2025

BINARY CLASSIFICATION OF SURVIVORS

E0 20o Machine Learning Assignment 1

Course Instructor: Ambedkar Dukkipati (ambedkar@iisc.ac.in)

Submitted By: Swapnil Trivedi (swapnilt@iisc.ac.in)

SR NO: 13-19-01-19-52-24-1-24723

Contents

Table of figures

Abstract

This report presents an in-depth analysis and implementation of four widely used binary classification algorithms: Naïve Bayes, Logistic Regression, K-Nearest Neighbours (KNN), and Support Vector Machines (SVMs).

The primary objective is to evaluate their effectiveness in predicting survival based on a structured dataset. The dataset, derived from real-world passenger records, contains categorical and numerical features that undergo preprocessing, including missing value handling, encoding, and standardization.

A key focus of this study is hyperparameter optimization, performed using Grid Search and Bayesian Search techniques. For SVMs, an extensive search is conducted over hyperparameters C, gamma, and kernel types (Linear and RBF). Models are trained using a 70%-10%-20% data split for training, validation, and testing. The final evaluation compares model accuracy and generalization performance based on the test set. Observations highlight the strengths and weaknesses of each model, providing insights into optimal hyperparameter settings and classifier suitability for similar classification tasks.

1. Data Engineering

The section covers all the pre-processing techniques to prepare the data for classification algorithms.

1.1 Preliminary Data Analysis

The objective is to understand the given data structure and identify potential relationships and attributes which can be leveraged to train multiple binary classification algorithms effectively.

1.1.1 Data Overview

The dataset consists of 891 rows and 12 columns, with a mix of numerical and categorical features. The target variable is Survived (0 = No, 1 = Yes)

1.1.2 Data Types and Feature Summary

The dataset includes **integer, float, and categorical variables**, summarized as follows:

- **Numerical Features:** Age, Fare, Pclass, SibSp, Parch, PassengerId
- **Categorical Features:** Sex, Embarked, Cabin, Ticket, Name
- **Target Variable:** Survived (Binary: 0 or 1)

1.1.3 Missing Values

The dataset contains missing values in three key columns:

Column	Missing Values	Percentage of Missing Data
Age	177	19.9%
Cabin	687	77.1%
Embarked	2	0.2%

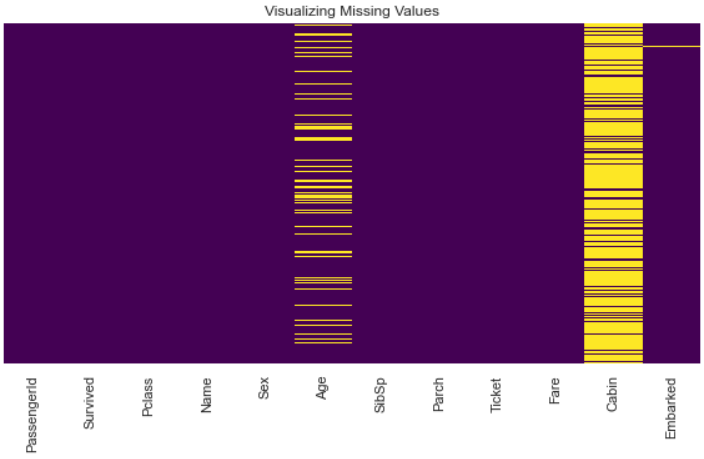

Figure 1 Visualization of missing data

- **Age** can be imputed using either median or mean.
- **Cabin** has a high percentage of missing values (77.1%) and cannot be used directly.
- **Embarked** has only **two missing values** and can be simply dropped.

1.1.4 Statistical Summary of Numerical Features

A statistical analysis of the key numerical features is presented below:

Feature	Mean	Std Dev	Min	25%	50%	75%	Max
Pclass	2.31	0.84	1	2	3	3	3
Age	29.70	14.53	0.42	20.1	28.0	38.0	80.0
SibSp	0.52	1.10	0	0	0	1	8
Parch	0.38	0.80	0	0	0	0	6
Fare	32.20	49.69	0.00	7.91	14.45	31.00	512.3

Key Observations:

- The average age of passengers is 29.7 years, with a standard deviation of 14.53.
- The fare prices are highly skewed, with some passengers paying as much as £512.3.
- Most passengers travelled in third class (Pclass = 3).
- Many passengers had no relatives onboard (SibSp and Parch median = 0).

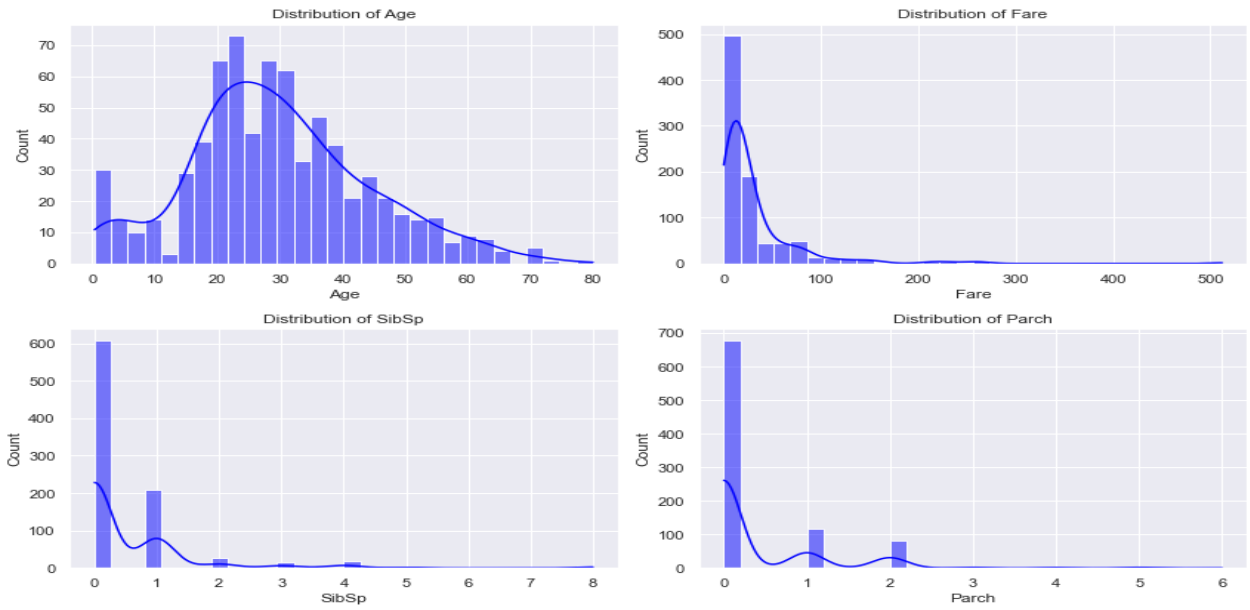

Figure 2 distribution of numerical data

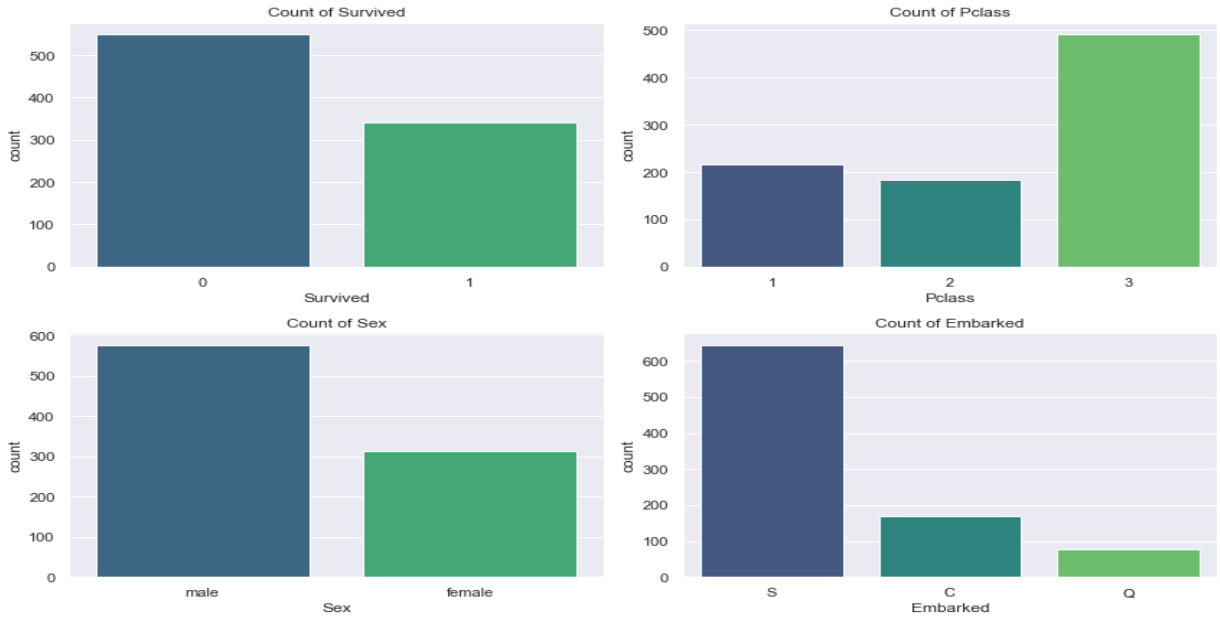

Figure 3 count of categorical data

The dataset is **imbalanced**, with **only 38.4% of passengers surviving**:

- **Survived (1): 342 passengers (38.4%)**
- **Not Survived (0): 549 passengers (61.6%)**

This imbalance may impact classification performance

- **Sex: Male passengers outnumber females**, which may impact survival rates.
- **Embarked:** Most passengers embarked from **Southampton (S)**.
- **Cabin:** High missing values suggest limited usability unless grouped by deck.

