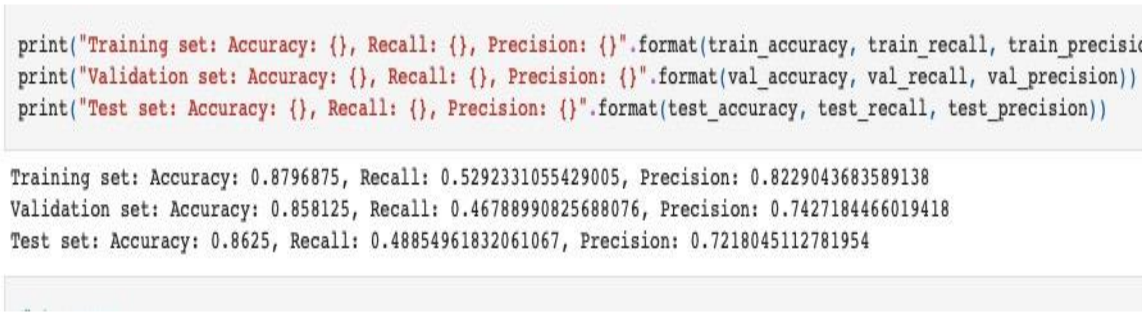
WEEK 10

This week, we employed several techniques in order to enhance the efficiency of our XGBoost variation 3 model. The model was trained with the following parameters (learning rate = 0.1, n_estimator =140, max_depth = 3, min_child_weight=1, gamma=0, subsample=0.8). We used accuracy, precision and recall as the evaluation metrics, with a focus on recall to ensure appropriate actions could be taken to retain customer likely to churn, reducing customer attrition rate. The performance of XGBoost model from last week is included at the end of this report for reference.

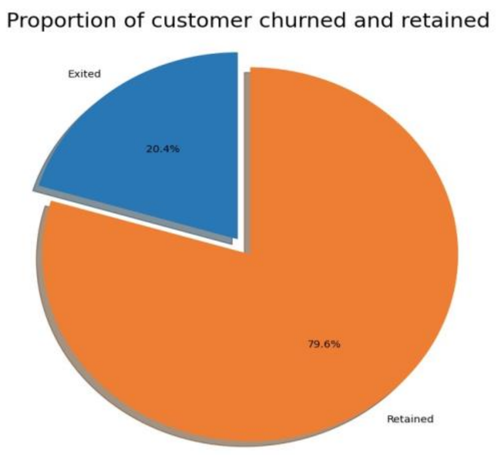
Firstly, we applied PCA (Principal Component Analysis) to reduce the dimensionality of our data. PCA transforms the original variables into a new set of variables called principal components, which capture most of the variance in the data. This technique was employed to address the correlation among some features, such as older people having higher income. However, reducing the number of features did not improve our model's performance by removing redundant information.

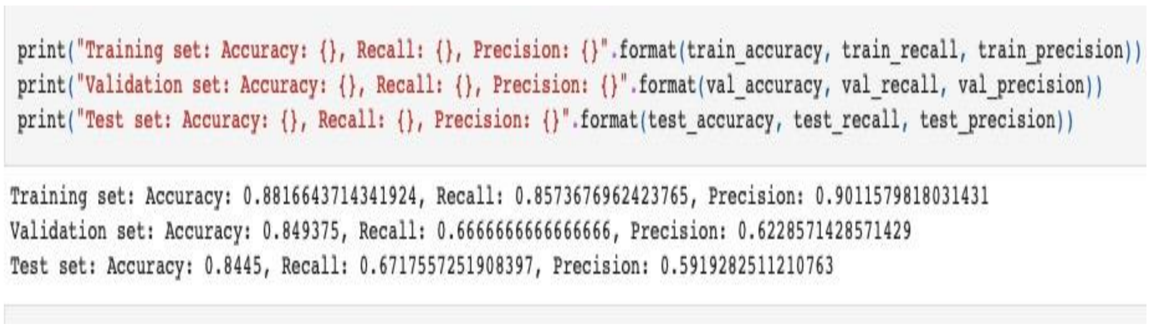


```
print("Training set: Accuracy: {}, Recall: {}, Precision: {}".format(train_accuracy, train_recall, train_precisi
print("Validation set: Accuracy: {}, Recall: {}, Precision: {}".format(val_accuracy, val_recall, val_precision))
print("Test set: Accuracy: {}, Recall: {}, Precision: {}".format(test_accuracy, test_recall, test_precision))
```

```
Training set: Accuracy: 0.8796875, Recall: 0.5292331055429005, Precision: 0.8229043683589138
Validation set: Accuracy: 0.858125, Recall: 0.46788990825688076, Precision: 0.7427184466019418
Test set: Accuracy: 0.8625, Recall: 0.48854961832061067, Precision: 0.7218045112781954
```

Secondly, we employed SMOTE (Synthetic Minority Over-sampling Technique) to address imbalanced data. Our dataset was imbalanced, with 80% of data in class 0 (Retained) and 20% of data in class 1 (Churned.) SMOTE generates synthetic samples for the minority class by interpolating between the minority samples. By balancing the dataset, we achieved a notable improvement in recall, which reached 67%, compared to the original 48%. However, this came at a cost, as the model's precisions decreased due to an increase in false positives.

