Building a Machine Learning Classification Model for an Unknown Dataset

Gupta, Moro, Kannan

1. Business Understanding

Introduction: The main objective of this case study is to build a machine learning model for an unknown dataset. There is neither metadata nor background information available to understand the business case. With the little details known about the independent and target variables, we were able to identify that the target is a binary field. (which takes the value either '0' or '1'). Hence the machine learning model to be built must be a binary classification model.

As mentioned above the real business scenario that handles or generates this dataset is unknown. Having stated that, the real challenge addressed here is to demonstrate the ability to model this dataset using best practices so that the model can be generalized to future unknown observations. In this case study, the focus is on (1) exploring different modelling approaches to determine a baseline model, (2) evaluating the model using a custom business centric metrics (provided by the data owners), (3) tuning the model to obtain the best possible metric, and (4) providing insights into which features are most influential in making the predictions both globally and locally.

The importance of this business problem stems from the fact that there is a cost associated with making wrong predictions. However, not all mistakes are equally costly for the data owners. For example, if the data belongs to a bank and the outcome being predicted is whether a loan should be given to a customer or not, then giving a loan to a sub-prime customer could be less costly (risk of default) compared to not giving a loan to a prime customer (loss of stable revenue). Hence usual metrics available for classification tasks cannot be used directly and special consideration must be made to this requirement when developing, tuning and selecting the best models. The final model must be the one that minimizes the overall cost for the data owners.

Additionally, it may be possible that there is a need for explaining the outcome and why a certain prediction was made. Again, if this data belongs to a regulated industry, such as banking, this is especially important since decisions made by models could be challenged and lead to discrimination lawsuits. Having the ability to explain the outcomes and what factors influenced the decision by the model would be especially helpful in (1) evaluating if the model is making ethical decisions before it is put into production, and (2) defend against lawsuits if it is decided to put the model into production.

Assumptions: Since not much information is known about the data or the business case, there are few assumptions made while solving this use case.

(1) The first assumption is that the distribution of dataset provided for this case will not change much over time, hence the future unknown dataset will be a sample of the same underlying distribution. This means that any model we develop now will still be valid in the future. In our banking example above, this implies that the mix of subprime and prime customers remains the same, the distribution of race, ethnicity, gender in the customer mix remains the same and so on.

(2) The next assumption is that the cost-based framework will not change in future hence the suggested final model will hold good in future. If this assumption does not hold true, then the process outlined in this case study should be repeated using the new cost structure. For the banking example above, this could happen if there is a change in market conditions which makes the cost associated with lending to subprime customers higher or lower.

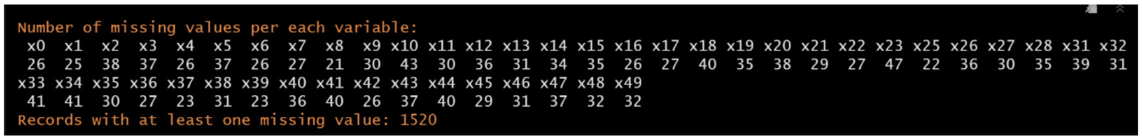
(3) Finally, without having much context into the features and the industry to which the data belongs, we assume that the model interpretability and explainability are important and we will focus on this in this case study in detail.

2. Data Quality

The dataset used for this case study has been made available via public this web address. It contains about 160,000 observations with 51 variables each. The target variable is named ‘y’, while the 50 predicting variables are named from ‘x0’ to ‘x49’. A first view of the dataset shows three variables containing text (x24, x29 and x30), while the others have numeric values. Two of the numeric variables, x37 and x32, are formatted with dollar ($) and percent (%) sign respectively, so cleanup must be done to convert these to numeric values before they can be used in a model.

2.1 Missing Values

We noticed that every variable has a very small amount of missing data. Fig. 1 shows the detailed count of missing value for each variable and in aggregate, we can notice than only 1,520, or 1% of total records, have some missing value. We think this classification task can be addressed by selecting the best model from a large variety of models but having missing values in numeric fields will not allow the use of certain models. One way to overcome this would be to impute the missing data. However, this can be very context specific and using the wrong imputation technique could lead to deteriorated performance on unseen datasets in the future. Since we don’t have enough knowledge of the dataset context, so we are not confident of leveraging any imputation technique. Given this, and the fact that the missing data represents less than 1% of the entire dataset, we decided to remove these records for the model’s training and testing. If in the future, context around the dataset is made available, we can evaluate this decision again to decide if imputation would be a better option.



```
Number of missing values per each variable:
 x0  x1  x2  x3  x4  x5  x6  x7  x8  x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 x25 x26 x27 x28 x31 x32
 26  25  38  37  26  37  26  27  21  30  43  30  36  31  34  35  26  27  40  35  38  29  27  47  22  36  30  35  39  31
x33 x34 x35 x36 x37 x38 x39 x40 x41 x42 x43 x44 x45 x46 x47 x48 x49
 41  41  30  27  23  31  23  36  40  26  37  40  29  31  37  32  32
Records with at least one missing value: 1520
```

Fig. 1 Counting of missing values per each variable

2.2 Variables Analysis

Target Variable

The target variable in the dataset is expressed by column ‘y’ which has no missing values. The target column is populated with values of ‘1’ or ‘0’. Of the 160,000 records, 59.9% have the value of ‘0’ and 40.1% have the value of ‘1’. This makes the dataset slightly unbalanced toward the ‘0’ class level, but it should not require any over/under sampling technique to re-balance the dataset.

Quantitative Variables

Density plots for the numerical variables depicted in Fig. 2 show a predominance of normal distributions, with most of them centered at or around zero (mean varies from -12 up to 6.7). The standard deviation of the features ranges from 0 up to 999 (with most being less than 10). The variance across the distributions may be important depending on the type of models we want to use. Some models are, in fact, resistant to large variance across variables, while other (e.g. distance based like Support Vectors, KNN, etc.) are much more sensitive and may require some scaling to get them to the same range. We decided to avoid scaling directly as most variables are within similar ranges and models that don’t require scaling benefit from a better interpretation of the non-scaled predictors. We will see later than an Automated Machine Learning (AutoML) framework will take care of scaling automatically for us as needed for certain algorithms while leaving the data non-scaled when it is not needed. Fig. 2 shows that one of the variables (‘x32’) contains non continuous values which have been grouped in bins evenly spaced by 0.0001 units (ranging from -0.0005 to +0.0005). This is the variable that was originally marked with a percentage (%) sign. Due to potential rounding of data, these seem to only be available in certain numeric values.

