# Credit Card Fraud Detection Report

By: Herman Lin and Mahika Jain

## Introduction

Credit cards are the most common method of electronic payment used today and credit card companies have the responsibility of protecting their customers against credit card fraud. Credit card fraud is the unauthorized use of an individual’s credit or debit card to obtain money or property and it can be difficult to catch those responsible for it, leaving the company liable to pay the customer for damages. For this reason, companies aim to prevent fraud by setting up a system to detect it early on before significant damage is done.

The goal of this project is to determine which model will best predict whether a credit card transaction is fraudulent or not. We are using a dataset obtained from Kaggle, called Credit Card Fraud Detection. The dataset was collected and analyzed during a research collaboration of Worldline and the Machine Learning Group of ULB and contains transactions made by credit card users within the span of two days, in September 2013 by European cardholders. The dataset has 284,807 transactions, out of which 492 are fraud and 284,315 are not. Looking at the numbers, we can clearly see that this is a highly unbalanced dataset as the fraud cases only account for 0.172% of all transactions.

There are 30 features in this dataset: Time, Amount, V1 through V28. As per the author(s) of the dataset, “due to confidentiality issues, [they] cannot provide the original features and more background information about the data”, which is why most of the feature labels are unknown. What we do know is that “feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset”, and the “feature 'Amount' is the transaction amount, this feature can be used for example - dependant cost-sensitive learning”. We also know that features ‘V1….V28’ are the “principal components obtained with PCA”, therefore the only features to have gone through PCA transformation. The last column of the dataset, ‘Class’, is our target vector and has two values [0: transaction is not a fraud, 1: transaction is a fraud]. Since there are only two values in the target vector, we know we will be performing a binary classification.

Our process for determining the best model will only include supervised learning. Since most of our features have gone through Principal Component Analysis (PCA) transformation, which is an unsupervised learning technique used to reduce the dimension of the data with minimum loss of information, we did not deem it necessary to perform PCA again on the dataset. The three supervised learning models used in this analysis are Logistic regression, SVM, and Neural Network.

## Data Pre-Processing

Before we began the supervised learning, we had to address the issue regarding the highly unbalanced dataset, with the number of Class 1s being significantly smaller than the number of Class 0s. This was an issue because the bias towards Class 0s in the training dataset may influence the models to overlook the Class 1 transactions. There were two options we considered in trying to reduce the imbalanced dataset: oversampling and undersampling.

## Oversampling

Oversampling is the process of replicating the data of the minority class to increase their population. This dataset is a 579:1 ratio of Case 0s to Case 1s. Regardless, if we took a smaller overall dataset, we would need many duplicates of the Case 1 data before we have enough transactions for a good dataset size. This may undermine our analysis of the data as replication causes features to appear to have lower variance than they actually do. In the real world, cases of fraud are unlikely and uncommon. Using replications of those rare cases may end up causing overfitting by training our model to only be sensitive to the particular conditions of those 492 cases.

## Undersampling

Undersampling is the process of randomly downsampling the majority class. Reducing the 579:1 ratio would help balance out the stark difference in the number of Case 0 and 1s while preserving the integrity of the dataset. We did note that in randomly undersampling, we may lose informative transactions that could be helpful for our model, but given the size of Class 0, we were willing to take the risk. Although undersampling may cause the features to seem to have a higher variance than they actually do, it helps us in reducing the overall size of the dataset. One of the reasons for needing to reduce the dataset size was to reduce the computational time and this is especially important as we are running multiple models and undersampling tends to have lower power consumption. Using the information we know about oversampling and undersampling, we decided to perform an undersampling of our very large dataset.

The dataset was reduced to a 5:1 ratio, 2,460 Case 0s that were randomly chosen and 492 Case 1s. While this is still an unbalanced dataset, meaning overfitting will still likely be an issue, it has been improved from what the original dataset was, and the unbalanced nature of the dataset will be taken into account when drawing conclusions from the models.

All models in the project are performed using the Scikit Learn library.

# Scikit Learn Model Implementation

In order to train all of our models, we have generalized our algorithm into a macro-model training function called sklearn_model() depicted below:

```
def sklearn_model(X_tr, y_tr, X_ts, y_ts, m_type, c, iters, penalty='none', kernel=None,
hidden_layer_sizes=None, activation=None, alpha=0.0001):
    acc_tr_model = []
    acc_ts_model = []
    c_model = []
    model = None

    # create model
    if m_type == 0:
        print('Training Logistic Regression Model...')
        model = LogisticRegression(penalty=penalty, C=c, solver='saga', max_iter=iters)
    elif m_type == 1:
        print('Training SVM Model...')
        model = svm.SVC(probability=True, kernel=kernel, C=c)
    elif m_type == 2:
        print('Training Neural Network...')
        model = MLPClassifier(hidden_layer_sizes=hidden_layer_sizes, activation=activation,
 alpha=alpha, max_iter=iters)

    # ========== Code below adapted from Homework 5 ==========
    # fit the model
    model.fit(X_tr, y_tr)

    # yhat values
    yhat_tr = model.predict(X_tr)
    yhat_ts = model.predict(X_ts)

    # calculate and add accuracy values to respective lists
    acc_tr = model.score(X_tr, y_tr)
    acc_tr_model.append(acc_tr)
    acc_ts = model.score(X_ts, y_ts)
    acc_ts_model.append(acc_ts)

    # appending value of c for graphing purposes if needed
    c_model.append(c)
    # ========================================================

    # creating a classification report for analysis
    class_report_tr = classification_report(y_tr, yhat_tr, output_dict=True)
    class_report_ts = classification_report(y_ts, yhat_ts, output_dict=True)

    print("Model Complete!")

    return (acc_tr_model, acc_ts_model, c_model), (class_report_tr, class_report_ts), model
```

Within this function, we determine which method of supervised learning we are trying to attempt and then perform the model training, testing, and data retrieval for the respective model. At the end of the function, we return the training and testing accuracies and C values associated with the model, as well as classification reports that contain the precision, recall, and f-scores for the model, and the model itself so that we can perform any further operations on the model if necessary.

## Analyzing Metrics

When analyzing the data from the models, we found not only the accuracy of the training and testing data but also the precision, recall, and f1-score. We cannot depend on accuracy as an analytic metric because our dataset is unbalanced. Accuracy is the number of correct predictions over the number of total predictions made. In an unbalanced class, accuracy does not make the distinction of accuracy for Class 0 and Class 1, causing this statistic to be misleading or incorrect. On the other hand, precision is how accurate the test data can predict frauds from the transactions and recall is the probability of predicting the transaction is a fraud and it actually is a fraud.

$$Precision \ = \ \# \, True\ Positives \ / \ \# \, Predicted\ positives \ = \ TP \ / \ (TP \ + \ FP)$$
$$Recall \ = \ \# \, True\ Positives \ / \ \# \, Positives \ = \ TP \ / \ (TP \ + \ FN)$$

Since we have a higher number of Class 0 transactions, the false positive rate will increase more slowly as the number of true negatives would be very high. Precision and recall are not affected by an unbalance between the two classes and can distinguish between them. They both also focus on the positive, fraud cases, which is what we want to be able to correctly detect from the dataset.

F1-score is a measure of how accurate a model is by taking the harmonic mean of the precision and recall score. It is used to compare the performance of the two classifiers and it gives a better measure of incorrectly classified cases than accuracy does. Since our data is unbalanced, the f1-score will yield a more unbiased estimate of the performance of the model and will be used in determining which model is a better predictor.

The data is represented using an Accuracy vs C values graph and a Precision-Recall Curve. The AP in the legend for the latter curve stands for Average Precision, which measures the mean of all precision scores at all possible thresholds, and is the area under the precision-recall curve. It helps compare how well models are at predictions without having to consider a particular threshold. Although we are graphing the Accuracy vs C graph, we will not be using it to draw conclusions about the dataset for the first two models, Logistic regression and SVM. For neural networks, Loss curves are graphed alongside the Precision-Recall Curve. The loss curve helps to identify the average cost of a model and the smaller the cost, the better predictor it is.

# Logistic Regression (LogReg)

Logistic regression is a linear classifier that will classify the data into Class 0 and Class 1 based on the linear input features. This is the first model trained to predict whether a transaction is fraudulent or not. We chose to use the logistic regression model as it is a popular regression model to use when doing binary classification.

## Data PreProcessing

The first step in performing Logistic Regression is data preprocessing. Preprocessing is an important step before analyzing data through different models and for this model we used standardization, or mean removal and variance scaling. There are two types of preprocessing performed on the dataset, Standardization and Polynomial Feature transformation.

Standardization is usually done before PCA to prevent features with higher variance having a greater weight than features with lower variance. However, since PCA was done by the author(s) of the dataset, we are unsure if any standardized preprocessing was done on the dataset. Standardization is also done before Lasso and Ridge Regression as the scale of the features affects the coefficients applied, and a feature with a large variance will have a smaller coefficient and thus a smaller penalty. Since we are running logistic regression with and without regularization, we needed to standardize the dataset. This was done using the StandardScaler function in the preprocessing module from the Scikit-Learn library. The function will perform mean subtraction, which is when the mean is subtracted from every feature, and then normalization, which will help make all the features roughly the same size. After applying this function, the scaled data has a mean of zero and unit variance.

Polynomial Feature transformation is taking a feature and raising it to a specified degree to create a new feature, for all features in the dataset. It can be used to dynamically increase the number of features, and a higher degree can help lower the in-sample error. However, there is a risk of worsening the generalization error. This transformation was performed using the PolynomialFeatures function from the Scikit-Learn library, with a degree of two.

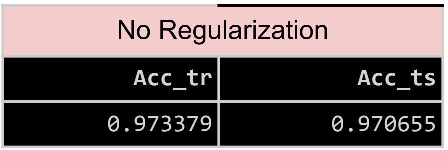
## Logistic regression model

The dataset is split randomly into 70% training set and 30% test set. The model was run in three different ways: without regularization, with L1 regularization, with L2 regularization.

### Logistic regression without regularization

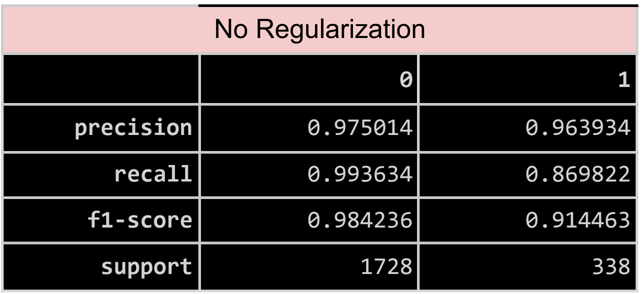
This model was performed with the penalty parameter set to ‘none’ which disregards the regularization term in Scikit Learn’s LogisticRegression function. The number of iterations is initialized to a large number, 10,000, as the LogisticRegression function in the Scikit Learn Library automatically stops iterating after the function converges, so the number of iterations does have any effect on the model. This is due to the solver parameter in the function that is assigned to ‘saga’. Since we have standardized our features to have approximately the same scale, the ‘saga’ solver will cause the function to  converge faster.

I. Standardization Data Observations



| No Regularization | |
|---|---|
| Acc_tr | Acc_ts |
| 0.973379 | 0.970655 |

*Table: Training/Testing Accuracies for LogReg, No Regularization*

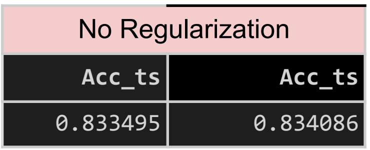


| No Regularization | | |
|---|---|---|
| | 0 | 1 |
| precision | 0.975014 | 0.963934 |
| recall | 0.993634 | 0.869822 |
| f1-score | 0.984236 | 0.914463 |
| support | 1728 | 338 |

*Table: Precision/Recall/F1-Score for LogReg, No Regularization*

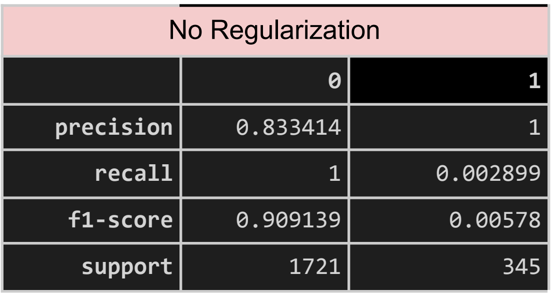
The model had very high and similar accuracies for both the training and test data. The accuracy of the training data is 0.973379 and the accuracy of the test data is 0.970655. The precision for Class 0 is 0.975014 and 0.963934 for Class 1. The recall for Class 0 is 0.993634 and 0.869822 for Class 1. The f1-score is 0.984236 for Class 0 and 0.914463 for Class 1. The Precision-Recall curve has a high AP of 0.95.

