Placement of Products in a Retail Store

Problem Statement

"We would like you to implement a solution for product verification in retail stores. Specifically, given a set of product spaces each containing some product images, write a solution to alarm a store operator whether a product is kept in its correct product space or not. A product that doesn't belong to the correct product space is called a Plug."

The Data

Description

We have 3 files:

1) **Embeddings.csv** – contains feature embeddings, sized (1280,1) of images of items in a retail store. I extracted this through **numpy.genfromtxt** to get an array of shape **(13008, 1280)**.
2) **Product_spaces.csv** – contains the identifier for the product space corresponding to each item. Shape -> **(13008, 1)**, after expanding by one dimension.
3) **Plug_labels.csv** – indicates whether the item is in the right product space. 0, if it the given product space is right, 1, otherwise. All misplaced objects are called plugs. Shape -> **(13008, 1)** after expanding by one dimension.

We have combined these three tables into a single data frame with column names ranging from **'1' to '1282'**.

Let us dive deeper into the data to understand it better.

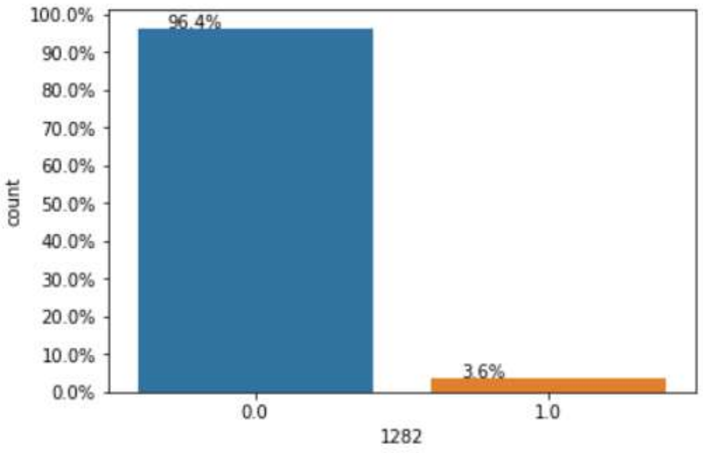
Data Preprocessing

1) Splitting the above created data set into the original three parts to perform **individual operations** on each.
2) Creating a dictionary that contains the **number of occurrences** for each product space.
3) Creating a dictionary that maps the 298 product spaces to 298 **index values**.
4) Using the index values as a single feature for product spaces in a new set called **'cat_data'**.
5) **Normalizing** all columns in cat_data.
6) **Normalizing** the embedding values for each column.
7) Creating **one-hot-encoded** vectors for each product space. The resulting shape would be **(n_samples, 298)**. The encoder returns a **sparse matrix**, which has to be converted to dense before joining back with rest of the data.

8) Horizontally **stacking** the normalized embeddings, one-hot-encoded product spaces, and plug labels to create a data frame called **‘ohe_data’**.
9) Giving names to all the columns ranging from **‘1’ to ‘1579’**. (1280 embedding features, 298 product space features + 1 plug label feature).

Exploratory Data Analysis

How many plugs are there in the whole set?

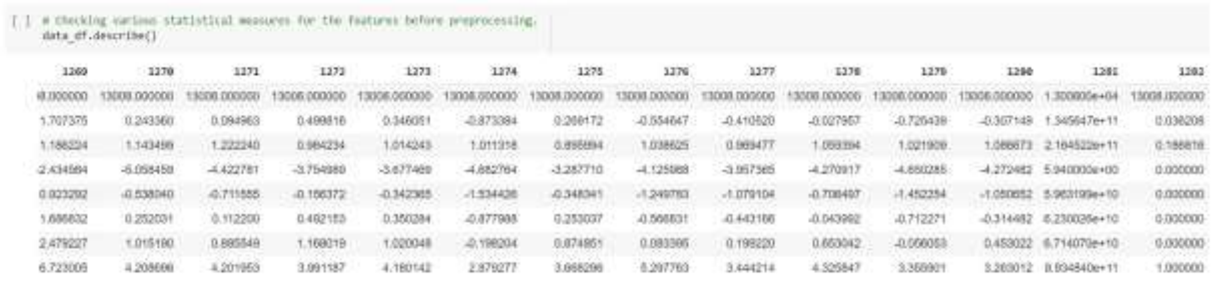


The data is grossly imbalanced. Specifically, there are **471 plugs and 12537 non-plugs**. We will have to rectify this during training.

How many unique product spaces do we have? What are the values like?

The values of product spaces are of the order ~ 10^{10}. There are **298 unique product spaces**. We will encode this as a categorical variable using one-hot encoding.

What do the other features look like?



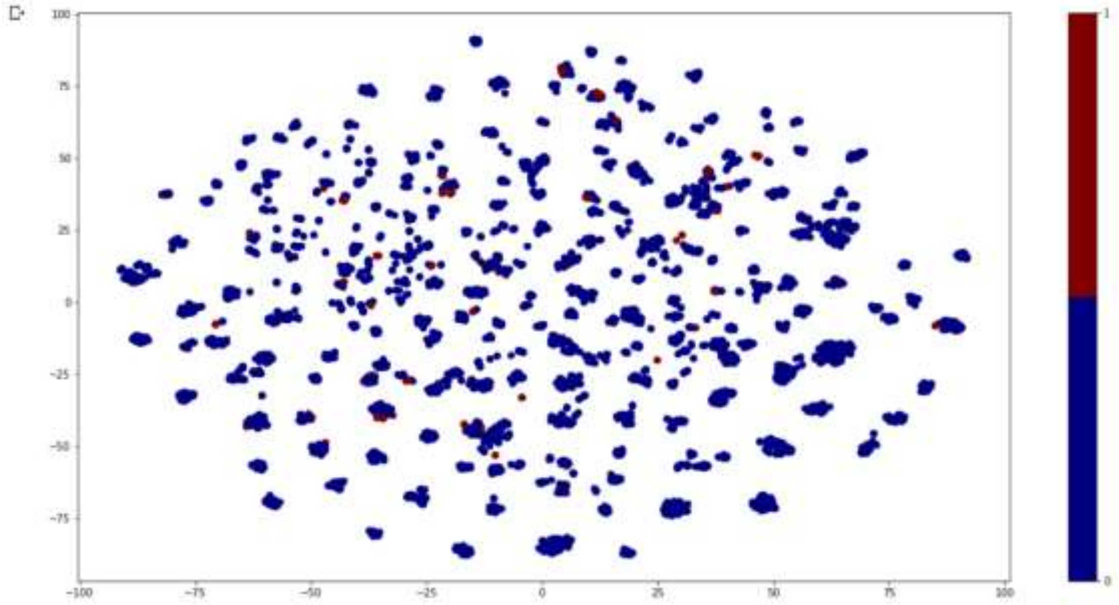
```
[ ] # Checking various statistical measures for the features before preprocessing.
    data_df.describe()
```

```
        1269          1270          1271          1272          1273          1274          1275          1276          1277          1278          1279          1280          1281          1282
13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  13008.000000  1.300800e+04  13008.000000
    1.707375      0.243360      0.094963      0.499816      0.346051     -0.873384      0.266172     -0.554647     -0.410520     -0.027957     -0.726439     -0.307149  1.345647e+11      0.036208
    1.188224      1.143499      1.222240      0.984234      1.014243      1.011318      0.895894      1.038625      0.989477      1.059394      1.021909      1.066673  2.164522e+11      0.186816
   -2.434584     -6.058458     -4.422781     -3.754989     -3.677469     -4.882764     -3.287710     -4.125988     -3.957365     -4.270917     -4.850285     -4.272482  5.940000e+00      0.000000
    0.023292     -0.538040     -0.711555     -0.166372     -0.342365     -1.534426     -0.348341     -1.249783     -1.079104     -0.708497     -1.452254     -1.050852  5.963199e+10      0.000000
    1.686632      0.252031      0.112200      0.492153      0.350284     -0.877985      0.253037     -0.566831     -0.443166     -0.043992     -0.712271     -0.314482  6.230026e+10      0.000000
    2.479227      1.015180      0.885549      1.168019      1.020048     -0.198204      0.874851      0.083386      0.198220      0.653042     -0.056053      0.453022  6.714070e+10      0.000000
    6.723005      4.208698      4.201953      3.991187      4.180142      2.879277      3.668296      5.297763      3.444214      4.325847      3.356901      3.263012  8.934840e+11      1.000000
```

The feature embeddings have both positive and negative values and lie between **(-10, 10)**.

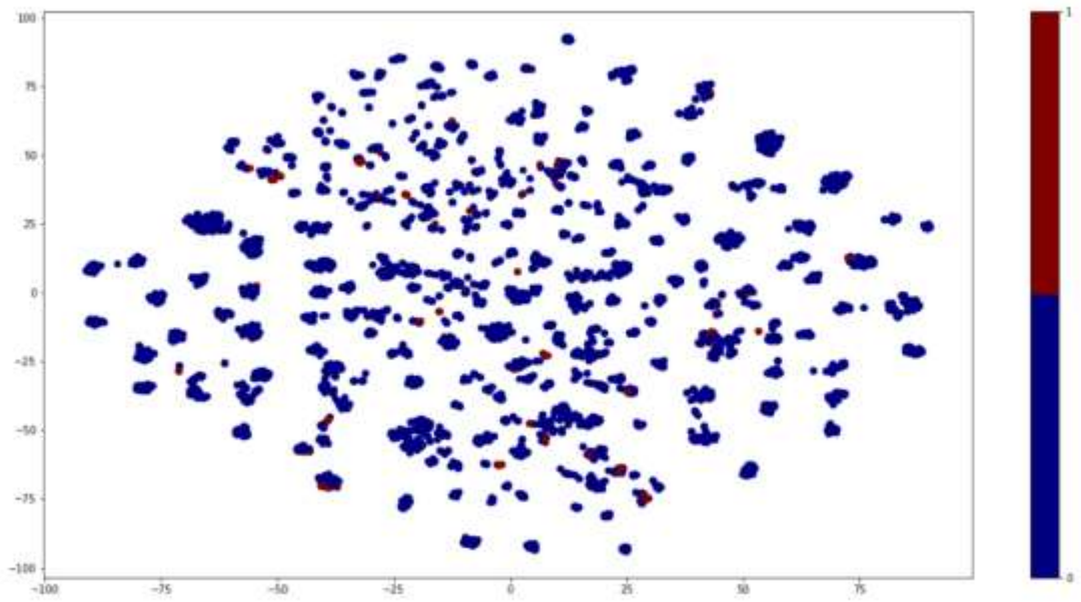
Are the plugs and non-plugs separable using this featurization?

For this we plotted the **TSNE** plot in 2 dimensions with labels as plug labels. Here is the plot for 'ohe_data':



Some amount of **separation** can be observed between clusters of plugs and those of non-plugs (the majority). It can also be observed that the number of clearly separable clusters is of the order of the number of unique product spaces. Hence, it won't be unwise to try to separate plugs from non-plugs using classifiers that use neighbor information.

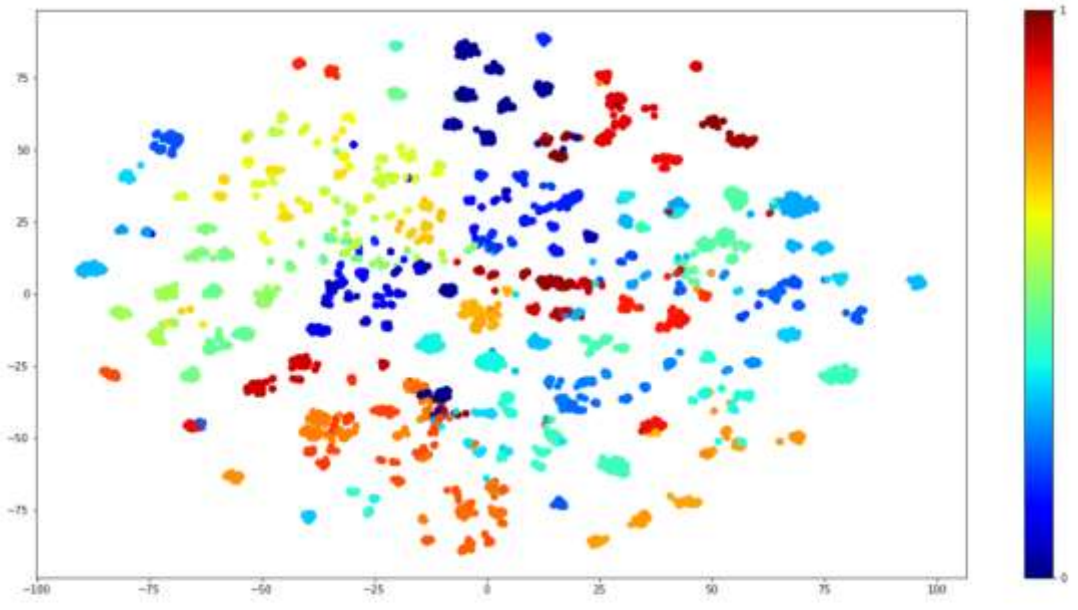
Now let us create the same plot for 'cat_data':



There is no significant difference between clusters of same and different classes in this plot as compared to the previous one. Hence, we should try both feature sets and determine what works better in practice.

Can all the product spaces be distinguished on the basis of the embeddings?

We will check if all the product spaces can be separated using the image feature embeddings. Here is the **TSNE** plot:



Quite clearly, we are able to separate the items based on product spaces significantly well in **two dimensions** itself. This is a motivation for us to train a model to learn the relationship between image features and product spaces, predict the product space for any item, and check whether it is correctly/incorrectly predicted.

Data sets used throughout

The two ways of using this data

1) **Using product_space as a feature and plug_label as the target.** This way we are checking for any given item (embeddings) and product space, if the placement/pairing is correct. The plug label (target) is 0 if yes, otherwise it is 1. We will try this in the following two ways:
   a) Using the **indices** of product spaces as the input along with embeddings. The total input shape would be **(13008, 1281)** in this case.
   b) Using the **one-hot-encoded** representations of the product spaces along with the embeddings. The total input shape will be **(13008, 1578)** in this case.

The shape of the output in both cases will be **(13008,).**

We will these approaches for out machine learning models as well as with an neural network architecture.

2) **Using product_space as the target.** We will only use only **non-plugs** while training this model. We will train a deep neural network to learn the relationship between items and product spaces. After it learns this, we will use the test data (already set aside) to predict the product spaces for each item. We will then check whether it is the prediction is matching in case of non-plugs, and whether it is different in the case of plugs. The shape of the input will be **(13008, 1280)** and that of the output will be **(13008, 298)**.

Here are the data sets that will be used at some point or the other:

1) **X**
Complete data to be used during training.
Derived from **ohe_data.**
Used for up sampling the training data, contains 4000 plugs up sampled from 377 plugs with replacement.
Final shape -> **(14029, 1579)**.

2) **x_unb, y_unb**
Data separated out from **ohe_data** before up sampling, to be used for testing.
Final shape -> **(2602, 1579), (2602,)**.
Contains 2508 non-plugs and 94 plugs.
This split is made to mimic the proportions given in the original set, which should be reflective of the real-world scenario.

3) **X_cat**
Complete data to be used during training.
Derived from **cat_data**.
Used for up sampling the training data, contains 4000 plugs up sampled from 377 plugs with replacement.
Final shape -> **(14029, 1579)**.

4) **x_unb_cat, y_unb_cat**
Data separated out from **cat_data** before up sampling, to be used for testing.
Final shape -> **(2602, 1579), (2602,)**.
Contains 2508 non-plugs and 94 plugs.
This split is made to mimic the proportions given in the original set, which should be reflective of the real-world scenario.

5) **x_train, y_train**
Derived from **train_test_split** of X.
All corresponding models will be trained on this.
Shape -> **(11223, 1578), (11223,)**.

6) **x_train_cat, y_train_cat**
Derived from **train_test_split** of X_cat.
All corresponding models will be trained on this.
Shape -> **(11223, 1281), (11223,)**.

7) **x_cv, y_cv**
Derived from **train_test_split** of X.
Cross-validation will be conducted using this data, for all corresponding models.
Shape -> **(2806, 1578), (2806)**.

8) **x_cv_cat, y_cv_cat**
Derived from **train_test_split** of X.
Cross-validation will be conducted using this data, for all corresponding models.
Shape -> **(2806, 1281), (2806)**.

9) **x_sep_train, y_sep_train**
Contains all the points from **x_train** that get predicted as plugs by the Gaussian Naïve Bayes model (**GNB**). The shape would vary depending on the predictions.
Here it was **(5137, 1578), (5137,)**.
y_sep_train is simply a copy of **y_train**.

10) **x_sep_cv, y_sep_cv**
Contains all the points from **x_cv** that get predicted as plugs by the **GNB**. The shape would vary depending on the predictions.
Here it was **(1298, 1578), (1298,)**.
y_sep_cv is simply a copy of **y_cv**.