1.1.5 Correlation Analysis

- **Strong Positive Correlation:** Fare and Pclass (lower classes paid lower fares).
- **Weak Correlation with Survival:** Age, SibSp, and Parch show little impact.
- **Feature Selection Needed:** Removing redundant or weakly correlated variables may improve model performance.

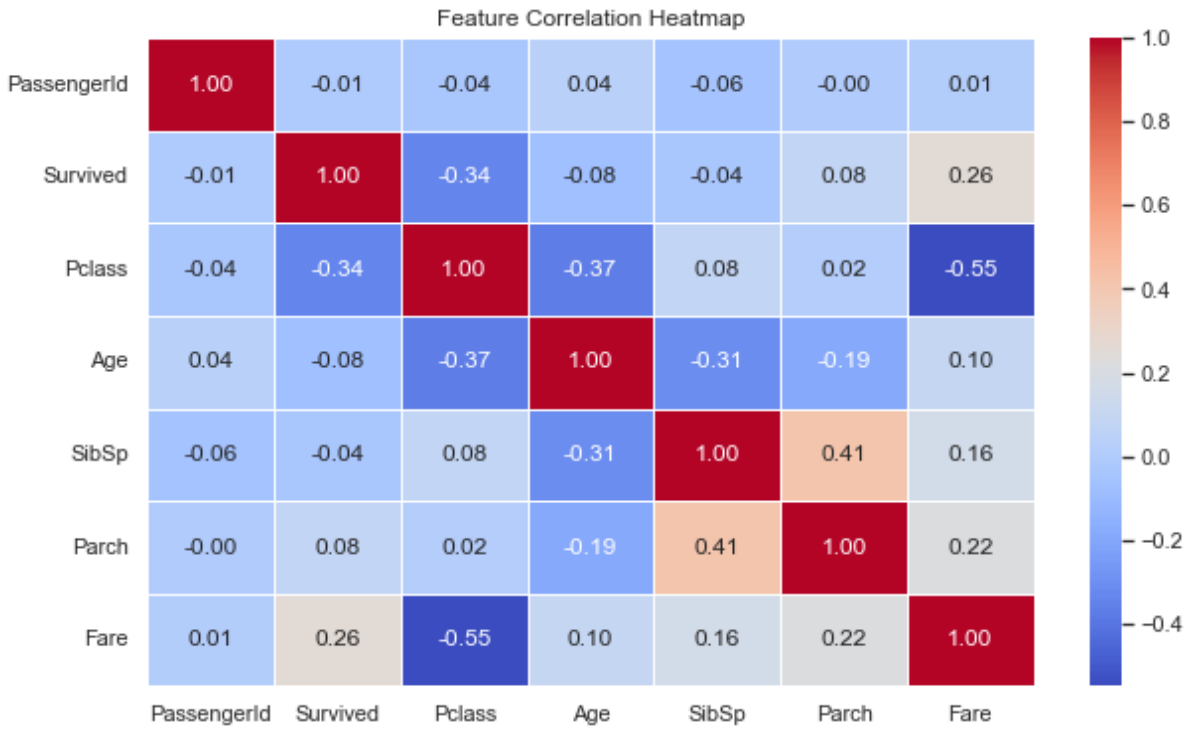

Figure 4 Correlation heatmap of the raw data

1.2 Feature Engineering

From the given data I have exploited the two features Cabin and ticket to extract the features as they seem to have plausible correlation with the target variable survived.

1.2.1 Exploiting Cabin feature

- Cabin data is categorical in nature, with a lot of missing values.
- We have around 147 unique cabins and 687 missing cabins.

As part of the analysis, we can check the survival rate of passengers as per the cabin availability.

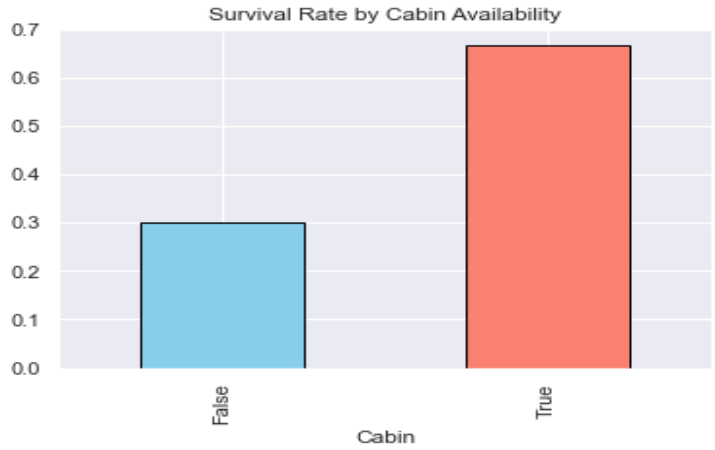

Figure 5 Survival rate by cabin availability

As shown in the figure above, we can conclude that almost 29.98% people who didn't have a registered cabin failed to survive, while 66.67% people survived.

Thus, we can conclude that having a registered cabin is highly associated with the rate of survival.

The available cabins are further divided into categories based on the cabin initials when mapped against the survival of passengers, we can clearly see that some cabins are associated with a higher rate of mortality, indicating potential grouping based on cabin class.

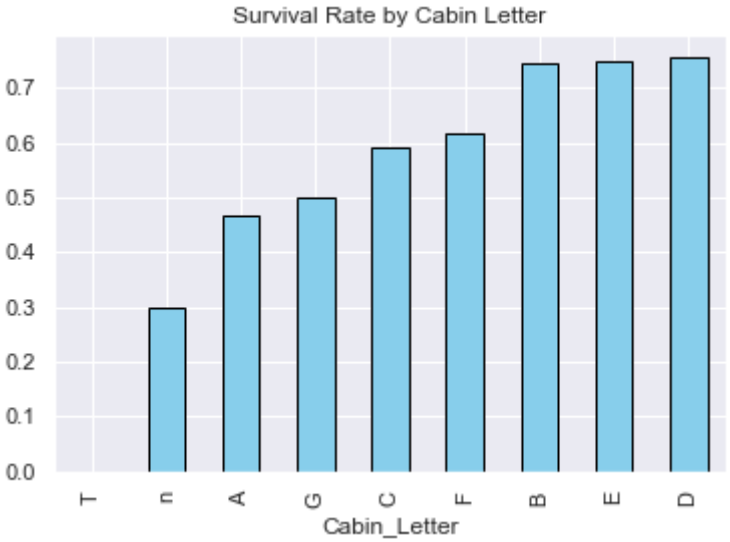

Figure 6 Survival rate by cabin prefix

Since the missing data on cabin is too high, data imputation can lead to false correlations, instead we can assume that having a cabin has higher rate of mortality.

Crafting a new feature HasCabin which is a binary indicator that shows whether the passenger has a cabin assigned or not.

1.2.2 Exploiting Ticket feature

Key observations from the data with respect to the ticket feature.

Everyone had a ticket, there is no missing data. Total number of tickets: 891

Total number of unique tickets: 681, this indicates that almost 210 tickets were being shared among passengers, and they could be travelling in group.

Tickets are a combination of numeric and non-numeric values, we can categories the prefix to distinguish between the ticket types.

Plotting the ticket prefix against the survivability help us understand that some tickets had a better rate of mortality than others, indicating at potential grouping based on class of tickets or passengers.

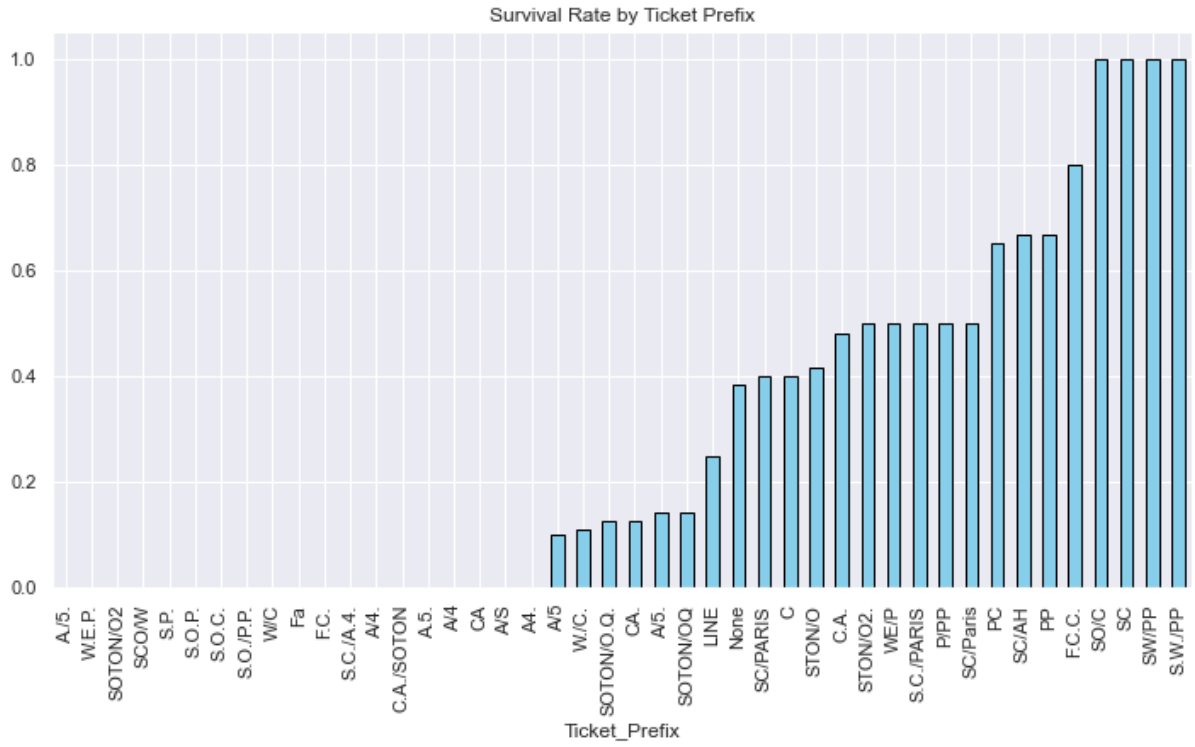

Figure 7 Survival rate by ticket prefix

exploiting the associated prefix of the tickets gives us another feature **HasGroupTickets**, which is a binary indicator grouping passengers having a shared ticket.