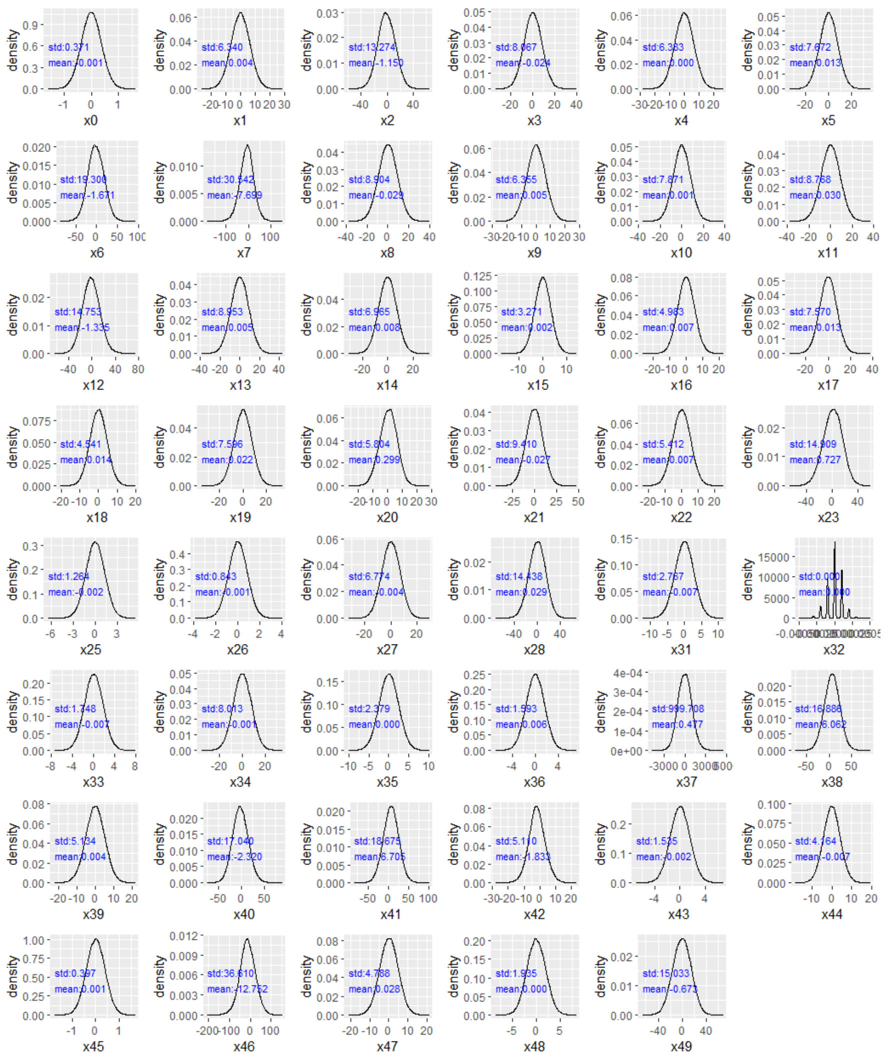


Fig. 2 Density Plots of numeric variables

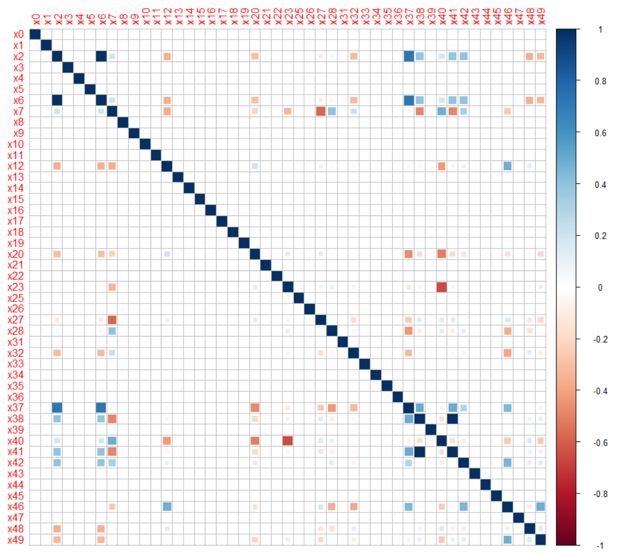
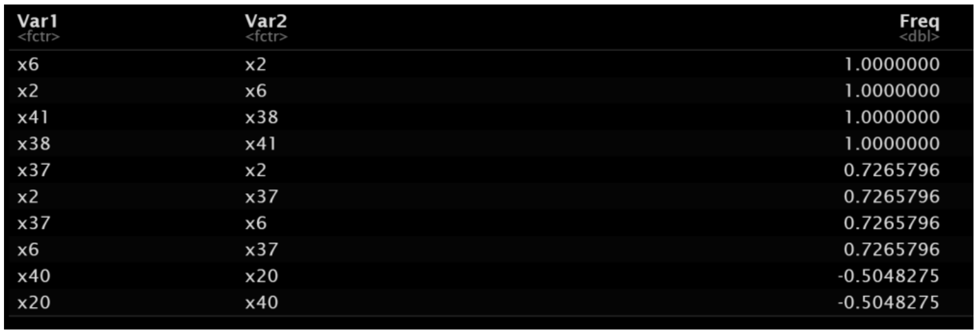


Fig. 3 Correlation plot of numeric variables.



Var1 <fctr>	Var2 <fctr>	Freq <dbl>
x6	x2	1.0000000
x2	x6	1.0000000
x41	x38	1.0000000
x38	x41	1.0000000
x37	x2	0.7265796
x2	x37	0.7265796
x37	x6	0.7265796
x6	x37	0.7265796
x40	x20	-0.5048275
x20	x40	-0.5048275

Fig. 4 Variables with correlation greater than 0.5

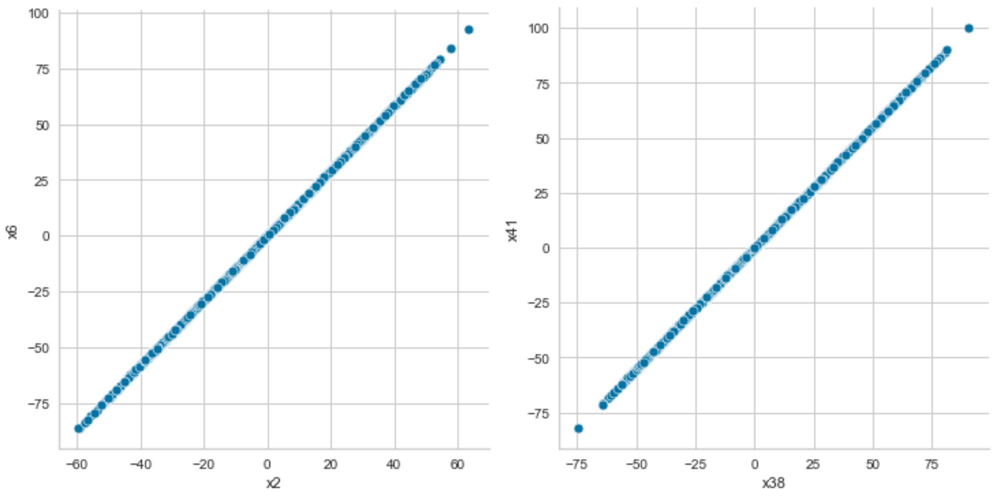


Fig. 5 Distribution of values between the variables with correlation of 1.0

The correlation plot between features (Fig. 3) shows an overall small degree of correlation, as most of the variables pairs a have very low correlation value, expressed as a white box. Only 5 pairs of variables have a correlation greater than 0.5, and 2 pairs ('x6' with 'x2', and 'x41' with 'x38') show a perfect correlation of 1.0 (Fig. 4). A correlation of 1.0 means that as one variable increases the other increases by a constant factor, so the change in a variable perfectly explains the change in the other variable. Fig. 5 shows the perfect linear correlation for these 2 pairs of variables. Such behavior (called multicollinearity) can dilute the interpretation power of the paired predictors in certain algorithms, so we are going to remove one of the paired variables before training any model.

To understand if the features can in isolation explain the target variable, we plotted the distribution of their values broken by the target variable's value. If the distribution of one of the predictors is drastically different between the two class values (1 and 0) of the target variable, this would mean that the predictor can largely explain the target value. Fig. 6 shows no feature is able to autonomously explain the target value since most of them have little or slight difference in the mean and standard deviation between the target value of 1 and 0. The few variables with a larger difference in the mean (such as 'x23' and 'x49') still have overlapping ranges when the target is 1 or 0. This shows the need for a multivariate model to capture any pattern in the predictor variables that can explain the target variables.

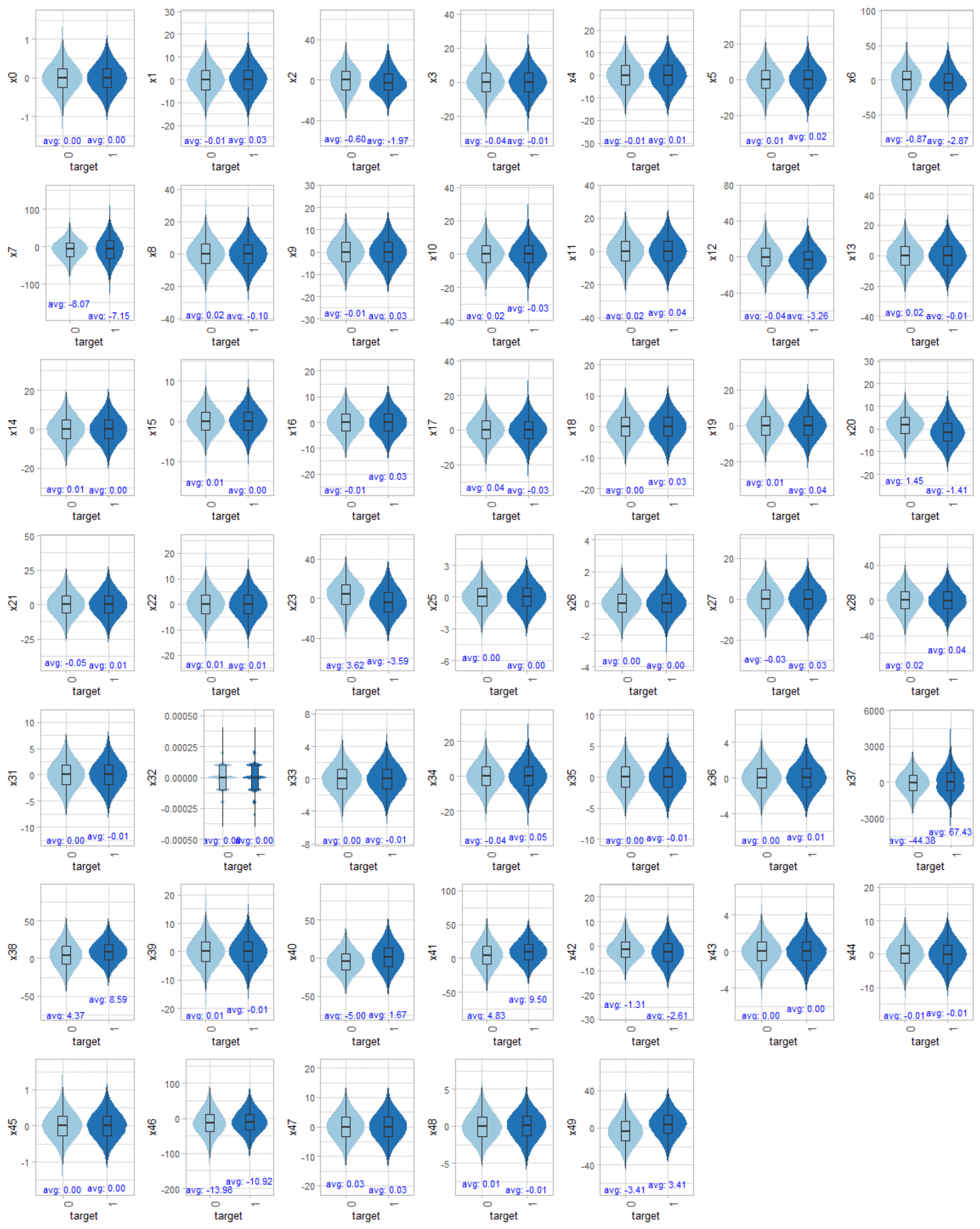
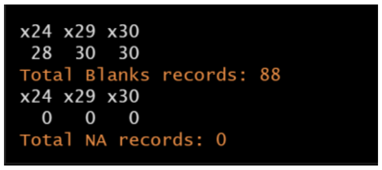


Fig. 6 Distribution of numeric variables by the value of the target variable.

Qualitative Variables

The dataset contains three non-numeric variables: ‘x24’, ‘x29’, and ‘x30’. Fig. 8 shows the domain for each of these variables. We notice that these variables represent geography and calendar related information. As we have no knowledge of the data context, we cannot further transform these data such as expressing the calendar columns as a time-based variable, or the locations as spatial coordinates, etc. Additionally, although there are no records with missing non-numeric values (expressed as NA), some records have a blank entry as shown in Fig. 7. Because of the lack of information about the context of the dataset, we decided to keep these observations as valid data points with a blank value representing a new class level. Given more context around the problem, we could have taken an alternate route such as imputing the data using appropriate means.



```
x24 x29 x30
 28  30  30
Total Blanks records: 88
x24 x29 x30
  0   0   0
Total NA records: 0
```

Fig. 7 Qualitative Variables with Null or Blank records

Fig. 8 shows the distribution of the values for each qualitative variable, and we can see that some values are predominant for certain features. 87% of the records have ‘x24’ equal to ‘Asia’. Similarly, 63% of the records have ‘x30’ equal to ‘wednesday’. Variable ‘x29’ also has a skewed distribution with ‘July’ and ‘Jun’ being the most common values (accounting for more than half of the observations). Interestingly, some values have very little representation in the dataset, such ‘monday’, ‘Feb’, etc. Initially, we assumed that this data is representative of the population from which it is drawn, and that the distribution will remain the same in the future. This analysis sheds some doubts on that assumption so we must proceed with caution. However, given the lack of context we will continue as if this assumption is true.

