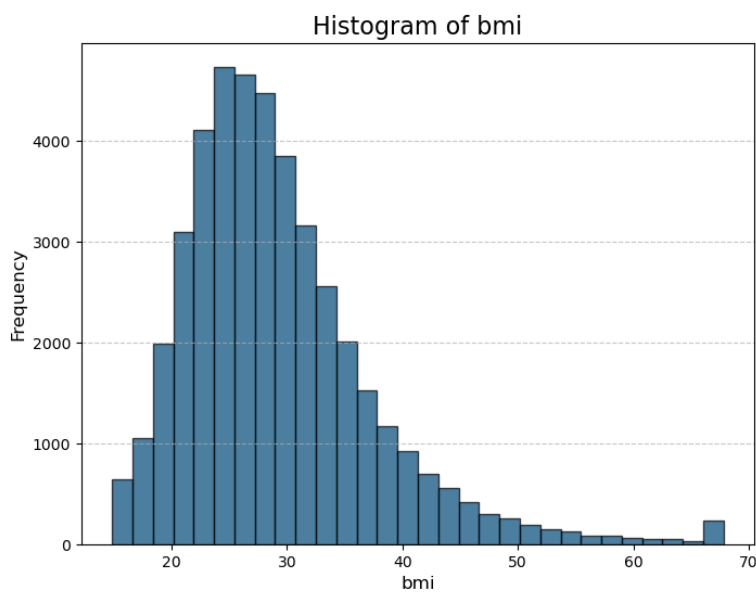


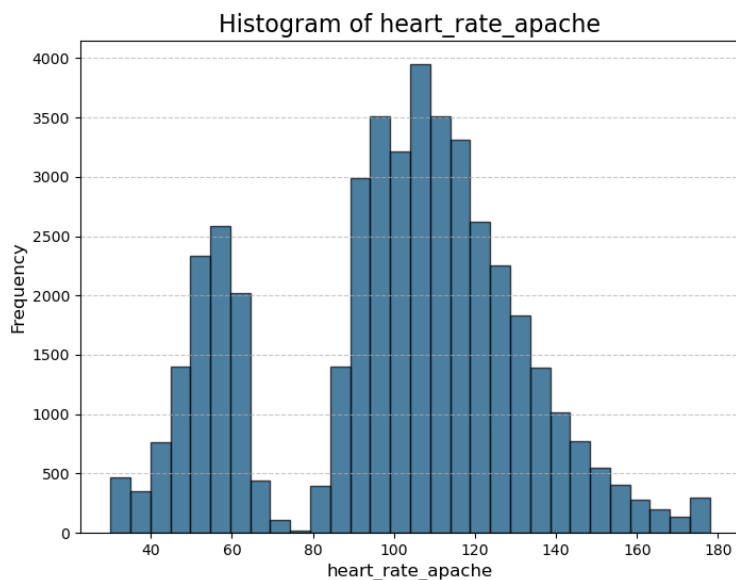
# Report

---

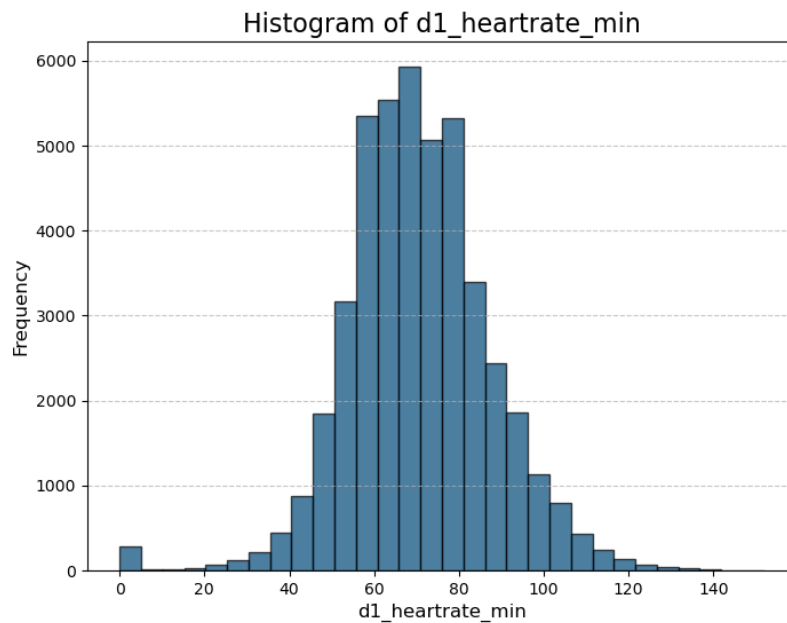
- Data pre-processing and any other data-centric procedures I conducted
  - First, I use histograms to overview the numerical data in the dataset. There are 83 histograms in total with Histogram\_analysis.py. There are 76 numerical data and 7 categorical data presented by histograms. The following figures are some examples.



