Development of an open-access and explainable machine learning prediction system to assess the mortality and recurrence risk factors of *Clostridioides difficile* infection patients: Model Training and Hyperparameter Optimization with Cross-Validation

Yui-Lun Ng[1], Chi-Kiu Lo[1], Kit-Hang Lee[1], Xiaochen Xie[1], Thomas N.Y. Kwong[2], Margaret Ip[3], Lin Zhang[2], Jun Yu[2], Joseph J.Y. Sung[2], William K.K. Wu[2,4], Sunny H. Wong[2], and Ka-Wai Kwok[1]

[1]Department of Mechanical Engineering, Faculty of Engineering, The University of Hong Kong, Hong Kong
[2]SKL Laboratory of Digestive Disease, Li Ka Shing Institute of Health Sciences, Department of Medicine and Therapeutics, Faculty of Medicine, The Chinese University of Hong Kong, Hong Kong
[3]Department of Microbiology, Faculty of Medicine, The Chinese University of Hong Kong, Hong Kong
[4]Department of Anaesthesia and Intensive Care, Faculty of Medicine, The Chinese University of Hong Kong, Hong Kong

October 22, 2020

Abstract

Identifying *Clostridioides difficile* infection (CDI) patients at risk of mortality or recurrence will facilitate prevention, timely treatment and improve clinical outcomes. We aim to establish an open-access web-based prediction system, which estimates CDI patients' mortality and recurrence outcomes, and explains the machine learning prediction with patients' characteristics. Prognostic models were developed using four various types of machine learning algorithms and statistical logistics regression model utilizing over 15,000 CDI patients from 41 hospitals in Hong Kong. The boosting-based machine learning algorithm Gradient Boosting Machine (Mortality AUC: 0.7878; Recurrence AUC: 0.7076) outperformed statistical models (Mortality AUC: 0.7573; Recurrence AUC: 0.6927) and other machine learning algorithms. The open-access prediction system for clinicians to assess and interpret the risk factors of CDI patients is now available at https://www.cdiml.care/. In this article, we explain the development of machine learning models and illustrate how to apply hyperparameter tuning with cross-validation to optimize the model accuracy.

Step 1: Train Test Split

In machine learning, we always split our data into the training dataset and test dataset. The purpose of splitting into distinct datasets is to evaluate the performance of machine learning models on unseen data. A

common approach to split the train and test dataset by certain percentages, for example *(80% train, 20% test)* or *(75% train, 25% test)*. In our model training framework, as we obtained electronic health records of over 15,000 Clostridioides *difficile* infection (CDI) episodes using the Clinical Data Analysis and Reporting System (CDARS), a well-established electronic database managed by the Hong Kong Hospital Authority, patient records of 41 Hong Kong public hospitals are available in our study. In our model training framework, we reserved several hospital institutions as an external validation set to evaluate the test performance of the fit machine learning models.

- Train Dataset: Fit the machine learning model
- Test Dataset: Evaluate the machine learning model

```
import pandas as pd

TARGET_LIST = ['Mortality','Recurrence']
TARGET = 'Mortality'
# Load dataset as pandas dataframe
df_data = pd.read_excel("./data.xlsx")

# Identify training fields and test fields
train_fields = [h for h in df_data.columns if h != TARGET_LIST]
test_fields = [h for h in df_data.columns if h == TARGET]
fields = train_fields + test_fields

# Split data into training set and test set according to hospital institutions
TEST_HOSPITAL_LIST = ['FYK','GH','HHH','KH','KWH','PWH','PYH','RH']
df_train = df_data.loc[df_data['Hospital'].isin(TEST_HOSPITAL_LIST)==False]
df_test = df_data.loc[df_data['Hospital'].isin(TEST_HOSPITAL_LIST)]
df_X_train = df_train[train_fields].copy().drop(columns='Hospital')
df_y_train = df_train[test_fields].copy()
df_X_test = df_test[train_fields].copy().drop(columns='Hospital')
df_y_test = df_test[test_fields].copy()
```