We can also notice some spelling mistakes in the data. For instance, ‘Dev’ may represent the month of December, while ‘euorpe’ should represent the continent of Europe. It is important to note that if this modeling system were to interact with other systems which also have location and calendar-based information, then there could be issues with merging of data if the other systems have the correct spellings and consistency in naming of months. For the purpose of this case though, since we were primarily concerned with modeling only and the misspelling were consistent across this dataset, we decided not to fix the inconsistencies and proceed as is. In addition, the lack of context around the dataset evades any further feature engineering process to explain and improve this distribution.

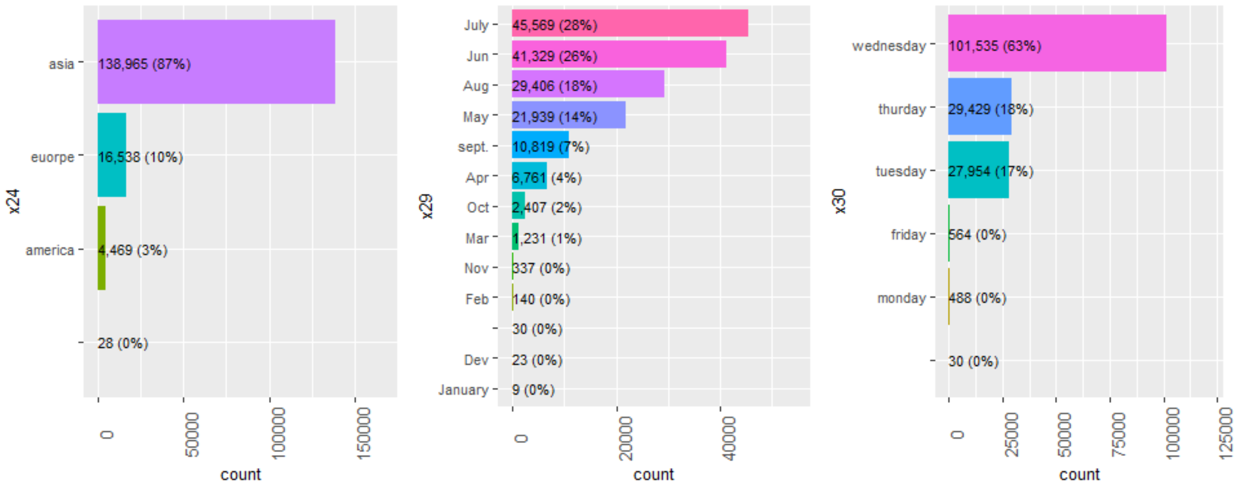


Fig. 8 Distribution of qualitative values

Like the previous section, we checked if any of the qualitative values can explain the target variable in isolation. Fig. 9 compares the distribution of each qualitative variable by the target variable’s value. We can see that the representation of each predictor across the target variable (‘1’ and ‘0’) is not drastically different and is always around 60% for y=0 and 40% of y=1. We can conclude that none of the qualitative values is able to directly explain the target variable, hence we need a multivariate model to capture any feature pattern that explains the target variable.

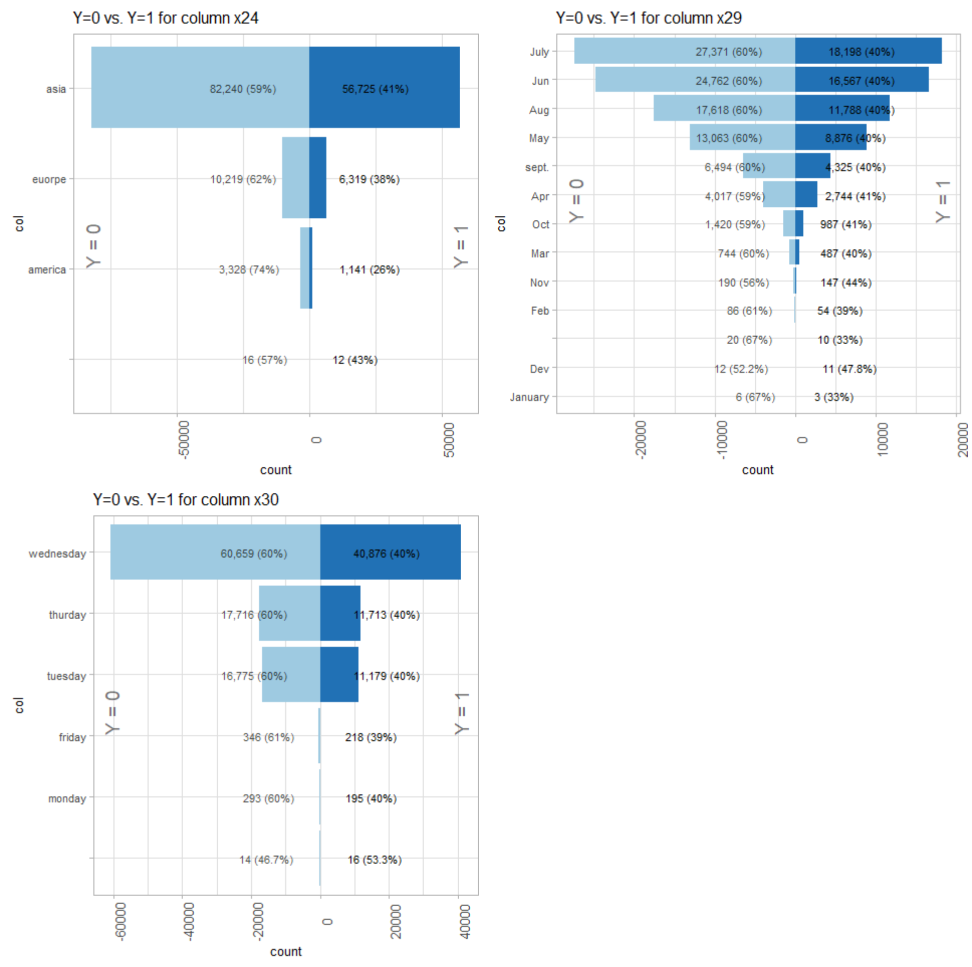


Fig. 9 Comparing Qualitative Variables between Y=0 and Y=1 records

3. Modeling Preparation

Modeling Approaches: Since the goal of this case study was a binary classification task, there were many modeling approaches that could have been used. However, it is time consuming to try many types of models on a dataset to determine the best one. Hence an Automated Machine Learning (AutoML) technique is suitable for this scenario which can try out various models and hyperparameter settings on a given dataset and select the best model from the list based on certain performance metrics. For this case study, we used PyCaret [1] as our AutoML framework of choice. This framework allows for fast prototyping while at the same time providing enough hooks to customize the needs of the modeling engineer. This flexibility is what we need in this case study since the metric of our choice is a custom one. In addition, it has the additional advantage of running certain algorithms on the GPU which can considerably speed up the training time, especially since many hyperparameter configurations need to be tried. Also, the framework automatically takes care of data preprocessing for categorical data appropriately (such as one-hot encoding) as well as scaling needed by certain algorithms such as K-Nearest Neighbors, so this reduces the burden on the modeling engineers.

Metrics: To identify the best model, we need to define a measure that best represent successful predictions. Binary classifications are usually measured using metrics based on the 'Confusion Matrix' table. This is a visual representation of the successful and unsuccessful predictions of a model and is comprised of four sections. Fig. 10 shows the Confusion Matrix utilized for this case study [2]. The objective is to reduce the False Positive and False Negative cases which are a simple count of the predictions that are not matching the actual values.

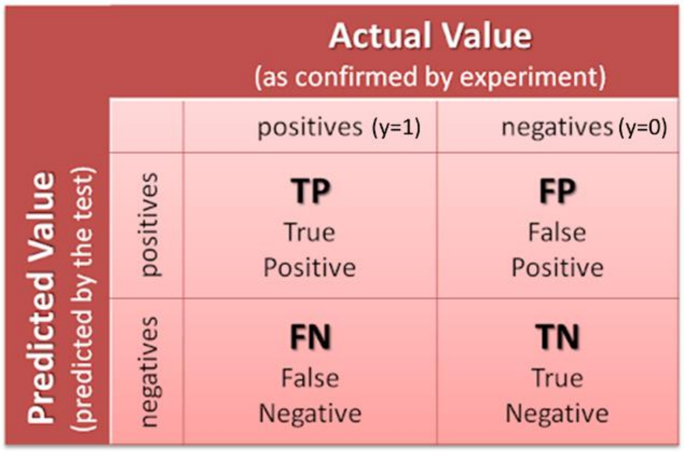


Fig. 10 Confusion Matrix

Based on the inputs for the case study, we know that there is a cost of \$10 for False Positives or FP (when the predicted value is 1 and the real value is 0) and a cost of \$500 for False Negatives or FN (predicted value is 0 and real value is 1). There is no cost associated with correct predictions. Consequently, we can calculate a custom metrics that assigns the cost of \$10 for FP, \$500 for FN, and a value of \$0 when the prediction is correct. Then we calculate the average of this value across all the predictions so that we can compare cases which have different number of predictions. The resulting number is an expression of the average cost of the model per prediction, and we want to find the model that best reduces this metric value.

$$Model\ Cost\ Per\ Prediction = \frac{Count\ of\ False\ Positives * 10 + Count\ of\ False\ Negatives * 500}{Count\ of\ Predictions}$$

Interpretability and Explainability: Given the fact that explainability is an important consideration for this case study, we need an explainability method that can work across all the models in our arsenal. While coefficient-based interpretation works for certain models like logistic regression and tree traversal based, interpretation works for others such as Random Forest, these are not generic and cannot be applied to other algorithms. Hence, for this case study, we rely on Shapley Values and Partial Dependence Plots to better explain the model's decisions in a way that is agnostic of the model that is eventually picked by the AutoML framework. Luckily, PyCaret provides a nice unified wrapper around the Shapley functionality which makes this task relatively easy.

