CS6316 Application Project Report

Project Members: Ajwa Shahid (ckf2dq), Ji Hyun Kim (mqa4qu), Nicki Choquette (nc2uzv)

# Data Processing

For the project3\_dataset1, and project3\_dataset2 we separated the features from the labels. For the dataset2 we mapped the fourth column from the features such that the “Absent” is 0 and “Present” is 1. Then we normalized both the datasets to avoid any scaling issue.The following algorithms were implemented for both of these datasets.

# 1. Logistic Regression

Logistic regression is similar to linear regression except it does categorization instead of a continuous prediction. This categorization is usually a binary classification (either this or that) and it utilizes different features in the dataset to make that classification.It is also able to decide which features are useful by what amount, to make a fitting classification.This is decided using weights or coefficients and the bias value. These values are updated through the process of training. When the training is complete, the model predicts, using the assigned weights and bias. the class to which a data point may belong to.

## 1.1 Parameter Testing and Performance Comparison

We learnt in class about the regularization coefficient, which means how closely can the model fit the training data. By making the regularization coefficient to be smaller, we can avoid overfitting. Strong regularization effect will give less accurate results since the model would then overfit on the training data and perform poorly on the tests data. So, for our model we decided to test out different regularization values by changing the C parameter (the inverse of regularization strength). A higher value of C would mean minimal regularization, and vice versa. The range of values we tested are listed below:

**C (Inverse of Regularization Strength)**

[ '0.001', '0.01', '0.1', '1', '10', '100']

Dataset 1

Values of C tested as hyperparameter: '0.001', '0.01', '0.1', '1', '10', '100'

****

**Bias/Variance Tradeoff**

As figure 1 shows, a low value of C, high regularization gives low training accuracy, which means that the model has not fitted itself closely to the training data, this is low variance and high bias. As we increase C, and regularization becomes low, the model better fits the training data giving high training accuracy and also improves the testing accuracy. But when we increase C beyond the value of 1 (i.e. 10, 100) the training accuracy increases since the model has to overfit on the training data and testing accuracy decreases as a result, this causes high variance and low bias. Considering this tradeoff, we ended up choosing C=1 for our model for the best results.

Dataset 2

Values of C tested as hyperparameter: '0.001', '0.01', '0.1', '1', '10', '100'



**Bias/Variance Tradeoff**

Similarly for the second dataset, a low value of C, high regularization gives low training accuracy, which means that the model has not fitted itself closely to the training data, this is low variance and high bias. Shown in figure 2, we increase C, and regularization becomes low, the model better fits the training data giving high training accuracy and also improves the testing accuracy. But when we increase C beyond the value of 0.1 (i.e. 1, 10, 100) the training accuracy decreases this behavior was unexpected, but since this resulted in low accuracy for both test and train data, the best choice was obvious. Considering this tradeoff, we ended up choosing C=0.1 for our model for the best results.

## 1.2 Evaluating Results and Bias-Variance Analysis

Dataset 1

Highest Mean Test Accuracy Achieved: 0.9824561403508772

Best C Value: 1

| Metric | Mean Train Score | Mean Test Score |
| --- | --- | --- |
| Accuracy | 0.988284 | 0.982456 |
| Precision | 0.992550 | 0.981366 |
| Recall | 0.975889 | 0.971429 |
| F1 | 0.984142 | 0.976175 |
| AUC | 0.997519 | 0.994976 |

**Table 1**



The cross validation results for dataset 1 is shown in Figure 3 and Table 1. The difference between the training and testing scores across all metrics is insignificant, indicating a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.988) is slightly higher than the testing accuracy (0.982), however the gap is not significant, indicating that it did not overfit.
* **Precision**: both training (0.992) and testing (0.981) scores are high, where the model has a higher rate of true positives than false positives.
* **Recall**: the gap between training (0.975) and testing (0.971) is very small, indicating that the model has generalized well.
* **F1 Score**: F1 is influenced both by precision and recall, thus the gap of F1 score is presumably from precision scores. This again indicates the model has generalized well.
* **AUC**: both training (0.997) and testing (0.995) scores are high, indicating the model classifies the dataset well.

Dataset 2

Highest Mean Test Accuracy Achieved: 0.7251156336725254

Best C Value: 0.1

| Metric | Mean Train Score | Mean Test Score |
| --- | --- | --- |
| Accuracy | 0.745310 | 0.725116 |
| Precision | 0.671571 | 0.647543 |
| Recall | 0.518056 | 0.493750 |
| F1 | 0.584824 | 0.547595 |
| AUC | 0.795515 | 0.777298 |

**Table 2**



The cross validation results for dataset 2 is shown in Figure 4 and Table 2. The difference between the training and testing scores across all metrics is relatively low, indicating a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.745) is slightly higher than the testing accuracy (0.725), however the gap is not significant, indicating that it did not overfit.
* **Precision**: both training (0.672) and testing (0.648) scores are arguably good, where the model has a higher rate of true positives than false positives.
* **Recall**: the gap between training (0.518) and testing (0.493) is small, but the values are low indicating that the model has a high rate of false negatives.
* **F1 Score**: F1 is influenced both by precision and recall, thus the low value of F1 score is presumably from recall scores. This indicates a high rate of false positives and false negatives. This may be caused by class imbalance in the dataset.
* **AUC**: both training (0.796) and testing (0.777) scores are high, indicating the model classifies the dataset well.

