PATIENT LENGTH OF STAY PREDICTION

in
Data Mining and Machine Learning in Healthcare

Under the Supervision of

Dr. Inas Yassine

Healthcare Engineering and Management
Engineering Department
Faculty of Engineering, Cairo University

Eng. (Peter Salah)

Healthcare Engineering and Management
Engineering Department
Faculty of Engineering, Cairo University

By

Ahmed Mohamed Yehia Habib – 1170323

Rawda Ahmed Shaker Rumaieh – 1170448

Youssef Salah Ahmed Hassan - 1170191

Yumna Hamdy Mohamed Khalifa – 3190001

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT

Table of Contents

Introduction

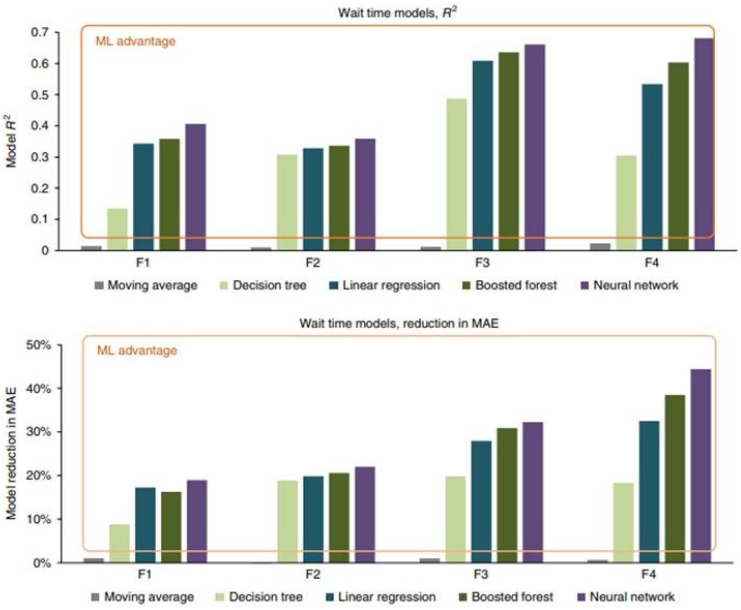
As mentioned in the previous reports, our project predicts the length of stay in hospital. Through this phase we tried some algorithms to find which are going to be proper for our data. Those algorithms were XG-Boost, KNN, SVM, Random Forest and Logistic Regression. The best results we got was the Random Forest and KNN after oversampling the data as will be mentioned through this report. We will also discuss the difference between what we found through the literature review phase from other papers worked on similar datasets and what we did and the difference between the achieved results.

XG-BOOSTING

Result and Discussion

“regarding the past literature review”

As per the literature we submitted in n phase two the paper’s final conclusion was that Boosted Forest gave the second-best accuracy after neural networks. As per the literature we submitted in n phase two the paper’s final conclusion was that Boosted Forest gave the second-best accuracy after neural networks.



That’s why the algorithm I employed was XG-Boosting. The need for using boosted forest was justified by the fact that the outcome of a clinical workflow is affected by a glut of different factors, none of them being decisive enough to exclude the rest. Each of them can have a minor effect therefore they would act as a weak predictor. That’s why ensemble learning was employed so these predictors are brought together for more concrete decisions and representation of the real process (predication of hospital stay)

“didn’t change the pre-processing and used it in”

We had two main obstacles; we tackled the first which was the class imbalance using the SMOTE technique which is (Synthetic Minority Oversampling Technique). SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line. Specifically, a random example from the minority class is first chosen. Then k of the nearest neighbors for that example are found (typically k=5). A randomly selected neighbor is chosen and a synthetic example is created at a randomly selected point between the two examples in feature space.

We tried to tackle the second through using the (Ordinal Encoding) instead of the normal one hot vector encoding the decrease the number of columns. Ordinal Encoding works by(encoding converts each label into integer values and the encoded data represents the sequence of labels) this encoding doesn’t create a new dummy variable for each category instead it encode this category with a label in-place.

“trials”

We First went for Scikit-learn to implement the algorithm however, it didn’t have a GPU support instead I used the Xgboost library itself and it did have a GPU support which worked on google-colab. Due to the limited quota in colab I didn’t have the privilege to retrain the model several times. Also, as I mentioned, the amount of data acted as a stumbling block for me to try a vast range of numbers in each hyper parameter when doing the gird-search.