1.2.3 Exploiting Siblings and Parents grouping

The data can be combined to get relative features.

Family Size: Compute FamilySize = SibSp + Parch + 1. Larger families might have different survival rates.

IsAlone: A binary feature where 1 means the passenger is traveling alone (FamilySize == 1) and 0 otherwise.

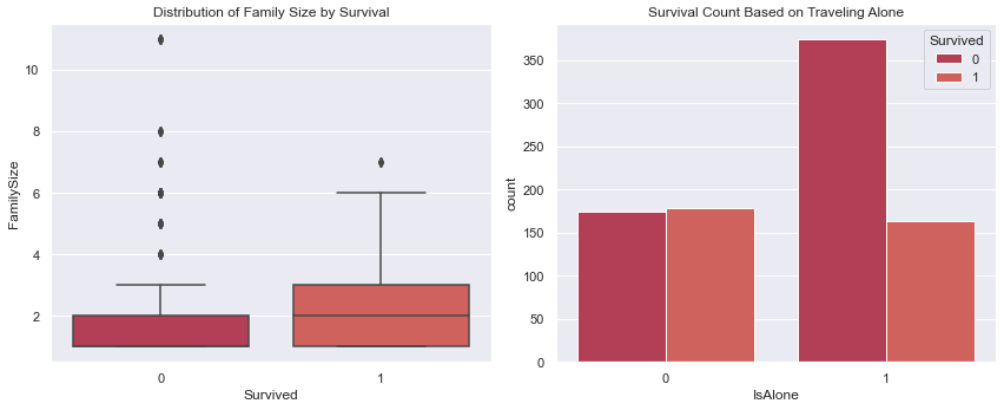

Figure 8 New family features and mortality rate

1.3 Handling Missing Values

From the data, we have the below missing values.

Age 177, Cabin 687, Embarked 2

Figure 9 Missing values in the data

- Cabin has too many missing values and cannot be directly used. We have already extracted required features from it and will drop it from any future processing.
- Embarked has only 2 missing records, which can simply be dropped.
- Age is a significant contributor and needs to be imputed. Can use Mean or Median.
- Name is not used for any feature engineering, will drop it.

1.3.1 Handling missing values in age

- Total available data for age: 712
- Total missing data for age: 177

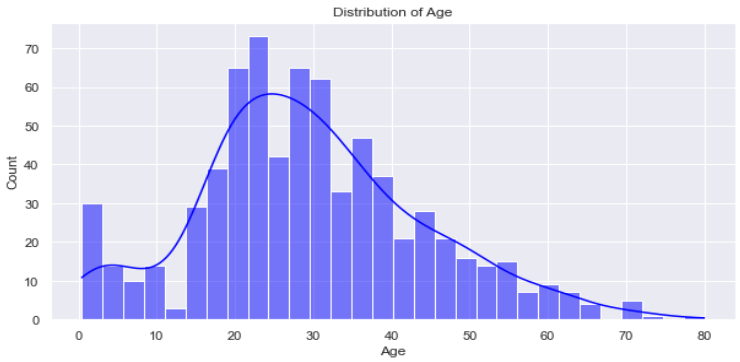

Figure 10 distribution of age

Mean age: 29.64209269662921, Median age: 28.0 Since there are outliers in age and median is more robust, median is used to impute the missing data.

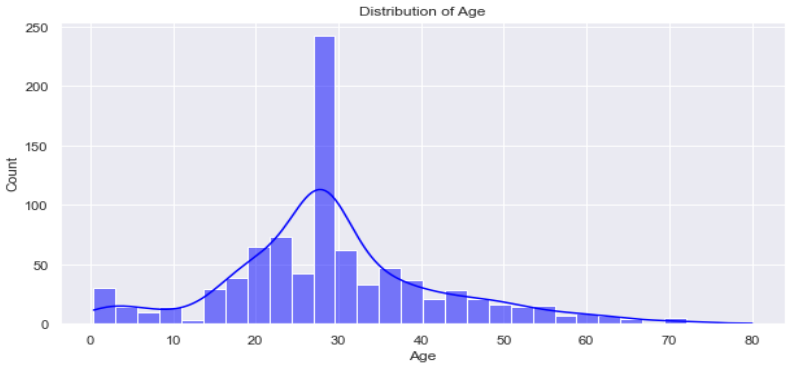

Figure 11 Age data after imputing missing data with median

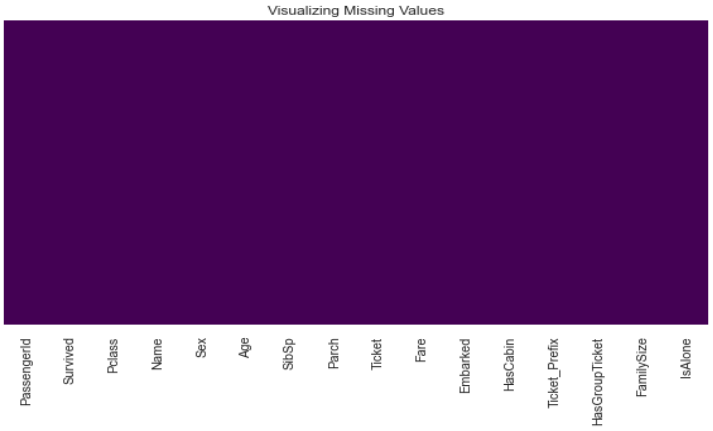

Figure 12 Visualizing missing values post imputation

1.4 Encoding Categorical Data

Columns with categorical data and the encoding strategies used.

- Sex: Label encoding. Male: 0 Female:1
- Embarked: Using one hot encoding to encode the categorical data for embarked.
- Ticket_Prefix: using one hot encoding for ticket prefix data as well.

1.5 Standardizing the values

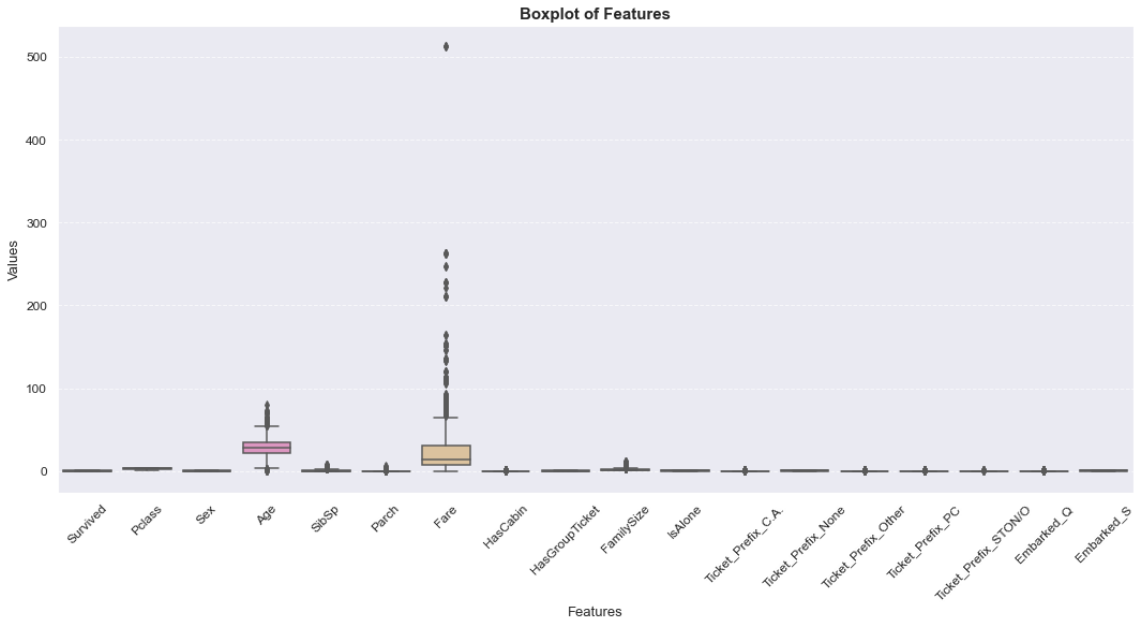

Figure 13 Boxplot to detect outliers

Age and Fare have visible outliers or extremes which can cause issues while training the model and lead to a biased decision making.

We have explored popular techniques like, log transform, Z transformation and square root transformation on Fare column.