# 2. K Nearest Neighbor

K Nearest Neighbors (KNN) is a simple model to classify data. It trains on an already classified data, and when given a datapoint it decides the class by looking at the nearest K number of neighbors, and assigns the datapoint the majority class.

## 2.1 Parameter Testing and Performance Comparison

**Number of Neighbours (K)**

A low value of K leads to a lot of noise, high variance, and low bias. This also leads to overfitting on the training data. Increasing the K helps to generalize the model better, reducing overfitting, variance and increasing bias. But very high values of K may lead to underfitting and become less sensitive, causing low variance and a high bias. To test what value of K would give us the best result, we tested our model with multiple values of K and selected based off of the best mean test accuracy.

Dataset 1

Number of Neighbours (K) tested as hyperparameters ranged from 1 to 99.



**Bias/Variance Tradeoff**

With initial low values of K the model overfit on the training data, giving high mean train accuracy and low test accuracy, as seen in the Figure 5. As we increase K to around 10, the test accuracy peaks and becomes comparable to the train accuracy, showing that the model is able to classify well. But as we keep increasing K, the accuracy for both test and train data decreases because the classification now is less influenced by local variations in the data.

Dataset 2

Number of Neighbours (K) tested as hyperparameters ranged from 1 to 99.



**Bias/Variance Tradeoff**

Similarly for the second dataset, as shown in the Figure 6, with initial low values of K the model overfit on the training data, giving high mean train accuracy and low test accuracy. As we increase K to around 25, the test accuracy peaks and becomes comparable to the train accuracy, showing that the model is able to classify well. But as we keep increasing K, the accuracy for both test and train data decreases because the classification now is less influenced by local variations in the data.

## 

## 2.2 Cross Validation Result Evaluation

Dataset 1

Highest Mean Test Accuracy Achieved: 0.9718984962406015

Best K Value: 11

| Metric | Mean Train Score | Mean Test Score |
| --- | --- | --- |
| Accuracy | 0.972662 | 0.971898 |
| Precision | 0.989499 | 0.990909 |
| Recall | 0.936580 | 0.933766 |
| F1 | 0.962295 | 0.960521 |
| AUC | 0.996117 | 0.993058 |

**Table 3**



The cross validation results for dataset 1 is shown in Table 3 and Figure 7. The difference between the training and testing scores across all metrics is insignificant, indicating a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.973) is very slightly higher than the testing accuracy (0.972), however the gap is not significant, indicating that it did not overfit.
* **Precision**: both training (0.989) and testing (0.990) scores are high, where the model has a higher rate of true positives than false positives.
* **Recall**: the gap between training (0.936) and testing (0.933) is very small, indicating that the model has generalized well.
* **F1 Score**: F1 is influenced both by precision and recall, this again indicates the model has generalized well.
* **AUC**: both training (0.996) and testing (0.993) scores are high, indicating the model classifies the dataset well.

Dataset 2

Number of Neighbours (K) tested as hyperparameters ranged from 1 to 99.

Highest Mean Test Accuracy Achieved: 0.727335800185014

Best K Value: 25

| Metric | Mean Train Score | Mean Test Score |
| --- | --- | --- |
| Accuracy | 0.747719 | 0.727336 |
| Precision | 0.738649 | 0.696883 |
| Recall | 0.421528 | 0.393750 |
| F1 | 0.535752 | 0.487000 |
| AUC | 0.790391 | 0.753958 |

**Table 4**



The cross validation results for dataset 2 is shown in Figure 4 and Table 8. The difference between the training and testing scores across all metrics is relatively low, indicating a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.748) is slightly higher than the testing accuracy (0.727), however the gap is not significant, indicating that it did not overfit.
* **Precision**: both training (0.739) and testing (0.697) scores are arguably high, where the model has a higher rate of true positives than false positives.
* **Recall**: the gap between training (0.421) and testing (0.392) is small, but both the values are low indicating that the model has a high rate of false negatives
* **F1 Score**: F1 is influenced both by precision and recall, thus the low value of F1 score is presumably from recall scores. This indicates a high rate of false positives and false negatives. This may be caused by class imbalance in the dataset.
* **AUC**: both training (0.790) and testing (0.754) scores are high, indicating the model classifies the dataset well but because the area is slightly higher on the training set, it may be overfit.

# 

# 3. Decision Tree



Decision Tree Classification is a supervised algorithm that is built top down, splitting the data based on attributes that optimizes certain criteria.

As figure 9 shows, it starts from the root node with all data points, goes down as it repeatedly looks at all current leaves and all possible splits to choose the split that most reduces uncertainty in prediction until it reaches the last leaf node. The classifier greedily chooses the best decision rules, which means it does not consider future implications, leading to resulting suboptimal trees and it is easy to overfit. However it is considered as a simple and effective classifier since it is intuitive, easy to interpret, and visualize them.