11) **x_sep_unb, y_sep_unb**
Contains all the points from **x_unb** that get predicted as plugs by the **GNB** model. The shape would vary depending on the predictions.
Here it was **(711, 1578), (711,)**.
y_sep_unb is simply a copy of **y_unb.**

12) **x_nb, y_nb**

Copy of **x_train +** feature containing predictions made by **GNB** model on x_train.
y_nb is simply a copy of **y_train**.

13) **x_nb_cv, y_nb_cv**
Copy of **x_cv +** feature containing predictions made by **GNB** model on x_cv.
y_nb_cv is simply a copy of **y_cv**.

14) **x_nb_unb, y_nb_unb**
Copy of **x_unb +** feature containing predictions made by **GNB** on x_unb.
y_nb_unb is simply a copy of **y_unb**.

Metrics and Losses

Here are the various losses/metrics used with the reason they were selected:

1) **Log-loss/Binary Cross-Entropy**
Regarded as a reliable metric for binary classification tasks that gives an indication of how confident a model is while making predictions.

2) **Accuracy score**
Gives an idea of the overall performance. However, this is susceptible to misleading results due to class imbalance (as can be observed in the baseline model).

3) **Area Under Receiver Operating Characteristic Curve (roc_auc_score)**
A metric that helps us optimize towards a balance between True Positive Rate and False Positive Rate. Accounts for class imbalance and gives a result that indicates a problem in case the minority class is not being handled well.

4) **F1 score**
Another metric that is very useful for binary classification, especially for imbalanced classes, as it strikes a trade-off between Precision and Recall.

5) **Confusion matrix**
Gives the clearest representation of the scenario after prediction, with the exact number of correctly/incorrectly classified points for each class. This will help us selected the strategies we have to use moving forward.

6) **Precision (matrix)**
Displays the values of Precision for all the classes involved.

7) **Recall (matrix)**
Displays the values of Recall for all the classes involved. All these matrices will help us decide if we are missing too many plugs, classifying too many-non plugs as plugs, or any other error being produced.

8) **Categorical Cross-Entropy**
The most-popular metric for multi-class classification. We will need this for our task of predicting product spaces from image feature embeddings.

Machine Learning models

Baseline Model

If we classify all products as non-plugs (majority) then the following are values we will get for various metrics:

```
Accuracy: 0.9638739431206764
ROC-AUC: 0.5
F1 score: 0.0
```

We should try and aim for an overall accuracy (and others) better than this. However, considering the fact that our main focus is to alert the retailer about all the **plugs**, False Positives may be allowed, in order to increase the number of plugs detected and reduce the number of false negatives (positive being where there is a plug, in this context).

General Approach

1) (Fitting model -> calibrated classifier -> calculating metrics) for all hyperparameter (hp) values in the hp range,
2) Selecting the best value of hp and fitting the model with that value,
3) Plotting the confusion matrices from the function above,
4) Interpreting results.

Calibrated Classifier

This classifier is used **on top of** the basic classification model in order to generate **probability values** for each prediction instead of hard predictions. This will be required to calculate the two-class log-loss and roc_auc_score.

The two ways to perform this calibration is through '**sigmoidal**' and '**isotonic**' regression. We have mostly used the former.

Logistic Regression

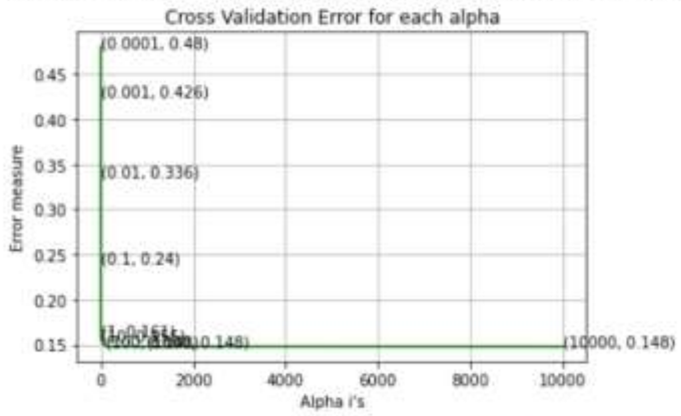
This is the first model one would test for binary classification to check whether the data is **linearly separable** or not.

It is **fast**, and can handle data with **high dimensionality** well. We tried various fits of this model, both on 'cat_data' and 'ohe_data'. It was observed that the performance was much better using ohe_data.

The hyperparameter we will vary is **C**, which is the inverse of **regularization** strength.

Here are the results for cat_data:

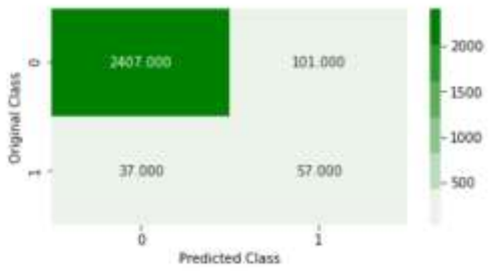
```
100% 9/9 [01:51<00:00, 12.35s/it]

log_loss for c =  0.0001 is 0.48025487951272805 ROC-AUC score is:  0.7941251246261216
log_loss for c =  0.001 is 0.42564756309727814 ROC-AUC score is:  0.8467771684945165
log_loss for c =  0.01 is 0.33619179163821744 ROC-AUC score is:  0.9119653539381855
log_loss for c =  0.1 is 0.2399765371222802 ROC-AUC score is:  0.9574389332003987
log_loss for c =  1 is 0.16067152290458422 ROC-AUC score is:  0.9794317048853439
log_loss for c =  10 is 0.15522159991789175 ROC-AUC score is:  0.9791718594217348
log_loss for c =  100 is 0.14830434048007826 ROC-AUC score is:  0.9804642323030907
log_loss for c =  1000 is 0.14808744067561994 ROC-AUC score is:  0.9805277916251246
log_loss for c =  10000 is 0.14780215677035585 ROC-AUC score is:  0.9802991026919242
```



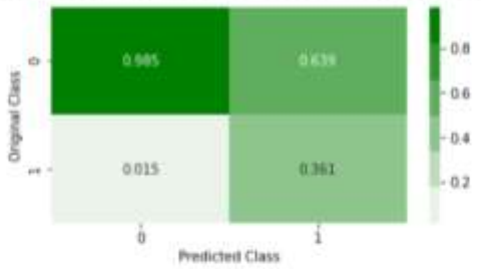
```
log loss for train data 0.09482515096727974
log loss for cv data 0.14780215677035585
log loss for test data 0.18307028666454087
Percentage of misclassified points  5.303612605687932
```

The best fit was observed for **C = 10000.**

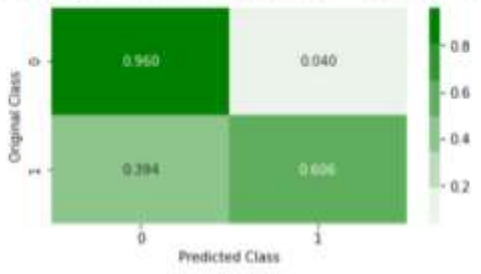
```
-------------------------------------------------- Confusion matrix --------------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



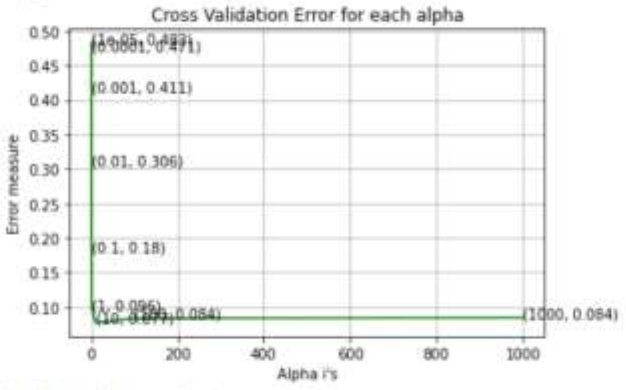
```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



This means that the approach we are using (product spaces as features) has the potential to yield satisfactory results. We are able to correctly detect **~60% plugs** and only **4% of non-plugs** are incorrectly classified. However, we still are getting a lot of False Positives and False Negatives.

Here are the results for ohe_data:

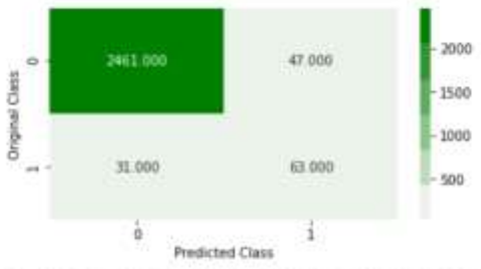
```
100% 9/9 [01:49<00:00, 12.21s/it]

log_loss for c =  1e-05 is 0.48254955985723735 ROC-AUC score is:  0.7918855932203389
log_loss for c =  0.0001 is 0.4709487011939191 ROC-AUC score is:  0.8041201395812562
log_loss for c =  0.001 is 0.411162634761793 ROC-AUC score is:  0.8578327517447657
log_loss for c =  0.01 is 0.3063161369256706 ROC-AUC score is:  0.9275392572283149
log_loss for c =  0.1 is 0.18003460915514322 ROC-AUC score is:  0.9760848703888334
log_loss for c =  1 is 0.09617821082030996 ROC-AUC score is:  0.990912886340977
log_loss for c =  10 is 0.0773518259601757 ROC-AUC score is:  0.9926869391824527
log_loss for c =  100 is 0.08365125610760203 ROC-AUC score is:  0.991811440677966
log_loss for c =  1000 is 0.08446228800535627 ROC-AUC score is:  0.9917204636091724
```



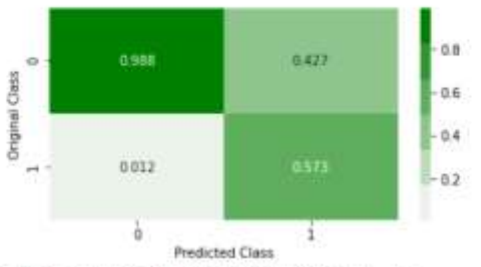
```
log loss for train data 0.02853554072568799
log loss for cv data 0.0773518259601757
log loss for test data 0.1312095909015226
Percentage of misclassified points  2.997694081475788
```

The best fit was observed with **C = 10.**

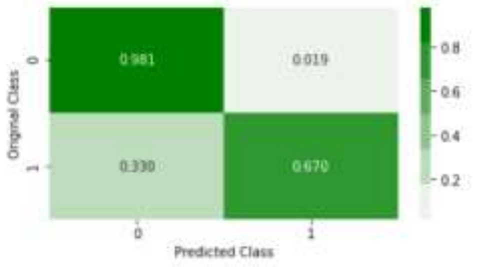
```
-------------------------------------------------- Confusion matrix ---------------------
```



```
-------------------------------------------------- Precision matrix ---------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix ------------------------
```



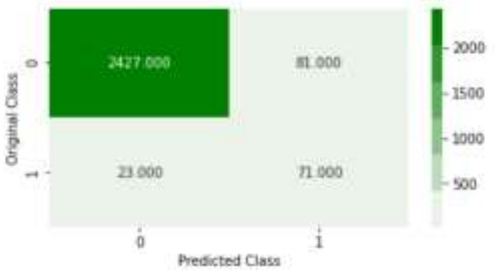
```
Sum of rows in recall matrix [1. 1.]
```

The results are even **better than before**. Logistic Regression is performing better with product spaces as one-hot-encoded features. The number of Falsely classified points has reduced on both sides. This in itself won't be too bad a solution.

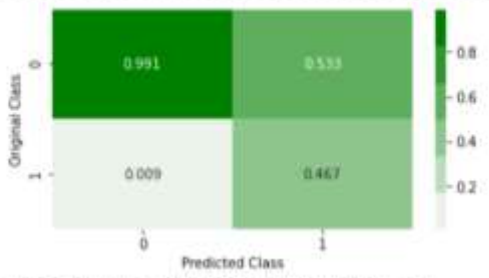
Here are various other metrics (obtained on refitting the model):

```
Accuracy score: 0.9600307455803229
ROC-AUC: 0.9534765346635448
F1 score: 0.5772357723577236
```

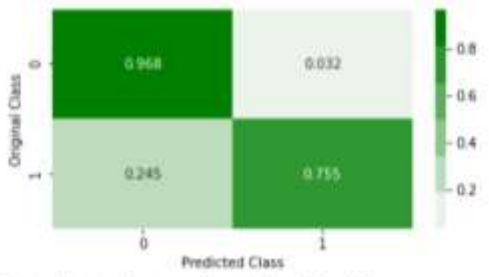
```
Percentage of misclassified points  3.996925441967717
------------------------------------------------- Confusion matrix ------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

The Recall is reasonably good but the Precision still has **scope for improvement.** We shall have to explore further models for this. We don't want the store employees making unnecessary rounds, do we?

Naïve Bayes

Naive Bayes has been extensively used for **text classification** with a large number of features. We are trying two types of NB models to see if this problem can be successfully solved the same principles:

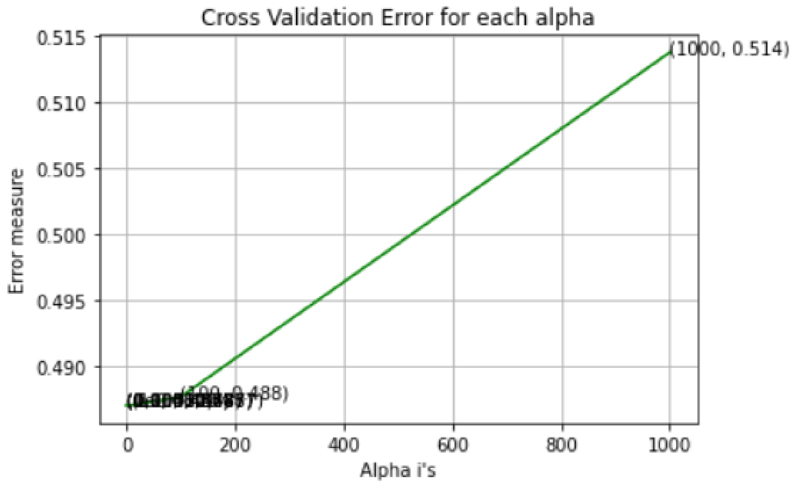
1. **Multinomial Naive Bayes** - even though this method is more suited for discrete distributions, it does work adequately well in practice with **tf-idf** features as well.

2. **Gaussian Naive Bayes** - This is meant for **continuous distributions** as it is, which is why we can expect positive results.

Multinomial Naïve Bayes

For all the fits, we are varying the value of 'alpha' which is the parameter for **Laplace smoothing.** Here are the results of various fits.

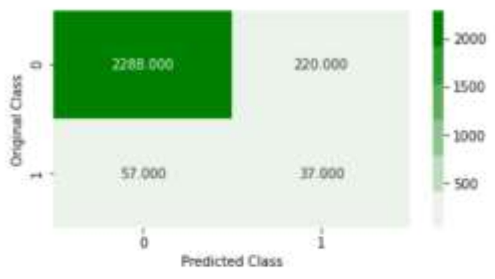
```
100% 9/9 [00:07<00:00, 1.18it/s]

log_loss for c =  1e-05 is 0.4870761983894084 ROC-AUC score is:  0.7803364905284148
log_loss for c =  0.0001 is 0.48707619846454625 ROC-AUC score is:  0.7803364905284148
log_loss for c =  0.001 is 0.48707619921618633 ROC-AUC score is:  0.7803364905284148
log_loss for c =  0.01 is 0.4870762067379456 ROC-AUC score is:  0.780337736789631
log_loss for c =  0.1 is 0.4870762824926579 ROC-AUC score is:  0.7803333748753739
log_loss for c =  1 is 0.4870770936983 ROC-AUC score is:  0.7803545613160519
log_loss for c =  10 is 0.4870905086576435 ROC-AUC score is:  0.7803333748753739
log_loss for c =  100 is 0.48769892143081467 ROC-AUC score is:  0.779168743768694
log_loss for c =  1000 is 0.5137164368796754 ROC-AUC score is:  0.7402268195413758
```



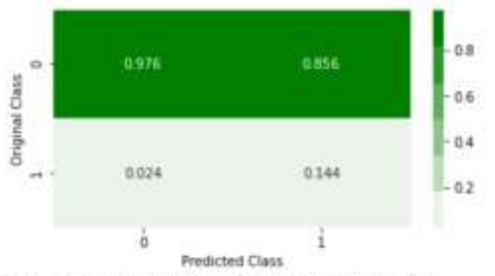
```
log loss for train data 0.47630095147888357
log loss for cv data 0.4870761983894084
log loss for test data 0.30771256276506853
Percentage of misclassified points  10.6456571867794
```

Looks like the model is performing better with small values of the smoothing parameter.

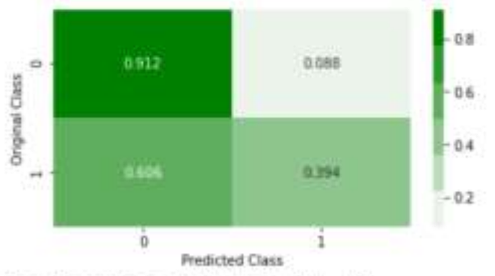
```
---------------------------------------------------- Confusion matrix ----------------------------------------------------
```