Shapley Values are derived from Game Theory and indicate the marginal payout of adding one more variable into the mix of existing variables [3]. In the context of machine learning, it means the addition gain in performance that is obtained by adding one more feature into the mix of existing features. Partial Dependence Plots are the result of an investigatory process on the trained model, where we analyze the marginal effect of one or more features to the target variable. These plots can show the relationship between the target and predictor variable, its shape and direction. They can show linear or complex relations and their impact to the probability of positive class prediction [4]. We believe the business decision-makers can utilize these approaches, in addition to the overall cost of the model, to understand the importance and the impact of each predictors to the overall outcome.

4. Model Building and Evaluation

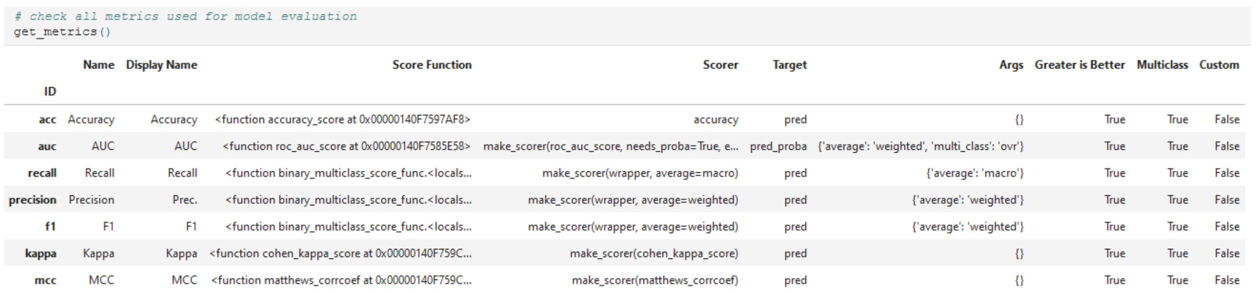
4.1 Setup

Data Setup (Mock Future Data): Since our goal was to get an unbiased estimate of the model that we develop, we removed 20% of the data to be used for final evaluation. This would not be used anywhere in the model development process and can be treated akin to any future data that we might receive for classification once our model has been deployed. The breakdown of the number of observations is shown in Table 1.

Table 1. Split for the Original Data to create Mock Future Data

Data	Number of Observations	Percentage
Original	158,392	100%
Available Data for Training	126,714	80%
Mock Future Data	31,679	20%

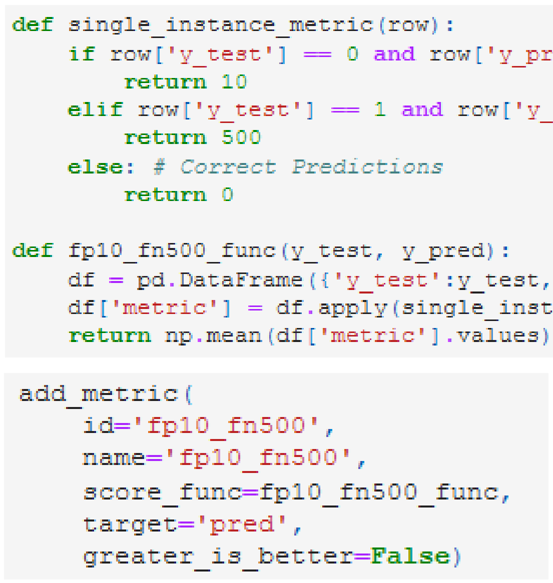
Metrics: For the binary classification task, the default metrics provided by PyCaret include “Accuracy”, “AUC”, “Recall”, “Precision”, “F1 Score”, “Kappa”, and “MCC” (Fig. 11). However, as explained in section 3, our evaluation metric is based on a cost benefit matrix. As such, using this matrix to select the best model seemed more appropriate in this case instead of any of the default metrics. This custom metric was added to PyCaret using the `add_metric` function (Fig. 12). The resultant metrics available after adding this custom metric (`fp10_fn500`) are shown in Fig. 13.



```python
# check all metrics used for model evaluation
get_metrics()
```

ID	Name	Display Name	Score Function	Scorer	Target	Args	Greater is Better	Multiclass	Custom
acc	Accuracy	Accuracy	<function accuracy_score at 0x00000140F7597AF8>	accuracy	pred	{}	True	True	False
auc	AUC	AUC	<function roc_auc_score at 0x00000140F7585E58>	make_scorer(roc_auc_score, needs_proba=True, e...	pred_proba	{'average': 'weighted', 'multi_class': 'ovr'}	True	True	False
recall	Recall	Recall	<function binary_multiclass_score_func.<locals...	make_scorer(wrapper, average=macro)	pred	{'average': 'macro'}	True	True	False
precision	Precision	Prec.	<function binary_multiclass_score_func.<locals...	make_scorer(wrapper, average=weighted)	pred	{'average': 'weighted'}	True	True	False
f1	F1	F1	<function binary_multiclass_score_func.<locals...	make_scorer(wrapper, average=weighted)	pred	{'average': 'weighted'}	True	True	False
kappa	Kappa	Kappa	<function cohen_kappa_score at 0x00000140F759C...	make_scorer(cohen_kappa_score)	pred	{}	True	True	False
mcc	MCC	MCC	<function matthews_corrcoef at 0x00000140F759C...	make_scorer(matthews_corrcoef)	pred	{}	True	True	False

Fig. 11. Default classification metrics available in PyCaret

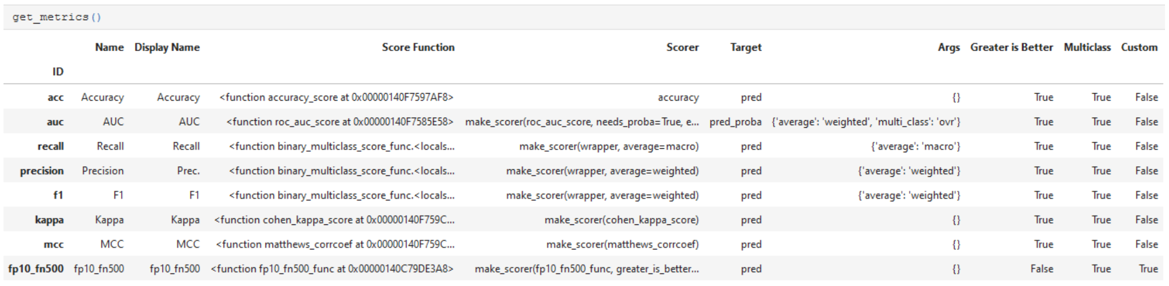


```python
def single_instance_metric(row):
    if row['y_test'] == 0 and row['y_pred'] == 1: # False Positive
        return 10
    elif row['y_test'] == 1 and row['y_pred'] == 0: # False Negative
        return 500
    else: # Correct Predictions
        return 0

def fp10_fn500_func(y_test, y_pred):
    df = pd.DataFrame({'y_test':y_test, 'y_pred':y_pred})
    df['metric'] = df.apply(single_instance_metric, axis=1)
    return np.mean(df['metric'].values)
```

```python
add_metric(
    id='fp10_fn500',
    name='fp10_fn500',
    score_func=fp10_fn500_func,
    target='pred',
    greater_is_better=False)
```

Fig. 12. Addition of Custom Metric to PyCaret Setup



```
get_metrics()
```

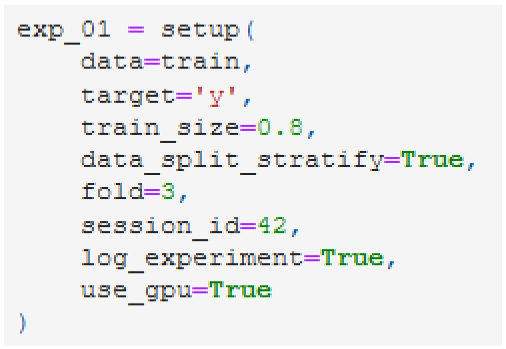
ID	Name	Display Name	Score Function	Scorer	Target	Args	Greater is Better	Multiclass	Custom
acc	Accuracy	Accuracy	<function accuracy_score at 0x00000140F7597AF8>	accuracy	pred	{}	True	True	False
auc	AUC	AUC	<function roc_auc_score at 0x00000140F7585E58>	make_scorer(roc_auc_score, needs_proba=True, e...	pred_proba	{'average': 'weighted', 'multi_class': 'ovr'}	True	True	False
recall	Recall	Recall	<function binary_multiclass_score_func.<locals...	make_scorer(wrapper, average=macro)	pred	{'average': 'macro'}	True	True	False
precision	Precision	Prec.	<function binary_multiclass_score_func.<locals...	make_scorer(wrapper, average=weighted)	pred	{'average': 'weighted'}	True	True	False
f1	F1	F1	<function binary_multiclass_score_func.<locals...	make_scorer(wrapper, average=weighted)	pred	{'average': 'weighted'}	True	True	False
kappa	Kappa	Kappa	<function cohen_kappa_score at 0x00000140F759C...	make_scorer(cohen_kappa_score)	pred	{}	True	True	False
mcc	MCC	MCC	<function matthews_corrcoef at 0x00000140F759C...	make_scorer(matthews_corrcoef)	pred	{}	True	True	False
fp10_fn500	fp10_fn500	fp10_fn500	<function fp10_fn500_func at 0x00000140C79DE3A8>	make_scorer(fp10_fn500_func, greater_is_better...	pred	{}	False	True	True

Fig. 13. Available Metrics after adding Custom Metric (last row)

Resampling Strategy: For PyCaret's internal model development and validation, the available data (126,714 observations) was sub divided into a training (80%) and test split (20%) as shown in Table 2. The split was stratified to ensure that both the train and the test split had the same distribution of the target variable. Three-fold cross validation was used on the training split to create a more robust estimate of the model's performance. This essentially means that during the training process, the training data was split into 3 parts (folds) with 2 parts used for training and 1 part used for validating the model. This process is repeated 3 times with each fold being used as the validation set once. Then an average of the metric across all three folds is taken as the final metric for evaluating various models. A seed of 42 was set to ensure that the results of this experiment could be recreated in the future. This setup is detailed in Fig. 14.

