**Code Report:**

**Computer Security Class**

**Libraries & Tools Used:**

* Data Manipulation: pandas, numpy
* Visualization: matplotlib, seaborn
* Text Preprocessing: nltk (lemmatization, stemming, stopword removal)
* Feature Extraction: TfidfVectorizer
* Machine Learning:
  + Classifiers: MultinomialNB, SVC, RandomForestClassifier, XGBClassifier, LogisticRegression, KNeighborsClassifier, DecisionTreeClassifier, AdaBoostClassifier, GradientBoostingClassifier
  + Utilities: train\_test\_split, LabelEncoder, cross\_val\_score
* Evaluation Metrics: accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

**Dataset Used:**

The Spam-ham dataset provided from Kaggle was used for this project (spam.csv), <https://www.kaggle.com/datasets/tmehul/spamcsv?resource=download>.

**Data Preprocessing:**

* Dataset was loaded using pandas
* We used various ways to clean up the text, like
  + Removing non-alphabetical characters.
  + Converting to lowercase.
  + Removing stopwords.
  + Stemming and Lemmatizing tokens.
* Dataset was shrunk, the remove any parts that were only ham. Like, removing all data that was over 350 characters, over 50 words, and over 10 sentences because all of them were ham.

**Models Trained & Classifiers used:**

MultinomialNB: Simple and fast but struggles with non-linear relationships.

LogisticRegression: Average performance but limited to complex patterns.

SVC & Linear SVC: Good for smaller datasets, but long loading times because its computationally expensive.

RandomForest: Good for handling overfitting, but it works slower on big datasets.

XGBClassifier: Gives high accuracy but takes longer to train.

DecisionTree: Easy to understand and interpret but runs into issues like overfitting a lot.

KNeighbours: Simple but sensitive to irrelevant features.

Adaboost & GradientBoosting: Good but usually requires more tuning for higher accuracy.

**Models were evaluated using:**

* Accuracy
* Precision
* Recall
* F1 Score
* Confusion Matrix
* Cross-validation Scores

**Results (Before using Hyperparameters & Cross-Validation):**

**Accuracy Results:**

NaiveBayes: 0.967296

RandomForest: 0.978959

KNeighbours: 0.910130

SVC: 0.976903

LinearSVC: 0.984677

LogisticRegression: 0.957693

GradientBoosting: 0.969353

AdaBoost: 0.926594

DecisionTree: 0.961578

XGBoost: 0.976902

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Model | Precision | Recall | F1 Score | Test Accuracy | Train Accuracy |
| Naive Bayes | 1.000000 | 0.787671 | 0.881226 | 0.977148 | 0.992911 |
| Random Forest | 1.000000 | 0.835616 | 0.910448 | 0.977148 | 0.992911 |
| K-Nearest Neighbors | 1.000000 | 0.390411 | 0.561576 | 0.977148 | 0.992911 |
| SVC | 0.992063 | 0.856164 | 0.919118 | 0.977148 | 0.992911 |
| Linear SVC | 0.984615 | 0.876712 | 0.927536 | 0.977148 | 0.992911 |
| Logistic Regression | 0.982456 | 0.767123 | 0.861538 | 0.977148 | 0.992911 |
| Gradient Boosting | 0.974790 | 0.794521 | 0.875472 | 0.977148 | 0.992911 |
| AdaBoost | 0.864583 | 0.568493 | 0.685950 | 0.977148 | 0.992911 |
| Decision Tree | 0.866197 | 0.842466 | 0.854167 | 0.977148 | 0.992911 |
| XGBoost | 0.961832 | 0.863014 | 0.909747 | 0.977148 | 0.992911 |

**Results (After using Hyperparameters & Cross-Validation):**

**Accuracy Results:**

NaiveBayes: 0.980558

RandomForest: 0.980103

KNeighbours: 0.942830

SVC: 0.983304

LinearSVC: 0.985134

LogisticRegression: 0.978273

GradientBoosting: 0.973926

AdaBoost: 0.958377

DecisionTree: 0.959520

XGBoost: 0.974160

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Model | Precision | Recall | F1 Score | Test Accuracy | Train Accuracy |
| Naive Bayes | 0.930070 | 0.910959 | 0.920415 | 0.978976 | 0.995198 |
| Random Forest | 1.000000 | 0.856164 | 0.922509 | 0.980804 | 0.998628 |
| K-Nearest Neighbors | 1.000000 | 0.589041 | 0.741379 | 0.945155 | 1.000000 |
| SVC | 0.984615 | 0.876712 | 0.927536 | 0.981718 | 0.994054 |
| Linear SVC | 0.977099 | 0.876712 | 0.924188 | 0.980804 | 1.000000 |
| Logistic Regression | 0.984496 | 0.869863 | 0.923636 | 0.980804 | 0.998628 |
| Gradient Boosting | 0.947480 | 0.815068 | 0.884758 | 0.971664 | 0.997942 |
| AdaBoost | 0.974576 | 0.787671 | 0.871212 | 0.968921 | 0.963641 |
| Decision Tree | 0.879433 | 0.869315 | 0.864111 | 0.964351 | 0.991639 |
| XGBoost | 0.968750 | 0.849315 | 0.905109 | 0.976234 | 0.983535 |

**Findings:**

Using hyperparameters for the most part improved the Recall, F1 score & reduced overfitting, but it took considerably more time to run. More than double the amount of time it takes to run the whole code without adding hyperparameters. General accuracy improved for all classifiers by 1-3% as well.

There were more significant changes in classifiers like:

* K-Nearest Neighbors (huge recall improvement)
* AdaBoost (massive recall and F1 score improvement)
* Random Forest (higher recall with perfect precision)

Best Performing Classifiers:

SVC, Linear SVC & Random Forest, they had some of the highest general accuracy, and a high f1 score as well as test accuracy.

Recommended Classifiers:

Despite not having the highest performance of all the classifiers, classifiers like Logistic Regression, XGBoost, Naive Bayes are recommended. Logistic Regression has a good balance of precision, recall and generalization. XGBoost generally does well on most datasets on my experience, showing in it’s high F1 score and stable accuracy. Naive Bayles (MultinomialNB) is a simpler model and generally runs faster than a lot of the other classifiers, it’s good classifier to have to balance out all the more complex classifiers included.

Best Model Overall: LinearSVC had the highest F1 Score of 0.9275, showing the best balance between precision and recall. It consistently performed well across all categories, making it a reliable classifier.

Most Precise Models: Random Forest & K-Nearest Neighbors (KNN) had a perfect precision score of 1.0, meaning they made no false positive predictions. This doesn’t mean they were the best models though, as K-Nearest Neighbours especially struggled with recall, failing to correctly identify many spam messages.

Worst Performing: Even though it had perfect precision, K-Nearest Neighbours (KNN) had the worst recall at just 0.39 and only improving to 0.58 with tuning. This means it missed many spam messages, meaning it was unreliable for finding actual spam messages even with our attempted improvements.