Advanced Machine Learning

CIS550

Spring ‘24

Lab Homework 7

Submitted by :

Name : Manas Vishal (01971464)

Email : mvishal@umassd.edu

Title : Hyperparametric tuning of an ML model

Deploying a model

This lab is continuation of previous labs (4,5 and 6) where we trained, deployed and evaluated a model. In the previous labs, we used the XGBoost library to train a model on the biomedical dataset and deployed it to the dataset but we also found that the model was not that good for some data. In the lab 6 we also evaluated the model and its efficiency by different metrics. However, in this lab we will be tuning the model by using a hyperparameter. After that we will be comparing the metrics of two models.

About the data

This biomedical dataset was built by Dr. Henrique da Mota during a medical residence period in the Group of Applied Research in Orthopaedics (GARO) of the Centre Médico-Chirurgical de Réadaptation des Massues, Lyon, France. The data has been organized in two different, but related, classification tasks.

The first task consists in classifying patients as belonging to one of three categories:

- *Normal* (100 patients)
- *Disk Hernia* (60 patients)
- *Spondylolisthesis* (150 patients)

For the second task, the categories *Disk Hernia* and *Spondylolisthesis* were merged into a single category that is labeled as *abnormal*. Thus, the second task consists in classifying patients as belonging to one of two categories: *Normal* (100 patients) or *Abnormal* (210 patients).

Each patient is represented in the dataset by six biomechanical attributes that are derived from the shape and orientation of the pelvis and lumbar spine (in this order):

- Pelvic incidence
- Pelvic tilt
- Lumbar lordosis angle
- Sacral slope
- Pelvic radius
- Grade of spondylolisthesis

The following convention is used for the class labels:

- DH (Disk Hernia)
- Spondylolisthesis (SL)
- Normal (NO)
- Abnormal (AB)

For more information about this dataset, see the [Vertebral Column dataset webpage](http://archive.ics.uci.edu/ml/datasets/Vertebral+Column).

Loading the data and evaluating the XGBoost model

We follow the same steps as in the previous labs to load and validate the data. We also evaluate the model.

```
time
zip = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00212/vertebral_column_data.zip'
= requests.get(f_zip, stream=True)
rtebral_zip = zipfile.ZipFile(io.BytesIO(r.content))
rtebral_zip.extractall()
ta = arff.loadarff('column_2C_weka.arff')
 = pd.DataFrame(data[0])
ass_mapper = {b'Abnormal':1,b'Normal':0}
['class']=df['class'].replace(class_mapper)
ls = df.columns.tolist()
ls = cols[-1:] + cols[:-1]
 = df[cols]
ain, test_and_validate = train_test_split(df, test_size=0.2, random_state=42, stratify=df['class'])
st, validate = train_test_split(test_and_validate, test_size=0.5, random_state=42, stratify=test_and_validate['class']
del = XGBClassifier(objective='binary:logistic', eval_metric='auc', num_round=42)
int(model.fit(train.drop(['class'], axis = 1).values, train['class'].values))
int("Training Completed")
```

[3] ✓ 0.4s Python

```
[13:12:35] WARNING: /croot/xgboost-split_1675457761144/work/src/learner.cc:767:
Parameters: { "num_round" } are not used.

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='auc', feature_types=None,
              gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              num_round=42, predictor=None, ...)
Training Completed
CPU times: user 5.31 s, sys: 335 ms, total: 5.64 s
Wall time: 410 ms
```

Fig. 1 Loading the data and evaluating the xGboost model

We also define multiple functions to make plotting the confusion matrices and ROC curve easier.

```
def plot_confusion_matrix(test_labels, target_predicted):
    matrix = confusion_matrix(test_labels, target_predicted)
    df_confusion = pd.DataFrame(matrix)
    colormap = sns.color_palette("BrBG", 10)
    sns.heatmap(df_confusion, annot=True, cbar=None, cmap=colormap)
    plt.title("Confusion Matrix")
    plt.tight_layout()
    plt.ylabel("True Class")
    plt.xlabel("Predicted Class")
    plt.show()
```

Fig. 2 Function for plotting confusion matrix

```python
def plot_roc(test_labels, target_predicted_binary):
    TN, FP, FN, TP = confusion_matrix(test_labels, target_predicted_binary).ravel()
    # Sensitivity, hit rate, recall, or true positive rate
    Sensitivity  = float(TP)/(TP+FN)*100
    # Specificity or true negative rate
    Specificity  = float(TN)/(TN+FP)*100
    # Precision or positive predictive value
    Precision = float(TP)/(TP+FP)*100
    # Negative predictive value
    NPV = float(TN)/(TN+FN)*100
    # Fall out or false positive rate
    FPR = float(FP)/(FP+TN)*100
    # False negative rate
    FNR = float(FN)/(TP+FN)*100
    # False discovery rate
    FDR = float(FP)/(TP+FP)*100
    # Overall accuracy
    ACC = float(TP+TN)/(TP+FP+FN+TN)*100


    print(f"Sensitivity or TPR: {Sensitivity}%")
    print(f"Specificity or TNR: {Specificity}%")
    print(f"Precision: {Precision}%")
    print(f"Negative Predictive Value: {NPV}%")
    print( f"False Positive Rate: {FPR}%")
    print(f"False Negative Rate: {FNR}%")
    print(f"False Discovery Rate: {FDR}%" )
    print(f"Accuracy: {ACC}%")

    test_labels = test.iloc[:,0];
    print("Validation AUC", roc_auc_score(test_labels, target_predicted_binary) )

    fpr, tpr, thresholds = roc_curve(test_labels, target_predicted_binary)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % (roc_auc))
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic')
    plt.legend(loc="lower right")

    ax2 = plt.gca().twinx()
    # ax2.plot(fpr, thresholds, markeredgecolor='r', linestyle='dashed', color='r')
    # ax2.set_ylabel('Threshold', color='r')
    valid_thresholds = thresholds[np.logical_and(~np.isnan(thresholds), ~np.isinf(thresholds))]
    ax2.set_ylim([valid_thresholds[-1], valid_thresholds[0]]) if valid_thresholds.size > 0 else None

    ax2.set_xlim([fpr[0], fpr[-1]])

    plt.show()
```