```
---------------------------------------------------- Precision matrix ----------------------------------------------------
```



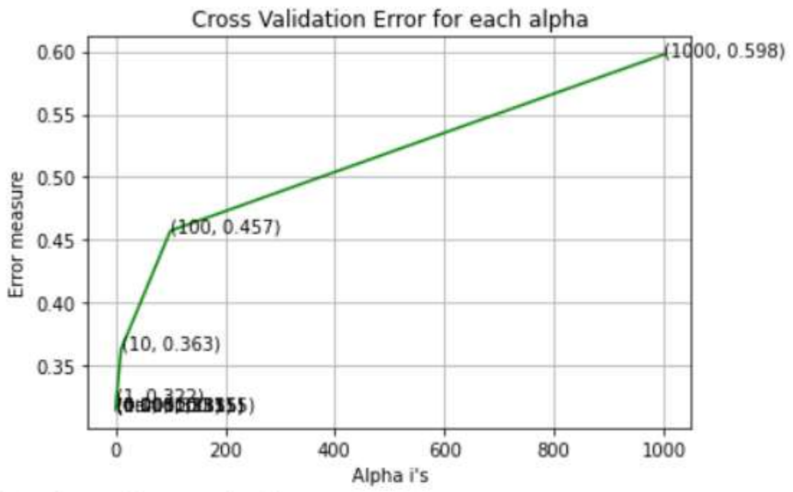
```
Sum of columns in precision matrix [1. 1.]
---------------------------------------------------- Recall matrix ----------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

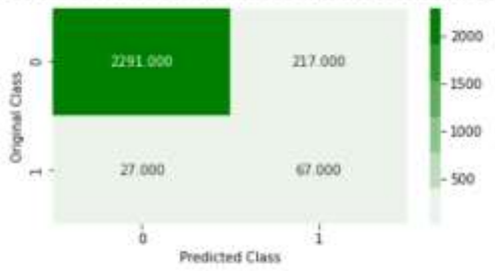
Seems like this **did not work** out as well as we had hoped. Both False Positives and False Negatives have **increased** as compared to the values Logistic Regression was producing. Let's try out 'ohe_data' and see if there is any improvement.

```
100% 9/9 [00:11<00:00, 1.26s/it]

log_loss for c =  1e-05 is 0.31466236273756365 ROC-AUC score is:  0.9390790129611166
log_loss for c =  0.0001 is 0.31466298608198523 ROC-AUC score is:  0.9390790129611166
log_loss for c =  0.001 is 0.3146692200420638 ROC-AUC score is:  0.9390771435692921
log_loss for c =  0.01 is 0.3147316103671519 ROC-AUC score is:  0.9390715353938186
log_loss for c =  0.1 is 0.31535979480521903 ROC-AUC score is:  0.938916999002991
log_loss for c =  1 is 0.321719581932059 ROC-AUC score is:  0.9306225074775673
log_loss for c =  10 is 0.363141470580931 ROC-AUC score is:  0.8891718594217348
log_loss for c =  100 is 0.4573514538093067 ROC-AUC score is:  0.8302754237288136
log_loss for c =  1000 is 0.5977088806745011 ROC-AUC score is:  0.5
```

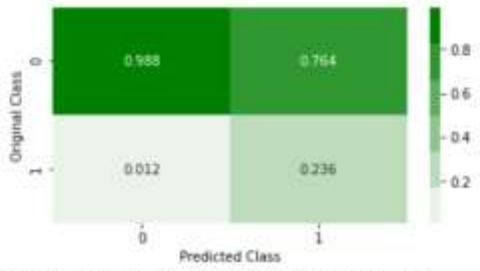


```
log loss for train data 0.28612227116858735
log loss for cv data 0.31466236273756365
log loss for test data 0.21556264906822814
Percentage of misclassified points  9.377401998462721
```

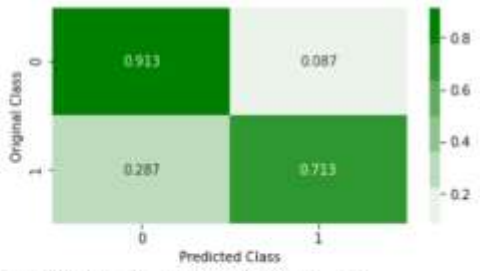
```
-------------------------------------------------- Confusion matrix --------------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

Considering the plugs, we can say there is a significant **improvement** after using 'ohe_data' instead of 'cat_data'. We have a decent plug detection rate but the False Positives are too many, hence the precision is very low.

Here are the values of various metrics for this model:

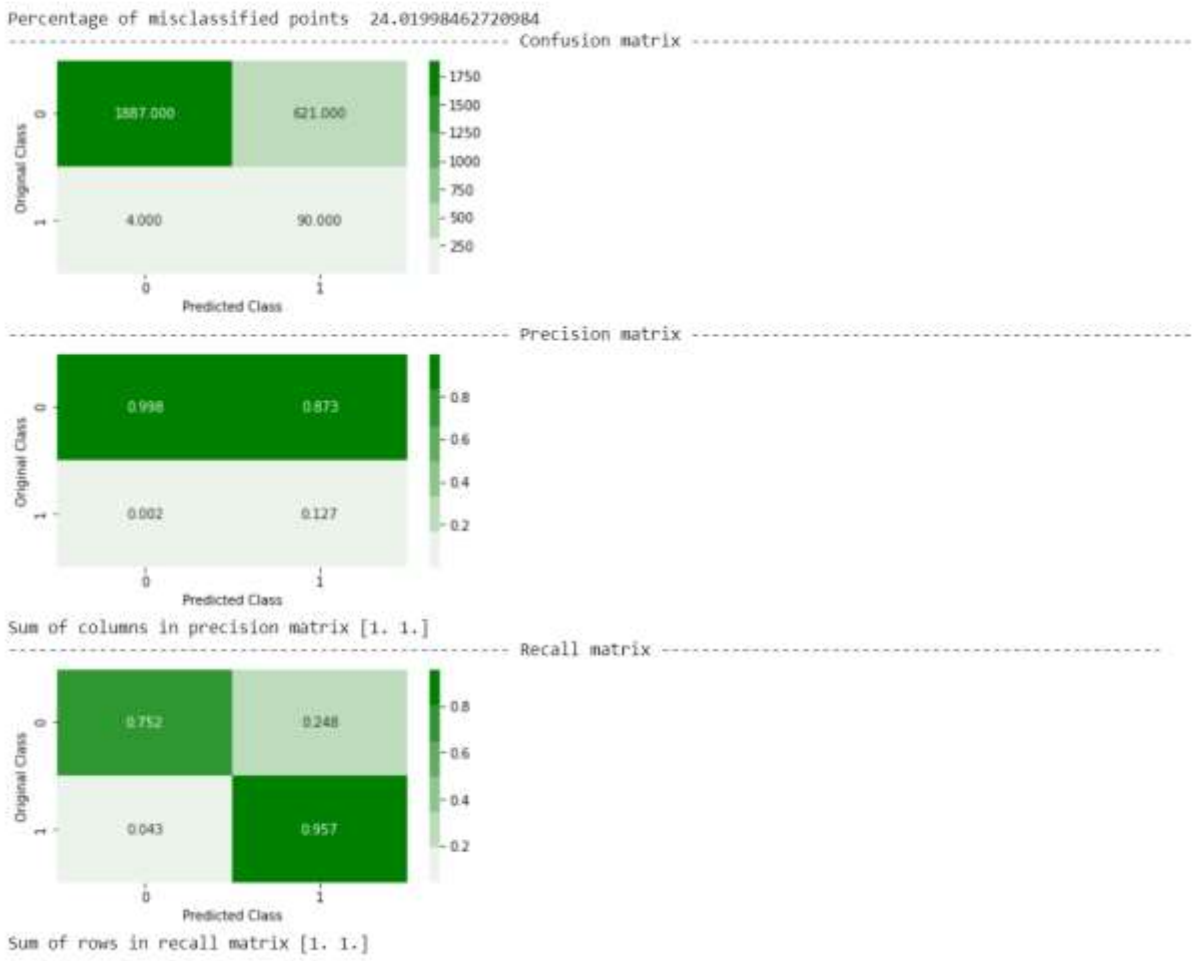
```
Accuracy score: 0.8912375096079939
ROC-AUC: 0.919177780040042
F1 score: 0.33096926713947994
```

The number of False Negatives has **decreased** even more in this fit but the number of False Positives has **increased**. We may explore the possibility that there is a **trade-off** between the two values in this data.

Gaussian Naïve Bayes

We are training this model with **var_smoothing = 0.01.** The performance is given below:

```
Accuracy score: 0.7598001537279017
ROC-AUC:0.8644592622756118
F1 score: 0.22360248447204972
```



```
Percentage of misclassified points  24.01998462720984
-------------------------------------------------- Confusion matrix --------------------------------------------------
-------------------------------------------------- Precision matrix --------------------------------------------------
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
Sum of rows in recall matrix [1. 1.]
```

Very interesting results:

1. The **Recall has improved** even more, to a very impressive level (~0.96).

2. The **Precision has decreased** even more, to a very troublesome level (~0.127).

3. We can think of a strategy in which we use the fact that most of the plugs are detected by GNB, and somehow **filter out the non-plugs** to return a result with high values of both Precision and Recall.

4. Both the Naive Bayes classifiers are **extremely fast**, which takes away the reservation against adding too many steps to the pipeline.

We shall have to explore further.

Support Vector Machines (SVC)

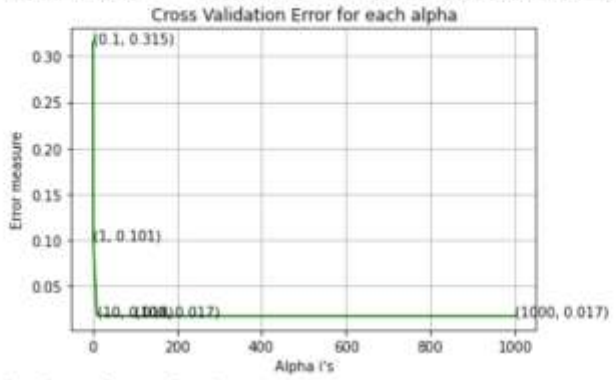
The SVC class is based on the **libsvm** library which implements the kernel feature.

The given features will be mapped to a much **higher dimensional space** for training.

As this is a medium sized dataset with **many features,** this classifier is an appropriate choice.

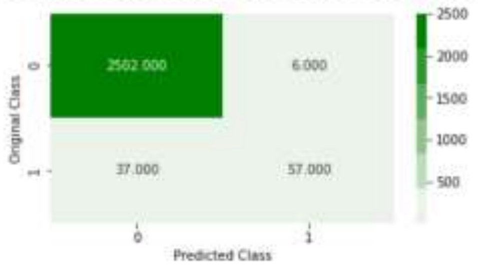
It also handles **sparse features** well. As we have seen better performance for 'ohe_data' in both previous models, we will use only that data from now on.

```
100%  5/5 [59:39<00:00, 715.91s/it]

log_loss for c =  0.1 is 0.3153906737877981 ROC-AUC score is:  0.9229860418743768
log_loss for c =  1 is 0.10096931735828735 ROC-AUC score is:  0.9920351445663012
log_loss for c =  10 is 0.017817896480902123 ROC-AUC score is:  0.9991070538384846
log_loss for c =  100 is 0.016974035187893078 ROC-AUC score is:  0.9990827517447657
log_loss for c =  1000 is 0.016974092337156115 ROC-AUC score is:  0.9990827517447657
```

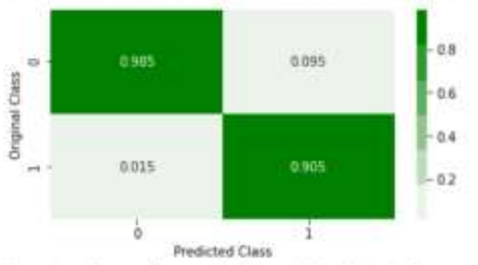


```
log loss for train data 0.0031316578772782816
log loss for cv data 0.016974035187893078
log loss for test data 0.1095187832239537
Percentage of misclassified points  1.6525749423520368
```

```
Percentage of misclassified points  1.6525749423520368
-------------------------------------------------- Confusion matrix --------------------------------------------------
```

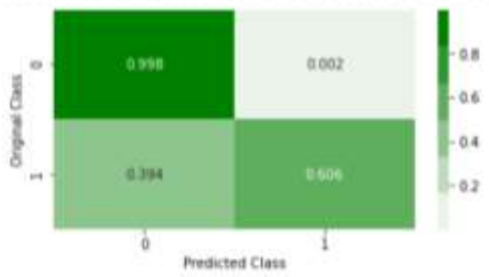


```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
```

```
--------------------------------------------------- Recall matrix ---------------------------------------------------
```



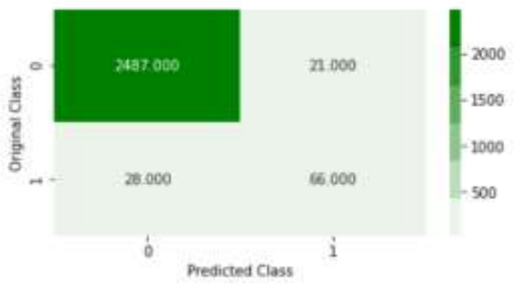
```
Sum of rows in recall matrix [1. 1.]
```

The number of False Positives has reduced significantly, to as low as **6.** However, we have **missed more plugs** as compared to the previous two models. Let us check other metrics after fitting it again.

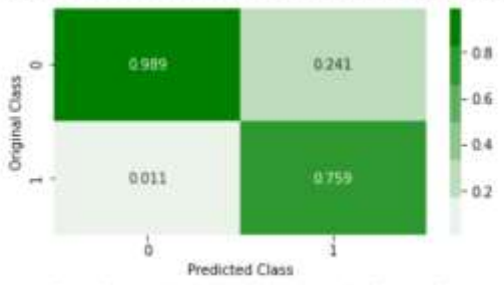
Let is fit the model again with **C = 1000.** The performance is given below:

```
Accuracy score: 0.9811683320522675
ROC-AUC: 0.9565645254335065
F1 score: 0.729281767955801
```

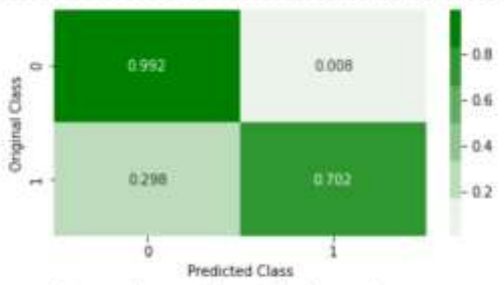
```
Percentage of misclassified points  1.8831667947732513
--------------------------------------------------- Confusion matrix ---------------------------------------------------
```



```
--------------------------------------------------- Precision matrix ---------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
--------------------------------------------------- Recall matrix ---------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

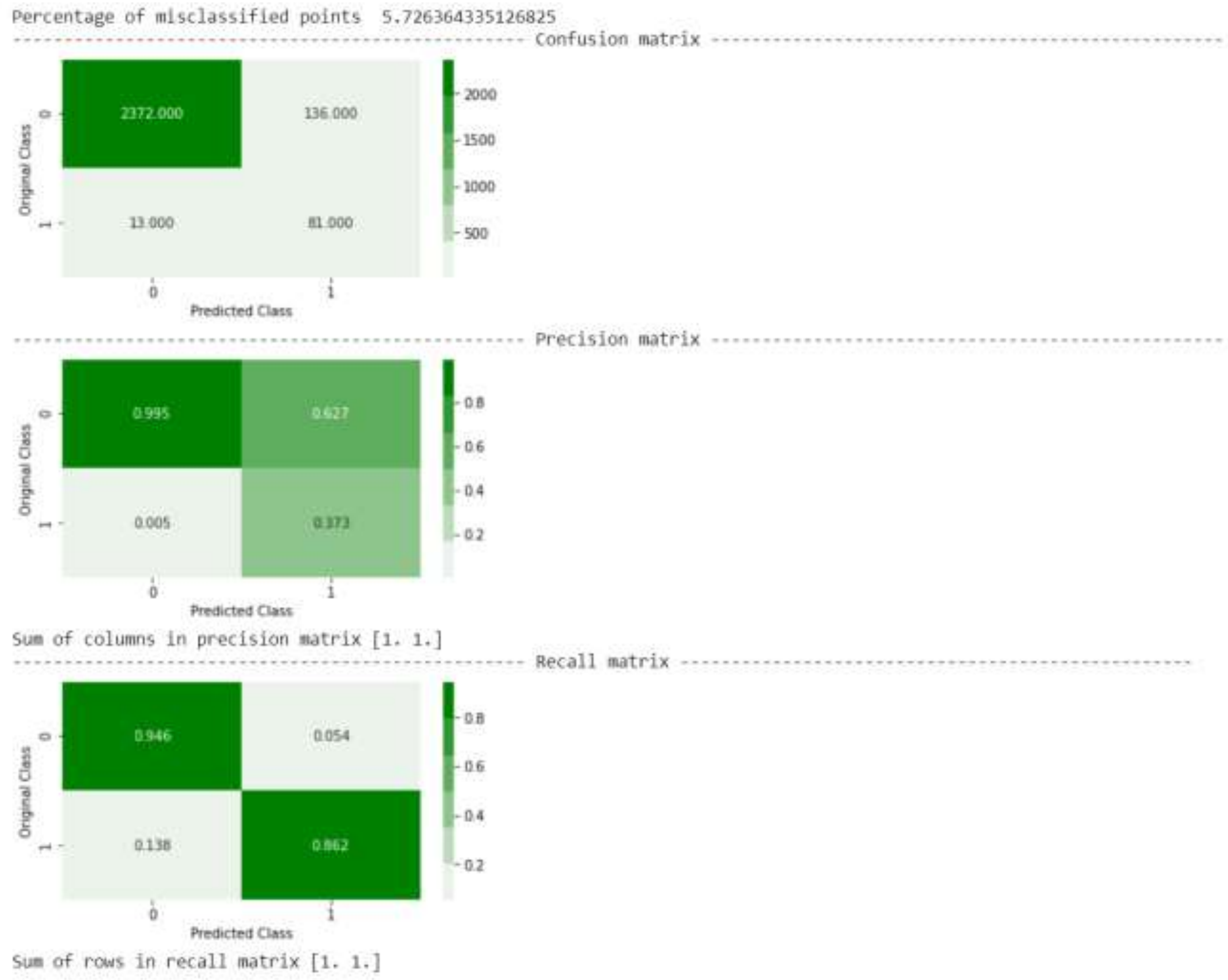
Very **promising** results.

