COMP6200 – Critical Analysis Task

Name : Santhosh Kumar Srinivasan

Student ID : 48057827

Introduction:

In a critical analysis task, the primary objective is to identify and address shortcomings and constraints within the data science portfolio. The goal is to improve the code's performance, resulting in a more efficient, accurate, and less time-consuming model. The dataset in question revolves around the loan records released by Lending Club. By leveraging these attributes, our aim is to assess the model's accuracy and evaluate the correctness of the analysis. This process allows us to apply the knowledge gained from the course, thoroughly examine the issue at hand, and make necessary adjustments to achieve the most optimal outcomes.

Issue 1 : Class Imbalance of Minority Class

In this critical analysis, it's crucial to acknowledge the challenge of an imbalanced target variable, where certain classes within a dataset are underrepresented compared to others, a situation known as "class imbalance." This imbalance can lead to biased machine learning models that perform well on the majority class but struggle with the minority class. The class imbalance can be observed in the below screenshot with the counts of 0 and 1 of the feature ‘credit.policy’.

Code and values count before the change:

```
In [19]: from sklearn.model_selection import train_test_split
         New_Data.sort_values(by=['credit.policy'])
         print(New_Data['credit.policy'].value_counts())
```

```
1    7710
0    1868
Name: credit.policy, dtype: int64
```

To mitigate this issue, one common strategy is "upsampling" the minority class, which involves increasing the number of instances in the underrepresented class to create a more balanced dataset. This approach aims to help models learn from both classes more effectively, ultimately leading to improved predictions for the minority class. In this analysis, you should discuss the implications of class imbalance, the benefits of upsampling, and its impact on model performance while considering any relevant context-specific factors. In the below screenshot, we can see the upsampling of minority class (0) for better predictions.

Code and values count after the change:

```
In [24]: from imblearn.over_sampling import RandomOverSampler
         ros = RandomOverSampler(random_state=0)

In [28]: X = New_Data.copy().drop(columns=['credit.policy', 'int.rate','revol.bal', 'inq.last.6mths', 'not.fully.paid' ])
         y = New_Data.copy()['credit.policy']
         X_resampled, y_resampled = ros.fit_resample(X, y)

In [29]: y_resampled.value_counts()

Out[29]: 1    7710
         0    7710
         Name: credit.policy, dtype: int64
```

In this way, upsampling aims to provide the learning model with a more balanced dataset, helping it learn from both classes more effectively and improving its ability to make accurate predictions for the minority class.

Issue 2 : Train-Test Split is not in Stratified Manner

The second issue is related to the way you split your dataset into training and testing subsets. This issue becomes more prominent due to the class imbalance problem we discussed earlier. When the train-test split is not done in a stratified manner, it can lead to an uneven distribution of class labels in the training and testing sets, which in turn can affect the model's performance.

In the below screenshot, we can see the train test split is done with the given dataset as it is:

```
x_ex1_train = x_ex1_array[0:int((len(y_ex1_array)+1)*0.9),:]
x_ex1_test = x_ex1_array[int((len(y_ex1_array)+1)*0.9):,:]
y_ex1_train = y_ex1_array[0:int((len(y_ex1_array)+1)*0.9)]
y_ex1_test = y_ex1_array[int((len(y_ex1_array)+1)*0.9):]
```

Class imbalance, as we discussed earlier, means that one class may be significantly smaller in size compared to another class. If you don't use a stratified split, there's a risk that a majority of instances from the overrepresented class end up in either the training or testing set, leaving the other set with very few instances of that class. This can lead to skewed training and evaluation, making it challenging for the model to generalize effectively.

To address this issue, I would recommend implementing a stratified train-test split. In a stratified split, the proportions of each class within the target variable are maintained in both the training and testing sets. This ensures that the model is exposed to a representative sample of each class during training and evaluation. By doing so, you'll create a more balanced and fair testing environment, allowing your model to better handle class imbalances and make more reliable predictions.