II. Polynomial Feature Transformation Data Observations (degree 2)



| No Regularization | |
|---|---|
| Acc_ts | Acc_ts |
| 0.833495 | 0.834086 |

*Table: Accuracies for LogReg, PFT, No Regularization*



No Regularization

| | 0 | 1 |
|---|---|---|
| precision | 0.833414 | 1 |
| recall | 1 | 0.002899 |
| f1-score | 0.909139 | 0.00578 |
| support | 1721 | 345 |

*Table: Precision/Recall/F1-Score for LogReg, PFT, No Regularization*

The model has an average accuracy and similar for both the training and test data. The accuracy of the training data is 0.833495 and the accuracy of the test data is 0.834086. The precision for Class 0 is 0.833414 and 1 for Class 1. The recall for Class 0 is 1 and 0.002899 for Class 1. The f1-score is 0.909139 for Class 0 and 0.00578 for Class 1. The Precision-Recall curve has a low AP of 0.22.

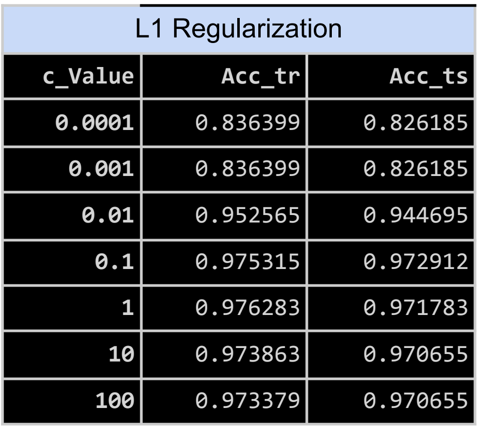
## Logistic regression with regularization

Regularization is applied to help reduce any generalization error that causes overfitting that may occur due to a feature having more weight over another. The amount of regularization that should be applied is determined by a parameter called lambda. Both regularizations are applied with a range of different C values: 0.0001, 0.001, 0.01, 0.1, 1, 10, 100. As the C values increase, the strength of the regularization decreases. The number of iterations for both are 10,000, for reasons explained in **I. Logistic regression without regularization** above.

## Logistic regression with L1 Regularization

L1, or Lasso, Regularization is when we remove features to remove variance. To do this, L1 norm adds the absolute value of magnitude of the coefficient as the penalty term to the loss function. Therefore, a large value for lambda will make the coefficient zero, removing the feature.

### I. Standardization Data Observations



L1 Regularization

| c_Value | Acc_tr | Acc_ts |
|---|---|---|
| 0.0001 | 0.836399 | 0.826185 |
| 0.001 | 0.836399 | 0.826185 |
| 0.01 | 0.952565 | 0.944695 |
| 0.1 | 0.975315 | 0.972912 |
| 1 | 0.976283 | 0.971783 |
| 10 | 0.973863 | 0.970655 |
| 100 | 0.973379 | 0.970655 |

*Table: Accuracies for LogReg, L1 Regularization*

| | cVal: 0.0001 | | cVal: 0.001 | | cVal: 0.01 | | cVal: 0.1 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.836399 | 0 | 0.836399 | 0 | 0.946331 | 1 | 0.973461 | 0.986441 |
| recall | 1 | 0 | 1 | 0 | 1 | 0.710059 | 0.997685 | 0.860947 |
| f1-score | 0.910912 | 0 | 0.910912 | 0 | 0.972425 | 0.83045 | 0.985424 | 0.919431 |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 |
| | cVal: 1 | | cVal: 10 | | cVal: 100 | | | |
| | 0 | 1 | 0 | 1 | 0 | 1 | | |
| precision | 0.975099 | 0.983278 | 0.975028 | 0.967105 | 0.975014 | 0.963934 | | |
| recall | 0.997106 | 0.869822 | 0.994213 | 0.869822 | 0.993634 | 0.869822 | | |
| f1-score | 0.98598 | 0.923077 | 0.984527 | 0.915888 | 0.984236 | 0.914463 | | |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | | |

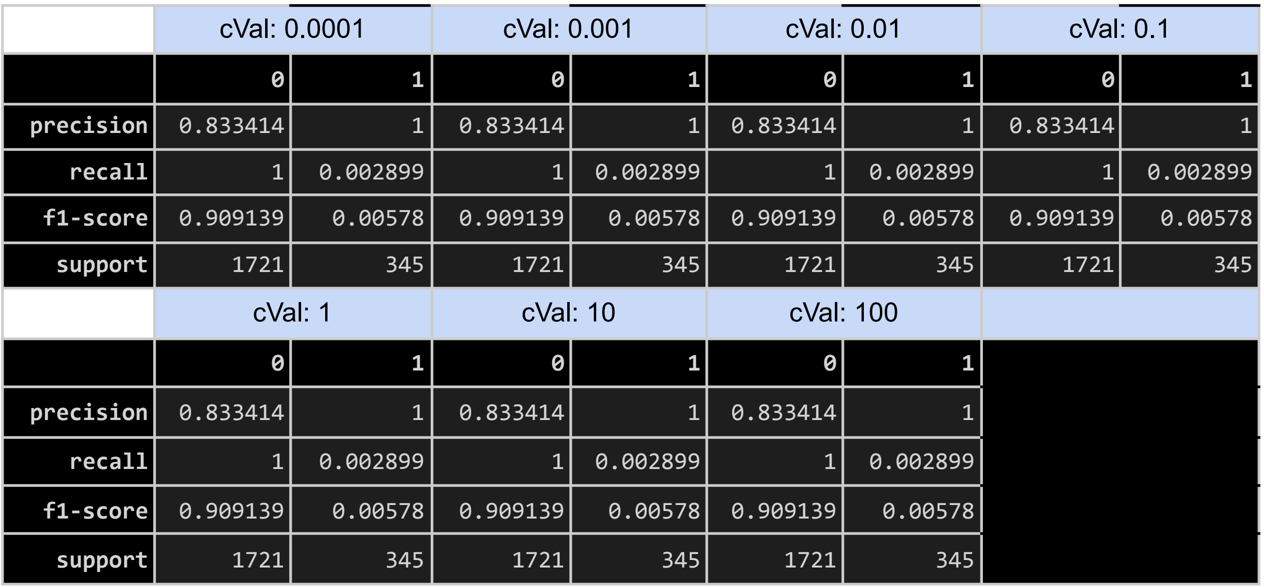
*Table: Precision/Recall/F1-Score for LogReg, L1 Regularization*

The model indicated that as the C values increased, the training and test data accuracies both increased from a range of 0.83 to 0.97. The accuracies of the training set are higher than the test set, but both sets have an overall high accuracy. For the C values 0.0001 and 0.001, the precision, recall, and f1-score were the same for both classes. An interesting point to note is that the metrics for Class 1 for these C values is 0. This changes 1 when the C value is increased to 0.01. For the C value 0.01, the precision and f1-score for both classes suddenly increased. The recall score for Class 0 remains the same, a value of 1, and for Class 1 it is 0.710059. As the C values increase from 0.1 to 100, the precision for Class 0 increases from 0.973461 to 0.975014, and for Class 1 the values decrease from 0.986441 to 0.963934. On the other hand, the recall score for Class 0 decreases from 0.997685 to 0.993634, and for Class 1 the values increase from 0.860947 to 0.869822. The f1-score for Class 0 and Class 1 increases slightly and then decreases. The Precision-Recall Curve has a high AP of 0.95 for C values 0.01 and greater, and a low AP of 0.17 for C values 0.001 and lower.

## II. Polynomial Feature Transformation Data Observations (degree 2)

| L1 Regularization | | |
|---|---|---|
| c_Value | Acc_ts | Acc_ts |
| 0.0001 | 0.833495 | 0.834086 |
| 0.001 | 0.833495 | 0.834086 |
| 0.01 | 0.833495 | 0.834086 |
| 0.1 | 0.833495 | 0.834086 |
| 1 | 0.833495 | 0.834086 |
| 10 | 0.833495 | 0.834086 |
| 100 | 0.833495 | 0.834086 |

*Table: Accuracies for LogReg, PFT, L1 Regularization*



| | cVal: 0.0001 | | cVal: 0.001 | | cVal: 0.01 | | cVal: 0.1 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.833414 | 1 | 0.833414 | 1 | 0.833414 | 1 | 0.833414 | 1 |
| recall | 1 | 0.002899 | 1 | 0.002899 | 1 | 0.002899 | 1 | 0.002899 |
| f1-score | 0.909139 | 0.00578 | 0.909139 | 0.00578 | 0.909139 | 0.00578 | 0.909139 | 0.00578 |
| support | 1721 | 345 | 1721 | 345 | 1721 | 345 | 1721 | 345 |

| | cVal: 1 | | cVal: 10 | | cVal: 100 | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.833414 | 1 | 0.833414 | 1 | 0.833414 | 1 |
| recall | 1 | 0.002899 | 1 | 0.002899 | 1 | 0.002899 |
| f1-score | 0.909139 | 0.00578 | 0.909139 | 0.00578 | 0.909139 | 0.00578 |
| support | 1721 | 345 | 1721 | 345 | 1721 | 345 |

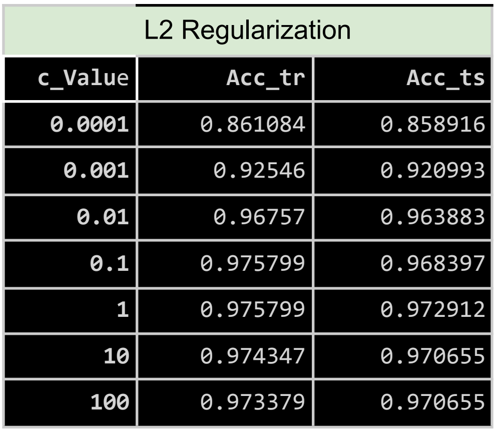
*Table: Precision/Recall/F1-Score for LogReg, PFT, L1 Regularization*

The model indicates that as the C values increase, the training and test accuracies stay the same. The model also indicates that as the C values increase, the precision, recall, and f1-score stay the same. The recall and f1-score for Class 1 identification are also very low compared to the Class 1 identifications.

## Logistic regression with L2 Regularization

L2, or Ridge, Regularization is when we add the squared magnitude of coefficients as the penalty term to the loss function. Unlike L1 regularization, a large value for lambda will add too much weight to the coefficient.