For the first time we are seeing something close to what is desired.

On the other hand, the **time taken** for training and for prediction is significantly more than the previously used models. Still, a prediction time of 1 minute for ~2600 items is not too slow either. Depends on what the retailer wants.

Now, for experimental purposes, let us try and fit the model with **C = 1.** Even though we might compromise on the overall accuracy, let us see if the results are favorable in some way. The performance is given below.

```
Accuracy score: 0.9427363566487318
ROC-AUC: 0.9565645254335062
F1 score: 0.5209003215434084
```



```
Percentage of misclassified points  5.726364335126825
-------------------------------------------------- Confusion matrix --------------------------------------------------
-------------------------------------------------- Precision matrix --------------------------------------------------
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
Sum of rows in recall matrix [1. 1.]
```

Fascinating.

Looks like our understanding of the **trade-off** between Precision and Recall might be correct.

Again, if the retailer does not mind checking some extra items that aren't actually plugs, in order to miss less of them, this solution would be more suitable for them.

Let us explore further.

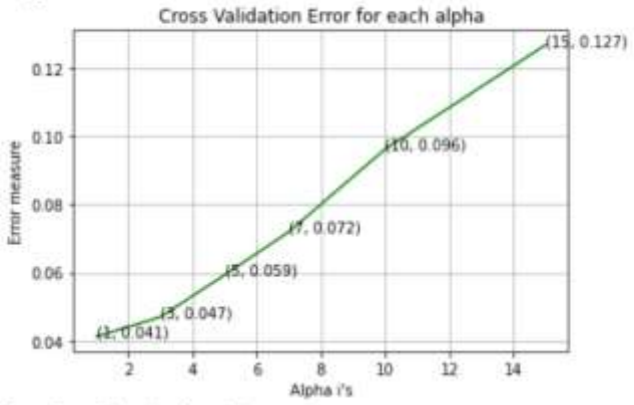
K Nearest Neighbors

This classifier fundamentally finds the closest 'n' **neighbors** of the query point using one of **'ball_tree', 'kd_tree'**, and brute force search methods, in the vector space.

KNN has been used at other places for learning the **similarity** between items using image feature embeddings. Our hope is that it will work here as well.

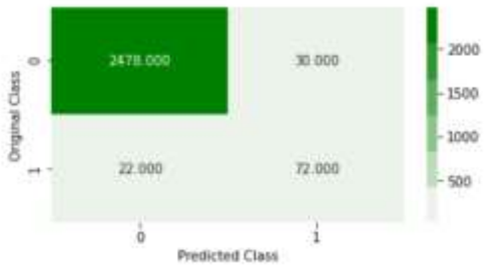
We are varying the values of **K** from **1** to **15** in the following fits.

```
100% ████████████████████ 6/6 [1:48:38<00:00, 1086.46s/it]

log_loss for c =  1 is 0.04127528182480418 ROC-AUC score is:  0.9940179461615155
log_loss for c =  3 is 0.047016796212395844 ROC-AUC score is:  0.9938609172482552
log_loss for c =  5 is 0.059319937257476564 ROC-AUC score is:  0.9938792996011964
log_loss for c =  7 is 0.07196602744794381 ROC-AUC score is:  0.9938643444666002
log_loss for c =  10 is 0.0961317937868028 ROC-AUC score is:  0.9922096211365902
log_loss for c =  15 is 0.12665972293925112 ROC-AUC score is:  0.9889693419740778
```

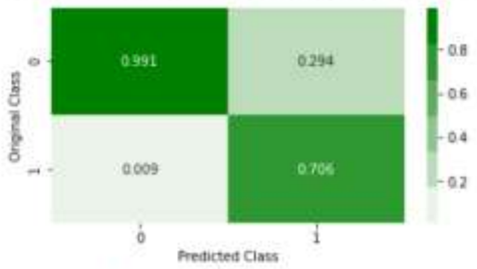


```
log loss for train data 0.013764235948197763
log loss for cv data 0.04127528182480418
log loss for test data 0.09096528337510616
Percentage of misclassified points  1.9984627209838586
```

```
-------------------------------------------------- Confusion matrix --------------------------------------------------
```

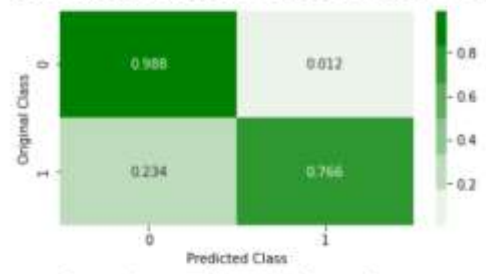


```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
```

```
-------------------------------------------------- Recall matrix --------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

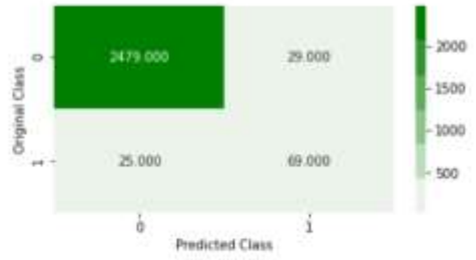
Overall, these seem to be the **best results so far**.

Out of 2602 points only **22/94** plugs were missed and only **30/2508** non-plugs were classified as plugs.

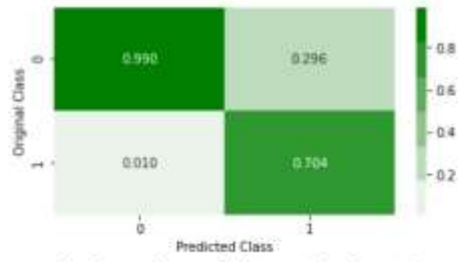
Just to experiment, let us see if the number of plugs missed reduces if we change **K to 5** and used **weighted distance.** The performance is shown below:

```
Accuracy score: 0.9792467332820907
ROC-AUC: 0.9406091994977773
F1 score: 0.71875
```

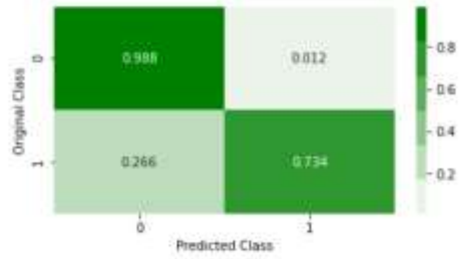
```
Percentage of misclassified points  2.07532667179093
-------------------------------------------------- Confusion matrix --------------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

Looks like the performance is **best with K = 1** only.

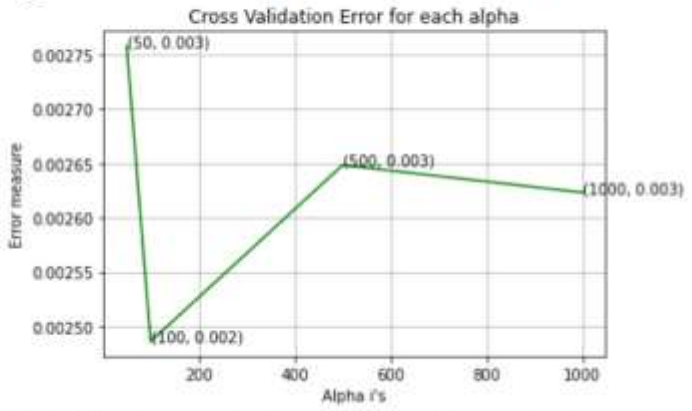
The major drawback of this approach is the **time taken during prediction**. KNN takes ~ 0 training time but the prediction time is significantly large. The retailer would have to decide whether this is alright (~ **14 minutes for 2602 queries**).

Random Forest

This is one of the most **advanced** machine learning models that handles all kinds of features well. Random sampling ensures **less susceptibility to overfitting**.

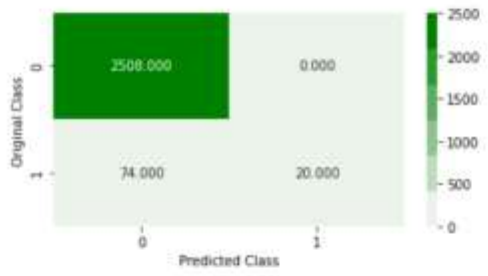
Once trained, these models can yield predictions very **fast** as it simply involves checking of some **if else** conditions.

```
100% 4/4 [33:18<00:00, 499.74s/it]

log_loss for c =  50 is 0.002757599897655697 ROC-AUC score is:  1.0
log_loss for c =  100 is 0.002486550422530597 ROC-AUC score is:  1.0
log_loss for c =  500 is 0.0026482814295678333 ROC-AUC score is:  1.0
log_loss for c =  1000 is 0.002623119172935434 ROC-AUC score is:  1.0
```

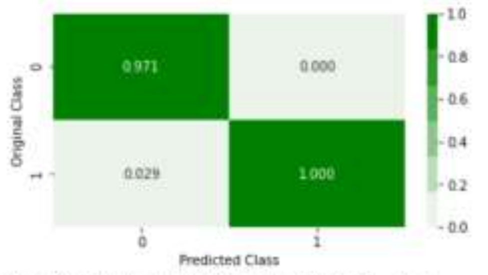


```
For values of best alpha =  100 The train log loss is: 0.0010154445729874213
For values of best alpha =  100 The cross validation log loss is: 0.002486550422530597
For values of best alpha =  100 The test log loss is: 0.13986254467077586
Percentage of misclassified points  2.8439661798616447
```

```
-------------------------------------------------- Confusion matrix --------------------------------------------------
```

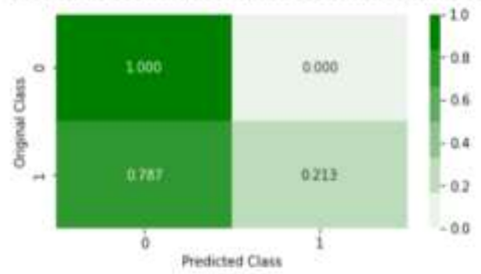


```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
```

```
------------------------------------------------ Recall matrix ------------------------------------------------
```



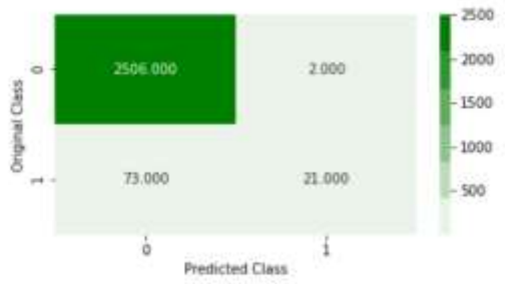
```
Sum of rows in recall matrix [1. 1.]
```

These results are disappointing. Although, there are absolutely **no False Positives,** a majority of **plugs are getting missed.** This kind of a model would not help our case.

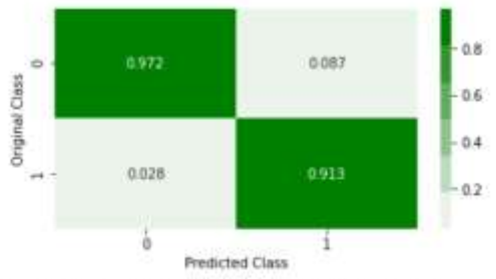
Just to make sure that this performance is not due to lack of training, let us train the model with **500 learners** instead of a **100**. The performance is given below:

```
Accuracy score: 0.9711760184473482
ROC-AUC: 0.9505582137161085
F1 score: 0.3589743589743589
```

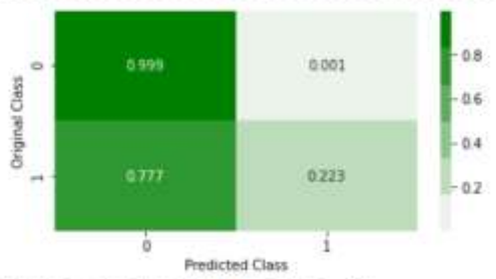
```
Percentage of misclassified points  2.8823981552651805
------------------------------------------------ Confusion matrix ------------------------------------------------
```



```
------------------------------------------------ Precision matrix ------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
------------------------------------------------ Recall matrix ------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

Clearly, there isn't much difference.

On the other hand, we can note that Random Forest is **biased towards non-plugs**. When we fitted Gaussian Naive Bayes on the same data, it captured almost all plugs and misclassified a lot of non-plugs. We can explore the idea of **stacking these two models** in the hope of balancing out both biases and getting a desirable result.

Before that, let us try some other tree-based models.

Gradient Boosting

Under this category, we will use the following 2 classifiers:

1. **XGBoost (XGBClassifier)**
2. **LightGBM (LGBMClassifier)**

Both of these classifiers work on the same principle of sequentially improving models where a new model learns from the previous model's **mistakes**.

They are often used with data having the order of features that we do and provide very strong results.

Prediction is also **fast** with these models.

XGBoost

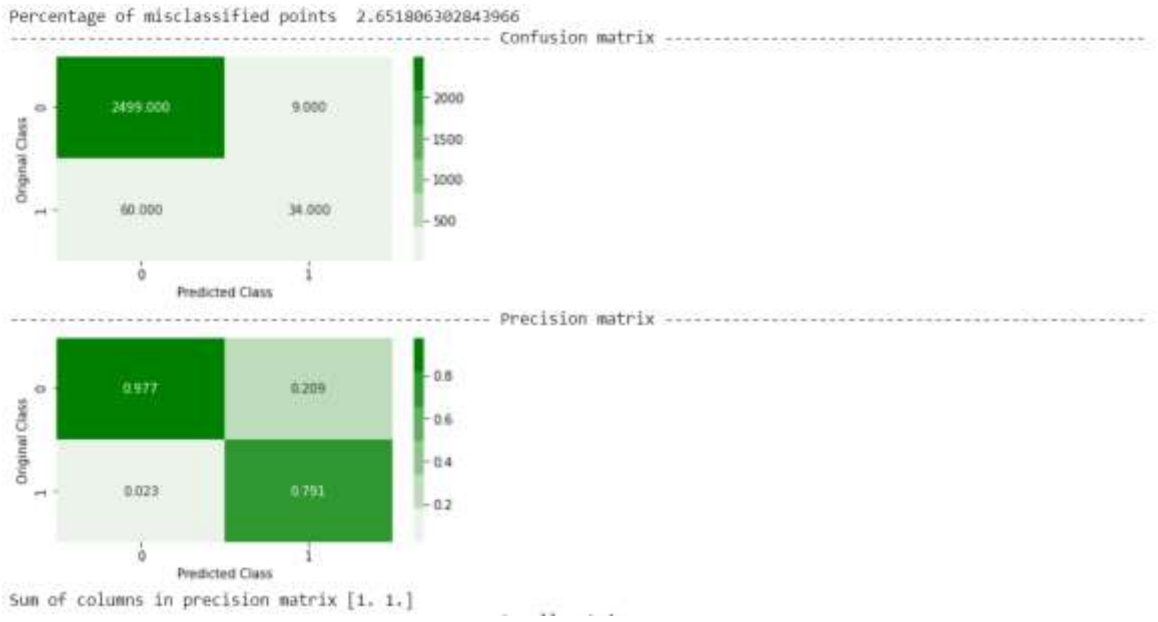
First, we will try and find the most suitable parameters using **RandomizedSearchCV**, as there are a lot of hyperparameters that can be tweaked.

The best parameters are:

```
{'subsample': 1, 'n_estimators': 100, 'max_depth': 10, 'learning_rate': 0.05, 'colsample_bytree': 0.3}
```

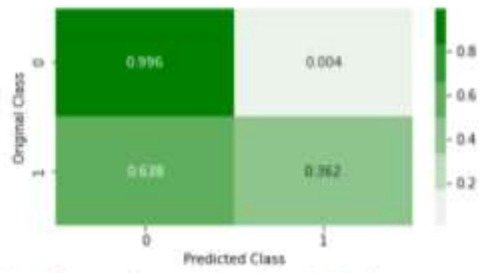
The performance of the model with the specifications given above is as follows:

```
Accuracy score: 0.9734819369715604
ROC-AUC: 0.9501043469408531
F1 score: 0.4963503649635036
```



```
Percentage of misclassified points  2.651806302843966
------------------------------------------------ Confusion matrix ------------------------------------------------
------------------------------------------------ Precision matrix ------------------------------------------------
Sum of columns in precision matrix [1. 1.]
```

```
--------------------------------------------------- Recall matrix ---------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

This result is **better** than Random Forest but is still not as good as the models we have seen before. It seems like a trend in tree-based classifiers to be **biased towards non-plugs**, at least the way we are training them.

