Introduction

Credit card fraud is a serious problem in today's digital world. With the increasing use of credit cards and the ease with which they can be used to make purchases, there is a greater risk for fraud.

The dataset that I used was found on Kaggle and is called Credit Card Fraud Detection. Simply, the dataset shows data from fraudulent and non-fraudulent transactions made by credit cards in September 2013 by European cardholders.

The data itself has features that already are processed PCA transformations. And due to confidentiality, each of the headers of the features is named V1 — V29. Only Amount, which describes the amount transacted, and Time, describing the time taken for a transaction to complete are non-anonymous. Additionally, there is a Class label that is one-hot encoded to indicate non-fraudulent (0), and fraudulent (1), which indicates that this is a binary classification dataset.

Preprocessing

I first scaled and preprocessed time and amount to fit the other values in my dataset. I did so by implementing a RobustScaler() and then inserted them both into the data frame. As of then, I had inserted a scaled_amt and scaled_time features because they were very important features that I knew of that were not anonymous. This data frame had 284807 rows × 31 columns. This was the original data frame I name df_org.

However, The dataset did not have an even number of fraudulent and non-fraudulent cases. There was only 0.17% (492 cases) of fraud cases out of more than 200-thousand non-fraudulent cases. Unbalanced datasets could fail under traditional algorithms biased towards the majority class.

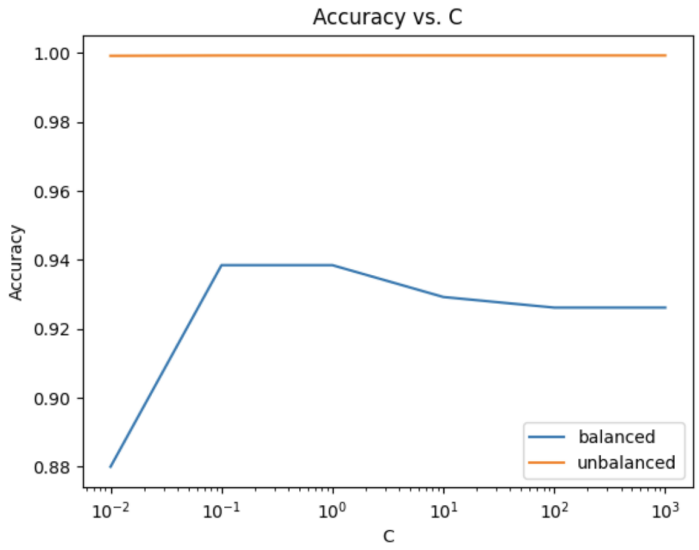
And so I evened them out by randomly selecting from non-fraudulent cases, and concatenated them into the fraudulent cases. Now I had a random selection of non-fraudulent and fraudulent data frames which had dimensions of 984 rows × 31 columns, with 492 in each classification. I named this data frame df1.

I then performed a train_test_split on both df1 and df_org. I wanted to test my SVM, neural network, and logistic regression on both a balanced and unbalanced dataset.

Logistic Regression

The independent variable I wanted to test was by varying the values of the c parameter with the values c = [0.01, 0.1, 1, 10, 100, 1000]. I also incorporated l2 regularization and L1 regularization. I then tested all of this against the balanced dataset vs the unbalanced dataset.

