4000
Macro-Precision: 0.2000
Macro-Recall: 0.5000
Macro-F1: 0.2857

Confusion Matrix:
               Healthy   Patient
   Healthy |         0        12
   Patient |         0         8
```

```
Cross-Validation (10-fold):
99 training samples, 17 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.5882
Macro-Precision: 0.6500
Macro-Recall: 0.6286
Macro-F1: 0.5825

Confusion Matrix:
               Healthy   Patient
   Healthy |         6         1
   Patient |         6         4

----- Classifier: Perceptron -----
Accuracy: 0.5882
Macro-Precision: 0.2941
Macro-Recall: 0.5000
Macro-F1: 0.3704

Confusion Matrix:
               Healthy   Patient
   Healthy |         0         7
   Patient |         0        10
```

K	Test Size	Naive Bayes Accuracy	Perceptron Accuracy
5	28	**67.9%**	57.1%
6	26	**65.4%**	53.9%
7	20	**65.0%**	**60.0%**
8	18	**66.7%**	44.4%
9	20	**70.0%**	40.0%
10	17	**58.8%**	**58.8%**

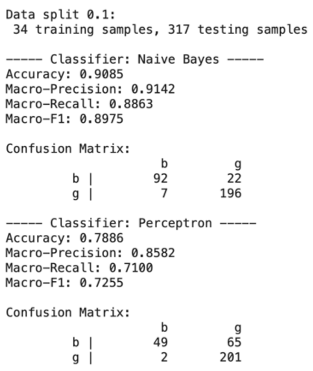
As K increases, training set grows, test set shrinks.

For naïve bayes performance is quite consistent across K=5 to 9, but a dip at K=10.Macro-F1 remains high (~0.62–0.67) from K=5 to 9. Slight accuracy drops at K=10 may stem from the smaller test set (~17 samples), increasing variance in evaluation. Confusion matrix stays balanced, which predicts both classes. **Naive Bayes is robust across folds (good generalization) but may overfit slightly when fold size becomes too high (K=10)**. For perceptron, it performs very poorly at K=6, 8, 9 (only predicts one class). Peaks at K=7 (Accuracy = 60%, F1 = 0.56), when the test set is still decent in size and class distribution. Recovers at K=10, but still only "ok" (Accuracy = 58.8%).Macro-F1 fluctuates wildly: 0.28 → 0.56 → 0.37 → 0.29 → 0.37.

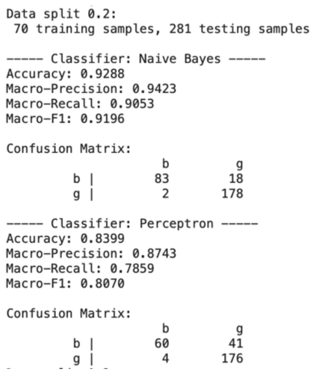
Confusion Matrix often only predicts one class, this shows that the model is underfitting or unstable, likely due to: high sensitivity to sample imbalance. Possibly not enough training data for a high-variance model like Perceptron. Perceptron is unreliable without proper tuning, and more folds don't necessarily help. In fact, they hurt in some cases. **Perceptron doesn't benefit, model is inconsistent and unstable across folds.**

- **<u>Naïve Bayes vs. Perceptron (single tarin-test split ratio, with ratio= 0.1~0.9, Ionosphere)</u>**

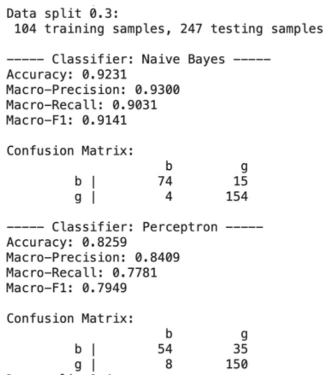
Smaller test set (smaller ratio): higher risk of **overfitting but better learning from data**. Larger test set (bigger ratio): **better evaluation**, but less data for training, which might impact performance.



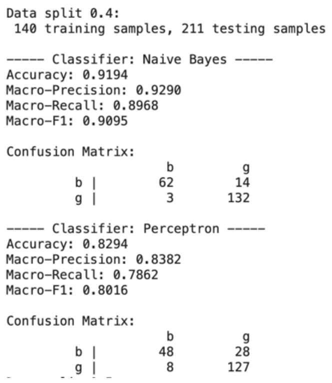
```
Data split 0.1:
 34 training samples, 317 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9085
Macro-Precision: 0.9142
Macro-Recall: 0.8863
Macro-F1: 0.8975

Confusion Matrix:
              b        g
        b |   92       22
        g |    7      196

----- Classifier: Perceptron -----
Accuracy: 0.7886
Macro-Precision: 0.8582
Macro-Recall: 0.7100
Macro-F1: 0.7255

Confusion Matrix:
              b        g
        b |   49       65
        g |    2      201
```



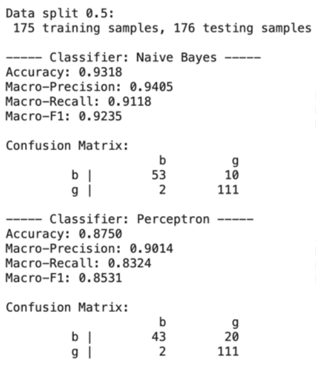
```
Data split 0.2:
 70 training samples, 281 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9288
Macro-Precision: 0.9423
Macro-Recall: 0.9053
Macro-F1: 0.9196

Confusion Matrix:
              b        g
        b |   83       18
        g |    2      178

----- Classifier: Perceptron -----
Accuracy: 0.8399
Macro-Precision: 0.8743
Macro-Recall: 0.7859
Macro-F1: 0.8070

Confusion Matrix:
              b        g
        b |   60       41
        g |    4      176
```



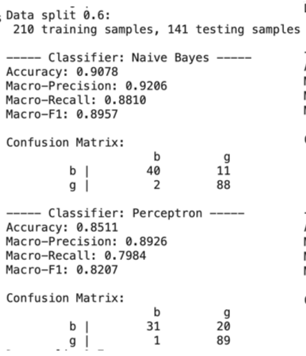
```
Data split 0.3:
 104 training samples, 247 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9231
Macro-Precision: 0.9300
Macro-Recall: 0.9031
Macro-F1: 0.9141

Confusion Matrix:
              b        g
        b |   74       15
        g |    4      154

----- Classifier: Perceptron -----
Accuracy: 0.8259
Macro-Precision: 0.8409
Macro-Recall: 0.7781
Macro-F1: 0.7949

