Patient Classification

Data Mining & Machine Learning Assignment

Kush Gupta, FHP6QZ - Kaggle user name- Kush Gupta 1121

1.) Introduction

Our task was to create a classifier that is capable of determining which department should admit a patient based on basic, easily accessible data (demographics, standard lab panel) about the patients. The data set contains **43 features** and **3696245 labeled rows**. I've used **Python 3.X** environment on Google Colab. Libraries used in the assignment are pandas, Numpy, sklearn, seaborn, matplotlib, sweetviz.

2.) Analyzing (preprocessing) data & Building Classifiers:

The provided data contained 5 categorical and 38 numerical features, in total 43 features. I performed the EDA on the data set using the Sweetviz python library and could see that there were a lot of missing values for different features and there seem to be outliers also in the dataset. The EDA is submitted in a separate file 'SWEETVIZ_REPORT.html'. The categorical features were then converted to numerical features using label encoders.

I've performed data preprocessing in different ways and tried to analyze and build different classifiers on each kind of preprocessed data.

2.1) Handling Missing values - drop null values from the training set

Initially, we had a huge dataset and a lot of values were missing in the dataset. So, I tried to drop all the null values from the dataset and tried building different classifiers on it. In the test data, the values were filled with mean, median, or mode depending on the individual feature's distribution.

- Sklearn's **Decision tree**: To get started and to understand the features better, I started with a decision tree classifier which splits the data on the feature that results in the largest information gain (IG) and got an accuracy of ~8%.

- **KNN**: I tried K-Nearest neighbors, Instance-based learning and here we do not learn weights from training data to predict output but used entire training instances to predict output for unseen data, Euclidean distance method was used to calculate the distance between the data points and got accuracy of ~ 9%.

2.2) Handling Missing values - replace null values in the training set

Instead of dropping the NA's in the features, I replaced them with their respective mean, subsampled the data, and implemented the below classifiers. Similarly, in the test data, the same set of features was used for predicting, and NA's were filled with their mean values.

- Since the results obtained by the decision tree classifier were not good, I tried fine-tuning the decision tree algorithm by tuning the below parameters and the accuracy increased to ~15%.
    1. Changed splitting criterion to entropy to calculate entropy at every split,
    2. to avoid overfitting of the tree used max_depth,
    3. used log2 of total available features.

```
DecisionTreeClassifier(criterion='Entropy',
                       splitter='best',
                       max_depth=8,
                       max_features='log2',
                       random_state=42)
```

- I tried tuning the **KNN model** to avoid overfitting by changing the below parameters and the accuracy increased to ~13% compared to the previous KNN model:

    1. Number of nearest neighbors: n_neighbors =10
    2. Setting the leaf_size : leaf_size=10,
    3. Setting the P (power parameter) p=2 for Euclidean distance

```
knn = KNeighborsClassifier(leaf_size=10, n_neighbors=10, p=2)
```

- Also, I tried to do a **GridSearchCV** with below parameters, however, due to CPU and RAM constraint, the operation below couldn't be completed.

```
leaf_size = list(range(1,30))
n_neighbors = list(range(1,20))
p=[1,2]
hyperparameters = dict(leaf_size=leaf_size,
  n_neighbors=n_neighbors, p=p)

from sklearn.model_selection import GridSearchCV
knn_clf = GridSearchCV(knn, hyperparameters, cv=3)
```

- I built a **RandomForestClassifier** from sklearn using the data set. Selected some important features after some research about the domain, to understand feature importances and tried random forest on the data. Initially, I chose a smaller value of decision trees (30), and increased it to 100 to see how the model performs. I could see that at n_estimators =100, I got an accuracy of ~ 21 % which was an improvement compared to previous classifiers.

```
rf=RandomForestClassifier(n_estimators=100)
```

2.3) Handling Outliers and Feature selection - recursive feature elimination

After visualizing the data & understanding the important features (I tried to do some research about the feature importance), I could see that there were outliers present in the data. For example, the Glucose level for 25 patients was reported as 1276103.0 which is higher than the normal range (most probably outliers). So, I tried to remove the outliers from the features, and choose important features, subsampled the data, and built classifiers using the data set. In the test data, the same set of features was used for predicting, and NA's were filled with their mean values.

