Final Report: Spam Classification

Author: Christian Laggui
Date: 4/6/2025
NJIT Email: cl623@njit.edu

1. Introduction

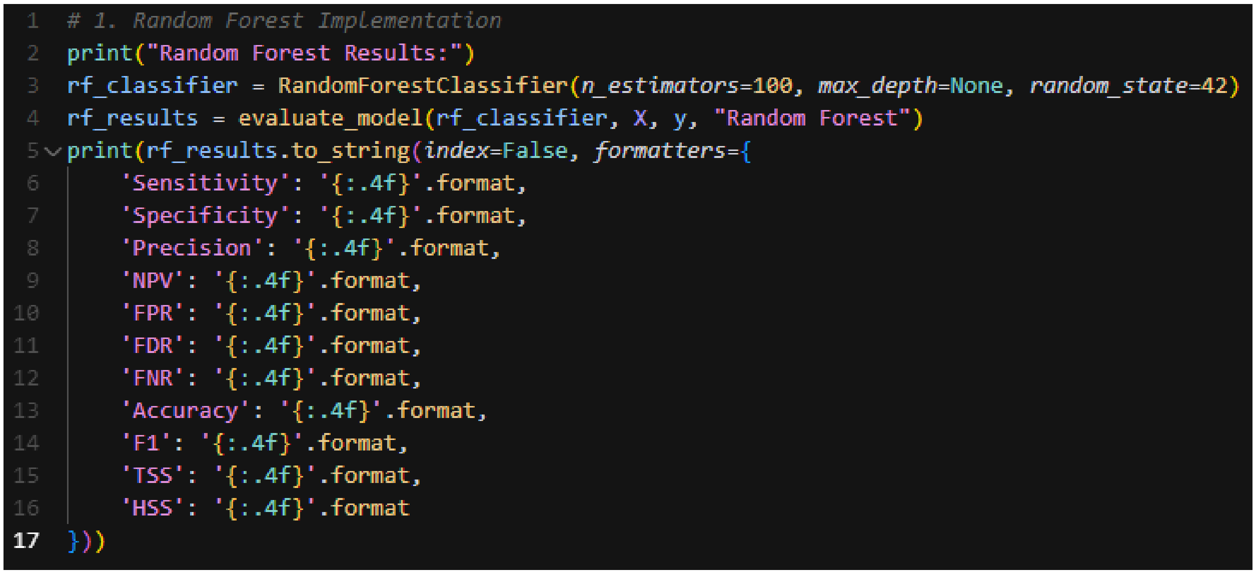
This project implements and evaluates three different classification algorithms (Random Forest, SVM, and GRU) to classify emails as either spam or not spam. The problem is to accurately classify emails as spam or not spam using machine learning techniques. The algorithms used are Random Forest, Support Vector Machine (SVM), and Gated Recurrent Unit (GRU). The dataset used is the Spambase Dataset from the UCI Machine Learning Repository.

- Dataset link: https://archive.ics.uci.edu/dataset/94/spambase
- Github Repo: https://github.com/cl623NJIT/634FinalProject

2. Algorithms Implemented

2.1 Random Forest

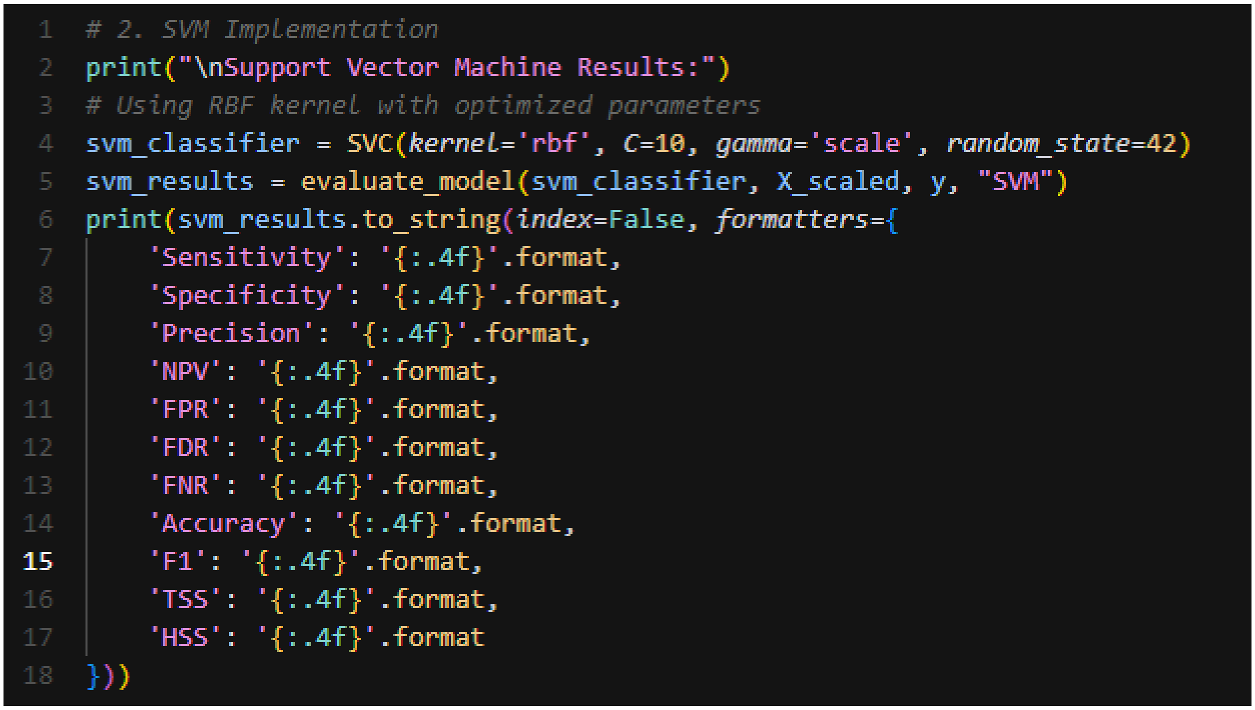
- Random Forest is an ensemble learning method that constructs a multitude of decision trees at training time and outputs the class that is the mode of the classes (classification) or the mean prediction (regression) of the individual trees.
- It is suitable for this task because it can handle high-dimensional data, capture nonlinear relationships, and is robust to overfitting.
- The parameters used are:
  - n_estimators: 100
  - max_depth: None
  - random_state: 42
- The number of trees was set to 100 to balance computational efficiency and performance. The max_depth was set to None to allow the trees to grow until all leaves are pure. The random_state was set to 42 for reproducibility of the results.



```python
# 1. Random Forest Implementation
print("Random Forest Results:")
rf_classifier = RandomForestClassifier(n_estimators=100, max_depth=None, random_state=42)
rf_results = evaluate_model(rf_classifier, X, y, "Random Forest")
print(rf_results.to_string(index=False, formatters={
    'Sensitivity': '{:.4f}'.format,
    'Specificity': '{:.4f}'.format,
    'Precision': '{:.4f}'.format,
    'NPV': '{:.4f}'.format,
    'FPR': '{:.4f}'.format,
    'FDR': '{:.4f}'.format,
    'FNR': '{:.4f}'.format,
    'Accuracy': '{:.4f}'.format,
    'F1': '{:.4f}'.format,
    'TSS': '{:.4f}'.format,
    'HSS': '{:.4f}'.format
}))
```