## 3.1 Parameter Testing

### 3.1.1 **Comparing Parameters**

In the decision tree classifier, it is important to determine how to split the data and when to stop splitting the data to find the best splits and results. To get the best results based on this determination, we considered the parameters below.

#### 

| *Gini vs Entropy* There are two parameters to quantify how to split the data: Gini and Entropy. Both methods quantify how impure each classified node is and tries to increase the homogeneity of the nodes (Figure 10), which indicates it is well classified. Gini is the default parameter for decision tree classification, and is faster to compute as it does not use log function as Entropy does. However, entropy can lead to more balanced trees. |  |
| --- | --- |

#### *max\_depth & min\_samples\_leaf*

In terms of determining when to stop splitting the data, we used max\_depth and min\_samples\_leaf parameters so that it can directly impact the bias-variance tradeoff. Finding the optimal value of the maximum depth of the decision tree, we aimed to prevent the trees from growing and eventually overfitting the dataset by perfectly classifying all training dataset. Also to prevent it from being overly complex. Similarly, by finding the optimal value of a minimum number of samples required on a leaf node, we can avoid the model having leaf nodes that represent only for a specific instance, which is again, overfitting.

### 3.1.2 **Finding** Optimized **Parameter**s

We chose Entropy instead of Gini for our parameters to get a more balanced dataset, which was also confirmed that they produced better scores before adding other parameters on the initial testing.

|  | Entropy | Gini |
| --- | --- | --- |
| Test accuracy for Dataset 1 | **0.934962** | 0.915664 |
| Test accuracy for Dataset 2 | **0.638575** | 0.616790 |

**Table 5**

To find the optimized value of the parameters for both max\_depth and min\_samples\_leaf in combination, we looped both **ranges from 1 to 20**. The results were as follows.

Dataset 1 Best Model - **max\_depth: 4**, **min\_samples\_leaf: 10**, testing accuracy: 0.942

Dataset 2 Best Model - **max\_depth: 3**, **min\_samples\_leaf: 10**, testing accuracy: 0.723

Both max\_depth was lower among the range of 1 to 20, yet had the highest testing accuracy among the others. We can assume that if the max\_depth is over 4 for dataset 1, and 3 for dataset 2, the model could overfit due to high variance. As the numbers are low, it indicates that the model only needs a few layers to adequately classify the dataset.

Based on the results, we ran a cross validation decision tree classification to see the final results.

## 3.2 Cross Validation Result Evaluation

### 3.2.1 Dataset 1

|  |
| --- |
| | Metric | Mean Train Score | Mean Test Score | | --- | --- | --- | | Accuracy | 0.964070 | 0.941980 | | Precision | 0.967200 | 0.942512 | | Recall | 0.936051 | 0.901299 | | F1 | 0.950883 | 0.920760 | | AUC | 0.995152 | 0.963000 |   **Table 6: Decision Tree Cross Validation Result - Dataset 1** |

The cross validation results for dataset 1 is shown on Figure 11 and Table 6. The difference between the training and testing scores across all metrics is relatively small, indicating a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.964) is higher than the testing accuracy (0.942), however the gap is not significant, indicating that it did not overfit.
* **Precision**: both training (0.967) and testing (0.943) scores are high, where the model has a higher rate of true positives than false positives.
* **Recall**: the gap between training (0.936) and testing (0.901) score is relatively higher than the other scores, where the model is missing more relevant cases in the test dataset. This might imply a slightly higher bias towards the training data.
* **F1 Score**: F1 is influenced both by precision and recall, thus the gap of F1 score is presumably from recall scores. Still, this might indicate the model is beginning to overfit.
* **AUC**: both training (0.995) and testing (0.963) scores are high, indicating the model classifies the dataset well.

### **3.2.2** Dataset 2

|  |
| --- |
| | Metric | Mean Train Score | Mean Test Score | | --- | --- | --- | | Accuracy | 0.757331 | 0.723034 | | Precision | 0.706577 | 0.648194 | | Recall | 0.517361 | 0.468750 | | F1 | 0.594145 | 0.530690 | | AUC | 0.789992 | 0.710870 |   **Table 7: Decision Tree Cross Validation Result - Dataset 2** |

The cross validation results for dataset 2 is shown on Figure 12 and Table 7. The overall score is relatively lower than the first dataset and the difference between the training and testing scores across all metrics is larger than the first dataset. This might indicate that there could be some room for improvements in the bias-variance tradeoff, however, there is not a significant amount of gap to consider that the model has overfitted.

For each score:

* **Accuracy**: The training accuracy (0.757) is higher than the testing accuracy (0.723). However, the gap is not substantially significant.
* **Precision**: Both the scores indicate that the model is good at predicting true positives. However the training score (0.707) is somewhat higher than the testing precision (0.648), indicating that it is slightly better tuned to the training data.
* **Recall**: The gap between the training (0.517) and testing (0.469) recall score is noticeable. The model is missing relevant cases in the test data suggesting it could be slightly overfitting to the training data.
* **F1 Score**: Presumably due to the recall gap, the model’s balance between precision and recall is better on the training than the test data.
* **AUC**: Both the training score (0.790) and the testing score (0.711) indicates that the model will perform well on differentiating classes, however the gap is higher in the training score.