Table 2. Train/Test Split for Available Data

Data	Number of Observations
Available Data for Training	126,714
Internal Training (PyCaret)	101,371
Internal Test (PyCaret)	25,343

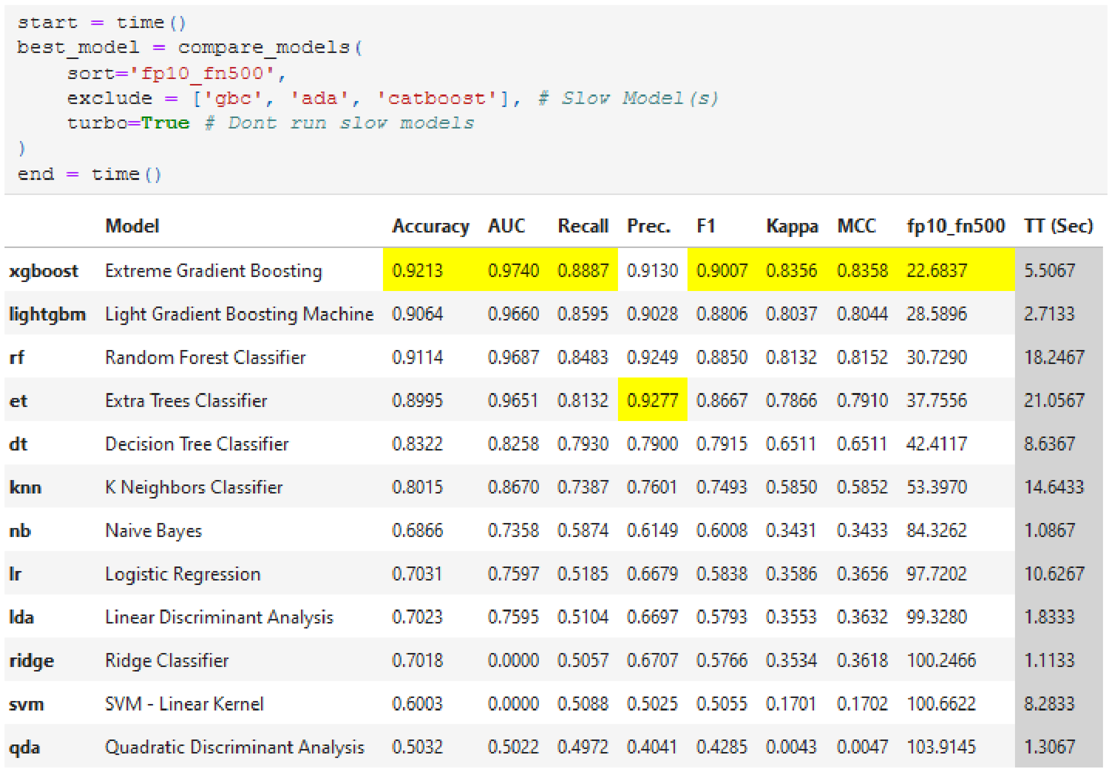


```
exp_01 = setup(
    data=train,
    target='y',
    train_size=0.8,
    data_split_stratify=True,
    fold=3,
    session_id=42,
    log_experiment=True,
    use_gpu=True
)
```

Fig. 14. PyCaret Setup

4.2 Baseline Model

Compare Multiple Baseline Models: The baseline model was developed using PyCaret `compare_models` function. This function essentially creates a baseline model for various algorithms with their default hyperparameters and returns the best model. By evaluating various types of models, we get a better baseline model as compared to randomly selecting an algorithm to try. The selection of the best model can be controlled based on the metric of choice. We used our custom metric to select the best model as described above. Also, certain slow models were removed from the evaluation for this case study. The best baseline model obtained was the `XGBoost` (extreme gradient bosting) model as shown in Fig. 15. This model had an associated cost of $22.68 per instance of the validation dataset. This cost is incurred due to a combination of False Positive and False Negative predictions by the model.

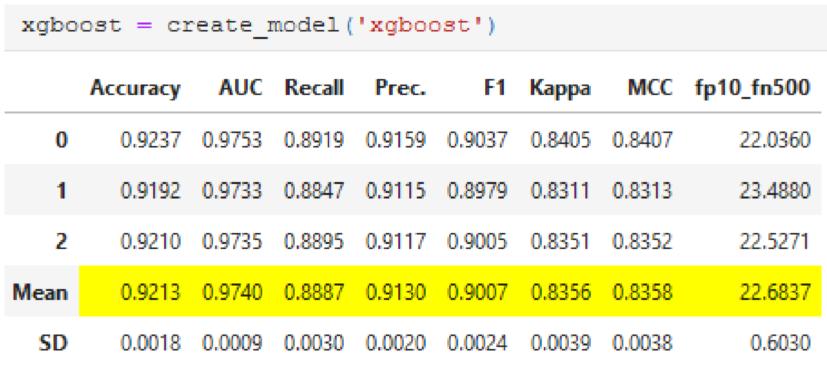


```
start = time()
best_model = compare_models(
    sort='fp10_fn500',
    exclude = ['gbc', 'ada', 'catboost'], # Slow Model(s)
    turbo=True # Dont run slow models
)
end = time()
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	fp10_fn500	TT (Sec)
xgboost	Extreme Gradient Boosting	0.9213	0.9740	0.8887	0.9130	0.9007	0.8356	0.8358	22.6837	5.5067
lightgbm	Light Gradient Boosting Machine	0.9064	0.9660	0.8595	0.9028	0.8806	0.8037	0.8044	28.5896	2.7133
rf	Random Forest Classifier	0.9114	0.9687	0.8483	0.9249	0.8850	0.8132	0.8152	30.7290	18.2467
et	Extra Trees Classifier	0.8995	0.9651	0.8132	0.9277	0.8667	0.7866	0.7910	37.7556	21.0567
dt	Decision Tree Classifier	0.8322	0.8258	0.7930	0.7900	0.7915	0.6511	0.6511	42.4117	8.6367
knn	K Neighbors Classifier	0.8015	0.8670	0.7387	0.7601	0.7493	0.5850	0.5852	53.3970	14.6433
nb	Naive Bayes	0.6866	0.7358	0.5874	0.6149	0.6008	0.3431	0.3433	84.3262	1.0867
lr	Logistic Regression	0.7031	0.7597	0.5185	0.6679	0.5838	0.3586	0.3656	97.7202	10.6267
lda	Linear Discriminant Analysis	0.7023	0.7595	0.5104	0.6697	0.5793	0.3553	0.3632	99.3280	1.8333
ridge	Ridge Classifier	0.7018	0.0000	0.5057	0.6707	0.5766	0.3534	0.3618	100.2466	1.1133
svm	SVM - Linear Kernel	0.6003	0.0000	0.5088	0.5025	0.5055	0.1701	0.1702	100.6622	8.2833
qda	Quadratic Discriminant Analysis	0.5032	0.5022	0.4972	0.4041	0.4285	0.0043	0.0047	103.9145	1.3067

Fig. 15. Baseline Model Details

Best Model Cross Validation Evaluation: Fig. 16 shows the detailed performance of the various folds for this model. The performance seems to be quite stable and consistent across the 3 folds with a standard deviation in the evaluation metric (`fp10_fn500`) of only $0.60. The mean value of the metric matches what was obtained in Fig. 15.



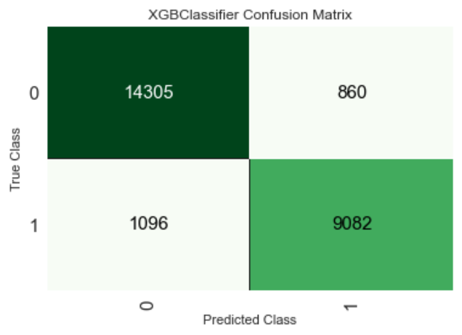
```
xgboost = create_model('xgboost')
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	fp10_fn500
0	0.9237	0.9753	0.8919	0.9159	0.9037	0.8405	0.8407	22.0360
1	0.9192	0.9733	0.8847	0.9115	0.8979	0.8311	0.8313	23.4880
2	0.9210	0.9735	0.8895	0.9117	0.9005	0.8351	0.8352	22.5271
Mean	0.9213	0.9740	0.8887	0.9130	0.9007	0.8356	0.8358	22.6837
SD	0.0018	0.0009	0.0030	0.0020	0.0024	0.0039	0.0038	0.6030

Fig. 16. Fold Details for XGBoost

Cost Benefit Evaluation: The performance of this baseline model can also be tested against the internal test split (25,343 observations). The results are shown in Fig. 17. Majority of the predictions (14,305 + 9,082) are accurate. However, there are 860 False Positives (associated with a cost of $10) and 1,096 False Negatives (associated with a cost of $500). Hence the average cost per prediction (for the internal test split) is $21.96.