2.2 Support Vector Machine (SVM)

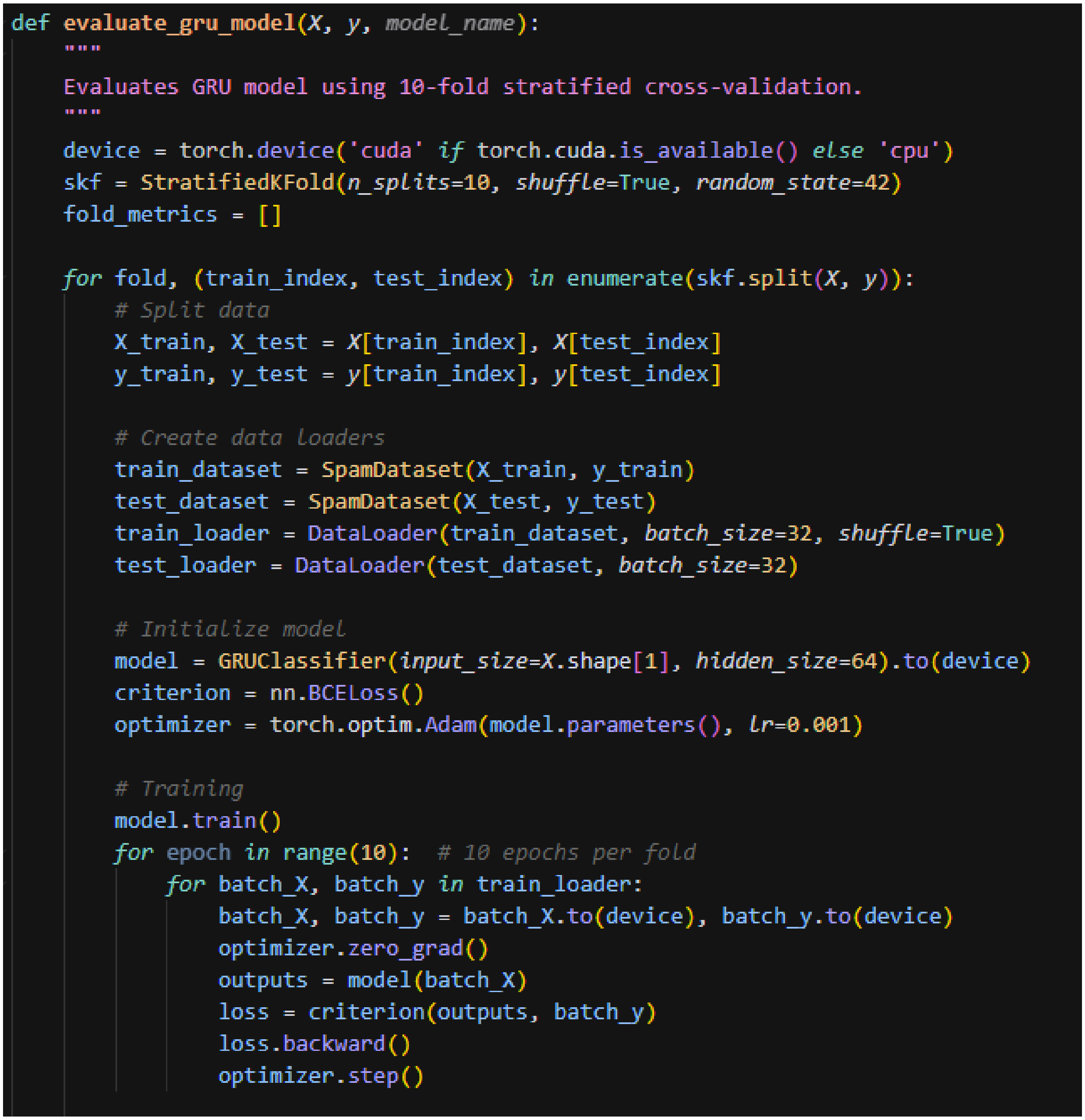
- SVM is a discriminative classifier formally defined by a separating hyperplane. Given labeled training data, the algorithm outputs an optimal hyperplane which categorizes new examples.
- It is suitable for this task because it is effective in high-dimensional spaces and can handle non-linear relationships through the use of kernels.
- The kernel used is the RBF kernel. The parameters are:
  - Kernel: RBF
  - C: 10
  - Gamma: ‘scale’
  - random_state: 42
- The RBF kernel was chosen for its ability to handle non-linear relationships. C was set to 10 to balance margin maximization and misclassification penalty. Gamma was set to 'scale' to automatically adjust based on the feature scale. The parameter random_state was set to 42 for reproducibility.
- Scaling the data is important for SVM because it calculates distances between data points, and features with larger scales can dominate the distance calculations, leading to suboptimal results.



```python
# 2. SVM Implementation
print("\nSupport Vector Machine Results:")
# Using RBF kernel with optimized parameters
svm_classifier = SVC(kernel='rbf', C=10, gamma='scale', random_state=42)
svm_results = evaluate_model(svm_classifier, X_scaled, y, "SVM")
print(svm_results.to_string(index=False, formatters={
    'Sensitivity': '{:.4f}'.format,
    'Specificity': '{:.4f}'.format,
    'Precision': '{:.4f}'.format,
    'NPV': '{:.4f}'.format,
    'FPR': '{:.4f}'.format,
    'FDR': '{:.4f}'.format,
    'FNR': '{:.4f}'.format,
    'Accuracy': '{:.4f}'.format,
    'F1': '{:.4f}'.format,
    'TSS': '{:.4f}'.format,
    'HSS': '{:.4f}'.format
}))
```