The parameters I used for grid-search were:

1. n_estimator:400 → Number of gradients boosted trees
2. objective: ['multi:softprob'] → Specify the learning task “multi class classification”
3. max_depth: [3,5,7,9] → Maximum tree depth for base learners

4. min_child_weight: [1,5,10,15] → Minimum sum of instance weight needed in a child
5. subsample: [0.6,0.8,1.0] → Subsample ratio of the training instance
6. colsample_bytree: [0.6,0.8,1.0] → Subsample ratio of columns

```
In [83]:
print(classification_report(y_test,y_pred))
```

	precision	recall	f1-score	support
0.0	0.66	0.66	0.66	12947
1.0	0.43	0.44	0.44	12947
2.0	0.43	0.64	0.51	12947
3.0	0.45	0.24	0.31	12948
4.0	0.68	0.68	0.68	12948
5.0	0.54	0.48	0.51	12947
6.0	0.87	0.91	0.89	12948
7.0	0.70	0.66	0.68	12948
8.0	0.84	0.89	0.86	12948
9.0	0.87	0.92	0.89	12947
10.0	0.88	0.83	0.86	12947
accuracy			0.67	142422
macro avg	0.67	0.67	0.66	142422
weighted avg	0.67	0.67	0.66	142422

when constructing each tree
7. gamma: [0.5,1,1.5,2] → Minimum loss reduction required to make a further partition on a leaf node of the tree
8. n_iter:4 → how many runs in total your randomized search will try

The result of the best estimator

1. colsample_bytree=0.6,
2. gamma=1.5
3. learning_rate=0.300000012
4. max_depth=9
5. min_child_weight=15
6. n_estimators=400

Evaluation

The accuracy score of the best estimator is 67%

Conclusion

The best result obtained by XG-Boost on Kaggle was 60%, I believe better results could be obtained by using different ranges of parameters in

grid-search, also the oversampling technique could be one of the factors of relatively not so high accuracy, we could also see some improved results using different resampling techniques however this wasn't possible due to the quota/ pc capacities.

Random forest

Random forest model was used in almost all the research papers we found. In the first paper Chrusciel et al., 2021, 9, it was used twice one on structured data, and another on unstructured data acquired from the patient's sentences in the EHR data of the hospital. They faced an issue with the imbalance of the data, so their approach was to turn it into a binary classification problem, using the median value of length of stay as a threshold. They achieved a good performance overall getting a 75% accuracy on unstructured data and 74.1% accuracy on structured data.

The second paper Merhan A. Abd-Elrazek et al., 2021, 12, they also used the Random forest algorithm. They got accuracy 57.56%.

The third paper Mekhaldi et al., 2020, 5, They marked the best result with the Random Forest. Since the dataset is large, So Random Forest was a good choice. However, the data was imbalanced missing the oversampling step.

Throughout our project we used the Random Forest algorithm, at the beginning we got accuracy 38%, then we realized that the data is not equally sampled. So, we used an oversampling technique "SMOT" to make the data equally sampled and that affected the accuracy and helped in its increase to reach 90%.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import time

forest_model = RandomForestClassifier(warm_start=True, n_jobs=9, n_estimators=64)
t1 = time.perf_counter()

#train_splits = np.array_split(x_train, 30)
#test_splits = np.array_split(y_train, 30)
# for i in range(0, 30):
#     forest_model.n_estimators += 1
#     forest_model.fit(train_splits[i], test_splits[i])

forest_model.fit(x_train_oversampled, y_train_oversampled)

forest_oversampled_preds = forest_model.predict(X_test_original)
t2 = time.perf_counter()
print(accuracy_score(Y_test_original, forest_oversampled_preds)*100)
print('time', t2-t1)
```

```
89.76436815979906
time 19.452801000000008
```

```
forests_proba = forest_model.predict_proba(X_test_original)
roc_auc_score(Y_test_original, forests_proba, multi_class="ovo", average="macro")
```