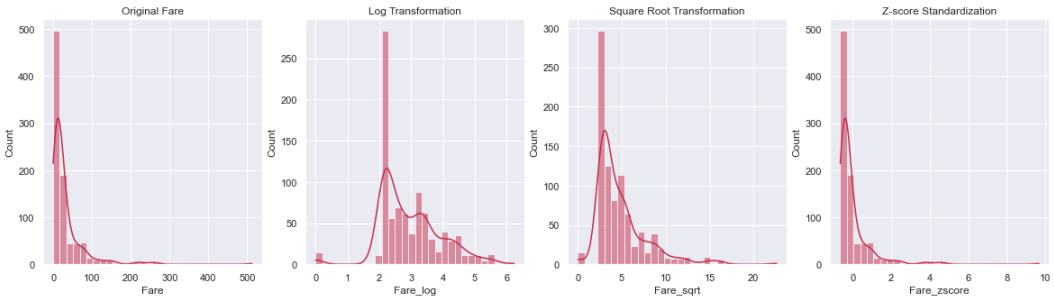

Figure 14 Applying multiple transformation techniques on Fare

Impact of transformation on the values and outliers.

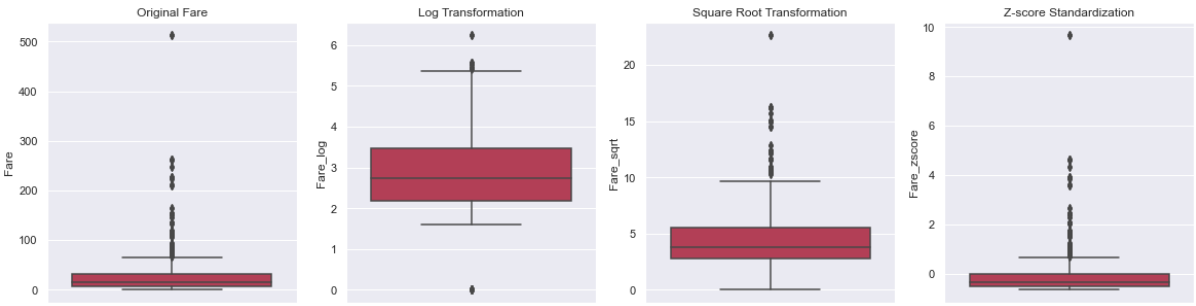

Figure 15 Transformation on Fare

Log transform seems to be effective on the given data by minimizing the outliers compared to Z score and square root transformation.

Similarly, the techniques are applied to the “Age” column.

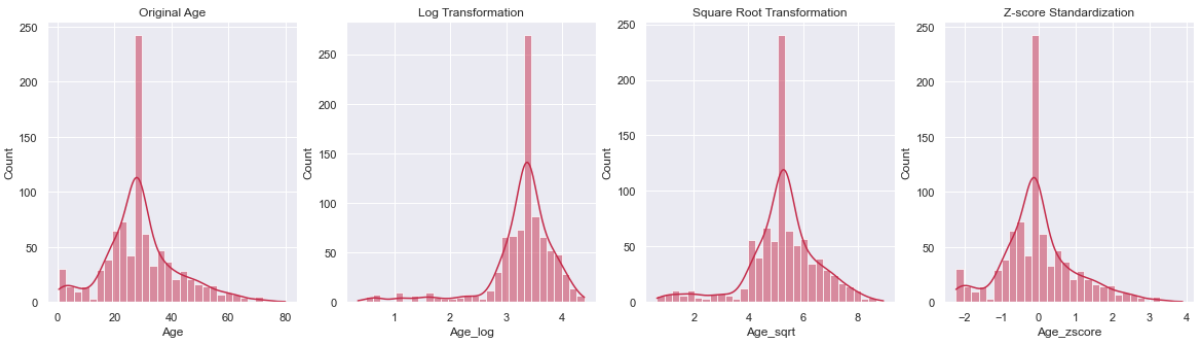

Figure 16 Applying transformation on Age

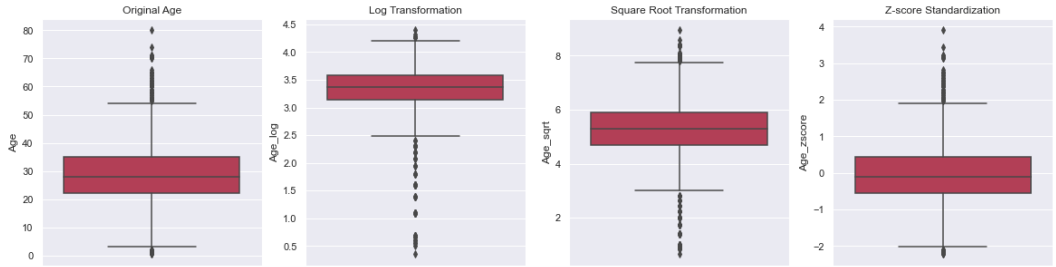

Figure 17 Outliers in age with standardizations

The Z-score standardization regularized the data better compared to other transformations, there still exists some outliers but are not extreme as earlier.

Post transformation the data still have some minimal outliers on multiple columns, which can either be refined or used as such based on the values from the specific models.

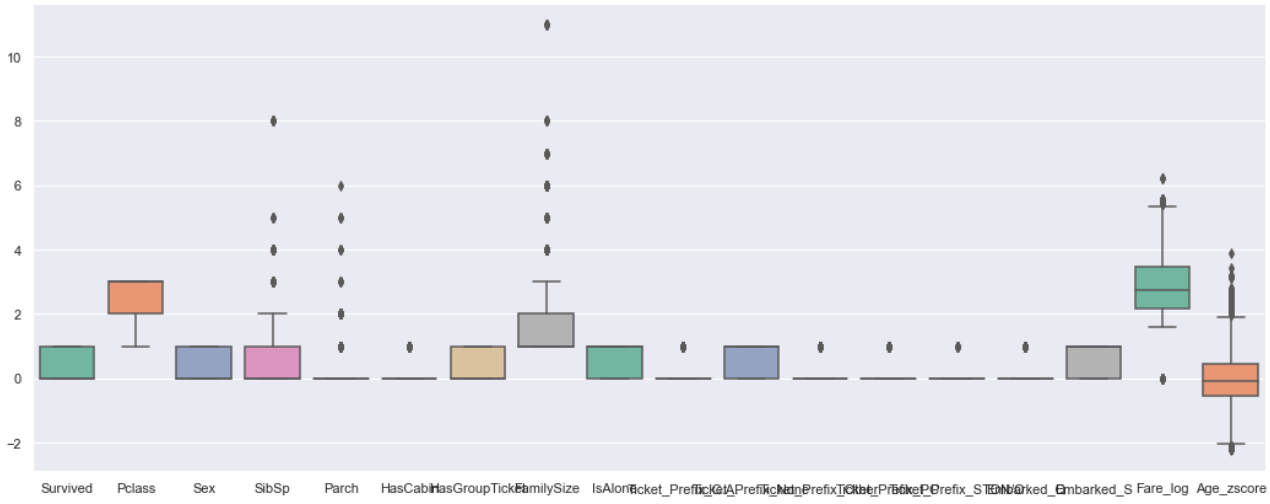

Figure 18 final data outliers

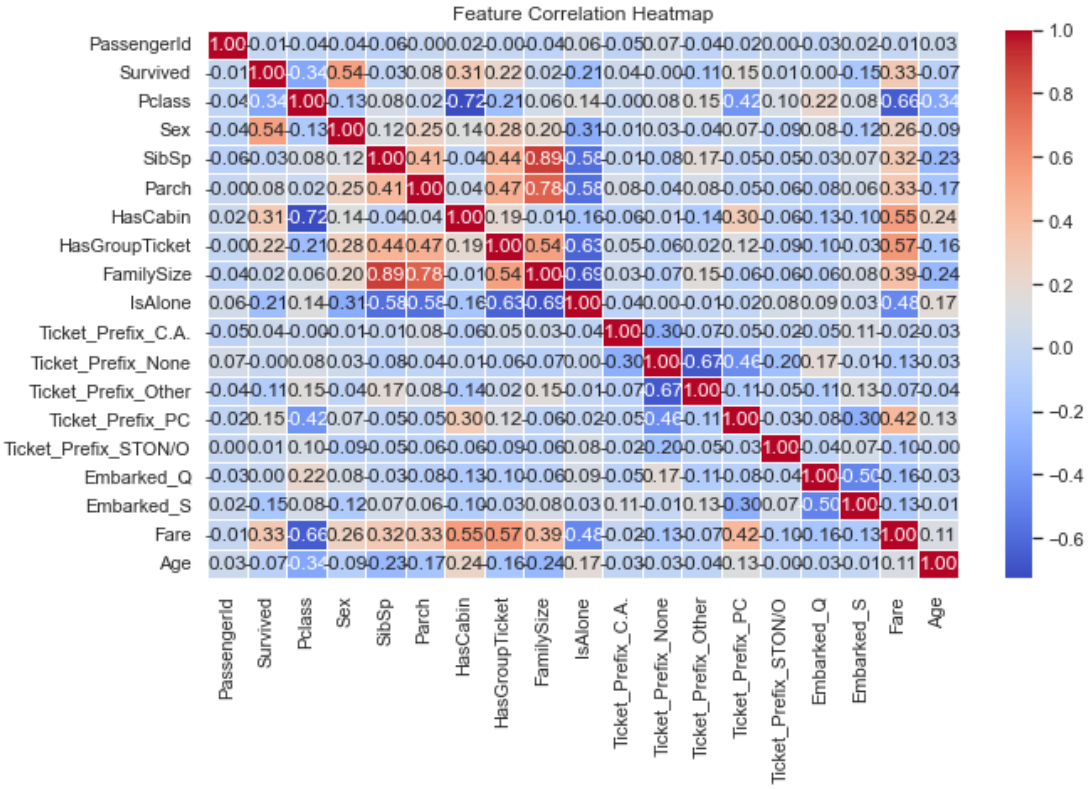

Figure 19 Feature correlation on the cleaned data

We now have a stronger data set with better features to train and test our models.