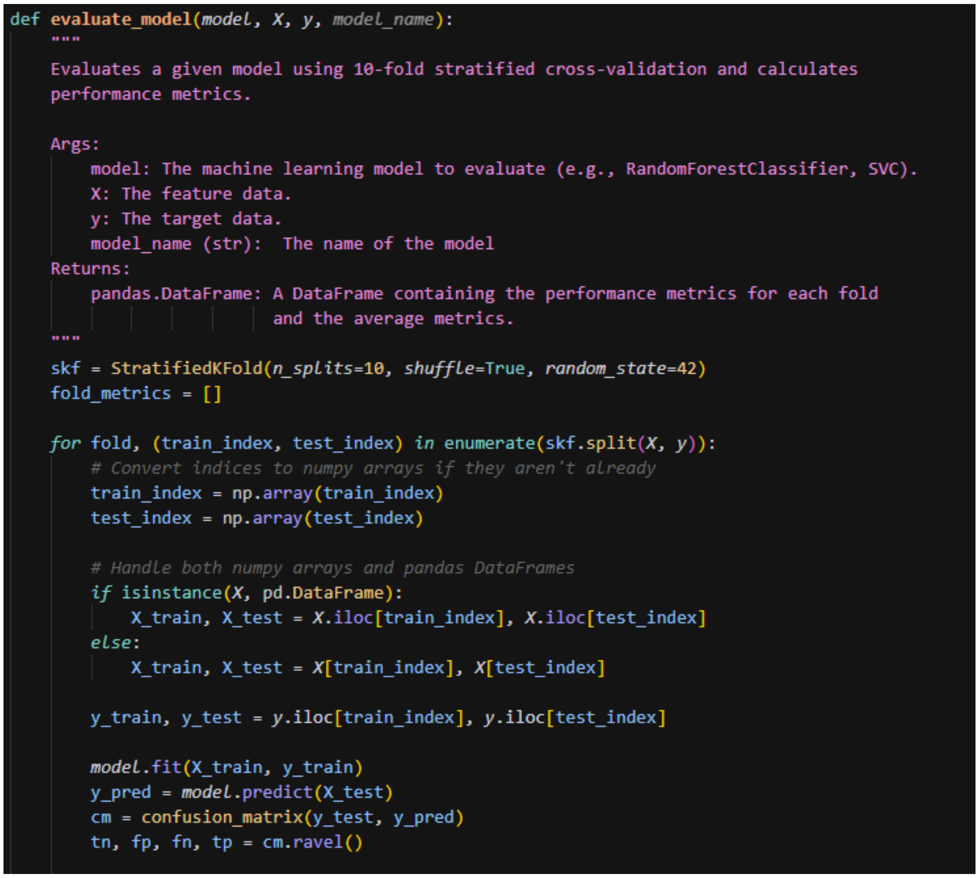
2.3 GRU (Gated Recurrent Unit)

- GRU is a type of recurrent neural network (RNN) that uses gates to control the flow of information, allowing it to selectively remember or forget information over long sequences.
- Even though the data is not strictly sequential, GRU can learn complex relationships between features. It can identify patterns and dependencies within the feature set that might indicate spam.
- The GRU model architecture consists of:
    - An embedding layer.
    - A GRU layer with 64 hidden units and 2 layers.
    - A dense output layer with a sigmoid activation function.
- The email features were scaled using StandardScaler() and then converted into PyTorch tensors. The GRU model expects input of shape (batch_size, sequence_length, input_size).
- The parameters used for GRU include an Adam optimizer with a learning rate of 0.001 and Binary Cross Entropy Loss. The number of epochs is 10, and the batch size is 32