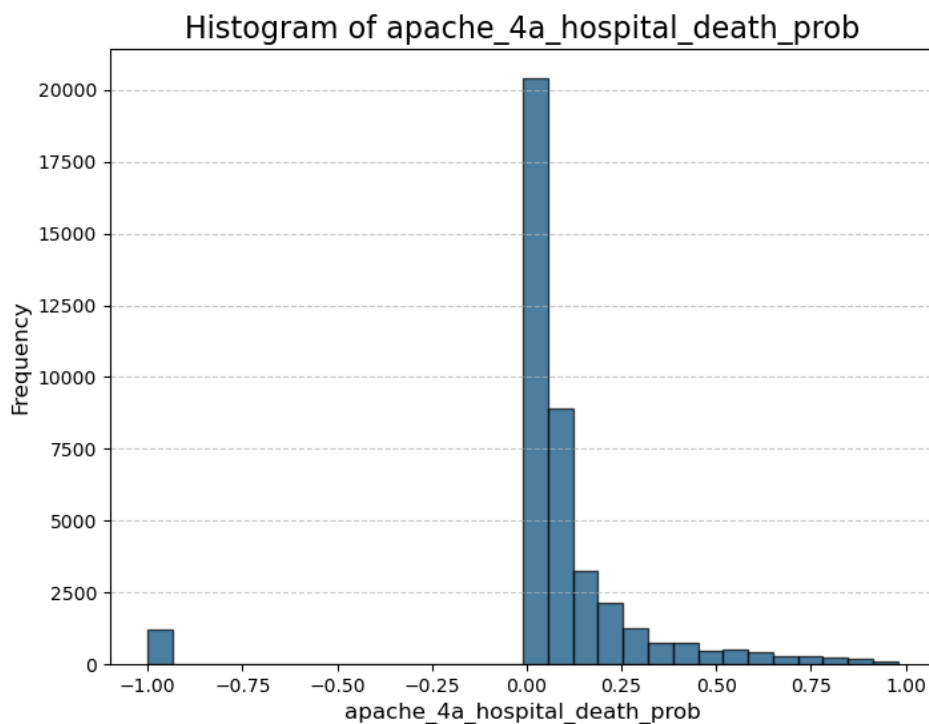
This figure shows that most people have a BMI of 20 to 30, which lies in about the normal values. Of course, there would be some outliers in the extreme values



This figure shows that very few people have a minimum heart rate of 70 to 80, which is close to being an outlier.

