Neural Computing Report

FRAUD DETECTION CLASSIFICATION: MLP VS SVM

I. BRIEF & MOTIVATION

Today, internet has seen business and wealth digitalised, making it easier for fraudsters to operate at a large scale. Fraud has become a huge concern and huge efforts are being placed into detecting and preventing it. Previously fraud detection methods were manual, rigid, and rule-based that were implemented ahead of time. Eventually it became a race between fraudsters, who would devise new ways to defraud and the detection systems figuring out these new methods. The modern alternative is to take advantage of big data and machine learning. To create models that are always running, always receiving new data, and always learning. This ongoing cycle will help create a dynamic and evolving model that will be able to spot new changes. Once the algorithm is exposed to huge amount of transactions as a fraud detection system, it can use supervised learning to be able to detect and predict fraudulent transactions.

Our aim is to build and critically evaluate two supervised machine learning algorithms in their ability to detect fraudulent transactions with the hope they can be adopted as fraud detections systems to help detect and prevent future fraudulent transactions. The first algorithm we will be looking at is a Multi-Layer Perceptron (MLP), specifically a Deep Neural Network. The second algorithm we will be looking at is called Support vector machine (SVM). The report will start with the exploring the dataset, describing it and the results of the exploratory analysis. We will then start with describing the two models by looking at the pros and cons of each. The next step is to formulate the hypotheses on the two models will compare. We will then look to understand the methodology and the experimental results of each model. In the end we will evaluate our findings and learnings from and how they compared to our hypotheses.

II. DATASET DESCRIPTION

There is a lack of transactional data available in the public domain, as a result, we will be using a synthetic dataset called “Synthetic Financial Datasets for Fraud Detection” from Kaggle. The dataset is created using a PaySim simulator, which aggregates private data ensuring it’s like the original but with fraudulent transactions inserted to test detection. The original data used is from money mobile transactions in an African country, which was provided by a multinational corporation. An example of the dataset and description are seen below in figure 1 and 2.

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	0.0	0.0	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	0.0	0.0	0	0
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	0.0	0.0	1	0
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	21182.0	0.0	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	0.0	0.0	0	0

Figure.1

```
step              int64
type             object
amount          float64
nameOrig         object
oldbalanceOrg   float64
newbalanceOrig  float64
nameDest         object
oldbalanceDest  float64
newbalanceDest  float64
isFraud           int64
isFlaggedFraud    int64
```

Figure.2

The next step is to understand our data, investigating any relationships or trend that can help us achieve our aim. As we can see we have eleven columns in our dataset with our target variable labelled as “isFraud”. Three columns are objects and the rest are numerical as integers and floats and are all discrete variables. Notably, “isFraud” & “isFlaggedFraud”

columns are binary with ones and zeros. We want to use the features of the transaction rather than who did the transaction take place between, hence, account codes will be dropped. We will also drop “isFlaggedFraud” as this is linked to automatically flagging any transfer over 200,000. We found out this doesn’t necessarily mean its fraudulent, in fact only 16 were out of over 8,000 fraudulent transactions. We will also convert the column “type” to a dummy variable. We have a healthy dataset with over six million rows, the important thing here is to check skewness and imbalance.

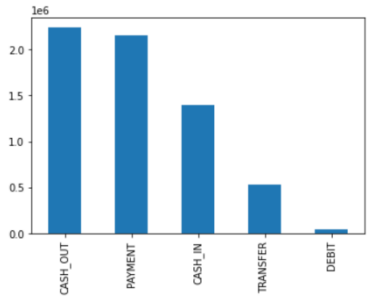


Figure.3

type	isFraud	
CASH_IN	0	1399284
CASH_OUT	0	2233384
	1	4116
DEBIT	0	41432
PAYMENT	0	2151495
TRANSFER	0	528812
	1	4097

Figure.4

Looking at figure.3, we can see column “type” is heavily skewed, with debt having the least number of samples. Looking at figure.4, we can see a clear case of class imbalance, with only “CASH_OUT” and “TRANSFER” values having fraudulent transactions, and at a disproportionate rate. Interestingly, if we look at the distribution of column “amount” we see a multimodal distribution as seen in figure.5. To investigate this further, we separated the data into fraudulent and non-fraudulent and then rerun the distribution.