# 4. SVM

## Support Vector Machines are classification models that can classify data with no obvious linear separation by transforming them to a higher dimension and using a support vector classifier in that relatively higher dimension. To decide which dimension would be best to transform the feature space to, we use different Kernels. Other parameters like gamma and C also affect the classification and have been discussed in more detail in the following sections. [4]

## 4.1 Parameter Testing and Performance Comparison

Below are presented the set of parameters used to test both the first and second dataset. The hyperparameters we chose to change and test with were the Kernel, the value of C, and gamma.

**Kernel** [3]

1. Linear

This works well when the data is linearly separable, resulting in an untransformed feature space.

1. Polynomial

This works well for relationships that are non-linear and can make more flexible decision boundaries, the data is transformed but no extra dimension is added.

1. RBF

These are even more flexible than polynomial kernels and the decision boundary complexity depends on the value of gamma, which we are also changing and testing with for this project. And so we hypothesized that this choice will probably give the best test accuracy. However, higher gamma values could lead to overfitting so we tested with a range of values. This is discussed more later in this section.This is also the default kernel in sci-kit learn.

1. Sigmoid

The sigmoid function is also nonlinear and the shape and flexibility of the decision boundary for this also depends on the gamma value. This also seemed like a good candidate to perform our tests due to the flexibility, but the complex boundaries that form may not be able to generalize well over the dataset.

**Parameter C**

[0.1, 1, 10]

Large value of parameter C gives a small margin, hence penalizing more on misclassified datapoints. This results in Low Bias and High Variance A small value of parameter C gives a large margin, this allows for more misclassifications, resulting in High Bias and Low Variance.



**Gamma (𝛾)**

[0.01, 0.1, 1, 10]

We learnt in class that when 𝛾 is small, a support vector has influence on deciding the class of a data even if the distance is large.This results in High Bias and Low Variance. When 𝛾 is large, a support vector does not have widespread influence. This results in Low Bias and High Variance.



We test with these parameters, value ranges for each are shown below for both datasets.

Dataset 1 and Dataset 2



## 4.2 Cross-Validation Result Evaluation and Bias-Variance Analysis

Dataset 1

Best Hyperparameters: {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}

| Metric | Mean Train Score | Mean Test Score |
| --- | --- | --- |
| Accuracy | 0.987893 | 0.982456 |
| Precision | 0.998387 | 0.995455 |
| Recall | 0.969080 | 0.957359 |
| F1 | 0.983508 | 0.975471 |
| AUC | 0.998087 | 0.996429 |

**Table 8**



The cross validation results for dataset 1 is shown in Figure 16 and Table 8. The difference between the training and testing scores across all metrics is insignificant, indicating a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.987) is slightly higher than the testing accuracy (0.982), however the gap is not significant, indicating that it did not overfit.
* **Precision**: both training (0.998) and testing (0.995) scores are high, where the model has a higher rate of true positives than false positives.
* **Recall**: the gap between training (0.969) and testing (0.957) is very small, indicating that the model classifies the data well.
* **F1 Score**: F1 is influenced both by precision and recall, thus the gap of F1 score is presumably from recall scores. This again indicates the model has generalized well.
* **AUC**: both training (0.998) and testing (0.996) scores are high, indicating the model classifies the dataset well.

Dataset 2

Best Hyperparameters: {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}

| Metric | Mean Train Score | Mean Test Score |
| --- | --- | --- |
| Accuracy | 0.772008 | 0.738067 |
| Precision | 0.729841 | 0.684401 |
| Recall | 0.543056 | 0.481250 |
| F1 | 0.622361 | 0.549867 |
| AUC | 0.816257 | 0.778898 |

**Table 9**



The cross validation results for dataset 2 is shown in Figure 17 and Table 9. The difference between the training and testing scores across all metrics is relatively low, indicating a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.772) is slightly higher than the testing accuracy (0.738), and this difference might indicate that the model might have overfit on the training data..
* **Precision**: both training (0.730) and testing (0.684) scores are comparable and relatively low, showing the model has a moderately good rate of true positives than false positives.
* **Recall**: the gap between training (0.543) and testing (0.481) is small, but both values are low showing that the model has a high rate of false negatives.
* **F1 Score**: F1 is influenced both by precision and recall, thus this indicates that the model may have overfit on the training data.
* **AUC**: both training (0.816) and testing (0.778) scores are good and comparable, indicating the model classifies the dataset relatively well.

# 5. Random Forest



As shown in figure 18 and also by its name, Random Forest is an algorithm that combines multiple decision trees to get the final results, which is an ensemble method. The algorithm uses bagging, which is building each tree from a random sample of the dataset, picks a random subset of features for each tree, and takes majority voting from all trees to get to the final result. This helps each tree to be less correlated to each other. Due to this, the random forest algorithm does not consider all possible features in the dataset as the decision tree did. Therefore, it is better in reducing overfitting and increasing precision. Also, the majority voting helps the model to be less prone to overfitting.

## 5.1 Parameter Testing

### 5.1.1 Comparing Parameters

Random forest algorithm works best with some hyper-parameter tuning so that it yields better accuracy than decision trees, effectively handles missing data, and so on. Among the several hyperparameters, we chose n\_estimator and max\_depth to get better results on the cross validation score.