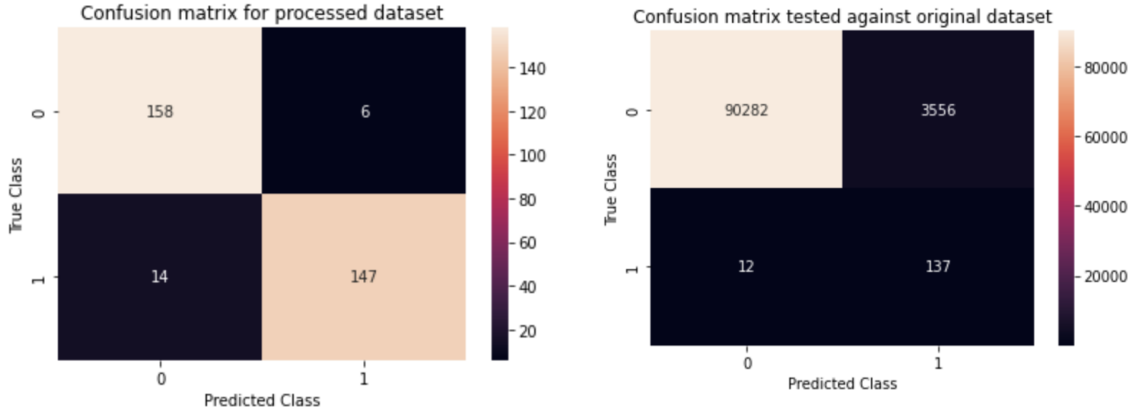
In my implementation, both L1 and L2 regularization happened to have the same result. This is probably due to the scarcity of fraudulent data, which only accounts for 0.17% in the unbalanced dataset, and merely 492 training examples. This is what led me to believe that the data is very easily logistically classifiable. For this reason, I have only graphed balanced and unbalanced accuracy scores on the Accuracy vs C graph as shown below.



Although the unbalanced dataset had accuracies that looked like it did not change in the graph, the change was actually very little across c-values as shown in this code snippet.

```
original accuracy:  [0.9991700979922755, 0.9992658559162437, 0.9992658559162437, 0.9992764956855735, 0.9992764956855735, 0.9992764956855735]
```

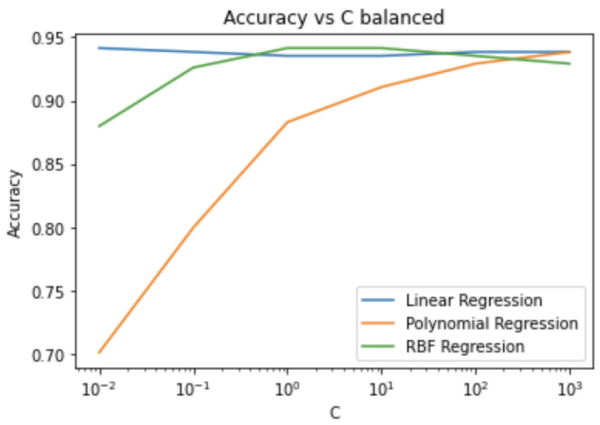
The following two figures are confusion matrices for the balanced and unbalanced datasets respectively.

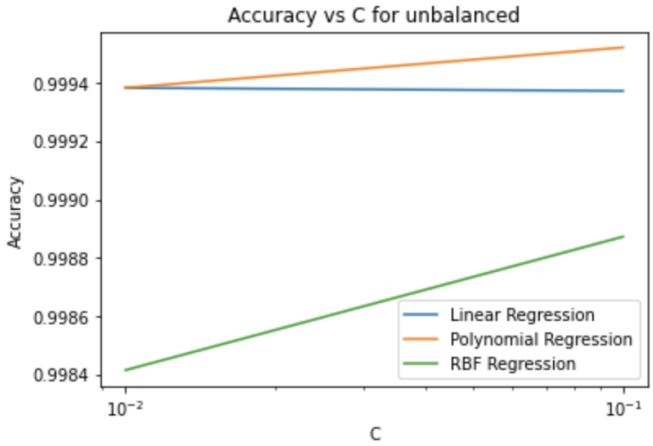


Support Vector Machines (SVM)

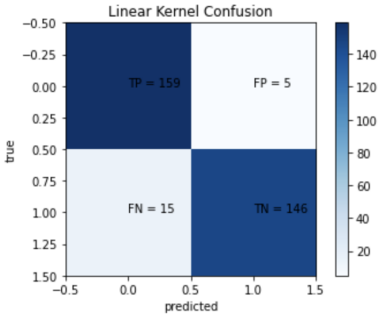
Using both the unbalanced and balanced data, I used the SVM model to further analyze the dataset. For the SVM I tested against 3 different kernels, Linear, Poly, and Radial. I also varied the C values that represented the lambda inverse of the values c = [0.01, 0.1, 1, 10, 100,

1000]. These values were only used for the balanced dataset, and not the original dataset because it took too long to run the model. Instead, I opted for c = [0.01, 0.1] instead. The following graph describes the Accuracy vs C for all the different types of kernels. I then plotted them in confusion matrices as shown below for further analysis.