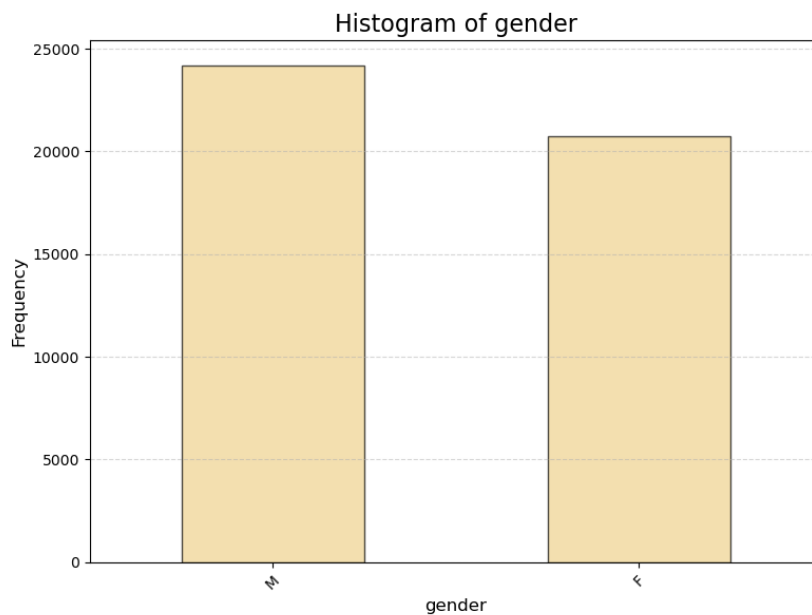


In this figure, we can find some outliers with a zero heart rate, which might be the error of assessment or the patients in the state of shock.

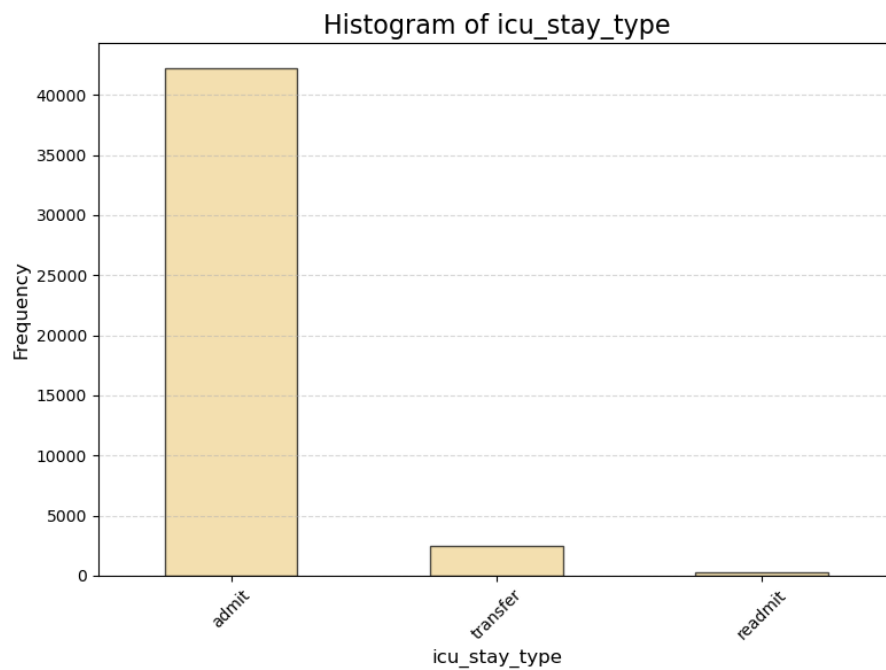


For the most important feature, we find that most patients have 0 probability of death. In this part, those with a -1 probability would be the

outliers and make nonsense of the minus probability value, which might cause some naive errors in the training model.



In this figure, we can argue that gender is almost balanced in the dataset.



In this figure, we can find that most patients' ICU stay type is admitted, while very few are readmitted, which might be seen as outliers.

There are still many features shown by the histogram that can be generated by `Histogram_analysis.py`.