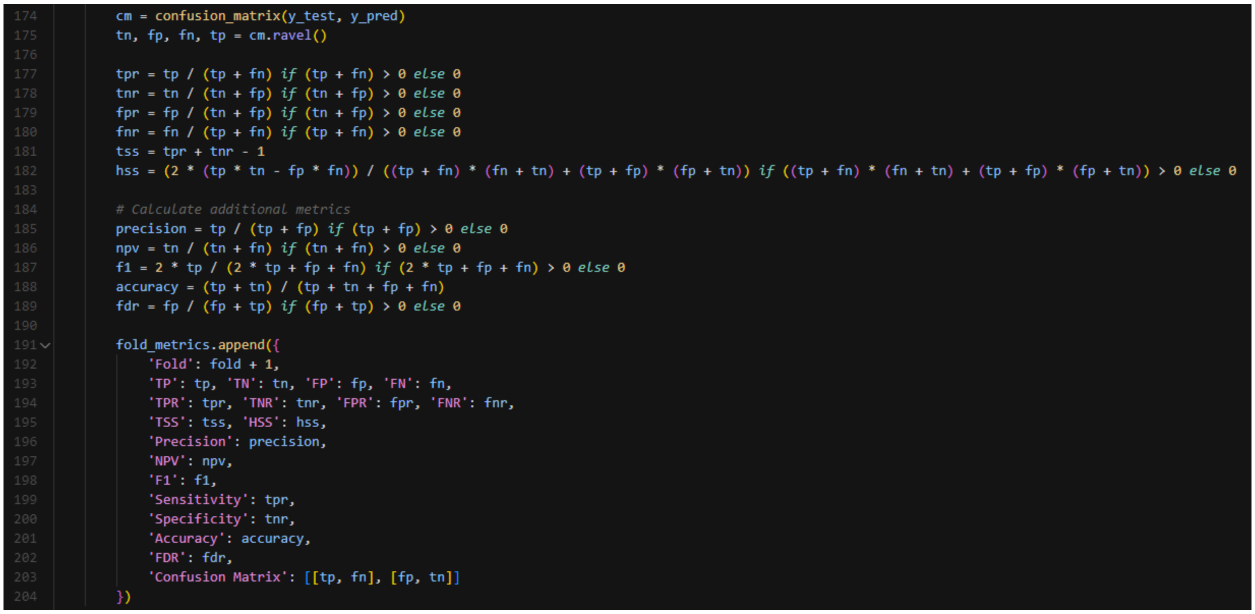
```
def evaluate_gru_model(X, y, model_name):
    """
    Evaluates GRU model using 10-fold stratified cross-validation.
    """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
    fold_metrics = []

    for fold, (train_index, test_index) in enumerate(skf.split(X, y)):
        # Split data
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Create data loaders
        train_dataset = SpamDataset(X_train, y_train)
        test_dataset = SpamDataset(X_test, y_test)
        train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
        test_loader = DataLoader(test_dataset, batch_size=32)

        # Initialize model
        model = GRUClassifier(input_size=X.shape[1], hidden_size=64).to(device)
        criterion = nn.BCELoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

        # Training
        model.train()
        for epoch in range(10):  # 10 epochs per fold
            for batch_X, batch_y in train_loader:
                batch_X, batch_y = batch_X.to(device), batch_y.to(device)
                optimizer.zero_grad()
                outputs = model(batch_X)
                loss = criterion(outputs, batch_y)
                loss.backward()
                optimizer.step()
```

Evaluating the Models using 10 Fold Cross Validation:



```python
def evaluate_model(model, X, y, model_name):
    """
    Evaluates a given model using 10-fold stratified cross-validation and calculates
    performance metrics.

    Args:
        model: The machine learning model to evaluate (e.g., RandomForestClassifier, SVC).
        X: The feature data.
        y: The target data.
        model_name (str):  The name of the model
    Returns:
        pandas.DataFrame: A DataFrame containing the performance metrics for each fold
                          and the average metrics.
    """
    skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
    fold_metrics = []

    for fold, (train_index, test_index) in enumerate(skf.split(X, y)):
        # Convert indices to numpy arrays if they aren't already
        train_index = np.array(train_index)
        test_index = np.array(test_index)

        # Handle both numpy arrays and pandas DataFrames
        if isinstance(X, pd.DataFrame):
            X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        else:
            X_train, X_test = X[train_index], X[test_index]

        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)
        tn, fp, fn, tp = cm.ravel()
```

Calculating the performance:



```python
        cm = confusion_matrix(y_test, y_pred)
        tn, fp, fn, tp = cm.ravel()

        tpr = tp / (tp + fn) if (tp + fn) > 0 else 0
        tnr = tn / (tn + fp) if (tn + fp) > 0 else 0
        fpr = fp / (tn + fp) if (tn + fp) > 0 else 0
        fnr = fn / (tp + fn) if (tp + fn) > 0 else 0
        tss = tpr + tnr - 1
        hss = (2 * (tp * tn - fp * fn)) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) if ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) > 0 else 0

        # Calculate additional metrics
        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        npv = tn / (tn + fn) if (tn + fn) > 0 else 0
        f1 = 2 * tp / (2 * tp + fp + fn) if (2 * tp + fp + fn) > 0 else 0
        accuracy = (tp + tn) / (tp + tn + fp + fn)
        fdr = fp / (fp + tp) if (fp + tp) > 0 else 0

        fold_metrics.append({
            'Fold': fold + 1,
            'TP': tp, 'TN': tn, 'FP': fp, 'FN': fn,
            'TPR': tpr, 'TNR': tnr, 'FPR': fpr, 'FNR': fnr,
            'TSS': tss, 'HSS': hss,
            'Precision': precision,
            'NPV': npv,
            'F1': f1,
            'Sensitivity': tpr,
            'Specificity': tnr,
            'Accuracy': accuracy,
            'FDR': fdr,
            'Confusion Matrix': [[tp, fn], [fp, tn]]
        })
```

3. Dataset

The Spambase dataset is a collection of email data used for spam classification. It contains a set of features that characterize emails, such as word frequencies, character frequencies, and other statistical measures. The dataset contains 4601 instances and 57 features which include frequencies of certain words and characters, the lengths of uninterrupted sequences of capital letters, and other statistical features. This dataset is appropriate for this classification problem because it provides a labeled dataset of spam and non-spam emails with a variety of features that can be used to distinguish between the two classes.

