Human Protein Atlas Bad Cell Recognition

Gabi Rivera, Itzel Cruz, and Edgar Rosales

Shiley-Marcos School of Engineering, University of San Diego

Table of Contents

Abstract

The "Human Protein Atlas - Single Cell Classification" dataset, from the 2021 challenge, supports the development of automated cell classification methods. The challenge centered on classifying cells from human tissue images, where segmentation quality was a key issue, affecting evaluation scores. The dataset includes ~9,600 manually labeled cell images, with each cell categorized as suitable or unsuitable based on segmentation quality, specifically requiring at least 50% of the nucleus to be visible. Labels are provided in the "IsBad" column of the "cell_stats_with_gt.csv" file, with corresponding multi-channel images available for analysis.

In this study, XGBoost, SVM, Neural Networks, Logistic Regression, K-means, and Agglomerative Clustering models were evaluated using different key metrics such as precision, recall, F-1 score and AUC. XGBoost outperformed other models with an AUC of 0.87, closely followed by Neural Networks with an AUC of 0.86. SVM and Logistic Regression both achieved an AUC of 0.82. The unsupervised methods, K-means and Agglomerative Clustering, performed poorly, with AUCs of 0.41 and 0.44, respectively. These results highlight the potential of XGBoost and Neural Networks as effective tools for accurate cell classification in biomedical research, with applications in diagnostic automation and cellular anomaly detection.

Keywords: Neural Network, Support Vector Machines, Logistic Regression, XGBoost, Agglomerative Clustering, K-means Clustering, segmentation error, Bad-cell recognition, binary classification

Human Protein Atlas Bad Cell Recognition

Single-cell analysis is a transformative technique that offers invaluable insights into cellular functions and mechanisms within a population. By examining individual cells in isolation, a deeper understanding of cellular processes and interconnections can be further explored for potential drug discovery avenues. In immunology for example, single-cell analysis enables the characterization of rare immune cell types and their responses which is critical for developing novel immunotherapies (Stubbington et al., 2017). In cancer research, this technique can identify distinct tumor cell subpopulations and interpret their roles in disease progression, leading to more targeted and effective therapies (Caron et al., 2022). Overall, single-cell analysis is essential for unlocking new frontiers in biological research and advancing personalized medicine.

Recently, the integration of Artificial Intelligence (AI) and Deep Learning into single-cell recognition platforms has greatly improved the efficiency and scalability of high-throughput image analysis. These advanced computational techniques excel at processing and interpreting the large volumes of data produced by single-cell experiments, which traditionally required extensive manual effort and time. However, challenges in segmentation errors remain a critical issue. In 2021, the Kaggle Human Protein Atlas Single-Cell Classification Challenge was launched, inviting participants to develop models that are capable of accurately segmenting and classifying individual cells. In response, Husain and colleagues developed an ensemble model that combines the strengths of U-Net for segmentation and ResNet for classification to enhance the accuracy of the protein localization results (Husain et al., 2023). The following year, HPA Bad Cell Detection Challenge released the segmentation results generated by the ensemble

model, aiming to enhance their system's ability to identify the segmentation errors using manually labeled data.

In response, the aim of this study is to develop a high-performing model that effectively identifies and mitigates the segmentation errors observed in the previous ensemble model. We will be using the generated features from the ensemble, along with the manually labeled outcome variable, to assess the accuracy of the selected models. Several models will be explored including K-means clustering, Agglomerative clustering, Logistic Regression, Neural Networks (NN), XGBoost (XGB), and Support Vector Machines (SVM). The goal is to identify the most effective model for accurately classifying cells as 'bad'. The model performance will be evaluated using precision, recall, and AUC to assess the effectiveness of the binary classification results.

Method

General Information of Dataset

The cell_stats_with_gt.csv dataset offers a comprehensive collection of statistical information about individual cells, with each row representing a specific cell identified by a unique `Id`. It includes a range of features derived from image analysis, such as the `heights` and `widths` of bounding boxes, `aspect_ratios`, `bbox_areas`, `mask_areas`, and `mask_perimeters`, which collectively describe the cell's size and shape. The dataset also incorporates shape compactness through the `compactness` feature. Additionally, it provides color-related features, detailing the count of red, green, blue, and yellow points within each cell's area, as well as their corresponding mean and maximum values (`red_pts_mean`, `green_pts_mean`, etc.). The `isBad` column serves as a binary target variable, indicating whether a cell is categorized as bad (1) or good (0). This dataset is well-suited for machine learning applications, particularly in tasks like classification and anomaly detection in cell image analysis.

Exploratory Data Analysis

Data exploration guided the approaches during the data wrangling step. There were no missing values found across the dataset using the Klib missing values function. Therefore, no artificial data creation was adopted to address that issue. Another Klib tool, the Klib distribution plot, was utilized to produce 27 density feature visualizations (see Figure 1). The density plots detailed various important elements such as mean, media, skew, and kurtosis values. This was enhanced with the use of Seaborn boxplot shown in Figure 2, which added the sense of scaling difference across the features. Taking the two visualizations into account, it is evident that heteroscedasticity or non-normality must be addressed during feature transformation to meet the distribution assumptions before modeling.

A pairwise plot was also generated to visualize the relationship of each feature and the outcome variable ‘isBad’, as sampled in Figure 3. The interpretation of the paired features revealed a mix of positive and negative correlations, as well as non-linear patterns. The main observation in Figure 3 however, is the striking unbalanced distribution between the outcome classes. This indicates that the dataset needs to be rebalanced to help the model characterize the minority class more effectively, especially since it represents the label for badly classified cells. The last visualization was the Pearson correlation plot that revealed highly correlated features across the dataset (see Figure 4). To reduce redundancy and avoid multicollinearity, this issue is addressed at an 80% selection criteria during data wrangling. This step will be essential in enhancing the model’s performance and simplifying the interpretability of the modeling process.

Data Wrangling and Pre-Processing

During the data wrangling and pre-processing phase, several essential steps were carried out to prepare the `cell_stats_with_gt.csv` dataset for machine learning tasks. Initially, the dataset was loaded into a pandas DataFrame, facilitating easier manipulation and analysis. The data was then cleaned by checking for and addressing any missing values, outliers, or inconsistencies, ensuring the integrity of the dataset. Relevant features were selected, and the target variable `isBad` was separated from the feature set. In some cases, new features were derived from existing ones to enhance the model's predictive capabilities.

Next, the dataset was split into training and testing sets using a standard 80-20 split, allowing the model to be trained on a portion of the data and tested on unseen data for performance evaluation. Given the imbalance in the target variable `isBad`, the Synthetic Minority Over-sampling Technique (SMOTE) was applied to the training set to generate synthetic samples for the minority class, thereby preventing the model from becoming biased towards the majority class. The features were then normalized or standardized to ensure they were on a similar scale, a crucial step for algorithms like SVM, which are sensitive to the scale of input data.

Additionally, the data was converted into a format suitable for machine learning algorithms, including the conversion of categorical variables into numerical formats where necessary. For models using OpenCV’s DNN module, images were converted into blobs—preprocessed input images suitable for neural networks. Through these data wrangling and pre-processing steps, the dataset was meticulously prepared for training robust and reliable machine learning models, ensuring that the features were well-scaled, balanced, and representative of the underlying patterns.

Data Splitting

The dataset was split into training (80%) and testing (20%) sets, with `y` representing the binary target (isBad). Initially, the training set was imbalanced, with 6,718 majority class instances (isBad = 0) and 1,708 minority class instances (isBad = 1). To address this, we applied Synthetic Minority Over-sampling Technique (SMOTE), balancing the training set to 6,718 instances per class, resulting in a total of 13,436 samples.

The resampled training set (X_train_resampled) contained 18 features across 13,436 samples, while the testing set (X_test) had 2,107 samples with the same 18 features. This balanced approach aimed to improve model performance and fairness by ensuring equal representation of both classes during training.

Model Strategies

K-means Algorithm

The K-means clustering algorithm was employed to identify distinct clusters within the dataset, with an initial focus on determining the optimal number of clusters through the Elbow method. The method involved varying the number of clusters (k) from 1 to 10 and calculating the within-cluster sum of squares (WCSS) for each value. The Elbow plot indicated that k = 3 was the optimal choice, balancing between model complexity and variance explanation. K-means was then applied to the resampled training data, and the model produced three distinct clusters.

Given that K-means is an unsupervised learning algorithm, the clusters were subsequently mapped to binary labels for evaluation against the true classification labels. The resulting AUC-ROC score of 0.41 suggests that while the algorithm was able to group similar

data points together, these clusters did not align closely with the actual binary classes in the data. The Silhouette Score of 0.135 indicates moderate cluster cohesion but suggests that some data points may be ambiguously placed between clusters.

In summary, while K-means was effective in identifying natural groupings within the data, its performance as a classification tool was limited by its unsupervised nature. This highlights the algorithm's utility for exploratory data analysis rather than as a primary classification method.

Agglomerative Clustering Algorithm

Agglomerative Clustering, a hierarchical clustering method, was applied to the dataset to identify underlying group structures without prior knowledge of class labels. The model was configured using the Ward linkage criterion, which minimizes the variance within each cluster. The number of clusters was set to 3, based on preliminary analysis and comparison with the K-means clustering results.

The Agglomerative Clustering model, like K-means, operates in an unsupervised manner, which necessitated mapping the resulting clusters to binary labels to evaluate its classification performance. The AUC-ROC score of 0.44 reflects the model's challenge in accurately differentiating between the two binary classes based on the clusters formed. The Silhouette Score of 0.14 further indicates that while the clusters are somewhat distinct, there is room for improvement in cluster cohesion.

In conclusion, Agglomerative Clustering provided valuable insights into the hierarchical structure of the data, offering an alternative perspective to K-means. However, its utility in direct

classification tasks is constrained, highlighting the model's strengths in exploratory data analysis rather than predictive accuracy.

Logistic Regression Algorithm

An initial hyperparameter tuning was conducted to determine the optimal configuration for the Logistic Regression model. The tuning process explored a range of regularization parameters (C = [0.01, 0.1, 1, 10, 100]) and solvers (liblinear and lbfgs). The Logistic Regression model was evaluated using 5-fold cross-validation to assess its performance across multiple folds. The AUC-ROC scoring metric identified the best combination of parameters as C = 1.0 and using the lbfgs solver. The C = 1.0 value represents a moderate level of regularization, balancing the trade-off between underfitting and overfitting. The lbfgs solver, which stands for Limited-memory Broyden-Fletcher-Goldfarb-Shanno, is an optimization algorithm using the quasi-Newton method and is well-suited for the large-scale dataset used. The Logistic Regression model was then finalized with best parameters mentioned.

Neural Network Algorithm

The hyperparameter tuning was carried out to optimize the complex architecture of the neural network model. The tuning process used the Keras Tuner library and involved defining a model-building function, build_model, which specified two dense layers with varying units and a final output layer with a sigmoid activation. The first dense layer had several units (units1) ranging from 32 to 128 in steps of 32, while the second layer (units2) had units ranging from 16 to 64 in steps of 16. The learning rate for the Adam optimizer was also tuned which varied logarithmically between 1×10^{-4} and 1×10^{-1}. Hyperband search was employed with a maximum

of 20 epochs, with tuning process starting from epochs 7, and 2 iterations to find the optimal parameter configuration. The best performing configuration with a validation accuracy score at 92%, yielded a value of 64 units for both dense layers and a learning rate of 0.386%. The algorithm was then set with the optimal parameters to finalize the Neural Network model.

XG Boost Algorithm

XGBoost, or Extreme Gradient Boosting, is a powerful machine learning library specifically designed to enhance the performance of predictive models. It is particularly well-suited for gradient boosted decision trees (GBDT) and is known for its scalability and distributed nature. In supervised machine learning, algorithms train models to identify patterns in labeled datasets and then apply those patterns to predict labels for new data.

The classification report from the XGBoost model indicates an overall accuracy of about 86.1%. The model excels in predicting the majority class (`0`), with precision, recall, and F1-scores all hovering around 0.91. This demonstrates that the model is highly effective at correctly identifying instances of the majority class with minimal errors. However, its performance on the minority class (`1`) is less robust, with a precision of 0.63, recall of 0.68, and an F1-score of 0.65. This suggests that while the model can identify some minority class instances, it struggles with misclassifications, resulting in more false positives and negatives. The macro average, which averages the scores across both classes, reveals lower overall scores (precision of 0.78, recall of 0.79, and F1-score of 0.78), highlighting the model's difficulty in evenly balancing performance across classes. The weighted average scores, closely aligned with the majority class performance, indicate that the model's accuracy is largely influenced by how well it handles the majority class. While the model is generally effective, these results suggest

that there is potential for improvement, particularly in enhancing its ability to handle the minority class more accurately and consistently.

SVM Algorithm

A support vector machine (SVM) is a machine learning algorithm that uses supervised learning to classify data and perform regression analysis. SVMs are great at binary classification problems, which involve separating data into two groups.

The SVM model achieves an overall accuracy of 86%, indicating a generally strong performance. However, a closer examination of the classification report reveals a significant disparity in the model's ability to handle the two classes. For the majority class (`0`), the model demonstrates high effectiveness, with a precision of 0.87, recall of 0.97, and an F1-score of 0.92. This suggests that the model is highly successful in correctly identifying and classifying the majority of positive instances with few errors. In contrast, the performance on the minority class (`1`) is considerably weaker, with a precision of 0.76, recall of 0.38, and an F1-score of 0.51. The low recall indicates that the model fails to identify a substantial portion of the minority class, leading to a high number of false negatives. The macro average metrics—particularly a recall of 0.68—reflect the overall imbalance in the model's performance across the classes. The weighted average metrics, more closely aligned with the majority class performance, suggest that while the model is accurate overall, it is primarily driven by its performance on the majority class. These results indicate that while the SVM model is effective, particularly for the majority class, there is room for improvement in its ability to accurately identify and classify instances of the minority class.

Validation and Testing

In the validation and testing phase, confusion matrices were utilized to evaluate the performance of various models, including K-means, Agglomerative Clustering, Logistic Regression, Neural Networks, XGBoost, and SVM. The confusion matrices provided a clear visual representation of each model's ability to correctly classify instances into true positive, true negative, false positive, and false negative categories. This allowed for a comprehensive comparison of model performance, highlighting areas where certain models excelled or struggled in terms of classification accuracy and balance between different classes. The analysis ensured that the strengths and weaknesses of each model were thoroughly assessed, contributing to informed decisions in selecting the most effective model for the classification task.

Results and Final Model Selection

XGBoost emerged as the top-performing model. It demonstrated the best balance of precision and recall with an AUC score of 87% (see Figure 5). XGBoost achieved a precision of 92% and recall of 91% which indicates a strong capability to both correctly identify bad cells and capture a high proportion of actual bad cells. The high AUC score confirms that XGBoost is highly effective in distinguishing between good and bad cells across various classification thresholds.

Table 1

Performance Evaluation: Precision and Recall

Model on Test Dataset	Precision on Class 0 of isBad	Recall on Class 0 of isBad
Logistic Regression	92%	75%
Neural Network	91%	91%
XGBoost	92%	91%

Support Vector Machines	87%	97%
K-Means Clustering	77%	52%
Agglomerative Clustering	77%	52%

The table above summarizes the performance of all models tested on the dataset, focusing on their precision and recall metrics for Class 0 (isBad). As reflected in both the table and the ROC curve (see Figure 5), XGBoost consistently outperformed the other models, solidifying its place as the most effective model for this classification task.

Conclusion

In this study, we aimed to address the critical issue of segmentation errors in single-cell analysis by evaluating various models for their effectiveness in identifying "bad" cells. The focus was on finding a model that not only identifies bad cells correctly but also minimizes false positives and false negatives. Given the manually labeled binary outcomes, our goal was to select the most effective model based on precision, recall, and AUC-ROC.

The performance evaluation underscores the effectiveness of XGBoost in addressing the problem of single-cell segmentation errors. Its high precision and recall along with the best AUC-ROC, make it the most reliable model for accurately identifying bad cells. Neural Networks also provide a strong performance and flexibility. This makes the Neural Network algorithm a viable alternative. Support Vector Machines may be considered if capturing the maximum number of bad cells is critical, despite the trade-off in precision. Both K-means and Agglomerative Clustering are not recommended for this classification task.

The insights gained from this study are essential for enhancing single-cell analysis workflows and improving the accuracy of cell segmentation. By bridging the realms of Artificial

Intelligence (AI) and Deep Learning, this research contributes to the development of reliable tools for cell classification and segmentation. These advancements not only benefit scientific research but also hold significant potential for medical applications, paving the way for more precise and effective approaches in future endeavors.

Figures

Figure 1

Klib Visualization of Select Feature Distribution

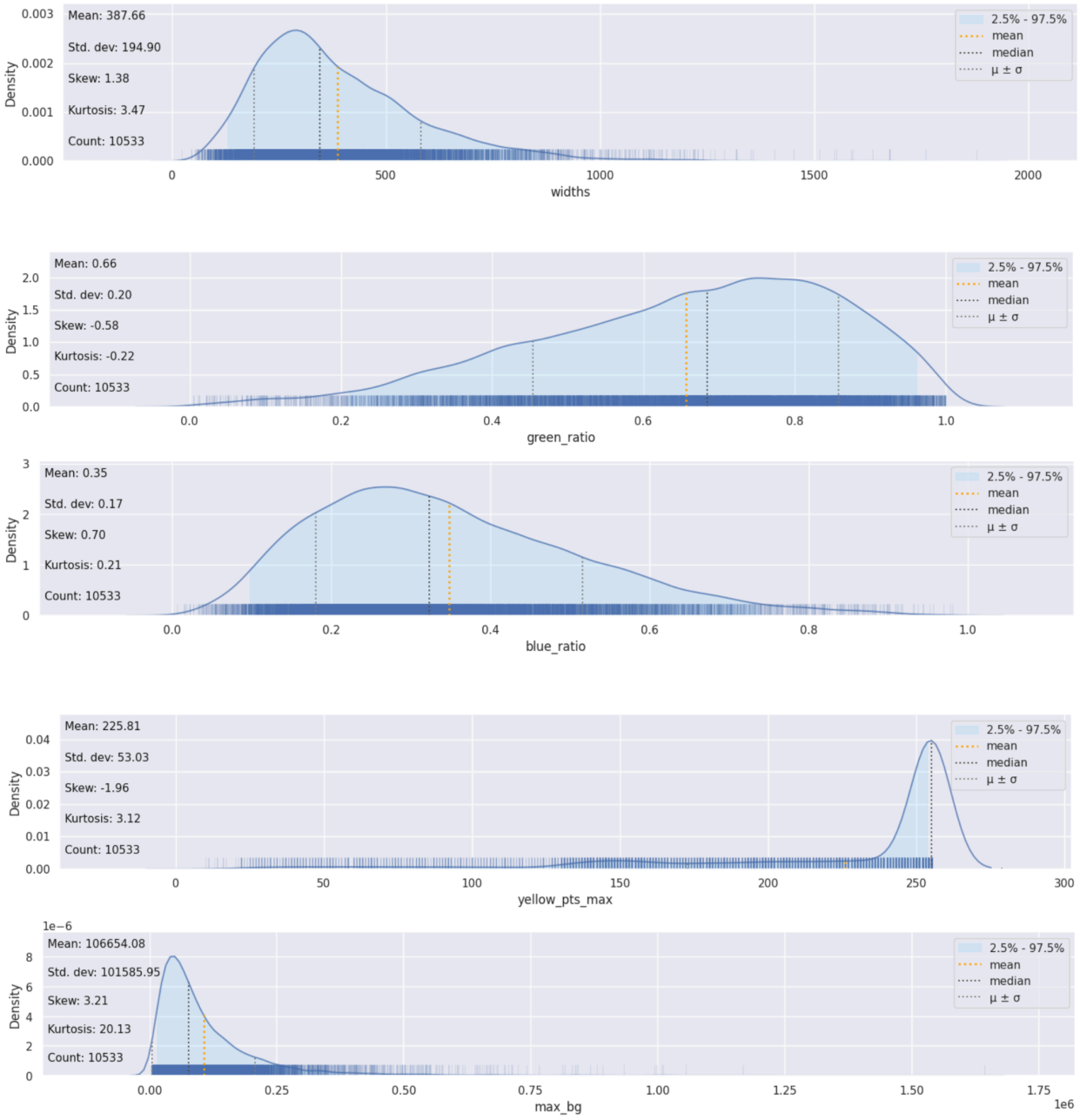


Figure 2

Boxplot of Features by isBad Outcome

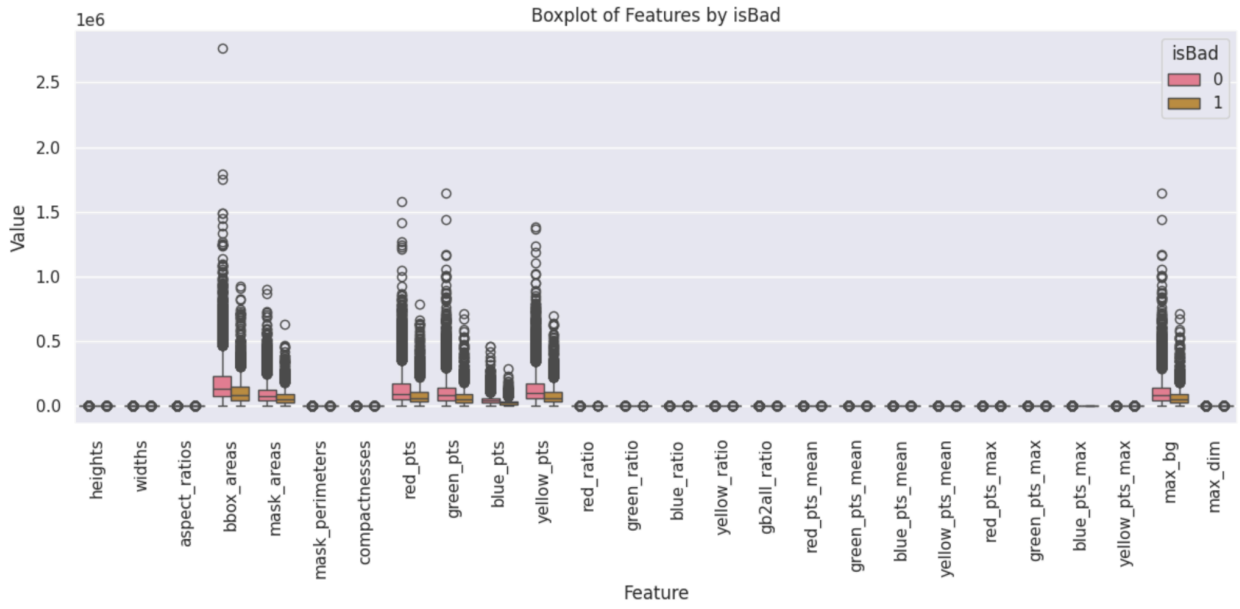
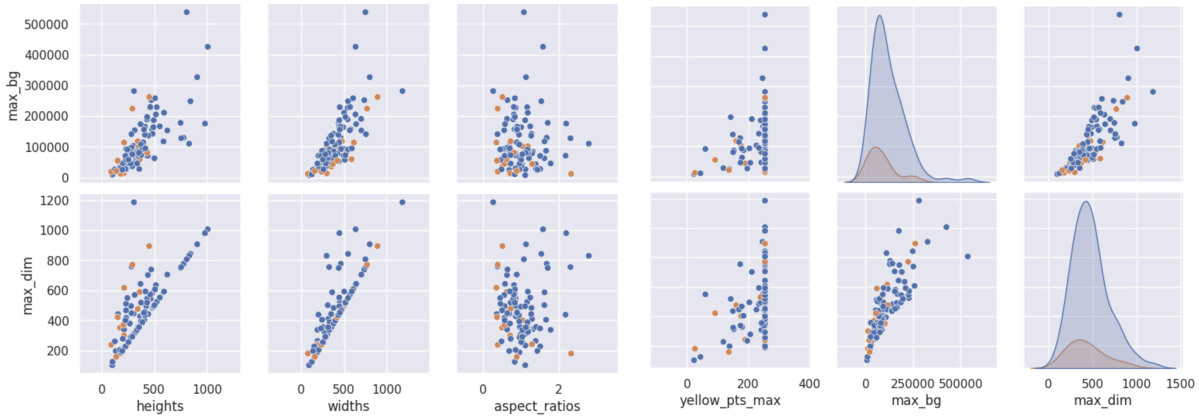