- In the data pre-process procedure, I first identified numerical columns and categories columns, respectively. For numerical data, I filled the loss value with the mean value of other data in the same column. Initially, I used the median value to get worse prediction results. Therefore, I changed to use the mean value in the end. I filled the loss value with the frequently appearing data for categorical data. Then, I used one-hot encoding to transform the categorical data into a value of 0 or 1 to use the data to train the model successfully. For the data imbalance problem, I found that about 91% of data in train data was 0 in the “has\_died” column, which showed the data imbalance. At first, I used SMOTE to try to handle the imbalance problem. However, It wouldn't help the training model. Therefore, I quit balancing the data and focused on adjusting other parameters. In my opinion, the reason behind the unuseful process might be that there are too many features in the dataset, and the number of data is too titanic. Thus, it's pretty difficult to use SMOTE to handle the data imbalance and get high accuracy of the prediction results of the training model. Yet, in the end, I found that I could give higher weight to minority-class samples, ensuring the model pays appropriate attention to both the 1-class and 0-class. Then, I use the parameter to initialize the training model, which I will introduce in detail in the following section.

- Classification methods

- I use extreme gradient boosting (XGBoost) in this assignment. The design of the algorithm of XGBoost is a gradient-boosted decision tree, which has high accuracy and fast training speed.

```
38 scale_pos_weight = len(train_y[train_y == 0]) / len(train_y[train_y == 1])
39 # Initialize XGBoost model
40 xgb_model = XGBClassifier(
41     objective='binary:logistic',
42     random_state=42,
43     eval_metric='logloss',
44     scale_pos_weight=scale_pos_weight
45 )
```

In the beginning, I used scale\_pos\_weight to balance classes by giving higher weights to the minority class samples. We know that the 0-class is about 91% of the training dataset. Thus, I give the higher weights by the division of the number of class 0 and the number of class 1. Then, I initialized the XGBoost model by providing the object of binary classification. (We aim to predict the “has\_died” column, which has a value consisting of either 0 or 1 that is binary.) The random\_state ensures that the random processes in the model are fixed across runs.

```

47 param_dist = {
48     'n_estimators': randint(1000, 3000),
49     'max_depth': randint(10, 15),
50     'learning_rate': uniform(0.01, 0.05),
51     'subsample': uniform(0.6, 0.4),
52     'colsample_bytree': uniform(0.6, 0.4),
53     'min_child_weight': randint(5, 15),
54     'reg_alpha': uniform(0, 5),
55     'reg_lambda': uniform(0, 5)
56 }
57
58 # Use random search to optimize parameters
59 random_search = RandomizedSearchCV(
60     estimator=xgb_model,
61     param_distributions=param_dist,
62     n_iter=150, #from 50 to 75 to 150
63     scoring='roc_auc',
64     cv=5,
65     random_state=42,
66     n_jobs=-1
67 )
68
69 random_search.fit(train_X_encoded, train_y)
70 best_xgb_model = random_search.best_estimator_

```

In this part, I defined the hyper-parameter range. “n\_estimators” means the number of decision trees between 1000 and 3000. In addition, the maximum depth of the trees is from 10 to 15. The learning rate I set initially was from 0.1 to 0.3. However, the range from 0.01 to 0.05 would get better model performance. Maybe it’s because a lower learning rate would get more accurate and stable performance in the condition of many decision trees. Also, I increase the value of “reg\_alpha” to choose less important features. Next, I used a random search to adjust the hyper-parameters of the XGBoost model. I initially set the “n\_iter” to be 50 iterations. However, after several tries, I found that 150 iterations would be better than 75 iterations than 50 iterations. That is, when I randomly choose as many samples as possible, I would get better performance. In addition, I make “n\_jobs” equal to -1 to let all cores compute to get faster speed. Ultimately, I used the best hyper-parameters to train the XGBoost model.

```

75 kf = KFold(n_splits=5, shuffle=True, random_state=42)
76 roc_auc_scorer = make_scorer(roc_auc_score, response_method='predict_proba')
77 f1_scorer = make_scorer(f1_score, average='macro')
78
79 roc_auc_scores = cross_val_score(best_xgb_model, train_X_encoded, train_y, cv=kf, scoring=roc_auc_scorer)
80 f1_scores = cross_val_score(best_xgb_model, train_X_encoded, train_y, cv=kf, scoring=f1_scorer)
81
82 print("Average ROC AUC:", np.mean(roc_auc_scores))
83 print("Average F1 Score:", np.mean(f1_scores))