-**Dataset Link:** https://archive.ics.uci.edu/ml/datasets/Spambase

4. Experimental Setup

The experimental setup involves training and evaluating three different classification algorithms (Random Forest, SVM, and GRU) on the Spambase dataset using 10-fold stratified cross-validation. The performance of each algorithm is measured using a set of evaluation metrics, and the results are compared to determine the best-performing algorithm for spam classification.

10-fold cross-validation was chosen to provide a reliable estimate of the models' performance by averaging the results across multiple folds, reducing the risk of overfitting, and ensuring that each data point is used for both training and testing. Stratification ensures that the class distribution is maintained across the folds, which is important for imbalanced datasets.

Required Packages and How to Install them:

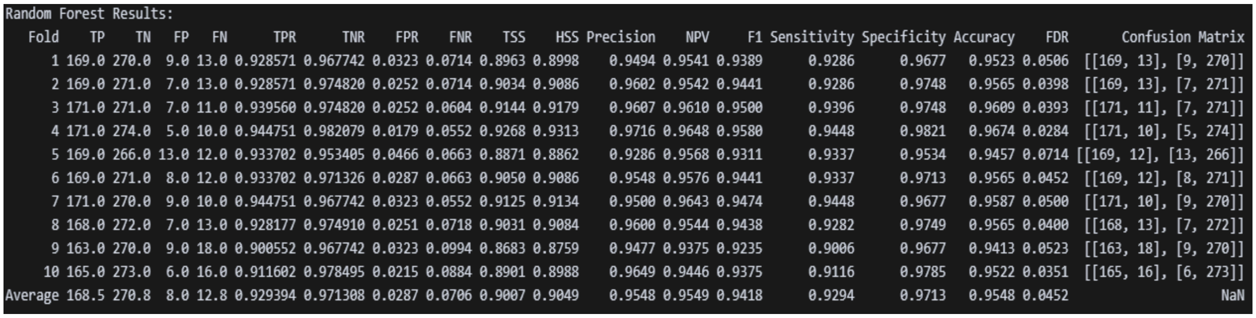
  - Python version: 3.9.13
  - scikit-learn version: 1.6.1
  - pandas version: 2.2.3
  - numpy version: 1.26.3
  - pytorch version: 2.6.0+cpu
- Installation of required packages:
  - pip install scikit-learn pandas numpy torch
- Execution of script:
  - Change directory to location of folder
  - Execute the following command:

```
python spam_classifier.py
```

- The code is designed to be run without any modifications, assuming the user has installed the required packages and placed the Spambase dataset in the specified location.

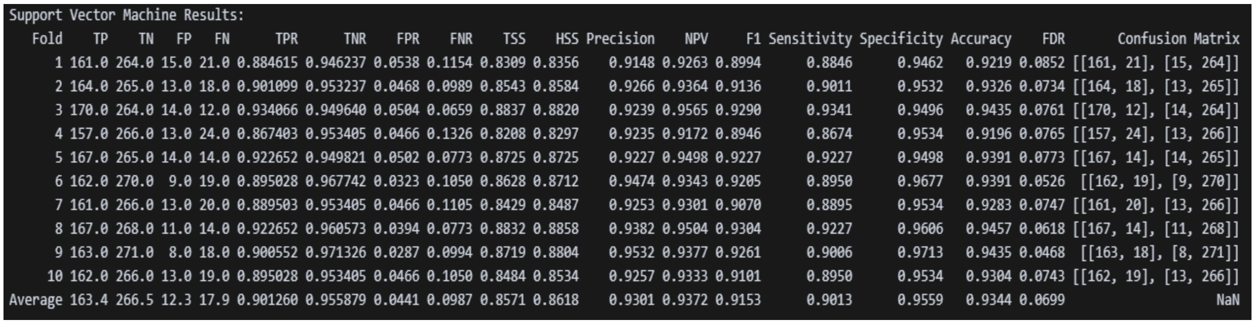
5. Results

- Random Forest Results (**6.9s**):