Confusion Matrix:
              b        g
        b |   54       35
        g |    8      150
```



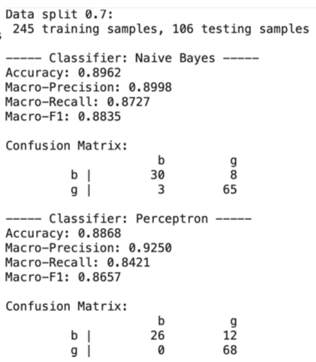
```
Data split 0.4:
 140 training samples, 211 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9194
Macro-Precision: 0.9290
Macro-Recall: 0.8968
Macro-F1: 0.9095

Confusion Matrix:
              b        g
        b |   62       14
        g |    3      132

----- Classifier: Perceptron -----
Accuracy: 0.8294
Macro-Precision: 0.8382
Macro-Recall: 0.7862
Macro-F1: 0.8016

Confusion Matrix:
              b        g
        b |   48       28
        g |    8      127
```



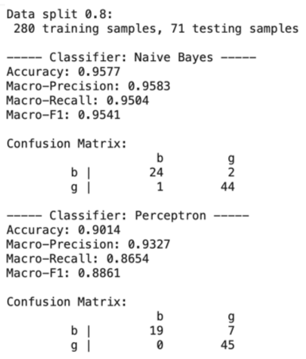
```
Data split 0.5:
 175 training samples, 176 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9318
Macro-Precision: 0.9405
Macro-Recall: 0.9118
Macro-F1: 0.9235

Confusion Matrix:
              b        g
        b |   53       10
        g |    2      111

----- Classifier: Perceptron -----
Accuracy: 0.8750
Macro-Precision: 0.9014
Macro-Recall: 0.8324
Macro-F1: 0.8531

Confusion Matrix:
              b        g
        b |   43       20
        g |    2      111
```



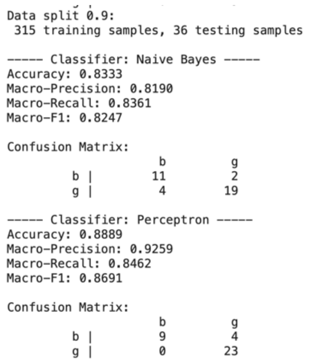
```
Data split 0.6:
 210 training samples, 141 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9078
Macro-Precision: 0.9206
Macro-Recall: 0.8810
Macro-F1: 0.8957

Confusion Matrix:
              b        g
        b |   40       11
        g |    2       88

----- Classifier: Perceptron -----
Accuracy: 0.8511
Macro-Precision: 0.8926
Macro-Recall: 0.7984
Macro-F1: 0.8207

Confusion Matrix:
              b        g
        b |   31       20
        g |    1       89
```



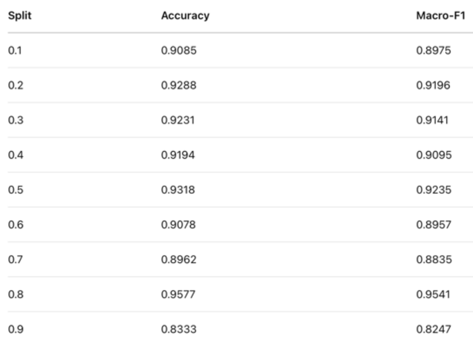
```
Data split 0.7:
 245 training samples, 106 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.8962
Macro-Precision: 0.8998
Macro-Recall: 0.8727
Macro-F1: 0.8835

Confusion Matrix:
              b        g
        b |   30        8
        g |    3       65

----- Classifier: Perceptron -----
Accuracy: 0.8868
Macro-Precision: 0.9250
Macro-Recall: 0.8421
Macro-F1: 0.8657

Confusion Matrix:
              b        g
        b |   26       12
        g |    0       68
```



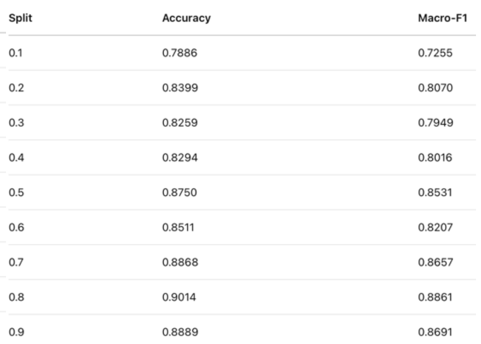
```
Data split 0.8:
 280 training samples, 71 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.9577
Macro-Precision: 0.9583
Macro-Recall: 0.9504
Macro-F1: 0.9541

Confusion Matrix:
              b        g
        b |   24        2
        g |    1       44

----- Classifier: Perceptron -----
Accuracy: 0.9014
Macro-Precision: 0.9327
Macro-Recall: 0.8654
Macro-F1: 0.8861

Confusion Matrix:
              b        g
        b |   19        7
        g |    0       45
```



```
Data split 0.9:
 315 training samples, 36 testing samples

----- Classifier: Naive Bayes -----
Accuracy: 0.8333
Macro-Precision: 0.8190
Macro-Recall: 0.8361
Macro-F1: 0.8247

Confusion Matrix:
              b        g
        b |   11        2
        g |    4       19

----- Classifier: Perceptron -----
Accuracy: 0.8889
Macro-Precision: 0.9259
Macro-Recall: 0.8462
Macro-F1: 0.8691