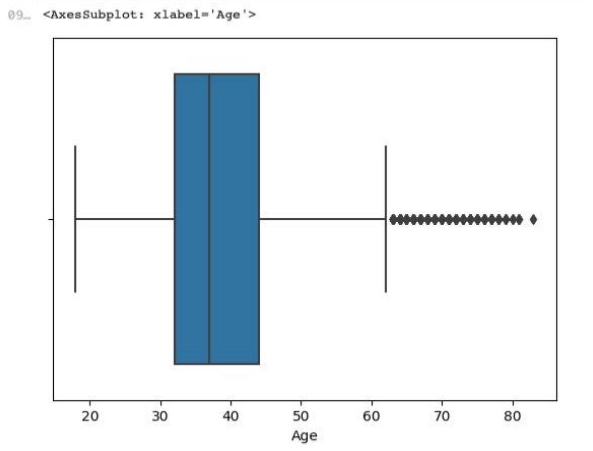


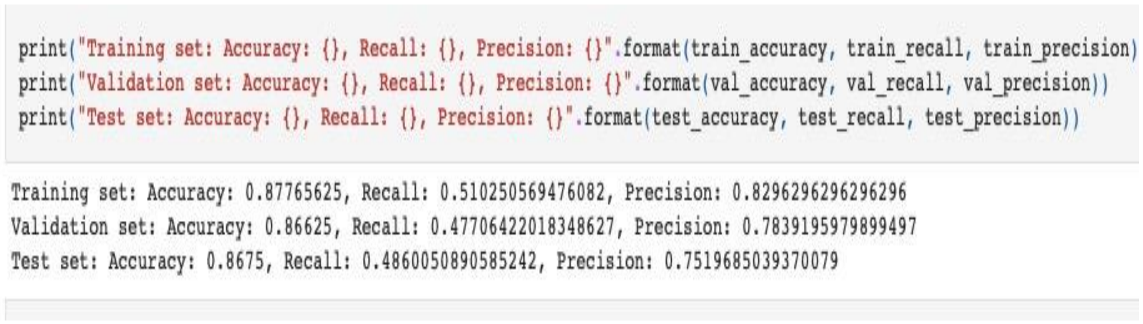


```
print("Training set: Accuracy: {}, Recall: {}, Precision: {}".format(train_accuracy, train_recall, train_precision))
print("Validation set: Accuracy: {}, Recall: {}, Precision: {}".format(val_accuracy, val_recall, val_precision))
print("Test set: Accuracy: {}, Recall: {}, Precision: {}".format(test_accuracy, test_recall, test_precision))
```

```
Training set: Accuracy: 0.8816643714341924, Recall: 0.8573676962423765, Precision: 0.9011579818031431
Validation set: Accuracy: 0.849375, Recall: 0.6666666666666666, Precision: 0.6228571428571429
Test set: Accuracy: 0.8445, Recall: 0.6717557251908397, Precision: 0.5919282511210763
```

We also utilized feature engineering to improve the model's performance by creating new features from the existing data. In our case, we identified numerous outliers in the age column that were predictive of banking customer churn. We opted to keep these outliers by adding a new column that indicated whether a data point was an outlier with respect to age. The result just improve a slight improvement, with an accuracy of 86%, precision of 75 and recall of 48%. For the test dataset, compared to the original accuracy of 86%, precision of 72 and recall of 47%.