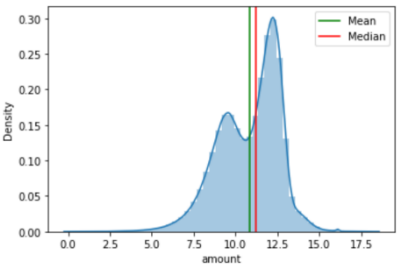


Figure.5

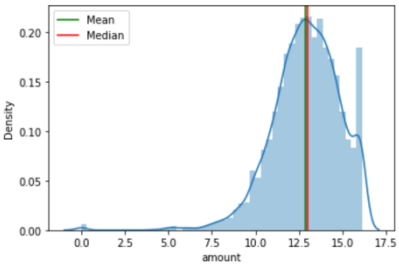


Figure.6

The distribution for fraudulent data is seen in figure.6. The results show the multimodal distribution seen earlier was indeed caused by the peaks from each fraudulent and non-fraudulent transactions. Meaning overall fraudulent transactions can be distinguished due to having higher mean. To address the class imbalance, we will adopt under sampling of the majority class, in this dataset it will be reducing the number of non-fraudulent transaction by the means of random sampling. This will help us significantly reduce the volume of the data, which will help with computation. We will avoid over sampling of the minority class as this mean creating more synthetic data. We decided to balance the data at type category level, this can be seen in figure.7.

type	isFraud	
CASH_IN	0	4097
CASH_OUT	0	4097
	1	4097
DEBIT	0	4097
PAYMENT	0	4097
TRANSFER	0	4097
	1	4097

Figure.7

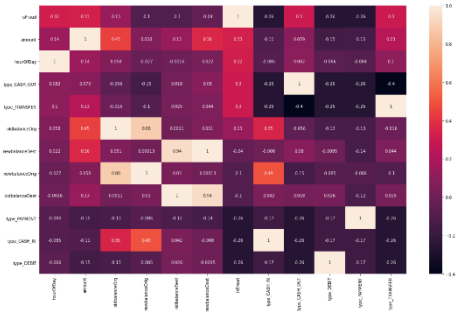


Figure.8

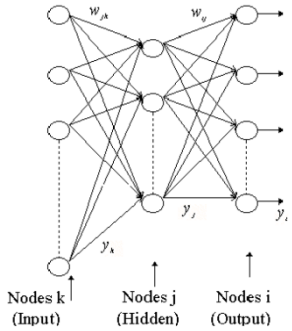
We also run a correlation matrix to ensure there are no highly correlated attributes with our target variable as seen in figure.8. The last step is to scale our data to ensure all comunns are at an equal ratio.

III. SUMMARY OF MODELS

A. Multi-Layer Perceptron (MLP)

The perceptron is a simple artificial neural network architecture, first introduced in the late 50s by Frank Rosenblatt. It's based on artificial neurons called threshold logic unit (TLU) or a linear threshold unit (LTU). The neurons are connected and have a weight associated with them. The TLU then computes the weighted sum of all the inputs and applies a step function to produce the results. MLP is just an extension of this by having one or more layers or TLUs called hidden layers with an input layer and an output layer. All the neurons are fully connected, and every layer has a bias neuron except the output layer. If MLP has more than two hidden layers its then referred to as a deep neural network (DNN). In the mid-80s Rumelhart introduced backpropagation training algorithm using gradient decent optimisation. It consists of an iterative process where the first step of the algorithm is known as the forward pass where the output of each neuron is calculated for each layer with respect to the target value. The next step is to calculate the error and then steps through the network in reverse to calculate the contributions made to the error. The final step is known as the gradient decent step where algorithm adjusts the weights to reduce the overall error. MLP can be used for regression or classification. The figure below on the right shows the architecture of a typical MLP [2] and the table below on the left are the pros and cons.