Confusion Matrix:
              b        g
        b |    9        4
        g |    0       23
```



Split	Accuracy	Macro-F1
0.1	0.9085	0.8975
0.2	0.9288	0.9196
0.3	0.9231	0.9141
0.4	0.9194	0.9095
0.5	0.9318	0.9235
0.6	0.9078	0.8957
0.7	0.8962	0.8835
0.8	0.9577	0.9541
0.9	0.8333	0.8247



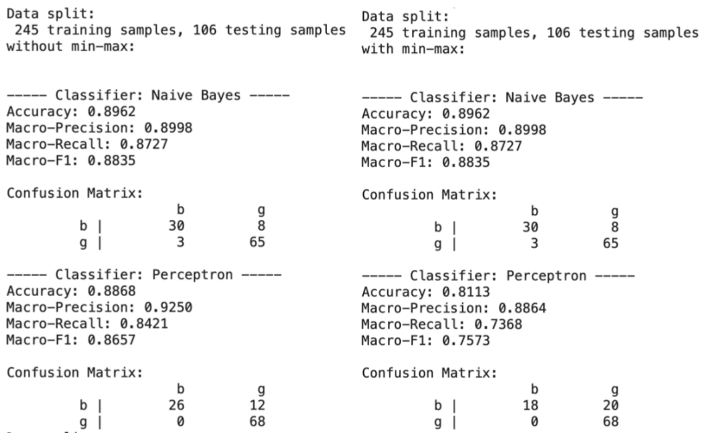
Split	Accuracy	Macro-F1
0.1	0.7886	0.7255
0.2	0.8399	0.8070
0.3	0.8259	0.7949
0.4	0.8294	0.8016
0.5	0.8750	0.8531
0.6	0.8511	0.8207
0.7	0.8868	0.8657
0.8	0.9014	0.8861
0.9	0.8889	0.8691

Naive Bayes performance is **generally stable or slightly improves, but dips at the very end** (0.9 split), likely due to very small test set (only 36 samples), making results noisy or unreliable (with the graph in the middle shows above). Performs consistently well from splits 0.1 to 0.5. Small performance dip at 0.6 and 0.7, possibly due to a less representative test set or class imbalance. Peak performance at 0.8 split (training on 280 samples), likely hitting the sweet spot of training data volume + sufficient test data. At 0.9, performance drops. **Perceptron shows clear improvement** with more training data and becomes competitive or better than Naive Bayes at larger splits (with the graph on the right shows above). Starts off weaker than Naive Bayes at small training sizes. Consistently improves with more training data, especially after split 0.5. Overtakes

Naive Bayes in Macro-F1 at splits 0.7 and 0.9, showing strong learning capacity. Perceptron is m**ore sensitive to training data volume than Naive Bayes —** classic behavior of discriminative vs. generative models.

■ Naïve Bayes vs. Perceptron (tarin-test split ratio=0.7, min-max scaling, Ionosphere)

Expect **better model performance** will using min-max scaling, because by scaling everything to the same range, each feature contributes equally to the model's decisions. Also expect improved Accuracy due to the model can learn patterns more fairly when features are on the same scale, leads to better generalization and less overfitting or underfitting.



```
Data split:
 245 training samples, 106 testing samples
without min-max:


----- Classifier: Naive Bayes -----
Accuracy: 0.8962
Macro-Precision: 0.8998
Macro-Recall: 0.8727
Macro-F1: 0.8835

Confusion Matrix:
                    b        g
        b |        30        8
        g |         3       65

----- Classifier: Perceptron -----
Accuracy: 0.8868
Macro-Precision: 0.9250
Macro-Recall: 0.8421
Macro-F1: 0.8657

Confusion Matrix:
                    b        g
        b |        26       12
        g |         0       68
```

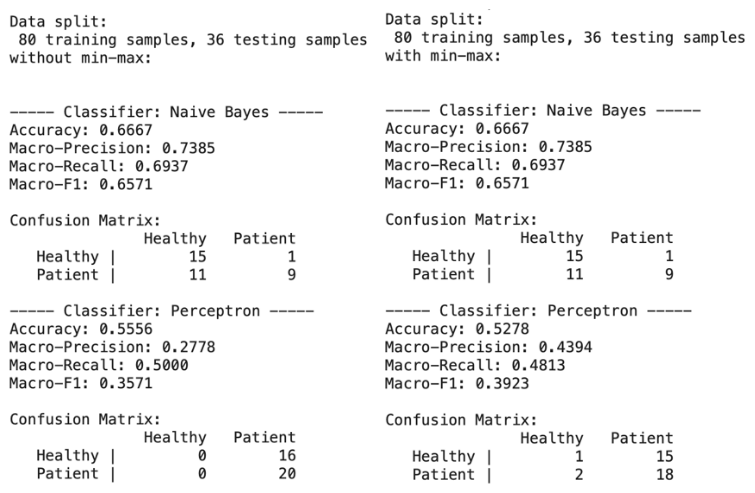
```
Data split:
 245 training samples, 106 testing samples
with min-max:


----- Classifier: Naive Bayes -----
Accuracy: 0.8962
Macro-Precision: 0.8998
Macro-Recall: 0.8727
Macro-F1: 0.8835

Confusion Matrix:
                    b        g
        b |        30        8
        g |         3       65

----- Classifier: Perceptron -----
Accuracy: 0.8113
Macro-Precision: 0.8864
Macro-Recall: 0.7368
Macro-F1: 0.7573

Confusion Matrix:
                    b        g
        b |        18       20
        g |         0       68
```

For naïve bayes, **without min-max and with min-max, results are identical**. This is because naïve bayes does not require feature scaling because it models the distribution of each feature independently using its own mean and variance. **Rescaling features doesn't affect the probability distributions relative to each other**, it just transforms them linearly, which doesn't matter here. For perceptron, **performance worsens after scaling.** This might be due to hyperparameters weren't re-tuned after scaling. Scaling changes the value range of all features but learning rate or regularization terms optimized for unscaled data may no longer work well post-scaling, causing underfitting or ineffective weight updates. Also, perceptron is a linear model If the class separation is not strictly linear in the scaled feature space, performance might dip. Scaling might compress feature differences that were previously helping the model distinguish class b. And min-max scaling is sensitive to outliers. If outliers exist, the scaling could compress most useful data into a smaller range, making decision boundaries fuzzier. **Some datasets (like Ionosphere) also might contain informative high-magnitude features.** Scaling them to [0, 1] could reduce their impact, hurting linear classifiers like the perceptron.

■ Naïve Bayes vs. Perceptron (arin-test split ratio=0.7, min-max scaling, Breast Cancer Coimbra)

Expect **better model performance** will using min-max scaling, because by scaling everything to the same range, each feature contributes equally to the model's decisions.

For n**aïve bayes, no change again. Same results with and without min-max scaling.** Naive Bayes doesn't care about feature scale—only the distribution (mean, variance) of each feature per class. A linear transformation like min-max doesn't affect how Naive Bayes calculates likelihoods, so results stay identical. For perceptron, slight gain. Still underfits; scaling not enough to fix poor boundary or data size

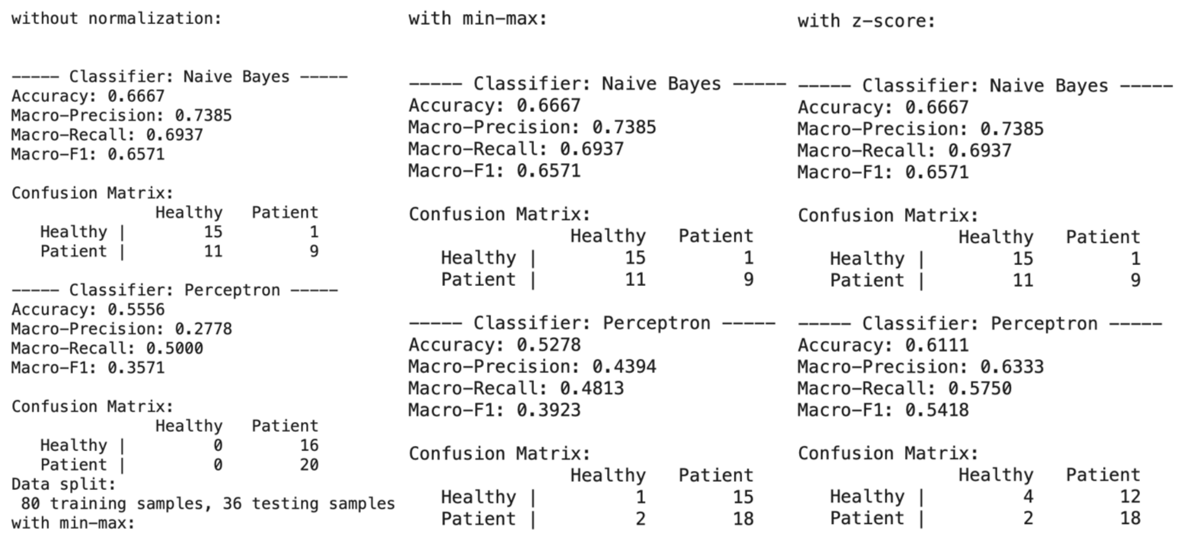