#### n\_estimator

This parameter determines the number of trees in the forest, which can directly impact the model’s accuracy score and complexity. More trees can lead to better accuracy, but at the same time require computational cost.

#### max\_depth

As mentioned in the decision tree, max\_depth is used to set the depth of the trees. It is important to find the best value for this parameter to prevent the model being overly complex and overfitting.

### 5.1.2 Finding Optimized Parameters

#### N\_estimator ONLY



#### 

To find the best parameter values, we first evaluated the model’s accuracy with varying n\_estimators by itself in the range of 1 to 200. Both datasets exhibited a significant increase in accuracy initially as the number of trees (n\_estimators) increased, which however, stopped at a certain point. The best n\_estimator for **dataset 1 was at 51 (accuracy: 0.968)**, and for **dataset 2 was at 64 (accuracy: 0.712)**.

Considering the bias-variance tradeoff, the initial stage with very few trees could imply that there is a high bias due to excessive simplicity. The model is not complex enough to classify the datasets correctly, therefore the accuracy is low. As the number of trees increases, the bias generally decreases, making better predictions that increases the testing accuracy. However, after a certain point, increasing the number of trees does not continue to decrease bias or affect variance. The bias has already been minimized as much as it can and the trees are sufficiently randomized, which does not affect the variance.

#### 

#### Combination of n\_estimator and max\_depth

To find the optimized value of the parameters for both the n\_estimator and max\_depth in combination, we looped them in ranges of 10 to 100, and 1 to 30, respectively. The results were as follows:  
Dataset 1 Best Model - **max\_depth: 7, n\_estimator: 17**, testing accuracy: 0.970

Dataset 2 Best Model - **max\_depth: 3, n\_estimator: 38**, testing accuracy: 0.725

Unlike when n\_estimator was solely evaluated, the optimal number of trees changed when it was tuned with multiple hyperparameters together. By adding the variations of max\_depth parameter, we can assume that the change in restricting the max\_depth prevented overfitting, which led to better accuracy with less number of trees.

Based on the results as below, we ran a cross validation random forest classification to see the final results.

## 5.2 Cross Validation Result Evaluation

### 5.2.1 Dataset 1

|  |
| --- |
| | Metric | Mean Train Score | Mean Test Score | | --- | --- | --- | | Accuracy | 0.999414 | 0.970144 | | Precision | 1.000000 | 0.971896 | | Recall | 0.998427 | 0.948268 | | F1 | 0.999211 | 0.959030 | | AUC | 0.999998 | 0.990094 |   **Table 10: Random Forest Cross Validation Result - Dataset 1** |

The cross validation results for dataset 1 is shown on Figure 21 and Table 10. The gap between the training and testing scores are minimized in all metrics, showing that the model has achieved a well-balanced bias-variance tradeoff.

For each score:

* **Accuracy**: training accuracy (0.999) slightly exceeds testing accuracy (0.970), suggesting that the model was well trained with minimal overfitting.
* **Precision**: the training score is perfect (1.000) and also a hgh testing precision (0.971) shows the model is great at identifying true positives.
* **Recall**: there is a difference between training (0.998) and testing (0.948), which is not significant to say that it is overfitting, however indicates that the model missed some relevant cases.
* **F1 Score**: both the training (0.999) and testing (0.959) scores are high showing a good balance between precision and recall.
* **AUC**: training (1.000) was near perfect where testing was also very high (0.990), demonstrating that the model classifies the dataset very well.

### 5.2.2 Dataset 2

|  |
| --- |
| | Metric | Mean Train Score | Mean Test Score | | --- | --- | --- | | Accuracy | 0.778016 | 0.724977 | | Precision | 0.793709 | 0.682411 | | Recall | 0.486806 | 0.393750 | | F1 | 0.602483 | 0.488774 | | AUC | 0.853746 | 0.752970 |   **Table 11: Random Forest Cross Validation Result - Dataset 2** |

The cross validation results for dataset 1 is shown on Figure 22 and Table 11. The overall accuracy and score is relatively lower than the first dataset, and has more notable differences between the training and the testing scores on all metrics, showing signs of possible overfitting.

For each score:

* **Accuracy**: training accuracy (0.778) is slightly higher than the testing accuracy (0.725)
* **Precision**: model's precision drops from training (0.794) to testing (0.682), showing a decline in the rate of true positives in unseen data.
* **Recall**: gap between training (0.487) and testing (0.394) recall scores suggest that the model is missing relevant cases in the test data
* **F1 Score**: Since both precision and recall has dropped in testing scores, F1 scores also reflects the drop (0.602 to 0.489)
* **AUC**: while training (0.854) relatively hight, the testing (0.753) shows it was less effectively classifying testing data

# 6. AdaBoost



AdaBoost, short for Adaptive Boosting, is also an ensemble method algorithm that combines multiple weak learners to form a strong learner. It assigns weights to each training instance, where misclassified ones are given more weight in the next iterations so that the model can focus more on difficult cases and improve where past models were lacking.