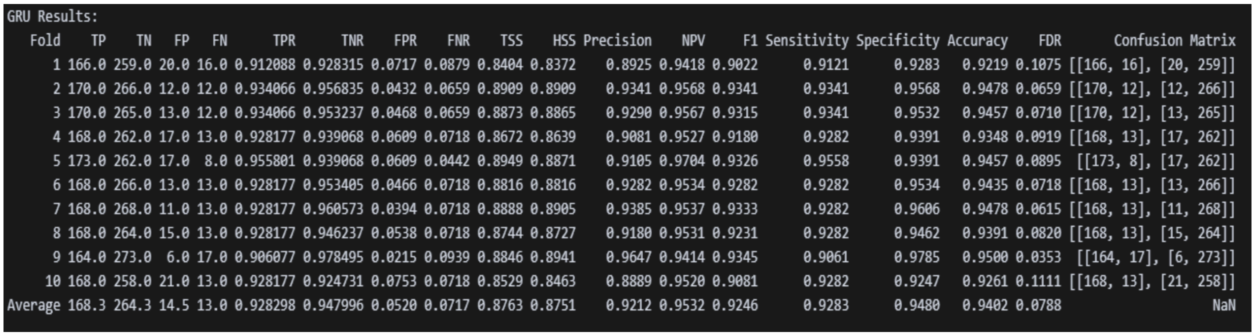
```
Random Forest Results:
   Fold    TP    TN   FP   FN      TPR      TNR    FPR    FNR    TSS    HSS Precision    NPV     F1 Sensitivity Specificity Accuracy    FDR       Confusion Matrix
      1 169.0 270.0  9.0 13.0 0.928571 0.967742 0.0323 0.0714 0.8963 0.8998    0.9494 0.9541 0.9389      0.9286      0.9677   0.9523 0.0506  [[169, 13], [9, 270]]
      2 169.0 271.0  7.0 13.0 0.928571 0.974820 0.0252 0.0714 0.9034 0.9086    0.9602 0.9542 0.9441      0.9286      0.9748   0.9565 0.0398  [[169, 13], [7, 271]]
      3 171.0 271.0  7.0 11.0 0.939560 0.974820 0.0252 0.0604 0.9144 0.9179    0.9607 0.9610 0.9500      0.9396      0.9748   0.9609 0.0393  [[171, 11], [7, 271]]
      4 171.0 274.0  5.0 10.0 0.944751 0.982079 0.0179 0.0552 0.9268 0.9313    0.9716 0.9648 0.9580      0.9448      0.9821   0.9674 0.0284  [[171, 10], [5, 274]]
      5 169.0 266.0 13.0 12.0 0.933702 0.953405 0.0466 0.0663 0.8871 0.8862    0.9286 0.9568 0.9311      0.9337      0.9534   0.9457 0.0714 [[169, 12], [13, 266]]
      6 169.0 271.0  8.0 12.0 0.933702 0.971326 0.0287 0.0663 0.9050 0.9086    0.9548 0.9576 0.9441      0.9337      0.9713   0.9565 0.0452  [[169, 12], [8, 271]]
      7 171.0 270.0  9.0 10.0 0.944751 0.967742 0.0323 0.0552 0.9125 0.9134    0.9500 0.9643 0.9474      0.9448      0.9677   0.9587 0.0500  [[171, 10], [9, 270]]
      8 168.0 272.0  7.0 13.0 0.928177 0.974910 0.0251 0.0718 0.9031 0.9084    0.9600 0.9544 0.9438      0.9282      0.9749   0.9565 0.0400  [[168, 13], [7, 272]]
      9 163.0 270.0  9.0 18.0 0.900552 0.967742 0.0323 0.0994 0.8683 0.8759    0.9477 0.9375 0.9235      0.9006      0.9677   0.9413 0.0523  [[163, 18], [9, 270]]
     10 165.0 273.0  6.0 16.0 0.911602 0.978495 0.0215 0.0884 0.8901 0.8988    0.9649 0.9446 0.9375      0.9116      0.9785   0.9522 0.0351  [[165, 16], [6, 273]]
Average 168.5 270.8  8.0 12.8 0.929394 0.971308 0.0287 0.0706 0.9007 0.9049    0.9548 0.9549 0.9418      0.9294      0.9713   0.9548 0.0452                    NaN
```

- SVM Results (**3.0s**):



```
Support Vector Machine Results:
   Fold    TP    TN   FP   FN      TPR      TNR    FPR    FNR    TSS    HSS Precision    NPV     F1 Sensitivity Specificity Accuracy    FDR       Confusion Matrix
      1 161.0 264.0 15.0 21.0 0.884615 0.946237 0.0538 0.1154 0.8309 0.8356    0.9148 0.9263 0.8994      0.8846      0.9462   0.9219 0.0852 [[161, 21], [15, 264]]
      2 164.0 265.0 13.0 18.0 0.901099 0.953237 0.0468 0.0989 0.8543 0.8584    0.9266 0.9364 0.9136      0.9011      0.9532   0.9326 0.0734 [[164, 18], [13, 265]]
      3 170.0 264.0 14.0 12.0 0.934066 0.949640 0.0504 0.0659 0.8837 0.8820    0.9239 0.9565 0.9290      0.9341      0.9496   0.9435 0.0761 [[170, 12], [14, 264]]
      4 157.0 266.0 13.0 24.0 0.867403 0.953405 0.0466 0.1326 0.8208 0.8297    0.9235 0.9172 0.8946      0.8674      0.9534   0.9196 0.0765 [[157, 24], [13, 266]]
      5 167.0 265.0 14.0 14.0 0.922652 0.949821 0.0502 0.0773 0.8725 0.8725    0.9227 0.9498 0.9227      0.9227      0.9498   0.9391 0.0773 [[167, 14], [14, 265]]
      6 162.0 270.0  9.0 19.0 0.895028 0.967742 0.0323 0.1050 0.8628 0.8712    0.9474 0.9343 0.9205      0.8950      0.9677   0.9391 0.0526  [[162, 19], [9, 270]]
      7 161.0 266.0 13.0 20.0 0.889503 0.953405 0.0466 0.1105 0.8429 0.8487    0.9253 0.9301 0.9070      0.8895      0.9534   0.9283 0.0747 [[161, 20], [13, 266]]
      8 167.0 268.0 11.0 14.0 0.922652 0.960573 0.0394 0.0773 0.8832 0.8858    0.9382 0.9504 0.9304      0.9227      0.9606   0.9457 0.0618 [[167, 14], [11, 268]]
      9 163.0 271.0  8.0 18.0 0.900552 0.971326 0.0287 0.0994 0.8719 0.8804    0.9532 0.9377 0.9261      0.9006      0.9713   0.9435 0.0468  [[163, 18], [8, 271]]
     10 162.0 266.0 13.0 19.0 0.895028 0.953405 0.0466 0.1050 0.8484 0.8534    0.9257 0.9333 0.9101      0.8950      0.9534   0.9304 0.0743 [[162, 19], [13, 266]]
Average 163.4 266.5 12.3 17.9 0.901260 0.955879 0.0441 0.0987 0.8571 0.8618    0.9301 0.9372 0.9153      0.9013      0.9559   0.9344 0.0699                    NaN
```