- From the sklearn package I tried **RFECV** recursive feature elimination, the external estimator that assigns weights to features, the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through any specific attribute or callable. Then, the least important features are pruned from the current set of features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

```
model=RFECV(RandomForestClassifier(),scoring='accuracy')
```

2.4) Handling Outliers, Feature selection - scaling and transforming data

Since some of the features in the data set had extreme values and their distribution was skewed, Scikit-learn's robust scaler and transform functions were used to remove outliers and to normalize the later set of features that helped to preserve the shape of the dataset (no distortion). I selected some important features, subsampled the data, and built classifiers on them. Similarly, the test data was prepared.

- Using the scaled data set I tried the **decision tree classifier** which gave an accuracy of ~ 20%.

- From the sklearn package, I used the **support vector machine** module to build an SVC classifier, which had an accuracy of ~ 23%, however, it took around 3 hours to train the model.

```
from sklearn.svm import SVC
svm = SVC(kernel = 'RBF')
```

2.5) Handling Outliers, Feature selection - unsupervised learning to select best features

I did an experiment with sklearn's unsupervised dimensionality reduction methods: PCA and Kbest feature selection method. I used the Kbest feature selection method and selected k (20) best features where the chi score was maximum. A chi-square test is used in statistics to test the independence of two events. Selecting independent events will help us reduce redundancy in the features. For Predicting, top 20 features were used.

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

bestfeatures = SelectKBest(score_func=chi2, k=20)
```

List of top 20 Features and their score:-

Feature column	Feature	Score
6	Age	62771.593591
2	Marital_status	42388.443788
0	Condition_importance	36619.416490
5	Gender	34368.699405
13	Bilirubin, Total	12880.928351
1	Admission_type	12408.976336
4	Hospital_death_flag	7986.227106
38	Urea Nitrogen	4174.579782
22	I	3857.781669
20	Hematocrit	3652.228460
21	Hemoglobin	2742.920769
3	Ethnicity	2653.120020
17	Creatinine	2152.891347
42	pO2	987.732304
36	Red Blood Cells	971.544554
27	MCV	520.586059
19	H	510.344249
12	Bicarbonate	471.845684
30	PTT	471.759365
34	RDW	370.820888

Using the top 20 features and subsampling the scaled data, I implemented below classifiers.

- I tried **RandomForestClassifier** from sklearn using the data set. I could see that at n_estimators =150, I got an accuracy of ~ 24 %, which was an improvement compared to previous random forest classifiers.

```
RandomForestClassifier(n_estimators=150,criterion='entropy')
```

- I implemented a gradient **boosting classifier** that combines many weak learning models together to create a strong predictive model. It minimizes a loss function by iteratively choosing a function that points towards the negative gradient. This classifier gave an accuracy of ~ 26%.

```
from sklearn.ensemble import GradientBoostingClassifier
ensemble=GradientBoostingClassifier(n_estimators=210,max_depth=1,
learning_rate = 0.1,random_state=123).fit(X_train, y_train)
score = ensemble.score(X_test, y_test)
```

- I tried the **XGBoost** classifier as it is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. It combines a set of weak learners and delivers improved prediction accuracy. At any instant t, the model outcomes are weighed based on the outcomes of the previous instant t-1. The outcomes predicted correctly are given a lower weight and the ones miss-classified are weighted higher. The XGBoost algorithm takes much time to train the model, and I got an accuracy of ~23.8%. The classifier got overfitted which reduced the accuracy. I tried tuning the classifier and to reduce the overfitting by using the early_stopping_rounds=10.

```
from xgboost import XGBClassifier
model = XGBClassifier(n_estimators = 400, learning_rate = 0.1)
```

- Since XGboost takes much time to train the classifier, I tried using **Light GBM**, which is a fast, distributed, high-performance gradient boosting framework based on a decision tree algorithm. It is designed to be distributed and efficient to have Faster training speed. I got an accuracy of ~25%, which was an improvement from the XGBoost classifier.