Figure 3

Pairwise Plot Sampled (Top) from the 27 Features by isBad Outcome Plot (Bottom)



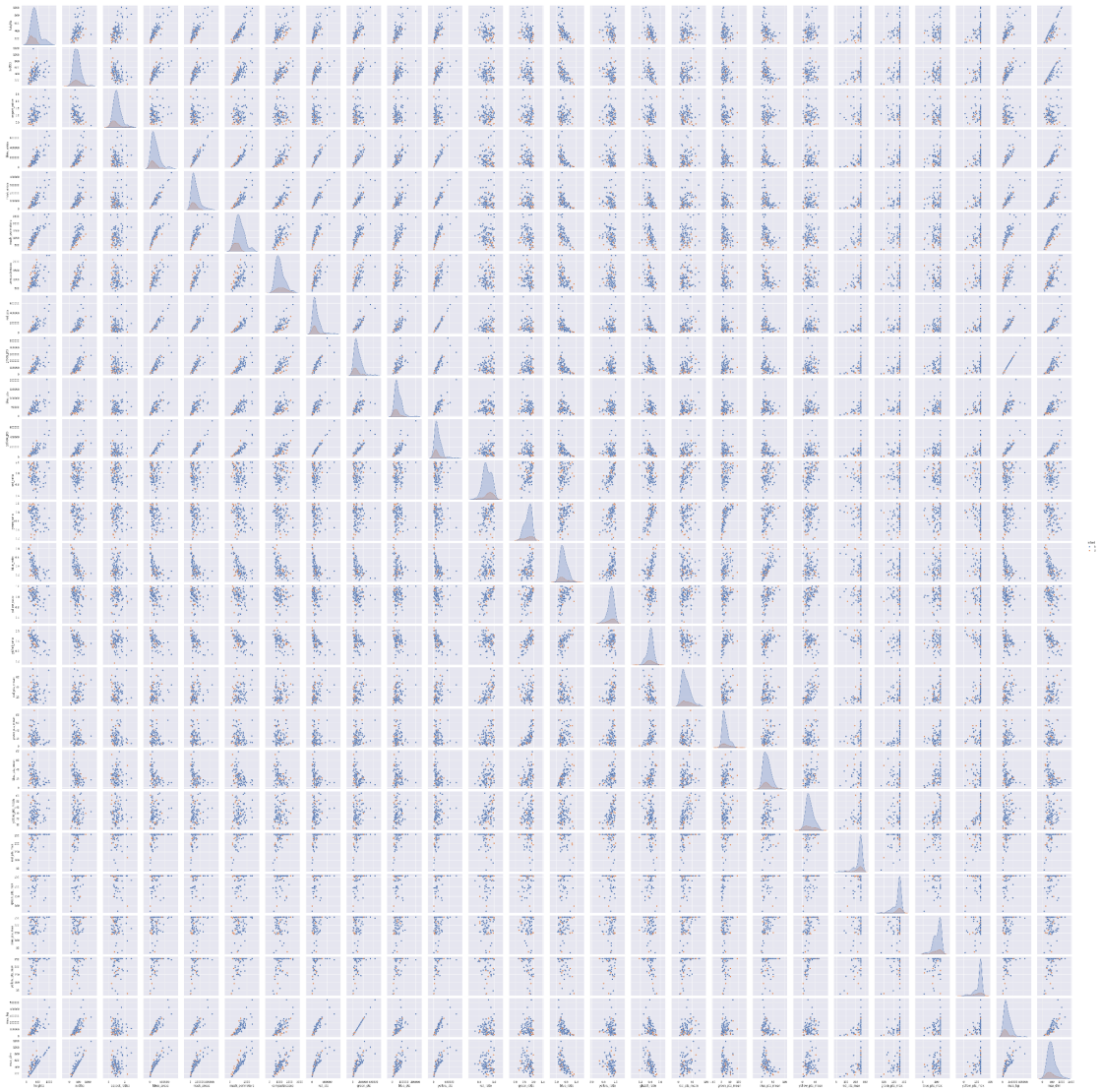


Figure 4

Pearson Correlation Plot

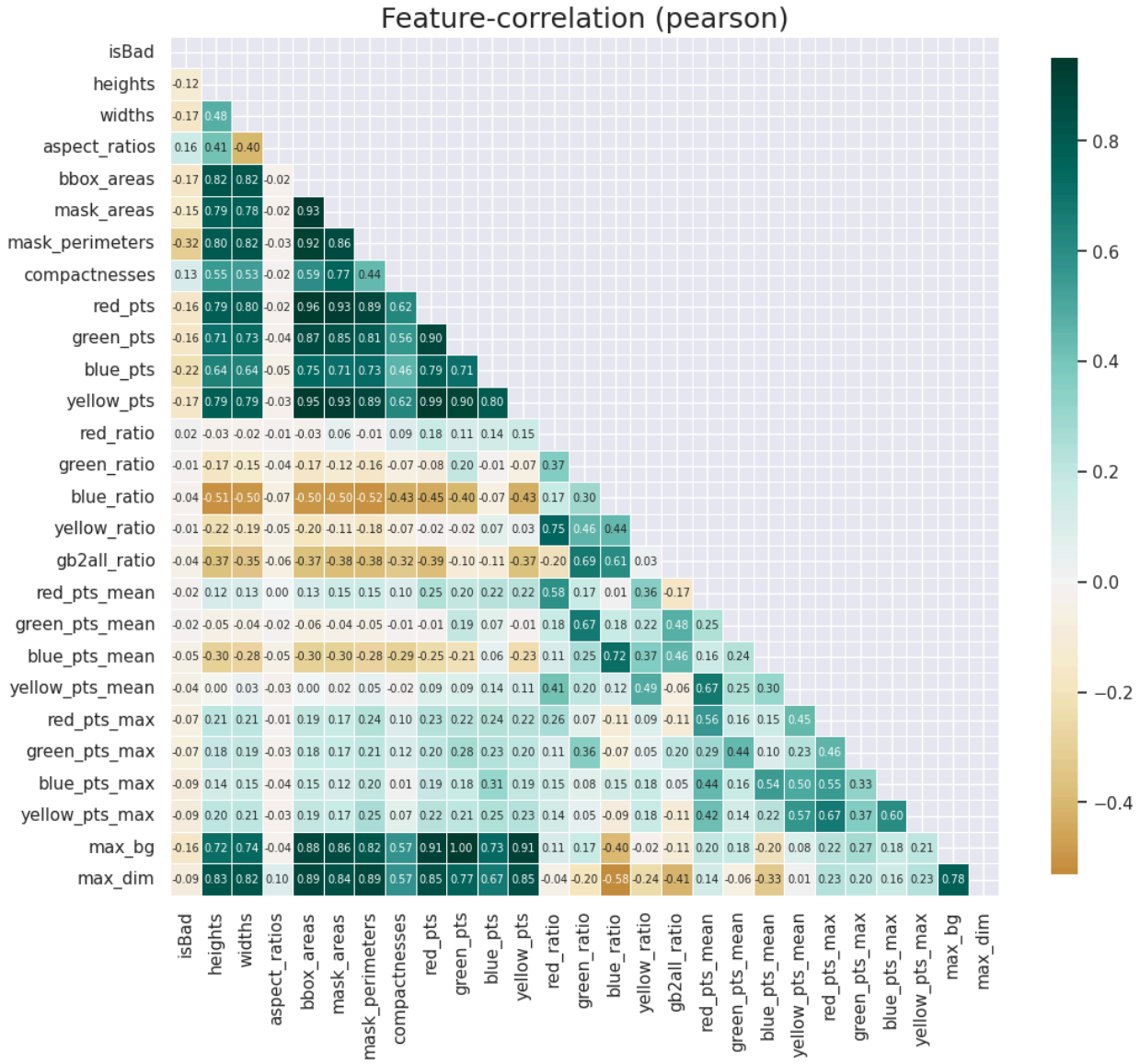
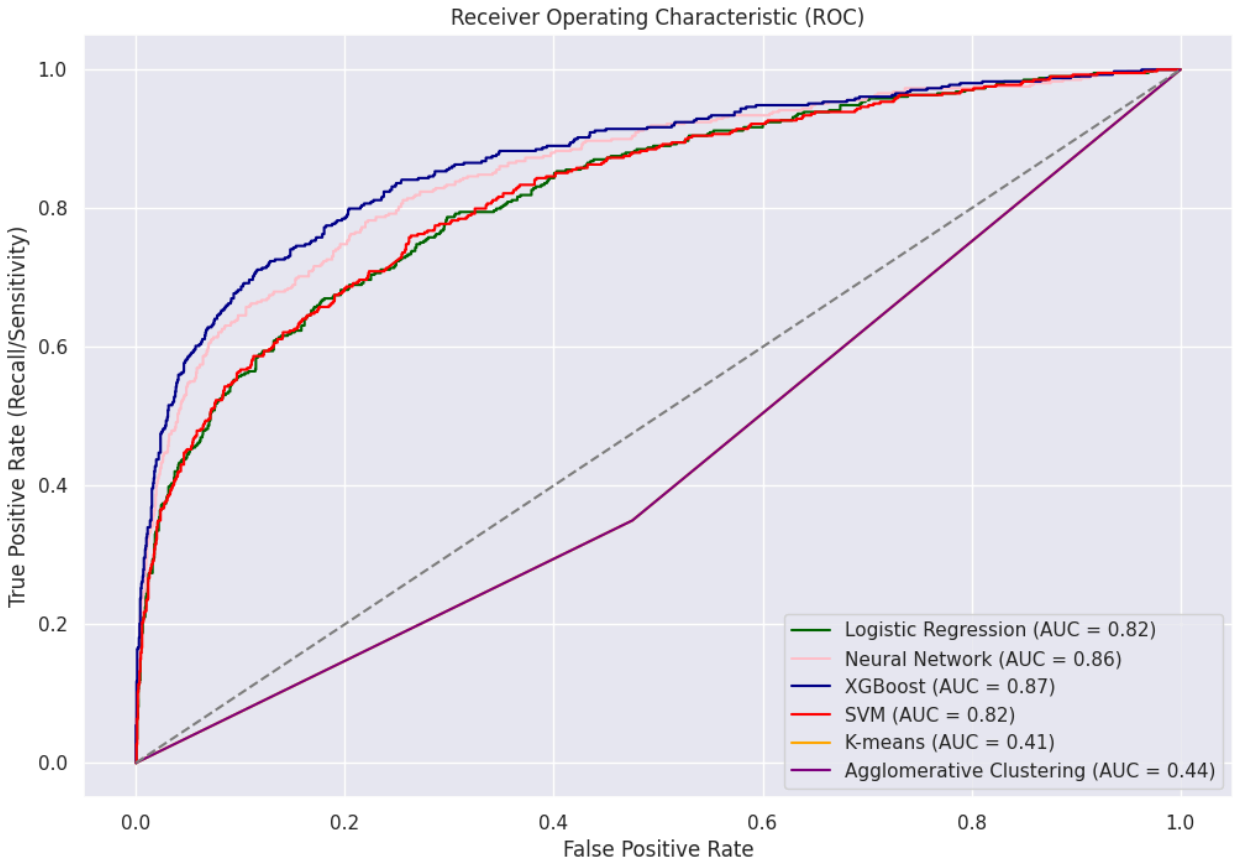


Figure 5

Combined AUC-ROC Plots for Model Performance Assessment



References

Caron, A., et al. (2022). *Single-cell transcriptomics of breast cancer reveals heterogeneous subpopulations with distinct immune microenvironments. Nature Communications*, 13, 1836. DOI: 10.1038/s41467-022-29676-7

Casper Winsnes, Emma Lundberg, Maggie, Phil Culliton, Trang Le, UAxelsson, Wei Ouyang. (2021). *Human Protein Atlas - Single Cell Classification*. Kaggle. https://kaggle.com/competitions/hpa-single-cell-image-classification

Husain, S.S., Ong, EJ., Minskiy, D. *et al.* Single-cell subcellular protein localisation using novel ensembles of diverse deep architectures. *Commun Biol* 6, 489 (2023). https://doi.org/10.1038/s42003-023-04840-z

Ong, E.-J., Minskiy, D., Bober-Irizar, M., Husain, S. S., & Bober, M. (2022). *HPA Bad Cell Detection* [Data set]. Kaggle. https://doi.org/10.34740/KAGGLE/DSV/3349136

Stubbington, M. J. T., et al. (2017). *T cell fate and clonality inference from single-cell transcriptomes. Nature Communications*, 8, 1080. DOI: 10.1038/s41467-017-01266-5

Appendix

PDF Code Link:

https://colab.research.google.com/drive/1v-ImvbEkSIWI658T_1euoirzEG-ILEgK?authuser=1#scrollTo=T9-9mqL74FYF

Final Project Team 6: Bad Cell Classification Model

Names: Gabi Rivera, Itzel Cruz, and Edgar Rosales
Course: ADS504-02 Machine Learning and Deep Learning

General Information

This project aims to improve the accuracy of an existing Human Protein Atlas (HPA) cell recognition algorithm by addressing segmentation errors. The focus is on developing a machine learning model capable of identifying populations of human cells that have been incorrectly segmented. Specifically, the model will target instances where cell structures are challenging to identify accurately, such as overlapping cells, irregular shapes, or varying intensities.

```
#Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
!pip install klib
!import klib
!pip install tabulate
from tabulate import tabulate
import klib
import warnings
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_sco
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.preprocessing import PowerTransformer
from sklearn.svm import SVC
!pip install imblearn
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
```

```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, roc_auc_score


warnings.filterwarnings("ignore")
```

```
Collecting klib
  Downloading klib-1.3.1-py3-none-any.whl.metadata (7.6 kB)
Requirement already satisfied: Jinja2<4.0.0,>=3.1.0 in /usr/local/lib/python3.
Requirement already satisfied: matplotlib<4.0.0,>=3.6.0 in /usr/local/lib/pyth
Requirement already satisfied: numpy<2.0.0,>=1.26.0 in /usr/local/lib/python3.
Requirement already satisfied: pandas<3.0,>=1.4 in /usr/local/lib/python3.10/d
Requirement already satisfied: plotly<6.0.0,>=5.11.0 in /usr/local/lib/python3
Requirement already satisfied: scipy<2.0.0,>=1.10.0 in /usr/local/lib/python3.
Collecting screeninfo<0.9.0,>=0.8.1 (from klib)
  Downloading screeninfo-0.8.1-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: seaborn>=0.12.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-pack
Downloading klib-1.3.1-py3-none-any.whl (23 kB)
Downloading screeninfo-0.8.1-py3-none-any.whl (12 kB)
Installing collected packages: screeninfo, klib
Successfully installed klib-1.3.1 screeninfo-0.8.1
/bin/bash: line 1: import: command not found
Requirement already satisfied: tabulate in /usr/local/lib/python3.10/dist-pack
Collecting imblearn
  Downloading imblearn-0.0-py2.py3-none-any.whl.metadata (355 bytes)
Requirement already satisfied: imbalanced-learn in /usr/local/lib/python3.10/d
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist
Requirement already satisfied: scipy>=1.5.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: scikit-learn>=1.0.2 in /usr/local/lib/python3.1
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Downloading imblearn-0.0-py2.py3-none-any.whl (1.9 kB)
Installing collected packages: imblearn
Successfully installed imblearn-0.0
```

```
# Upload Meta Data
df = pd.read_csv('cell_stats_with_gt.csv')
df.head()
```

	Id	isBad	heights	widths	aspect_ratios	bbox_areas	ma
0	40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_1374...	0	579	471	1.229299	272709	
1	40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_82_1...	0	759	491	1.545825	372669	
2	40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_1424...	0	623	495	1.258586	308385	
3	40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_850_...	0	587	439	1.337130	257693	
4	40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_290_...	0	791	507	1.560158	401037	

5 rows × 28 columns

```
# General Information
cell_status = df
cell_status.info()
cell_status.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10533 entries, 0 to 10532
Data columns (total 28 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Id               10533 non-null  object
 1   isBad            10533 non-null  int64
 2   heights          10533 non-null  int64
 3   widths           10533 non-null  int64
 4   aspect_ratios    10533 non-null  float64
 5   bbox_areas       10533 non-null  int64
 6   mask_areas       10533 non-null  int64
 7   mask_perimeters  10533 non-null  int64
 8   compactnesses    10533 non-null  float64
 9   red_pts          10533 non-null  int64
 10  green_pts        10533 non-null  int64
 11  blue_pts         10533 non-null  int64
 12  yellow_pts       10533 non-null  int64
 13  red_ratio        10533 non-null  float64
```

```
 14  green_ratio      10533 non-null  float64
 15  blue_ratio       10533 non-null  float64
 16  yellow_ratio     10533 non-null  float64
 17  gb2all_ratio     10533 non-null  float64
 18  red_pts_mean     10533 non-null  float64
 19  green_pts_mean   10533 non-null  float64
 20  blue_pts_mean    10533 non-null  float64
 21  yellow_pts_mean  10533 non-null  float64
 22  red_pts_max      10533 non-null  int64
 23  green_pts_max    10533 non-null  int64
 24  blue_pts_max     10533 non-null  int64
 25  yellow_pts_max   10533 non-null  int64
 26  max_bg           10533 non-null  int64
 27  max_dim          10533 non-null  int64
dtypes: float64(11), int64(16), object(1)
memory usage: 2.3+ MB
```

	isBad	heights	widths	aspect_ratios	bbox_areas	mask_
count	10533.000000	10533.000000	10533.000000	10533.000000	1.053300e+04	10533.0
mean	0.200987	388.589196	387.664768	1.134730	1.689077e+05	95710.7
std	0.400758	195.835398	194.898937	0.662509	1.570042e+05	80300.0
min	0.000000	37.000000	23.000000	0.103056	6.825000e+03	5492.0
25%	0.000000	249.000000	251.000000	0.745833	6.720000e+04	42667.0
50%	0.000000	347.000000	346.000000	1.000000	1.197960e+05	71840.0
75%	0.000000	482.000000	487.000000	1.346604	2.150160e+05	121803.0
max	1.000000	1955.000000	1879.000000	21.478261	2.762130e+06	902307.0

8 rows × 27 columns

Exploratory Data Analysis

```
# Identify if there are any missing values
klib.missingval_plot(cell_status)
```

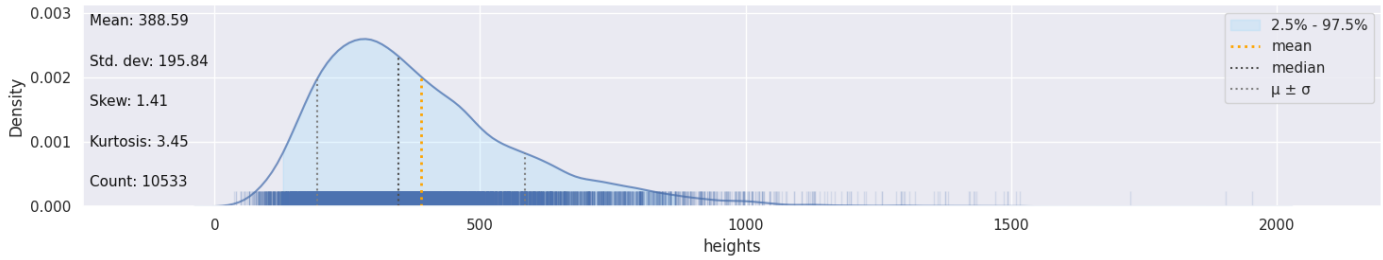
```
No missing values found in the dataset.
```