While there are several types of boosting, we chose to use AdaBoost instead of Gradient Boosting or XGBoosting. The biggest difference of AdaBoost from the other two is that AdaBoost adjusts weights on the subsequent iterations, whereas XGBoost and Gradient Boosting optimizes a loss function. Since we wanted to keep the model simple and AdaBoost is less prone to overfitting compared to the other models, we chose to use AdaBoost.

## 6.1 Parameter Testing

### 6.1.1 Parameters

#### n\_estimators

Similar to the n\_estimator parameters used in the previous models, this in AdaBoost refers to the maximum number of weak learners that will be created in the model. It is important to find the optimal n\_estimators since it can affect the model complexity and overfitting. Too large a number of estimators can lead to the model’s capability to capture complex patterns, however, this might only be possible on the training set, but generalized on the testing dataset.

### 6.1.2 Finding Optimized Parameters





To find the best parameter value, we evaluated the model’s accuracy with varying n\_estimators in the range of 1 to 200. The results of the two dataset was different as clearly shown in figures 24 and 25.

For Dataset 1, the graph shows a clear upwards trend as the number of estimators increase, which improved significantly in the initial stages, which stabilizes after a certain point. The optimal number of n\_estimators for Dataset 1 is 162, indicating that increasing the number of estimators beyond this number of estimators does not lead to significant improvements in accuracy. In terms of bias-variance trade-off, a higher number of estimators help reduce bias without increasing the variance, showing a higher accuracy.

On the other hand, Dataset 2 shows a different pattern. The mean test accuracy initially starts on a higher point, still tends to increase with the number of estimators, however begins to decline very soon. The optimal number of n\_estimators for Dataset 2 is 20, where the performance of the model starts to decrease after that. Therefore, n\_estimaters beyond 20 indicates high variance, meaning the model starts to overfit.

The differences of these results can be assumed due to the patterns of the datasets such as dataset 2 being overly simple that were captured with only a few weak learners, and started to overfit after that.

## 6.2 Cross Validation Result Evaluation

### 6.2.1 Dataset 1

|  |
| --- |
| | Metric | Mean Train Score | Mean Test Score | | --- | --- | --- | | Accuracy | 1.0 | 0.977162 | | Precision | 1.0 | 0.985429 | | Recall | 1.0 | 0.952597 | | F1 | 1.0 | 0.968392 | | AUC | 1.0 | 0.993257 |   **Table 12: AdaBoost Cross Validation Result - Dataset 1** |

The cross validation results for dataset 1 is shown on Figure 26 and Table 12. All the training scores reached a perfect score, which was concerning whether it is overfitting the training dataset, however, the gap between the testing scores are minimal across all metrics, indicating a well-managed bias-variance tradeoff within the model.

For each score:

* **Accuracy**: testing accuracy (0.977) is slightly lower than the perfect training score(1.0), suggesting the model is accurately trained
* **Precision**: perfect training score (1.0) leading to a high testing score (0.985) indicates that the model is exceptionally good at identifying true positives.
* **Recall**: the gap between training (1.0) and testing (0.953) score is relatively higher than the other scores, still, the gap is minor where the model may have missed a few relevant cases.
* **F1 Score**: Both scores of train (1.0) and test (0.968) of F1 are high, showing a robust balance between precision and recall.
* **AUC**: the auc of testing is also close to perfection (0.993) reflecting the model's strong capability to classify.

### 6.2.2 Dataset 2

|  |
| --- |
| | Metric | Mean Train Score | Mean Test Score | | --- | --- | --- | | Accuracy | 0.788832 | 0.696901 | | Precision | 0.723765 | 0.577149 | | Recall | 0.632639 | 0.531250 | | F1 | 0.674640 | 0.541320 | | AUC | 0.864420 | 0.736092 |   **Table 13: AdaBoost Cross Validation Result - Dataset 2** |

The cross-validation results for the model presented in the figure 27 and table 13. Notably, there is a more significant gaps between the training and testing scores across all metrics, which could suggest a potential for improvement to reduce possible overfitting.

For each score:

* **Accuracy**: the training accuracy (0.789) falls in the testing accuracy (0.697)
* **Precision**: the precision score shows a drop from training (0.724) to testing (0.577), indicating the model is not as good as predicting true positives in the test data.
* **Recall**: recall scores is lower in the training set compared to the other metrics, and is even decreasing from training (0.633) to testing (0.531), indicating that the model is less able to find relevant instances.
* **F1 Score**: due to the decrease of both precision and recall scores in testing, F1 scores also fall from 0.675 to 0.541
* **AUC**: the auc score is robust compared to the other metrics, however still decreases in the test score (0.864 → 0.736), suggesting that the model can overall classify data pretty well, but better in the training data.