In the below screenshot, the code changes to make the train test split in stratified manner:

```
In [69]: x_ex1_train, x_ex1_test, y_ex1_train, y_ex1_test = train_test_split(
             x_ex1_array, y_ex1_array, stratify=y_ex1_array, test_size=0.1, random_state=42
         )
```

Issue 3: Data Leakage in Feature Transformation

The third issue in our model is because that we have used "fit_transform" on both the training and test data, the model can inadvertently learn characteristics of the test data, compromising its ability to make accurate predictions on unseen data. This practice essentially blurs the line between training and testing, potentially leading to over-optimistic model performance.

The code we used for feature transformation is shown in below screenshot:

```
In [11]: from sklearn.preprocessing import StandardScaler

         obje_ss=StandardScaler()

         x_ex1_train=obje_ss.fit_transform(x_ex1_train)
         x_ex1_test=obje_ss.fit_transform(x_ex1_test)
```

To mitigate data leakage, it's crucial to fit any feature transformations exclusively on the training data. This ensures that the model learns from the training data only, preserving the integrity of the test set as a true measure of model performance. By following this approach, we can maintain a clear separation between the data used for training and evaluation, promoting more reliable and unbiased results.

The right code for feature transformation and to avoid data leakage is shown in the below screenshot:

```
In [70]: from sklearn.preprocessing import StandardScaler

         obje_ss = StandardScaler()

         x_ex1_train = obje_ss.fit_transform(x_ex1_train)
         x_ex1_test = obje_ss.transform(x_ex1_test)
```

Issue 4: Significance of Negatively Correlated Features

The statement that "negative correlations are useless" is not entirely accurate. Negative correlation between features in a dataset can actually provide valuable information. Negative correlation implies that as one feature increases, the other tends to decrease, and this relationship can be meaningful in various analytical contexts.

```
From the heatmaps, we can find different correlations between each feature and 'credit.policy'.
We only reserve features that have positive correlations with 'credit.policy' by removing all features, i.e., 'int.rate',
'revol.bal', 'inq.last.6mths' and 'not.fully.paid' which are negatively correlated with 'credit.policy'. We believe that
features with negative correlations are useless for model training.
```

So, it's important to be cautious about labeling negatively correlated features as "useless". Negative correlations can be just as informative as positive correlations, depending on the problem we are trying to solve.

Conclusion:

By Identifying the above issues, our critical analysis successfully identified and resolved key issues in our model. These included class imbalance, non-stratified train-test splits, and data leakage during feature transformation. As a result, the model's training success rate dropped to 68.2302925493587%, and the testing success rate improved to 68.28793774319067%, which is shown in the below screenshots.

The training and testing success rate before code changes:

```
In [21]: from sklearn.metrics import r2_score,classification_report,accuracy_score, plot_confusion_matrix
         final_model = clf.best_estimator_
         y_pred=final_model.predict(x_ex1_test)
         print(final_model)
         print('Train success rate: %',final_model.score(x_ex1_train,y_ex1_train)*100)
         print('Test success rate: %',accuracy_score(y_ex1_test,y_pred)*100)
         plot_confusion_matrix(final_model, x_ex1_test, y_ex1_test)
```

```
DecisionTreeClassifier(max_depth=2)
Train success rate: % 95.59215868228745
Test success rate: % 32.9153605015674
```

The training and testing success rate after code changes:

```
In [80]: from sklearn.metrics import r2_score,classification_report,accuracy_score, plot_confusion_matrix
         final_model = clf.best_estimator_
         y_pred=final_model.predict(x_ex1_test)
         print(final_model)
         print('Train success rate: %',final_model.score(x_ex1_train,y_ex1_train)*100)
         print('Test success rate: %',accuracy_score(y_ex1_test,y_pred)*100)
         plot_confusion_matrix(final_model, x_ex1_test, y_ex1_test)
```

```
DecisionTreeClassifier(criterion='entropy', max_depth=2, min_samples_split=0.1)
Train success rate: % 68.2302925493587
Test success rate: % 68.28793774319067
```

These changes demonstrate the importance of addressing data-related challenges to enhance model performance, ultimately achieving a more robust and accurate model.