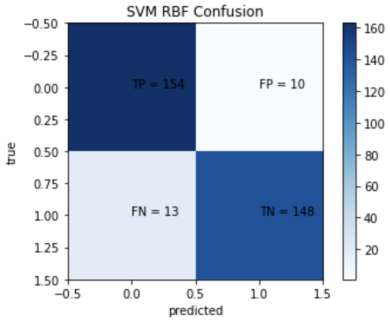




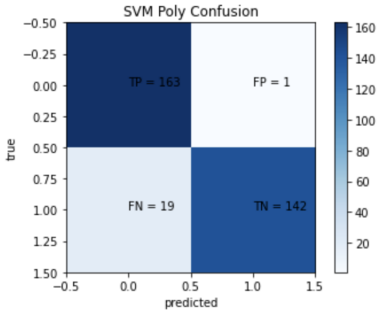
precision 0.9403402603334095
recall 0.9381722466292985
f-score 0.9383629191321499



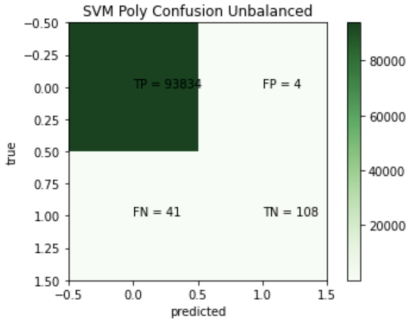
precision 0.9294322746911241
recall 0.9291395243144978
f-score 0.9292066408432696



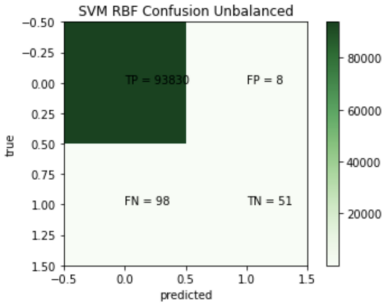
precision 0.9443056943056943
recall 0.9379450083320708
f-score 0.9382035290538485



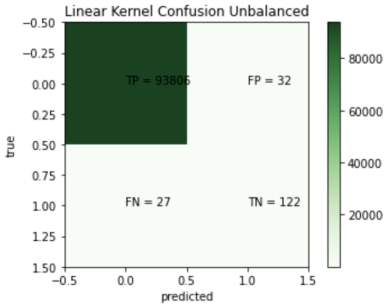
precision 0.9819244816435229
recall 0.8623947940553268
f-score 0.913673239613592



precision 0.9316817136530109
recall 0.6710983129428684
f-score 0.7449100414673255



precision 0.8959600234791265
recall 0.9092254665365743
f-score 0.9024830740503276



Neural Networks

For Neural Networks, the independent variable that I was testing against was the different activation functions. I tested against ReLU, Tanh, and the Sigmoid activation functions. Each of these functions was placed at the input, output, and hidden layer nodes. Then I ran the model 10 times and cumulated the average accuracy for each of the activation functions.

I also tested against different sizes of the input and hidden layers as shown in this list: [2, 10, 20, 50, 100, 500]. However, there yielded no clear result. The following print() statement shows the respective accuracies on the balanced dataset.

```
accuracy: 0.5046
accuracy: 0.9015
accuracy: 0.9262
accuracy: 0.9354
accuracy: 0.5046
accuracy: 0.9323
```

The figure on the left is tested against the balanced dataset, whereas the figure on the right is tested against the unbalanced dataset.