```
import lightgbm as lgb
clf = lgb.LGBMClassifier()
```

2.6) Handling Missing values based on gender

Generally, the features given in the dataset have values in different ranges based on gender. For example, generally of Pco2 lies in the range of 12.1-15.0 gm/dL for females, and for males it's 13.5-17.0 gm/dL. Hence, I tried to divide the data set into 2 parts, and then based on the patient's gender I tried to fill the NA's with their respective mean values of the features. I did the same steps to handle NA's in test data. Once the data preprocessing was done, I tried the below classifiers on them:

- I tried tuning LGBM classifier with the below parameters:-
  - objective = Multi-class classification
  - num_leaves=specifying the number of max leaves allowed per tree.
  - metric= metrics to be evaluated on evaluation datasets
  - learing_rate= defines how fast the model learns, uses the small value to minimize the loss function.
  - feature_fraction= Fraction of total features used in training.
  - L1,L2= regularization term on weights.
  - Min_child_samples= is the minimum number of samples (data) to group into a leaf.
  - Boosting type= gradient-based decision trees.

```
import lightgbm as lgb
clf = lgb.LGBMClassifier(
      objective= 'multiclass',
      metric=  ['multi_logloss'],#'auc',
      verbosity= -1,
      boosting_type= 'gbdt', #gbrt
      num_iterations=210,
      learning_rate=0.5,
      lambda_l1= 0.0005523588106120283,
      lambda_l2= 6.72929973145413e-07,
      num_leaves= 6,
      feature_fraction= 0.8,
      bagging_fraction= 0.9,
      bagging_freq= 1,
      min_child_samples= 20
)
```

- I tried tuning XGBoostclassifier using the below parameters.
  - ➔ n_estimator= Number of trees
  - ➔ learning_rate=defines how fast the model learns
  - ➔ subsample= XGBoost would sample the training data prior to growing trees. and this will prevent overfitting.
  - ➔ max_depth= Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit.
  - ➔ Sampling_method= the selection probability for each training instance is proportional to the *regularized absolute value* of gradients.
  - ➔ process_type=update: Starts from an existing model and only updates its trees. In each boosting iteration, a tree from the initial model is taken, a specified sequence of updaters is run for that tree, and a modified tree is added to the new model.
  - ➔ reg_alpha=L1 regularization term on weights. Increasing this value will make the model more conservative.
  - ➔ gamma=Minimum loss reduction required to make a further partition on a leaf node of the tree.

```
#Model type 1
model=XGBClassifier(
      n_estimators=250,
      learning_rate=0.1,
      subsample=0.8,
      max_depth=5,
      sampling_method='gradient_based',
      process_type='update'
)

#Model type2
model = XGBClassifier(
      silent=False,
      scale_pos_weight=1,
      learning_rate=0.1,
      colsample_bytree = 0.4,
      subsample = 0.8,
      objective='multiclass',#'multi:softprob',
      n_estimators=400,
      reg_alpha = 0.3,
      max_depth=3,
      gamma=10
)
```

- I tried Catboost, which is a boosted decision tree machine learning algorithm developed by Yandex. CatBoost converts categorical values into numbers using various statistics on combinations of categorical features and combinations of categorical and numerical features. I got an accuracy of ~25%.

```
from catboost import CatBoostClassifier
model = CatBoostClassifier(
   iterations=1000,
   random_seed=143,
   #learning_rate=0.01,
   #eval_metric='AUC',
   early_stopping_rounds=10,
  loss_function='MultiClass'
)
model.fit(
   X_train, y_train,
   #cat_features=cat_features,  (X_train, y_train)
   eval_set=[(X_train, y_train),(X_test, y_test)],
   use_best_model = True, #(x_validation, y_validation),
   verbose=False #,plot=True
)
```

Note: Unlisted parameters are the scikit defaults. See their documentation for details.

3.) Hyper-parameter optimization

3.1) Importance of Hyper-parameter optimization

Hyperparameters are crucial as they control the overall behaviour of a machine learning model. The ultimate goal is to find an optimal combination of hyperparameters that minimizes a predefined loss function to give better results. Failure to do so would give sub-optimal results as the model didn't converge and minimize the loss function effectively.

3.2) Implementing Hyper-parameter optimization

I was unable to perform the below hyper parameter tuning operations, due to Memory and CPU constraints, I tried reducing the numbers of parameters but due to memory issues the operation terminated after running for 2 hours due to which I'm unable to find the optimal parameters for LGBM and XGBM (the codes are available in the Annexure).

4.) Conclusion

A number of different classification methods were examined. Some of them didn't perform very well initially, probably due to no/minimum data preprocessing and skewness in data, but once the data preprocessing has been performed, most of them performed relatively well.

I tried hyper-parameter optimization and grid searchCV to optimize parameters in the case of LGBM, XGB however, due to memory and CPU constraints, the operation couldn't be completed. Higher scores require some domain knowledge during feature selection and handling null values so that important features are selected and missing values are handled in a better way, because outliers in the data set may impact mean values of the features, and replacing nulls with mean might not be the best approach.

Also, the availability of GPU would have helped in hyper-parameter optimization, to achieve better results. This was a challenging task as the data set was huge and contained a lot of missing values. According to me, EDA operation on the dataset and Hyper-parameter tuning was the most challenging part of the assignment, but I've learned a lot of things while completing the assignment.

4.1) A Brief Summary of the accuracy obtained on the Public/Private Data set (available on Kaggle).

Type of Classifiers	Public Score on Kaggle	Private Score on Kaggle
Gradient_Boosting	0.26028	0.2596
LGBM_Classifier	0.25771	0.25864
Modified_LGBM_classifier	0.25046	0.25102
Catboost_classifier	0.25521	0.25576
Random_forest_classifier	0.24248	0.24254
XGB_Classifier	0.23741	0.23834

Annexure

1. Hyper parameter optimization for LGBM.