```

In this part, I did the internal KFold cross-validation to split the dataset into five data groups. In each iteration, 4 subsets are used for training, and 1 subset is used for testing. The process is repeated 5 times, ensuring that all data is used for both training and testing. In addition, I used “random\_state” to ensure the results can be reproduced. In addition, I counted the average AUROC and macro F1-score with the cross-validation method and printed the results.

```
85 # Train the model and do the prediction
86 best_xgb_model.fit(train_X_encoded, train_y)
87 test_predictions = best_xgb_model.predict(test_X_encoded)

90 submission = pd.DataFrame({
91     'patient_id': test_X['patient_id'].astype(int),
92     'has_died': test_predictions
93 })
94 submission.to_csv('./testing_result.csv', index=False)
95 print("Predictions saved to testing_result.csv")
```

In the end, I trained the model with the best hyper-parameters and made the predictions of the testing dataset. The prediction results would be stored in testing\_result.csv with only 2 columns, patient\_id and has\_died.

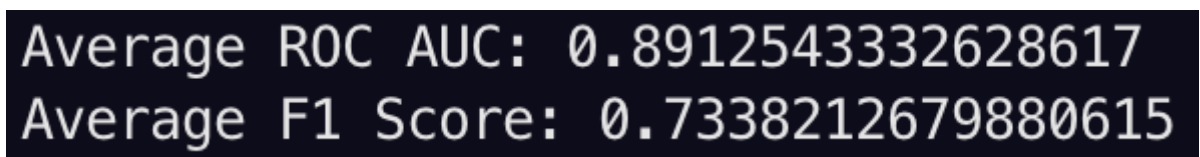
- How to reproduce the results with my source code files?

Two terminal window screenshots. The first shows the command 'python3 ./Code/XGBoost.py' being executed in a directory path of '~/.Desktop/DataMining/Lab\_Assignment\_#2'. The second shows the command 'python3 ./Code/histogram\_analysis.py' being executed in the same directory path.

```
~/.Desktop/DataMining/Lab_Assignment_#2 python3 ./Code/XGBoost.py
~/.Desktop/DataMining/Lab_Assignment_#2 python3 ./Code/histogram_analysis.py
```