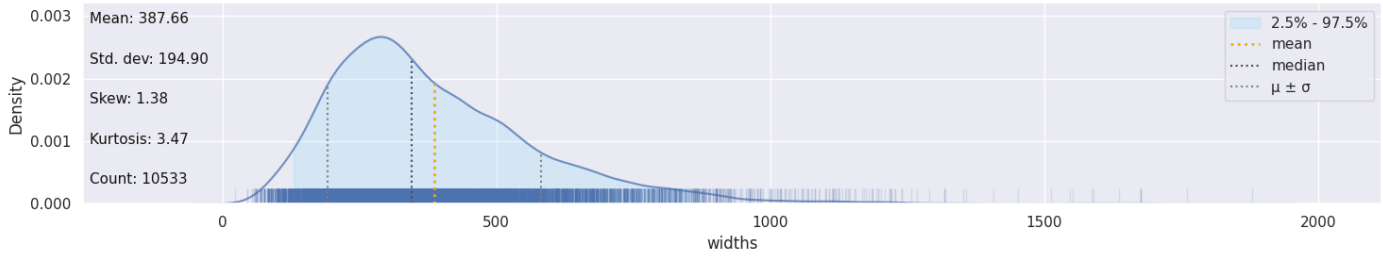
```
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

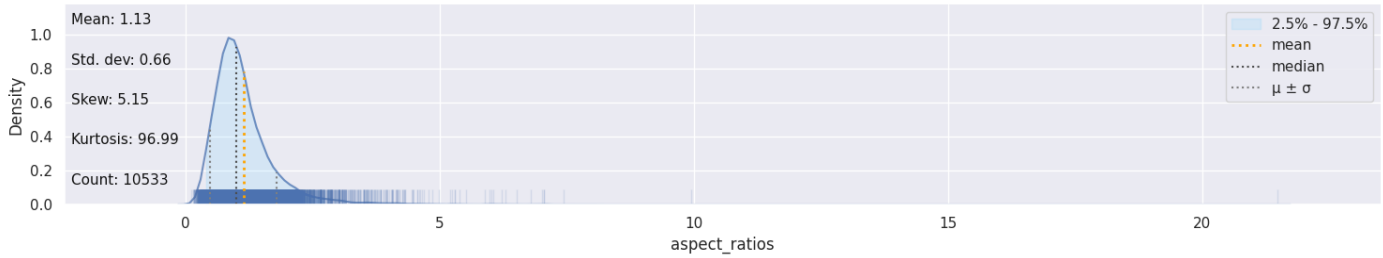
```
# Plot distributions for each group of numerical features
numerical_features = cell_status.select_dtypes(include=['int', 'float']).columns
num_groups = 27
group_size = len(numerical_features) // num_groups

feature_groups = []
for i in range(num_groups):
    start_idx = i * group_size
    end_idx = (i + 1) * group_size
    group_features = numerical_features[start_idx:end_idx]
    feature_groups.append(group_features)

for i, group_features in enumerate(feature_groups):
    print(f"Group {i+1} Features: {group_features}")
    klib.dist_plot(cell_status[group_features], showall=True)
```

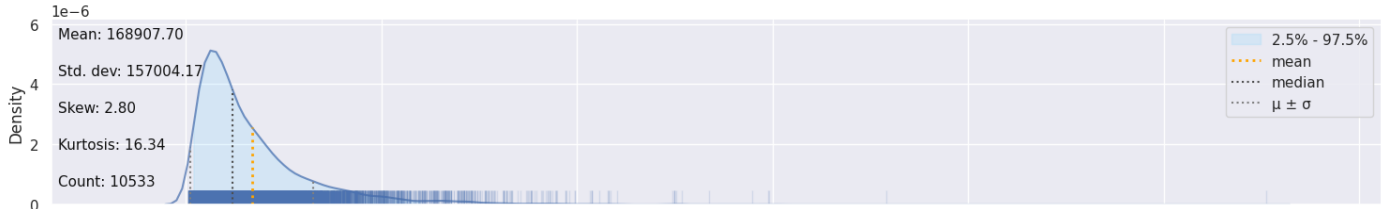
```
Group 1 Features: Index(['isBad'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
No columns with numeric data were detected.
Group 2 Features: Index(['heights'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 3 Features: Index(['widths'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 4 Features: Index(['aspect_ratios'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 5 Features: Index(['bbox_areas'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 6 Features: Index(['mask_areas'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 7 Features: Index(['mask_perimeters'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 8 Features: Index(['compactnesses'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 9 Features: Index(['red_pts'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 10 Features: Index(['green_pts'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 11 Features: Index(['blue_pts'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 12 Features: Index(['yellow_pts'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 13 Features: Index(['red_ratio'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 14 Features: Index(['green_ratio'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 15 Features: Index(['blue_ratio'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 16 Features: Index(['yellow_ratio'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
```

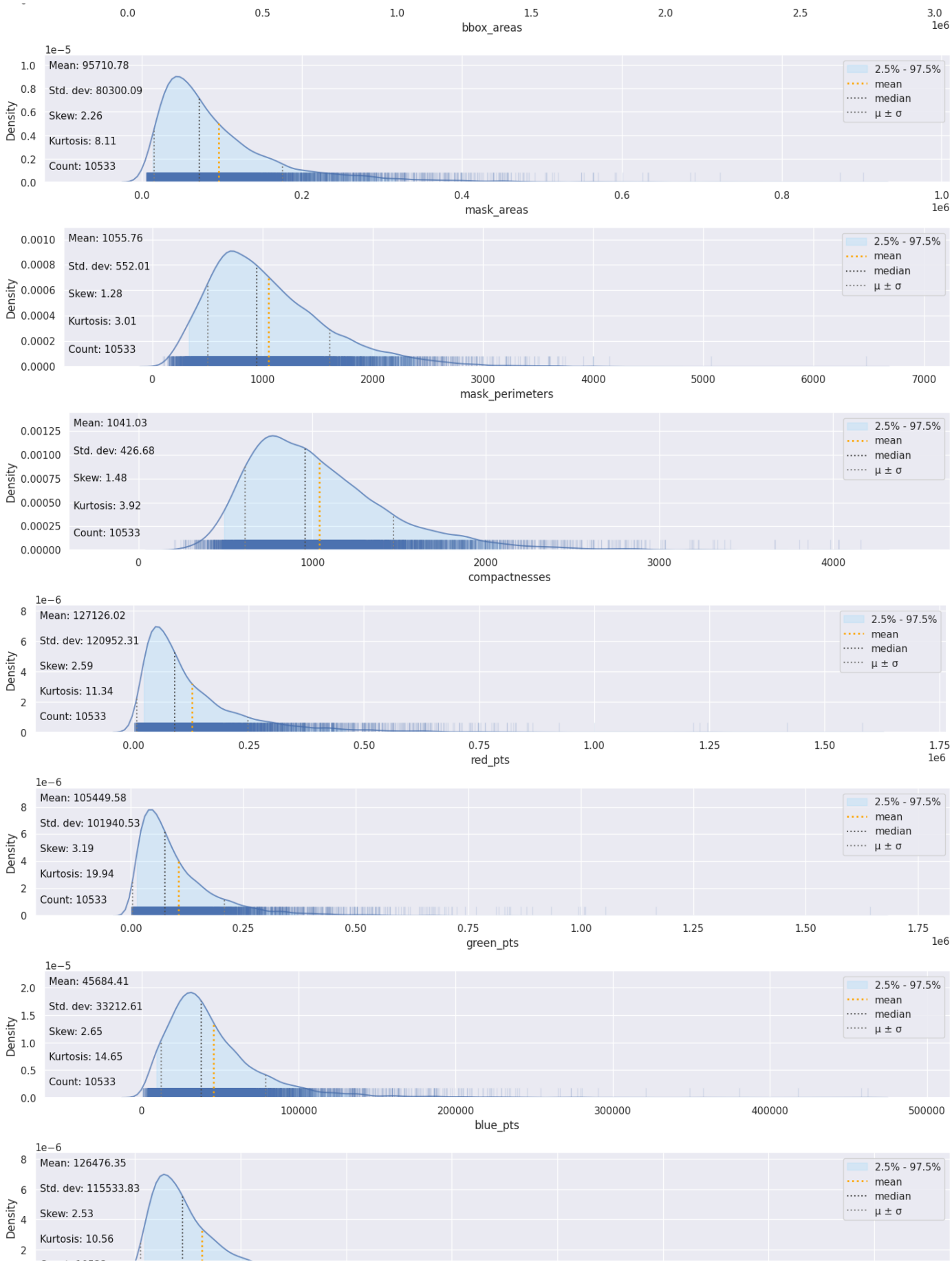
```
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 17 Features: Index(['gb2all_ratio'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 18 Features: Index(['red_pts_mean'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 19 Features: Index(['green_pts_mean'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 20 Features: Index(['blue_pts_mean'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 21 Features: Index(['yellow_pts_mean'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 22 Features: Index(['red_pts_max'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 23 Features: Index(['green_pts_max'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 24 Features: Index(['blue_pts_max'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 25 Features: Index(['yellow_pts_max'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 26 Features: Index(['max_bg'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
Group 27 Features: Index(['max_dim'], dtype='object')
Large dataset detected, using 10000 random samples for the plots. Summary stat
```



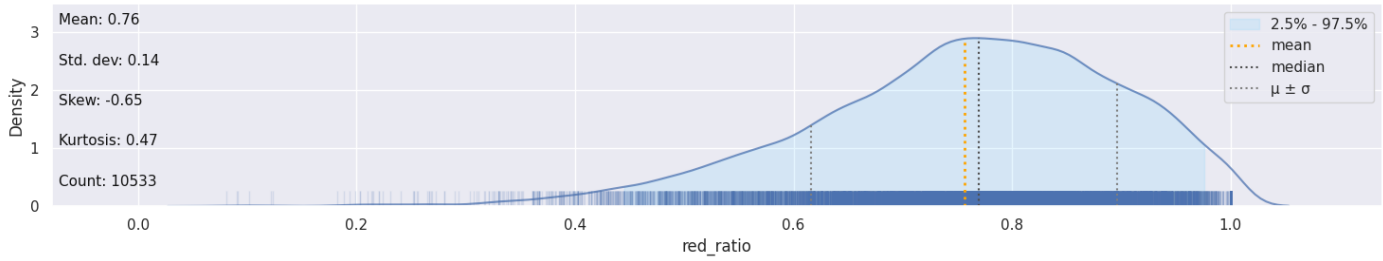


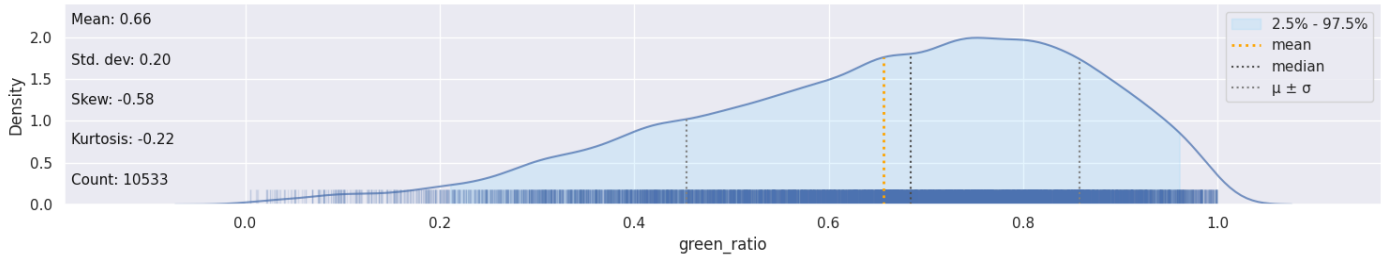


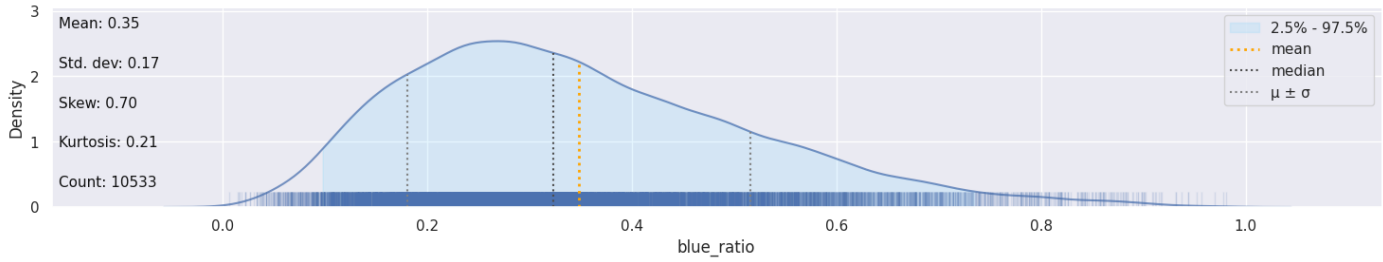


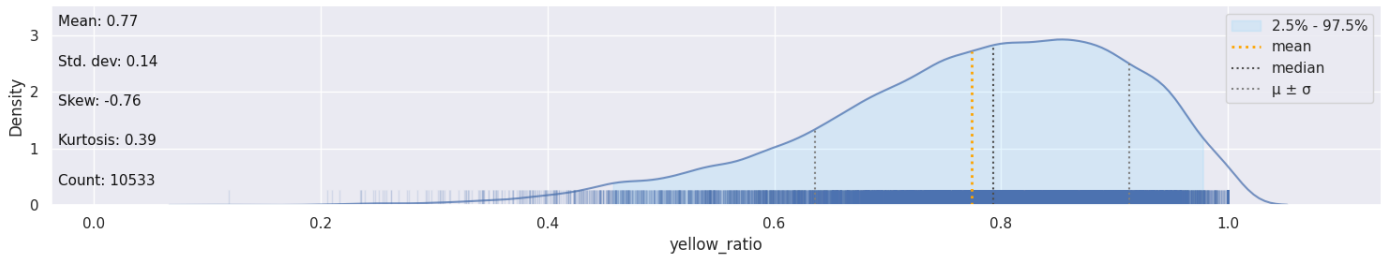

bbox_areas
1e6
mask_areas
Mean: 95710.78
Std. dev: 80300.09
Skew: 2.26
Kurtosis: 8.11
Count: 10533
mask_perimeters
Mean: 1055.76
Std. dev: 552.01
Skew: 1.28
Kurtosis: 3.01
Count: 10533
compactnesses
Mean: 1041.03
Std. dev: 426.68
Skew: 1.48
Kurtosis: 3.92
Count: 10533
red_pts
Mean: 127126.02
Std. dev: 120952.31
Skew: 2.59
Kurtosis: 11.34
Count: 10533
green_pts
Mean: 105449.58
Std. dev: 101940.53
Skew: 3.19
Kurtosis: 19.94
Count: 10533
blue_pts
Mean: 45684.41
Std. dev: 33212.61
Skew: 2.65
Kurtosis: 14.65
Count: 10533
Mean: 126476.35
Std. dev: 115533.83
Skew: 2.53
Kurtosis: 10.56
Density
2.5% - 97.5%
mean
median
μ ± σ

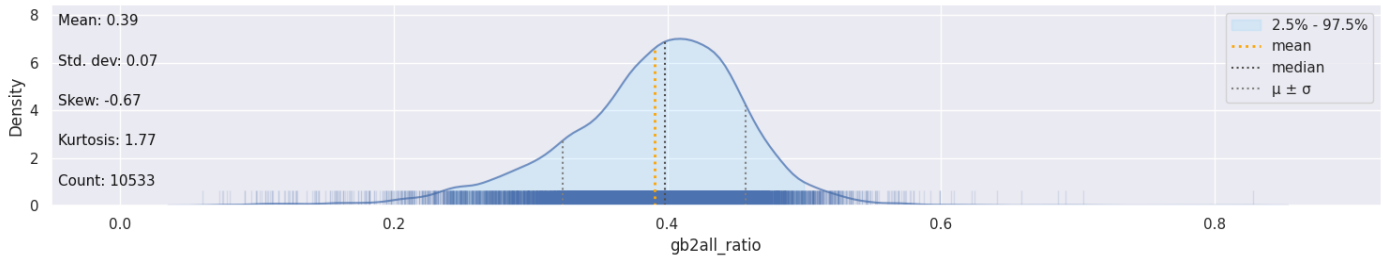

Count: 10533
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1e6
yellow_pts

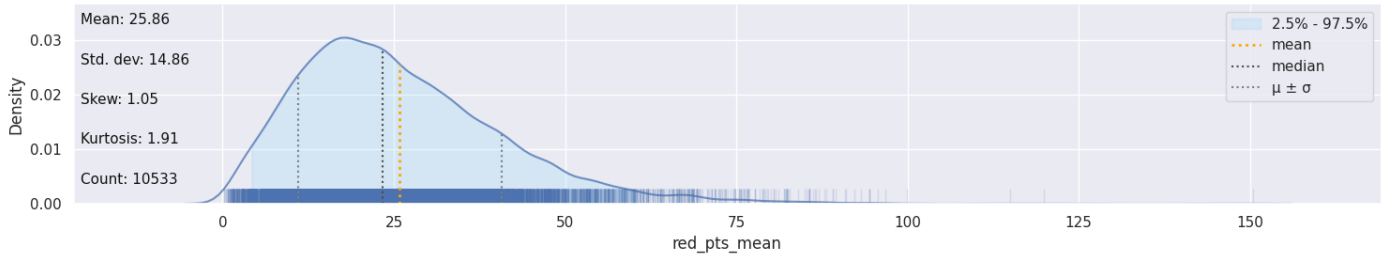

Mean: 0.76
Std. dev: 0.14
Skew: -0.65
Kurtosis: 0.47
Count: 10533
Density
2.5% - 97.5%
mean
median
μ ± σ
red_ratio


Mean: 0.66
Std. dev: 0.20
Skew: -0.58
Kurtosis: -0.22
Count: 10533
Density
2.5% - 97.5%
mean
median
μ ± σ
green_ratio

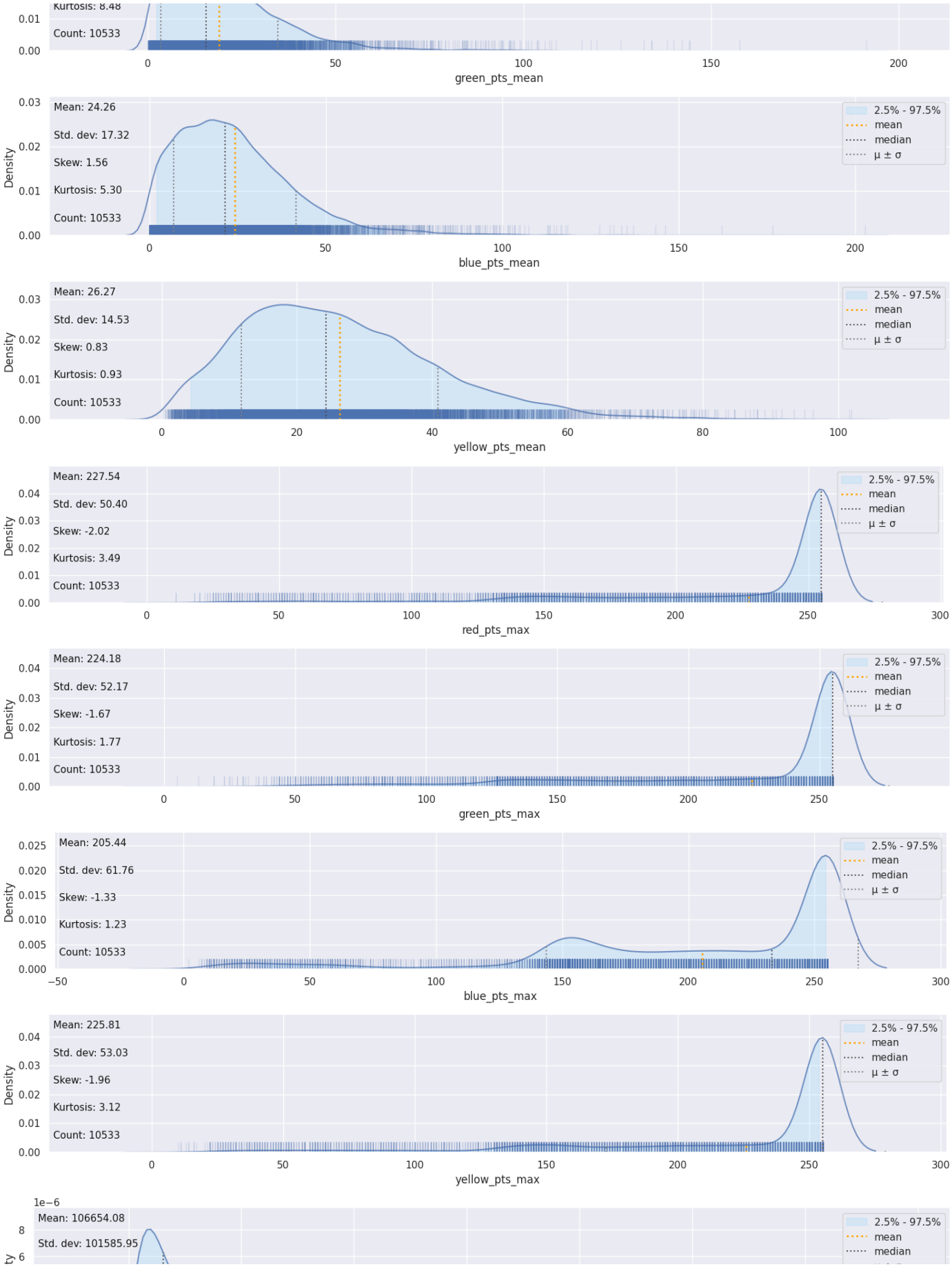

Mean: 0.35
Std. dev: 0.17
Skew: 0.70
Kurtosis: 0.21
Count: 10533
Density
2.5% - 97.5%
mean
median
μ ± σ
blue_ratio

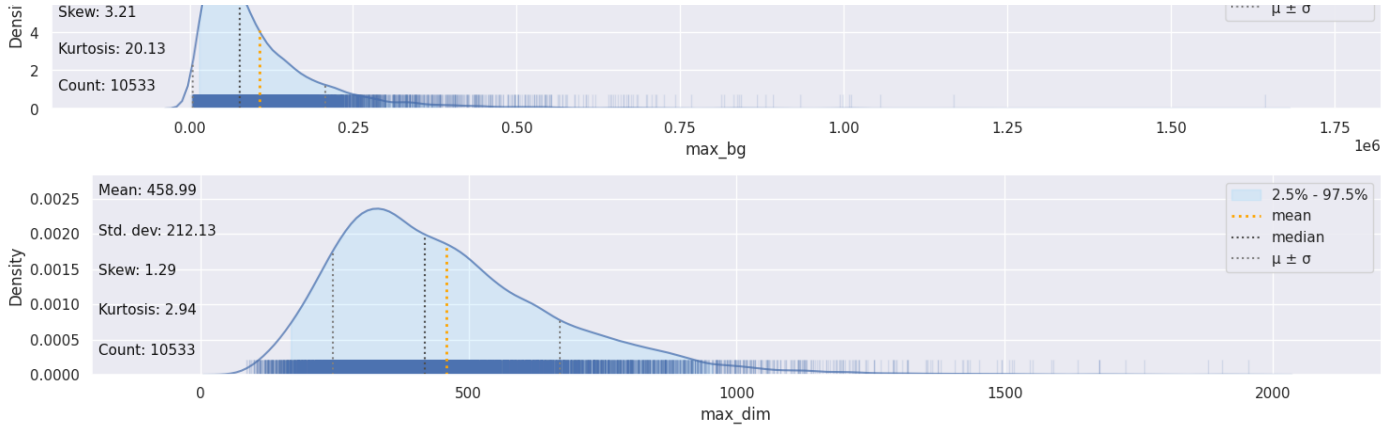

Mean: 0.77
Std. dev: 0.14
Skew: -0.76
Kurtosis: 0.39
Count: 10533
Density
2.5% - 97.5%
mean
median
μ ± σ
yellow_ratio


Mean: 0.39
Std. dev: 0.07
Skew: -0.67
Kurtosis: 1.77
Count: 10533
Density
2.5% - 97.5%
mean
median
μ ± σ
gb2all_ratio


Mean: 25.86
Std. dev: 14.86
Skew: 1.05
Kurtosis: 1.91
Count: 10533
Density
2.5% - 97.5%
mean
median
μ ± σ
red_pts_mean

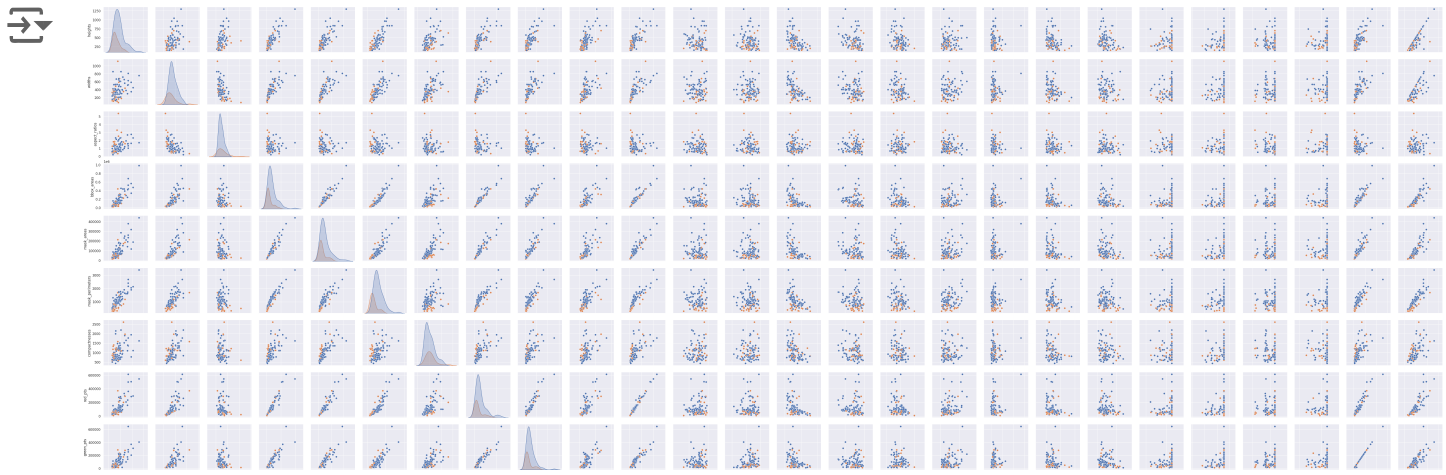

Mean: 19.01
Std. dev: 15.61
Skew: 2.08
Density
2.5% - 97.5%
mean
median
μ ± σ

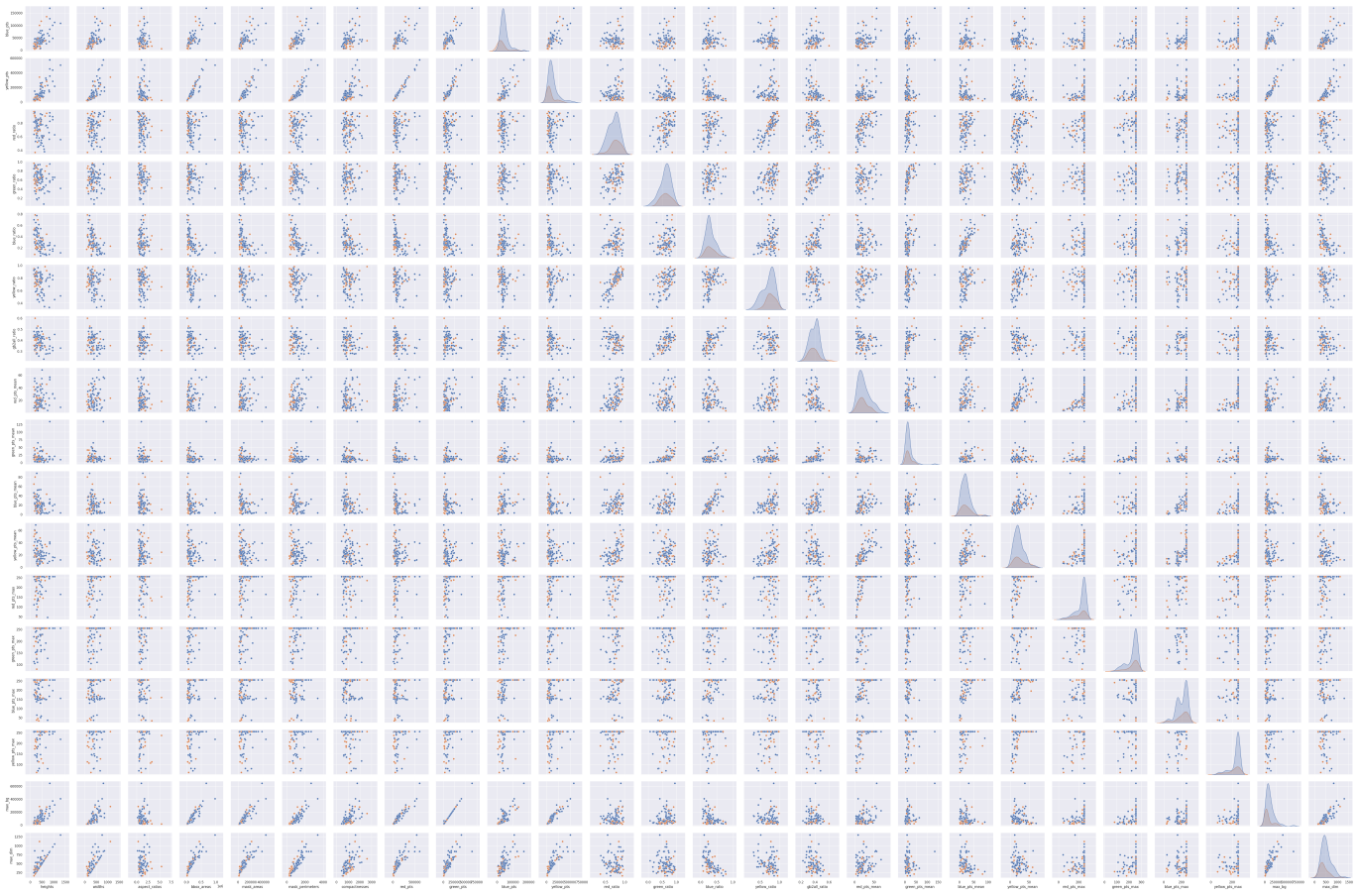

Kurtosis: 8.48
Count: 10533
green_pts_mean
Mean: 24.26
Std. dev: 17.32
Skew: 1.56
Kurtosis: 5.30
Count: 10533
blue_pts_mean
Mean: 26.27
Std. dev: 14.53
Skew: 0.83
Kurtosis: 0.93
Count: 10533
yellow_pts_mean
Mean: 227.54
Std. dev: 50.40
Skew: -2.02
Kurtosis: 3.49
Count: 10533
red_pts_max
Mean: 224.18
Std. dev: 52.17
Skew: -1.67
Kurtosis: 1.77
Count: 10533
green_pts_max
Mean: 205.44
Std. dev: 61.76
Skew: -1.33
Kurtosis: 1.23
Count: 10533
blue_pts_max
Mean: 225.81
Std. dev: 53.03
Skew: -1.96
Kurtosis: 3.12
Count: 10533
yellow_pts_max
Mean: 106654.08
Std. dev: 101585.95
2.5% - 97.5%
mean
median
μ ± σ
Density


Skew: 3.21
Kurtosis: 20.13
Count: 10533
μ ± σ
max_bg
1e6

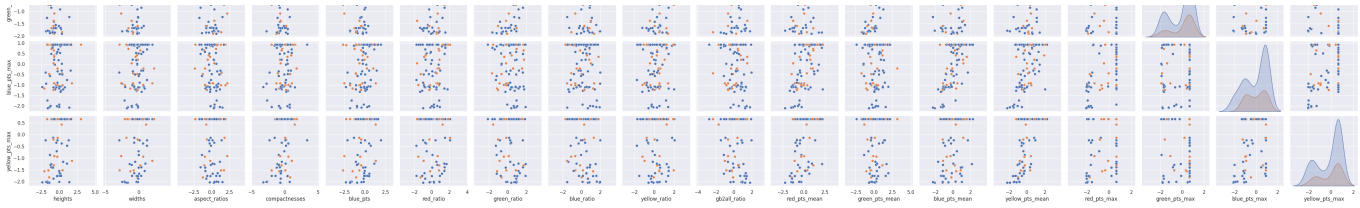
Mean: 458.99
Std. dev: 212.13
Skew: 1.29
Kurtosis: 2.94
Count: 10533
2.5% - 97.5%
mean
median
μ ± σ
Density
max_dim