```
0.9808336457578593
```

```
print(classification_report(Y_test_original, forest_oversampled_preds))
```

```
              precision    recall  f1-score   support

         0.0       0.91      0.76      0.83      2589
         1.0       0.88      0.92      0.90      9014
         2.0       0.88      0.92      0.90     10047
         3.0       0.92      0.88      0.90      5828
         4.0       0.94      0.87      0.90      1217
         5.0       0.91      0.92      0.91      3104
         6.0       0.97      0.88      0.92       242
         7.0       0.94      0.88      0.91       855
         8.0       0.96      0.89      0.92       137
         9.0       0.95      0.82      0.88       215
        10.0       0.95      0.87      0.91       194
```

Support Vector Machine

The paper Merhan A. Abd-Elrazek et al., 2021, 12, used the SVM model on their used dataset and they got 65.89% accuracy. It produced highly biased results.

Throughout our project we used the SVM algorithm, and without the oversampling technique we got accuracy 30%. But after using Grid search to get the most proper parameters and kernel to be used and we got very low, accuracy which was making sense as the oversampling works on magnifying the small classes in the data to be at the same size of big classes. So, the data became so big which affects the SVM accuracy in a negative way.

Logistic regression

DIFFERENCES FROM PHASE II:

Preprocessing:

1) Null values: “same as phase II”
2) Data and features connection and understanding we take “bed grade” values in consideration as an effective parameter in phase II we drop this feature.
3) Dealing with outliers with interquartile range technique
4) In normalization “same as phase II”
5) In categorical encoding separated the data into
    a. Nominal scale is a naming scale, where variables are simply “named” or labeled, with no specific order.
    b. Ordinal scale has all its variables in a specific. Using (ordinal encoder) and (onehotencoder)

6) We found that oversampling with SMOTE in logistic regression have a negative impact on the model so used the data after splitting without oversampling.

Model:

1) Trained and test data were (0.85,0.15) we changed it to (0.8,0.2).
2) Increasing the max number of iteration in the model to 10,000

Grid search on all parameters took infinite time to run on our laptops using both Colab and Jupyter, so by research and studying the documentation of logistic regression on scikit-learn we conclude the best combinations between all the parameters and also by trying to run them on small data or small number of iterations.

Parameters in grid search were

1) Penalty: [l1, l2, elasticnet, none],
2) C: np.logspace(-4, 4, 10),
3) solver: [lbfgs , newton-cg , sag , saga],
4) maximum number of iterations: [1000,2500, 5000]

- We found that number of iteration needed to be higher to get better accuracy and to perform good on such a huge data so we made it 10,000, also the best C value was the default one which is 1.0 and the same goes for fit_intercept=True, intercept_scaling=1 and class_weight=None, So we tried to run the combination between the penalties and solver that can work together since according to their documentation not all penalties can work with all solvers.

Examples from the Combinations in the notebook:

```
## Grid search on all parameters took infinite time to run:) so by research and studing the docmentation we conclude that these are the best combinati
from sklearn.linear_model import LogisticRegression
# from sklearn.model_selection import GridSearchCV
logModel3 = LogisticRegression(penalty='l2',solver='liblinear',max_iter=10000,n_jobs=-1)
t1 = time.perf_counter()
logModel3.fit(X_train, y_train.values.ravel())
t2 = time.perf_counter()
print('time', t2-t1)
####
print (f'Accuracy - : {logModel3.score(X_test,y_test):.3f}')
```

