Assignment 4: Report

Sajja Patel - 2021101107, Bhumika Joshi - 2022121006

24 October 2024

1 Preprocessing

In this project, we first loaded the training dataset and we handled missing values by dropping any rows that contained `NaN` values, ensuring a clean dataset for training. The features were then divided into two categories: numerical and categorical. For the numerical columns, we applied standard scaling using the `StandardScaler` which normalizes the numerical data by transforming the features to have zero mean and unit variance. Next, for the categorical columns, we used `LabelEncoder` to convert binary categorical variables like `Gender` and `Ever_Married` into numerical values (0 or 1). For the remaining categorical variables, we applied one-hot encoding using the `pd.get_dummies` function, which created dummy variables for each category, allowing the machine learning model to interpret them.

To address class imbalance in the target variable `Segmentation`, we applied the Synthetic Minority Oversampling Technique (SMOTE). SMOTE generates synthetic samples for minority classes by interpolating between existing samples, balancing the distribution of classes in the dataset.

2 Model Training and Evaluation

2.1 One vs All Classifier

After preprocessing, the dataset was split into training and testing sets using an 80-20 ratio, ensuring that 20% of the data was kept aside for testing. We then trained a Support Vector Machine (SVM) classifier with a radial basis function (RBF) kernel. The `class_weight='balanced'` argument was specified in the SVM to handle the class imbalance by assigning higher weights to the minority classes. This ensures that the model does not favor the majority classes during training.

The SVM model was trained on the training data and subsequently used to predict the target variable (`Segmentation`) for the test set. The performance of the model was evaluated using the accuracy score, confusion matrix, and classification report. The confusion matrix provides insights into how well the model distinguishes between different classes, while the classification report provides a detailed breakdown of precision, recall, and F1-score for each class. Finally, the predicted labels were saved into a CSV file.

```
Accuracy: 42.97%
Confusion Matrix:
[[34 25 19 22]
 [16 39 19 14]
 [14 16 63  8]
 [27 17 22 29]]
Classification Report:
              precision    recall  f1-score   support

           A       0.37      0.34      0.36       100
           B       0.40      0.44      0.42        88
           C       0.51      0.62      0.56       101
           D       0.40      0.31      0.35        95

    accuracy                           0.43       384
   macro avg       0.42      0.43      0.42       384
weighted avg       0.42      0.43      0.42       384
```

Figure 1: Using SMOTE

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best Parameters: {'C': 1, 'gamma': 1, 'kernel': 'rbf'}
Accuracy with best parameters: 48.18%
Confusion Matrix:
[[44 19 16 21]
 [14 53  7 14]
 [21 11 54 15]
 [35 14 12 34]]
Classification Report:
              precision    recall  f1-score   support

           A       0.39      0.44      0.41       100
           B       0.55      0.60      0.57        88
           C       0.61      0.53      0.57       101
           D       0.40      0.36      0.38        95

    accuracy                           0.48       384
   macro avg       0.49      0.48      0.48       384
weighted avg       0.49      0.48      0.48       384
```

Figure 2: Using GridCV and cross validation

2.2 One vs One Classifier

The OvO classifier implements a One-vs-One (OvO) classification approach using Support Vector Machines (SVM) for multi-class classification on a customer segmentation dataset. It first loads and preprocesses the data by removing missing values, standardizing numeric features, and encoding categorical variables. The dataset is then split into training and validation sets. In OvO, we create binary classifiers for every possible pair of classes (e.g., "A vs B"). For each class pair, an SVM model is trained to distinguish between the two. During prediction, the OvO classifiers are applied to each instance, where each classifier votes for one of the two classes it was trained on. The class with the most votes is selected as the final predicted label for that instance.

```
Resampled class distribution: Counter({'B': 392, 'D': 385, 'A': 380, 'C': 379})
Accuracy: 36.98%
Confusion Matrix:
[[37 23 14 26]
 [21 30 20 17]
 [14 24 50 13]
 [34 17 19 25]]
Classification Report:
              precision    recall  f1-score   support

           B       0.35      0.37      0.36       100
           D       0.32      0.34      0.33        88
           C       0.49      0.50      0.49       101
           A       0.31      0.26      0.28        95

    accuracy                           0.37       384
   macro avg       0.37      0.37      0.37       384
weighted avg       0.37      0.37      0.37       384
```

Figure 3: Using SMOTE

```
Accuracy: 36.60%
Predictions saved to 'ovo_predictions.csv'.
Confusion Matrix:
[[81  0  0 12]
 [48  1  0  7]
 [56  1  0  1]
 [69  0  0 30]]
Classification Report:
              precision    recall  f1-score   support