### I. Standardization Data Observations



| L2 Regularization | | |
|---|---|---|
| c_Value | Acc_tr | Acc_ts |
| 0.0001 | 0.861084 | 0.858916 |
| 0.001 | 0.92546 | 0.920993 |
| 0.01 | 0.96757 | 0.963883 |
| 0.1 | 0.975799 | 0.968397 |
| 1 | 0.975799 | 0.972912 |
| 10 | 0.974347 | 0.970655 |
| 100 | 0.973379 | 0.970655 |

*Table: Accuracies for LogReg, L2 Regularization*

|  | cVal: 0.0001 |  | cVal: 0.001 |  | cVal: 0.01 |  | cVal: 0.1 |  | cVal: 1 |  | cVal: 10 |  | cVal: 100 |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.857568 | 1 | 0.918172 | 1 | 0.962674 | 1 | 0.97241 | 0.996552 | 0.975085 | 0.98 | 0.975043 | 0.970297 | 0.975014 | 0.963934 |
| recall | 1 | 0.150888 | 1 | 0.544379 | 1 | 0.801775 | 0.999421 | 0.85503 | 0.996528 | 0.869822 | 0.994792 | 0.869822 | 0.993634 | 0.869822 |
| f1-score | 0.923324 | 0.262211 | 0.957341 | 0.704981 | 0.980982 | 0.889984 | 0.985731 | 0.920382 | 0.98569 | 0.92163 | 0.984818 | 0.917317 | 0.984236 | 0.914463 |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 |

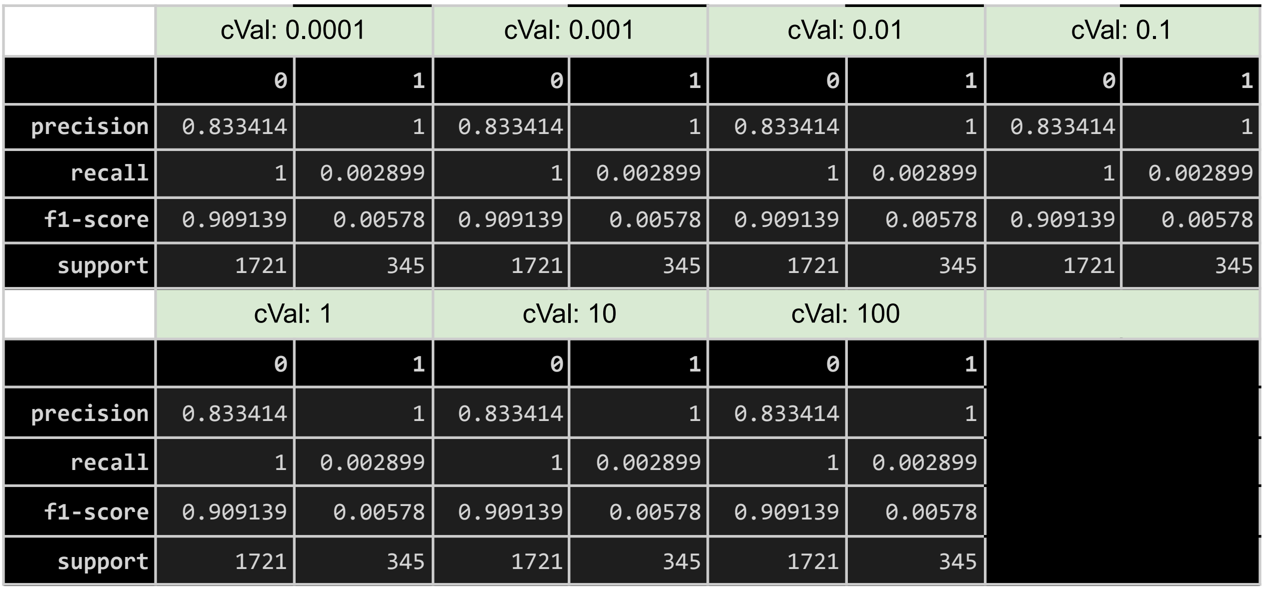
*Table: Precision/Recall/F1-Score for LogReg, L2 Regularization*

The model indicates that as the C values increase, the accuracies for both training and test data increase, and then decrease when C is 10 and higher. Looking at the other metrics, the precision for Class 0 increases as the C values increase and then a negligible sign of decrease can be seen when C is 10 and higher. For Class 1, the precision is 1 until C is 0.1, from when it starts to decrease. The recall for Class 0 is 1 until C is 0.1, from when it starts to decrease. For Class 1, the recall score is very low, 0.150888, when C is 0.0001, but as the C values increase, the score also increases. The fi-score for Class 0 increases to 0.985731, before it starts to decrease to 0.984236 when C is 1. For Class 1, the score started at a low 0.262211 and then quickly increased to 0.921630 when C was 1, but then decreased to 0.914463. The Precision-Recall Curve has a high AP for all C values, the highest being 0.96 for Cs 0.01 and 0.1.

## II. Polynomial Feature Transformation Data Observations (degree 2)

| L2 Regularization |  |  |
|---|---|---|
| c_Value | Acc_ts | Acc_ts |
| 0.0001 | 0.833495 | 0.834086 |
| 0.001 | 0.833495 | 0.834086 |
| 0.01 | 0.833495 | 0.834086 |
| 0.1 | 0.833495 | 0.834086 |
| 1 | 0.833495 | 0.834086 |
| 10 | 0.833495 | 0.834086 |
| 100 | 0.833495 | 0.834086 |

*Table: Accuracies for LogReg, PFT, L2 Regularization*



| | cVal: 0.0001 | | cVal: 0.001 | | cVal: 0.01 | | cVal: 0.1 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.833414 | 1 | 0.833414 | 1 | 0.833414 | 1 | 0.833414 | 1 |
| recall | 1 | 0.002899 | 1 | 0.002899 | 1 | 0.002899 | 1 | 0.002899 |
| f1-score | 0.909139 | 0.00578 | 0.909139 | 0.00578 | 0.909139 | 0.00578 | 0.909139 | 0.00578 |
| support | 1721 | 345 | 1721 | 345 | 1721 | 345 | 1721 | 345 |

| | cVal: 1 | | cVal: 10 | | cVal: 100 | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.833414 | 1 | 0.833414 | 1 | 0.833414 | 1 |
| recall | 1 | 0.002899 | 1 | 0.002899 | 1 | 0.002899 |
| f1-score | 0.909139 | 0.00578 | 0.909139 | 0.00578 | 0.909139 | 0.00578 |
| support | 1721 | 345 | 1721 | 345 | 1721 | 345 |

*Table: Precision/Recall/F1-Score for LogReg, PFT, L2 Regularization*

The model indicates that as the C values increase, the training and test accuracies stay the same. The model also indicates that as the C values increase, the precision, recall, and f1-score stay the same. The recall and f1-score for Class 1 identification are also very low compared to the Class 1 identifications.

## Summary of Results

Overall, the logistic regression models using the StandardScaler transformed data outperformed the logistic regression models using the Polynomial Feature transformed data. The lack of change in the accuracy values as well as the precision, recall, and f1-score also suggest that the use of regularization, L1 or L2, has no impact on the use of the polynomial feature transformed data for the logistic regression model. On the other hand, the logistic regression models that used the StandardScaler transformed data performed well when regularization was added, with complementing better f1-scores and AP values.

# Support Vector Machines (SVMs)

Support Vector Machines are models that are used for both classification and regression analysis. In classification, the algorithm works by taking training examples and separating them into their categories, while maximizing the width of the gap between the two categories. The test examples are then predicted to belong to a category based on which side of the gap they fall. This is the second model trained to predict whether a transaction is fraudulent or not. We chose to use SVMs as they are effective for a large number of features and binary classification and the effect of imbalance in the dataset can be reduced by using regularization.

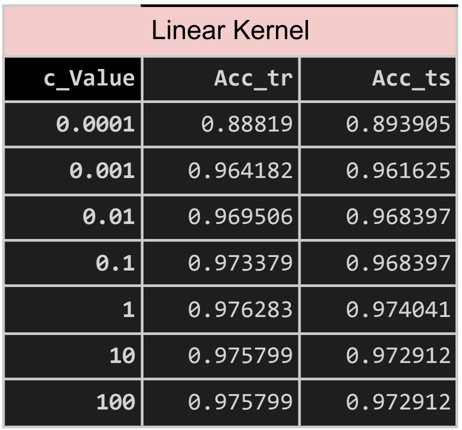
## SVM Model

The same 70/30 split dataset was also run through the SVM implementation of our sklearn_model(). The data was pre-processed using StandardScaler transformation. We tested three different SVMs using different kernels: linear, radial basis function (RBF), and polynomial.

## SVM with Linear Kernel

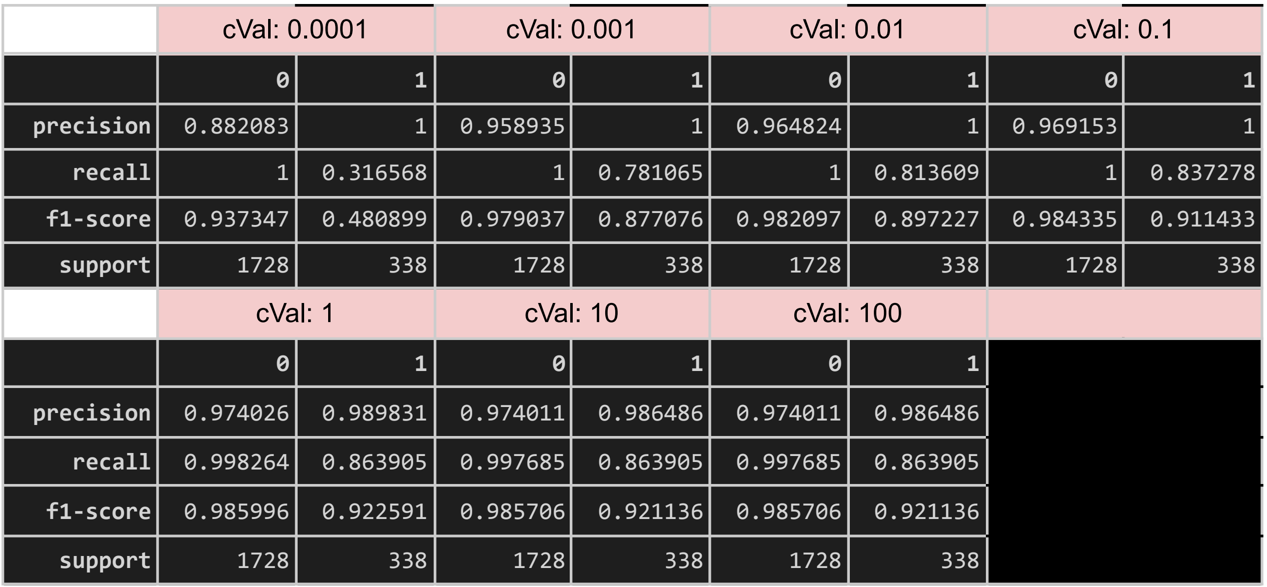
A linear kernel is the most common type of kernel, and is used when the data is linearly separable and the data has a large number of features. This kernel is the fastest among the three used. To test this kernel, we used a range of C values: 0.0001, 0.001, 0.01, 0.1, 1, 10, 100. The C parameter is the L2 regularization penalty.

### Data Observations



| Linear Kernel | | |
|---|---|---|
| c_Value | Acc_tr | Acc_ts |
| 0.0001 | 0.88819 | 0.893905 |
| 0.001 | 0.964182 | 0.961625 |
| 0.01 | 0.969506 | 0.968397 |
| 0.1 | 0.973379 | 0.968397 |
| 1 | 0.976283 | 0.974041 |
| 10 | 0.975799 | 0.972912 |
| 100 | 0.975799 | 0.972912 |

*Table: Accuracies for SVM with Linear Kernel*



| | cVal: 0.0001 | | cVal: 0.001 | | cVal: 0.01 | | cVal: 0.1 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.882083 | 1 | 0.958935 | 1 | 0.964824 | 1 | 0.969153 | 1 |
| recall | 1 | 0.316568 | 1 | 0.781065 | 1 | 0.813609 | 1 | 0.837278 |
| f1-score | 0.937347 | 0.480899 | 0.979037 | 0.877076 | 0.982097 | 0.897227 | 0.984335 | 0.911433 |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 |

| | cVal: 1 | | cVal: 10 | | cVal: 100 | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.974026 | 0.989831 | 0.974011 | 0.986486 | 0.974011 | 0.986486 |
| recall | 0.998264 | 0.863905 | 0.997685 | 0.863905 | 0.997685 | 0.863905 |
| f1-score | 0.985996 | 0.922591 | 0.985706 | 0.921136 | 0.985706 | 0.921136 |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 |

*Table: Precision/Recall/F1-Score for SVM with Linear Kernel*

The model indicates that as the C values increase, the accuracies for both training and test set increased until when C was 1 and then slightly decreased. The training accuracy increased from 0.888190 to 0.976283 and then stabilized at 0.975799. The test accuracy increased from 0.893905 to 0.974041 and then stabilized at 0.972912. Following a similar trajectory for the Class 0 precision score, they increased from 0.882083 to 0.974026 and then stabilized at 0.974011. The precision score for Class 1 decreased from 1 when C was 1 and stabilized at 0.986486. The recall score for Class 0 was approximately 1 for all Cs, but for Class 1 the score increased from 0.316568 to 0.863905. The f1-score for Class 0 increased from 0.937347 to 0.985996 and then stabilized at 0.985706. For Class1, the score increased from

0.480899 to 0.922591 and then stabilized at 0.921136. The Precision-Recall Curve has a high AP for all C values. The lowest AP 0.92 is for C value 0.0001 and highest is 0.96 for all Cs except 0.001.