- GRU Results (**37.7s**):



```
GRU Results:
   Fold    TP    TN   FP   FN      TPR      TNR    FPR    FNR    TSS    HSS Precision    NPV     F1 Sensitivity Specificity Accuracy    FDR       Confusion Matrix
      1 166.0 259.0 20.0 16.0 0.912088 0.928315 0.0717 0.0879 0.8404 0.8372    0.8925 0.9418 0.9022      0.9121      0.9283   0.9219 0.1075 [[166, 16], [20, 259]]
      2 170.0 266.0 12.0 12.0 0.934066 0.956835 0.0432 0.0659 0.8909 0.8909    0.9341 0.9568 0.9341      0.9341      0.9568   0.9478 0.0659 [[170, 12], [12, 266]]
      3 170.0 265.0 13.0 12.0 0.934066 0.953237 0.0468 0.0659 0.8873 0.8865    0.9290 0.9567 0.9315      0.9341      0.9532   0.9457 0.0710 [[170, 12], [13, 265]]
      4 168.0 262.0 17.0 13.0 0.928177 0.939068 0.0609 0.0718 0.8672 0.8639    0.9081 0.9527 0.9180      0.9282      0.9391   0.9348 0.0919 [[168, 13], [17, 262]]
      5 173.0 262.0 17.0  8.0 0.955801 0.939068 0.0609 0.0442 0.8949 0.8871    0.9105 0.9704 0.9326      0.9558      0.9391   0.9457 0.0895  [[173, 8], [17, 262]]
      6 168.0 266.0 13.0 13.0 0.928177 0.953405 0.0466 0.0718 0.8816 0.8816    0.9282 0.9534 0.9282      0.9282      0.9534   0.9435 0.0718 [[168, 13], [13, 266]]
      7 168.0 268.0 11.0 13.0 0.928177 0.960573 0.0394 0.0718 0.8888 0.8905    0.9385 0.9537 0.9333      0.9282      0.9606   0.9478 0.0615 [[168, 13], [11, 268]]
      8 168.0 264.0 15.0 13.0 0.928177 0.946237 0.0538 0.0718 0.8744 0.8727    0.9180 0.9531 0.9231      0.9282      0.9462   0.9391 0.0820 [[168, 13], [15, 264]]
      9 164.0 273.0  6.0 17.0 0.906077 0.978495 0.0215 0.0939 0.8846 0.8941    0.9647 0.9414 0.9345      0.9061      0.9785   0.9500 0.0353  [[164, 17], [6, 273]]
     10 168.0 258.0 21.0 13.0 0.928177 0.924731 0.0753 0.0718 0.8529 0.8463    0.8889 0.9520 0.9081      0.9282      0.9247   0.9261 0.1111 [[168, 13], [21, 258]]
Average 168.3 264.3 14.5 13.0 0.928298 0.947996 0.0520 0.0717 0.8763 0.8751    0.9212 0.9532 0.9246      0.9283      0.9480   0.9402 0.0788                    NaN
```

- The results of the 10-fold cross-validation for each algorithm (Random Forest, SVM, and GRU) are presented in separate tables. Each table includes the following performance metrics for each fold and the average across all folds: True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN), True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR), False Negative Rate (FNR), True Skill Statistic (TSS), and Heidke Skill Score (HSS).

The tables are referred to in the discussion section to compare the performance of the algorithms.

6. Discussion

Comparison

- **Accuracy**: Random Forest has the highest accuracy (0.9548), followed by GRU (0.9402), and then SVM (0.9344).
- **Sensitivity (TPR)**: Random Forest has the highest sensitivity (0.9294), very closely followed by GRU (0.9283), indicating both are effective at correctly identifying spam emails. SVM has the lowest sensitivity (0.9013).
- **Specificity (TNR)**: Random Forest has the highest specificity (0.9713), followed by SVM (0.9559) and GRU (0.9479). This indicates Random Forest is the best at correctly identifying non-spam emails.
- **False Positive Rate (FPR)**: Random Forest has the lowest FPR (0.0287), followed by SVM (0.0441) and GRU (0.0520). This means Random Forest is least likely to incorrectly classify a non-spam email as spam.
- **False Negative Rate (FNR)**: Random Forest and GRU have similar FNR (0.0706 and 0.0717, respectively), which is lower than SVM (0.0987). This means Random Forest and GRU are less likely to incorrectly classify a spam email as non-spam.
- **Precision**: Random Forest has the highest precision (0.9549), followed by GRU (0.9525), and SVM (0.9382)
- **NPV**: Random Forest has the highest NPV (0.9494), followed by SVM (0.9372), and GRU (0.9353)
- **F1 Score**: Random Forest has the highest F1 score (0.9418), followed by GRU (0.9246), and SVM (0.9153). This indicates that Random Forest has the best balance of precision and recall.
- **TSS and HSS**: Random Forest has the highest TSS (0.9007) and HSS (0.9049), indicating the best overall performance and skill relative to chance. GRU is in the middle (TSS: 0.8763, HSS: 0.8751), and SVM has the lowest scores (TSS: 0.8571, HSS: 0.8618).
- **Computational Time**: SVM is the fastest (3 seconds), followed by Random Forest (6.9 seconds), and GRU is the slowest (37.7 seconds).

Random Forest

- **Strengths:**
    - **High Performance:** Achieved the best overall performance in terms of accuracy, TPR, TNR, TSS, and HSS. It is very effective at both correctly identifying spam and non-spam emails.
    - **Robustness to Overfitting:** Ensemble of decision trees reduces the risk of overfitting.
    - **Feature Importance:** Provides a measure of feature importance, which can help in understanding the data.
    - **Handles Non-linearities:** Can capture complex non-linear relationships between features.
- **Weaknesses:**
    - **Computational Time:** Slower than SVM (6 seconds) in this experiment.
    - Less interpretable than a single decision tree.