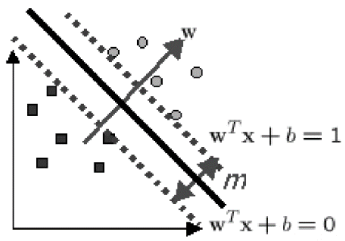
PROS	CONS
Capability - One hidden layer can be sufficient to deal with complex functions given enough neurons	Flexibility - Too many hyperparameter to adjust
Adaptability - Converges faster to good results with more than one hidden layer and generalises better on new data.	Resources - Requires a huge amount of training data.
Transfer Learning – Reuse parts of pretrained models, which train faster and use less data.	Prone to overfitting due to its complexity.
Overfitting can be dealt with different types of regularisation.	Very expensive to compute.



B. Support Vector Machine (SVM)

The SVM is a powerful machine learning model that is capable of preforming both linear and non-linear classification and regression. The algorithm looks to classify using decision boundaries known as hyperplanes and maximising the distance between the two boundaries as seen in the figure below on the right [2]. The decision boundaries are determined by instances called the support vectors. SVM is designed for binary classification but can be adapted for multi-class using the methods such as one-vs-one and one-vs-all. The width of the hyperplane can be adjusted so all instances are on the right side of the decision boundary, known as hard-margin classification. Which works if the data is linear separable but will be very sensitive to outliers. The aim is to find a right balance between keeping the hyperplane as wide as possible and limiting the number of violations. If violations are found, then this is known as soft margin classification. In terms of nonlinear classification SVM adopts polynomial features, with high polynomial degree a huge number of features are created. To overcome this, a kernel trick is applied which allows for the same results but without adding any new features. The trick maps the data into a 3-D space to find a decision surface to divide the different classes then returns the data back to a 2-D space. We will look at both linear (linear) and nonlinear (Polynomial & Gaussian RBF) kernels. Below are the pros and cons.

PROS	CONS
Suitable for small - medium datasets.	Sensitve to featrure scales.
Overfitting can be solved using regularisation.	Finding the optimal hyperplane.
High generalisation performance without prior knowledge.	No built-in function for multi-class classification.
Can work with high dimensional data.	Kernels add complexity and computational time.



IV. HYPOTHESIS STATEMENT

Our aim is to compare the performance of MLP vs SVM in detecting fraudulent transactions by focusing on accuracy and loss.

- We expect both algorithms to have a high-end performance, bettering a random prediction. We predict the SVM model to potentially outperform the MLP.
- We expect the training loss to be lower on the MLP due to backpropagation.
- We expect difficulty in optimising the hyperparameters for MLP due to the amount available.
- We expect the MLP and SVM non-linear kernels to be much more computationally expensive, in the range of $C(m^2 * n)$ and $C(m^3 * n)$ in comparison to linear SVM which is $C(m * n)$.

V. TRAINING & EVALUATION METHODOLOGY

For evaluating the models, we have adopted to split the data at 80:20 split for training and testing. A 20% split for the test set is sufficient to test the accuracy of our model as our dataset is considered as a large dataset. We need to validate our optimising changes, but we cannot utilise our test data until testing begins. On the other hand, we cannot rely on the training data alone as the model can overfit and fail to generalise on new data. Traditionally, K-fold Cross-Validation can be used but we anticipate our models to be computationally expensive. As a result, we have decided to further split the training data into training and validation sets at a ratio of 75:25.

In terms of metrics, we will look at comparing the model using accuracy and each of their loss functions. For SVM we will look at the hinge loss and for MLP cross entropy loss. In terms of optimisation, we looked at tweaking hyperparameters for each model. For MLP, conducting a grid search would have been too computationally expensive so we conducted a random manual search of tweaking the number of neurons in each hidden layer, learning rate and the optimizer. As discussed, there are plenty more hyperparameters we could have changes such as number of hidden layers and activation function. RuLU will be the activation function we use as its faster to compute and does not suffer from vanishing or exploding gradient decent. For SVM, we attempted to run a grid search for optimising the hyperparameter but unfortunately it was too computationally expensive, so we decided to manually tweak the following hyperparameters: kernel, misclassification cost and gamma.