## SVM with RBF Kernel

The Radial Basis Function (RBF) kernel is also a commonly used kernel as it has the advantages that the classes don't have to be linearly separable and requires less memory space as it only stores the support vectors during training and not the entire dataset. This kernel was also tested with a range of C values: 0.0001, 0.001, 0.01, 0.1, 1, 10, 100. The C parameter is the L2 regularization penalty.

### Data Observations

| Radial Basis Function Kernel | | |
|---|---|---|
| c_Value | Acc_tr | Acc_ts |
| 0.0001 | 0.836399 | 0.826185 |
| 0.001 | 0.836399 | 0.826185 |
| 0.01 | 0.836399 | 0.826185 |
| 0.1 | 0.963698 | 0.960497 |
| 1 | 0.979187 | 0.96614 |
| 10 | 0.992256 | 0.970655 |
| 100 | 0.999516 | 0.957111 |

*Table: Accuracies for SVM with Radial Basis Function Kernel*

| | cVal: 0.0001 | | cVal: 0.001 | | cVal: 0.01 | | cVal: 0.1 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.836399 | 0 | 0.836399 | 0 | 0.836399 | 0 | 0.968272 | 0.936877 |
| recall | 1 | 0 | 1 | 0 | 1 | 0 | 0.989005 | 0.83432 |
| f1-score | 0.910912 | 0 | 0.910912 | 0 | 0.910912 | 0 | 0.978528 | 0.882629 |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 |
| | cVal: 1 | | cVal: 10 | | cVal: 100 | | | |
| | 0 | 1 | 0 | 1 | 0 | 1 | | |
| precision | 0.976797 | 0.993311 | 0.991954 | 0.993865 | 0.999422 | 1 | | |
| recall | 0.998843 | 0.878698 | 0.998843 | 0.95858 | 1 | 0.997041 | | |
| f1-score | 0.987697 | 0.932496 | 0.995386 | 0.975904 | 0.999711 | 0.998519 | | |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | | |

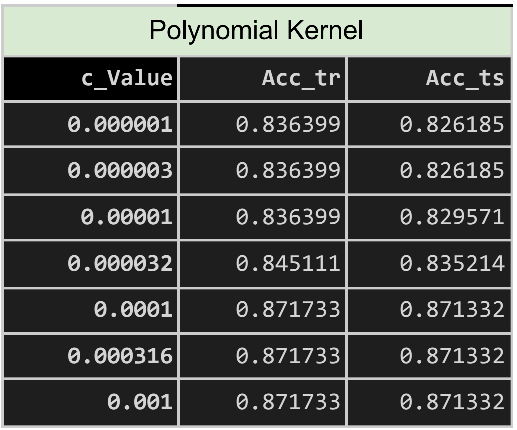
*Table: Precision/Recall/F1-Score for SVM with Radial Basis Function Kernel*

The model indicates that as the C value increases, the training accuracy increases from 0.836399 to 0.999516. The test accuracy increased from 0.826185 to 0.970655, before it slightly decreased to 0.957111 when C was 100. The precision score for Class 0 increased from 0.836399 to 0.999422 while C increased. For Class 1, the precision, recall score, and f1-score was 0 for C values 0.01 and lower. Then the precision score increased from 0.936877 to 1, the recall score increased from 0.834320 to 0.997041, and the f1-score increased from 0.882629 to 0.998519. The recall score for Class 0 was approximately 1 for all Cs. The f1-score for Class 0 increased from 0.910912 to 0.999711. The Precision-Recall Curve has a high AP for all C values. The lowest AP 0.92 is for C value 100 and highest is 0.96 for C values 1 and 10.

## SVM with Polynomial Kernel

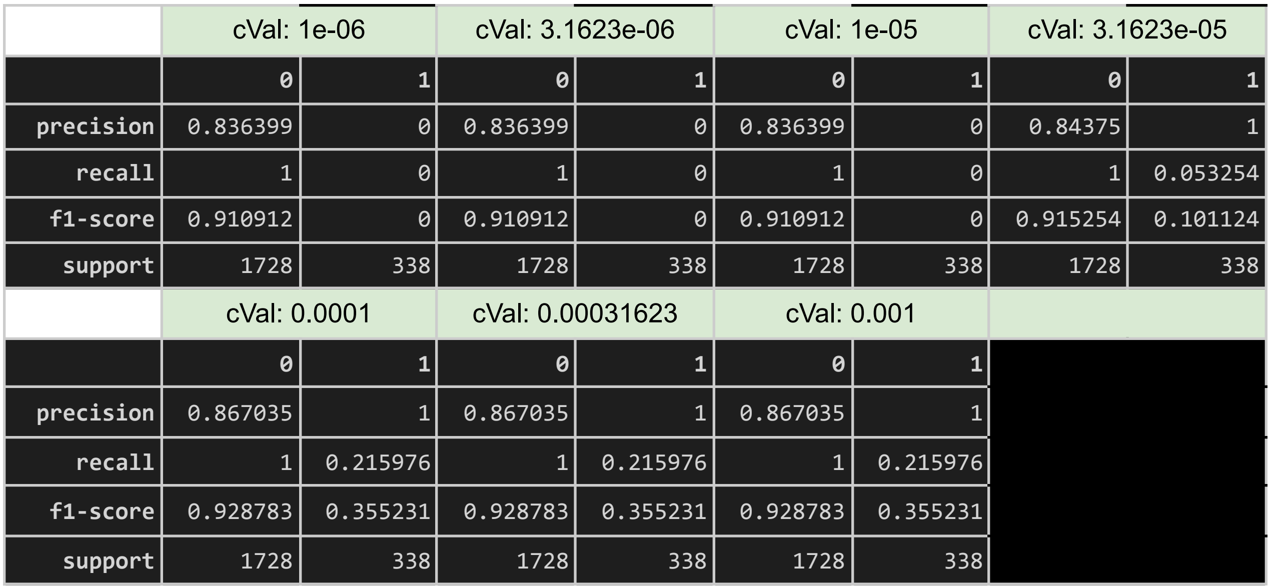
The Polynomial kernel is used because it allows us to learn from non-linear models. It not only looks at the original features but also combinations of them. The 7 C values for this kernel used are from 0.000001 to 0.001, each evenly spaced on a log scale.

### Data Observations



| Polynomial Kernel | | |
|---|---|---|
| c_Value | Acc_tr | Acc_ts |
| 0.000001 | 0.836399 | 0.826185 |
| 0.000003 | 0.836399 | 0.826185 |
| 0.00001 | 0.836399 | 0.829571 |
| 0.000032 | 0.845111 | 0.835214 |
| 0.0001 | 0.871733 | 0.871332 |
| 0.000316 | 0.871733 | 0.871332 |
| 0.001 | 0.871733 | 0.871332 |

*Table: Accuracies for SVM with Polynomial Kernel*



| | cVal: 1e-06 | | cVal: 3.1623e-06 | | cVal: 1e-05 | | cVal: 3.1623e-05 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.836399 | 0 | 0.836399 | 0 | 0.836399 | 0 | 0.84375 | 1 |
| recall | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0.053254 |
| f1-score | 0.910912 | 0 | 0.910912 | 0 | 0.910912 | 0 | 0.915254 | 0.101124 |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | 1728 | 338 |
| | cVal: 0.0001 | | cVal: 0.00031623 | | cVal: 0.001 | | | |
| | 0 | 1 | 0 | 1 | 0 | 1 | | |
| precision | 0.867035 | 1 | 0.867035 | 1 | 0.867035 | 1 | | |
| recall | 1 | 0.215976 | 1 | 0.215976 | 1 | 0.215976 | | |
| f1-score | 0.928783 | 0.355231 | 0.928783 | 0.355231 | 0.928783 | 0.355231 | | |
| support | 1728 | 338 | 1728 | 338 | 1728 | 338 | | |

*Table: Precision/Recall/F1-Score for SVM with Polynomial Kernel*

The model indicates that as the C value increases, the training accuracy increases from 0.836399 to 0.871733. The test accuracy increased from 0.826185 to 0.871332. Starting with Class 0, as the C values increased, the precision score increased from 0.836399 to 0.867035. The recall score was 1 for all C values and the f1-score increased from 0.910912 to 0.928783. Now looking at Class 1, for the first three C values, the precision, recall, and f1-score was 0. After that the precision score was 1 for the remaining C values, the recall score increased from

0.053254 to 0.215976, and the f1-score increased from 0.101124 to 0.355231. The Precision-Recall Curve has a slightly different range of AP values from the other curves of 0.89 to 0.91. The AP increases as C values increase.

## Summary of Results

For the SVM using a Linear Kernel, the precision, recall, and f1-score plateau starting from a C value of 1 and higher. This tells us that the model is likely to exhibit the same behavior for further iterations of the model with higher C values. We can also see this happen with the SVM using a Polynomial Kernel, where the values begin to plateau starting from a C value of 0.0001. The SVM using a RBF Kernel had precision, recall, and f1-score kept increasing as the C value increased all the way to the highest C value we tested. Thus, the best models for the Linear, RBF, and Polynomial kernels were when the C value was equal to 1, 100, and 0.0001 respectively.

Of the three kernels used to train the SVM models, the SVM using a RBF Kernel showed the most promising results in terms of precision, recall, and f1-score. Comparing the f1-scores obtained at the end of all the model training and testing, we see that all the models exhibited an increase in the f1-scores for both Class 0 and Class 1. On the other hand, the SVM using a linear kernel achieved the best average precision results out of the three kernel types. In the end, from our results we are able to conclude that both the Linear and RBF kernels are better than their Polynomial kernel counterpart.

# Neural Networks (NNs)

Neural Networks are a series of algorithms used to recognize underlying relationships in a dataset to draw correlations and be able to classify data correctly. This is the third model trained to predict whether a transaction is fraudulent or not. We chose to use Neural Networks as they are especially helpful in predicting rare events like fraud detection and can help in the decision making process.

## Neural Networks Model

Our Neural Network models are also trained on the same 70/30 split dataset. The data was pre-processed using StandardScaler transformation. We trained several models of neural networks based on different activation functions, different hidden layers based on the number and size of the hidden layers, and different L2 regularization penalty parameters. These variables are as follows:

Activation Functions: ReLU, tanh, or logistic/sigmoid
Hidden Layers: [(22), (22, 22), (22, 22, 22), (30)]
Alphas: [0.01, 0.001, 0.0001, 0.00001]

Each activation function was run with the different variations of hidden layers and each variation of hidden layers was run with the different alpha values. The number of iterations for each of the neural networks was initialized to a large number of 10,000, as the MLPClassifier function in the Scikit Learn Library automatically halts further iterations after the function converges. The ‘solver’ we used for weight optimization is the default called ‘adam’ as it works well on relatively large datasets. The model function used optimizes the log-loss function using stochastic gradient descent.

In order to draw conclusions from the data, we will focus on the comprehensive data found from the graphs.

## NN with logistic (sigmoid) Activation Function

The Sigmoid function is generally used for binary classification problems.