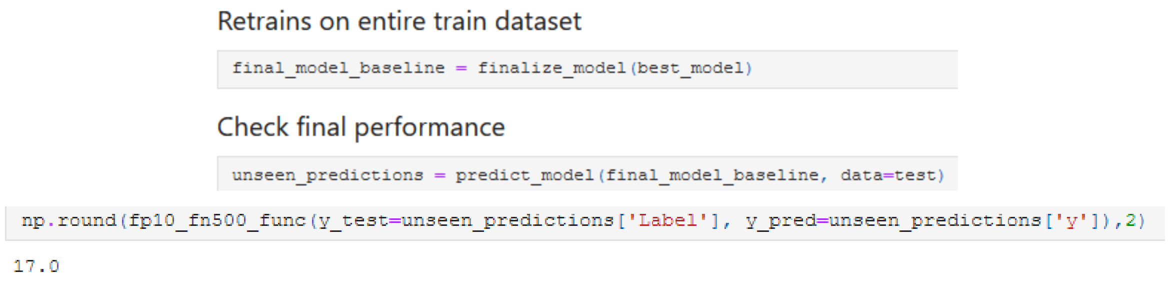
```
plot_model(best_model, plot = 'confusion_matrix')
```



$$\textit{Cost Per Observation} = \frac{860 * 10 + 1096 * 500}{25343} = \$\mathbf{21.96}$$

Fig. 17. Confusion Matrix and Cost Benefit for XGBoost Baseline Model against PyCaret's Internal Test Data Split

Retrain (Finalize) Baseline Model: Once this baseline model was obtained, the next step is to retrain this baseline model on the entire training dataset (126,714 observations) so that it can be (potentially) deployed and used to make predictions on future data. This process is done using PyCaret's `finalize_model` function as shown in Fig. 18. Once the model has been finalized, it can then be tested against the Mock Future Data (completely unseen) from Table 1. The custom metric obtained on the Mock Future data is $17.0 per observation which is even better than any of the metrics obtained during the training process above. Hence, we are confident that this baseline model will hold up well in the future if the distribution of the data remains consistent with what was used during the training process.



Retrains on entire train dataset

```
final_model_baseline = finalize_model(best_model)
```

Check final performance

```
unseen_predictions = predict_model(final_model_baseline, data=test)
```

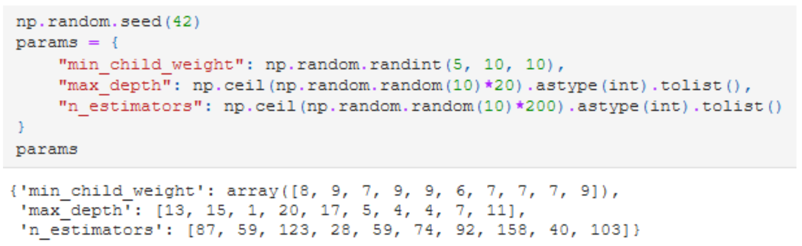
```
np.round(fp10_fn500_func(y_test=unseen_predictions['Label'], y_pred=unseen_predictions['y']),2)
```

```
17.0
```

Fig. 18. Finalized Baseline XGBoost Model and Evaluation on Future Data

4.3 Model Tuning

Hyperparameter Search: Once the baseline model has been finalized, the next step was to try to improve the performance of the model using hyperparameter tuning. Again, PyCaret makes this process very simple and this can be accomplished using its `tune_model` function. This function leverages PyCaret's internally defined grid to perform grid search in order to select the best tuned model. However, it also allows the modeling engineer to provide their own custom grid which we choose to do in this case as shown in Fig. 19. We choose the tune the hyperparameters described in Table 3 since these generally have the most impact on bias (underfitting) vs. variance (overfitting) tradeoff. i.e. the ability to capturing the trend vs. the ability to generalize to unseen data.



```
np.random.seed(42)
params = {
    "min_child_weight": np.random.randint(5, 10, 10),
    "max_depth": np.ceil(np.random.random(10)*20).astype(int).tolist(),
    "n_estimators": np.ceil(np.random.random(10)*200).astype(int).tolist()
}
params
```

```
{'min_child_weight': array([8, 9, 7, 9, 9, 6, 7, 7, 7, 9]),
 'max_depth': [13, 15, 1, 20, 17, 5, 4, 4, 7, 11],
 'n_estimators': [87, 59, 123, 28, 59, 74, 92, 158, 40, 103]}
```

Fig. 19. Custom Grid Search for Tuning the Baseline XGBoost Model

Table 3: Important Hyperparameters in XGBoost Classifier [5]

Hyperparameter	Meaning	Impact on Model Performance	Default Value
`min_child_weight`	Minimum sum of instance weight needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than `min_child_weight`, then the building process will give up further partitioning.	Larger `min_child_weight` results less overfitting since decisions will be made based on more observations in the node.	1
`max_depth`	The maximum depth of the tree.	Increase this parameter to reduce bias (underfitting). Increasing it too much may lead to high variance (overfitting)	6
`n_estimators`	Number of gradient boosted trees. Equivalent to number of boosting rounds.	Increase this parameter to reduce bias (underfitting).	100

Important Hyperparameters:

1. **min_child_weight:** Extreme Gradient Boosted Trees work by successively building tree based models where subsequent trees (rounds) are incentivized to make correct prediction on observations which were incorrectly predicted in the past (using weights associated with observations). Previous incorrect observations are given a higher weight which increases the incentive for future trees to try to classify these correctly. The `min_child_weight` hyperparameter determines the minimum weight of all the elements in a node of a tree needed to split it further into child nodes. If the sum of the weights of the observations in a node is lower than `min_child_weight`, then that node is not split any further and it becomes a leaf node. By increasing the value of `min_child_weight`, we are essentially forcing more observations in the leaf nodes. This leads to the final classification decision not being skewed by a small number of observations and hence prevents overfitting.

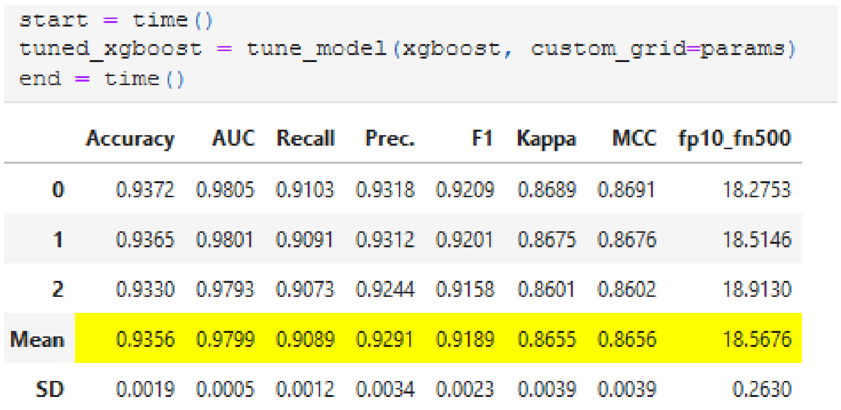
2. **max_depth:** This parameter controls the depth of the tree models that are used. Trees with smaller `max_depth` (shallow) cannot capture the trend in the data leading to underfitting. On the other hand, too deep trees tend to capture noise in the data leading to overfitting [6]. Hence a balance between underfitting and overfitting must be obtained through hyperparameter tuning.

3. **n_estimators:** This is the number of boosting rounds. The more the number of rounds, the more chance and incentive the model has to correctly classify previously incorrectly classified observations, hence reducing underfitting. The best value can be obtained using hyperparameter tuning.

Tuned Model Cross Validation Evaluation: Fig. 20 shows the results obtained after hyperparameter tuning. The mean value for the `fp10_fn500` custom metric across the 3 folds is now $18.57 per observation which is more than 18% better than the performance of the baseline model ($22.68). The difference in the hyperparameter values between the baseline and the tuned model are shown in Table 4. The `min_child_weight` has been increased to reduce overfitting, but this is balanced by the increase in `max_depth` and `n_estimators` to reduce the underfitting.

Table 4. Hyperparameter for Baseline vs. Tuned Model

Hyperparameter	Baseline Model	Tuned Model
`min_child_weight`	1	8
`max_depth`	6	15
`n_estimators`	100	123

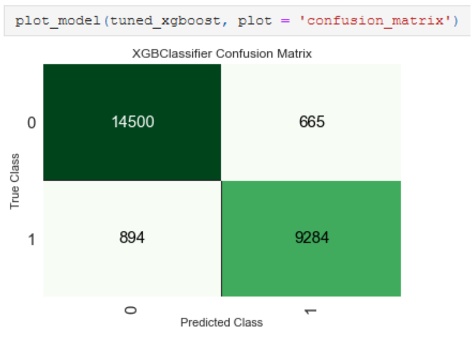


```
start = time()
tuned_xgboost = tune_model(xgboost, custom_grid=params)
end = time()
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	fp10_fn500
0	0.9372	0.9805	0.9103	0.9318	0.9209	0.8689	0.8691	18.2753
1	0.9365	0.9801	0.9091	0.9312	0.9201	0.8675	0.8676	18.5146
2	0.9330	0.9793	0.9073	0.9244	0.9158	0.8601	0.8602	18.9130
Mean	0.9356	0.9799	0.9089	0.9291	0.9189	0.8655	0.8656	18.5676
SD	0.0019	0.0005	0.0012	0.0034	0.0023	0.0039	0.0039	0.2630

Fig. 20. Tuning Results for the XGBoost Model

Cost Benefit Evaluation: The performance of this tuned model can also be tested against the internal test split (25,343 observations) like what was done for the baseline model. The results are shown in Fig. 21. The performance is better than the baseline model. There are only 665 False Positives compared to 860 in the baseline model and there are only 894 False Negatives compared to 1,096 in the baseline model. The average cost per prediction (for the internal test split) is $17.90 which is about 18.5% better than the baseline model ($21.96).

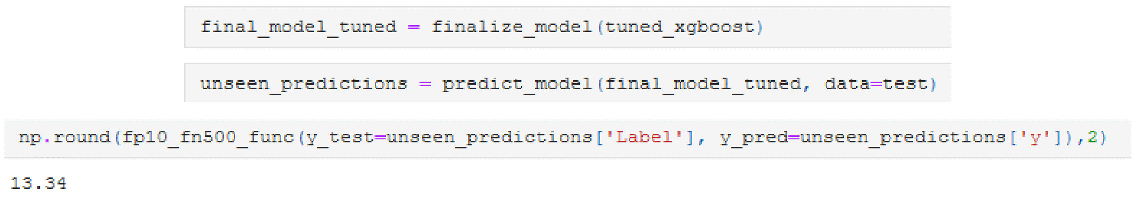


```
plot_model(tuned_xgboost, plot = 'confusion_matrix')
```

$$\textit{Cost Per Observation} = \frac{665 * 10 + 894 * 500}{25343} = \$\mathbf{17.90}$$

Fig. 21. Confusion Matrix and Cost Benefit for Tuned XGBoost Model against PyCaret's Internal Test Data Split

Retrain (Finalize) Tuned Model: Like what we did for the baseline model, we can retrain the tuned model on the entire training dataset (126,714 observations) so that it can be deployed and used to make predictions on future data (Fig. 22). The custom metric obtained on the Mock Future data is $13.34 per observation which is again better than any of the metrics obtained during the training process above. Hence, we are confident that this tuned model will also hold up well in the future if the distribution of the data remains consistent with what was used during the training process.