1.6 Splitting data into training, validation and testing dataset

As per the requirement the entire dataset is split into 3 major chunks training_data which composes of 70% of the data, validation_data which is 10% of the data and test_data which is 20% of the data.

The data is stored in 3 separate files available in the root directory of the project, namely train_data.csv, test_data.csv and validation_data.csv

2. Algorithms and observations

The sections cover all the algorithms and key observations recorded during the assignment.

2.1 Setup and overview

In the section we will cover

- The algorithm used.
- The hyperparameter selected for tuning.
- The optimal hyperparameter using grid search method.
- The optimal hyperparameter using Bayesian search method.
- Comparison between the optimized and unoptimized model.
- Model's performance on the validation set with and without hyperparameter tuning.
- Model's performance on the unseen test data.

Model training and setup

- Each model is trained using the training_data.csv, which consists of 70% the cleaned data from section 1.
- The model is validated using the validation_data.csv which is 10% of the total clean data.
- The model is tested using the test_data.csv which is 20% of the total clean data.
- For hyperparameter tuning the base model (no parameter is selected) and validation data with cv = 3 is used. Training data is not used to avoid overfitting.

2.2 K Nearest Neighbours algorithm

2.2.1 Hyperparameters selected

For the KNN model based the following hyperparameters are selected

- **n_neighbors**: (default=5) Number of neighbours to use by default for Kneighbours queries
- **weights {'uniform', 'distance'}:** (default='uniform')
    - Weight function used in prediction.
    - **'uniform'**: uniform weights. All points in each neighbourhood are weighted equally.

- **'distance'**: weight points by the inverse of their distance. in this case, closer neighbours of a query point will have a greater influence than neighbours which are further away.

- **P** (default=2)
  - Power parameter for the Minkowski metric. When p = 1, this is equivalent to using Manhattan distance (l1), and Euclidean distance (l2) for p = 2. For arbitrary p, Minkowski distance (l_p) is used. This parameter is expected to be positive.

2.2.2 Hyperparameter tuning results

Method	Accuracy	n_neighbors	p	weights
Grid Search	0.7862	14	8	distance
Bayesian Search	0.7747	1	2	distance

based on the above accuracy score the KNN model was trained again using the hyperparameters from the grid search method which promised better accuracy on the validation data set.

2.2.3 Model performance and evaluation

The section covers the model performance and evaluation on the validation data set and test data set before and after the optimization.

Metric	Unoptimized KNN (Validation Set)	Optimized KNN (Validation Set)	Unoptimized KNN (Test Set)	Optimized KNN (Test Set)
Accuracy	75.28%	75.28%	73.60%	75.84%
Precision (Class 0 / 1)	83.00% / 63.00%	85.00% / 62.00%	80.00% / 64.00%	78.00% / 72.00%
Recall (Class 0 / 1)	78.00% / 71.00%	76.00% / 74.00%	75.00% / 71.00%	84.00% / 62.00%
F1-Score (Class 0 / 1)	80.00% / 67.00%	80.00% / 68.00%	78.00% / 68.00%	81.00% / 67.00%
ROC AUC	0.7428	0.7503	0.7312	0.7336

Key Insights:

1. **Optimization Improved Accuracy:**

   - Validation Accuracy remained **the same** at **75.28%**.
   - Test Accuracy improved **from 73.60% → 75.84%**.

2. **Optimized Model Achieved a Higher Recall for Class 0 but Lower Recall for Class 1:**

   - Class 0 recall increased from **75 → 84** (on test data).
   - Class 1 recall **dropped from 71 → 62** (on test data).

3. **ROC AUC Score Improvement was Minimal:**

- Validation: **74.28 → 75.03**.
- Test: **73.12 → 73.36**.

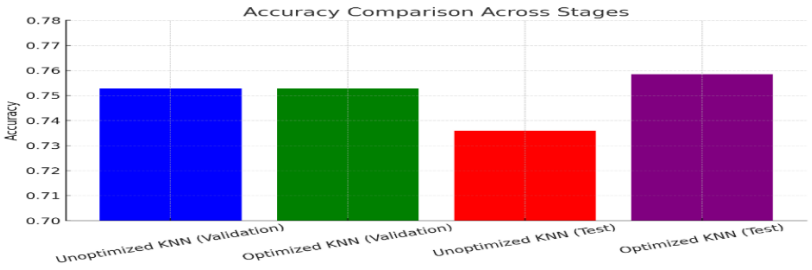

Figure 20 Accuracy comparison of the KNN model

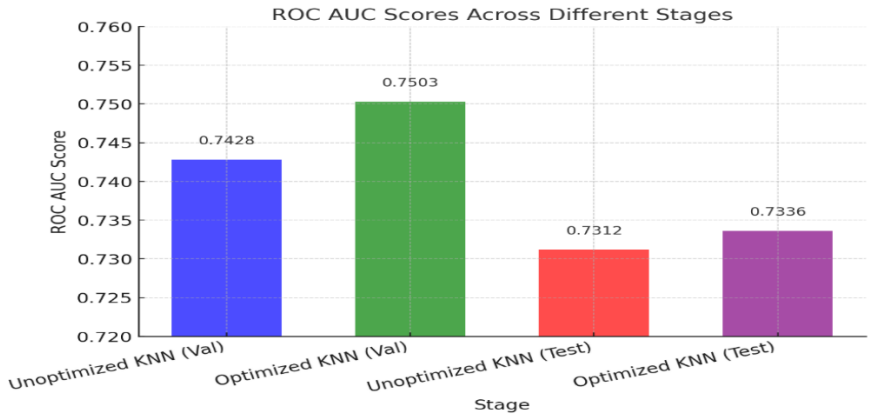

Figure 21 ROC score comparison of the KNN model

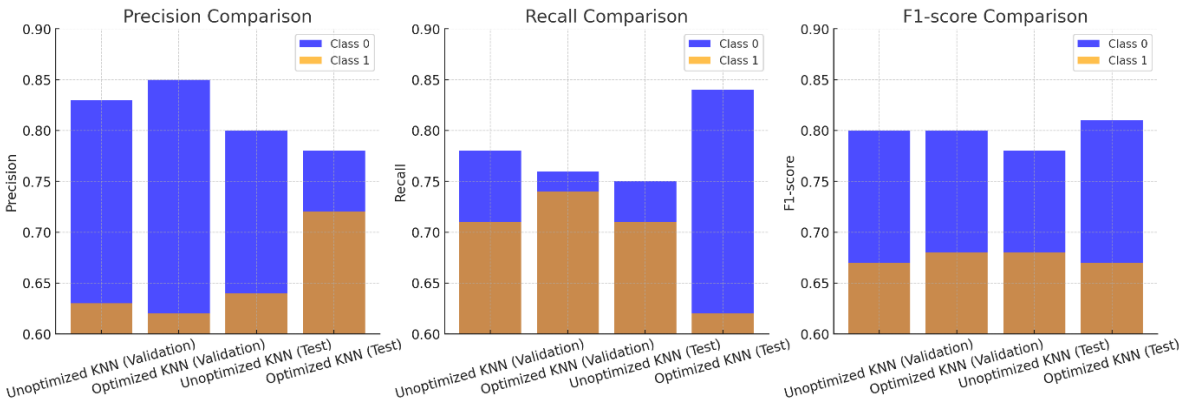

Figure 22 Precision, Recall and F1 Score comparison

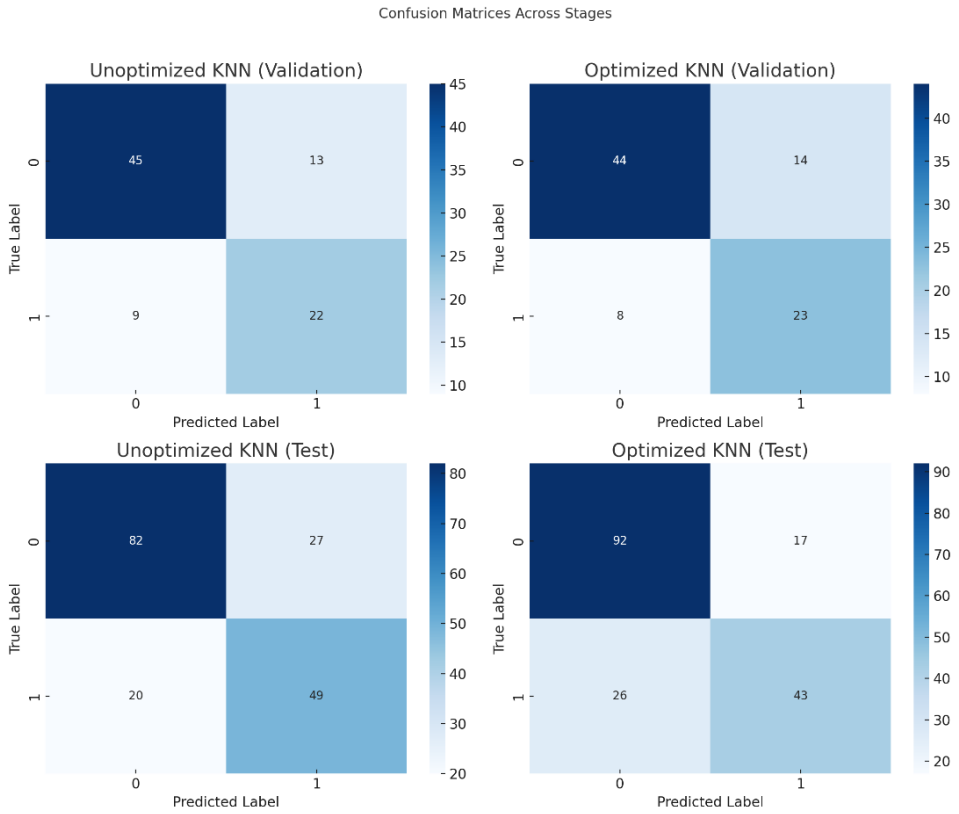

Figure 23 confusion matrix across multiple stages of training and validation for the KNN model

2.2.4 Conclusion

Parameter tuning via Grid Search resulted in only a marginal performance boost in this KNN model. The accuracy and ROC AUC score saw slight improvements, but no drastic gains were observed. This suggests that KNN's performance may be inherently limited by data distribution and feature scaling, and further enhancements might require different preprocessing techniques or an alternative model with better generalization capability.