### Data Observations

| Logistic/Sigmoid Activation Function | | |
|---|---|---|
| Hidden Layer, Alpha | Acc_tr | Acc_ts |
| layers=22, alpha=0.01 | 0.977735 | 0.978555 |
| alpha=0.001 | 0.977251 | 0.975169 |
| alpha=0.0001 | 0.976767 | 0.976298 |
| alpha=1e-05 | 0.976767 | 0.979684 |
| layers=(22, 22), alpha=0.01 | 0.979187 | 0.979684 |
| alpha=0.001 | 0.999516 | 0.975169 |
| alpha=0.0001 | 0.998548 | 0.969526 |
| alpha=1e-05 | 0.999516 | 0.971783 |
| layers=(22, 22, 22), alpha=0.01 | 0.999032 | 0.971783 |
| alpha=0.001 | 0.999516 | 0.967269 |
| alpha=0.0001 | 0.999516 | 0.96614 |
| alpha=1e-05 | 0.999032 | 0.959368 |
| layers=30, alpha=0.01 | 0.976767 | 0.975169 |
| alpha=0.001 | 0.976283 | 0.977427 |
| alpha=0.0001 | 0.976767 | 0.975169 |
| alpha=1e-05 | 0.977251 | 0.978555 |

*Table: Accuracies for NN with Logistic/Sigmoid Activation Function*

| layers=22 | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.977452 | 0.979452 | 0.976365 | 0.982699 | 0.976351 | 0.97931 | 0.977427 | 0.972789 |
| recall | 0.996552 | 0.877301 | 0.997126 | 0.871166 | 0.996552 | 0.871166 | 0.995402 | 0.877301 |
| f1-score | 0.98691 | 0.925566 | 0.986636 | 0.923577 | 0.986348 | 0.922078 | 0.986333 | 0.922581 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=(22, 22) | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.979108 | 0.979661 | 0.999426 | 1 | 0.998279 | 1 | 0.999426 | 1 |
| recall | 0.996552 | 0.886503 | 1 | 0.996933 | 1 | 0.990798 | 1 | 0.996933 |
| f1-score | 0.987753 | 0.930757 | 0.999713 | 0.998464 | 0.999139 | 0.995378 | 0.999713 | 0.998464 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=(22, 22, 22) | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| precision | 0.999425 | 0.996933 | 0.999426 | 1 | 0.999426 | 1 | 0.998852 | 1 |
| recall | 0.999425 | 0.996933 | 1 | 0.996933 | 1 | 0.996933 | 1 | 0.993865 |
| f1-score | 0.999425 | 0.996933 | 0.999713 | 0.998464 | 0.999713 | 0.998464 | 0.999426 | 0.996923 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=30 | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.975281 | 0.986014 | 0.976338 | 0.975945 | 0.976351 | 0.97931 | 0.977439 | 0.976109 |
| recall | 0.997701 | 0.865031 | 0.995977 | 0.871166 | 0.996552 | 0.871166 | 0.995977 | 0.877301 |
| f1-score | 0.986364 | 0.921569 | 0.98606 | 0.920583 | 0.986348 | 0.922078 | 0.986621 | 0.924071 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |

*Table: Precision/Recall/F1-Score for NN with Logistic/Sigmoid Activation Function*

The training set accuracy was approximately either 0.97 or 1 and test set accuracy was in the general range of 0.96 to 0.97, for all the different iterations of the model. Looking at the Precision-Recall Curve, the highest AP score of 0.96 is for the following iterations:

Hidden Layers: 22, Alphas: 0.01
Hidden Layers: 22, Alphas: 0.001
Hidden Layers: 22, Alphas: 0.0001
Hidden Layers: 22, Alphas: 0.00001
Hidden Layers: 30, Alphas: 0.01
Hidden Layers: 30, Alphas: 0.001
Hidden Layers: 30, Alphas: 0.0001
Hidden Layers: 30, Alphas: 0.00001

From the Loss curve we know that the smallest average cost is for the iteration:

Hidden Layers: (22,22), Alphas: 0.01

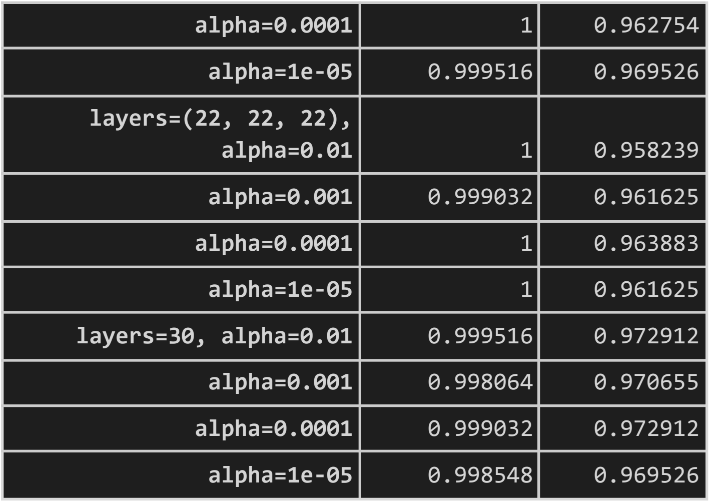
The sigmoid function, unfortunately, is not zero-centered, is computationally expensive, has high saturation and causes huge Vanishing Gradient Problems. For this reason, we will look at another activation function

## NN with tanh Activation Function

The Tanh function is zero centered.

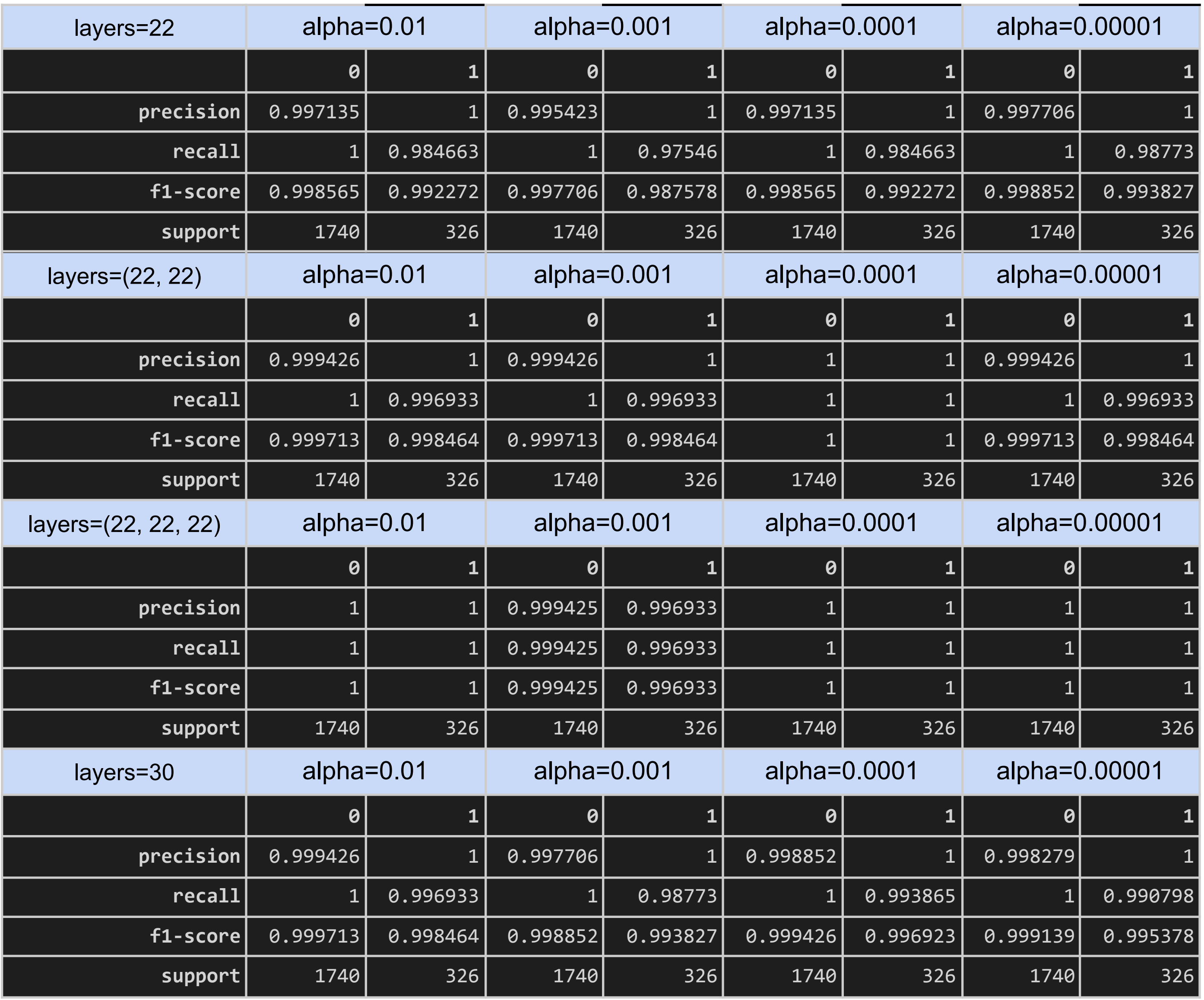
### Data Observations

| tanh Activation Function | | |
|---|---|---|
| Hidden Layer, Alpha | Acc_tr | Acc_ts |
| layers=22, alpha=0.01 | 0.99758 | 0.971783 |
| alpha=0.001 | 0.996128 | 0.971783 |
| alpha=0.0001 | 0.99758 | 0.975169 |
| alpha=1e-05 | 0.998064 | 0.977427 |
| layers=(22, 22), alpha=0.01 | 0.999516 | 0.962754 |
| alpha=0.001 | 0.999516 | 0.968397 |



| | | |
|---|---|---|
| alpha=0.0001 | 1 | 0.962754 |
| alpha=1e-05 | 0.999516 | 0.969526 |
| layers=(22, 22, 22), alpha=0.01 | 1 | 0.958239 |
| alpha=0.001 | 0.999032 | 0.961625 |
| alpha=0.0001 | 1 | 0.963883 |
| alpha=1e-05 | 1 | 0.961625 |
| layers=30, alpha=0.01 | 0.999516 | 0.972912 |
| alpha=0.001 | 0.998064 | 0.970655 |
| alpha=0.0001 | 0.999032 | 0.972912 |
| alpha=1e-05 | 0.998548 | 0.969526 |

*Table: Accuracies for NN with tanh Activation Function*



| layers=22 | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.997135 | 1 | 0.995423 | 1 | 0.997135 | 1 | 0.997706 | 1 |
| recall | 1 | 0.984663 | 1 | 0.97546 | 1 | 0.984663 | 1 | 0.98773 |
| f1-score | 0.998565 | 0.992272 | 0.997706 | 0.987578 | 0.998565 | 0.992272 | 0.998852 | 0.993827 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=(22, 22) | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.999426 | 1 | 0.999426 | 1 | 1 | 1 | 0.999426 | 1 |
| recall | 1 | 0.996933 | 1 | 0.996933 | 1 | 1 | 1 | 0.996933 |
| f1-score | 0.999713 | 0.998464 | 0.999713 | 0.998464 | 1 | 1 | 0.999713 | 0.998464 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=(22, 22, 22) | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 1 | 1 | 0.999425 | 0.996933 | 1 | 1 | 1 | 1 |
| recall | 1 | 1 | 0.999425 | 0.996933 | 1 | 1 | 1 | 1 |
| f1-score | 1 | 1 | 0.999425 | 0.996933 | 1 | 1 | 1 | 1 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=30 | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.999426 | 1 | 0.997706 | 1 | 0.998852 | 1 | 0.998279 | 1 |
| recall | 1 | 0.996933 | 1 | 0.98773 | 1 | 0.993865 | 1 | 0.990798 |
| f1-score | 0.999713 | 0.998464 | 0.998852 | 0.993827 | 0.999426 | 0.996923 | 0.999139 | 0.995378 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |

*Table: Precision/Recall/F1-Score for NN with tanh Activation Function*

The training set accuracy was approximately 1 and test set accuracy was in the general range of 0.95 to 0.97, for all the different iterations of the model. Looking at the Precision-Recall Curve, the highest AP score of 0.96 is for the following iterations:

Hidden Layers: 22, Alphas: 0.01
Hidden Layers: 22, Alphas: 0.00001
Hidden Layers: 30, Alphas: 0.01
Hidden Layers: 30, Alphas: 0.001

From the Loss curve we know that the smallest average cost is for the iteration:

Hidden Layers: 30, Alphas: 0.01

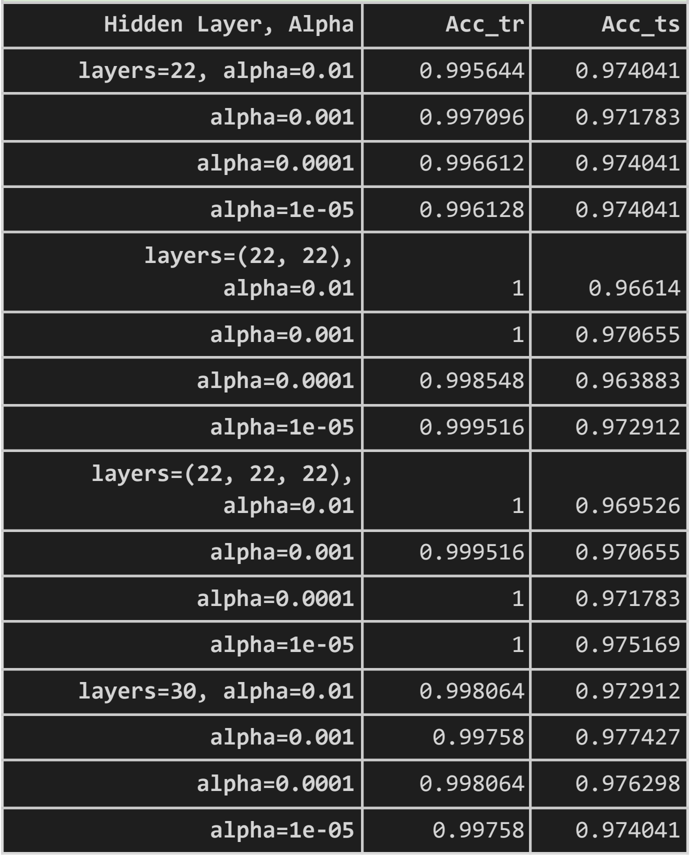
The tanh function, like the sigmoid function, has high saturation and causes huge Vanishing Gradient Problems. For this reason, we will look at another activation function

## NN with ReLU Activation Function

The Rectified Linear Unit (ReLU) function is easy to compute, does not saturate the region, and does not cause the Vanishing Gradient Problem.