In terms of the process, for each epoch in the MLP we will train the model and validate it. The loss and accuracy function will be calculated and plotted to check for overfitting. As for the SVM model, we will train the model after each tweak, validate it, and test it. For each we will calculate the accuracy score and the loss function. We will produce a confusion matrix of the results to display true positives, false positives, false negative and true negatives.

VI. PARAMETERS & EXPERIMENTAL RESULTS

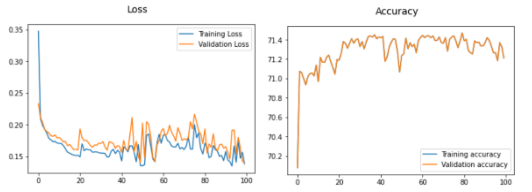
For the MLP, our best performing model achieved an accuracy of 72.2% on validation and 73% on testing, as seen from the table of results on the right. The architecture of model consisted of 11 neurons as an input layer, 8 neurons in the first hidden layer, 4 neurons in the second hidden layer and two neurons in the output layer. Batch normalisation was implemented at each layer and before the output layer a dropout function was used. For the activation functions, ReLU was used and for the optimiser, Adam was used with a learning rate of 0.001. We ran each altered model on 100 epochs to minimise computation and then plotted the results of each epoch to see the loss and accuracy as seen on the plots on the right.

MLP

Hyperparameter	Total Number of Neurons in NN					
Value	48		25		20	
Set	Val	Test	Val	Test	Val	Test
	70.7	72.1	72.2	73	70.4	70.8

Hyperparameter	Learning Rate					
Value	0.001		0.01		0.1	
Set	Val	Test	Val	Test	Val	Test
	72.2	73	70.8	72	70.8	72

Hyperparameter	Optimizer					
Value	SGD		Adam		Adagrad	
Set	Val	Test	Val	Test	Val	Test
	70.8	72	72.2	73	70.4	72



For the SVM, our best performing model achieved an accuracy of 98.7% on validation and 98.7% on testing, as seen from the table of results on the right. The model consisted of a poly kernel, C equal to 1 and gamma equal to 7. We trained the model, predicted on the validation and testing data to receive consistent accuracy and loss results. We then used a confusion matrix to analyse the extent of the accuracy as seen on the right. We attempted a grid search, but it was too computationally expensive and had to abandon it, as a result we manually tweaked the hyperparameters, trained, validated, tested the model then recorded the results. We also wished to display the decision boundaries for our best model by running PCA to reduce the input variables and run the model again to plot the poly boundary, but it was too computationally expensive.

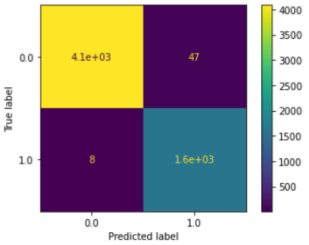
SVM

Hyperparameter		C				
Value		0.5		5		10
Set		Val	Test	Val	Test	Val	Test
Kernel	Linear	92.3	92.4	94.4	95	95	95

Hyperparameter		Gamma					
Value		1		3		7	
Set		Val	Test	Val	Test	Val	Test
Kernel	Poly	97.3	97.1	98.4	98.6	98.7	98.7

Hyperparameter		Gamma					
Value		1		5		10	
Set		Val	Test	Val	Test	Val	Test
Kernel	RBF	93.7	93.8	93.8	94.1	92.9	93

```
[[4085   47]
 [   8 1596]]
Test Accuracy: 98.73 %
Test Standard Deviation: 0.59 %
Test Loss 0.73
```



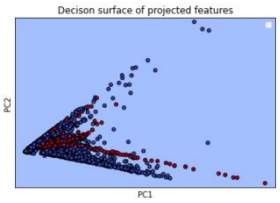
VII. ANALYSIS & CRITICAL EVALUATION

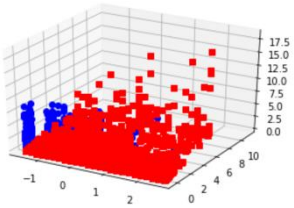
For the MLP model, our highest accuracy level was only 72%, we expected a much higher accuracy for this model due to its reputation as a neural network. In terms of architecture, we ensured we had two hidden layers for each tweak making it a deep network. We changed the number of neurons, we ensured it maintained a funnel like shape as this is known as a better architecture [1]. Between each layer we adopted batch normalisation to avoid the exploding gradient problem and bring consistency to the model. Before the output layer we adopted a dropout function which is known to even improve the accuracy of even the most state-of-the-art model by 1-2% [1]. In terms of parameters, changing the number of neurons didn't make any significant change in improving the accuracy of the model and neither did changing the optimizer. This backs up our hypothesis about the MLP being hard to optimise.