2.3 Logistic Regression

2.3.1 Hyperparameters selected

- **Penalty**: 'l1', 'l2'
- **C:** (default=1.0) Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.
- **Solver:** 'liblinear', 'saga'. For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones.
- **Max_iter:** 500

2.3.2 Hyperparameter tuning results

Method	C	max_iter	penalty	solver	Score
Grid Search	1	500	l2	liblinear	0.7628
Bayesian Search	3.9058	100	l1	saga	0.7854

Based on the above results, we have tunned the model using the parameters from Bayesian search.

2.3.3 Model performance and evaluation

Metric	Unoptimized (Validation Set)	Unoptimized (Test Set)	Optimized (Validation Set, Bayesian Search)	Optimized (Test Set)
Accuracy	80.89%	78.09%	83.15%	79.78%
Precision (Class 1)	72.00%	73.00%	74.00%	74.00%
Recall (Class 1)	74.00%	68.00%	81.00%	74.00%
F1-Score (Class 1)	73.00%	71.00%	77.00%	74.00%
ROC AUC	0.7934	0.7626	0.8256	0.787

The improvements in accuracy, recall, and ROC AUC highlight the impact of hyperparameter tuning.

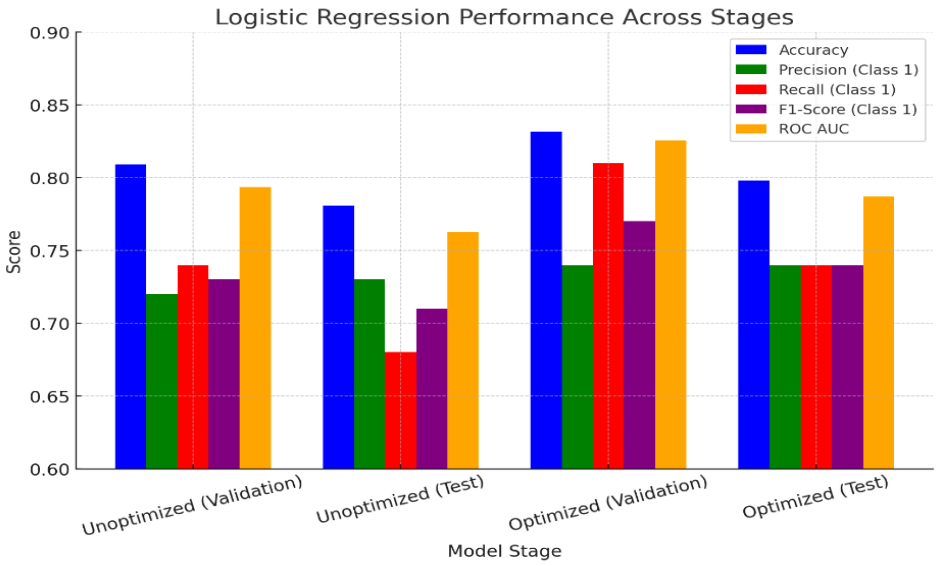

Figure 24 Performance of logistic regression model

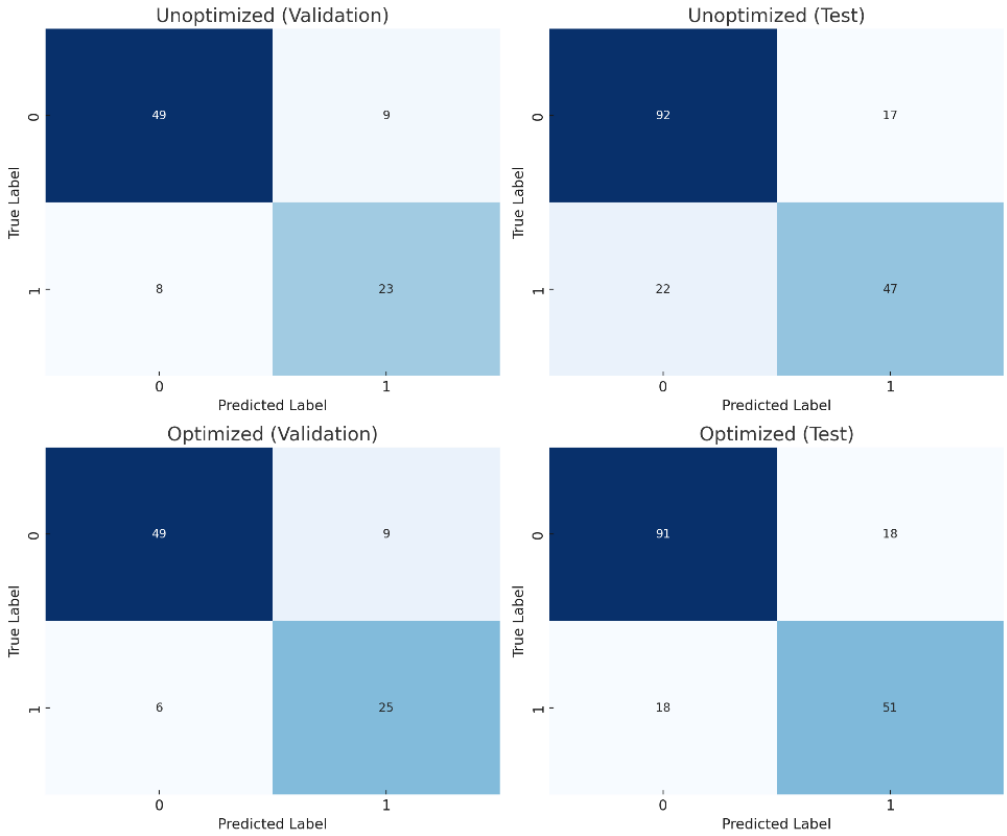

Figure 25 Confusion matrix for logistic regression

2.3.4 Conclusion

Parameter tuning through Bayesian Search significantly improved the Logistic Regression model's performance. The optimized model demonstrated:

- Higher accuracy (Validation: 83.1%, Test: 79.8%) compared to its unoptimized version.
- Increased precision and recall, leading to a more balanced classification performance.
- A better ROC-AUC score, indicating improved separability between classes.

comparing Logistic Regression vs. KNN:

- Logistic Regression outperformed KNN in both validation and test sets, achieving higher accuracy, precision, recall, and AUC.
- KNN struggled with class imbalances, leading to lower recall for the minority class, whereas Logistic Regression handled this better.
- Hyperparameter tuning had a more significant impact on Logistic Regression, improving its robustness, while KNN's performance gains were marginal.

2.4 Naïve Bayes model

The section covers the implementation of a naïve bayes model, specifically a Gaussian naïve bayes classifier.

2.4.1 Hyperparameter selected

var_smoothing*: (default 1e-9)*

Portion of the largest variance of all features that is added to variances for calculation stability.

2.4.2 Hyperparameter tuning result

Method	Best Validation Accuracy	Best Parameter (var_smoothing)
Grid Search	0.7513	0.01
Bayesian Search	0.7513	var_smoothing=0.01 (GaussianNB model)

Both the grid search and gaussian search resulted in the same parameter hence the model was tuned with the var_smoothing value of 0.01

2.4.3 Model performance and evaluation

Metric	Unoptimized - Validation	Unoptimized - Test	Optimized - Validation	Optimized - Test
Accuracy	79.80%	74.20%	82.00%	74.20%
Precision (Class 0)	86.00%	81.00%	86.00%	79.00%
Precision (Class 1)	70.00%	65.00%	74.00%	67.00%
Recall (Class 0)	83.00%	75.00%	86.00%	79.00%
Recall (Class 1)	74.00%	72.00%	74.00%	67.00%
F1-score (Class 0)	84.00%	78.00%	86.00%	79.00%
F1-score (Class 1)	72.00%	68.00%	74.00%	67.00%
ROC AUC	0.785	0.738	0.802	0.728

Key Observations:

- The optimized model improves validation accuracy slightly (+2.2%) but does not improve test accuracy.
- The precision and recall remain relatively stable, indicating minimal gain from hyperparameter tuning.
- The ROC AUC improved for validation data, but slightly decreased for test data, suggesting some overfitting.
- The performance on test data remains the same, indicating that tuning had minimal generalization benefits.