LightGBM

Again, we will first find the best parameters of this model for this task.

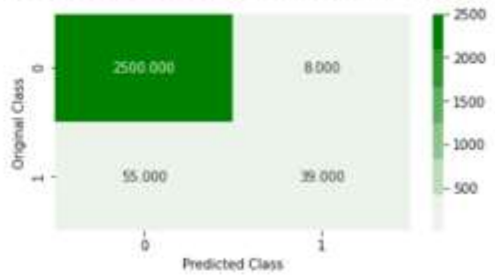
The best parameters are:

```
{'subsample': 1, 'n_estimators': 500, 'max_depth': 10, 'learning_rate':
0.05, 'colsample_bytree': 0.5}
```

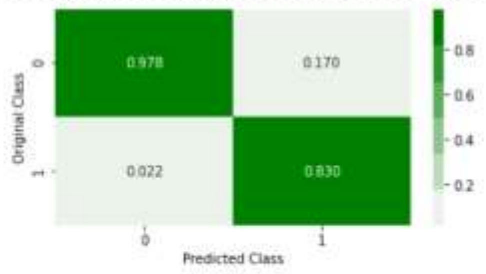
The performance of the model trained using the parameters given above, is as follows:

```
Accuracy score: 0.9757878554957725
ROC-AUC: 0.9521276595744681
F1 score: 0.553191489361702
```

```
Percentage of misclassified points  2.421214450422752
--------------------------------------------------- Confusion matrix ---------------------------------------------------
```

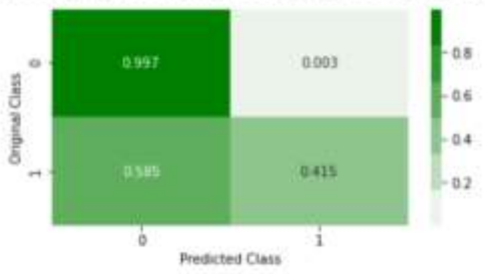


```
--------------------------------------------------- Precision matrix ---------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
```

```
--------------------------------------------------- Recall matrix ---------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

This result is **better** than XGBoost but is still not the best we have seen. We will have to think of different approaches.

Stacked Models

We will use the following combinations of models in the hope of getting better results:

1) **Gaussian Naive Bayes + Random Forest**
2) **Gaussian Naive Bayes + K Nearest Neighbors**
3) **Gaussian Naive Bayes + Support Vector Machines**
4) **Gaussian Naive Bayes + Deep Neural Network (Model 2)**
5) **Gaussian Naive Bayes + Deep Neural Network (Model 3)**

Gaussian Naive Bayes + Random Forest

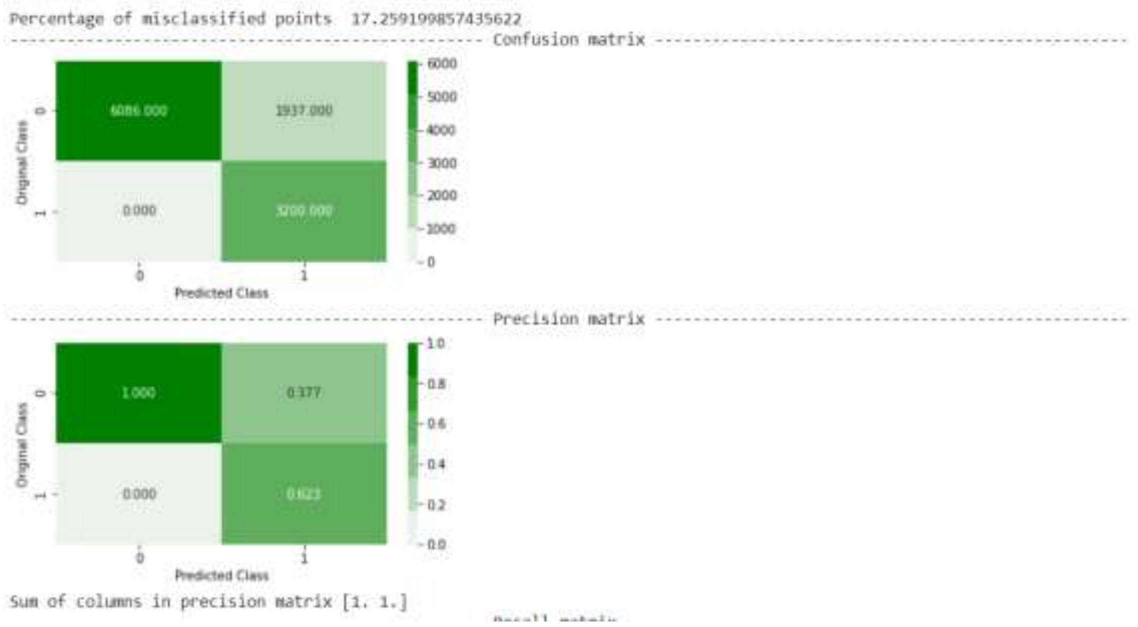
As discussed before, we have observed that GNB is biased towards plugs and RF is biased towards non-plugs.

In this stacked model, we will use **GNB** to predict plugs and will add the **predictions as a feature** to the original data.

This data will then be passed into a Random Forest classifier.

Our expectation is that the added feature would **prevent** the Random Forest Classifier from generating too many **False Negatives**.

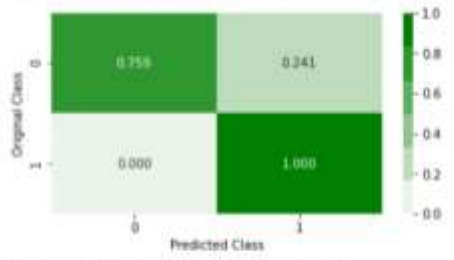
These are the confusion matrices generated after prediction by a GNB classifier trained on x_train with **var_smoothing = 0.01.**



```
Percentage of misclassified points  17.259199857435622
-------------------------------------------------- Confusion matrix --------------------------------------------------
```

```
-------------------------------------------------- Precision matrix --------------------------------------------------
```

```
Sum of columns in precision matrix [1. 1.]
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



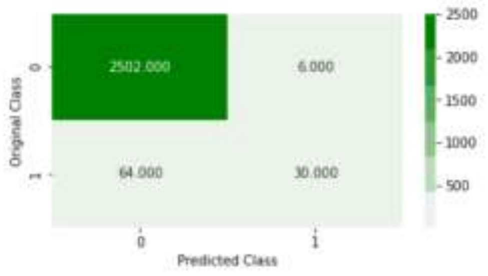
```
Sum of rows in recall matrix [1. 1.]
```

It is clear that **all the plugs** are being correctly classified. We have a **lot of false positives,** which RF will hopefully take care of.

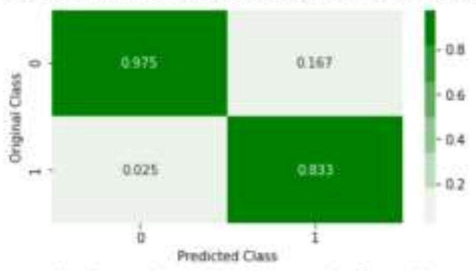
The next step is to generate **predictions** for **x_unb** and add those as a **feature** to the same. This new data is passed into **Random Forest,** and the predictions of this model are compared with y_unb. The details of the performance are shown below:

```
Accuracy score: 0.973097617217525
ROC-AUC: 0.9589992873867453
F1 score: 0.46153846153846l6
```

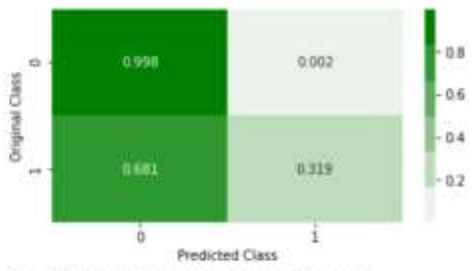
```
Percentage of misclassified points  2.690238278247502
-------------------------------------------------- Confusion matrix --------------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



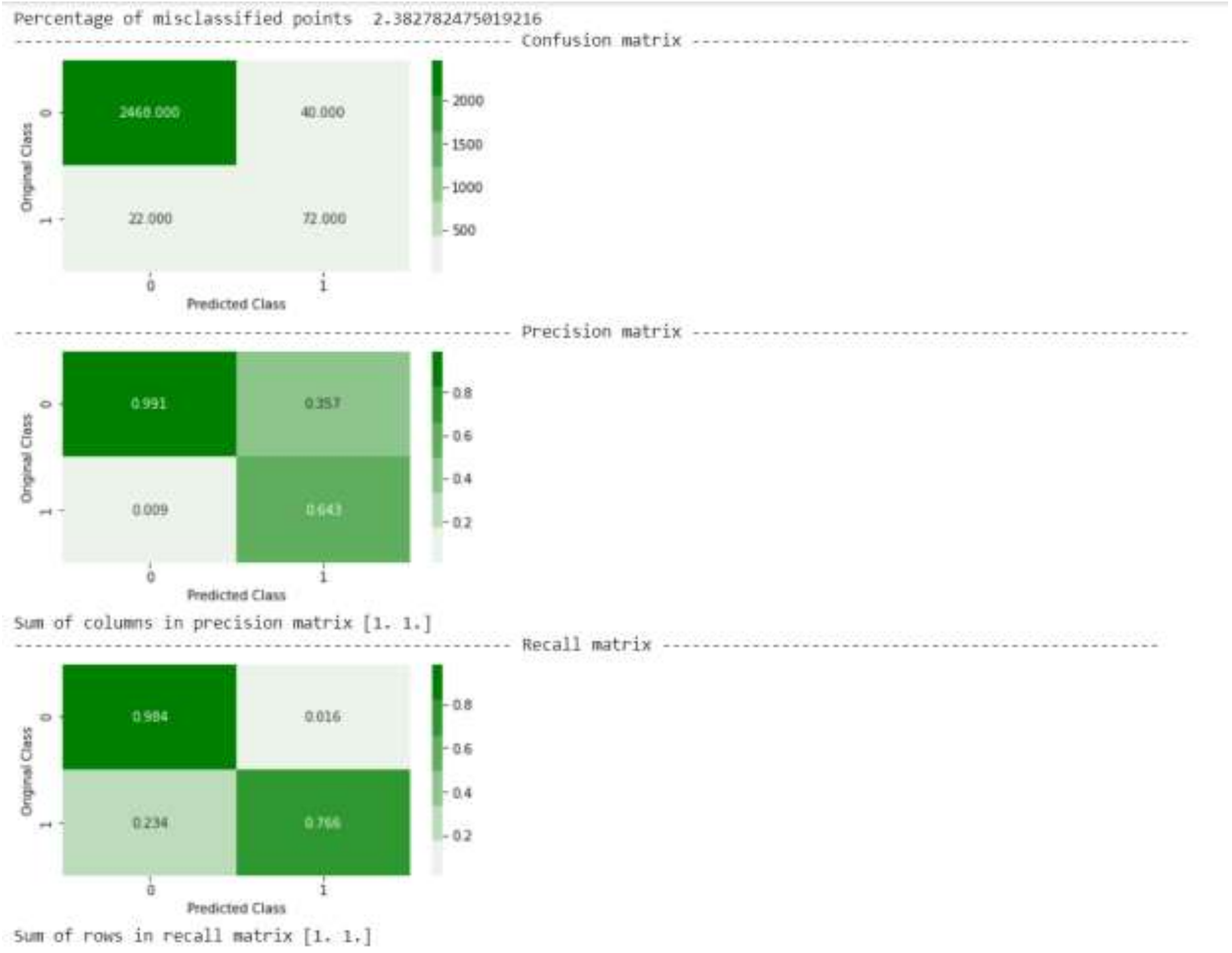
```
Sum of rows in recall matrix [1. 1.]
```

Although there is a **slight improvement** in the direction we had expected, as compared to when we pass the original data into RF, it is not significant enough for us to consider this as a solution. It is possible, though, that the task we were trying to achieve, was **successful for certain points**.

Gaussian Naive Bayes + K Nearest Neighbors

We will pass the data we used for the RF classifier in the previous case, into a KNeighborsClassifier with K = 1. This data contains the **GNB predictions** as a feature. The pipeline is the same with only a replacement of Random Forest by KNN. The performance is shown below.

```
Accuracy score: 0.9788624135280554
ROC_AUC: 0.8712799891411313
F1 score: 0.7208121827411168
```



```
Percentage of misclassified points  2.382782475019216
-------------------------------------------------- Confusion matrix --------------------------------------------------
-------------------------------------------------- Precision matrix --------------------------------------------------
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
Sum of rows in recall matrix [1. 1.]
```

The number of **False Negatives did not reduce and the number of False Positives increased.** It seems like some points that would have been correctly classified as non-plugs without the additional feature have been classified as plugs. However, this change is not useful for us.

Gaussian Naive Bayes + Support Vector Machines

The purpose of this stacking is to specifically solve the **problem of extra False Positives** generated by GNB.

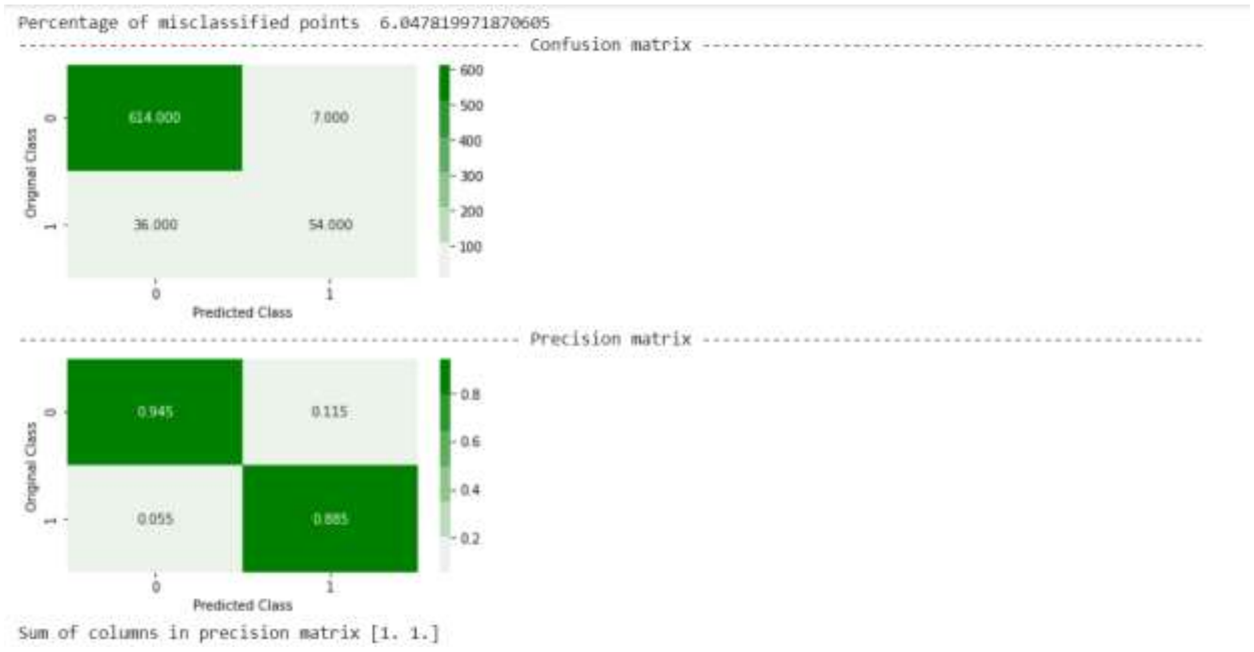
We will pass only the points (from the original data) that were **predicted to be plugs** by GNB, into SVC.

SVC is a good option when the number of training instances is of the order of the number of features.

We hope it will be able to learn a **distinction** between the False Positives and True positives generated by GNB. If it is able to **separate these two categories**, we will have a strong result.

We are training the SVC model on **x_sep_train** and evaluating the metrics on **x_sep_unb:**

```
Accuracy score: 0.939521800281294
ROC-AUC: 0.9365897298264448
F1 score: 0.7152317880794702
```

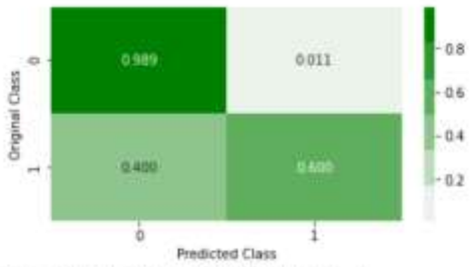


```
Percentage of misclassified points  6.047819971870605
-------------------------------------------------- Confusion matrix --------------------------------------------------
```

```
-------------------------------------------------- Precision matrix --------------------------------------------------
```

```
Sum of columns in precision matrix [1. 1.]
```

```
-------------------------------------------------- Recall matrix --------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

Although this combination has **eliminated most False positives**, it is resulting in a **40/94 False negatives.** Overall, this model will not be of much help to us.