Step 2: One-hot encoding for categorical variables

Categorical variables contain distinct data values to represent different groups. Some machine learning algorithms, for example, decision trees, are designed to handle categorical data structure, however, a majority of machine learning algorithms, including support vector machine (SVM), neural network (NN) only process numeric data. Therefore, the categorical variables require conversion before our data training process. The *one-hot encoding* can transform the categorical variables into binary columns, which represent one category in one column. In python pandas package, the one-hot encoding can be attained using the function "get_dummies".

```
# Identify numeric features and categorical features
numerical_features = df_X_train.dtypes == 'float'
categorical_features = ~numerical_features

numerical_features_name = numerical_features.loc[numerical_features==True].index.values

for k,v in zip(categorical_features.index,categorical_features.values):
    if v == True:
```

```
        df_X_train[k] = df_X_train[k].apply(lambda x:x if pd.isnull(x) else str(x))
        df_X_test[k] = df_X_test[k].apply(lambda x:x if pd.isnull(x) else str(x))

# One-hot encoding for categorical variables
df_X_train_1 = pd.get_dummies(df_X_train,dummy_na=True)
df_X_test_1 = pd.get_dummies(df_X_test,dummy_na=True)

# Retrieve numeric columns and categorical columns
numerical_features = df_X_train_1.dtypes == 'float'
categorical_features = ~numerical_features

# Fill missing one-hot-encoded columns in the test dataset
missing_cols = set( df_X_train_1.columns ) - set( df_X_test_1.columns )
for c in missing_cols:
    df_X_test_1[c] = 0
df_X_test_1 = df_X_test_1[df_X_train_1.columns]


fields = df_X_train_1.columns.values
train_fields_dummies = [h for h in fields if h not in test_fields]
```

Step 3: Numeric data pre-processing

Numeric features often exist in a variety of data scale. Standardization of feature values not only can prevent the objective function of algorithms hindered by features with a board range of values, but also facilitate the algorithms using gradient descent as an optimizer as the converging speed is much faster. Features with missing values are first filled with the mean values before standardization.

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

si = SimpleImputer(strategy='mean', fill_value='MISSING')
sc = StandardScaler()

df_X_train_2 = df_X_train_1.copy()
df_X_test_2 = df_X_test_1.copy()

df_X_train_2[numerical_features_name] = si.fit_transform(df_X_train_2[numerical_features_name])
df_X_train_2[numerical_features_name] = sc.fit_transform(df_X_train_2[numerical_features_name])

df_X_test_2[numerical_features_name] = si.transform(df_X_test_2[numerical_features_name])
df_X_test_2[numerical_features_name] = sc.transform(df_X_test_2[numerical_features_name])
```

Step 4: Model Training and Hyperparameter Optimization with Cross-Validation

After all the data preprocessing steps, the goal here is to find out which algorithms can attain the highest test accuracy. In model training of logistic regression, random forest, and gradient boosting machine, we need to first identify the best set of hyperparameters for each algorithm. The best hyperparameter set can be found using either grid search or randomized grid search. The *grid search* method enumerates all the combinations based on the hyperparameter space and repeat the model training process for each combination. *Randomized grid search* is a more efficiency way as a fixed number of iterations is specified to sample the parameters. A total of 1,000 iterations had been carried out to determine the best hyperparameter set. *K-fold cross validation* is performed to validate whether the selected hyperparameter set can generalize well on unseen data. In K-fold cross validation, the data is equally divided into k subsets. The models are trained on k-1 subsets and validated on the remaining subsets. The training process is repeated k-times until all subsets had been used as validation once. All 5 algorithms have been repeatedly trained for 50 times to develop the best models with best set of hyperparameters.

The 5 algorithms include:

- Logistic Regression
- Random Forest
- Gradient Boosting Machine
- Support Vector Machine
- Neural Network

```
import datetime
import pickle
import sys
import os
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from lightgbm import LGBMClassifier
from keras.models import Sequential
from keras.layers import Dense
from sklearn.metrics import roc_auc_score


X_train = df_X_train_2.copy()
y_train = df_y_train.copy().astype('int')
X_test = df_X_test_2.copy()
y_test = df_y_test.copy().astype('int')


time_st = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')

target_data = {}
target_data['X_train'] = X_train
target_data['y_train'] = y_train
target_data['X_test'] = X_test
target_data['y_test'] = y_test
```

```
def train(objective,model_name,data_dict):
    ref_time = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
    path = '.'
    n_iter_search = 1000
    X_train, y_train, X_test, y_test = data_dict['X_train'], data_dict['y_train'], data_dict['X_test'],
    n_cols = X_train.shape[1]

    print(objective,model_name)

    if model_name == 'LR':
        param_dist = {
                       'C': [0.001, 0.01,0.1, 1.0],
                       'penalty': ['l1', 'l2']
                     }
        clf = LogisticRegression()
        random_search = RandomizedSearchCV(clf,
                                           param_distributions=param_dist,
                                           n_iter=n_iter_search,
                                           cv=5,
                                           scoring='roc_auc',
                                           n_jobs=-1,
                                           verbose=1,
                                           )
        random_search.fit(X_train, y_train.values.ravel())
        clf_best = random_search.best_estimator_
        clf_best.fit(X_train, y_train.values.ravel())
        y_predict_proba = clf_best.predict_proba(X_test)[:,1]