From the learning curves above, a clear drop in the loss value and a huge spike in accuracy initially means our learning rate was too high. We decreased it, but not much improvement was seen. We can see from the learning curves that our model does generalise very well on new data. If there is a huge difference in the loss curve for the training and validating data, this suggests overfitting but for us, the curves intertwine which is a good sign, this is reinforced by our accuracy score on the test data. The accuracy level seems to plateau, but the loss value is still slowly decreasing, which mean our model is still learning with every epoch. In relation to our hypothesis, the best MLP did only score a loss of 0.12 on the test

data in compassions to SVM model which recorded 0.73 which backs our hypothesis that MLP will have a lower loss rate due to backpropagation.

As for the SVM, our best model achieved an accuracy of 98.7% on the test data. This exceeded our expectation but as hypothesised the SVM model did outperform the MLP in terms of accuracy. We had to be aware as SVMs are prone to overfitting, so we ensured to run the model on a validation set, the results were reinforced by the test set. In terms of changing the hyperparameters, computing the linear kernel doesn't use too much computational power but once we use poly or RBF kernel, we complicate the model and the computational power intensives. As a result, we were unable to compute a gird-search and had to manually change our hyperparameters, this backs up our hypothesis that nonlinear kernels are computationally expensive. In terms of hyperparameters, we noticed the model improved as we increased C for the linear kernel. As a result, we decide to look at the separability of our data as seen on the right. It doesn't seem the two classes are linearly separable, but if we look at our data in a higher dimension using a poly kernel. We can see a clear distinction between the two classes in the second figure on the right-below. We can visually see why the poly kernel achieves such a high accuracy. This takes us back to our distribution of "value" which had a multimodal distribution which meant overall fraudulent transaction had a higher value. Overall, tunning hyperparameters for SVM were easy in terms of seeing noticeable differences in accuracy. This backs our hypothesis that only the MLP would be difficult.





VIII. CONCLUSION, LESSONS LEARNED AND FUTURE WORK

The report critically evaluated the accuracy of MLP VS SVM in classifying a binary dataset. The models were improved by tweaking hyperparameters by understanding the decision boundaries and their computation complexity. We discovered that the SVM model was significantly more accurate in comparison to the MLP, this could have been due to the sheer number of hyperparameters required to amend for the MLP. We conclude there is a trade-off between accuracy and computational power required, the more complex the model the more accurate it can be but at a cost of computation, this was evident for both models.
We also concluded that if we had more computational power, we could increase the number of epochs for MLP and conduct a grid search while implementing early stopping to find the optimal hyperparameters to improve accuracy instead of a sample manual search and similarly for the SVM model. Finally, the most damaging aspect of a fraud detection system is the false positive rate, these are undetected fraudulent transaction. Our best model achieved a false positive rate of 0.5%. Meaning, only 8 fraudulent transactions were not detected out of 1604.

IX. REFERENCES

[1] Aurelien, G., (2017). *Hands-On Machine Learning with Scikit-Learn & TensorFlow.* O'REILLY, (pp.107-167, 231-317)
[2] Frias-Martinez, E., Sanchez, A. and Velez, J., (2006). Support vector machines versus multi-layer perceptrons for efficient off-line signature recognition. *Engineering Applications of Artificial Intelligence*, 19(6), pp.693–704.
[3] Osowski, Stanislaw & Siwek, Krzysztof & Markiewicz, T., (2004). MLP and SVM networks - a comparative study. 46. 37- 40.
[4] Zanaty, E., (2012). Support Vector Machines (SVMs) versus Multilayer Perception (MLP) in data classification. Egyptian Informatics Journal. 13. 177–183.