```
final_model_tuned = finalize_model(tuned_xgboost)
```

```
unseen_predictions = predict_model(final_model_tuned, data=test)
```

```
np.round(fp10_fn500_func(y_test=unseen_predictions['Label'], y_pred=unseen_predictions['y']),2)
```

```
13.34
```

Fig. 22. Finalized Tuned XGBoost Model and Evaluation on Future Data

4.4 Final Model Selection

We have obtained several evaluations for the baseline and the tuned model. These are summarized in Table 5. One final question that remains to be answered is which model should be chosen as the final model and potentially deployed to production. In this case, since the Tuned Model outperforms the Baseline model in all comparisons, the decision is simple (use the Tuned Model). However, note that this decision should be based purely on "PyCaret Internal Test Metric" since that is the only unbiased estimate left. The "Mean 3 Fold Metric" has already been used to determine which of the many potential baseline models should be selected as the best baseline model (Fig. 15) and hence should not be used again for selecting between the best baseline and the tuned model. The "Mock Future Data Metric" is calculated on potential future data. Its value is just a sanity check to see how the models will perform in the future and should not be used to make any decisions about selecting the final model.

Table 5. Metric (`fp10_fn500`) Comparison for the Baseline vs. Tuned Model

Metric	Baseline Model	Tuned Model	Improvement over Baseline
Mean 3 Fold Metric	$22.68	$18.57	18.1%
PyCaret Internal Test Metric	$21.96	$17.90	18.5%
Mock Future Data Metric	$17.00	$13.34	21.5%

5. Model Interpretability and Explainability

Model interpretability and explainability can be performed at two levels – (1) Global and (2) Local. Global importance explains how certain features impact the outcome overall (on average) but may fail to consider nuances for individual observations. Local explainability, on the other hand, deals with explaining the factors that are influential in making a single prediction. In this case study, we will evaluate both in detail.

5.1 Global Explainability

Global explainability can be achieved with the help of Shapley Values. The Shapley Value for a feature explains whether the feature value for an observation contributes to moving the outcome ("y" in this case) towards the higher end (1) or the lower end (0). A positive Shapley value for a feature implies that the feature contributes to moving the outcome variable value higher while a negative Shapley value indicates that the feature contributes to moving the outcome variable value lower [3]. In addition, interpretation based on Shapley value looks at whether the feature value itself is less than or greater than the mean value of that feature used during training. This is eventually used to decide the directionality (positive or negative correlation) of the impact of the feature on the outcome.

The concept of Shapley Values for this model is explained in detail in Fig. 23. The X-axis represents the Shapley Value (positive value corresponding to moving the output higher and negative value corresponding to moving the value lower). The plot contains a colored dot for each observation in the training set. A colored 'red' dot represents a feature value for a single observation that is higher than the mean value of that feature across the training data. A colored 'blue' dot represents a feature value for a single observation that is lower than the mean value of that feature across the training data. In addition, the features are ordered from top to bottom in terms of the ones that have the highest impact on the outcome (which can be used as a surrogate for feature importance).

```
interpret_model(best_model, plot = 'summary')
```

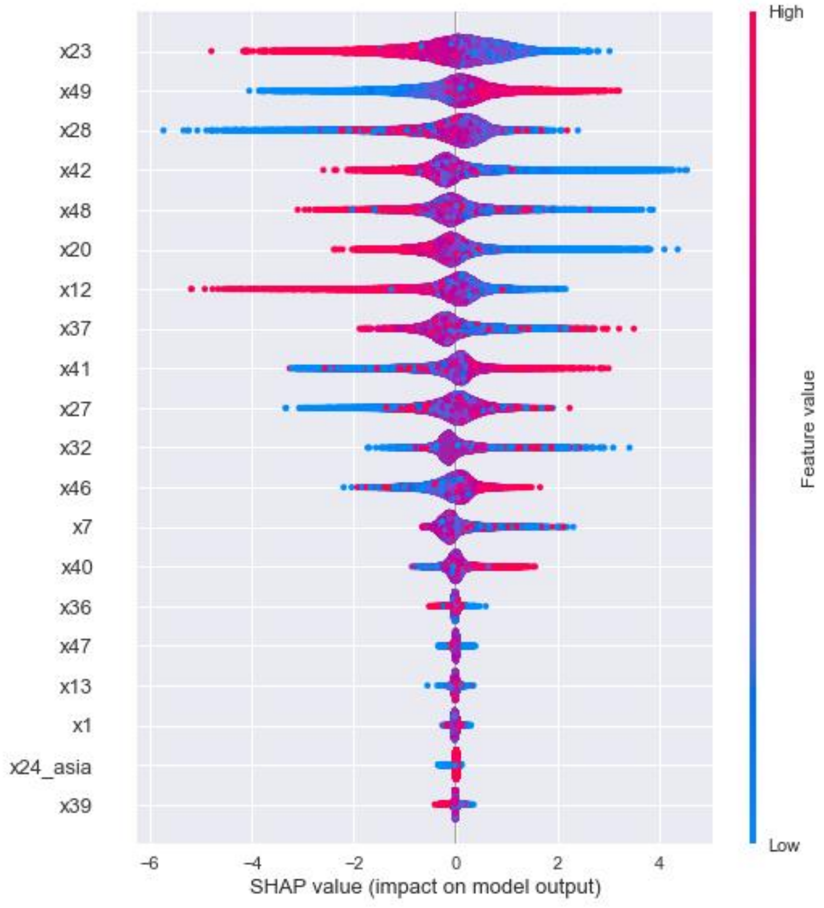


Fig. 23. Global Explainability using Shapley Values.

Specifically, looking at 'x23', we see that the higher values ('red' dots) of this feature are associated with negative Shapley values (X-axis) that contribute to reducing the output variable towards 0. Conversely, lower values (blue dots) are associated with a positive Shapley values (X-axis) which contributes to increasing the outcome towards 1. This inverse relation is essentially equivalent to negative correlation. On the other end of the spectrum, we can see that 'x49' exhibits that opposite behavior. Higher values (red) are associate with positive Shapley values in general and hence push the outcome towards 1, while lower values (blue) are associated with negative Shapley values and hence move the outcome towards 0. This direct relation is equivalent to positive correlation.

Additionally, some features are harder to interpret from this chart. For example, feature 'x27' shows a blob of purple (mix of red and blue) in the center. It is hard to decipher whether this is positively or negatively correlated with the outcome. A simplified representation of this chart along with partial dependence plots are an important tool that can clear some of this confusion.

The simplified version of Fig. 23 is shown Fig. 24 and it represents each feature with a single color indicating the directionality of the impact (red is associated with positive correlation and blue with negative correlation) [7]. The length of the bar represents the magnitude of the impact (features having longer bars have a higher impact and hence are globally more important). This is a little clearer to interpret compared to Fig. 23 and we can see that 'x27' is positively correlated in general (globally).

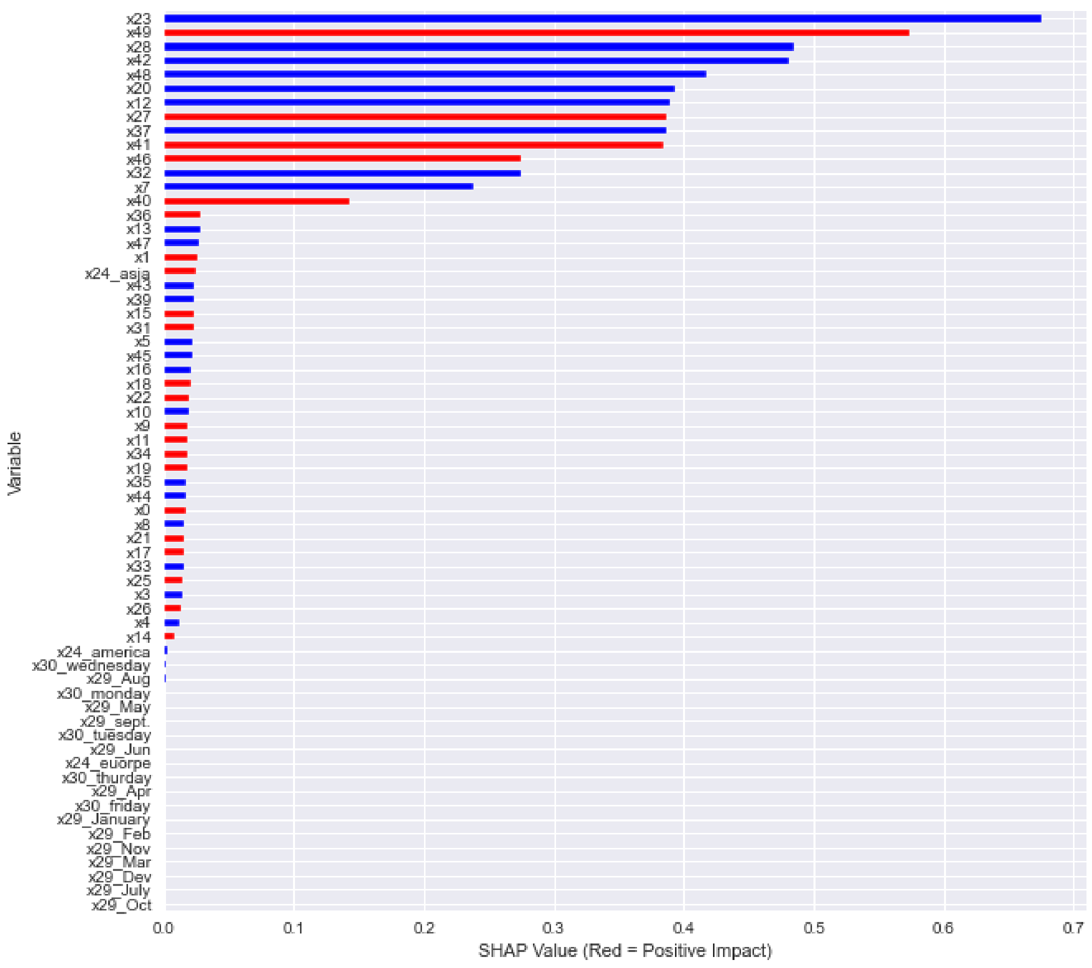
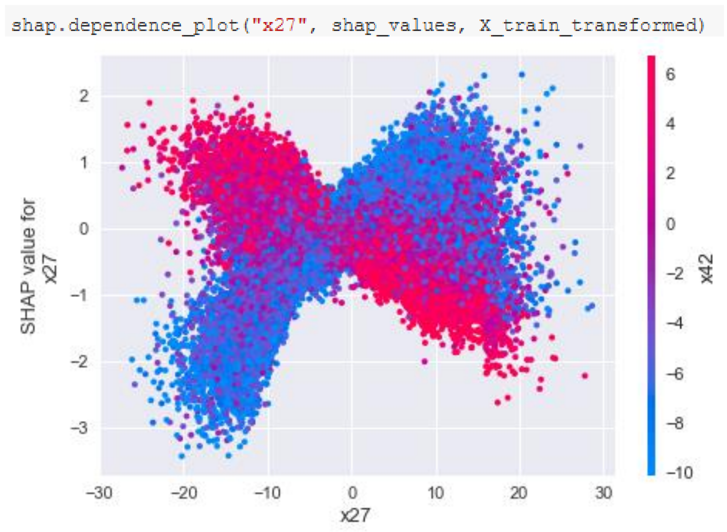


Fig. 24. Simplified Shapley Representation