```
Data split:
 80 training samples, 36 testing samples
without min-max:


----- Classifier: Naive Bayes -----
Accuracy: 0.6667
Macro-Precision: 0.7385
Macro-Recall: 0.6937
Macro-F1: 0.6571

Confusion Matrix:
               Healthy   Patient
   Healthy |        15         1
   Patient |        11         9

----- Classifier: Perceptron -----
Accuracy: 0.5556
Macro-Precision: 0.2778
Macro-Recall: 0.5000
Macro-F1: 0.3571

Confusion Matrix:
               Healthy   Patient
   Healthy |         0        16
   Patient |         0        20
```

```
Data split:
 80 training samples, 36 testing samples
with min-max:


----- Classifier: Naive Bayes -----
Accuracy: 0.6667
Macro-Precision: 0.7385
Macro-Recall: 0.6937
Macro-F1: 0.6571

Confusion Matrix:
               Healthy   Patient
   Healthy |        15         1
   Patient |        11         9

----- Classifier: Perceptron -----
Accuracy: 0.5278
Macro-Precision: 0.4394
Macro-Recall: 0.4813
Macro-F1: 0.3923

Confusion Matrix:
               Healthy   Patient
   Healthy |         1        15
   Patient |         2        18
```

■ <u>Naïve Bayes vs. Perceptron (tarin-test split ratio=0.7, z-score, Breast Cancer Coimbra)</u>

Expecting **Z-Score Standardization be better than Min-Max**. Min-max scaling is very sensitive to outliers, because the min and max values are directly used in the calculation. Just one extreme value can stretch or squash the rest of the data. Z-score scaling uses the mean and standard deviation, so it's **more robust to moderate outliers**. Min-max compresses features into a small range (like [0, 1]), which can flatten variation in features or dampen the importance of features with useful spread. Z-score scaling **preserves the shape of the original distribution** better, just on a normalized scale. If your features don't have clear natural bounds (like lab measurements, economic indicators), min-max might squash useful info into narrow bands. Standard scaler **doesn't assume any known boundaries**, it adapts to the spread of data more intelligently. Some models assume inputs follow roughly **Gaussian distributions. Z-scaling nudges your data closer to that behavior,** even if it's not perfectly normal. Gradient descent (used by Perceptron) performs best when: features are centered around 0 and all features have comparable variance. Z-score standardization checks both boxes. Min-max only aligns scales, not centers or variances.



```
without normalization:


----- Classifier: Naive Bayes -----
Accuracy: 0.6667
Macro-Precision: 0.7385
Macro-Recall: 0.6937
Macro-F1: 0.6571

Confusion Matrix:
               Healthy   Patient
   Healthy |        15         1
   Patient |        11         9

----- Classifier: Perceptron -----
Accuracy: 0.5556
Macro-Precision: 0.2778
Macro-Recall: 0.5000
Macro-F1: 0.3571

Confusion Matrix:
               Healthy   Patient
   Healthy |         0        16
   Patient |         0        20
Data split:
 80 training samples, 36 testing samples
with min-max:
```

```
with min-max:


----- Classifier: Naive Bayes -----
Accuracy: 0.6667
Macro-Precision: 0.7385
Macro-Recall: 0.6937
Macro-F1: 0.6571

Confusion Matrix:
               Healthy   Patient
   Healthy |        15         1
   Patient |        11         9

----- Classifier: Perceptron -----
Accuracy: 0.5278
Macro-Precision: 0.4394
Macro-Recall: 0.4813
Macro-F1: 0.3923

Confusion Matrix:
               Healthy   Patient
   Healthy |         1        15
   Patient |         2        18
```

```
with z-score:


----- Classifier: Naive Bayes -----
Accuracy: 0.6667
Macro-Precision: 0.7385
Macro-Recall: 0.6937
Macro-F1: 0.6571

Confusion Matrix:
               Healthy   Patient
   Healthy |        15         1
   Patient |        11         9

----- Classifier: Perceptron -----
Accuracy: 0.6111
Macro-Precision: 0.6333
Macro-Recall: 0.5750
Macro-F1: 0.5418

Confusion Matrix:
               Healthy   Patient
   Healthy |         4        12
   Patient |         2        18
```

As expected, **Naïve bayes ignores scaling** since it models each feature distribution independently using class-conditional probabilities. The math stays the same regardless of transformation. For **perceptron, Z-Score standardization clearly helps.** Z-Score is better here because, centered data (mean = 0) improves gradient descent. Also, standardizes variance across features avoids one feature dominating. And it avoids distortion from outliers (better than min-max). Also, matches what perceptron expects: consistent scale and distribution.

Appendix (Code)

```python
import numpy as np
import matplotlib.pyplot as plt
from urllib.request import urlopen
import pandas as pd
from io import StringIO

class NaiveBayesClassifier:
    def __init__(self):
        self.class_priors = None
        self.feature_means = None
        self.feature_vars = None
        self.classes = None

    def train(self, X, y):
        """
        Train the Naive Bayes classifier

        Parameters:
        X: Training data features [n_samples, n_features]
        y: Training data labels [n_samples]
        """
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)