```
# Pairplot with respect to isBad
sampled_data = cell_status.sample(n= 100)
sns.pairplot(sampled_data, hue='isBad')
plt.show()
```



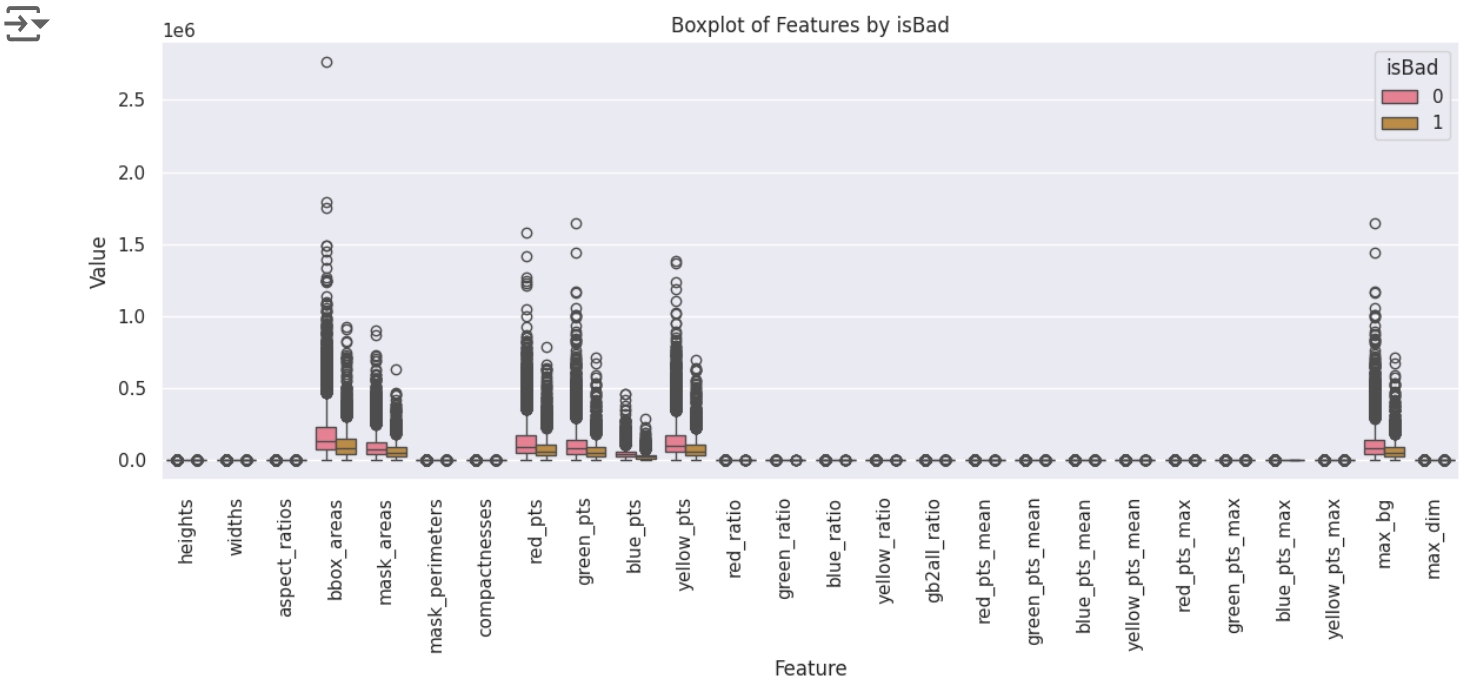


```
#### SHOULD BE RUN AFTER DATA WRANGLING #####

# Pairplot with respect to isBad
processed_df['isBad'] = data_reduced['isBad']
sampled_data = processed_df.sample(n= 100)
sns.pairplot(sampled_data, hue='isBad')
plt.show()
```



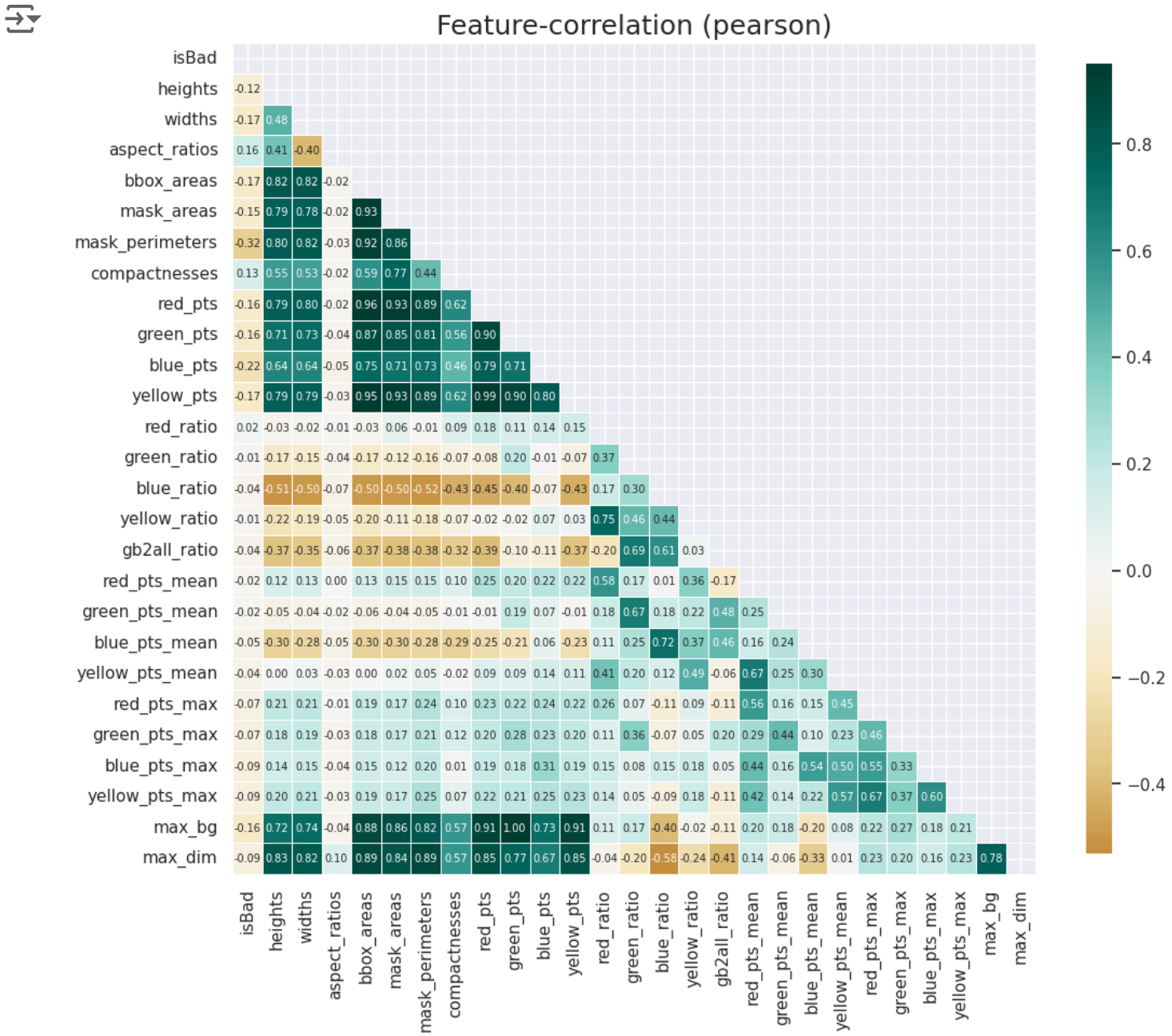

```
# Melt the dataframe to long format for easier plotting
df_melted = pd.melt(cell_status, id_vars=['isBad'],
                    value_vars=cell_status.columns[2:],
                    var_name='feature', value_name='value')

custom_palette = sns.husl_palette(8)
plt.figure(figsize=(12, 6))
sns.boxplot(x='feature', y='value', hue='isBad', data=df_melted,
```

```
            palette=custom_palette)
plt.xticks(rotation=90)
plt.xlabel('Feature')
plt.ylabel('Value')
plt.title('Boxplot of Features by isBad')
plt.legend(title='isBad', loc='upper right')
plt.tight_layout()
plt.show()
```




```
# Feature Correlation (Pearson)
ax = klib.corr_plot(cell_status)
for text in ax.texts:
```

```
    text.set_fontsize(7)
plt.show()
```



Pre-Processing/Data Wrangling

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

# Load the data
df = pd.read_csv('cell_stats_with_gt.csv')

# Inspect the data
print(df.info())
print(df.describe())
print(df.head())
```

```
            green_pts  ...  red_pts_mean  green_pts_mean  blue_pts_mean  \
count  1.053300e+04  ...  10533.000000    10533.000000   10533.000000
mean   1.054496e+05  ...     25.855552       19.010941      24.263599
std    1.019405e+05  ...     14.859445       15.610970      17.324462
min    1.170000e+02  ...      0.001032        0.015107       0.008642
25%    4.175700e+04  ...     15.059133        7.795207      11.679846
50%    7.541400e+04  ...     23.421683       15.490704      21.492674
75%    1.355580e+05  ...     34.147965       25.881093      32.894018
max    1.643200e+06  ...    150.405891      191.270706     202.777328

       yellow_pts_mean   red_pts_max  green_pts_max  blue_pts_max  \
count     10533.000000  10533.000000   10533.000000  10533.000000
mean         26.265069    227.541441     224.176873    205.436343
std          14.526722     50.403702      52.170433     61.762675
min           0.182212      2.000000       5.000000      2.000000
```

```
25%          15.444955    220.000000     208.000000    164.000000
50%          24.291322    255.000000     255.000000    233.000000
75%          34.803906    255.000000     255.000000    255.000000
max         101.931111    255.000000     255.000000    255.000000

       yellow_pts_max        max_bg       max_dim
count    10533.000000  1.053300e+04  10533.000000
mean       225.813728  1.066541e+05    458.990506
std         53.027258  1.015860e+05    212.125818
min         10.000000  1.431000e+03     86.000000
25%        216.000000  4.344800e+04    307.000000
50%        255.000000  7.632500e+04    418.000000
75%        255.000000  1.361720e+05    567.000000
max        255.000000  1.643200e+06   1955.000000

[8 rows x 27 columns]
                                                   Id  isBad  heights  widths
0  40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_1374...      0      579     471
1  40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_82_1...      0      759     491
2  40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_1424...      0      623     495
3  40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_850_...      0      587     439
4  40b819ec-bbb5-11e8-b2ba-ac1f6b6435d0_bbox_290_...      0      791     507

   aspect_ratios  bbox_areas  mask_areas  mask_perimeters  compactnesses  \
0       1.229299      272709      155022             1445    1348.141111
1       1.545825      372669      263655             1773    1868.689478
2       1.258586      308385      142551             1564    1145.363617
3       1.337130      257693      167007             1380    1520.776708
4       1.560158      401037      237002             1764    1688.353157

   red_pts  ...  red_pts_mean  green_pts_mean  blue_pts_mean  yellow_pts_mean
0   200903  ...     26.823581       20.574858      16.353864        19.954296
1   272478  ...     45.072501       25.781699       9.198616        36.508542
2   231746  ...     40.725113       30.762884      16.431908        41.164061
3   223590  ...     33.357604       28.960468      14.825696        29.261559
4   307845  ...     32.798607       27.674748      19.567130        22.804572

   red_pts_max  green_pts_max  blue_pts_max  yellow_pts_max  max_bg  max_dim
0          255            255           255             255  189719      579
1          255            255           255             255  271587      759
2          255            255           255             255  222357      623
3          255            255           255             255  211428      587
4          255            255           255             255  296375      791

[5 rows x 28 columns]
```