### Data Observations

ReLU Activation Function



| Hidden Layer, Alpha | Acc_tr | Acc_ts |
|---|---|---|
| layers=22, alpha=0.01 | 0.995644 | 0.974041 |
| alpha=0.001 | 0.997096 | 0.971783 |
| alpha=0.0001 | 0.996612 | 0.974041 |
| alpha=1e-05 | 0.996128 | 0.974041 |
| layers=(22, 22), alpha=0.01 | 1 | 0.96614 |
| alpha=0.001 | 1 | 0.970655 |
| alpha=0.0001 | 0.998548 | 0.963883 |
| alpha=1e-05 | 0.999516 | 0.972912 |
| layers=(22, 22, 22), alpha=0.01 | 1 | 0.969526 |
| alpha=0.001 | 0.999516 | 0.970655 |
| alpha=0.0001 | 1 | 0.971783 |
| alpha=1e-05 | 1 | 0.975169 |
| layers=30, alpha=0.01 | 0.998064 | 0.972912 |
| alpha=0.001 | 0.99758 | 0.977427 |
| alpha=0.0001 | 0.998064 | 0.976298 |
| alpha=1e-05 | 0.99758 | 0.974041 |

*Table: Accuracies for NN with ReLU Activation Function*

| layers=22 | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.994854 | 1 | 0.996564 | 1 | 0.995993 | 1 | 0.995423 | 1 |
| recall | 1 | 0.972393 | 1 | 0.981595 | 1 | 0.978528 | 1 | 0.97546 |
| f1-score | 0.99742 | 0.986003 | 0.998279 | 0.990712 | 0.997993 | 0.989147 | 0.997706 | 0.987578 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=(22, 22) | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 1 | 1 | 1 | 1 | 0.998851 | 0.996923 | 0.999426 | 1 |
| recall | 1 | 1 | 1 | 1 | 0.999425 | 0.993865 | 1 | 0.996933 |
| f1-score | 1 | 1 | 1 | 1 | 0.999138 | 0.995392 | 0.999713 | 0.998464 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=(22, 22, 22) | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 1 | 1 | 0.999426 | 1 | 1 | 1 | 1 | 1 |
| recall | 1 | 1 | 1 | 0.996933 | 1 | 1 | 1 | 1 |
| f1-score | 1 | 1 | 0.999713 | 0.998464 | 1 | 1 | 1 | 1 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |
| layers=30 | alpha=0.01 | | alpha=0.001 | | alpha=0.0001 | | alpha=0.00001 | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| precision | 0.997706 | 1 | 0.997135 | 1 | 0.997706 | 1 | 0.997135 | 1 |
| recall | 1 | 0.98773 | 1 | 0.984663 | 1 | 0.98773 | 1 | 0.984663 |
| f1-score | 0.998852 | 0.993827 | 0.998565 | 0.992272 | 0.998852 | 0.993827 | 0.998565 | 0.992272 |
| support | 1740 | 326 | 1740 | 326 | 1740 | 326 | 1740 | 326 |

*Table: Precision/Recall/F1-Score for NN with ReLU Activation Function*

The training set accuracy was approximately 1 and test set accuracy was in the general range of 0.96 to 0.97, for all the different iterations of the model. Looking at the Precision-Recall Curve, the highest AP score of 0.96 is for the following iterations:

Hidden Layers: 22, Alphas: 0.001
Hidden Layers: 22, Alphas: 0.00001

From the Loss curve we know that the smallest average cost is for the iteration:

Hidden Layers: 22, Alphas: 0.00001

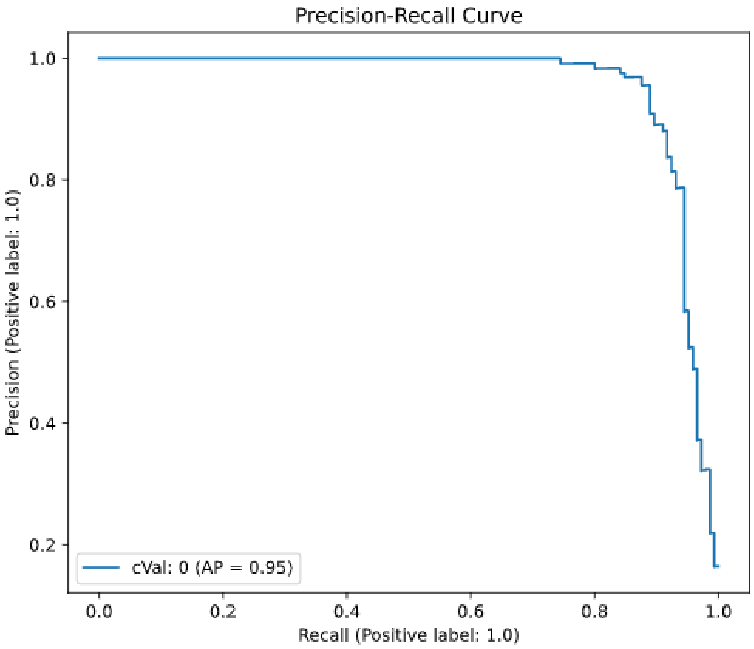
It is important to note that the ReLU function is not zero centered and has the 'dying ReLU problem', which means that any nodes receiving a negative input will have a zero output causing the node to die and not learn anything.

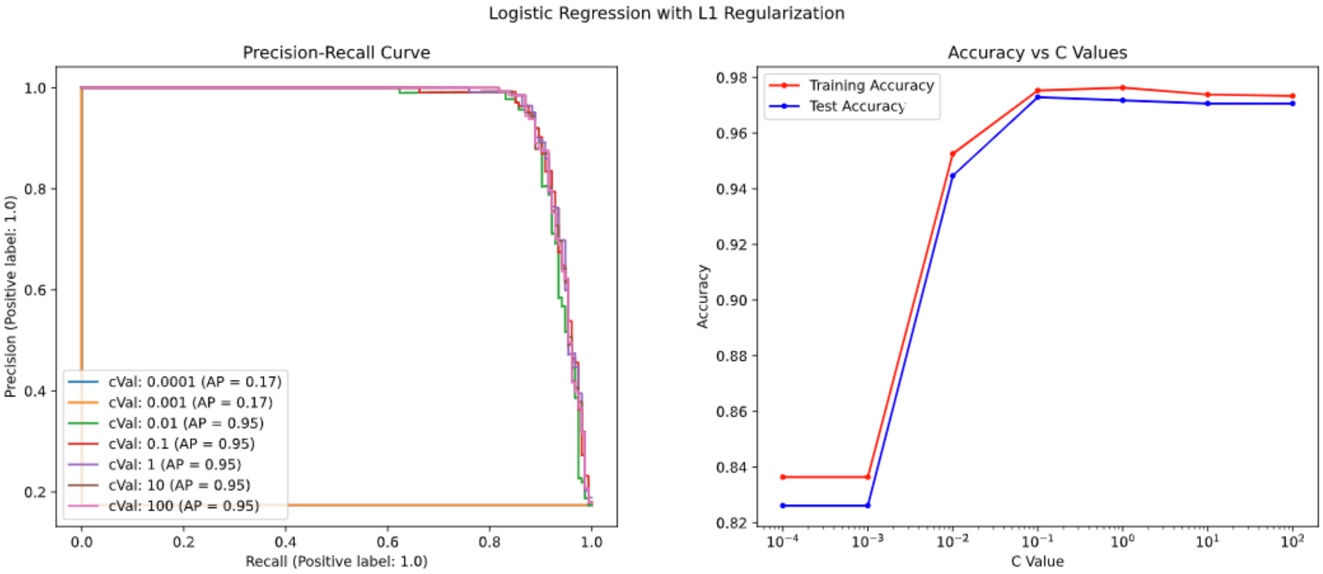
## Summary of Results

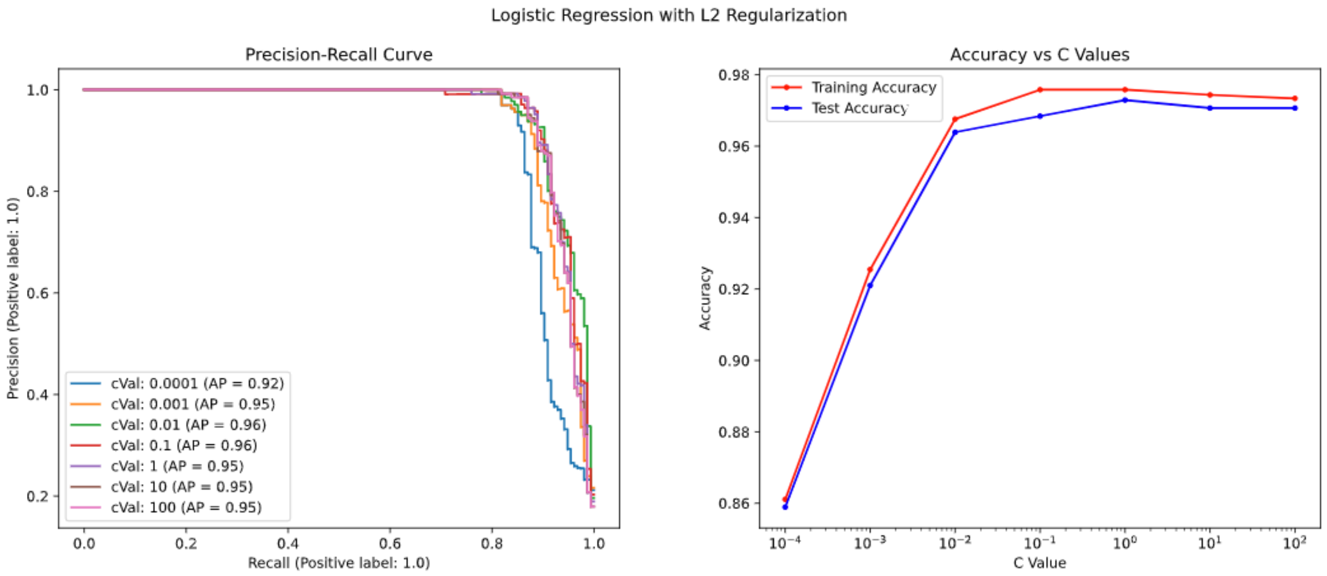
Overall, all three activation functions performed very well in terms of precision, recall, and f1-score as their scores were high and approximately or equal to 1 for all the different types of iterations. However the ReLU activation function did exceptionally well in terms of consistency regarding the scores and its loss function also has lower average costs for the different types of iterations.