```
import optuna
import lightgbm as lgbm
from sklearn.metrics import log_loss
from sklearn.model_selection import StratifiedKFold
from optuna.integration import LightGBMPruningCallback

def objective(trial, X, y):
    param_grid = {
         #"device_type": trial.suggest_categorical("device_type", ['gpu']),
        "n_estimators": trial.suggest_categorical("n_estimators", [400]),
        "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3),
        "num_leaves": trial.suggest_int("num_leaves", 20, 1000, step=20),
        "max_depth": trial.suggest_int("max_depth", 3, 10),
        "min_data_in_leaf": trial.suggest_int("min_data_in_leaf", 200, 1000, step=100),
        "lambda_l1": trial.suggest_int("lambda_l1", 0, 100, step=5),
        "lambda_l2": trial.suggest_int("lambda_l2", 0, 100, step=5),
        "min_gain_to_split": trial.suggest_float("min_gain_to_split", 0, 15),
        "bagging_fraction": trial.suggest_float("bagging_fraction", 0.2, 0.95, step=0.1),
        "bagging_freq": trial.suggest_categorical("bagging_freq", [1]),
        "feature_fraction": trial.suggest_float("feature_fraction", 0.2, 0.95, step=0.1),
    }

    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1121218)
    cv_scores = np.empty(5)
    for idx, (train_idx, test_idx) in enumerate(cv.split(X, y)):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]#y[train_idx], y[test_idx]

        model = lgbm.LGBMClassifier(objective="multiclass", **param_grid)
        model.fit(
            X_train,
            y_train,
            eval_set=[(X_test, y_test)],
            eval_metric="multi_logloss",
            early_stopping_rounds=100,
            callbacks=[
                LightGBMPruningCallback(trial, "multi_logloss")
            ]  # Add a pruning callback
        )
        preds = model.predict_proba(X_test)
        cv_scores[idx] = log_loss(y_test, preds)

    return np.mean(cv_scores)

study = optuna.create_study(direction="minimize", study_name="LGBM Classifier")
func = lambda trial: objective(trial, training_data, train_y)
study.optimize(func, n_trials=10)
```

2. Hyper parameter optimization for XGB Classifier:

```
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
estimator = XGBClassifier(
    objective= 'multiclass',
    nthread=4,
    seed=42
)

parameters = {
    'max_depth': range (2, 10, 1),
    'n_estimators': range(60, 400, 40),
    'learning_rate': [0.1, 0.01, 0.05],
    'reg_alpha' : [0.1,0.2,0.3,0.4],
    'colsample_bytree' : [0.4,0.5,0.6,0.7],
    'subsample' :  [0.5,0.6,0.7,0.8,0.9],
    'max_depth': [3,4,5,6,7,8,9]
    'gamma': range (0,10,1)
}

grid_search = GridSearchCV(
    estimator=estimator,
    param_grid=parameters,
    scoring = 'mlogloss',
    n_jobs = 10,
    cv = 10,
    verbose=True
)
grid_search.fit(X_train, y_train)
```