           B       0.32      0.87      0.47        93
           D       0.50      0.02      0.03        56
           C       0.00      0.00      0.00        58
           A       0.60      0.30      0.40        99

    accuracy                           0.37       306
   macro avg       0.35      0.30      0.23       306
weighted avg       0.38      0.37      0.28       306
```

Figure 4: OvO without SMOTE

3 Random Forest Implementation

For preprocessing, I handled missing values by removing any entries with `NaN`, encoded categorical features using `LabelEncoder`, and normalized numerical features with `StandardScaler`.

In my implementation of the Random Forest algorithm, I defined a custom class called `CustomRandomForest` to encapsulate the functionality of this ensemble learning model. I started by initializing parameters like the number of trees (`n_estimators`), the maximum depth for each tree, and criteria for splitting nodes. To enhance the model's generalization ability, I employed the bootstrapping method, which generates random samples from the training data, allowing each tree to be trained on a unique subset of data.
As I fitted the model, I created a decision tree for each bootstrap sample using the `DecisionTreeClassifier`, selecting entropy as the criterion for splitting nodes. This approach helped in capturing the complexity of the data. After training, I aggregated the predictions from all trees through majority voting, which ensured that the final output reflected the most common class predicted by the individual trees. By leveraging the

strengths of multiple decision trees, I aimed to achieve higher accuracy and robustness compared to using a single decision tree.

- **Precision**:
  - *Macro Precision*: This is the average precision across all classes, treating each class equally regardless of its size. It gives a sense of how well the model performs across different categories.
  - *Micro Precision*: This is calculated globally by counting the total true positives and false positives across all classes. It provides a more holistic view of the model's performance, especially in imbalanced datasets.
- **Recall**:
  - *Macro Recall*: This metric averages the recall across all classes, giving equal weight to each class. It helps in understanding how well the model can identify instances of each class.
  - *Micro Recall*: Similar to micro precision, this metric is calculated globally by counting total true positives and false negatives across all classes, providing an overall effectiveness measure in identifying relevant instances.
- **F1 Score**:The F1 score is the harmonic mean of precision and recall. It combines both metrics into a single score that reflects the model's overall performance in identifying positive instances while minimizing false positives.

4 Analysis

- **How does class imbalance affect multiclass classification, and what strategies can be employed to mitigate its impact, especially with small datasets?**

  Class imbalance in multiclass classification occurs when some classes have significantly fewer examples than others. This imbalance can negatively affect model performance, as the classifier may become biased toward the majority classes and fail to learn meaningful patterns for the minority classes. This can lead to poor generalization for minority classes, with low precision, recall, and F1-scores. It's impacts include: Bias toward majority classes, decreased performance on minority classes and misleading evaluation metrics masking poor performance on minority classes.