```
shap.dependence_plot("x27", shap_values, X_train_transformed)
```

Fig. 25. Partial Dependence Plot for ‘x27’

Another way to look at ‘x27’ is through the partial dependence plots as shown in Fig. 25. Partial dependence plots represent the marginal impact (y-axis) of adding an additional feature (x-axis) into the model. This chart also shows each observation with a color. The color represents the value of an addition feature (‘x42’ in this case) that interacts the most with the feature under consideration (‘x27’) with respect to its mean value used during training. This figure shows some interesting findings. We can see that if we imagine the plot without the color, then it is not clear whether the feature ‘x27’ is positively correlated or negatively correlated with the outcome (similar to what we saw in Fig. 23). However, the addition of ‘x42’ clarifies this to a large extent. When ‘x42’ is lower than the mean value of ‘x42’ used in the training (blue dots), then ‘x27’ seems to be largely positively correlated with the outcome (a higher value of ‘x27’ on the X-axis is associated with a higher Shapley value on the y-axis). On the contrary, when ‘x42’ is lower than the mean value of ‘x42’ used during training (red dots), then ‘x27’ seems to be negatively correlated to the outcome. This is interesting since it points to potential conflicts between the global explainability and local explainability. Global explainability indicated that ‘x27’ was positively correlated with the outcome (one blanket statement), but local explainability seems to indicate that this is in fact context dependent (particularly related to the value of ‘x42’). Even then, some observations that have a higher value of ‘x42’ compared to the mean (red) still show up intermixed with the region predominantly containing blue dots and vice versa, which again points to the fact that local explainability may be dependent on other factors beyond ‘x42’ as well.

Side note: Another observation from Fig. 23 is that we now have a feature called “x24_asia”. There was no such feature in our original dataset. This is a result of the automatic one hot encoding that is performed by PyCaret internally which created this new feature for modeling. As indicated above, this is another advantage of using PyCaret. It takes care of preprocessing the data appropriately and the modeling engineer need not take care of this explicitly.

5.2 Local Explainability

We indicated in the section above section that having global explainability may not be enough and we may have to dive deeper to understand the nuances of what factors are correlated with the outcome. This can again be accomplished with the help of Shapley Values. Fig. 26 shows the local explainability for a single observation which was picked at random. The subset of the feature values for this observation is shown in Table 6. This table also shows the corresponding mean value for these features that was used during training. Fig. 26 represents the impact that each feature has on the outcome of the prediction. `f(x)` represents a function that is directly proportional to the outcome variable ‘y’. Features in red help to move the prediction higher towards ‘1’ whereas features in blue help to move the prediction lower towards ‘0’. The magnitude of the red or blue bars for each feature shows the magnitude of the local impact of feature on this single observation.

Diving a little deeper, we recall from Fig. 23 that ‘x23’ was negatively correlated with the outcome. In this case, the value of ‘x23’ is lower than the mean (Table 6), hence it helps to push the outcome higher. Conversely, ‘x49’ was positively correlated to the outcome, but its value in this case is higher than the mean so it also helps to push the outcome higher. It is interesting to note that while the global importance indicated that ‘x23’ and ‘x49’ were the most important features, local importance for this observation seems to indicate that while these two features are important for this observation as well, ‘x41’ is the most impactful (biggest bar is associated with this feature).

Finally, looking at the impact of ‘x27’, we see that it is above the mean and its interacting variable ‘x42’ is also above the mean. Per the partial dependence plot in Fig. 25, we would have expected ‘x27’ to be negatively correlated to the outcome and hence pull the value towards 0. However, we observe the opposite here (it is pushing the value higher). This likely indicates that this is one of the red dots that is intermingled with the predominantly blue sections in Fig. 25. This again points to the importance of local explainability and the fact that feature importance can be context dependent and we should be careful while dealing with explaining feature importance with a broad (global) brush.

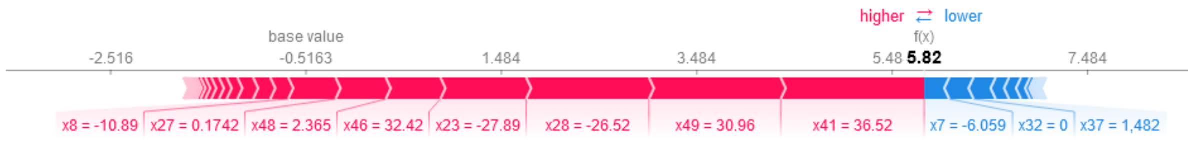


Fig. 26. Local Explainability for a Single observation.

Table 6. Mean Training Values and Single Observation Values for a Subset of Features

Feature	Mean Training Value	Observation Value	Above or Below Mean
x23	0.73	-27.89	Below
x49	-0.71	30.96	Above
x27	0.00	0.17	Above
x42	-1.86	5.57	Above

6. Conclusion

In this case study, we were provided with an anonymized dataset of roughly 160,000 observations containing 50 features and were asked to model the target binary variable. Not much context was provided around the data, so we decided not to impute missing values (instead, we removed observations with missing data since these represented less than 1% of the data). However, a couple of important pieces of information were provide which guided the decisions taken in this case study.

(1) All mistakes are costly, but some are more costly than others. In this case, a false negative was 50 times more costly ($500) compared to a false positive ($10).

(2) Model interpretability was an important consideration.

In order to choose the best model for the dataset, we made use of an AutoML framework (PyCaret) which helped us find the optimal model while reducing the coding and time overhead. This allowed for rapid prototyping. The best baseline model obtained was an Extreme Gradient Boosted Model (`XGBoost`). This model had an unbiased cost of $22.68 per observation which was more than 20% better than the next best model (cost of $28.59 per observation) which was also another gradient boosted model. However, we were able to fine tune this `XGBoost` model further using hyperparameter tuning and the resultant unbiased cost was reduced to $17.90 per observation (an improvement of 18.5% over the baseline model).

Additionally, global and local interpretation was detailed using Shapley Values and Partial Dependence plots. Global importance can be a valuable tool to get a quick overall picture of the factors that influence the model's decisions in general. However, this may miss out on the impact of features on individual observations which might be important in certain regulated industries such as banking and healthcare. Hence local explainability is also an important consideration. Luckily, both these can be calculated using Shapley values and the process was detailed in this case study. While we provided the example of local explainability using a single observation, the same idea can be extended to any observation in the dataset. With the power of global and local feature importance, a subject matter expert having the context behind this dataset will be able to determine the factors influencing predictions and appropriately tie them back to the business use case.

Finally, we made some assumptions about the validity of the dataset and the cost benefit matrix and expected this to remain consistent in the future. If this were to change, then we should rerun this analysis with the new distribution and cost structure and redeploy a new model more suitable to the new structure in the future.

7. References

[1] A. Moez, "Welcome to PyCaret," 2020. [Online]. Available: https://pycaret.org/guide/. [Accessed 26 November 2020].

[2] R. Rajiv, "Confusion matrix," 14 September 2016. [Online]. Available: https://alearningaday.blog/2016/09/14/confusion-matrix/. [Accessed 25 November 2020].

[3] "Analytics Vidya | Shapley values," [Online]. Available: https://www.analyticsvidhya.com/blog/2019/11/shapley-value-machine-learning-interpretability-game-theory/. [Accessed 28 11 2020].

[4] M. Christoph, "Partial Dependence Plot (PDP)," in *Interpretable Machine Learning*, 2020.

[5] "XGBoost," [Online]. Available: https://xgboost.readthedocs.io/en/latest/python/python_api.html. [Accessed 28 11 2020].

[6] "Machine Learning Mastery | XGBoost," [Online]. Available: https://machinelearningmastery.com/tune-number-size-decision-trees-xgboost-python/. [Accessed 28 11 2020].

[7] "Explain Your Model with the SHAP Values," Towards Data Science, [Online]. Available: https://towardsdatascience.com/explain-your-model-with-the-shap-values-bc36aac4de3d. [Accessed 28 11 2020].

[8] A. Moez, "Classification Module," 2020. [Online]. Available: https://pycaret.org/classification/. [Accessed 26 November 2020].

[9] "scikit-learn," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. [Accessed 04 10 2020].

8. Appendix

The entire code used for this case study is available in the following GitHub repository: Repository Link