        # Initialize parameters
        self.class_priors = np.zeros(n_classes)
        self.feature_means = np.zeros((n_classes, n_features))
        self.feature_vars = np.zeros((n_classes, n_features))

        # Calculate class priors and feature statistics for each class
        for i, c in enumerate(self.classes):
            X_c = X[y == c]
            self.class_priors[i] = X_c.shape[0] / n_samples
            self.feature_means[i, :] = X_c.mean(axis=0)
            self.feature_vars[i, :] = X_c.var(axis=0) + 1e-6  # Add small value to
avoid zero variance

    def _calculate_likelihood(self, X):
        """
        Calculate likelihood of the data under each class
        Parameters:
        X: Test data features [n_samples, n_features]

        Returns:
        likelihoods: Likelihood for each sample under each class [n_samples,
n_classes]
        """
        n_samples, n_features = X.shape
        n_classes = len(self.classes)
        likelihoods = np.zeros((n_samples, n_classes))

        for i in range(n_classes):
            # Gaussian probability density
            deviations = X - self.feature_means[i, :]
            exponent = -0.5 * np.sum(deviations**2 / self.feature_vars[i, :], axis=1)
            normalizer = 1 / np.sqrt((2 * np.pi) ** n_features *
np.prod(self.feature_vars[i, :]))
            likelihoods[:, i] = normalizer * np.exp(exponent)

        return likelihoods

    def predict(self, X):
        """
        Predict class labels and calculate discriminant functions

        Parameters:
        X: Test data features [n_samples, n_features]

        Returns:
        predicted_classes: Predicted class labels [n_samples]
        discriminant_values: Values of discriminant functions [n_samples, n_classes]
        """
        likelihoods = self._calculate_likelihood(X)
        # Calculate posterior probabilities (discriminant functions)
        discriminant_values = likelihoods * self.class_priors

        # Normalize to get proper probabilities (optional)
        discriminant_values = discriminant_values / np.sum(discriminant_values,
axis=1, keepdims=True)

        # Get predicted class (maximum posterior)
```

```python
        predicted_indices = np.argmax(discriminant_values, axis=1)
        predicted_classes = self.classes[predicted_indices]

        return predicted_classes, discriminant_values

class PerceptronClassifier:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        """
        Initialize Perceptron classifier

        Parameters:
        learning_rate: Learning rate for weight updates
        n_iterations: Maximum number of iterations
        """
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None
        self.classes = None

    def _initialize_weights(self, n_features, n_classes):
        """Initialize weights and bias"""
        if n_classes == 2:
            # Binary classification: One set of weights
            self.weights = np.zeros(n_features)
            self.bias = 0
        else:
            # Multi-class: One set of weights per class
            self.weights = np.zeros((n_classes, n_features))
            self.bias = np.zeros(n_classes)

    def train(self, X, y):
        """
        Train the Perceptron classifier
        Parameters:
        X: Training data features [n_samples, n_features]
        y: Training data labels [n_samples]
        """
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)
        # Map class labels to integers starting from 0
        y_mapped = np.zeros_like(y, dtype=int)
        for i, c in enumerate(self.classes):
            y_mapped[y == c] = i

        # Initialize weights
        self._initialize_weights(n_features, n_classes)

        # Train the model
        if n_classes == 2:
            # Binary classification
            for _ in range(self.n_iterations):  # Fixed: removed asterisk
                for idx, x_i in enumerate(X):
                    # Convert class 0 to -1 for binary classification
                    y_i = 1 if y_mapped[idx] == 1 else -1

                    # Calculate activation
                    activation = np.dot(x_i, self.weights) + self.bias

                    # Update weights if misclassified
                    if y_i * activation <= 0:
                        self.weights += self.learning_rate * y_i * x_i
                        self.bias += self.learning_rate * y_i
        else:
            # Multi-class classification (one-vs-rest)
            for _ in range(self.n_iterations):  # Fixed: removed asterisk
                for idx, x_i in enumerate(X):
                    y_true = y_mapped[idx]

                    # Calculate activations for all classes
                    activations = np.dot(x_i, self.weights.T) + self.bias  # Fixed dot
product orientation
                    y_pred = np.argmax(activations)

                    # Update weights if misclassified
                    if y_pred != y_true:
                        self.weights[y_true] += self.learning_rate * x_i
                        self.weights[y_pred] -= self.learning_rate * x_i
                        self.bias[y_true] += self.learning_rate
                        self.bias[y_pred] -= self.learning_rate

    def predict(self, X):
        """
        Predict class labels and calculate discriminant functions

        Parameters:
        X: Test data features [n_samples, n_features]

        Returns:
        predicted_classes: Predicted class labels [n_samples]
        discriminant_values: Values of discriminant functions [n_samples, n_classes]
        """
        n_classes = len(self.classes)

        if n_classes == 2:
            # Binary classification
            discriminant_values = np.column_stack([
                -np.dot(X, self.weights) - self.bias,  # Class 0
                np.dot(X, self.weights) + self.bias     # Class 1
            ])
            predicted_indices = np.argmax(discriminant_values, axis=1)

        else:
            # Multi-class classification
            discriminant_values = np.dot(X, self.weights.T) + self.bias
            predicted_indices = np.argmax(discriminant_values, axis=1)

        predicted_classes = self.classes[predicted_indices]

        return predicted_classes, discriminant_values


class ClassifierEvaluator:
    def __init__(self):
        pass

    def compute_confusion_matrix(self, y_true, y_pred, classes=None):
        """
        Compute confusion matrix

        Parameters:
        y_true: True class labels
        y_pred: Predicted class labels
        classes: List of class labels (if None, will be computed from the data)
        Returns:
        confusion_matrix: Confusion matrix [n_classes, n_classes]
        """
        if classes is None:
            classes = np.unique(np.concatenate((y_true, y_pred)))

        n_classes = len(classes)
        confusion_mat = np.zeros((n_classes, n_classes), dtype=int)

        for i in range(len(y_true)):
            true_idx = np.where(classes == y_true[i])[0][0]
            pred_idx = np.where(classes == y_pred[i])[0][0]
            confusion_mat[true_idx, pred_idx] += 1

        return confusion_mat, classes

    def compute_metrics(self, confusion_matrix):
        """
        Compute accuracy, precision, recall, and F1 score from confusion matrix

        Parameters:
        confusion_matrix: Confusion matrix [n_classes, n_classes]