Activation Function	Average Accuracy
ReLU	0.7101538419723511
Tanh	0.8846153914928436
Sigmoid	0.8815384566783905

Activation Function	Average Accuracy
ReLU	0.9994573593139648
Tanh	0.9994893074035645
Sigmoid	0.9994467496871948

Table of Results

unbalanced NN			
	relu	**tanh**	**sigmoid**
	0.9994573593	0.9994893074	0.9994467497

balanced NN			
	relu	**tanh**	**sigmoid**
	0.7101538	0.88615	0.881538

Logistic Regression	**0.01**	**0.1**	**1**	**10**	**100**	**1000**
balanced	0.88	0.9384615385	0.9384615385	0.9292307692	0.9261538462	0.9261538462
unbalanced	0.999170098	0.9992658559	0.9992658559	0.9992764957	0.9992764957	0.9992764957
balanced L2	0.88	0.9384615385	0.9384615385	0.9292307692	0.9261538462	0.9261538462
unbalanced L2	0.999170098	0.9992658559	0.9992658559	0.9992764957	0.9992764957	0.9992764957

SVM balanced	**0.01**	**0.1**	**1**	**10**	**100**	**1000**
Linear	0.9415384615	0.9384615385	0.9353846154	0.9353846154	0.9384615385	0.9384615385
Poly	0.88	0.9261538462	0.9415384615	0.9415384615	0.9353846154	0.9292307692
RBF	0.7015384615	0.8	0.8830769231	0.9107692308	0.9292307692	0.9384615385

SVM balanced	**POLY**	**LINEAR**	**RBF**
Precision	0.9443056943	0.9403402603	0.9294322747
Recall	0.9379450083	0.9381722466	0.9291395243
f-score	0.9382035291	0.9383629191	0.9292066408

SVM unbalanced	**0.01**	**0.1**
Linear	0.9993828934	0.9993722536
Poly	0.9993828934	0.9995212104
RBF	0.9984146744	0.9988721845

SVM unbalanced	**POLY**	**LINEAR**	**RBF**
Precision	0.9819244816	0.8959600235	0.9316817137
Recall	0.8623947941	0.9092254665	0.6710983129
f-score	0.9136732396	0.9024830741	0.7449100415

Conclusion

For most of the data that I tested on, it made more sense to test it on the balanced data because that would allow the model to test on more realistic scenarios, in which the algorithms are not as biased as compared to the unbalanced dataset.

And through my findings, all 3 models have further confirmed that to be the case for unbalanced test sets. The accuracies for most of my tests against the unbalanced set yielded very good results, of which most were in the 98% and above.

Other than an unbalanced and balanced set, there are several other things that I think the dataset has problems with that could have yielded very unusually high accuracy scores. It could have been that there was not enough data to test, hence the fraud cases which acted as the bottleneck could mean the data was heavily biased. More shuffling of the dataset could be beneficial, or perhaps implementing k-folds to offset the strong bias that resulted from high accuracies (as I will further explain below). Perhaps implementing outlier removal could have improved the results as well, however that only becomes feasible in the amount and time columns as all previous columns have been PCA'd already. The problem mostly comes in the smallness and bias of the dataset, which I will explain with each model.

Firstly for Logistic Regression, graphing the accuracies vs the C-values showed that, in general, as the c-values increased, the accuracy increased. The most optimum point is at a c-value of 0.1. Afterward, increasing the c-values decreased the overall accuracy of the regression model. This could be due to using a smaller value for c if the data set is large and/or if the data set is particularly noisy. Whereas using a larger value for c if the data set is small and/or if the desired output is very sensitive to small changes in input. Furthermore, using a larger c value may help prevent overfitting the data. Therefore, as seen in the data, when increasing for a larger c-value it shows that our desired output for c-values beyond 0.1 that yielded decreasing accuracies shows that the dataset itself means that the data is simple. There is no need to increase the c-value beyond what we think is overfitted. Hence, the dataset is implied that the separation between fraud and non-fraud cases is relatively easy to distinguish.

Secondly, the SVM accuracies further showed why the dataset could have been very simple in distinguishing between the classifications. The linear SVM was always most accurate across different c-values, whereas the polyfit and the radial fit did not perform as well, albeit the radial inched above the linear fit for one of the middle c-values, I would not be completely confident in saying that a radial would have been needed as it only performed slightly better, and did poorly against the unbalanced dataset.

Lastly for neural networks, it was difficult to piece together which parameters served importance for creating better accuracies. Overall, the tanh activation function performed significantly better than the ReLU for the balanced dataset, and still outperformed the others in the unbalanced dataset. The reason for this could be because the tanh is not only differentiable, but has a range from -1 to 1, and is also a faster function than sigmoid.