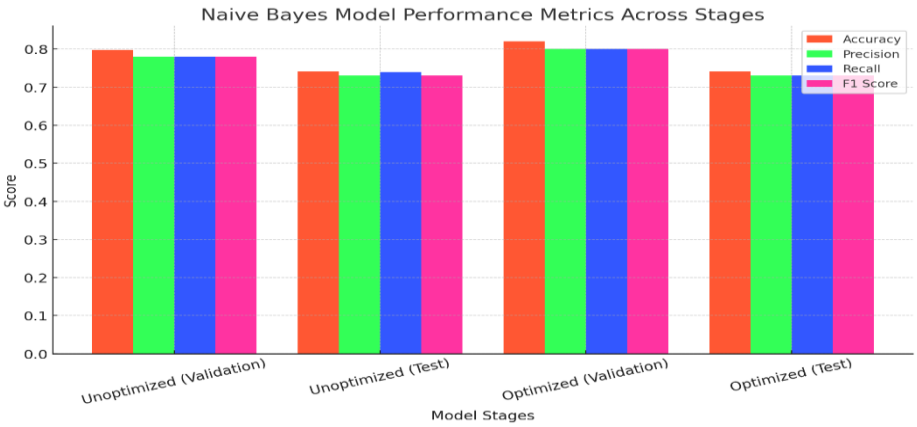

Figure 26 naive bayes model performance matrix

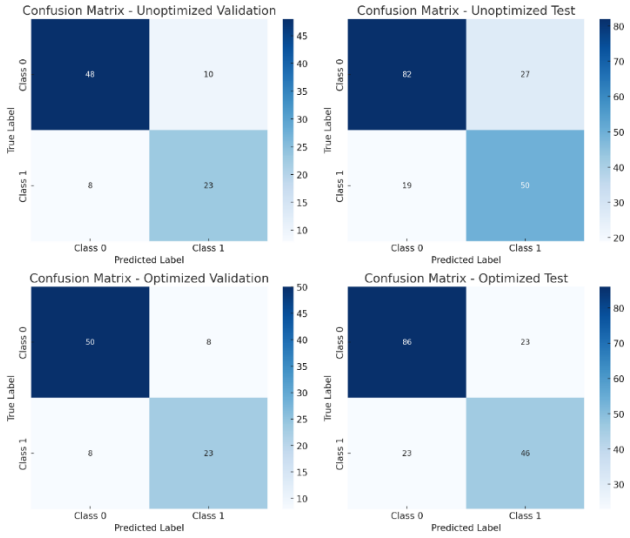

Figure 27 confusion matrix for the naive bayes model across multiple stages

2.4.4 Conclusion

The Naïve Bayes model showed slight improvement after tuning, with validation accuracy increasing from 79.77% to 82.02%, but test accuracy remained 74.15%, indicating limited generalization benefits.

Comparison with KNN & Logistic Regression:

KNN Improved significantly after tuning but not much the test data. Logistic Regression outperformed both models, achieving 79.77% test accuracy and better recall, making it more robust.

Key Takeaways:

- Naïve Bayes is efficient but limited by its feature independence assumption.
- Tuning had minimal impact on test performance.

2.5 Linear SVM

The section implements support vector machine algorithm for binary classification using a linear kernel.

2.5.1 Hyperparameter selected

C*: (default=1.0)*

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty

Other parameters are not compatible with linear kernel

2.5.2 Hyperparameter tuning result

Method	Best Parameters (C)	Best Validation Accuracy
Grid Search	10	0.8418
Bayesian Search	0.2194	0.7739

Based on the above results from tuning, we will be using C=10 as it promises better accuracy on the validation set.

2.5.3 Model performance and evaluation

Metric	Unoptimized (Validation)	Unoptimized (Test)	Optimized (Validation)	Optimized (Test)
Accuracy	80.90%	79.20%	80.90%	79.20%
Precision (Class 0)	86%	81%	86%	81%
Precision (Class 1)	72%	77%	72%	77%
Recall (Class 0)	84%	87%	84%	87%
Recall (Class 1)	74%	67%	74%	67%
F1-Score (Class 0)	85%	84%	85%	84%
F1-Score (Class 1)	73%	71%	73%	71%
ROC Score	0.793	0.769	0.793	0.769

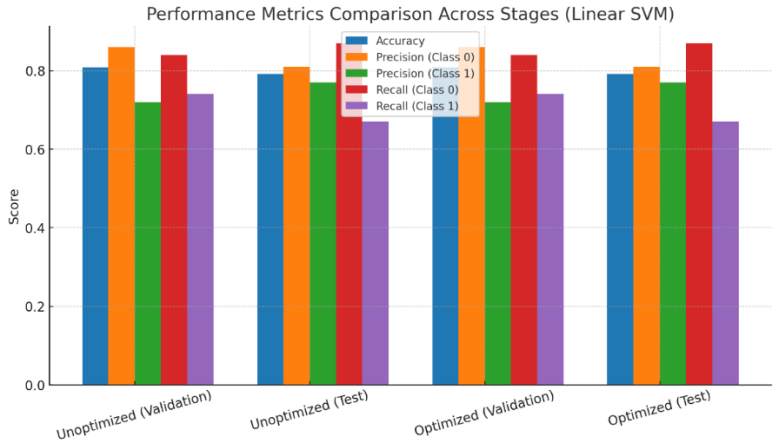

Figure 28 linear svm performance across multiple stages

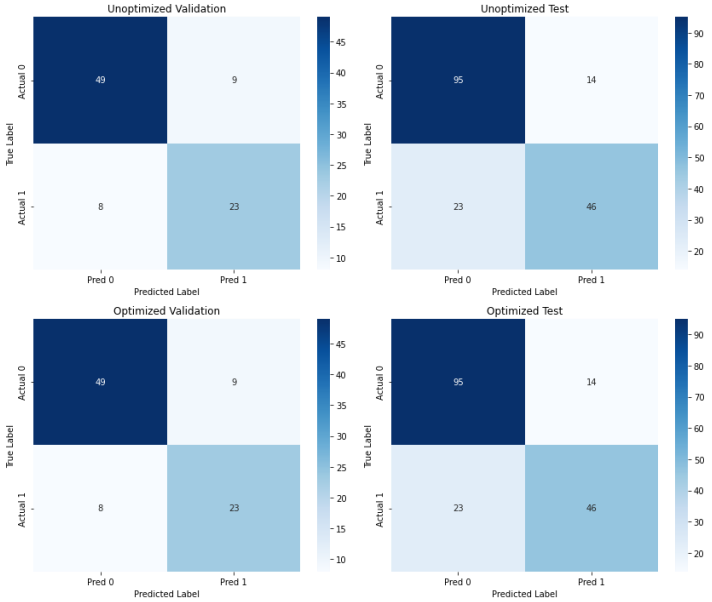

Figure 29 confusion matrix for the linear SVM for all stages

2.5.4 Conclusion

The Linear SVM model demonstrated consistent performance across both the validation and test datasets, with an accuracy of 79.2% on the test set. Hyperparameter tuning did not lead to further improvements, indicating that the model was already performing optimally under the given conditions.

- Compared to KNN, Linear SVM performed significantly better in terms of accuracy and overall generalization, as KNN tends to struggle with high-dimensional data.
- Against Naïve Bayes, Linear SVM had a higher recall and precision, particularly for class 1, suggesting that it handled decision boundaries better.
- Compared to Logistic Regression, both models had similar accuracy, but SVM showed better class separation, particularly in precision and recall for the minority class.

2.6 SVM with Radial basis kernel

The section discusses the performance and analysis on the implementation of SVM classifier using the RBF kernel.

2.6.1 Hyperparameter selected

C*: (default=1.0)*

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty

Gamma: 'scale', 'auto' (default='scale')

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

2.6.2 Hyperparameter tuning result

Method	C	gamma	Best Validation Accuracy
Grid Search	1	scale	0.7628
Bayesian Search	0.7459	scale	0.7739

Based on the parameter tuning Bayesian search parameter proved to be slightly more effective, hence the model was optimized using the same.

2.6.3 Model performance and evaluation

Metric	Validation (Unoptimized)	Test (Unoptimized)	Validation (Optimized)	Test (Optimized)
Accuracy	80.90%	77.50%	82.00%	81.50%
Precision (0)	84.00%	81.00%	86.00%	83.00%
Precision (1)	75.00%	72.00%	74.00%	79.00%
Recall (0)	88.00%	83.00%	86.00%	88.00%
Recall (1)	68.00%	68.00%	74.00%	71.00%
F1-Score (0)	86.00%	82.00%	86.00%	85.00%
F1-Score (1)	71.00%	70.00%	74.00%	75.00%
ROC Score	0.7780	0.7580	0.8020	0.7950

Key observations:

- Optimization improved accuracy and recall, especially for Class 1.
- Balanced precision-recall trade-off, reducing false positives and false negatives.
- Higher ROC score indicates better class separation after tuning.