        Returns:
        metrics_dict: Dictionary containing metrics
        """
        n_classes = confusion_matrix.shape[0]
        metrics = {}

        # Overall accuracy
        metrics['accuracy'] = np.sum(np.diag(confusion_matrix)) /
np.sum(confusion_matrix)

        # Per-class metrics
        metrics['precision'] = np.zeros(n_classes)
        metrics['recall'] = np.zeros(n_classes)
        metrics['f1_score'] = np.zeros(n_classes)

        for i in range(n_classes):
            # Precision
            if np.sum(confusion_matrix[:, i]) > 0:
                metrics['precision'][i] = confusion_matrix[i, i] / 
np.sum(confusion_matrix[:, i])
            else:
                metrics['precision'][i] = 0

            # Recall
            if np.sum(confusion_matrix[i, :]) > 0:
                metrics['recall'][i] = confusion_matrix[i, i] / 
np.sum(confusion_matrix[i, :])
            else:
                metrics['recall'][i] = 0

            # F1 score
            if metrics['precision'][i] + metrics['recall'][i] > 0:
                metrics['f1_score'][i] = 2 * metrics['precision'][i] * 
metrics['recall'][i] / (metrics['precision'][i] + metrics['recall'][i])
            else:
                metrics['f1_score'][i] = 0

        # Macro-averaged metrics
        metrics['macro_precision'] = np.mean(metrics['precision'])
        metrics['macro_recall'] = np.mean(metrics['recall'])
        metrics['macro_f1'] = np.mean(metrics['f1_score'])

        return metrics

    def plot_confusion_matrix(self, confusion_matrix, class_names, title='Confusion 
Matrix'):
        """
        Plot confusion matrix

        Parameters:
        confusion_matrix: Confusion matrix [n_classes, n_classes]
        class_names: Names of classes
        title: Title of the plot
        """
        plt.figure(figsize=(8, 6))
        plt.imshow(confusion_matrix, cmap='Blues')
        plt.title(title)
        plt.colorbar()

        n_classes = len(class_names)
        tick_marks = np.arange(n_classes)
        plt.xticks(tick_marks, class_names, rotation=45)
        plt.yticks(tick_marks, class_names)

        # Add text annotations
        thresh = confusion_matrix.max() / 2
        for i in range(n_classes):
            for j in range(n_classes):
                plt.text(j, i, format(confusion_matrix[i, j], 'd'),
                         ha="center", va="center",
                         color="white" if confusion_matrix[i, j] > thresh else "black")

        plt.ylabel('True Class')
        plt.xlabel('Predicted Class')
        plt.tight_layout()

    def calculate_roc_curve(self, y_true_binary, discriminant_values_positive,
n_points=100):
        """
        Calculate ROC curve for binary classification

        Parameters:
        y_true_binary: Binary true class labels (0 or 1)
        discriminant_values_positive: Discriminant values for the positive class
        n_points: Number of threshold points for the ROC curve

        Returns:
        fpr_values: False positive rates
        tpr_values: True positive rates
        auc: Area under the ROC curve
        """
        # Convert labels to binary (0 or 1)
        y_true_binary = np.array(y_true_binary).astype(int)

        # Get positive and negative scores
        pos_scores = discriminant_values_positive[y_true_binary == 1]
        neg_scores = discriminant_values_positive[y_true_binary == 0]

        # Calculate ROC curve
        thresholds = np.linspace(np.min(discriminant_values_positive),
                                 np.max(discriminant_values_positive), n_points)
        fpr_values = []
        tpr_values = []

        for threshold in thresholds:
            # True positive rate
            tp = np.sum(pos_scores >= threshold)
            fn = np.sum(pos_scores < threshold)
            tpr = tp / (tp + fn) if (tp + fn) > 0 else 0

            # False positive rate
            fp = np.sum(neg_scores >= threshold)
            tn = np.sum(neg_scores < threshold)
            fpr = fp / (fp + tn) if (fp + tn) > 0 else 0

            fpr_values.append(fpr)
            tpr_values.append(tpr)

        # Calculate AUC using trapezoidal rule
        fpr_values = np.array(fpr_values)
        tpr_values = np.array(tpr_values)

        # Sort by increasing FPR
        sorted_indices = np.argsort(fpr_values)
        fpr_values = fpr_values[sorted_indices]
        tpr_values = tpr_values[sorted_indices]

        # Add endpoints if needed
        if fpr_values[0] > 0 or tpr_values[0] > 0:
            fpr_values = np.concatenate(([0], fpr_values))
            tpr_values = np.concatenate(([0], tpr_values))

        if fpr_values[-1] < 1 or tpr_values[-1] < 1:
            fpr_values = np.concatenate((fpr_values, [1]))
            tpr_values = np.concatenate((tpr_values, [1]))

        auc = np.trapezoid(tpr_values, fpr_values)

        return fpr_values, tpr_values, auc

    def plot_roc_curve(self, fpr_values, tpr_values, auc, label=None):
        """
        Plot ROC curve
        Parameters:
        fpr_values: False positive rates
        tpr_values: True positive rates
        auc: Area under the ROC curve
        label: Label for the curve
        """
        if label is None:
            label = f'AUC = {auc:.3f}'
        else:
            label = f'{label} (AUC = {auc:.3f})'

        plt.plot(fpr_values, tpr_values, label=label)
        plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('Receiver Operating Characteristic (ROC) Curve')
        plt.legend(loc='lower right')
        plt.grid(True, alpha=0.3)

def load_iris_dataset():
    """Load Iris dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"
    column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, :-1].values
        y = df.iloc[:, -1].values
        return X, y, column_names[:-1], np.unique(y)
    except:
        print("Error loading Iris dataset from URL. Using synthetic data instead.")
        # Create synthetic Iris data if URL fails
        from sklearn.datasets import load_iris
        iris = load_iris()
        X = iris.data
        y = np.array(['Iris-setosa', 'Iris-versicolor', 'Iris-
virginica'])[iris.target]
        return X, y, iris.feature_names, np.unique(y)

def load_breast_cancer_coimbra_dataset():
    """Load Breast Cancer Coimbra dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/00451/dataR2.csv"
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data))
        X = df.iloc[:, :-1].values  # Features
        y = df.iloc[:, -1].values   # Classification column
        # Convert class to strings for consistency
        y = np.array(['Healthy' if label == 1 else 'Patient' for label in y])
        feature_names = df.columns[:-1].tolist()
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Breast Cancer Coimbra dataset from URL. Generating
synthetic data instead.")
        # Generate synthetic data if URL fails
        np.random.seed(42)
        n_samples = 116  # Actual dataset size
        n_features = 9   # Actual number of features
        X = np.random.randn(n_samples, n_features)
        y = np.array(['Healthy' if i < 58 else 'Patient' for i in range(n_samples)])
        feature_names = [
            'Age', 'BMI', 'Glucose', 'Insulin', 'HOMA', 'Leptin',
            'Adiponectin', 'Resistin', 'MCP.1'
        ]
        class_names = np.unique(y)
        return X, y, feature_names, class_names