# Graphs

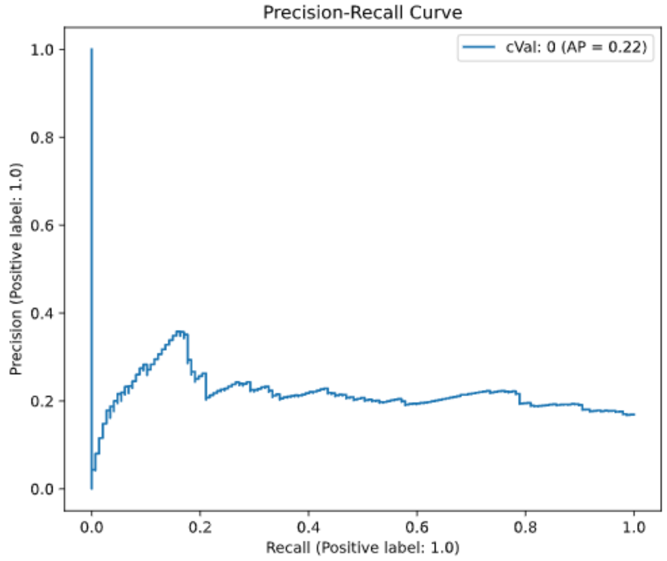
### Logistic Regression with StandardScaler

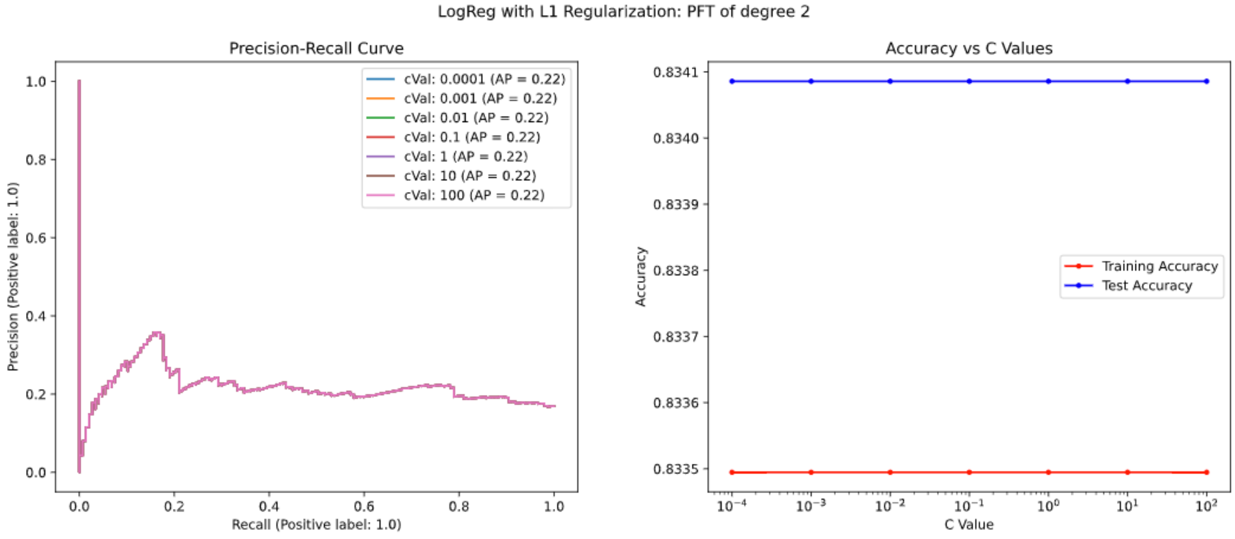


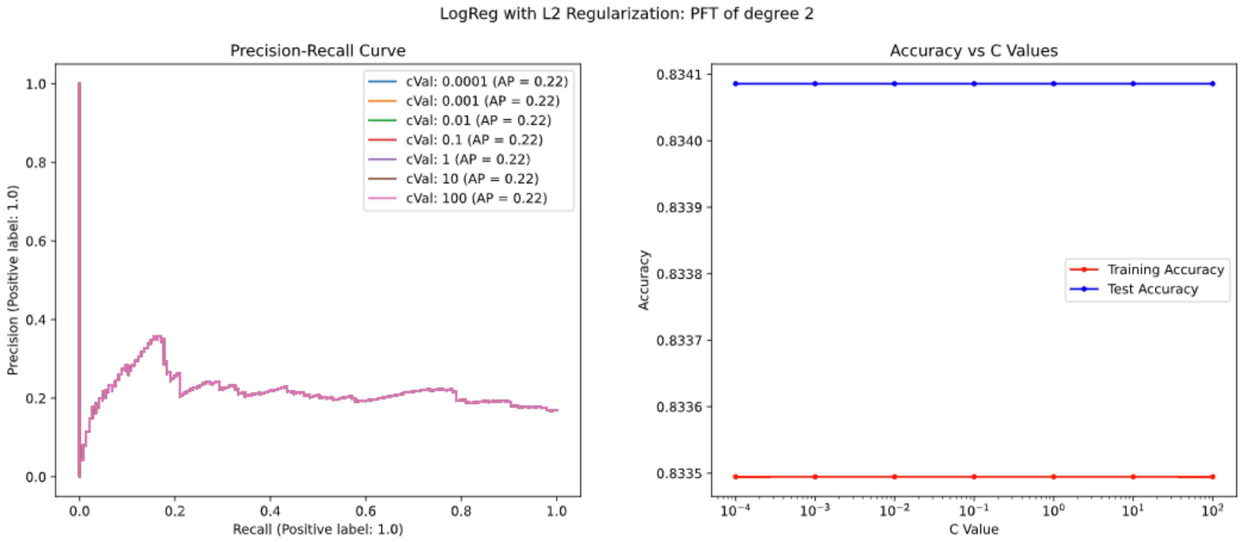




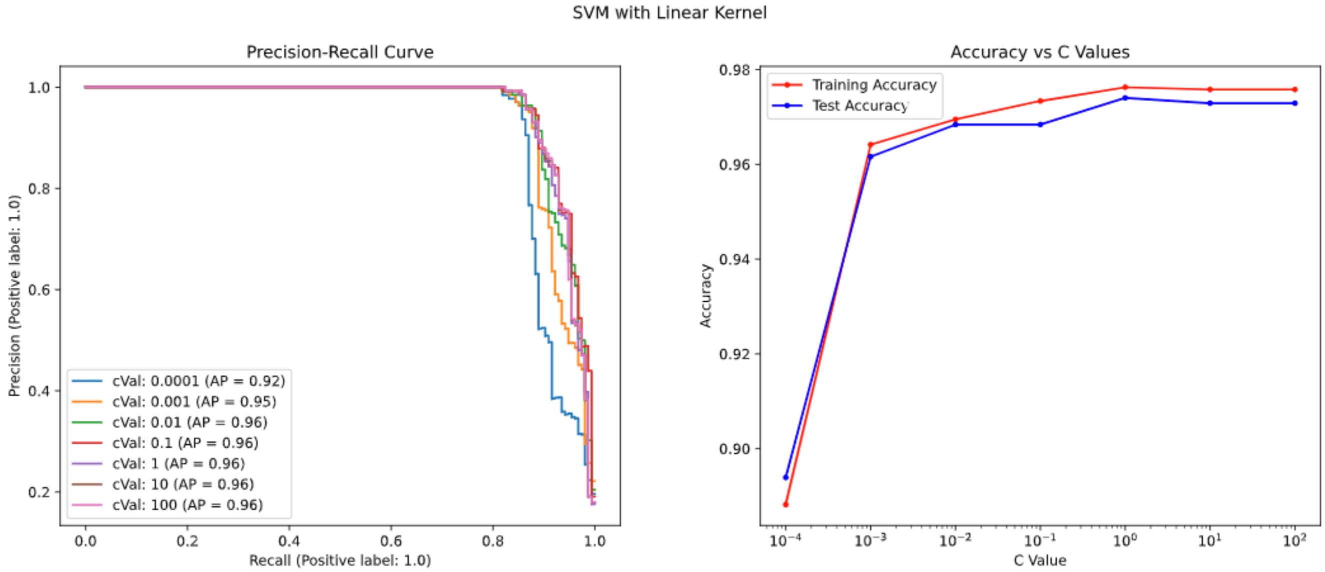
## Logistic Regression with Polynomial Feature Transformation