Deep Learning models

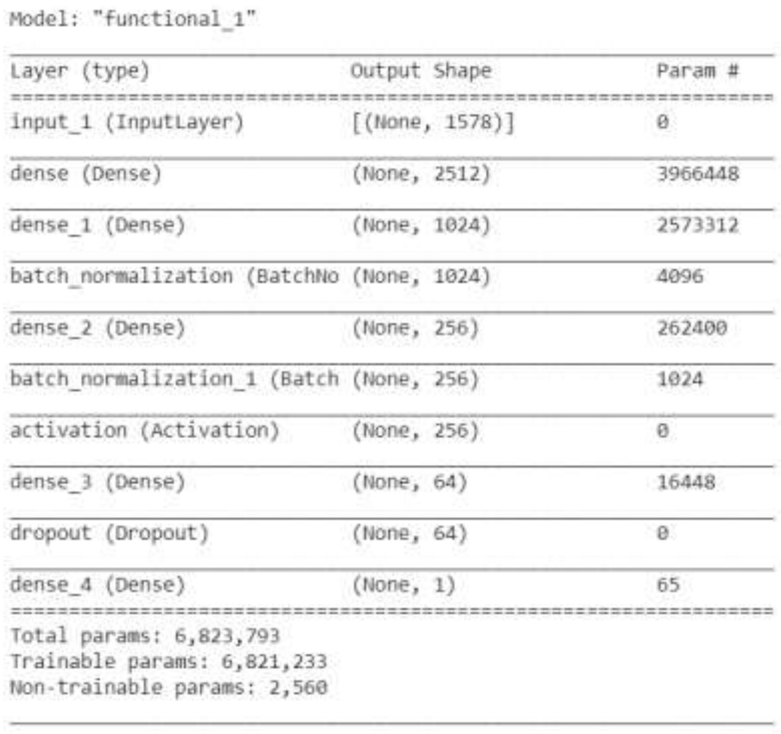
Model 1

This model uses the original data with product spaces as **one-hot-encoded features** in order to predict if an item-space pair is a plug or not.

The loss used is **'binary cross-entropy'**. The metric used is '**accuracy**'.

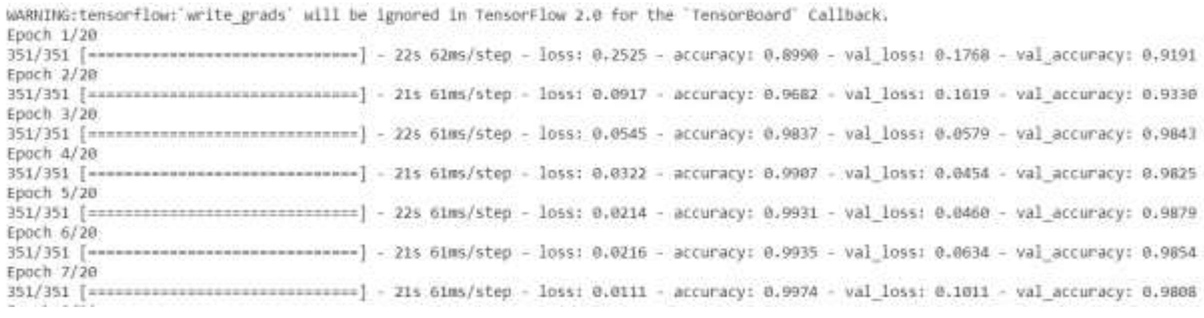
The optimizer being used is **SGD (learning rate = 0.01)**.

The model architecture/parameter count is shown below:

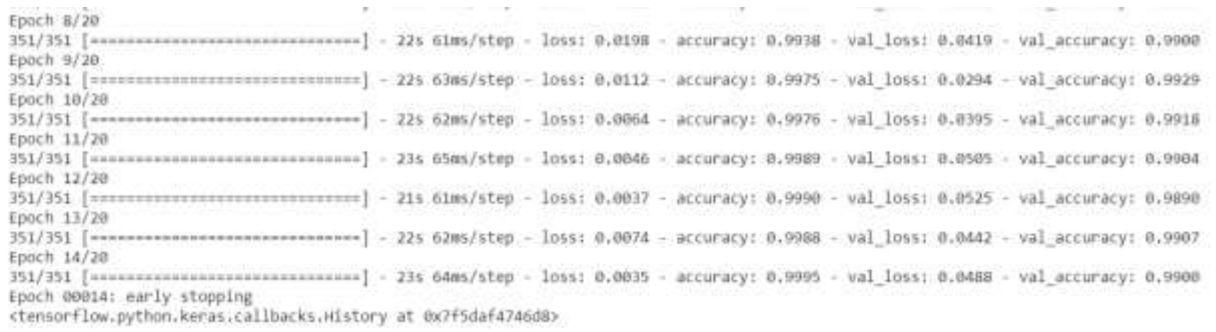


```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 1578)]            0
_________________________________________________________________
dense (Dense)                (None, 2512)              3966448
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              2573312
_________________________________________________________________
batch_normalization (BatchNo (None, 1024)              4096
_________________________________________________________________
dense_2 (Dense)              (None, 256)               262400
_________________________________________________________________
batch_normalization_1 (Batch (None, 256)               1024
_________________________________________________________________
activation (Activation)      (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                16448
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 65
=================================================================
Total params: 6,823,793
Trainable params: 6,821,233
Non-trainable params: 2,560
_________________________________________________________________
```

The following are the logs of training:



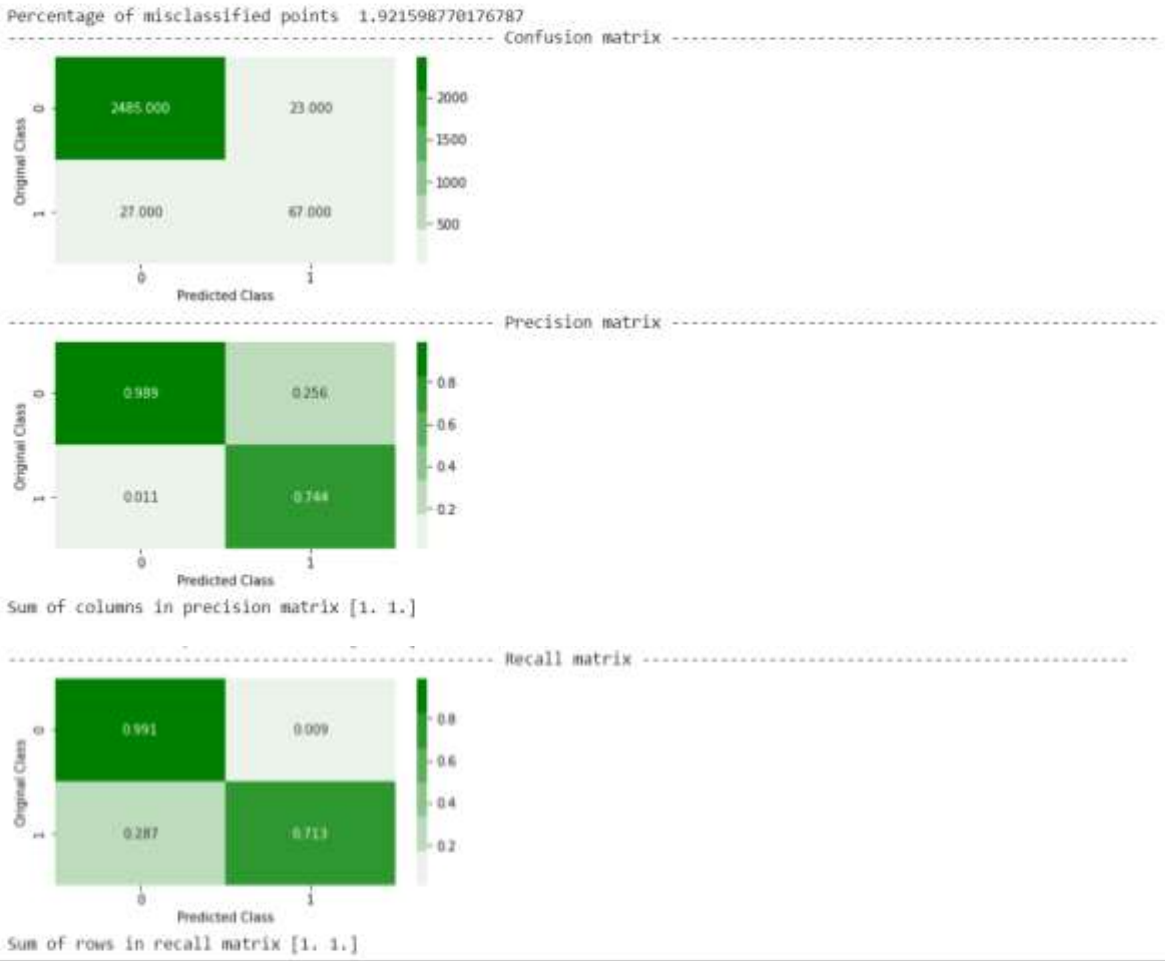
```
WARNING:tensorflow:`write_grads` will be ignored in TensorFlow 2.0 for the `TensorBoard` Callback.
Epoch 1/20
351/351 [==============================] - 22s 62ms/step - loss: 0.2525 - accuracy: 0.8990 - val_loss: 0.1768 - val_accuracy: 0.9191
Epoch 2/20
351/351 [==============================] - 21s 61ms/step - loss: 0.0917 - accuracy: 0.9682 - val_loss: 0.1619 - val_accuracy: 0.9330
Epoch 3/20
351/351 [==============================] - 22s 61ms/step - loss: 0.0545 - accuracy: 0.9837 - val_loss: 0.0579 - val_accuracy: 0.9843
Epoch 4/20
351/351 [==============================] - 21s 61ms/step - loss: 0.0322 - accuracy: 0.9907 - val_loss: 0.0454 - val_accuracy: 0.9825
Epoch 5/20
351/351 [==============================] - 22s 61ms/step - loss: 0.0214 - accuracy: 0.9931 - val_loss: 0.0460 - val_accuracy: 0.9879
Epoch 6/20
351/351 [==============================] - 21s 61ms/step - loss: 0.0216 - accuracy: 0.9935 - val_loss: 0.0634 - val_accuracy: 0.9854
Epoch 7/20
351/351 [==============================] - 21s 61ms/step - loss: 0.0111 - accuracy: 0.9974 - val_loss: 0.1011 - val_accuracy: 0.9808
```



```
Epoch 8/20
351/351 [==============================] - 22s 61ms/step - loss: 0.0198 - accuracy: 0.9938 - val_loss: 0.0419 - val_accuracy: 0.9900
Epoch 9/20
351/351 [==============================] - 22s 63ms/step - loss: 0.0112 - accuracy: 0.9975 - val_loss: 0.0294 - val_accuracy: 0.9929
Epoch 10/20
351/351 [==============================] - 22s 62ms/step - loss: 0.0064 - accuracy: 0.9976 - val_loss: 0.0395 - val_accuracy: 0.9918
Epoch 11/20
351/351 [==============================] - 23s 65ms/step - loss: 0.0046 - accuracy: 0.9989 - val_loss: 0.0505 - val_accuracy: 0.9904
Epoch 12/20
351/351 [==============================] - 21s 61ms/step - loss: 0.0037 - accuracy: 0.9990 - val_loss: 0.0525 - val_accuracy: 0.9890
Epoch 13/20
351/351 [==============================] - 22s 62ms/step - loss: 0.0074 - accuracy: 0.9988 - val_loss: 0.0442 - val_accuracy: 0.9907
Epoch 14/20
351/351 [==============================] - 23s 64ms/step - loss: 0.0035 - accuracy: 0.9995 - val_loss: 0.0488 - val_accuracy: 0.9900
Epoch 00014: early stopping
<tensorflow.python.keras.callbacks.History at 0x7f5daf4746d8>
```

Clearly, the performance of the model on the train and validation sets is very good. The performance of this model on **x_unb** is as follows:

```
Accuracy score: 0.9807840122982321
ROC-AUC: 0.8517976517696563
F1 score: 0.7282608695652174
```



```
Percentage of misclassified points  1.921598770176787
-------------------------------------------------- Confusion matrix --------------------------------------------------
```

```
-------------------------------------------------- Precision matrix --------------------------------------------------
```

```
Sum of columns in precision matrix [1. 1.]
```

```
-------------------------------------------------- Recall matrix --------------------------------------------------
```

```
Sum of rows in recall matrix [1. 1.]
```

This is a very **promising result.**

All metrics are amongst the **best encountered yet.**

Let us explore a little more in order to improve these metrics even more.

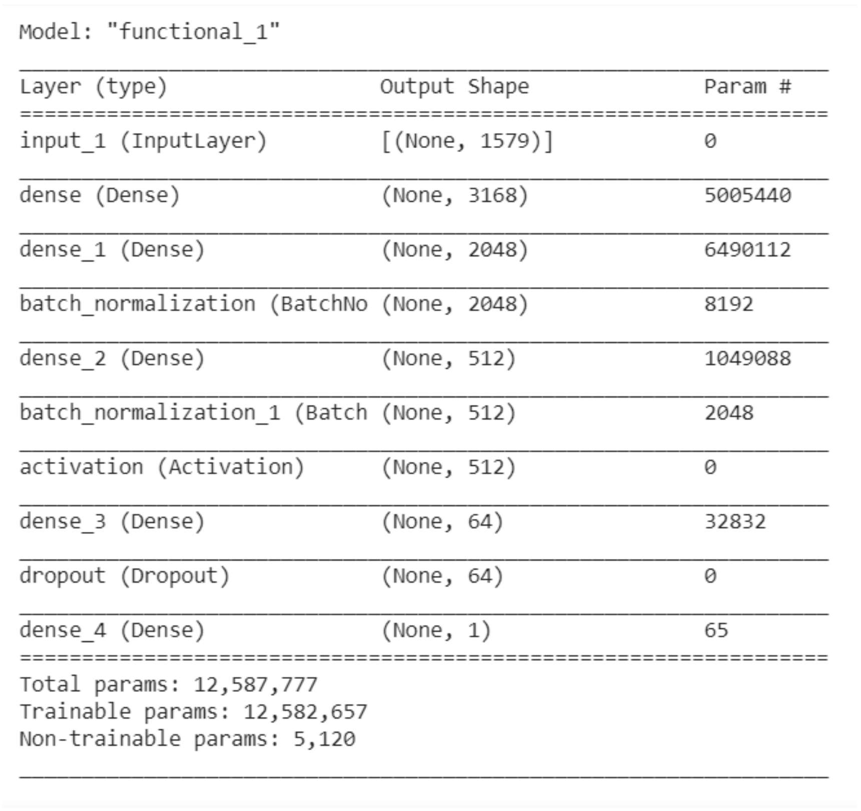
Gaussian Naive Bayes + Model 2

This model will use the data which has the feature containing the **predictions made by GNB**.

The loss is again '**binary cross-entropy**'. The metric is again '**accuracy**'.

The optimizer used is **SGD (learning rate = 0.01)**.