def load_ionosphere_dataset():
    """Load Ionosphere dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/ionosphere/ionosphere.data"
    column_names = [f'feature_{i}' for i in range(34)] + ['class']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, :-1].values
        y = df.iloc[:, -1].values
        feature_names = column_names[:-1]
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Ionosphere dataset from URL. Using synthetic data
instead.")
        # Create synthetic ionosphere data if URL fails
        X = np.random.randn(351, 34)
        y = np.random.choice(['g', 'b'], size=351)
        feature_names = [f'feature_{i}' for i in range(34)]
        return X, y, feature_names, np.unique(y)

def load_wine_dataset():
    """Load Wine dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data"
    column_names = ['class', 'alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash',
'magnesium',
                   'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
'proanthocyanins',
                   'color_intensity', 'hue', 'od280/od315_of_diluted_wines',
'proline']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, 1:].values  # Features
        y = df.iloc[:, 0].values   # Class column
        # Convert class to strings for consistency
        y = np.array([f'Class_{int(label)}' for label in y])
        feature_names = column_names[1:]
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Wine dataset from URL. Using synthetic data instead.")
        # Create synthetic wine data if URL fails
        from sklearn.datasets import load_wine
        wine = load_wine()
```

```python
        X = wine.data
        y = np.array([f'Class_{i+1}' for i in wine.target])
        return X, y, wine.feature_names, np.unique(y)

def split_data(X, y, train_ratio=0.7, random_state=42):
    """
    Split data into training and testing sets with similar class distributions

    Parameters:
    X: Features
    y: Labels
    train_ratio: Ratio of training data
    random_state: Random seed for reproducibility

    Returns:
    X_train, X_test, y_train, y_test
    """
    np.random.seed(random_state)

    # Get unique classes and their indices
    classes = np.unique(y)
    indices_per_class = [np.where(y == c)[0] for c in classes]

    # Split indices for each class
    train_indices = []
    test_indices = []

    for class_indices in indices_per_class:
        np.random.shuffle(class_indices)
        n_train = int(len(class_indices) * train_ratio)

        train_indices.extend(class_indices[:n_train])
        test_indices.extend(class_indices[n_train:])

    # Get train/test splits
    X_train = X[train_indices]
    X_test = X[test_indices]
    y_train = y[train_indices]
    y_test = y[test_indices]

    return X_train, X_test, y_train, y_test
```

```
def cross_validation(X, y, n_folds=5, random_state=42):
    """
    Perform k-fold cross-validation

    Parameters:
    X: Features
    y: Labels
    classifier: Classifier object with train and predict methods
    n_folds: Number of folds
    random_state: Random seed for reproducibility

    Returns:
    X_train, X_test, y_train, y_test
    #mean_accuracy: Mean accuracy across folds
    #std_accuracy: Standard deviation of accuracy across folds
    """
    np.random.seed(random_state)

    # Get indices for each class
    classes = np.unique(y)
    indices_per_class = [np.where(y == c)[0] for c in classes]

    # Create stratified folds
    fold_indices = [[] for _ in range(n_folds)]

    for class_indices in indices_per_class:
        np.random.shuffle(class_indices)
        # Split class indices into n_folds parts
        fold_size = len(class_indices) // n_folds

        for fold_idx in range(n_folds):
            start_idx = fold_idx * fold_size
            end_idx = (fold_idx + 1) * fold_size if fold_idx < n_folds - 1 else
len(class_indices)
            fold_indices[fold_idx].extend(class_indices[start_idx:end_idx])

    # Run cross-validation
    accuracies = []

    for test_fold in range(n_folds):
        # Create train/test split
        test_indices = fold_indices[test_fold]
```

```python
        train_indices = []
        for fold_idx in range(n_folds):
            if fold_idx != test_fold:
                train_indices.extend(fold_indices[fold_idx])

        X_train = X[train_indices]
        y_train = y[train_indices]
        X_test = X[test_indices]
        y_test = y[test_indices]

    return X_train, X_test, y_train, y_test

    """
        # Train and evaluate classifier
        classifier.train(X_train, y_train)
        y_pred, _ = classifier.predict(X_test)

        # Calculate accuracy
        accuracy = np.mean(y_pred == y_test)
        accuracies.append(accuracy)

    return np.mean(accuracies), np.std(accuracies)
    """


def normalize_data(X_train, X_test):
    """
    Normalize data using min-max scaling based on training data

    Parameters:
    X_train: Training features
    X_test: Testing features

    Returns:
    X_train_norm: Normalized training features
    X_test_norm: Normalized testing features
    """
    # Convert boolean values to integers before normalization
    X_train = X_train.astype(float)
    X_test = X_test.astype(float)

    # Calculate min and max values from training data
    min_vals = np.min(X_train, axis=0)
    max_vals = np.max(X_train, axis=0)
    range_vals = max_vals - min_vals

    # Avoid division by zero
    range_vals[range_vals == 0] = 1

    # Normalize
    X_train_norm = (X_train - min_vals) / range_vals
    X_test_norm = (X_test - min_vals) / range_vals

    return X_train_norm, X_test_norm

import numpy as np

def standardiize_data(X_train, X_test):
    """
    Normalize data using Z-score standardization based on training data

    Parameters:
    X_train: Training features
    X_test: Testing features

    Returns:
    X_train_norm: Standardized training features
    X_test_norm: Standardized testing features
    """
    # Convert boolean values to floats before standardization
    X_train = X_train.astype(float)
    X_test = X_test.astype(float)

    # Calculate mean and standard deviation from training data
    mean_vals = np.mean(X_train, axis=0)
    std_vals = np.std(X_train, axis=0)

    # Avoid division by zero
    std_vals[std_vals == 0] = 1

    # Standardize
    X_train_norm = (X_train - mean_vals) / std_vals
    X_test_norm = (X_test - mean_vals) / std_vals

    return X_train_norm, X_test_norm
```

```
def main():
    print("Classifier Evaluation Program")
    print("------------------------------")