Feature Selection and Transformation

```
# Drop unnecessary predictors
pre_df = df.drop(columns=['Id'])


# Convert relevant columns to numeric type before calculating correlation
numeric_columns = pre_df.select_dtypes(include=['number']).columns
pre_df[numeric_columns] = pre_df[numeric_columns].apply(pd.to_numeric, errors='co

# Calculate the correlation matrix
correlation_matrix = pre_df[numeric_columns].corr()

# Identify features with high correlation (threshold > 0.8)
high_corr_features = set()
correlation_threshold = 0.8

for i in range(len(correlation_matrix.columns)):
    for j in range(i):
        if abs(correlation_matrix.iloc[i, j]) > correlation_threshold:
            colname = correlation_matrix.columns[i]
            high_corr_features.add(colname)

# Assuming 'cell_status' is the original DataFrame you want to reduce
data_reduced = pre_df.drop(columns=high_corr_features)

# Display the remaining features
remaining_features = data_reduced.columns
print(remaining_features)
```

```
Index(['isBad', 'heights', 'widths', 'aspect_ratios', 'compactnesses',
       'blue_pts', 'red_ratio', 'green_ratio', 'blue_ratio', 'yellow_ratio',
       'gb2all_ratio', 'red_pts_mean', 'green_pts_mean', 'blue_pts_mean',
       'yellow_pts_mean', 'red_pts_max', 'green_pts_max', 'blue_pts_max',
       'yellow_pts_max'],
      dtype='object')
```

```
# Import the required library
from scipy.stats import skew

# Identify numerical columns
numerical_cols = data_reduced.select_dtypes(include=['int64', 'float64']).columns

# Calculate skewness for each numerical feature
skewness_scores = data_reduced[numerical_cols].apply(skew)

# Display the skewness scores
print("Skewness scores for numerical features:")
print(skewness_scores)
```

```
Skewness scores for numerical features:
isBad              1.492307
heights            1.408004
widths             1.379608
aspect_ratios      5.153433
compactnesses      1.478885
blue_pts           2.653632
red_ratio         -0.648513
green_ratio       -0.580910
blue_ratio         0.699210
yellow_ratio      -0.757554
gb2all_ratio      -0.672435
red_pts_mean       1.045377
green_pts_mean     2.076013
blue_pts_mean      1.557051
yellow_pts_mean    0.831373
red_pts_max       -2.020581
green_pts_max     -1.670824
blue_pts_max      -1.330121
yellow_pts_max    -1.958703
dtype: float64
```

Box-Cox Transformation

```
# Import the required library
from scipy.stats import skew, boxcox # Import the boxcox function

# Identify numerical columns
reduced_df1 = data_reduced.drop('isBad', axis=1)
outcome = data_reduced['isBad']
numerical_cols = reduced_df1.select_dtypes(include=['int64', 'float64']).columns
```

```
# Apply Box-Cox transformation
# Note: Box-Cox requires input data to be positive
boxcox_transformed_data = reduced_df1[numerical_cols].apply(lambda x: boxcox(x +

# Display the transformed data
print("Box-Cox transformed data:")
print(boxcox_transformed_data.head())

# If you prefer using sklearn's PowerTransformer, which includes Box-Cox as one o
# Note: The PowerTransformer automatically shifts data to be positive if it conta

reduced_df2 = data_reduced.drop('isBad', axis=1)
outcome = data_reduced['isBad']
numerical_cols = reduced_df2.select_dtypes(include=['int64', 'float64']).columns
power_transformer = PowerTransformer(method='box-cox', standardize=False)
transformed_data = power_transformer.fit_transform(reduced_df2[numerical_cols] +

# Convert the transformed data back to a DataFrame
transformed_df = pd.DataFrame(transformed_data, columns=numerical_cols)

# Display the transformed data using sklearn's PowerTransformer
print("Box-Cox transformed data using sklearn's PowerTransformer:")
print(transformed_df.head())
```

```
Box-Cox transformed data:
    heights     widths  aspect_ratios  compactnesses   blue_pts  red_ratio  \
0  9.029954  10.159210       0.536156       5.239627  39.103746   1.865320
1  9.562225  10.264724       0.588137       5.403161  42.858389   1.839761
2  9.172474  10.285389       0.541603       5.156109  43.899579   1.934658
3  9.056572   9.982213       0.555511       5.300556  40.742349   2.538578
4  9.644771  10.346545       0.590175       5.352864  43.334840   2.012207

   green_ratio  blue_ratio  yellow_ratio  gb2all_ratio  red_pts_mean  \
0     1.252570    0.162490      1.766688      0.721851      6.860456
1     1.344787    0.175641      2.237284      0.726846      8.922776
2     1.322947    0.220104      2.028751      0.782146      8.484568
3     1.618531    0.196003      2.738361      0.719264      7.673592
4     1.374082    0.172403      2.136342      0.720417      7.607912

   green_pts_mean  blue_pts_mean  yellow_pts_mean   red_pts_max  \
0        4.308351       5.178745         6.596325  1.661281e+11
1        4.728804       3.743726         9.272539  1.661281e+11
2        5.074237       5.192145         9.901652  1.661281e+11
3        4.954544       4.908498         8.202041  1.661281e+11
4        4.865634       5.702485         7.123248  1.661281e+11
```

```
   green_pts_max  blue_pts_max  yellow_pts_max
0   1.920626e+10  72545.133719    1.571848e+10
1   1.920626e+10  72545.133719    1.571848e+10
2   1.920626e+10  72545.133719    1.571848e+10
3   1.920626e+10  72545.133719    1.571848e+10
4   1.920626e+10  72545.133719    1.571848e+10
Box-Cox transformed data using sklearn's PowerTransformer:
    heights     widths  aspect_ratios  compactnesses   blue_pts  red_ratio  \
0  9.029954  10.159210       0.536156       5.239627  39.103746   1.865320
1  9.562225  10.264724       0.588137       5.403161  42.858389   1.839761
2  9.172474  10.285389       0.541603       5.156109  43.899579   1.934658
3  9.056572   9.982213       0.555511       5.300556  40.742349   2.538578
4  9.644771  10.346545       0.590175       5.352864  43.334840   2.012207

   green_ratio  blue_ratio  yellow_ratio  gb2all_ratio  red_pts_mean  \
0     1.252570    0.162490      1.766688      0.721851      6.860456
1     1.344787    0.175641      2.237284      0.726846      8.922776
2     1.322947    0.220104      2.028751      0.782146      8.484568
3     1.618531    0.196003      2.738361      0.719264      7.673592
4     1.374082    0.172403      2.136342      0.720417      7.607912

   green_pts_mean  blue_pts_mean  yellow_pts_mean   red_pts_max  \
0        4.308351       5.178745         6.596325  1.661281e+11
1        4.728804       3.743726         9.272539  1.661281e+11
2        5.074237       5.192145         9.901652  1.661281e+11
3        4.954544       4.908498         8.202041  1.661281e+11
4        4.865634       5.702485         7.123248  1.661281e+11

   green_pts_max  blue_pts_max  yellow_pts_max
0   1.920626e+10  72545.133719    1.571848e+10
1   1.920626e+10  72545.133719    1.571848e+10
2   1.920626e+10  72545.133719    1.571848e+10
3   1.920626e+10  72545.133719    1.571848e+10
4   1.920626e+10  72545.133719    1.571848e+10
```

```
# Calculate the new skewness scores for the transformed features
new_skewness_scores = transformed_df.apply(skew)

# Display the new skewness scores
print("New skewness scores using PowerTransform:")
print(new_skewness_scores)
```

```
New skewness scores using PowerTransform:
heights            0.001325
widths             0.003030
aspect_ratios     -0.008009
compactnesses     -0.001961
blue_pts           0.033165
red_ratio         -0.054577
green_ratio       -0.105897
blue_ratio         0.031022
yellow_ratio      -0.089167
gb2all_ratio       0.107105
red_pts_mean      -0.005266
green_pts_mean    -0.015376
blue_pts_mean     -0.013497
yellow_pts_mean   -0.018703
red_pts_max       -1.098482
green_pts_max     -0.976335
blue_pts_max      -0.602641
yellow_pts_max    -1.090265
dtype: float64
```

Center and Scaling

```
# Standardize the numeric features
scaler = StandardScaler()
scaled_features = scaler.fit_transform(transformed_df)

# Convert processed features back to DataFrame
processed_df = pd.DataFrame(scaled_features, columns=transformed_df.columns,
                            index=transformed_df.index)

# Check the processed data
print(processed_df.info())
print(processed_df.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10533 entries, 0 to 10532
```

```
Data columns (total 18 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   heights          10533 non-null  float64
 1   widths           10533 non-null  float64
 2   aspect_ratios    10533 non-null  float64
 3   compactnesses    10533 non-null  float64
 4   blue_pts         10533 non-null  float64
 5   red_ratio        10533 non-null  float64
 6   green_ratio      10533 non-null  float64
 7   blue_ratio       10533 non-null  float64
 8   yellow_ratio     10533 non-null  float64
 9   gb2all_ratio     10533 non-null  float64
 10  red_pts_mean     10533 non-null  float64
 11  green_pts_mean   10533 non-null  float64
 12  blue_pts_mean    10533 non-null  float64
 13  yellow_pts_mean  10533 non-null  float64
 14  red_pts_max      10533 non-null  float64
 15  green_pts_max    10533 non-null  float64
 16  blue_pts_max     10533 non-null  float64
 17  yellow_pts_max   10533 non-null  float64
dtypes: float64(18)
memory usage: 1.4 MB
None
    heights    widths  aspect_ratios  compactnesses  blue_pts  red_ratio  \
0  1.055145  0.611215       0.454327       0.873611  0.553686  -0.254847   
1  1.645206  0.699445       0.938747       1.686230  1.222756  -0.294703   
2  1.213138  0.716724       0.505088       0.458601  1.408294  -0.146725   
3  1.084652  0.463212       0.634696       1.176375  0.845682   0.794998   
4  1.736714  0.767862       0.957739       1.436296  1.307658  -0.025799   

   green_ratio  blue_ratio  yellow_ratio  gb2all_ratio  red_pts_mean  \
0     0.091059   -0.886676     -1.185183      0.057409      0.234109   
1     0.274116   -0.721876     -0.730607      0.085091      1.247098   
2     0.230762   -0.164699     -0.932042      0.391593      1.031855   
3     0.817514   -0.466723     -0.246588      0.043069      0.633512   
4     0.332268   -0.762456     -0.828113      0.049458      0.601251   

   green_pts_mean  blue_pts_mean  yellow_pts_mean  red_pts_max  green_pts_max  \
0        0.394907      -0.291706        -0.307353     0.653642       0.686076   
1        0.691753      -0.894196         0.781548     0.653642       0.686076   
2        0.935633      -0.286080         1.037523     0.653642       0.686076   
3        0.851128      -0.405169         0.345982     0.653642       0.686076   
4        0.788357      -0.071814        -0.092958     0.653642       0.686076   

   blue_pts_max  yellow_pts_max  
0      0.923699        0.653209  
1      0.923699        0.653209  
2      0.923699        0.653209  
3      0.923699        0.653209  
```

```
4       0.923699        0.653209
```

Data Splitting: 20% Test Set and 80% Train Set

```
# Split the data into features and target
X = processed_df
y = data_reduced['isBad']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

# Check the class distribution in the training set
print("Training set class distribution before balancing:")
print(y_train.value_counts())

# Create resampling pipelines
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# Check the class distribution in the balanced training set
print("Training set class distribution after balancing:")
print(pd.Series(y_train_resampled).value_counts())

# Display the shapes of the training and testing sets
print(f"Shape of X_train_resampled: {X_train_resampled.shape}")
print(f"Shape of y_train_resampled: {y_train_resampled.shape}")
print(f"Shape of X_test: {X_test.shape}")
print(f"Shape of y_test: {y_test.shape}")
```

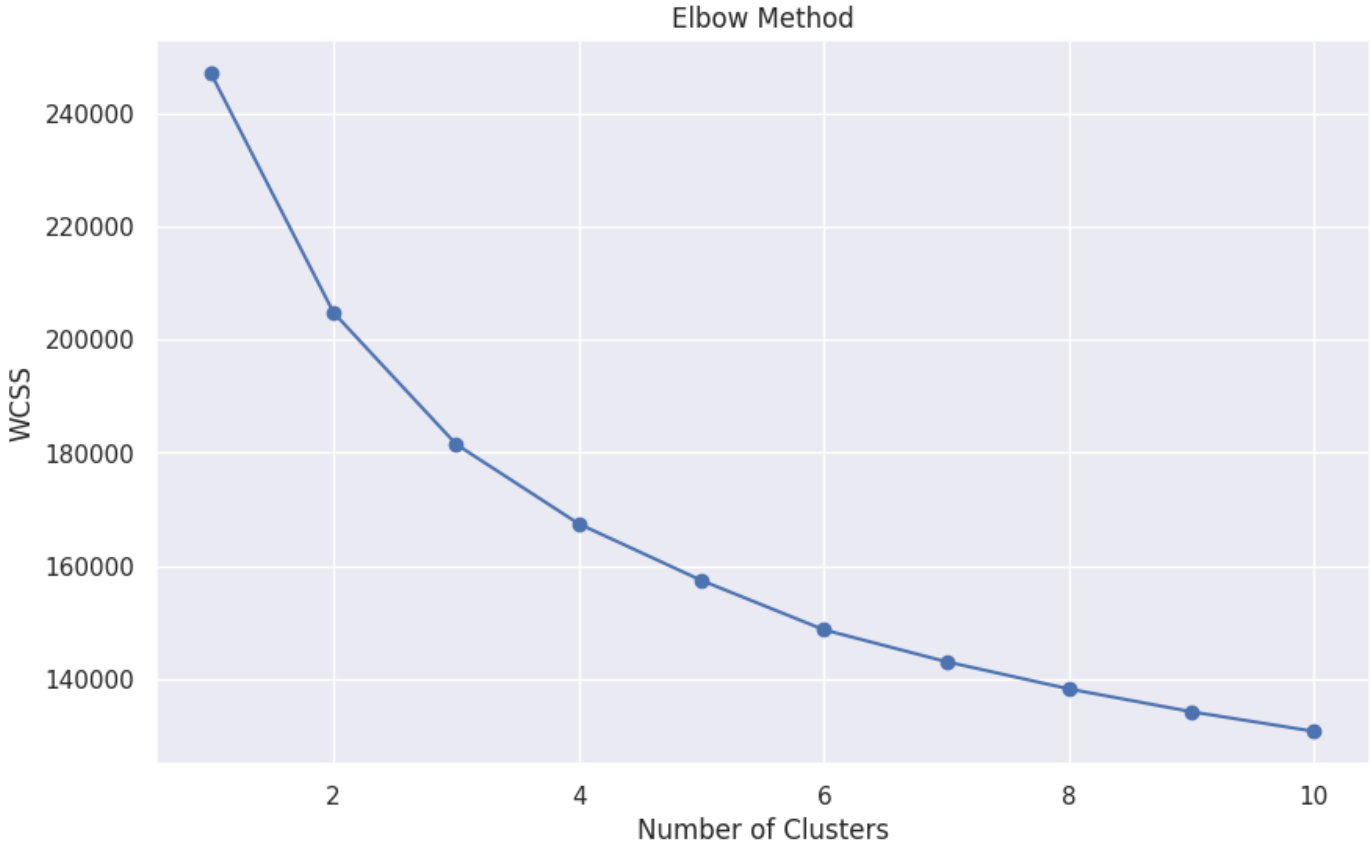
```
Training set class distribution before balancing:
isBad
0    6718
1    1708
Name: count, dtype: int64
Training set class distribution after balancing:
isBad
0    6718
1    6718
Name: count, dtype: int64
Shape of X_train_resampled: (13436, 18)
Shape of y_train_resampled: (13436,)
Shape of X_test: (2107, 18)
Shape of y_test: (2107,)
```

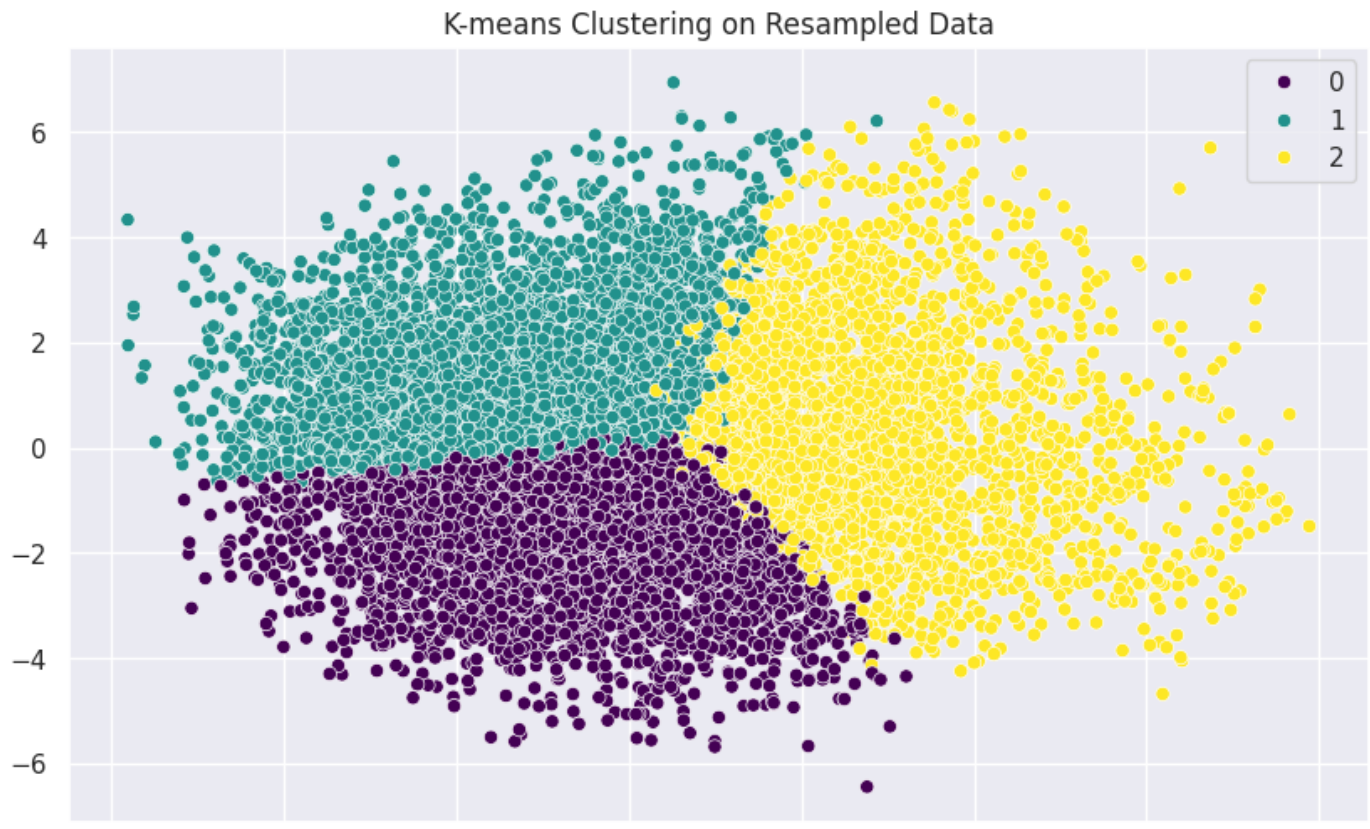
Model Strategies

K-Means

```
# Import necessary libraries for clustering and visualization
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Determine the optimal number of clusters using the Elbow method
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)
    kmeans.fit(X_train_resampled)  # Use X_train_resampled for fitting the model
    wcss.append(kmeans.inertia_)

# Plot the Elbow graph
plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), wcss, marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.show()

# Fit the K-means model with the optimal number of clusters (let's assume K=3 for
optimal_k = 3
kmeans = KMeans(n_clusters=optimal_k, init='k-means++', random_state=42)
cluster_labels = kmeans.fit_predict(X_train_resampled)  # Use X_train_resampled f

# Create a new DataFrame from the resampled data
resampled_df = pd.DataFrame(X_train_resampled, columns=df.columns)  # Use the sam
resampled_df['Cluster'] = cluster_labels  # Add cluster labels to this new DataFr

# Visualize the clusters using PCA (on resampled data)
pca = PCA(n_components=2)
pca_features = pca.fit_transform(X_train_resampled)  # Use X_train_resampled for

plt.figure(figsize=(10, 6))
sns.scatterplot(x=pca_features[:, 0], y=pca_features[:, 1], hue=cluster_labels, p
plt.title('K-means Clustering on Resampled Data')
plt.show()

# Display the first few rows of the new resampled dataframe with cluster labels
```

```
resampled_df.head()
```