[5] Collobert, R., Ch, C., Bengio, S., and Ch, B., (n.d.). *Links between Perceptrons, MLPs and SVMs*.

X. APPENDIX

A. Glossary

K-fold Cross-Validation – Statistical technique to test the validity of a model by resampling the data K number of times, where K are subsets called folds.

Cross Entropy Loss – A measure how well a set of estimated class probabilities match the target variables.

Hinge Loss – The function max(0,1,t) is called the hinge loss function. Its equal to 0 when t is greater or equal to 1. Its derivative is equal to -1 if t is less than 1 and 0 if t is greater than 1. Its not differentiable at t equals 1. You can use gradient decent with a sub derivative at t=1.

Early Stopping – A process of interrupting training of the model if the performance of the validation sets starts declining.

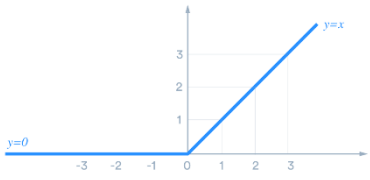
Dropout – For every training step, every neuron has the probability p of being temporarily dropped but will be functioning again for the next step. Drop out has parameter called dropout rate which is usually set to 50%.

Adam Optimiser– Which stands for “adaptive moment estimation”, which keeps track of an exponentially decaying average of past gradients and decaying average of past squared gradients.

Batch Normalisation – A technique of adding a layer before the activation function to simply zero-centre and normalise inputs, then scaling and shifting the results using two new parameters per layer.

Vanishing/Exploding Gradient Problem – Gradients get smaller towards lower levels as the algorithm process through the neural networks. Gradient Descent update leave the lower-level connection weights unchanged, and training doesn’t coverage to a good solution.

ReLU – Stands for Rectified linear unit, it computes a linear function of the inputs, and outputs the results if it is positive, and 0 otherwise. Defined as y=max(0,x), represented below.



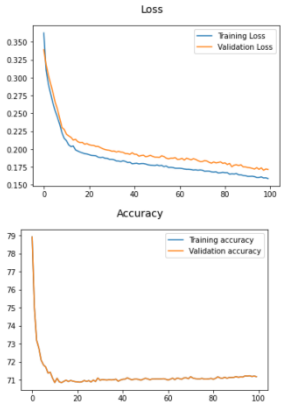
B. IMPLEMNATION DETAILS

For this section, we show the different results we received through implementing the manual hyperparameter tunning.

MLP Models:

```
# NN architecture 11-8-4-2
self.layer_1 = nn.Linear(11, 8)
self.layer_2 = nn.Linear(8, 4)
self.layer_out = nn.Linear(4, 2)
```

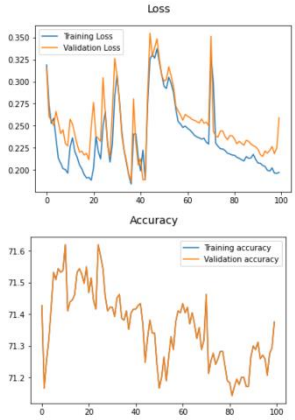
```
# variantion of optimizers used in maual grid search
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01)
```



epoch : 0
test_loss/accuracy:0.1634,72.0610

```
# NN architecture 11-8-4-2
self.layer_1 = nn.Linear(11, 8)
self.layer_2 = nn.Linear(8, 4)
self.layer_out = nn.Linear(4, 2)
```

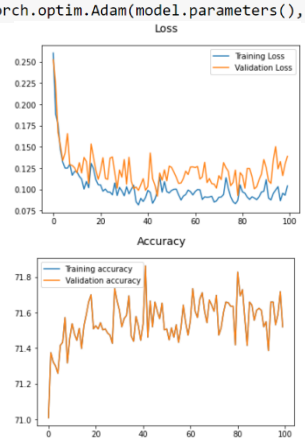
```
# optimizer with adjustable learning rate
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```



epoch : 0
test_loss/accuracy:0.2628,71.9216