    # Load datasets
    print("\nLoading datasets...")


    # Breast Cancer Coimbra dataset (binary)
    X_bc, y_bc, feature_names_bc, class_names_bc =
load_breast_cancer_coimbra_dataset()
    print(f"Breast Cancer Coimbra dataset loaded: {X_bc.shape[0]} samples,
{X_bc.shape[1]} features, {len(class_names_bc)} classes")


    # Ionosphere (binary)
    X_ion, y_ion, feature_names_ion, class_names_ion = load_ionosphere_dataset()
    print(f"Ionosphere dataset loaded: {X_ion.shape[0]} samples, {X_ion.shape[1]}
features, {len(class_names_ion)} classes")


    # Iris dataset (multi-class)
    X_iris, y_iris, feature_names_iris, class_names_iris = load_iris_dataset()
    print(f"Iris dataset loaded: {X_iris.shape[0]} samples, {X_iris.shape[1]}
features, {len(class_names_iris)} classes")

    # Wine dataset (multi-class)
    X_wine, y_wine, feature_names_wine, class_names_wine = load_wine_dataset()
    print(f"Wine dataset loaded: {X_wine.shape[0]} samples, {X_wine.shape[1]}
features, {len(class_names_wine)} classes")

    # Process datasets
    datasets = [
        {
            'name': 'Breast Cancer Coimbra',
            'X': X_bc,
            'y': y_bc,
            'feature_names': feature_names_bc,
            'class_names': class_names_bc,
            'binary': True
        },
```

```python
    {
        'name': 'Ionosphere',
        'X': X_ion,
        'y': y_ion,
        'feature_names': feature_names_ion,
        'class_names': class_names_ion,
        'binary': True
    },
    {
        'name': 'Iris',
        'X': X_iris,
        'y': y_iris,
        'feature_names': feature_names_iris,
        'class_names': class_names_iris,
        'binary': False
    },
    {
        'name': 'Wine',
        'X': X_wine,
        'y': y_wine,
        'feature_names': feature_names_wine,
        'class_names': class_names_wine,
        'binary': False
    }
]


# Create classifier instances
classifiers = [
    {
        'name': 'Naive Bayes',
        'model': NaiveBayesClassifier()
    },
    {
        'name': 'Perceptron',
        'model': PerceptronClassifier(learning_rate=0.01, n_iterations=1000)
    }
]

# Create evaluator
evaluator = ClassifierEvaluator()

# Store ROC results
```

```
    roc_results = []


    # Process each dataset
    for dataset in datasets:
        print(f"\n\n===== Dataset: {dataset['name']} =====")

        for i in range(0,3):
          # Split data
          X_train, X_test, y_train, y_test = split_data(dataset['X'], dataset['y'],
train_ratio=0.7)
          print(f"Data split: \n {X_train.shape[0]} training samples,
{X_test.shape[0]} testing samples")


        #for k  in range(5,11):
          # Cross-validation
          #X_train, X_test, y_train, y_test = cross_validation(dataset['X'],
dataset['y'], n_folds=k)
          #print(f"\nCross-Validation ({k}-fold): \n{X_train.shape[0]} training
samples, {X_test.shape[0]} testing samples")

          # Without normalization
          if (i==0):
            print(f"without normalization: \n")
            X_train_norm = X_train
            X_test_norm = X_test
          # Normalize data
          elif(i==1):
            print(f"with min-max: \n")
            X_train_norm, X_test_norm = normalize_data(X_train, X_test)
          else:
            print(f"with z-score: \n")
            X_train_norm, X_test_norm = standardiize_data(X_train, X_test)

          # Evaluate each classifier
          for clf_info in classifiers:
              classifier = clf_info['model']
              clf_name = clf_info['name']

              print(f"\n----- Classifier: {clf_name} -----")
              # Train classifier
              classifier.train(X_train_norm, y_train)

              # Test classifier
              y_pred, discriminant_values = classifier.predict(X_test_norm)

              # Compute confusion matrix
              conf_matrix, classes = evaluator.compute_confusion_matrix(y_test,
y_pred, dataset['class_names'])

              # Compute metrics
              metrics = evaluator.compute_metrics(conf_matrix)

              # Display results
              print(f"Accuracy: {metrics['accuracy']:.4f}")
              print(f"Macro-Precision: {metrics['macro_precision']:.4f}")
              print(f"Macro-Recall: {metrics['macro_recall']:.4f}")
              print(f"Macro-F1: {metrics['macro_f1']:.4f}")

              # Display confusion matrix
              print("\nConfusion Matrix:")
              print(" " * 12, end="")
              for c in classes:
                  print(f"{c[:7]:>10}", end="")
              print()

              for i, c in enumerate(classes):
                  print(f"{c[:10]:>10} |", end="")
                  for j in range(len(classes)):
                      print(f"{conf_matrix[i, j]:>10}", end="")
                  print()


              # Plot confusion matrix
              evaluator.plot_confusion_matrix(conf_matrix, classes,
title=f'{clf_name} Confusion Matrix - {dataset["name"]}')

              # For binary datasets, compute ROC curve
              if dataset['binary']:
                  # Convert class labels to binary (0 or 1)
                  binary_labels = np.zeros(len(y_test))
                  positive_class = dataset['class_names'][0]  # First class is
positive
                  binary_labels[y_test == positive_class] = 1

                  # Get discriminant values for positive class
                  if len(discriminant_values.shape) > 1 and
discriminant_values.shape[1] > 1:
                      # Multi-output discriminant (use first class)
                      disc_positive = discriminant_values[:, 0]
                  else:
                      # Single output discriminant
                      disc_positive = discriminant_values

                  # Calculate and plot ROC curve
                  fpr, tpr, auc = evaluator.calculate_roc_curve(binary_labels,
disc_positive)
                  print(f"\nAUC: {auc:.4f}")

                  # Store ROC results for later comparison
                  if not hasattr(main, 'roc_results'):
                      main.roc_results = []

                  main.roc_results.append({
                      'dataset': dataset['name'],
                      'classifier': clf_name,
                      'fpr': fpr,
                      'tpr': tpr,
                      'auc': auc
                  })

              # Plot ROC curves for binary dataset
              if hasattr(main, 'roc_results'):
                  plt.figure(figsize=(10, 8))

                  for result in main.roc_results:
                      label = f"{result['classifier']} - {result['dataset']}"
                      evaluator.plot_roc_curve(result['fpr'], result['tpr'],
result['auc'], label=label)

                  plt.title('ROC Curves Comparison')
                  plt.tight_layout()
              # Show all plots
              plt.show()


if __name__ == "__main__":
    main()
```