−6 −4 −2 0 2 4 6 8

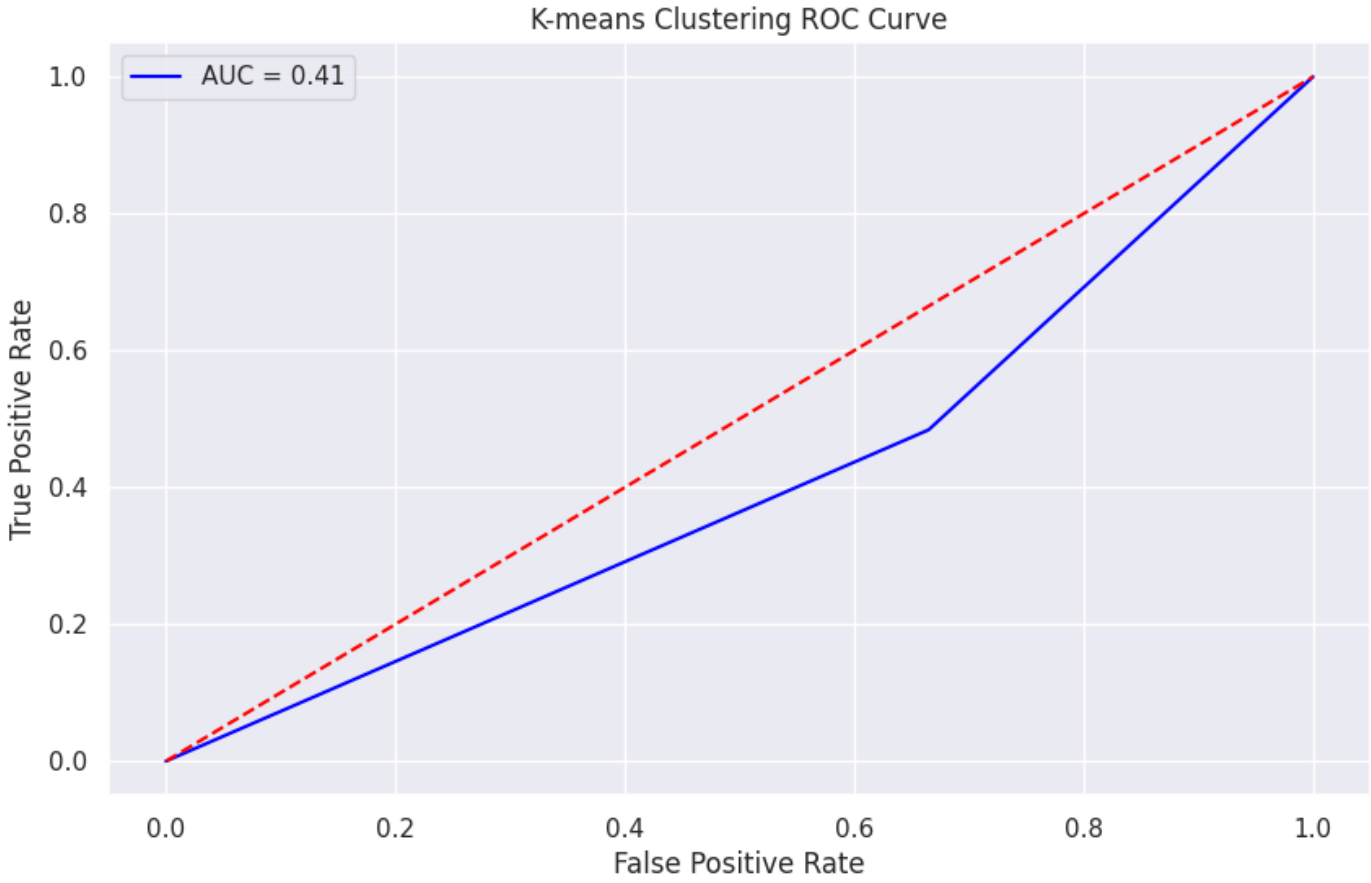
	Id	isBad	heights	widths	aspect_ratios	bbox_areas	mask_areas	mask_p
0	NaN	NaN	2.123474	1.110710	0.986894	NaN	NaN	
1	NaN	NaN	-0.632257	-1.352069	0.840110	NaN	NaN	
2	NaN	NaN	-0.586412	-0.182546	-0.445818	NaN	NaN	
3	NaN	NaN	0.076344	-0.201307	0.300363	NaN	NaN	
4	NaN	NaN	0.109894	0.565904	-0.487092	NaN	NaN	

5 rows × 29 columns

```
# Import necessary libraries
from sklearn.metrics import roc_auc_score, roc_curve
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans

# For binary classification, we'll use 2 clusters
n_clusters = 2

# Fit K-means clustering with 2 clusters on the training data
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
kmeans_labels_train = kmeans.fit_predict(X_train_resampled)  # Use X_train_resamp

# Predict clusters for the test data
kmeans_labels_test = kmeans.predict(X_test)  # Use X_test for predicting
```

```
# Convert cluster labels to binary classification (assuming binary target variabl
# Here we assume cluster 1 is positive class and cluster 0 is negative class for
binary_labels_train = (kmeans_labels_train == 1).astype(int)
binary_labels_test = (kmeans_labels_test == 1).astype(int)

# Calculate AUC-ROC score for K-means
auc_score_kmeans = roc_auc_score(y_test, binary_labels_test)
print(f'K-means AUC-ROC Score: {auc_score_kmeans}')

# Plotting the ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, binary_labels_test)
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='blue', label=f'AUC = {auc_score_kmeans:.2f}')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('K-means Clustering ROC Curve')
plt.legend()
plt.show()
```

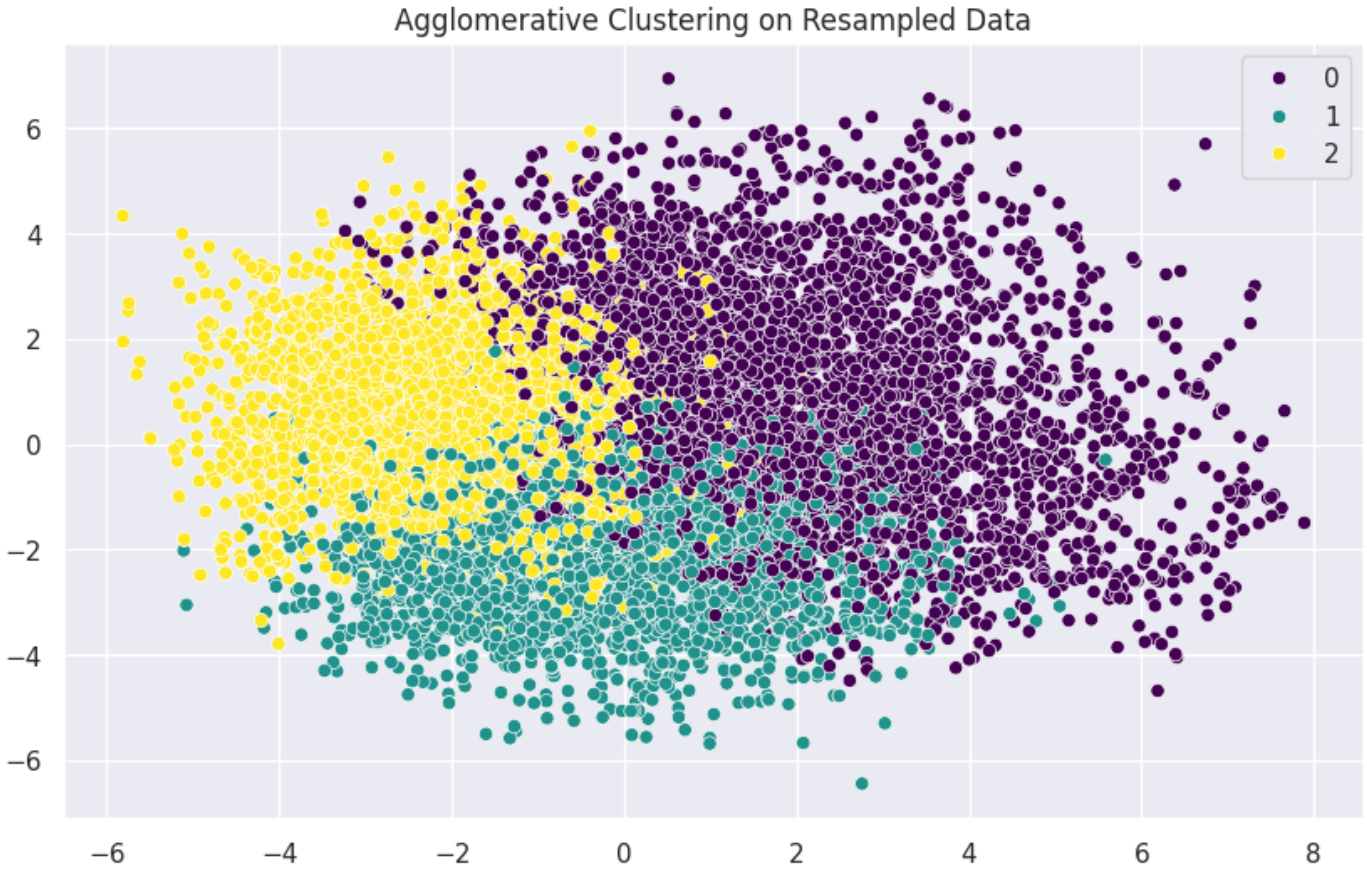
```
K-means AUC-ROC Score: 0.40960384862386645
```



Agglomerative Clustering

```
# Import necessary libraries
from sklearn.cluster import AgglomerativeClustering
from sklearn.decomposition import PCA
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Define the number of clusters and linkage criterion
n_clusters = 3  # Number of clusters, can be tuned
```

```
linkage = 'ward'  # Linkage criterion: 'ward', 'complete', 'average', 'single'

# Fit the Agglomerative Clustering model on resampled data
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage=linkage)
agg_labels = agg_clustering.fit_predict(X_train_resampled)

# Create a new DataFrame from the resampled data
resampled_df = pd.DataFrame(X_train_resampled, columns=df.columns)  # Use the sam
resampled_df['Agg_Cluster'] = agg_labels  # Add cluster labels to this new DataFr

# Visualize the Agglomerative Clustering clusters (using the first two principal
pca = PCA(n_components=2)
pca_features = pca.fit_transform(X_train_resampled)

plt.figure(figsize=(10, 6))
sns.scatterplot(x=pca_features[:, 0], y=pca_features[:, 1], hue=agg_labels, palet
plt.title('Agglomerative Clustering on Resampled Data')
plt.show()

# Display the first few rows of the new resampled dataframe with Agglomerative Cl
resampled_df.head()
```



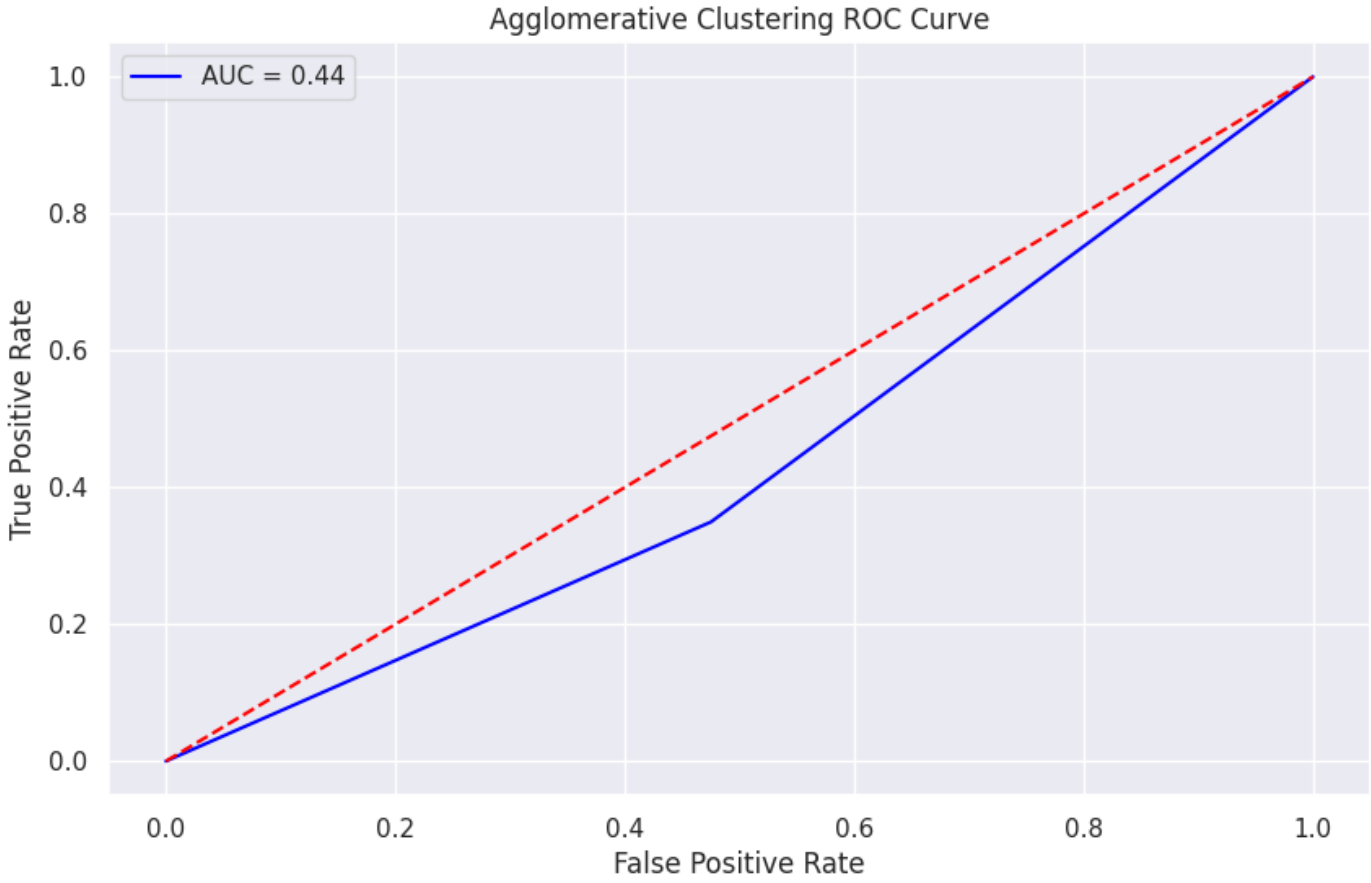
	Id	isBad	heights	widths	aspect_ratios	bbox_areas	mask_areas	mask_p
0	NaN	NaN	2.123474	1.110710	0.986894	NaN	NaN	
1	NaN	NaN	-0.632257	-1.352069	0.840110	NaN	NaN	
2	NaN	NaN	-0.586412	-0.182546	-0.445818	NaN	NaN	
3	NaN	NaN	0.076344	-0.201307	0.300363	NaN	NaN	
4	NaN	NaN	0.109894	0.565904	-0.487092	NaN	NaN	

5 rows × 29 columns

```
# Import necessary libraries
from sklearn.cluster import AgglomerativeClustering
```

```
from sklearn.metrics import roc_auc_score, roc_curve
import matplotlib.pyplot as plt
import numpy as np

# Define the number of clusters (for binary classification, use 2 clusters)
n_clusters = 2

# Fit Agglomerative Clustering with 2 clusters on the training data
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward')
agg_labels_train = agg_clustering.fit_predict(X_train_resampled)

# Predict clusters for the test data
agg_labels_test = agg_clustering.fit_predict(X_test)

# Convert cluster labels to binary classification (assuming binary target variabl
binary_labels_train = (agg_labels_train == 1).astype(int)
binary_labels_test = (agg_labels_test == 1).astype(int)

# Calculate AUC-ROC score for Agglomerative Clustering
auc_score_agg = roc_auc_score(y_test, binary_labels_test)
print(f'Agglomerative Clustering AUC-ROC Score: {auc_score_agg}')

# Plotting the ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, binary_labels_test)
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='blue', label=f'AUC = {auc_score_agg:.2f}')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Agglomerative Clustering ROC Curve')
plt.legend()
plt.show()
```

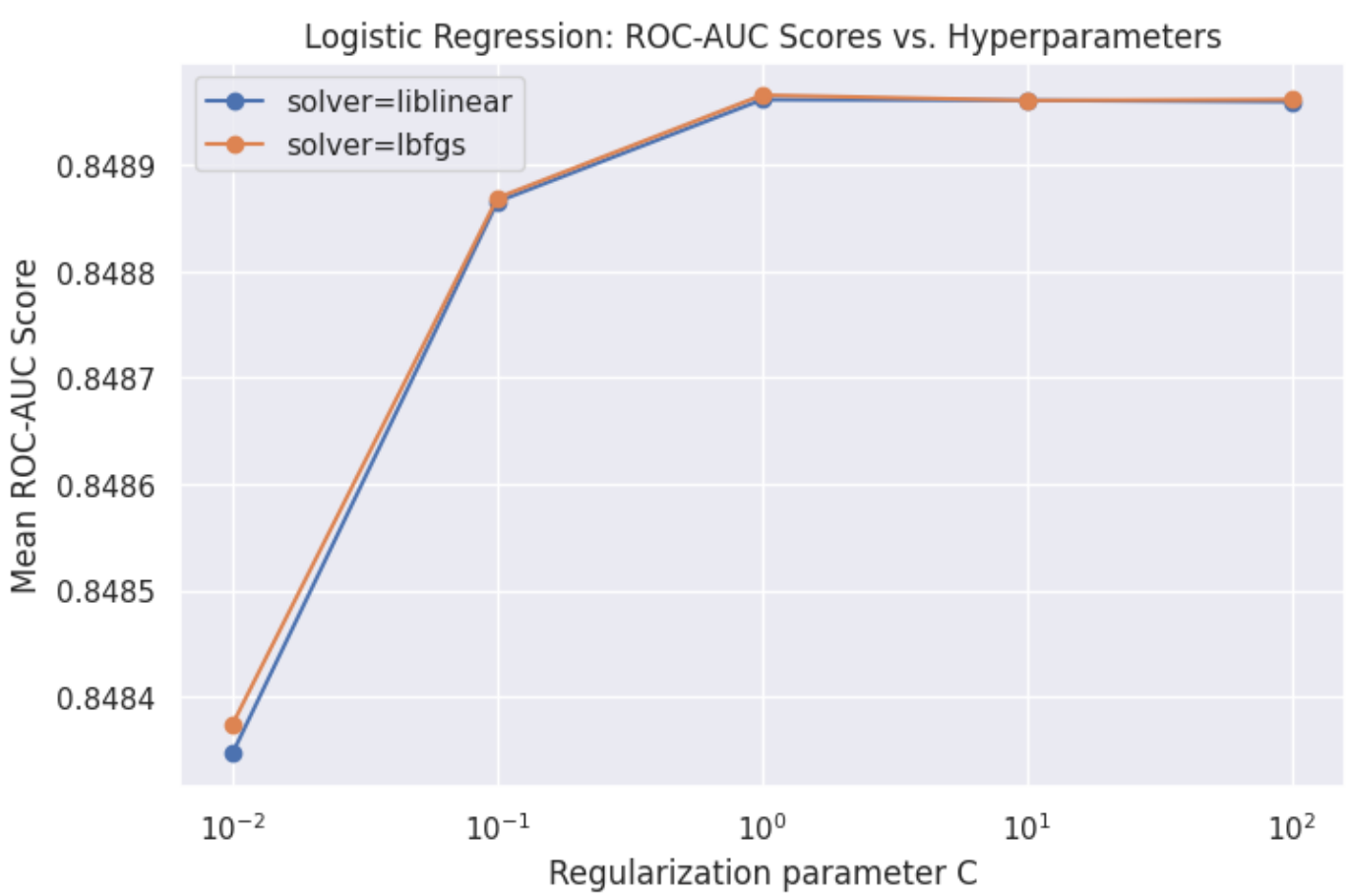
```
Agglomerative Clustering AUC-ROC Score: 0.43718411708294813
```



Logistic Regression

```
# Perform hyperparameter tuning of Logistic Regression
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'lbfgs']
}

lr = LogisticRegression()
grid_search = GridSearchCV(lr, param_grid, cv=5, scoring='roc_auc',
                           return_train_score=True)
```

```
grid_search.fit(X_train_resampled, y_train_resampled)

results = grid_search.cv_results_

# Get the mean ROC-AUC scores for each parameter combination
mean_test_scores = results['mean_test_score']
param_combinations = [f"C={c}, solver={s}" for c in param_grid['C'] for s in para
mean_test_scores_reshaped = np.array(mean_test_scores).reshape(len(param_grid['C'
                                                               len(param_grid['so

# Plot ROC-AUC scores
plt.figure(figsize=(8, 5))

for i, solver in enumerate(param_grid['solver']):
    plt.plot(param_grid['C'], mean_test_scores_reshaped[:, i], marker='o',
             label=f'solver={solver}')

plt.xscale('log')
plt.xlabel('Regularization parameter C')
plt.ylabel('Mean ROC-AUC Score')
plt.title('Logistic Regression: ROC-AUC Scores vs. Hyperparameters')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```




```
# Final Logistic Regression Model
lr_model = LogisticRegression(C=1.0, solver='lbfgs')
lr_model.fit(X_train_resampled, y_train_resampled)
```

```
▾ LogisticRegression
LogisticRegression()
```

Neural Networks

```
!pip install tensorflow
!pip install keras
!pip install tensorflow scikit-learn
!pip install keras-tuner
```

```
import keras_tuner as kt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import backend as K
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.10/
Requirement already satisfied: flatbuffers>=23.5.26 in /usr/local/lib/python3.
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/loc
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.1
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: ml-dtypes~=0.2.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in /usr/local/lib/python3.
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.10/
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/pyth
Requirement already satisfied: wrapt<1.15,>=1.11.0 in /usr/local/lib/python3.1
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/lo
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.1
Requirement already satisfied: tensorboard<2.16,>=2.15 in /usr/local/lib/pytho
Requirement already satisfied: tensorflow-estimator<2.16,>=2.15.0 in /usr/loca
Requirement already satisfied: keras<2.16,>=2.15.0 in /usr/local/lib/python3.1
Requirement already satisfied: scipy>=1.5.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.10
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3
Requirement already satisfied: google-auth-oauthlib<2,>=0.5 in /usr/local/lib/
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.10/di
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.1
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/l
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/pyth
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pyth
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.10/
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in /usr/local/lib/python3.
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.10/di
```

```
Collecting keras-tuner
  Downloading keras_tuner-1.4.7-py3-none-any.whl.metadata (5.4 kB)
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-pacl
Collecting kt-legacy (from keras-tuner)
  Downloading kt_legacy-1.0.5-py3-none-any.whl.metadata (221 bytes)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pytl
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10
Downloading keras_tuner-1.4.7-py3-none-any.whl (129 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 129.1/129.1 kB 1.4 MB/s eta 0:00:0
Downloading kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.4.7 kt-legacy-1.0.5
```

```
# Define model functions
def build_model(hp):
    model = Sequential()
    model.add(Dense(
        units=hp.Int('units1', min_value=32, max_value=128, step=32),
        activation='relu',
        input_shape=(X_train_resampled.shape[1],)
    ))
    model.add(Dense(
        units=hp.Int('units2', min_value=16, max_value=64, step=16),
        activation='relu'
    ))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(
        optimizer=Adam(hp.Float('learning_rate', min_value=1e-4, max_value=1e-1,
                                sampling='LOG')),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model


# Hyperparameter tuning parameters
def tuner_search():
    tuner = kt.Hyperband(
        build_model,
        objective='val_accuracy',
        max_epochs=20,
```

```
        hyperband_iterations=2,
        overwrite=True,
        directory='keras_tuner_dir',
        project_name='binary_classification'
    )