```
# NN architecture 11-24-24-2
self.layer_1 = nn.Linear(11, 24)
self.layer_2 = nn.Linear(24, 24)
self.layer_out = nn.Linear(24, 2)
```

```
# optimizer with adjustable learning rate
optimizer=torch.optim.Adam(model.parameters(),lr=0.001)
```

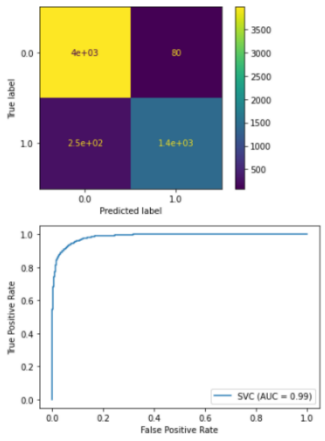


epoch : 0
test_loss/accuracy:0.1267,71.0157

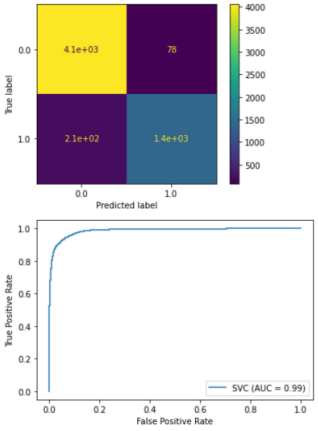
SVM Models:

```
# create SVM model
classifier = SVC(kernel = 'linear', C= 1, random_state = 0)
```

```
[[3980   80]
 [ 252 1424]]
Validation Accuracy: 93.10 %
Validation Standard Deviation: 1.05 %
Validation Loss 0.77
```

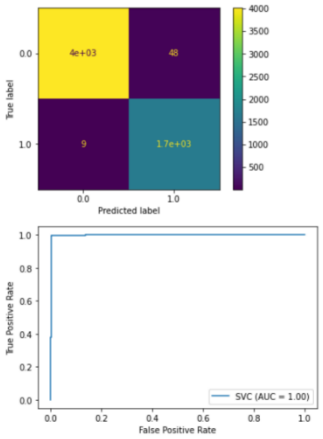



```
[[4054   78]
 [ 213 1391]]
Test Accuracy: 93.34 %
Test Standard Deviation: 0.77 %
Test Loss 0.77
```

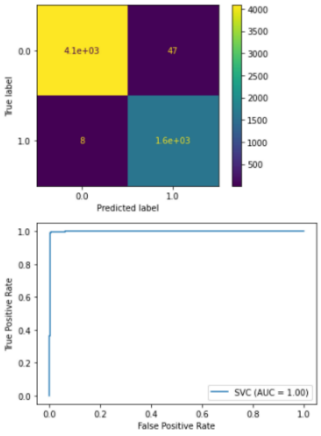



```
# create SVM model
classifier = SVC(kernel = 'poly', C= 1, gamma = 7, random_state = 0)
```

```
[[4012   48]
 [   9 1667]]
Validation Accuracy: 98.71 %
Validation Standard Deviation: 0.29 %
Validation Loss 0.72
```

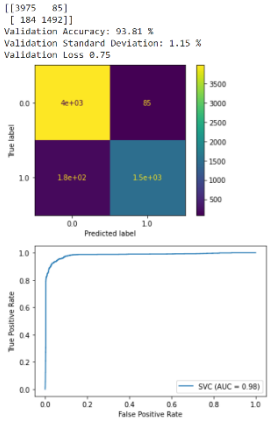



```
[[4085   47]
 [   8 1596]]
Test Accuracy: 98.73 %
Test Standard Deviation: 0.59 %
Test Loss 0.73
```




```
# create SVM model
classifier = SVC(kernel = 'rbf', C= 1, gamma = 3, random_state = 0)
```

```
[[3975   85]
 [ 184 1492]]
Validation Accuracy: 93.81 %
Validation Standard Deviation: 1.15 %
Validation Loss 0.75
```




```
[[4053   79]
 [ 184 1420]]
Test Accuracy: 94.12 %
Test Standard Deviation: 0.79 %
Test Loss 0.77
```