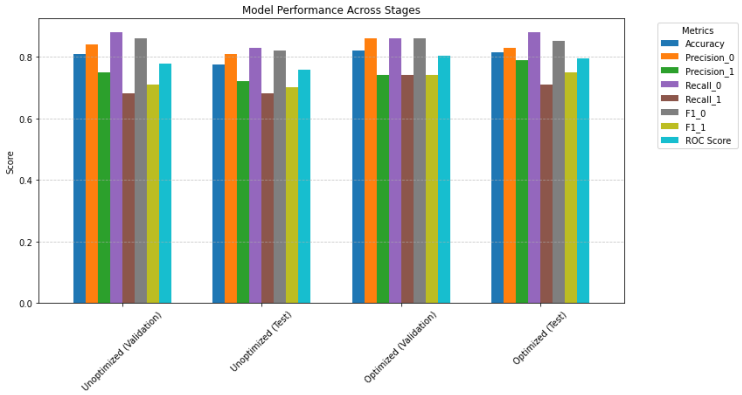

Figure 30 model parameters on all the stages

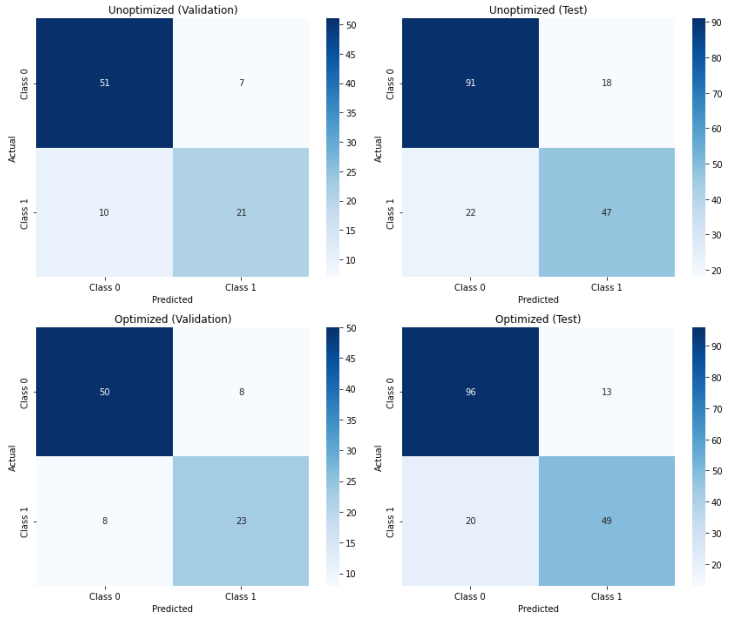

Figure 31 confusion matrix for the RBF SVM

2.6.4 Conclusion

The unoptimized model achieved 77.5% accuracy on test data, but after hyperparameter tuning, performance increased to 81.5% accuracy, along with improved precision, recall, and F1 scores across both classes.

Impact of Model Tuning:

- Better Generalization: The optimized model showed a higher recall for class 1 (positive class), indicating improved detection of minority class instances.
- Reduced Misclassification: The confusion matrix shows fewer false positives and false negatives compared to the unoptimized model.
- Higher ROC Score: An increase from 0.758 to 0.795 suggests better overall discrimination ability.

3. Conclusion and performance overview

Model	Unoptimized model's accuracy on test data	Optimized model's accuracy on test data
KNN	73.6	75.84
Logistic regression	78.09	79.78
Naïve bayes	74.2	74.2
Linear SVM	79.2	79.2
RBF SVM	77.5	81.5

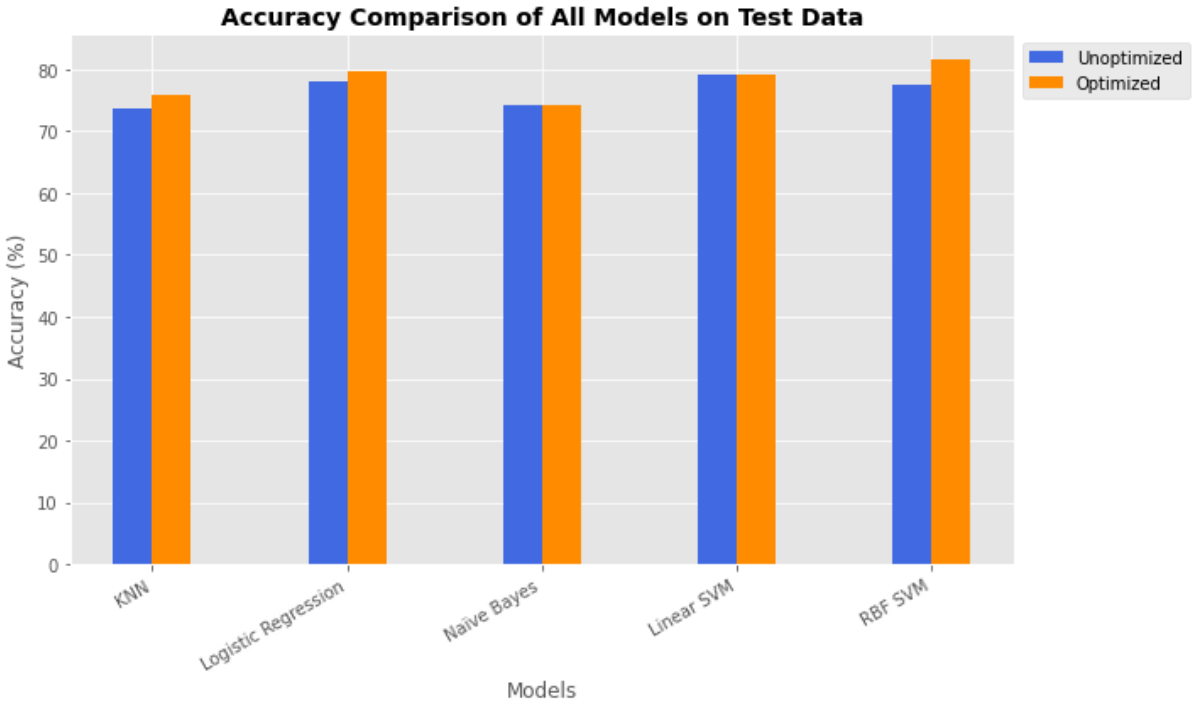

Figure 32 accuracy comparison of all models before and after optimization

1. **Performance Comparison Across Models**
   - KNN: Initially had the lowest accuracy (73.6%), with minor improvement post-tuning (75.84%). Struggles with high-dimensional data.
   - Logistic Regression: A strong baseline model with decent performance (78.09%), improving slightly after tuning (79.78%). Works well for linear relationships.
   - Naïve Bayes: Consistently underperformed (74.2%) and was least affected by tuning, indicating its assumptions didn't align well with the dataset.
   - Linear SVM: Provided stable results (79.2%), but tuning did not further improve its performance.
   - RBF SVM: Showed the most significant improvement from 77.5% to 81.5%, benefiting greatly from hyperparameter tuning and demonstrating strong non-linear decision boundaries.

2. **Key Observations from Parameter Tuning**
    - Models like KNN and Naïve Bayes showed little improvement, indicating that their inherent limitations were more impactful than tuning.
    - Linear models (Logistic Regression, Linear SVM) saw minor boosts, suggesting that their performance was already near optimal.
    - RBF SVM had the most noticeable improvement, proving its ability to capture complex patterns when properly tuned.

3. **Best Performing Model**
    - Final Winner: RBF SVM (81.5%) – It outperformed all other models post-tuning, indicating that the dataset benefited from non-linear decision boundaries.
    - Runner-up: Logistic Regression (79.78%) – A strong, interpretable model that performed well even with minimal tuning.
    - Linear SVM (79.2%) remained competitive but did not surpass RBF SVM.

4. **Final Takeaways**
    - Model selection depends on data complexity: For simpler datasets, Logistic Regression or Linear SVM are reliable. For non-linear data, RBF SVM is superior.
    - Hyperparameter tuning can significantly improve non-linear models, as seen in RBF SVM's jump in accuracy.
    - Naïve Bayes struggled, highlighting the importance of understanding data assumptions before model selection.

4. Code setup and local run

The section briefly covers the code setup and how to run the file locally.

The entire code setup can be found on the github link below:

https://github.com/Swapnil-Trivedi/E0_270_0__ML_2025/tree/main/assignment-01

1. Data cleaning and file generation, this is done using the data_exploration.ipynb file, which explores the data, performs validations, handles missing data, does standardization and split the data into 3 data files.
    - Test_data.csv 20 % of the entire data
    - Train_data.csv 70% of the entire data
    - Validation_data.csv 10% of the entire data
2. Data_handler.py: the file is key and is used to handle the data for all the model classes. It is used to load the data sets and perform a target and feature split from respective data.
3. Algorithms and implementation, all the algorithms follow a general class base model, that has certain methods to load the train data, validation data, test data, fit the model, evaluate the model and to generate the respective evaluation matrix.
4. The algorithms are implemented within the files of respective names, KNN.py, naïve_bayes.py, logistic.py, linear_svm.py and svm_rbf.py
5. grid_search.py: implements functions to tune respective models using the grid search method from the SKLEARN library
6. bayes_search.py: implements function to tune respective models using the Bayesian search from the SKOPT library.
7. Install the required modules or use base condo with all modules packed in.
8. Run individual files to execute the respective models, example “python3 knn.py”

5. References

- KNN parameters https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
- KNN models https://scikit-learn.org/stable/modules/neighbors.html#classification
- Logistic Regression https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- Logistic regression model and guide https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
- Naïve bayes classifier https://scikit-learn.org/stable/modules/naive_bayes.html
- Gaussian Naïve bayes classifier https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB
- SVM classifier https://scikit-learn.org/stable/modules/svm.html
- SVM classification https://scikit-learn.org/stable/modules/svm.html#svm-classification
- SVC https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
- Inspiration for data cleanup and engineering https://www.kaggle.com/code/dellalkhaled/onehot-encoding
- How to handle missing data https://machinelearningmastery.com/handle-missing-data-python/
- https://www.kaggle.com/code/prasenjitsharma/beginner-guide-feature-engineering-data-cleaning