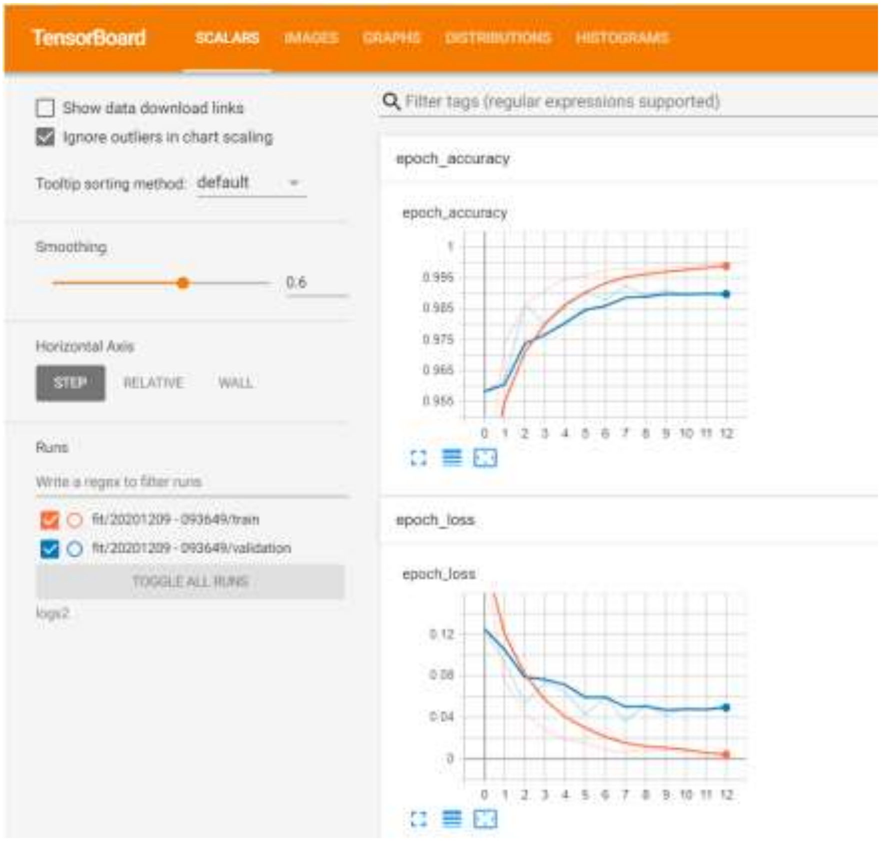
The model architecture is given below:



```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 1579)]            0
_________________________________________________________________
dense (Dense)                (None, 3168)              5005440
_________________________________________________________________
dense_1 (Dense)              (None, 2048)              6490112
_________________________________________________________________
batch_normalization (BatchNo (None, 2048)              8192
_________________________________________________________________
dense_2 (Dense)              (None, 512)               1049088
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
activation (Activation)      (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                32832
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 65
=================================================================
Total params: 12,587,777
Trainable params: 12,582,657
Non-trainable params: 5,120
_________________________________________________________________
```

The first layer after the input has **twice as many neurons** as the input, which works well in practice. Size of the next layer is the **closest power of 2** which is less than the value in the first hidden layer, which also works well in practice. All subsequent layers also have number of neurons in decreasing power of 2. There is one output neuron with activation as **sigmoid**.

Given below are the graphs for training and validation accuracy and loss. The validation accuracy reaches ~0.99, which is very good.

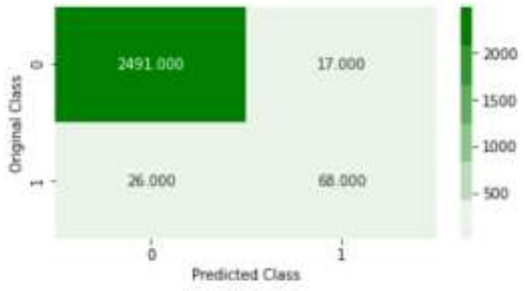


Show data download links
[✓] Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing: 0.6
Horizontal Axis: [✓ STEP] RELATIVE WALL
Runs
[✓] fit/20201209 - 093649/train
[✓] fit/20201209 - 093649/validation

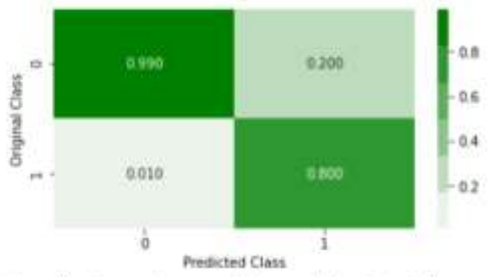
The performance of this model on **x_nb_unb** is hown below:

```
Accuracy score: 0.9834742505764796
ROC-AUC: 0.9604923818249687
F1 score: 0.7597765363128491
```

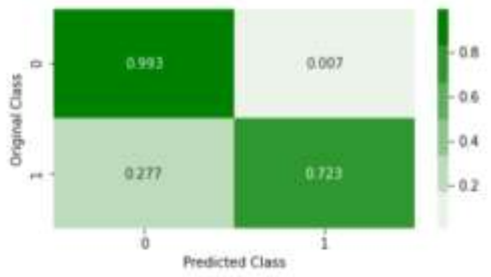
```
Percentage of misclassified points  1.6525749423520368
--------------------------------------------------- Confusion matrix --------------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

Best result so far!

Even though the we are missing more plugs than in some cases, it is possible to **tweak the neural network** to adjust that value in the right direction (observed to be true). Our **overall** accuracy, roc_auc_score, and f1 score are the **highest**.

This pipeline would also be **fast** considering less training and prediction time for both models involved.

There is one observation at this point, though. Most of the models (that are performing well, are misclassifying around **22-28 plugs**. One could explore whether these are the **same points**. If that is true, further study can be done into why all models are getting confused with these points. Let us try to confirm this hypothesis.

```
# Printing the set of
print(fn_dnn)
print(fn_knn)
# Printing the common False Negatives
print(fn_dnn.intersection(fn_knn))
```

```
2602 2602
{2308, 778, 1419, 1170, 1427, 1682, 1178, 1439, 1314, 509, 2474, 2355, 2488, 960, 2243, 2116, 2372, 1606, 719, 2384, 1240, 861, 1506, 1635, 624, 2303}
{642, 2308, 778, 1419, 1170, 1427, 1682, 1178, 1694, 1439, 2355, 2488, 1474, 2243, 2116, 2372, 75, 719, 2384, 1240, 861, 1506, 1635, 2303}
{1506, 2243, 2308, 2116, 2372, 1635, 2303, 778, 1419, 861, 719, 2384, 1170, 1427, 1682, 2355, 2488, 1178, 1240, 1439}
```

Clearly, there is a lot of **overlap** between these two sets, which suggests that there is some **common property** of these items/spaces that is confusing the models. **Deeper study** can be conducted on this topic.

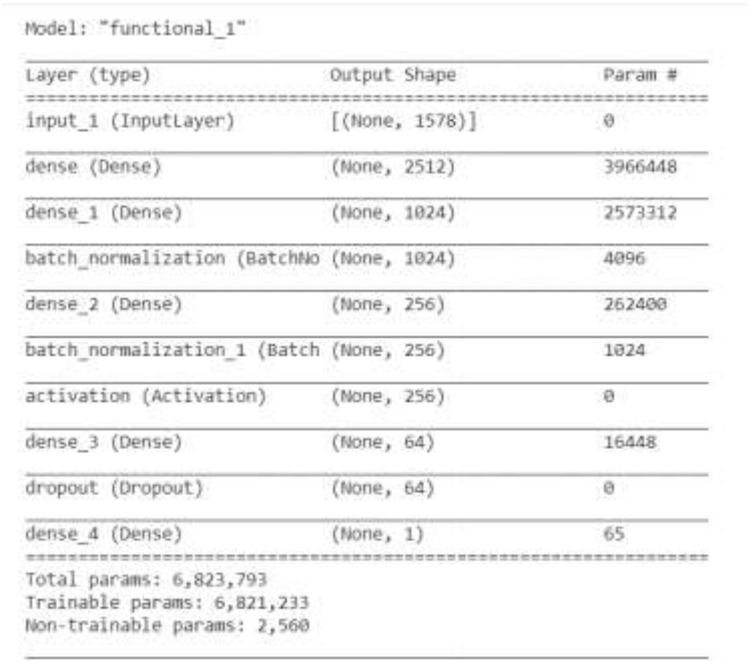
Gaussian Naive Bayes + Model 3

This model is going to try and **separate** the non-plugs from the plugs among all the points predicted as plugs by **GNB**. It will only take relevant data points as input, the same as those taken for GNB + SVC.

The loss used is **'binary cross-entropy'**. The metric used is **'accuracy'**.

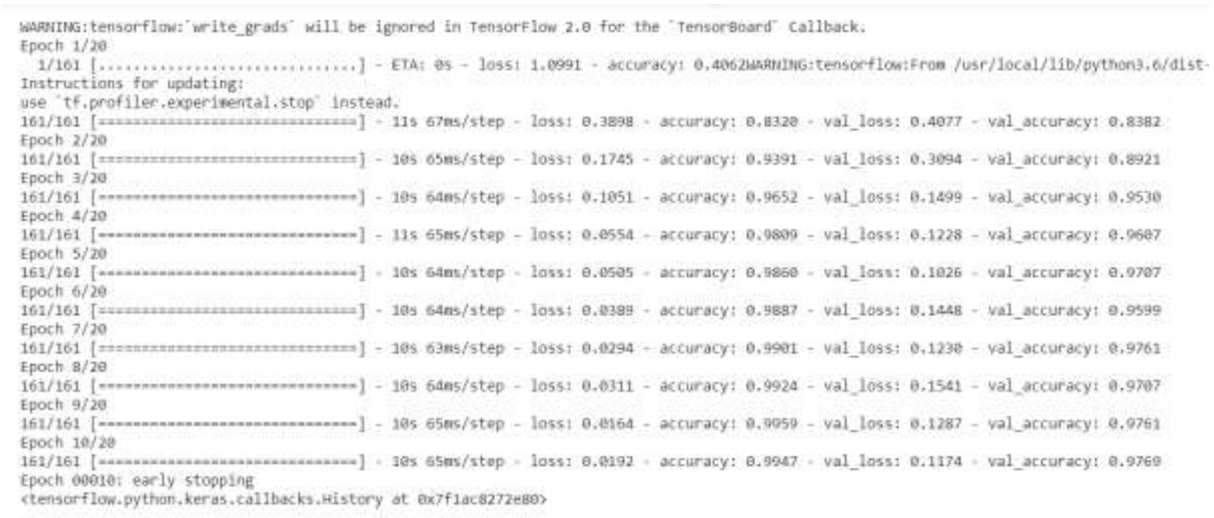
The optimizer is **SGD (learning rate = 0.01).**

The model architecture is given below:



```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 1578)]            0         
_________________________________________________________________
dense (Dense)                (None, 2512)              3966448   
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              2573312   
_________________________________________________________________
batch_normalization (BatchNo (None, 1024)              4096      
_________________________________________________________________
dense_2 (Dense)              (None, 256)               262400    
_________________________________________________________________
batch_normalization_1 (Batch (None, 256)               1024      
_________________________________________________________________
activation (Activation)      (None, 256)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 64)                16448     
_________________________________________________________________
dropout (Dropout)            (None, 64)                0         
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 65        
=================================================================
Total params: 6,823,793
Trainable params: 6,821,233
Non-trainable params: 2,560
_________________________________________________________________
```

This architecture was chosen after experimentation with variations. The training logs are shown below:

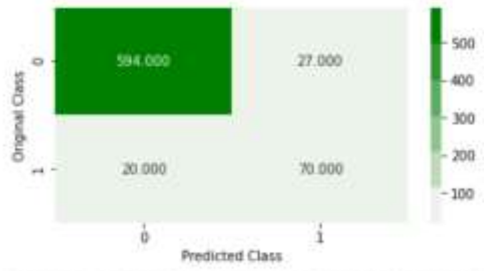


```
WARNING:tensorflow:`write_grads` will be ignored in TensorFlow 2.0 for the `TensorBoard` Callback.
Epoch 1/20
  1/161 [..............................] - ETA: 0s - loss: 1.0991 - accuracy: 0.4062WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
Instructions for updating:
use `tf.profiler.experimental.stop` instead.
161/161 [==============================] - 11s 67ms/step - loss: 0.3898 - accuracy: 0.8320 - val_loss: 0.4077 - val_accuracy: 0.8382
Epoch 2/20
161/161 [==============================] - 10s 65ms/step - loss: 0.1745 - accuracy: 0.9391 - val_loss: 0.3094 - val_accuracy: 0.8921
Epoch 3/20
161/161 [==============================] - 10s 64ms/step - loss: 0.1051 - accuracy: 0.9652 - val_loss: 0.1499 - val_accuracy: 0.9530
Epoch 4/20
161/161 [==============================] - 11s 65ms/step - loss: 0.0554 - accuracy: 0.9809 - val_loss: 0.1228 - val_accuracy: 0.9607
Epoch 5/20
161/161 [==============================] - 10s 64ms/step - loss: 0.0505 - accuracy: 0.9860 - val_loss: 0.1026 - val_accuracy: 0.9707
Epoch 6/20
161/161 [==============================] - 10s 64ms/step - loss: 0.0389 - accuracy: 0.9887 - val_loss: 0.1448 - val_accuracy: 0.9599
Epoch 7/20
161/161 [==============================] - 10s 63ms/step - loss: 0.0294 - accuracy: 0.9901 - val_loss: 0.1230 - val_accuracy: 0.9761
Epoch 8/20
161/161 [==============================] - 10s 64ms/step - loss: 0.0311 - accuracy: 0.9924 - val_loss: 0.1541 - val_accuracy: 0.9707
Epoch 9/20
161/161 [==============================] - 10s 65ms/step - loss: 0.0164 - accuracy: 0.9959 - val_loss: 0.1287 - val_accuracy: 0.9761
Epoch 10/20
161/161 [==============================] - 10s 65ms/step - loss: 0.0192 - accuracy: 0.9947 - val_loss: 0.1174 - val_accuracy: 0.9769
Epoch 00010: early stopping
<tensorflow.python.keras.callbacks.History at 0x7f1ac8272e80>
```

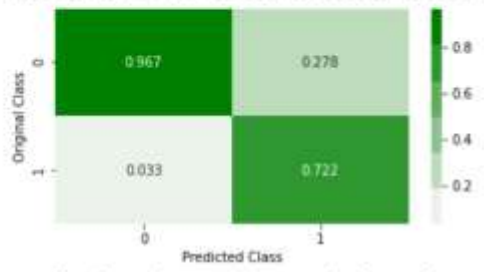
The performance of this model on **x_sep_unb** is shown below:

```
Accuracy score: 0.9338959212376934
ROC-AUC: 0.9372249060654858
F1 score: 0.7486631016042781
```

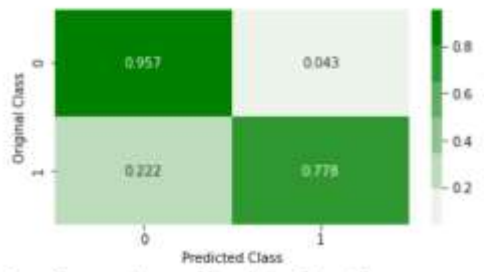
```
Percentage of misclassified points  6.610407876230662
------------------------------------------------- Confusion matrix -------------------------------------------------
```



```
------------------------------------------------- Precision matrix -------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
------------------------------------------------- Recall matrix -------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

The performance of this model is **very good**.

By and large, it is **able to separate** the points as required. However, the overall result is not significantly better than model 2. The misclassified plugs are **24/94** and misclassified non-plugs are **27/2508**.

This stacking can also be considered as a **solution**.

Now, we will try the second approach for using the data

Model 4

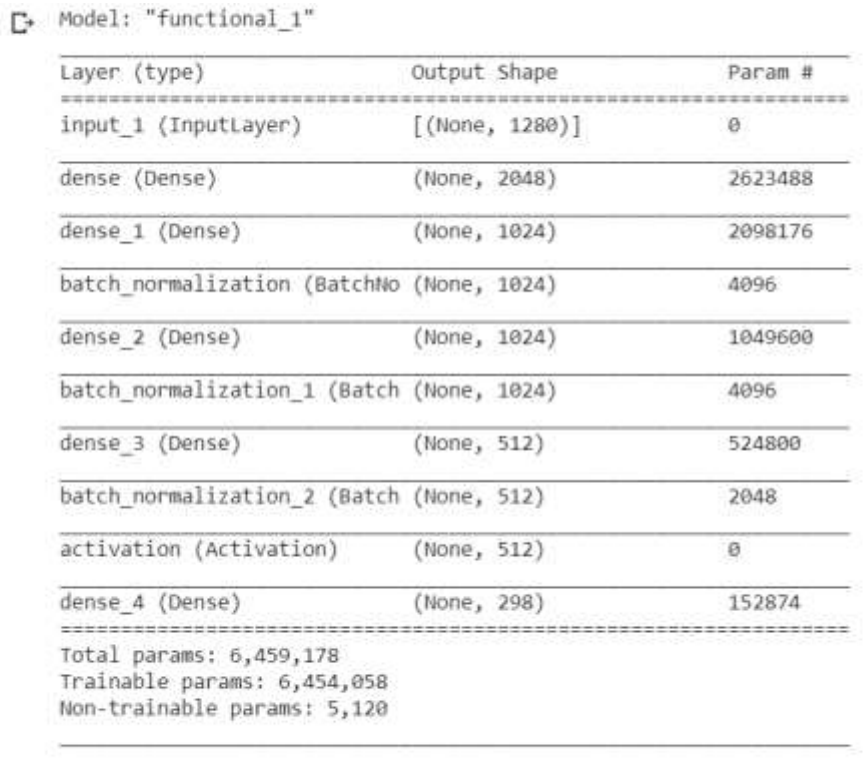
This model will take in the image feature embeddings as input, one-hot-encoded **product spaces as output**, and try to learn the **item -> product_space mapping**.

We will then predict the product space for each query point in the test set and use the plug labels to determine whether the prediction is permissible.

For this, the training (and cv) data we will use will consist of the first **11000 non-plugs**. This data will be **up sampled** to twice the size to ensure that no class has only 1 training instance and for better training.

The test data will consist of the all (471) plugs and remaining non-plugs.