    # Perform the search
    tuner.search(X_train_resampled, y_train_resampled,
                 epochs=20,
                 batch_size=32,
                 validation_split=0.2)

    return tuner

# Run the search
tuner = tuner_search()

# Retrieve the results from the tuner
best_trials = tuner.oracle.get_best_trials()
results = []
hyperparams = []

for trial in best_trials:
    trial_id = trial.trial_id
    hyperparameters = trial.hyperparameters.values
    best_result = trial.metrics.get_best_value('val_accuracy')

    results.append({
        'trial_id': trial_id,
        'hyperparameters': hyperparameters,
        'best_val_accuracy': best_result
    })
    hyperparams.append(hyperparameters)

print("Best Trials and their Results:")
for result in results:
    print(result)
```

```
Trial 60 Complete [00h 00m 16s]
val_accuracy: 0.7868303656578064

Best val_accuracy So Far: 0.890997052192688
Total elapsed time: 00h 06m 55s
Best Trials and their Results:
{'trial_id': '0058', 'hyperparameters': {'units1': 96, 'units2': 32, 'learning
```

```
hyperparams
```

```
[{'units1': 96,
  'units2': 32,
  'learning_rate': 0.0025413858304798856,
  'tuner/epochs': 20,
  'tuner/initial_epoch': 0,
  'tuner/bracket': 0,
  'tuner/round': 0}]
```

```
results
```

```
[{'trial_id': '0058',
  'hyperparameters': {'units1': 96,
   'units2': 32,
   'learning_rate': 0.0025413858304798856,
   'tuner/epochs': 20,
   'tuner/initial_epoch': 0,
   'tuner/bracket': 0,
   'tuner/round': 0},
  'best_val_accuracy': 0.890997052192688}]
```

```
# Build the final model
best_hyperparameters = tuner.get_best_hyperparameters(num_trials=1)[0]
nn_model = build_model(best_hyperparameters)

# Train the final model on the entire training data
nn_model.fit(X_train_resampled, y_train_resampled,
               epochs=20,
               batch_size=32,
               validation_split=0.2)
```

```
Epoch 1/20
336/336 [==============================] - 2s 3ms/step - loss: 0.4437 - accura
Epoch 2/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3861 - accura
Epoch 3/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3717 - accura
Epoch 4/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3562 - accura
Epoch 5/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3474 - accura
Epoch 6/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3357 - accura
Epoch 7/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3254 - accura
Epoch 8/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3183 - accura
Epoch 9/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3085 - accura
Epoch 10/20
336/336 [==============================] - 1s 2ms/step - loss: 0.3015 - accura
Epoch 11/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2924 - accura
Epoch 12/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2869 - accura
Epoch 13/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2800 - accura
Epoch 14/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2710 - accura
Epoch 15/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2617 - accura
Epoch 16/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2591 - accura
Epoch 17/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2506 - accura
Epoch 18/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2430 - accura
Epoch 19/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2389 - accura
Epoch 20/20
336/336 [==============================] - 1s 2ms/step - loss: 0.2276 - accura
<keras.src.callbacks.History at 0x788895f6a200>
```

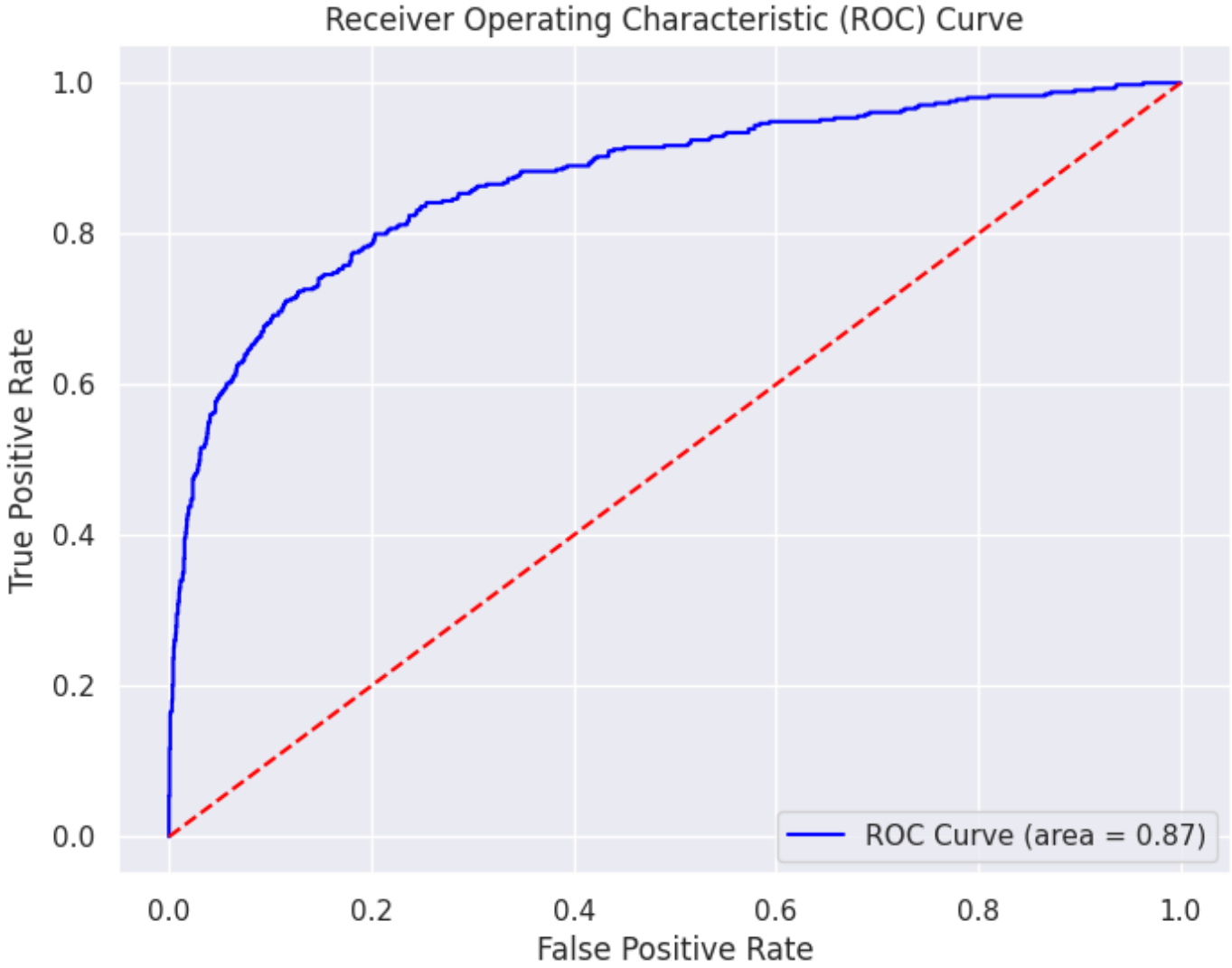
XG Boost

```
!pip install xgboost imbalanced-learn
from imblearn.over_sampling import SMOTE
from xgboost import XGBClassifier
```

```
Collecting xgboost
  Downloading xgboost-2.1.1-py3-none-manylinux_2_28_x86_64.whl.metadata (2.1 k
Requirement already satisfied: imbalanced-learn in /usr/local/lib/python3.10/d
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-package
Collecting nvidia-nccl-cu12 (from xgboost)
  Downloading nvidia_nccl_cu12-2.22.3-py3-none-manylinux2014_x86_64.whl.metada
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: scikit-learn>=1.0.2 in /usr/local/lib/python3.1
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Downloading xgboost-2.1.1-py3-none-manylinux_2_28_x86_64.whl (153.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 153.9/153.9 MB 6.8 MB/s eta 0:00:0
Downloading nvidia_nccl_cu12-2.22.3-py3-none-manylinux2014_x86_64.whl (190.9 M
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 190.9/190.9 MB 5.4 MB/s eta 0:00:0
Installing collected packages: nvidia-nccl-cu12, xgboost
Successfully installed nvidia-nccl-cu12-2.22.3 xgboost-2.1.1
```

```
# Initialize and train the XGBoost model
model = XGBClassifier(random_state=42)
model.fit(X_train_resampled, y_train_resampled)

# Predict probabilities on the test set
y_pred_proba = model.predict_proba(X_test)[:, 1]

# Calculate the ROC-AUC score
roc_auc_xgb = roc_auc_score(y_test, y_pred_proba)
print(f"ROC-AUC Score: {roc_auc_xgb:.2f}")

# Generate the ROC curve
fpr_xgb, tpr_xgb, thresholds = roc_curve(y_test, y_pred_proba)

# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr_xgb, tpr_xgb, color='blue', label=f'ROC Curve (area = {roc_auc_xgb:.
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

```
ROC-AUC Score: 0.87
```

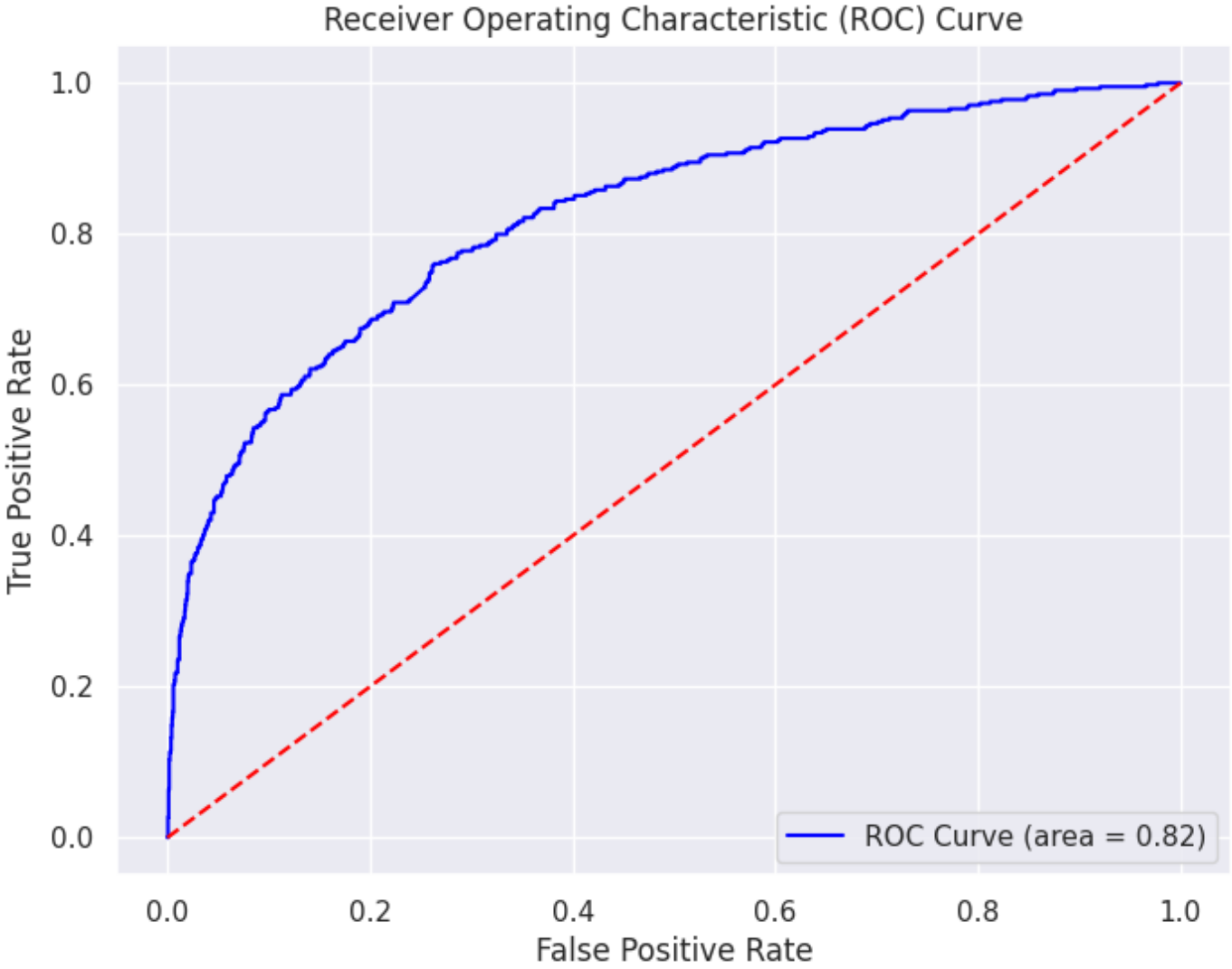


SVM

```
# Initialize and train the SVM model with probability estimates
model = SVC(kernel='linear', probability=True, random_state=42)
model.fit(X_train_resampled, y_train_resampled)

# Predict probabilities on the test set
y_pred_proba = model.predict_proba(X_test)[:, 1]

# Calculate the ROC-AUC score
roc_auc_svm = roc_auc_score(y_test, y_pred_proba)
```

```
print(f"ROC-AUC Score: {roc_auc_svm:.2f}")

# Generate the ROC curve
fpr_svm, tpr_svm, thresholds = roc_curve(y_test, y_pred_proba)

# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr_svm, tpr_svm, color='blue', label=f'ROC Curve (area = {roc_auc_svm:.
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

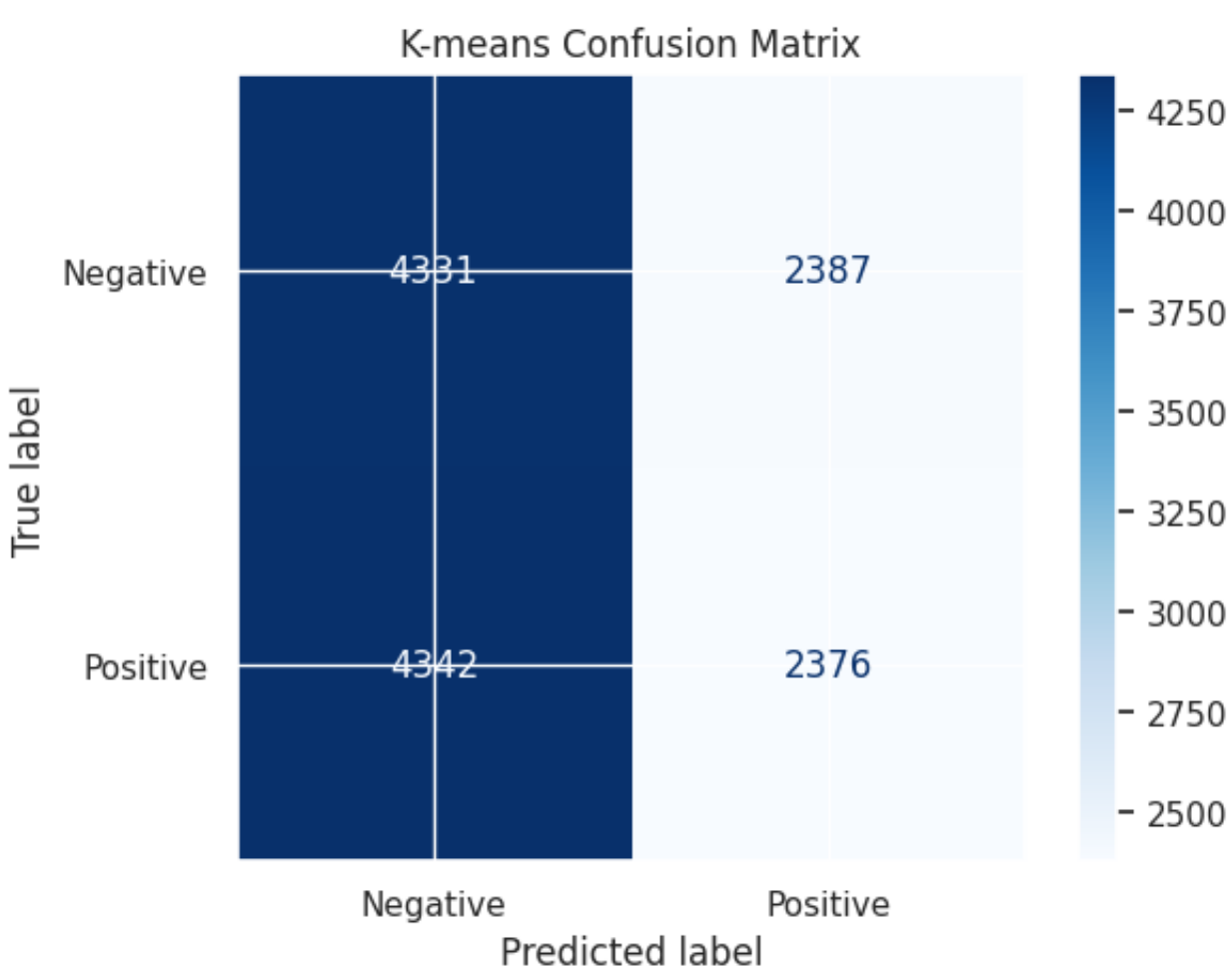
```
ROC-AUC Score: 0.82
```



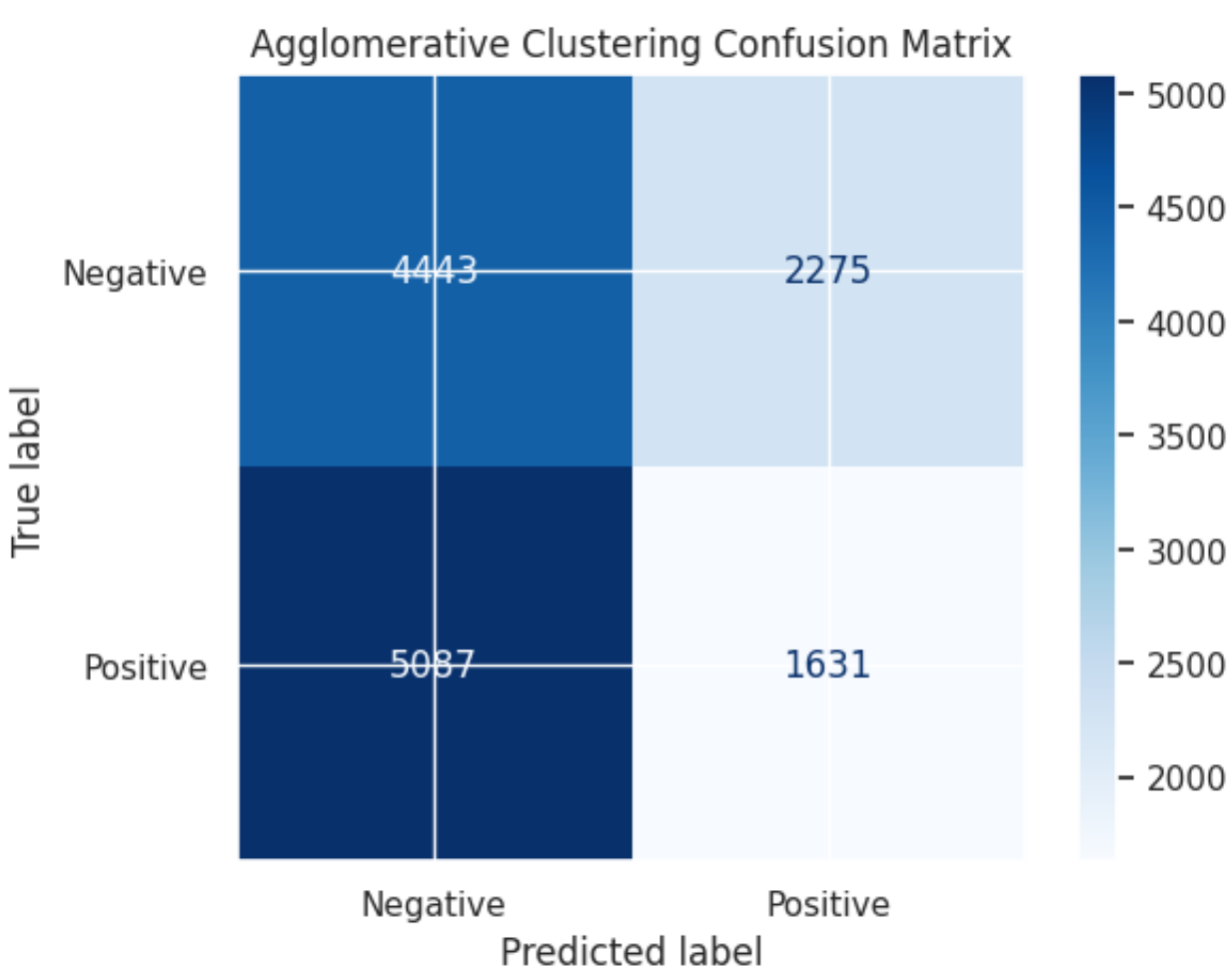
Validation and Testing

K-Means and Agglomerative Clustering

```
kmeans_pred_binary = (cluster_labels == 1).astype(int)

# Generate the confusion matrix
cm_kmeans = confusion_matrix(y_train_resampled, kmeans_pred_binary)

# Plot the confusion matrix
disp_kmeans = ConfusionMatrixDisplay(confusion_matrix=cm_kmeans, display_labels=[
disp_kmeans.plot(cmap=plt.cm.Blues)
plt.title("K-means Confusion Matrix")
plt.show()
```

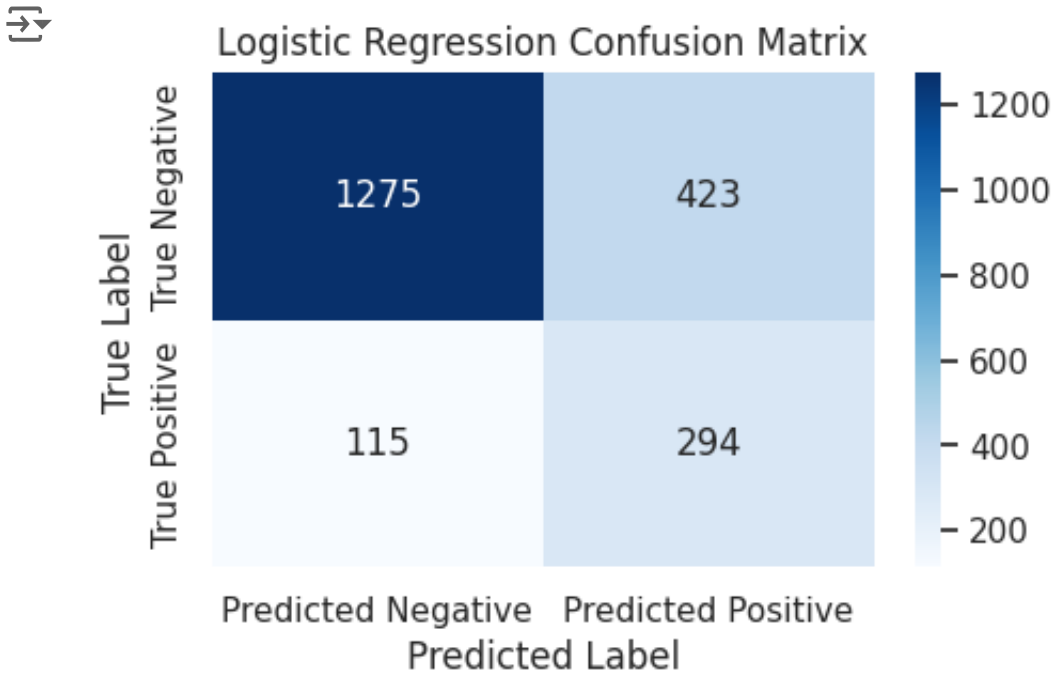