# 7. Neural Network

## Implementation Description

For implementing our neural network, we used **pytorch**. Our **two hidden layers** use torch.nn.Linear as the input functions, using **torch.nn.functional.sigmoid** as the activation functions. Then a final torch.nn.Linear function is used to transition to the output layer. Please note that we used **torch.nn.CrossEntropyLoss** during our training to calculate cross entropy loss, and that this also applies a **softmax output layer**, so we did not need to implement softmax separately [2]. Our neural network has several hyperparameters, and each of the options for these hyperparameters are shown in Table 14.

| Hyperparameters (With Default Settings): | Options (Ranges): |
| --- | --- |
| h\_1\_o\_s=256 | [256, 392] |
| h\_2\_o\_s=128 | [128, 196] |
| w=None | [None, ["hidden\_1"], ["hidden\_2"], ["hidden\_1", "hidden\_2"]] |
| b=[1, 1] | [[1, 1], [1, 0], [0, 1], [0, 0]] |
| lr=0.5 | [0.25, 0.5, 0.75] |

**Table 14**

### Hyperparameters

#### Different Number of Hidden Units

This project asked for different numbers of hidden units to be implemented, so we did this through the ‘h\_1\_o\_s’ and ‘h\_2\_o\_s’ hyperparameters. ‘h\_1\_o\_s’ stands for hidden\_1\_output\_size, so this value was used to set the output size for the first hidden layer and the input size for the second hidden layer within the neural network. Similarly, ‘h\_2\_o\_s’ stands for hidden\_2\_output\_size, so this value was used to set the output size for the second hidden layer and the input size for the final (output) layer. Of note, while all other possible combinations of hyperparameters were tried, ‘h\_1\_o\_s’ was only set to 256 when ‘h\_2\_o\_s’ was set to 128, and likewise ‘h\_1\_o\_s’ was only set to 392 when ‘h\_2\_o\_s’ was set to 196.

#### Different Weight Initializations

This project asked for different weight initializations to be implemented, so we did this through the ‘w’ hyperparameter. When set to ‘None’, this resulted in keeping both of the functions to the hidden layers using their default weight initializations, which for a function of shape , the weights are initialized as where [1]. When set to ‘[“hidden\_1”]’, the function to the second hidden layer uses the default weight initialization, whereas the function to the first hidden layer uses ‘nn.init.xavier\_normal\_’ to initialize its weights. When set to ‘[“hidden\_2”]’, the function to the first hidden layer uses the default weight initialization, whereas the function to the second hidden layer uses ‘nn.init.xavier\_normal\_’ to initialize its weights. When set to ‘["hidden\_1", "hidden\_2"]’, the functions to both hidden layers use ‘nn.init.xavier\_normal\_’ to initialize their weights.

#### Different Bias Initializations

This project asked for different bias initializations to be implemented, so we did this through the ‘b’ hyperparameter. For the torch.nn.Linear functions, they have a ‘bias’ argument that if set to ‘True’, the layer will learn an additive bias, but if set to ‘False’ it will not. Thus when ‘b’ was set to [1, 1], bias was set to true for the functions to both of the hidden layers. When set to [1, 0], the function to the first hidden layer had its bias set to true and the function to the second hidden layer had its bias set to false. When set to [0, 1], the function to the first hidden layer had its bias set to false and the function to the second hidden layer had its bias set to true. When set to [0, 0], bias was set to false for the functions to both of the hidden layers.

#### Different Learning Rates

This project asked for different learning rates to be implemented, so we did this through the ‘lr’ hyperparameter. Please note that ‘lr’ was not used within the neural network initialization apart from setting self.learning\_rate equal to the value of ‘lr’. This learning rate that was set was then used later in our optimizer during training and was set to either 0.25, 0.5, or 0.75.

### Training

For each hyperparameter combination (with the exception of the output sizes stated above), training was conducted on the 50k training images. During training, we used torch.optim.SGD as our optimizer for faster learning, for which we were able to set the learning rate based on the hyperparameter setting previously decided for each model. Additionally, we used 3 epochs in hopes of minimizing results of underfitting. As stated previously, we used torch.nn.CrossEntropyLoss to keep track of the loss throughout training. Then as data to analyze later, we kept track of the average loss per epoch for each model.

### Testing

For each hyperparameter combination (with the exception of the output sizes stated above), testing was conducted on the 10k testing images. As data to analyze later, we kept track of the accuracy for each model.

### Outputs

When the training and testing is run for all models, an output file is created. (Note, it is setup to create a new output file each time, so if the code is run again it will not delete the previous results, but rather create ‘output\_#.txt’ where # will start at 0 and increase each time it is run.) Reported in the output file are the hyperparameters, average loss per epoch, and accuracy for each model.

## Results and Analysis

### General Results: Accuracy vs. Average Loss Per Epoch

In Figure 28 is a graph representing the accuracy versus the average loss per epoch for each model. While the trend is not perfect, a general trend can be seen that as the average loss per epoch increases, the accuracy decreases. The points are labeled with their corresponding accuracy.



This general trend makes sense because if a model is having more trouble during training, it makes sense that it will not have as accurate results when testing.

### Effects of Different Hyperparameters

Below are several graphs comparing each of the hyperparameters with average loss per epoch and accuracy. These graphs were generated by transferring the results from ‘output.txt’ to a Google Sheets spreadsheet (linked [here](https://docs.google.com/spreadsheets/d/1qzYMNcnCT2Up4tmVSCMZmwvsZsv7UiYtyACtKVmZgA0/edit#gid=196499879)), and then making the corresponding graphs. The purpose of these graphs are to discuss how the different hyperparameters may or may not affect the training/testing performance.