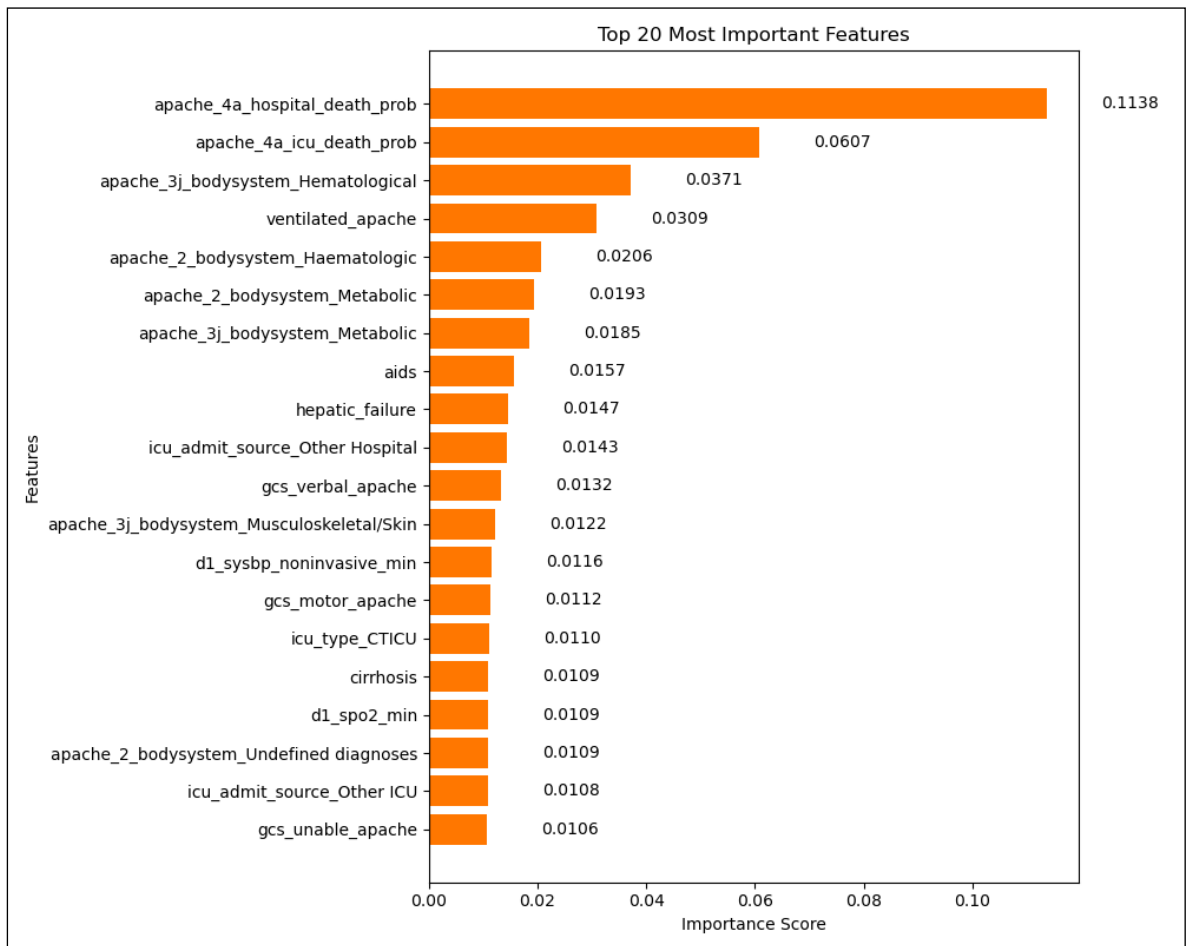
- Results & Analysis

- Screenshot of the results

A screenshot of a terminal window displaying two lines of text: 'Average ROC AUC: 0.8912543332628617' and 'Average F1 Score: 0.7338212679880615'.

```
Average ROC AUC: 0.8912543332628617
Average F1 Score: 0.7338212679880615
```

- The top 20 most important features



Feature	Importance
apache_4a_hospital_death_prob	0.113784
apache_4a_icu_death_prob	0.060689
apache_3j_bodysystem_Hematological	0.037132
ventilated_apache	0.030913
apache_2_bodysystem_Haematologic	0.020569
apache_2_bodysystem_Metabolic	0.019255
apache_3j_bodysystem_Metabolic	0.018516
aids	0.015749
hepatic_failure	0.014676
icu_admit_source_Other Hospital	0.014345
gcs_verbal_apache	0.013212
apache_3j_bodysystem_Musculoskeletal/Skin	0.012218
d1_sysbp_noninvasive_min	0.011598
gcs_motor_apache	0.011232
icu_type_CTICU	0.011015
cirrhosis	0.010946
d1_spo2_min	0.010945
apache_2_bodysystem_Undefined diagnoses	0.010856
icu_admit_source_Other ICU	0.010809
gcs_unable_apache	0.010609

- Some analysis and insights about the high-importance features We can find that the two most important features are the probability of death in the hospital. That is, pre-calculated clinical risk scores are important indicators influencing the prediction of mortality. The feature aggregates the multiple data of the patients, which is critical to predicting mortality. Moreover, conditions such as hematological diseases (e.g., Hematological) and chronic illnesses (e.g., AIDS and hepatic failure) have a significant impact on patient outcomes, indicating that chronic or systemic diseases are critical factors for mortality. In addition, we can also find that the severity of the patient's condition and the ICU medical environment significantly impact the outcomes.
- In summary, since clinical risk scores play a crucial role, it is essential to ensure that the model correctly handles and interprets the impact of these features on prediction results. The high interpretability of the model's feature importance helps provide a reliable basis for medical decision-making.