    if model_name == 'RF':
        param_dist = {
                    'n_estimators': range(80,200,4),
                    'max_depth': range(2,15,1),
                    'min_samples_split': np.linspace(0.1, 1.0, 10, endpoint=True),
                    'min_samples_leaf': np.linspace(0.1, 0.5, 5, endpoint=True),
                     }
        clf = RandomForestClassifier()
        random_search = RandomizedSearchCV(clf,
                                           param_distributions=param_dist,
                                           n_iter=n_iter_search,
                                           cv=5,
                                           scoring='roc_auc',
                                           n_jobs=-1,
                                           verbose=1,
                                           )
        random_search.fit(X_train, y_train.values.ravel())
        clf_best = random_search.best_estimator_
        clf_best.fit(X_train, y_train.values.ravel())
        y_predict_proba = clf_best.predict_proba(X_test)[:,1]

    if model_name == 'GBM':
        param_dist = {
```

```
                "learning_rate": [0.001, 0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2],
                "max_depth":range(2,63,1),
                "subsample":[0.5, 0.6, 0.7, 0.8, 0.85, 0.9, 0.95, 1.0],
                "n_estimators":range(10,210,10),
                "num_leaves":range(10,210,10), 12).astype('float32'),
                }
    clf = LGBMClassifier()
    random_search = RandomizedSearchCV(clf,
                                       param_distributions=param_dist,
                                       n_iter=n_iter_search,
                                       cv=5,
                                       scoring='roc_auc',
                                       n_jobs=-1,
                                       verbose=1,
                                       )
    random_search.fit(X_train, y_train.values.ravel())
    clf_best = random_search.best_estimator_
    clf_best.fit(X_train, y_train.values.ravel())
    y_predict_proba = clf_best.predict_proba(X_test)[:,1]

if model_name == 'SVM':
    clf = SVC(probability=True)
    clf_best = clf
    clf_best.fit(X_train, y_train.values.ravel())
    y_predict_proba = clf_best.predict_proba(X_test)[:,1]

if model_name == 'NN':
    clf = Sequential()
    clf.add(Dense(100, activation='relu', input_shape=(n_cols,)))
    clf.add(Dense(100, activation='relu'))
    clf.add(Dense(1, activation='sigmoid'))
    clf.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    clf.fit(X_train, y_train.values.ravel(), epochs=10, batch_size=10)
    clf_best = clf
    y_predict_proba = clf_best.predict_proba(X_test)[:,0]

path_gs = './gs_result_'+time_st+'/'+objective+'/'+model_name
os.makedirs(path_gs,exist_ok=True)
original = sys.stdout
sys.stdout = open(path_gs+'/'+'gs_'+ref_time+'.txt', "w")
try:
    print(report(random_search.cv_results_))
    print('Test Accuracy:')
    print(roc_auc_score(y_test, y_predict_proba))
except:
    print(roc_auc_score(y_test, y_predict_proba))

sys.stdout = original

try:
    pickle.dump(clf_best, open(path_gs+'/'+'model_'+ref_time+'.pkl', 'wb'))
```

```
    except:
        clf_best.save(path_gs+'/'+'model_'+ref_time+'/')

    with open(path_gs+'/'+'y_predict_proba_'+ref_time+'.txt', "w") as f:
        for h in y_predict_proba:
            f.write(str(h)+'\n')

for _ in range(50):
    for obj in ['D']:
        for model in ['SVM','GBM','LR','RF','NN']:
            train(obj,model,target_data)
```

Hyperparameter Optimization with Cross-validation results

The tuning result of the ***gradient boosting machine*** model is shown below. It can be observed that the ***mean validation score (0.811)*** is similar to the ***test accuracy (0.7878)***, this implies that the model is neither overfitting nor underfitting the training data. The *optimal hyperparameter set* can be obtained via this method.

```
Model with rank: 1
Mean validation score: 0.811 (std: 0.008)
Parameters: {'subsample': 1.0, 'num_leaves': 180, 'n_estimators': 190, 'max_depth': 2, 'learning_rate':

Model with rank: 2
Mean validation score: 0.811 (std: 0.008)
Parameters: {'subsample': 0.85, 'num_leaves': 70, 'n_estimators': 180, 'max_depth': 2, 'learning_rate':

Model with rank: 3
Mean validation score: 0.810 (std: 0.009)
Parameters: {'subsample': 0.95, 'num_leaves': 180, 'n_estimators': 160, 'max_depth': 2, 'learning_rate'

Model with rank: 3
Mean validation score: 0.810 (std: 0.009)
Parameters: {'subsample': 0.8, 'num_leaves': 40, 'n_estimators': 160, 'max_depth': 2, 'learning_rate': 0

None
Test Accuracy:
0.7878260598244373
```