Given below is the model architecture:



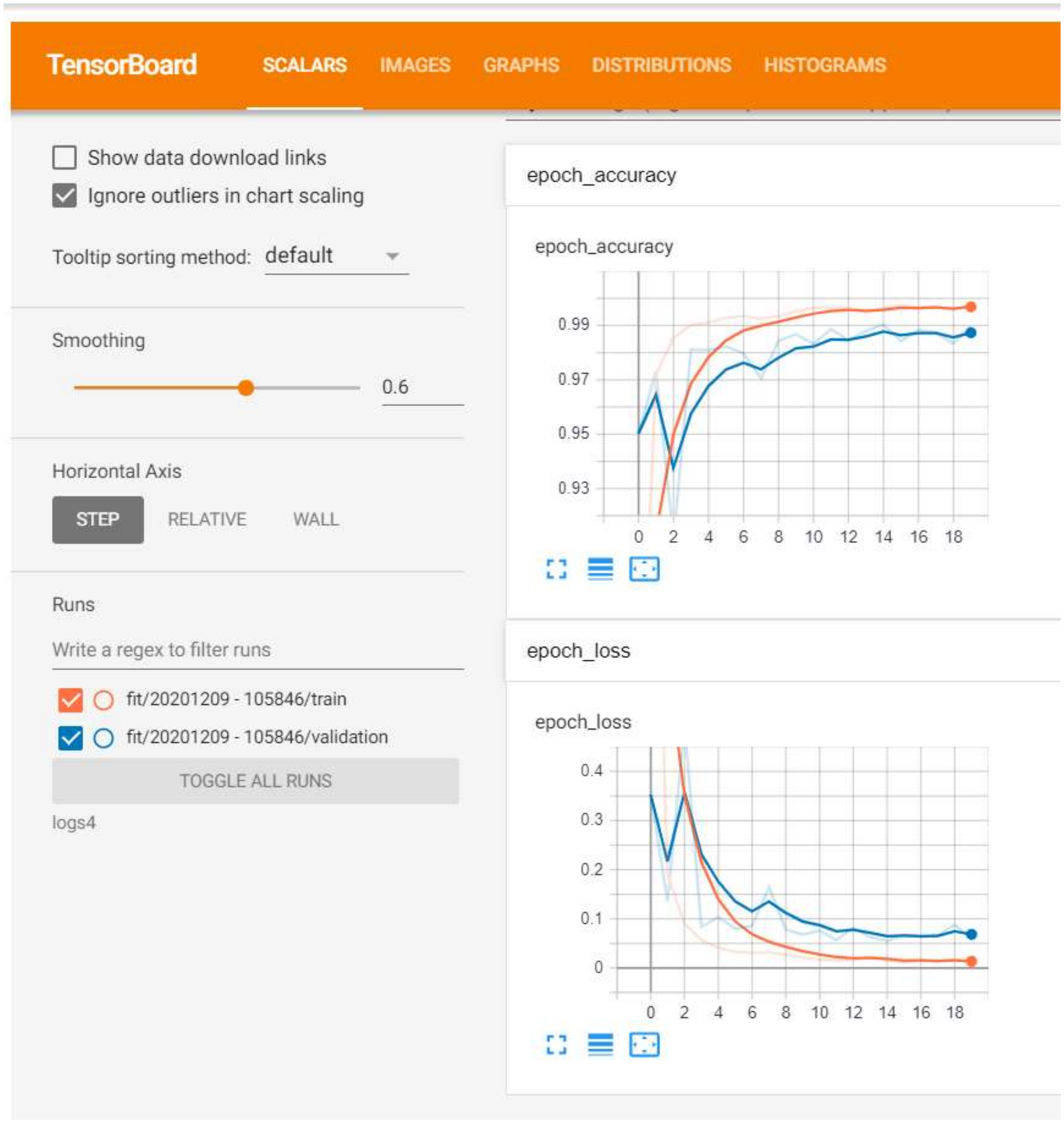
```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 1280)]            0
_________________________________________________________________
dense (Dense)                (None, 2048)              2623488
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              2098176
_________________________________________________________________
batch_normalization (BatchNo (None, 1024)              4096
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
batch_normalization_1 (Batch (None, 1024)              4096
_________________________________________________________________
dense_3 (Dense)              (None, 512)               524800
_________________________________________________________________
batch_normalization_2 (Batch (None, 512)               2048
_________________________________________________________________
activation (Activation)      (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 298)               152874
=================================================================
Total params: 6,459,178
Trainable params: 6,454,058
Non-trainable params: 5,120
_________________________________________________________________
```

The first hidden layer contains number of neurons equal to the **power of 2** which is less than twice the number of inputs. The rest of the layers have number of neurons in reducing powers of two, and the last layer has **298 outputs** (number of product spaces) with activation as **SoftMax**.

The loss used is '**categorical cross-entropy**'. The metric used is '**accuracy**'.

The optimizer used is **Adam (learning rate = 0.001).**

The training and cross-validation logs of this model can be found below:



Show data download links
✓ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP RELATIVE WALL

Runs

Write a regex to filter runs

✓ fit/20201209 - 105846/train

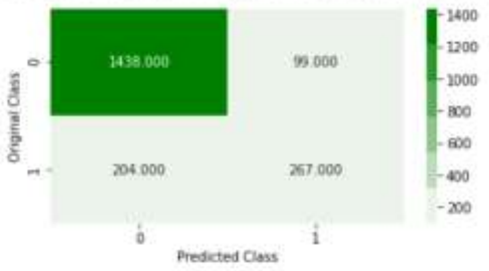
✓ fit/20201209 - 105846/validation

logs4

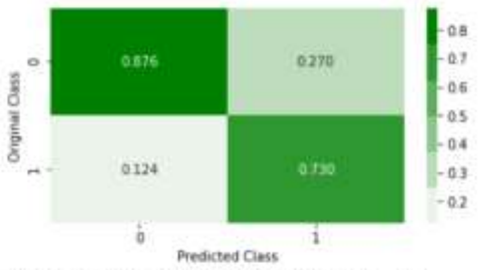
The accuracy of this model seems to be **very high**. We can assume that it has **learnt the mapping** of item -> product space, adequately. Here are the results on the test data:

```
Accuracy score: 0.8491035856573705
ROC-AUC: 0.7512338951303101
F1 score: 0.6379928315412187
```

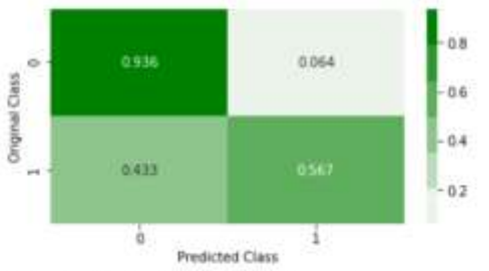
```
Percentage of misclassified points  15.08964143426295
-------------------------------------------------- Confusion matrix --------------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix ------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

Not so thrilled.

The ratio of **misclassified non-plugs** is much higher than that in the previous approach and the **Recall is only ~0.57**.

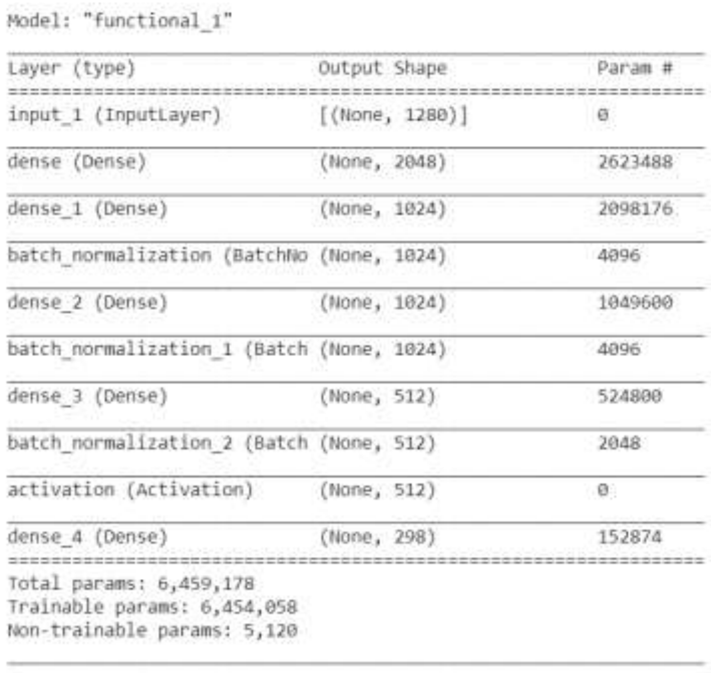
We can try and increase the number of training instances.

Model 5

Now, we will pass **almost all non-plugs** into the model and will detect whether the predicted product spaces are different from those given for plugs.

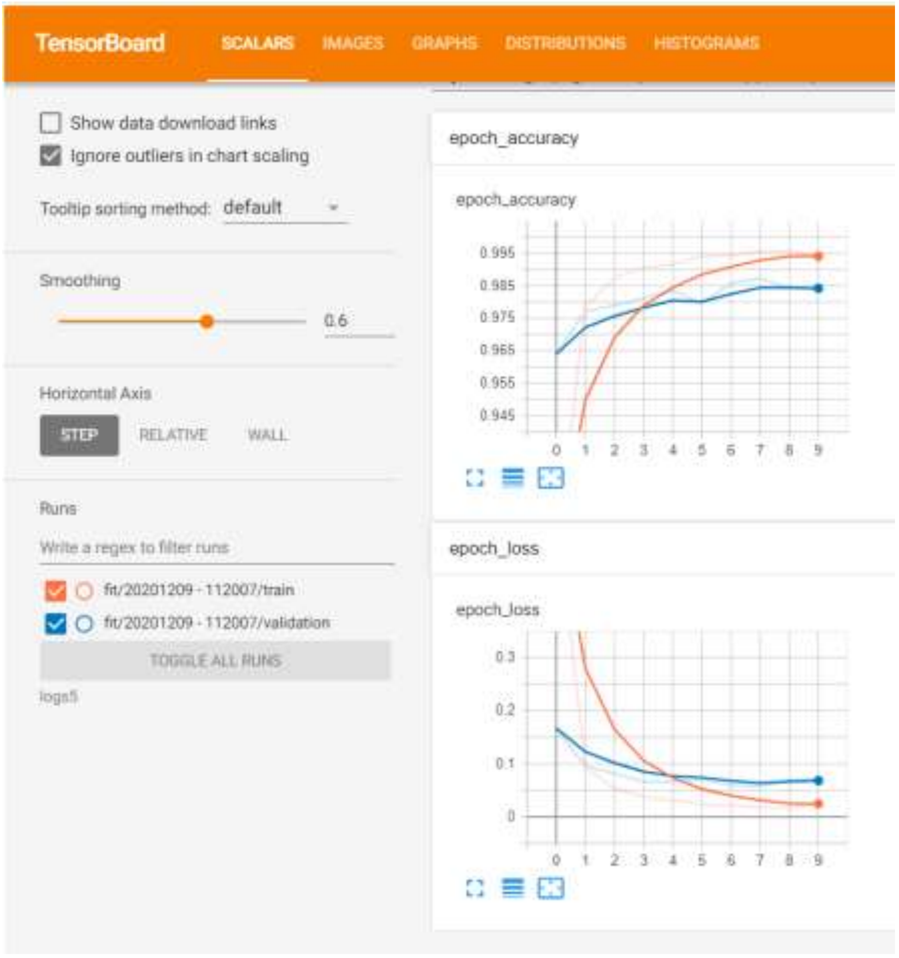
100 non-plugs are still kept in the test set as a **sanity check** for whether the model is actually predicting the correct product space properly.

The model architecture, given below, is identical to the one used in **Model 4**.



```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 1280)]            0         
_________________________________________________________________
dense (Dense)                (None, 2048)              2623488   
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              2098176   
_________________________________________________________________
batch_normalization (BatchNo (None, 1024)              4096      
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600   
_________________________________________________________________
batch_normalization_1 (Batch (None, 1024)              4096      
_________________________________________________________________
dense_3 (Dense)              (None, 512)               524800    
_________________________________________________________________
batch_normalization_2 (Batch (None, 512)               2048      
_________________________________________________________________
activation (Activation)      (None, 512)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 298)               152874    
=================================================================
Total params: 6,459,178
Trainable params: 6,454,058
Non-trainable params: 5,120
_________________________________________________________________
```

The training and cross validation curves for CCE loss and accuracy can be seen below:

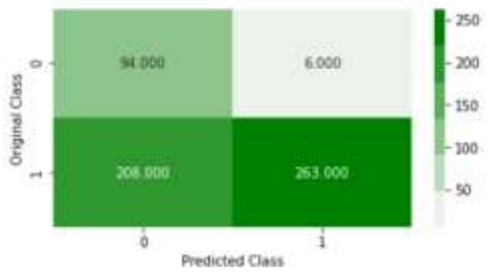


☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis: STEP RELATIVE WALL
Runs
Write a regex to filter runs
☑ fit/20201209 - 112007/train
☑ fit/20201209 - 112007/validation
logs5

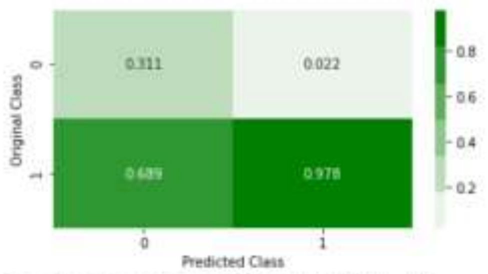
Again, the accuracy of the model is **high enough**. Given below, is the performance of the model on the test set:

```
Accuracy score: 0.6252189141856392
ROC-AUC: 0.7491932059447983
F1 score: 0.7108108108108109
```

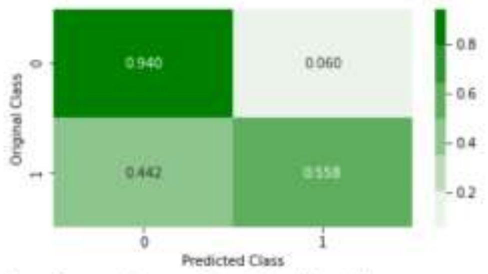
```
Percentage of misclassified points  37.478108581436075
-------------------------------------------------- Confusion matrix --------------------------------------------------
```



```
-------------------------------------------------- Precision matrix --------------------------------------------------
```



```
Sum of columns in precision matrix [1. 1.]
-------------------------------------------------- Recall matrix --------------------------------------------------
```



```
Sum of rows in recall matrix [1. 1.]
```

The accuracy of classification of non-plugs has **improved** understandably. However, the accuracy for plugs has shown a slight **decrease**.

This means that the product space given in the data is being predicted by the model as well. A hypothesis arises that the model is also learning the **same confusion a person would be facing** who has misplaced the item. This can be tested through a **deeper study of plugs**.

Conclusions

1) We have tried to solve the problem by using the data in two ways. As per our observations, using the **product spaces as one-hot-encoded features is better**.
2) Even after learning a proper mapping between items and product spaces (evident from the accuracy), **type-II error** is prevalent in the predictions. This could be due to the fact that misclassified items are **very similar** to the items found in the product spaces that the neural network is predicting for them. This might explain why they were **wrongly placed** by humans in the first place.
3) There seems to be a **trade-off** between the **Precision** and **Recall** for this problem, which is concluded after seeing similar results for multiple models.
4) Some plugs seem to be especially susceptible to the **type-II error**.
5) Overall, multiple models are giving **satisfactory results**, individual models can be selected depending on the requirements of the store manager. These models are:
    a) **KNeighborsClassifier with K = 1**. If the retailer does not have a requirement for very low latency, this model can give good predictions overall.
    b) **Support Vector Classifier with C = 1000.** If the retailer does not mind missing **<= 30%** of the plugs, this solution can be used.
    c) **Support Vector Classifier with C = 1.** If the retailer does not mind checking many False positives in order to have a Recall of ~ **86%,** this solution can be used.
    d) **Gaussian Naïve Bayes (var_smoothing = 0.01) + Model 2.** This is a good solution overall and can be used without reservations apart from the fact that the number of False Negatives is not the best we have seen.
    e) **Gaussian Naïve Bayes (var_smoothing = 0.01) + Model 3.** The same goes for this solution. However, there are a few extra False positives in this solution compared to the last.

Further Exploration

- As observed above, certain plugs seem to be extra susceptible to misclassification. Given exact images for the item and information about product spaces, one can study why the **type-II error** is arising, in case of both approaches of using the data.
- Different degrees of **up sampling** can be tried out for different models.
- More diverse **compound architectures** can be explored with better computational resources.
- Different architectures for **image classification** can be used for generating feature embeddings or directly mapping to product spaces. Various techniques such as **augmentation** and **pre-processing** can be used to selectively reduce errors.
- A **multi-input** convolutional and dense architecture can be used to combine image features and product space features in the same model to predict plugs directly.

References

- **Applied Machine Learning Course (https://www.appliedaicourse.com/#)**
- **Hands on Machine Learning with Scikit-Learn, Keras & Tensorflow, by Aurelien Geron**
- **https://cloud.google.com/solutions/machine-learning/overview-extracting-and-serving-feature-embeddings-for-machine-learning#introduction**
- **https://rom1504.medium.com/image-embeddings-ed1b194d113e**