  To mitigate this, we can try out the following:

  - Oversampling the Minority Classes: Duplicate or synthetically generate more instances of the minority classes using techniques like SMOTE (Synthetic Minority Over-sampling Technique) to balance the class distribution.
  - Class Weighing: Algorithms like SVM and decision trees allow assigning higher weights to the minority classes during training, forcing the model to give more attention to them. This penalizes misclassification of minority classes more heavily than majority ones.
  - Cross-Validation with Stratified Sampling: When performing cross-validation, use stratified sampling to ensure that each fold has a representative distribution of classes, which ensures fairer model evaluation across the classes.

- **How can the choice of hyperparameters make the random forest classifier and SVM classifiers more prone to over or under fitting?**

  The choice of hyperparameters in machine learning models like Random Forest and Support Vector Machine (SVM) classifiers is important for performance, particularly concerning overfitting and underfitting. In Random Forests, parameters such as the number of trees (n_estimators) and the maximum depth of trees (max_depth) significantly influence the model's ability to generalize. Too few trees can lead to underfitting, while too many can cause overfitting, especially if the trees are allowed to

grow too deep. Similarly, the minimum samples required for splitting (min_samples_split) and at leaf nodes (min_samples_leaf) need careful tuning to avoid overly simplistic or excessively complex models.

For SVM classifiers, the choice of kernel type and the regularization parameter (C) are critical. A linear kernel may underfit if the data is complex, while a complex kernel like RBF with a high gamma can lead to overfitting by fitting noise in the training data. The C parameter influences flexibility; a high C can restrict the model too much, leading to underfitting, whereas a low C might allow too much error, increasing the risk of overfitting. The gamma parameter in RBF kernels also plays a significant role; low values can oversimplify the model, while high values can complicate it excessively.

- **Plot the confusion matrix and include the precision, recall, f1-score metrics in the report.**

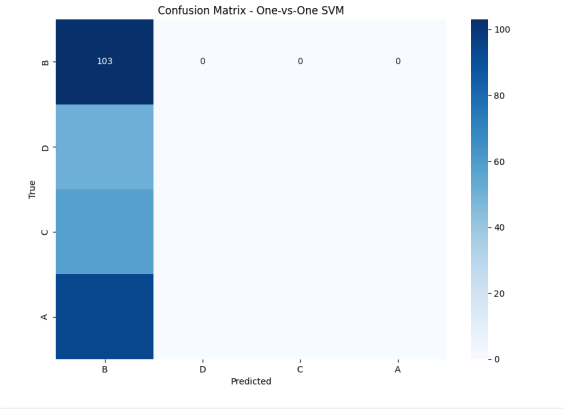

Figure 5: OvA without SMOTE: Confusion Matrix

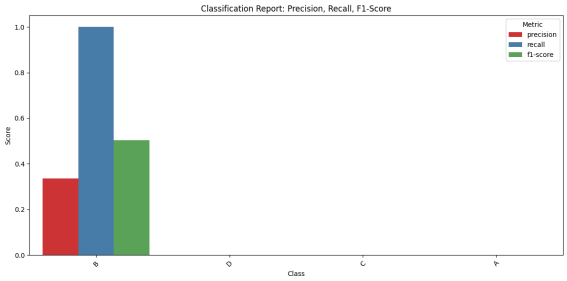

Figure 6: OvA without SMOTE: Classification Report

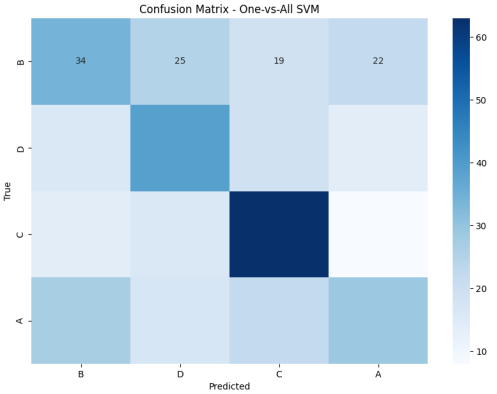

Figure 7: OvA with SMOTE: Confusion Matrix

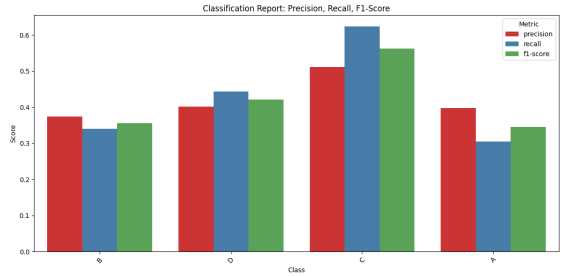

Figure 8: OvA with SMOTE: Classification Report

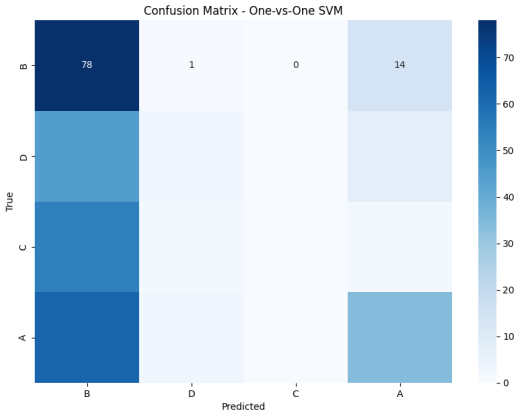

Figure 9: OvO without SMOTE: Confusion Matrix

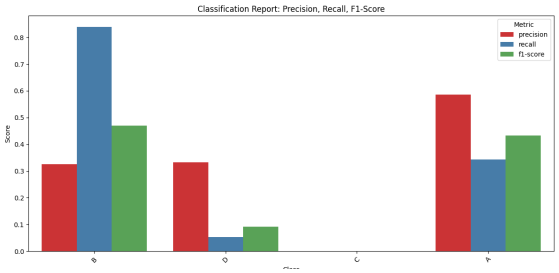

Figure 10: OvO without SMOTE: Classification Report

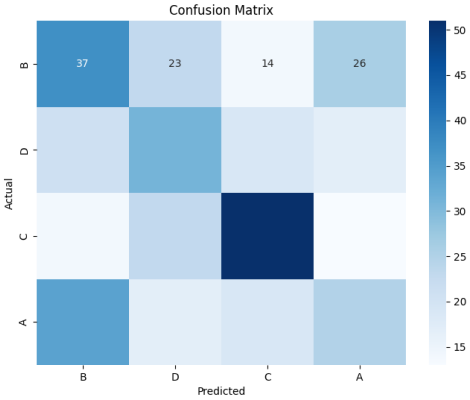

Figure 11: OvO with SMOTE: Confusion Matrix

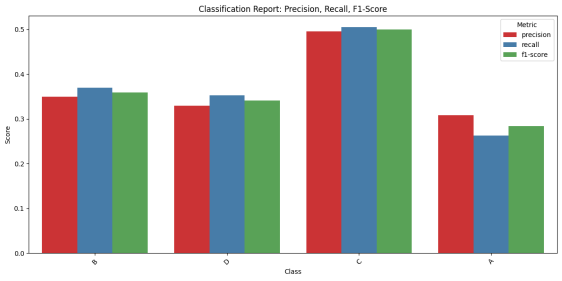

Figure 12: OvO with SMOTE: Classification Report

- **Compare the results obtained for one-vs-one and one-vs-all (which according to you performs better for the above dataset?)**

  From the above results we can see that OvA performs better than OvO overall for the given dataset. Also, we can see that that using SMOTE increases performance and helps improve the performance of minority classes substantially.