SVM

- **Strengths:**
    - **Computational Efficiency:** Fastest evaluation time (3 seconds).
    - **Effective in High-Dimensional Spaces:** Performs well with a large number of features.
- **Weaknesses:**
    - **Lower Performance:** Lowest performance among the three algorithms in terms of all metrics (accuracy, TPR, TNR, TSS, HSS). It is more prone to misclassifying both spam and non-spam emails compared to Random Forest and GRU.
    - **Sensitive to Feature Scaling:** Requires careful feature scaling for optimal performance.
    - **Less Interpretable:** The RBF kernel makes the model less interpretable.

GRU

- **Strengths:**
    - **Good Performance:** Performance is very close to Random Forest.
    - **Handles Complex Relationships:** Can learn complex relationships and patterns in the data.
- **Weaknesses:**
    - **Computational Cost:** Significantly slower than both SVM and Random

    Forest (37 seconds).
    - **Less Interpretable:** Deep learning models are generally less interpretable than tree-based models like Random Forest or linear models like SVM.
    - **Potential for Overfitting:** Requires careful tuning and regularization to prevent overfitting, especially with smaller datasets.

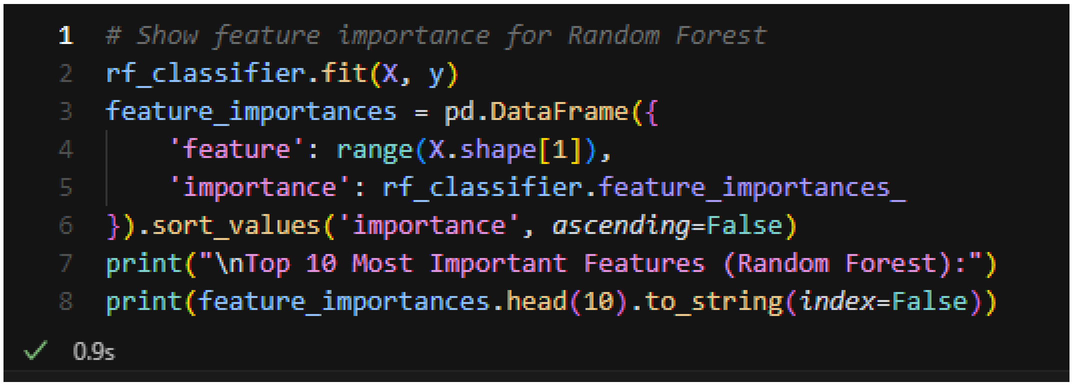
- During set up of the experiment, I encountered the following issues:
    - Installing tensorflow for python to utilize GRU. Instead, I opted to use PyTorch for ease-of-use.
    - Creating the evaluation for GRU which was more complex than Random Forest and SVM
- To improve this project, I would look to optimize parameters. Additionally, better datasets may provide better performance for the models in unforeseen ways. A more complex dataset may allow GRU to outperform Random Forest.

7. Conclusion

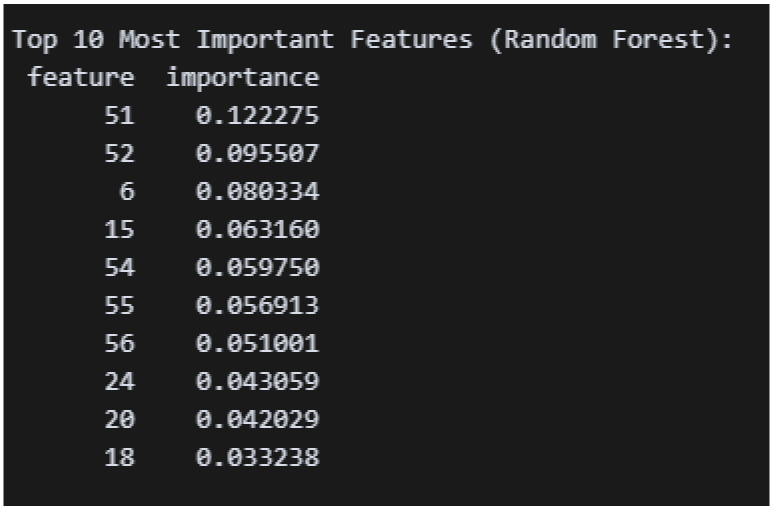
Random Forest demonstrates the best overall performance in terms of accuracy and all other classification metrics. GRU performs comparably to Random Forest in many metrics, but it is significantly slower. SVM is the fastest but has the lowest overall performance. Based on this task, I would recommend Random Forest for accuracy. SVM might be applicable if speed is necessary, assuming very large datasets. GRU is best for sequential data, so this dataset doesn't take full advantage of its capabilities. It might be best when the dataset incorporates time-series features.

8. Appendix

- I also experimented with feature importance for the random forest classifier:



```python
# Show feature importance for Random Forest
rf_classifier.fit(X, y)
feature_importances = pd.DataFrame({
    'feature': range(X.shape[1]),
    'importance': rf_classifier.feature_importances_
}).sort_values('importance', ascending=False)
print("\nTop 10 Most Important Features (Random Forest):")
print(feature_importances.head(10).to_string(index=False))
```



```
Top 10 Most Important Features (Random Forest):
 feature  importance
      51    0.122275
      52    0.095507
       6    0.080334
      15    0.063160
      54    0.059750
      55    0.056913
      56    0.051001
      24    0.043059
      20    0.042029
      18    0.033238
```