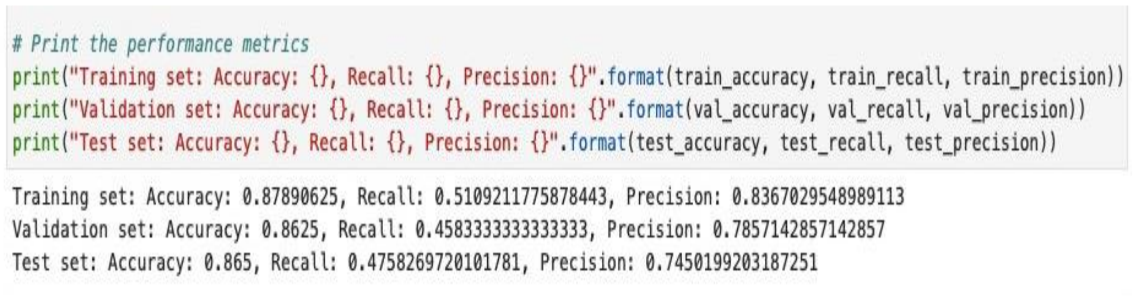




```
print("Training set: Accuracy: {}, Recall: {}, Precision: {}".format(train_accuracy, train_recall, train_precision)
print("Validation set: Accuracy: {}, Recall: {}, Precision: {}".format(val_accuracy, val_recall, val_precision))
print("Test set: Accuracy: {}, Recall: {}, Precision: {}".format(test_accuracy, test_recall, test_precision))
```

```
Training set: Accuracy: 0.87765625, Recall: 0.510250569476082, Precision: 0.8296296296296296
Validation set: Accuracy: 0.86625, Recall: 0.47706422018348627, Precision: 0.7839195979899497
Test set: Accuracy: 0.8675, Recall: 0.4860050890585242, Precision: 0.7519685039370079
```

Lastly, we employed normalization to transform the dataset into a normal distribution. This technique helps to scale the data so that it has a mean of 0 and standard deviation of 1, which could improve the performance of models that are sensitive to the scale of the data. However, this technique didn't improve our model's performance. The reason normalization did not improve the performance of our XGBoost model is that decision tress, on which XGBoost is based, are insensitive to the scale of data. Moreover, when continuous variables are normalized, their original meaning can be lost, which can adversely affect the performance of the data. Thus, in our case, normalization was not beneficial in addressing the fundamental issues affecting the model's performance, such as imbalanced data.



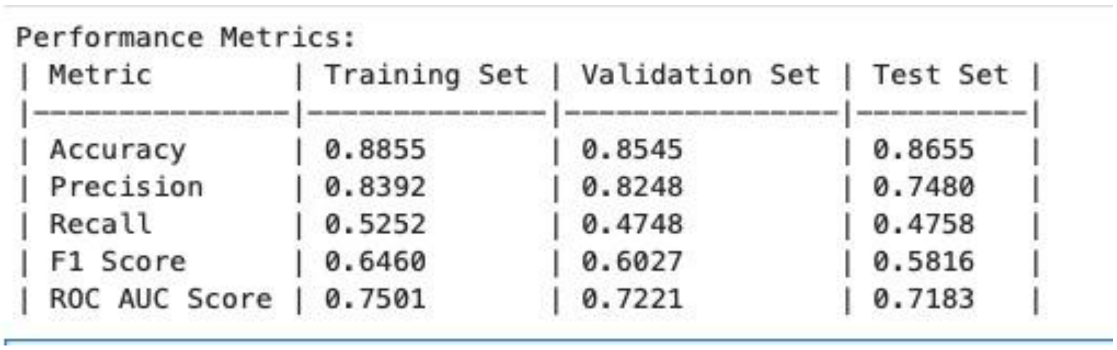
```
# Print the performance metrics
print("Training set: Accuracy: {}, Recall: {}, Precision: {}".format(train_accuracy, train_recall, train_precision))
print("Validation set: Accuracy: {}, Recall: {}, Precision: {}".format(val_accuracy, val_recall, val_precision))
print("Test set: Accuracy: {}, Recall: {}, Precision: {}".format(test_accuracy, test_recall, test_precision))
```

```
Training set: Accuracy: 0.87890625, Recall: 0.5109211775878443, Precision: 0.8367029548989113
Validation set: Accuracy: 0.8625, Recall: 0.4583333333333333, Precision: 0.7857142857142857
Test set: Accuracy: 0.865, Recall: 0.4758269720101781, Precision: 0.7450199203187251
```

In conclusion, SMOTE performed well in increasing recall, but this improvement came at the cost of precision. Therefore, a trade-off should be considered when using SMOTE. Further exploration of feature engineering could also be explored to improve the model's performance.

Reference:

Below is the performance of our XGBoost model from last week:



```
Performance Metrics:
| Metric        | Training Set | Validation Set | Test Set |
|---------------|--------------|----------------|----------|
| Accuracy      | 0.8855       | 0.8545         | 0.8655   |
| Precision     | 0.8392       | 0.8248         | 0.7480   |
| Recall        | 0.5252       | 0.4748         | 0.4758   |
| F1 Score      | 0.6460       | 0.6027         | 0.5816   |
| ROC AUC Score | 0.7501       | 0.7221         | 0.7183   |
```