```
time 43.492888759999914
Accuracy - : 0.385
```

```python
import time
## Grid search on all parameters took infinite time to run:) so by research and studing the docmentation we conclude that these are the best combinati
from sklearn.linear_model import LogisticRegression
# from sklearn.model_selection import GridSearchCV
logModel = LogisticRegression(penalty='l2',solver='newton-cg',max_iter=10000,n_jobs=-1)
t1 = time.perf_counter()
logModel.fit(X_train, y_train.values.ravel())
t2 = time.perf_counter()
print('time', t2-t1)
####
print (f'Accuracy - : {logModel.score(X_test,y_test):.3f}')
```

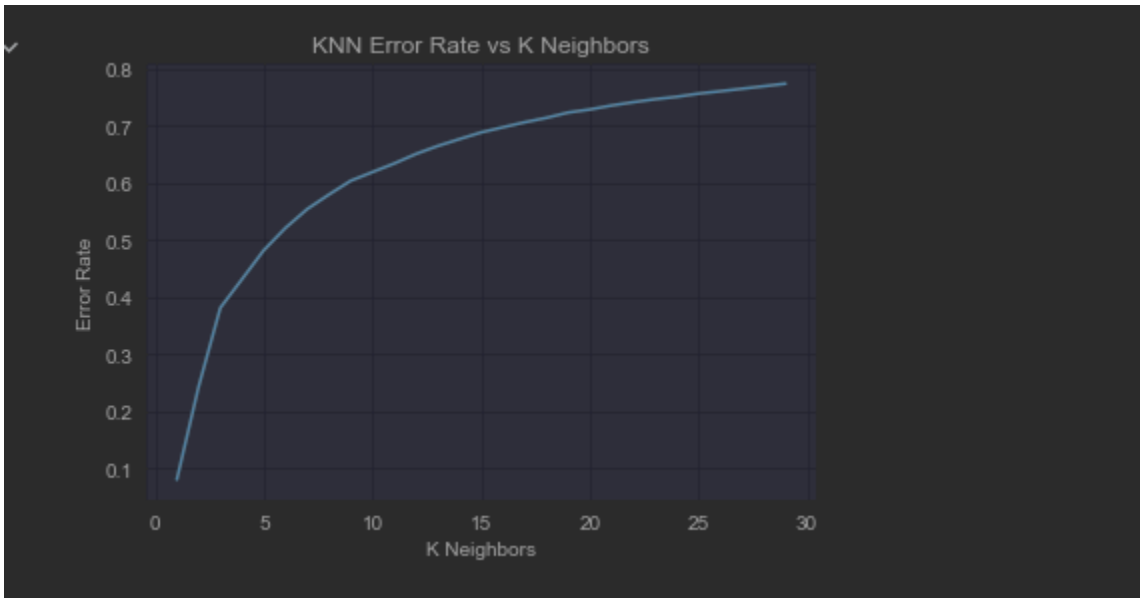
```
time 667.8699520449999
Accuracy - : 0.391
```

KNN

Per the literature review, the KNN was only used in the second paper and its performance wasn't that good, it achieved around 56% accuracy, but I thought that it might be a good model for this case since we have a lot of data which is usually good for the KNN when we preprocess the data well, using normalization with the oversampling made the KNN much better achieving around 92% accuracy in our case as well as the best run time with 4.5 seconds to fit.

I think the main reason behind this performance is the amount of data, as well as the oversampling method we used, SMOTE mainly uses KNN to generate new data of the smaller classes to balance the data, so I thought that using KNN to train on the data with this way the augmented data is generated was actually kind of intuitive, and when I started to test the model with the default parameters of SK-learn it

actually did pretty good and the run time was a bit surprising to me but it was good, so I decided to investigate further with the elbow method and try different values of neighbors and from the plot we can see that as the number of neighbor increases the accuracy actually decreases but the runtime was always between 4 seconds and 5 seconds so it was the best run time we could get out of all the models as well as the best accuracy.



The KNN model had the highest accuracy and right behind it was the random forests, so we investigated further and decided to study another evaluation metric which is the AUC and interestingly the random forests had a better AUC score which introduced a trade-off and we went for the KNN model as the best model mainly for the run time as well as the accuracy score.

```
knn_proba = knn_model.predict_proba(X_test_original)
roc_auc_score(Y_test_original, knn_proba, multi_class="ovo", average="macro")
```

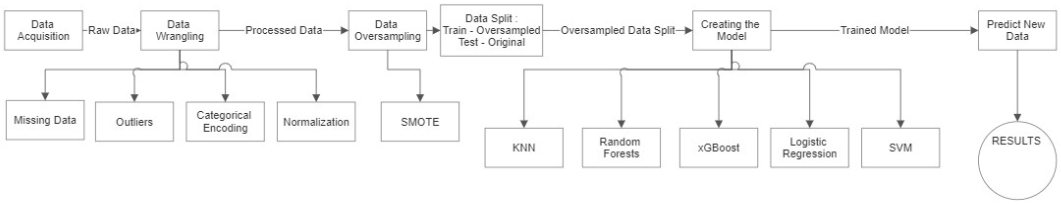
```
0.9206994240791342
```

```
print(classification_report(Y_test_original, knn_preds))
```

```
              precision    recall  f1-score   support

         0.0       0.89      0.96      0.92      2589
         1.0       0.94      0.90      0.92      9014
         2.0       0.95      0.89      0.92     10047
         3.0       0.92      0.93      0.92      5828
         4.0       0.85      0.99      0.92      1217
         5.0       0.90      0.97      0.94      3104
         6.0       0.76      1.00      0.87       242
         7.0       0.87      1.00      0.93       855
         8.0       0.76      0.99      0.86       137
         9.0       0.82      1.00      0.90       215
        10.0       0.81      0.99      0.89       194
```

The Final Design:



Conclusion:

The best algorithm we tried for our dataset was the KNN for getting highest accuracy 92% and performance (least running time). The Random Forest comes next for getting accuracy 89% after oversampling.