```
agglo_pred_binary = (agg_labels == 1).astype(int)

# Generate the confusion matrix
cm_agglo = confusion_matrix(y_train_resampled, agglo_pred_binary)

# Plot the confusion matrix
disp_agglo = ConfusionMatrixDisplay(confusion_matrix=cm_agglo, display_labels=['N
disp_agglo.plot(cmap=plt.cm.Blues)
plt.title("Agglomerative Clustering Confusion Matrix")
plt.show()
```



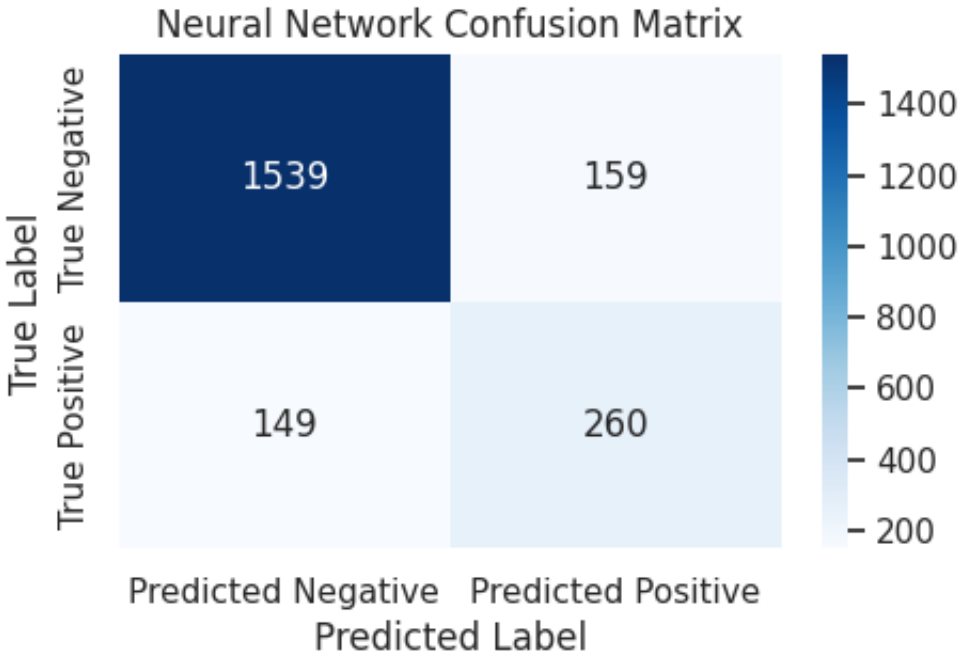
Logistic Regression and Neural Networks

```
# Predict on the test set using Logistic Regression
y_pred_lr = lr_model.predict(X_test)
y_pred_proba_lr = lr_model.predict_proba(X_test)[:, 1]


# Compute the confusion matrix
cm_lr = confusion_matrix(y_test, y_pred_lr)
cm_df_lr = pd.DataFrame(cm_lr, index=['True Negative', 'True Positive'],
                        columns=['Predicted Negative', 'Predicted Positive'])

# Plot the confusion matrix
plt.figure(figsize=(5, 3))
sns.heatmap(cm_lr, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['True Negative', 'True Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Logistic Regression Confusion Matrix')
plt.show()
```



```
# Predict on the test set using Nueral Network
y_pred_proba_nn = nn_model.predict(X_test)
y_pred_nn = (y_pred_proba_nn > 0.5).astype(int)

# Create a confusion matrix
cm_nn = confusion_matrix(y_test, y_pred_nn)
cm_df_nn = pd.DataFrame(cm_nn, index=['True Negative', 'True Positive'],
                        columns=['Predicted Negative', 'Predicted Positive'])

# Plot the confusion matrix
plt.figure(figsize=(5, 3))
sns.heatmap(cm_nn, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['True Negative', 'True Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Neural Network Confusion Matrix')
plt.show()
```

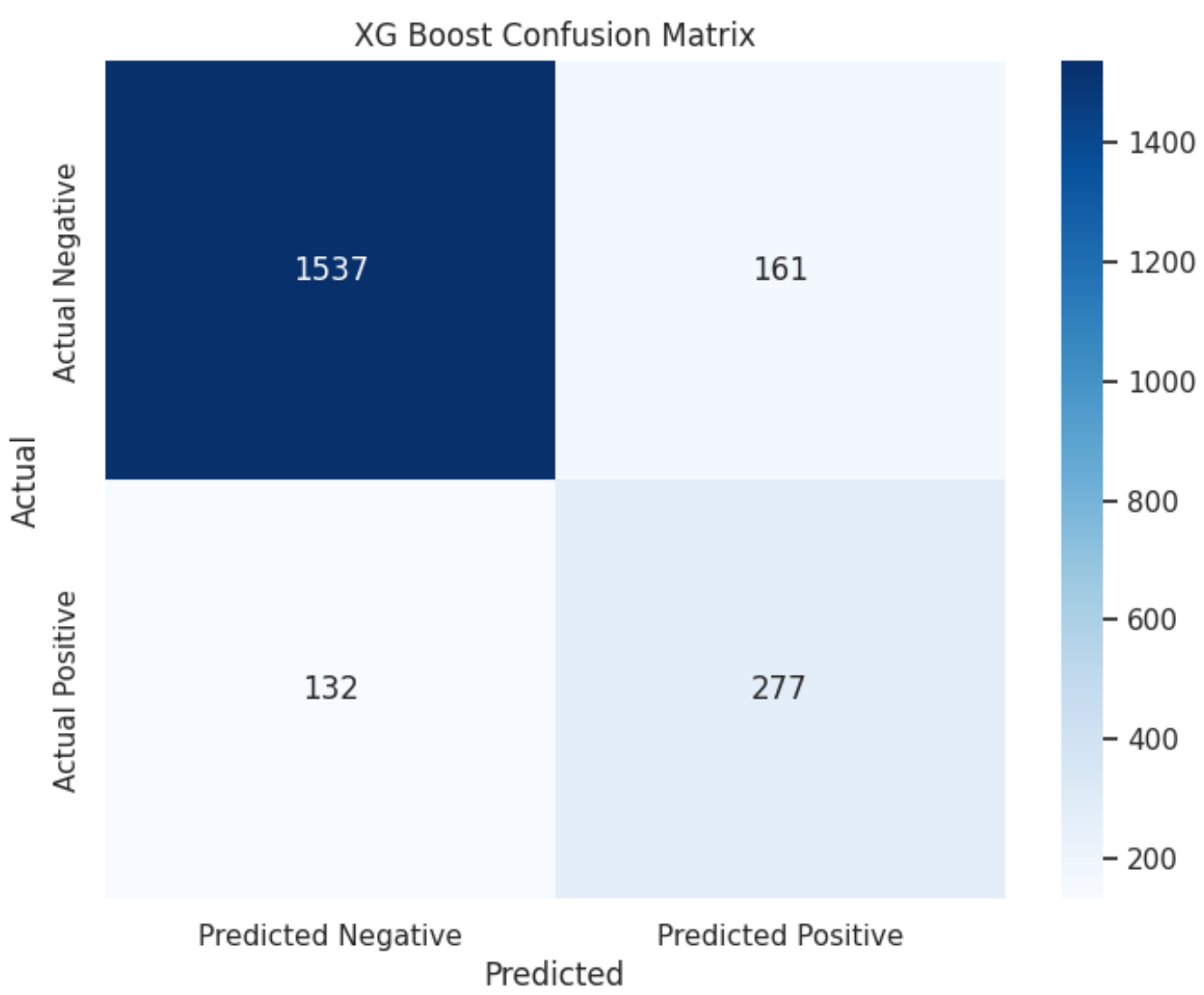
```
66/66 [==============================] - 0s 1ms/step
```



XG Boost and SVM

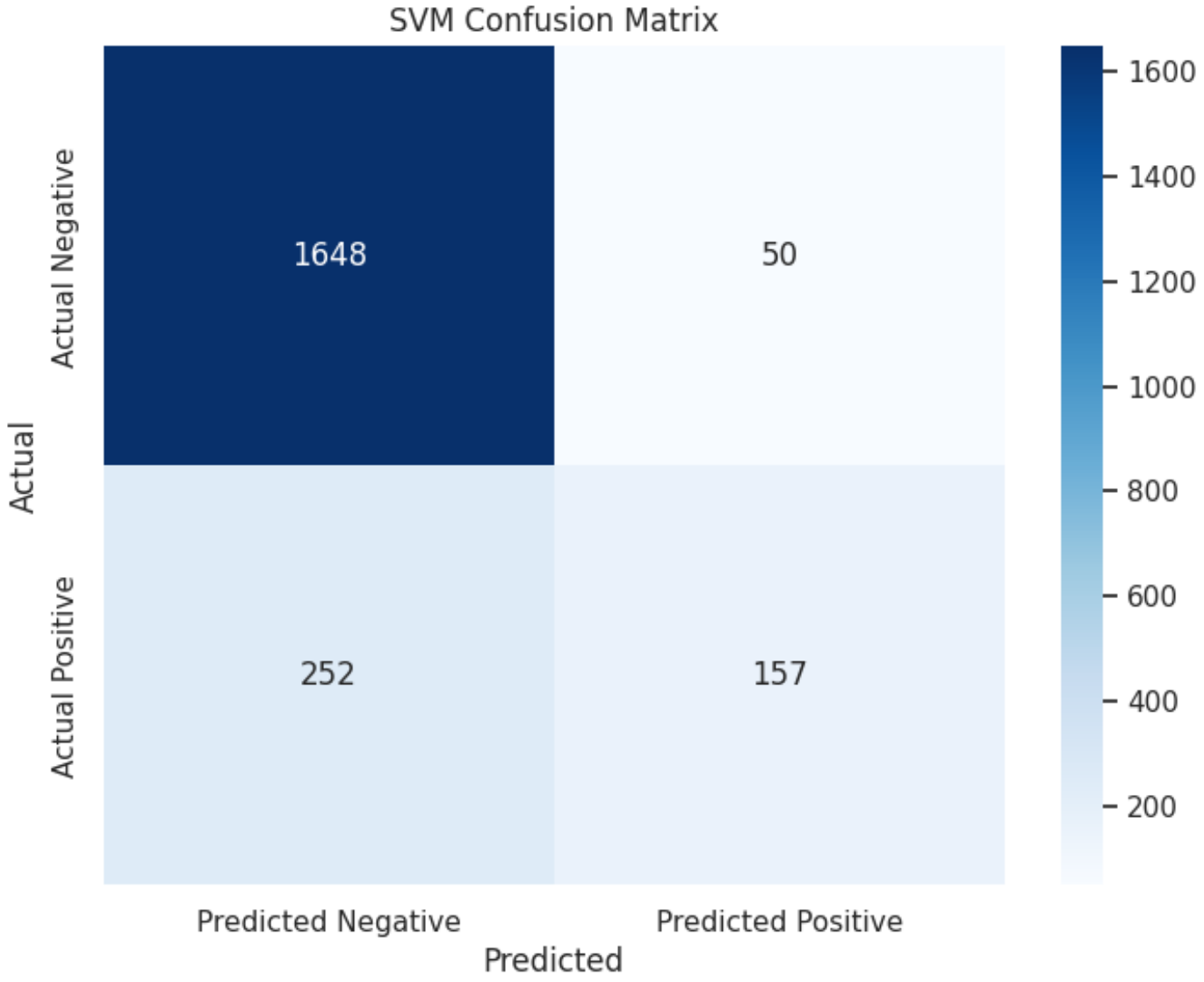
```
# Initialize and train the XGBoost model
model_XGB = XGBClassifier(random_state=42)
```

```
model_XGB.fit(X_train_resampled, y_train_resampled)

# Predict on the test set
y_pred_XGB = model_XGB.predict(X_test)

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred_XGB)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted Negati
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('XG Boost Confusion Matrix')
plt.show()
```




```
# Initialize and train the SVM model with probability estimates
model_SVM = SVC(kernel='linear', probability=True, random_state=42)
model_SVM.fit(X_train, y_train)

# Predict on the test set
y_pred_SVM = model_SVM.predict(X_test)

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred_SVM)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted Negati
```

```
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('SVM Confusion Matrix')
plt.show()
```



Model Performance Evaluation

K-Means and Agglomerative Clustering

```
# Print K-means classification report
print("K-means Classification Report:")
print(classification_report(y_test, binary_labels_test))
```

```
K-means Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.52      0.62      1698
           1       0.15      0.35      0.21       409

    accuracy                           0.49      2107
   macro avg       0.46      0.44      0.42      2107
weighted avg       0.65      0.49      0.54      2107
```

```
# Print Agglomerative Clustering classification report
print("Agglomerative Clustering Classification Report:")
print(classification_report(y_test, binary_labels_test))
```

```
Agglomerative Clustering Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.52      0.62      1698
           1       0.15      0.35      0.21       409

    accuracy                           0.49      2107
   macro avg       0.46      0.44      0.42      2107
weighted avg       0.65      0.49      0.54      2107
```

Logistic Regression and Neural Networks

```
# Print classification report
print("Logistic Regression Classification Report:")
print(classification_report(y_test, y_pred_lr))
```

```
Logistic Regression Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.75      0.83      1698
           1       0.41      0.72      0.52       409

    accuracy                           0.74      2107
   macro avg       0.66      0.73      0.67      2107
weighted avg       0.82      0.74      0.77      2107
```

```
# Print classification report
print("Neural Network Classification Report:")
print(classification_report(y_test, y_pred_nn))
```

```
Neural Network Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.91      0.91      1698
           1       0.62      0.64      0.63       409

    accuracy                           0.85      2107
   macro avg       0.77      0.77      0.77      2107
weighted avg       0.86      0.85      0.85      2107
```

```
# Evaluate the final model on the test set
test_loss, test_accuracy = nn_model.evaluate(X_test, y_test)
print(f"Neaural Network Test Loss: {test_loss}")
```

```
66/66 [==============================] - 0s 1ms/step - loss: 0.4114 - accuracy
Neaural Network Test Loss: 0.41139963269233704
```

XG Boost and SVM

```
# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred_XGB))
print("XGB Classification Report:")
print(classification_report(y_test, y_pred_XGB))
```

```
Accuracy: 0.8609397247271001
XGB Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.91      0.91      1698
           1       0.63      0.68      0.65       409

    accuracy                           0.86      2107
   macro avg       0.78      0.79      0.78      2107
weighted avg       0.86      0.86      0.86      2107
```

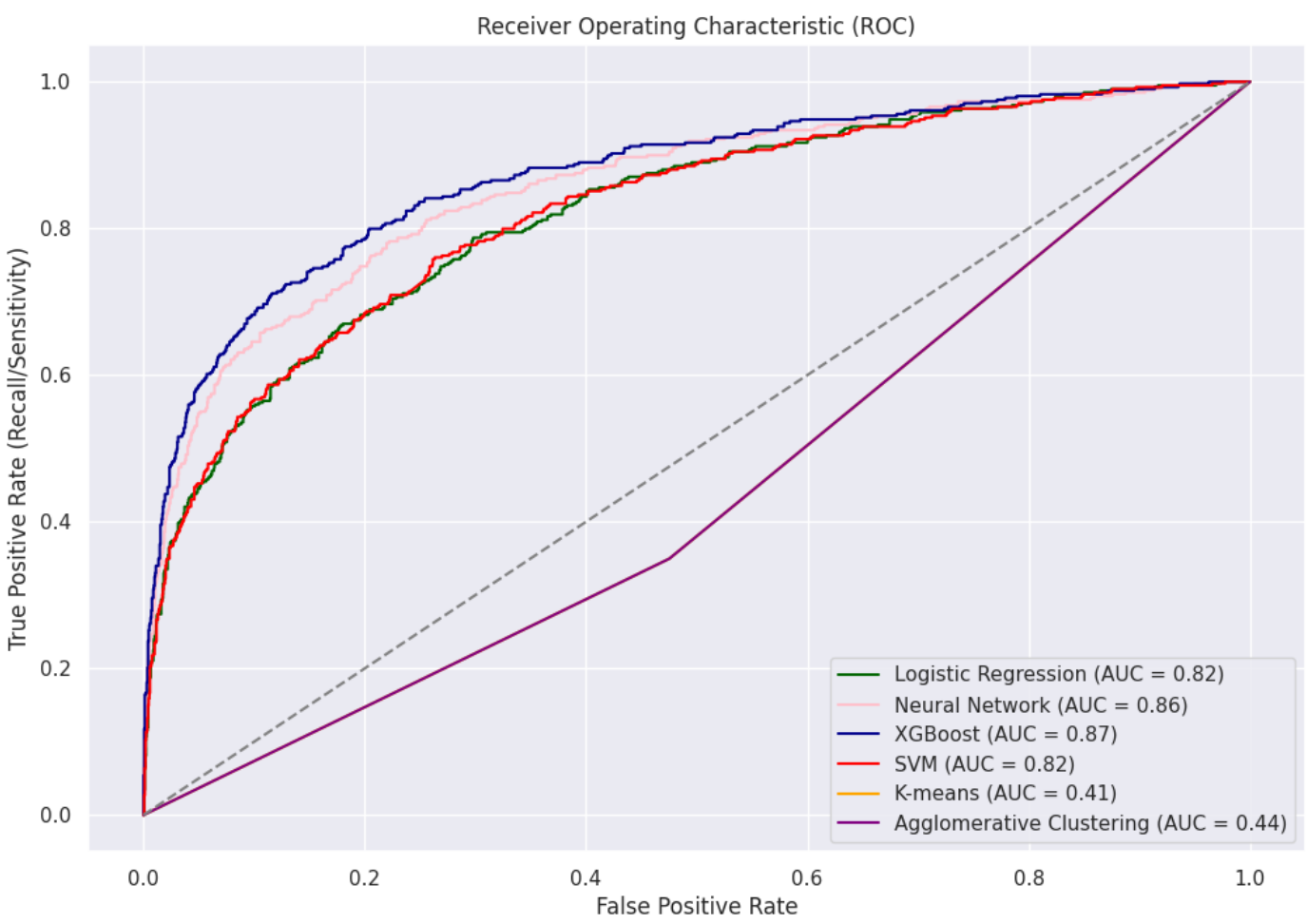
```
# Print classification report and accuracy
y_pred_SVM = model_SVM.predict(X_test)
accuracy = accuracy_score(y_test, y_pred_SVM)
print(f"Accuracy: {accuracy:.2f}")
print("SVM Classification Report:")
print(classification_report(y_test, y_pred_SVM))
```

```
Accuracy: 0.86
SVM Classification Report:
              precision    recall  f1-score   support

           0       0.87      0.97      0.92      1698
           1       0.76      0.38      0.51       409

    accuracy                           0.86      2107
   macro avg       0.81      0.68      0.71      2107
weighted avg       0.85      0.86      0.84      2107
```

AUC-ROC Model Performance Plot

```
# Calculate ROC AUC scores
roc_auc_lr = roc_auc_score(y_test, y_pred_proba_lr)
roc_auc_nn = roc_auc_score(y_test, y_pred_proba_nn)
```

```
# Compute ROC curves
fpr_lr, tpr_lr, _ = roc_curve(y_test, y_pred_proba_lr)
fpr_nn, tpr_nn, _ = roc_curve(y_test, y_pred_proba_nn)

roc_auc_kmeans = auc_score_kmeans
fpr_kmeans, tpr_kmeans, _ = roc_curve(y_test, binary_labels_test)

roc_auc_agg = auc_score_agg
fpr_agg, tpr_agg, _ = roc_curve(y_test, binary_labels_test)

# Plot ROC curves
plt.figure(figsize=(12, 8))
plt.plot(fpr_lr, tpr_lr, color='darkgreen',
         label=f'Logistic Regression (AUC = {roc_auc_lr:.2f})')
plt.plot(fpr_nn, tpr_nn, color='pink',
         label=f'Neural Network (AUC = {roc_auc_nn:.2f})')
plt.plot(fpr_xgb, tpr_xgb, color='darkblue',
         label=f'XGBoost (AUC = {roc_auc_xgb:.2f})')
plt.plot(fpr_svm, tpr_svm, color='red',
         label=f'SVM (AUC = {roc_auc_svm:.2f})')
plt.plot(fpr_kmeans, tpr_kmeans, color='orange',
         label=f'K-means (AUC = {roc_auc_kmeans:.2f})')
plt.plot(fpr_agg, tpr_agg, color='purple',
         label=f'Agglomerative Clustering (AUC = {roc_auc_agg:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate (Recall/Sensitivity)')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc='lower right')
plt.show()
```


Receiver Operating Characteristic (ROC)
True Positive Rate (Recall/Sensitivity)
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Logistic Regression (AUC = 0.82)
Neural Network (AUC = 0.86)
XGBoost (AUC = 0.87)
SVM (AUC = 0.82)
K-means (AUC = 0.41)
Agglomerative Clustering (AUC = 0.44)