#### Hidden Units

As can be seen in Figure 29, the trends seem pretty similar; however, when the data has a lower average loss (for about ⅓ of the models), the loss seems significantly lower for the hidden units (256, 128). This indicates that while a different parameter may result in lower average loss (perhaps a parameter that affects ⅓ of the models like learning rate), the number of hidden units then further affects the average loss. Thus these results indicate that the number of hidden units does significantly affect the training performance. Furthermore, as can be seen in the Figure 30, the trends once again seem pretty similar; however, when the data has higher accuracies, the accuracy seems consistently higher for the hidden units (256, 128). This indicates that while a different parameter may result in higher accuracy, the number of hidden units then further affects the accuracy. Thus these results indicate that the number of hidden units does significantly affect the testing performance. Thus we can conclude that the number of hidden units significantly affects training and testing performance.





#### Weight Initializations

As can be seen in Figure 31, all of the trendlines for each of the weight initializations are pretty similar, indicating that the different weight initializations did not significantly affect the training performance. Furthermore, as can be seen in Figure 32, all of the trendlines for each of the weight initializations are once again pretty similar, with the exception of the trendline for the weight initialization of [“hidden\_1”]. Furthermore, this trendline seems fairly higher than the others, indicating its significance. Thus we can conclude that weight initialization does not significantly affect training performance, but does affect testing performance.





#### Bias Initializations

As can be seen in Figure 33, all of the trendlines for each of the bias initializations are pretty similar, indicating that the different bias initializations did not significantly affect the training performance. Furthermore, as can be seen in Figure 34, all of the trendlines for each of the bias initializations are once again pretty similar, indicating that the different bias initializations did not significantly affect the testing performance. Of note for this graph, the trendline for the bias initialization of [0, 1] indicates a seemingly higher accuracy; however, the bias initialization is not consistently higher in accuracy for the section of higher accuracies for all the models, and the trendline is still pretty similar to the others, so this trendline does not seem too significant. Thus we can conclude that bias initialization does not significantly affect training or testing performance.





#### Learning Rates

As can be seen in Figure 35, while the results for learning rates 0.5 and 0.75 are pretty similar, the results for the learning rate of 0.25 are significantly different. This indicates that a lower learning rate of 0.25 greatly improved (by decreasing) the average loss per epoch in comparison to the other learning weights. This makes sense because this lower learning rate results in the model taking more time to learn the training data, resulting in lower loss and better training results. Furthermore, as can be seen in Figure 36, the accuracy also appears to be significantly impacted by the lower learning rate of 0.25. Again, this makes sense because having a lower learning rate results in more time training, which results in better training results, which then impacts testing results, resulting in the much higher accuracy pictured in the graph. Thus we can conclude that the learning rate significantly affects training and testing performance.





### Conclusions

#### Hyperparameter Affects

From the above sections, we have seen that a loose general trend can be seen that as the average loss per epoch increases, the accuracy decreases. Furthermore, because of how stark the learning rate results are, it appears that the learning rate is the hyperparameter that most significantly affected both the training and testing performance, with the number of hidden units also affecting the training and testing performance. Weight initialization did not appear to affect training performance, but did seem to affect testing performance. Bias initialization did not appear to affect training or testing performance.

#### Best Performing Models

The highest accuracy was 0.5156, with the model’s hyperparameters as (h\_1\_o\_s=256, h\_2\_o\_s=128, w=[“hidden\_1”], b=[0, 0], lr=0.25), and the corresponding average loss per epoch was 1.611397136 (which of note was the second lowest average loss per epoch).

The lowest average loss per epoch was 1.541628739, with the model’s hyperparameters as (h\_1\_o\_s=256, h\_2\_o\_s=128, w=[“hidden\_2”], b=[0, 0], lr=0.25), and the corresponding accuracy was 0.2136 (which was the 27th highest accuracy).

Of note, while both of these models had the best average losses per epoch, the hyperparameter difference between them was weight, demonstrating [“hidden\_1”]'s effect leading to higher accuracy. Additionally, they both have lr=0.25 and hidden units (256, 128), solidifying the effects seen by learning rate and hidden units affecting both training and testing performance.

# Sources

[1] <https://github.com/pytorch/pytorch/blob/d954ef208f88f9b0ec1aef73edcd2149cc7845fa/torch/nn/modules/linear.py#L49>

[2] <https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html#:~:text=Note%20that%20this%20case%20is%20equivalent%20to%20applying%20LogSoftmax%20on%20an%20input%2C%20followed%20by%20NLLLoss>

[3] [Plot classification boundaries with different SVM Kernels — scikit-learn 1.3.2 documentation](https://scikit-learn.org/stable/auto_examples/svm/plot_svm_kernels.html#sphx-glr-auto-examples-svm-plot-svm-kernels-py)

[4] [Support Vector Machines Part 1 (of 3): Main Ideas!!! (youtube.com)](https://www.youtube.com/watch?v=efR1C6CvhmE)

[5] <https://www.smartdraw.com/decision-tree/>

[6] <https://www.spotfire.com/content/dam/spotfire/images/graphics/inforgraphics/random-forest-diagram.svg>