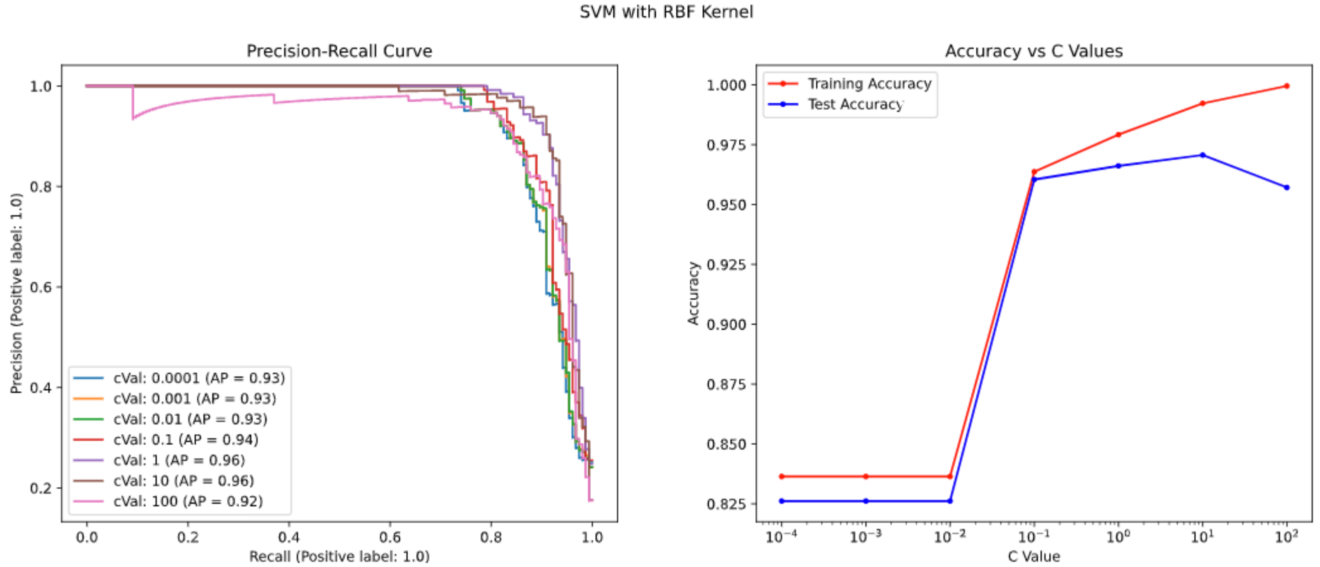


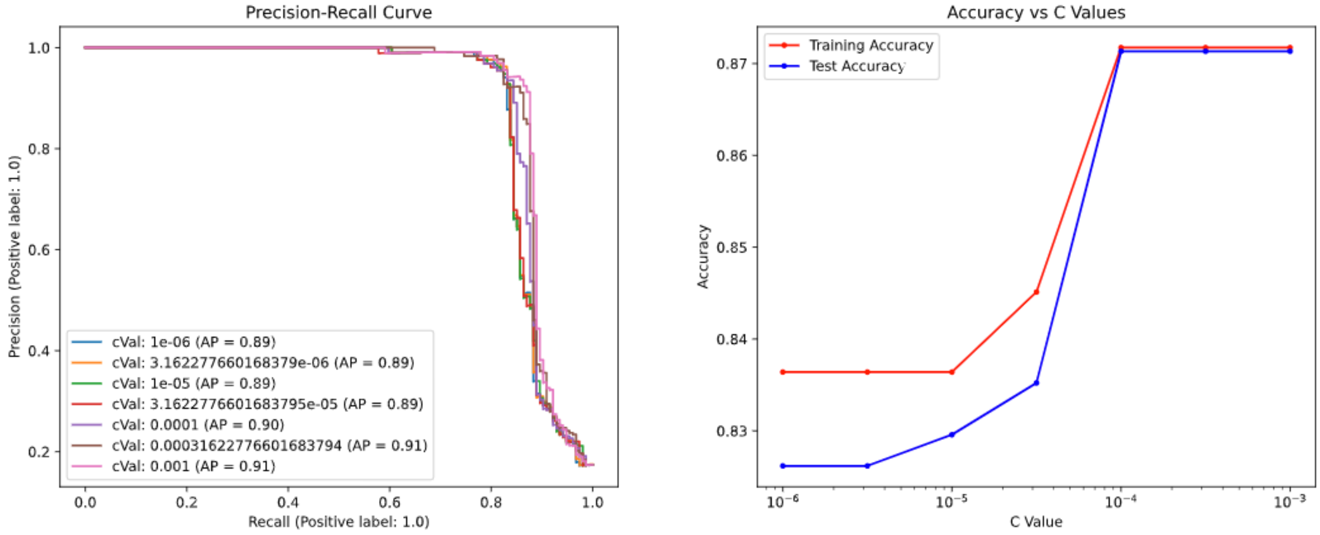




# Support Vector Machines

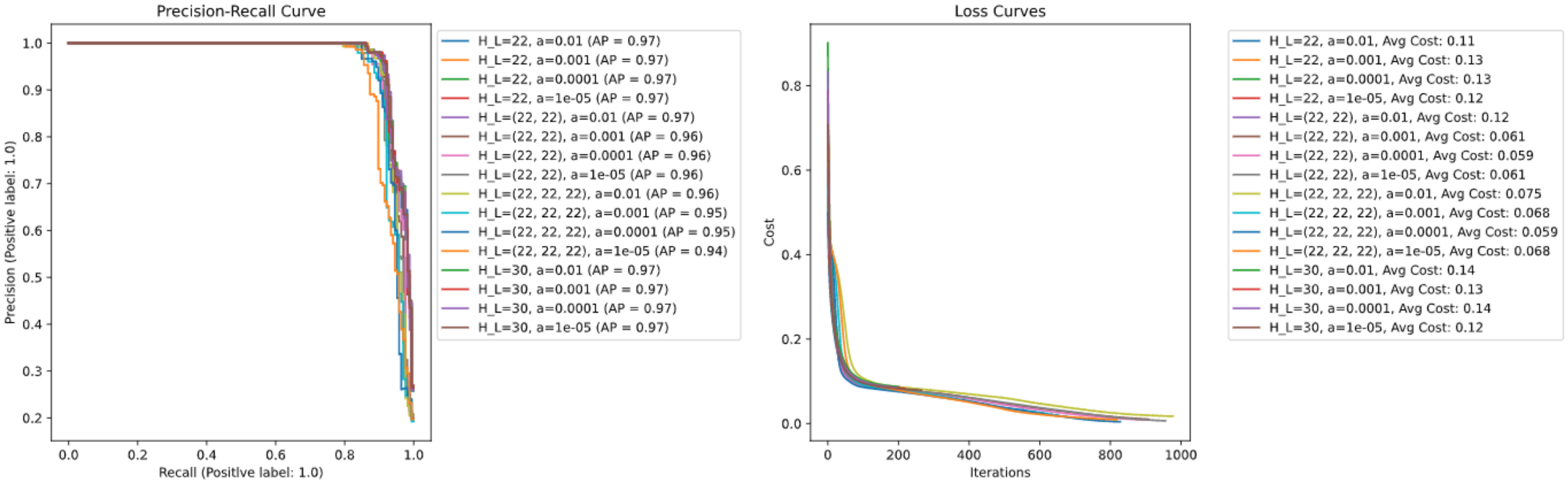




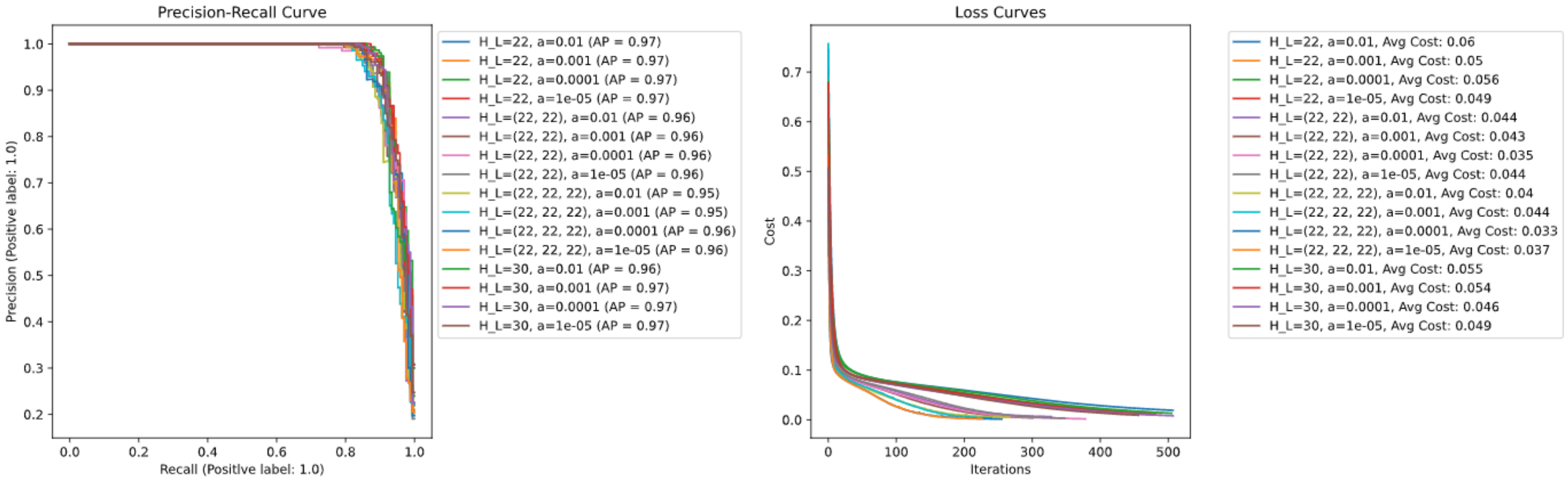


SVM with Polynomial Kernel

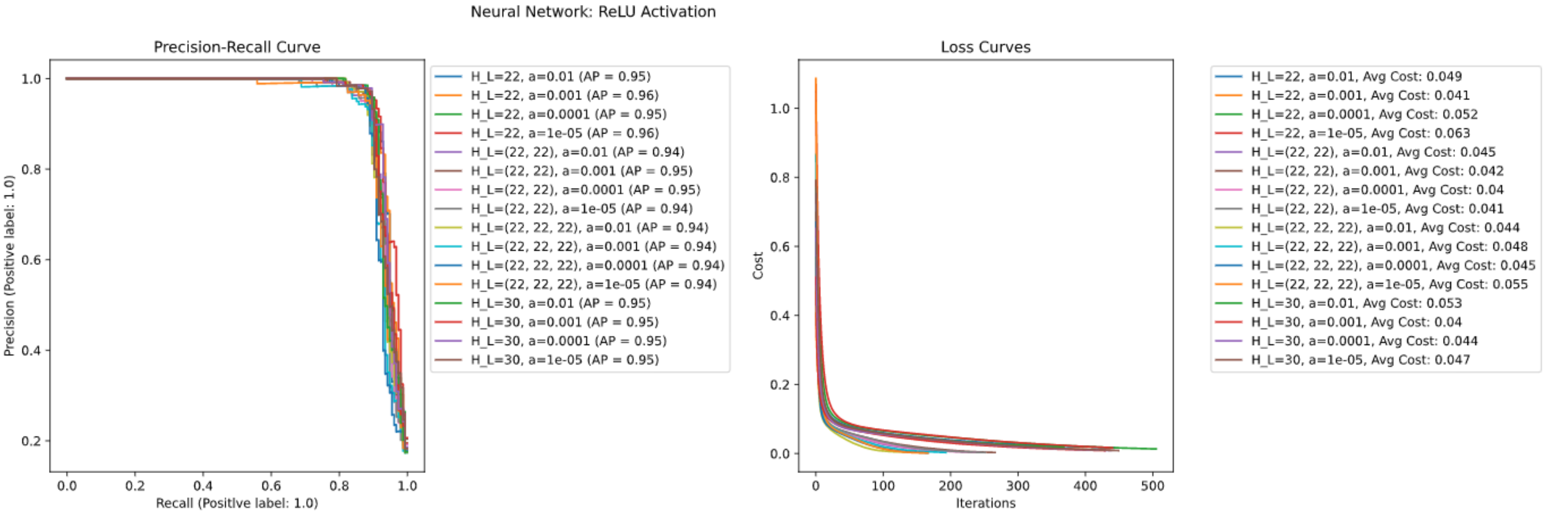
# Neural Networks



Neural Network: Logistic/Sigmoid Activation



Neural Network: tanh Activation



# Conclusion

It is important to remember that the data we are drawing conclusions from in this report is unbalanced, so there will be a higher bias towards Class 0 and the lesser number of Class 1 transactions will tune our models to overfit to those examples. An obvious indication that our models have a case of overfitting are the Accuracy vs C graphs created for the first two model types. For logistic regression using StandardScaler data and SVMs, we see that the test data learns the testing data model very quickly and after having found a good fit, the learning eventually stagnates and even decreases a little. We also see that the training set has a slightly higher accuracy trajectory than the test set, which is another indication of overfitting.

Another issue to note is that the numbers in the data are also extremely close and often differ by ± 0.0001. For the supervised learning models, we made specific observations from the data and used that to help guide us towards one or another specific iteration of the model. As such, drawing a final conclusion from such specificity would be an example of observational overfitting. Therefore, in order to make an educated choice of model, we looked at the metrics in a more holistic way.

As discussed earlier in the report, the high variance from overfitting does not allow accuracies to be a good analytic metric even though all three models showed that the accuracy of the training and test set were always very high, often reaching 97%-99% accuracy. On the other hand, precision, recall, and the f1-scores that were derived from the two metrics, were better indicators of predictive accuracy as we hone in on Class 1 fraud cases. Being able to correctly detect fraud cases is our main goal, so anything that can show us how well our model can do that is a valid measurement. In any model, we are looking for high precision, meaning most of our predicted fraud transactions are true, and high recall, meaning most of our fraud cases are predicted.

After performing supervised learning on the logistic regression models using the standardized and polynomial transformed data, we can determine that the use of the polynomial feature transformation has no positive effect on the learning of the logistic regression model. After performing supervised learning on the support vector machine models using the standardized data, we notice similar results to how the logistic regression models performed between using the standardized data and the polynomial feature transformed data. The SVM using the polynomial kernel exhibited little variation in the accuracies as the C value increased as well as poor f1-scores for Class 1 identification. This behavior mimicked that of the logistic regression model using polynomial feature transformed data.

Among the three model types, neural networks showed the most promising results in terms of precision, recall, and f1-score. The scores were high, consistent, and approximately/equal to 1 for different types of iterations, hidden layer sizes, and regularization terms. This type of accuracy was not seen as prominent with logistic regression or SVMs. Further, the ReLU activation function showed a lower average cost, from the loss curve graph, and overall high AP score in the precision-recall curve. In terms of overfitting, we know that a less complex model for neural networks tends to perform better than having multiple hidden layers. We also know that adding a regularization, the alpha term, allows us to penalize large weights by adding a cost to the loss function. Unfortunately, with how close the data points are, these reasonings are a bit difficult to actually observe. However, from our understanding, testing, and observations, we believe that a neural network model with the ReLU activation function and having two to three hidden layers with the number of neurons being ⅔ of the input layer plus the number of output neurons (2), and a relatively small alpha (L2 regularization parameter), will be able to best detect fraud credit card transactions.

In order to improve upon our existing models, there are several different approaches that we could take. For this project, we used undersampling to reduce the size of our dataset and the bias towards Class 0. We could try other ways of reducing overfitting by creating a balanced dataset or oversampling Class 1 cases to determine if there is any significant improvement in prediction. We could also attempt to use different ratios for splitting the training and test data, instead of the 70/30 split. In addition, the high probability of overfitting in credit card fraud detection warrants a need for more regularization in the set and using a much larger range of C values could help tune different models better. In addition, we could try to use cross validation for all models to reduce error and improve model performance. While this method will not remove overfitting, it can help in identifying overfitting, allowing us to better combat the issue.

For supervised learning, there are also improvements that could be made specific to each model. For logistic regression, the data for with and without regularization were very close to one another, so testing logistic regression without regularization with a wider range of C values may help in determining whether there really is a significance of regularization to the model or not. For SVMs, we did not perform any additional data preprocessing as the data had already been transformed with PCA. In the future, we could try a different type of transformation, such as normalizing, to see if that helps improve the model. Lastly, for neural networks, we could focus on varying the number of neurons in the hidden layers or even trying to use dropout layers, which randomly set output features of a layer to zero and delays the effect of overfitting.