Fig. 3 Function for plotting the ROC curve

We familiarize ourself with the current model's metrics/

```
batch_X = test.iloc[:,1:];
predicted_probabilities = model.predict_proba(batch_X)
target_predicted = pd.DataFrame(predicted_probabilities[:, 1], columns=['class'])
def binary_convert(x):
    threshold = 0.5
    if x > threshold:
        return 1
    else:
        return 0

target_predicted_binary = target_predicted['class'].apply(binary_convert)

print(target_predicted_binary.head(5))
test_labels = test.iloc[:,0]
```

[5] ✓ 0.0s Python

```
0    1
1    1
2    1
3    1
4    1
Name: class, dtype: int64
```

Fig. 4 Evaluating the model for threshold 0.5

```
plot_confusion_matrix(test_labels, target_predicted_binary)
```

[6] ✓ 0.1s Python

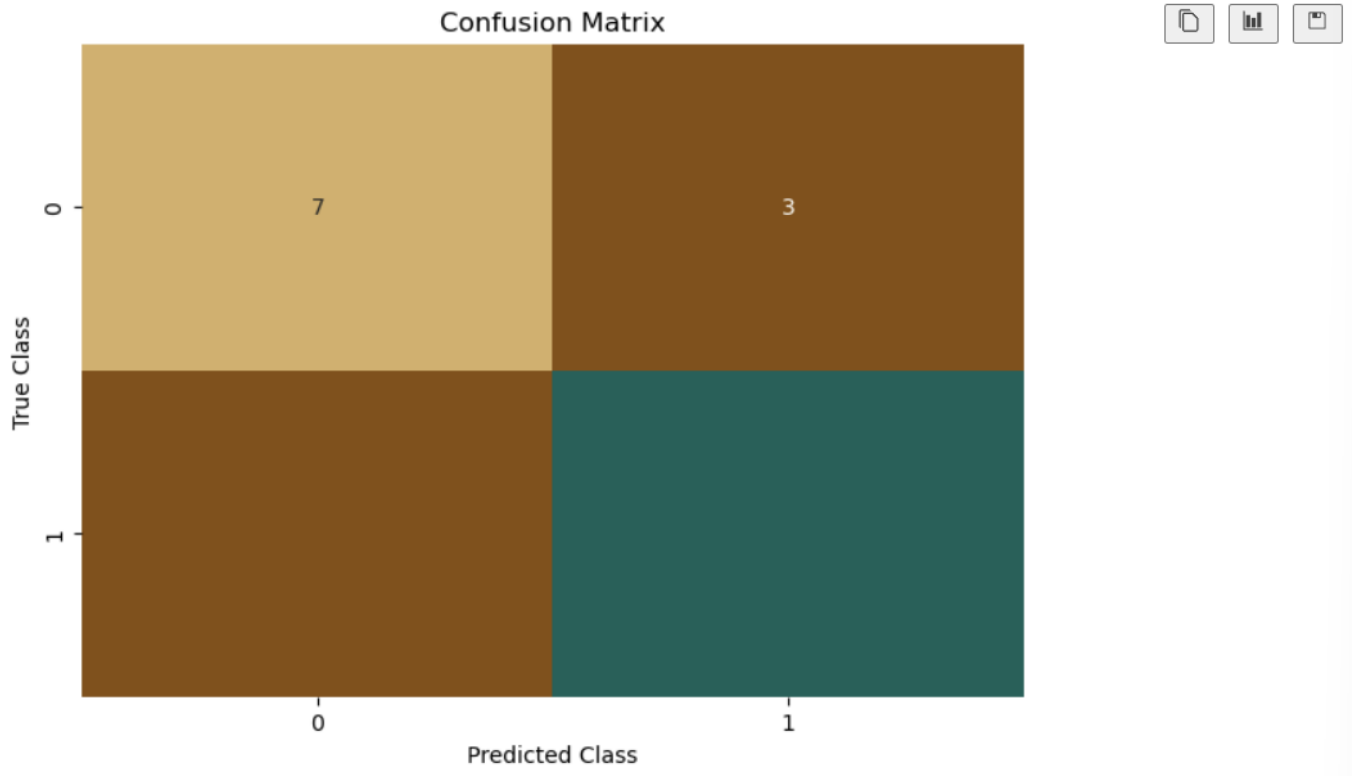

Fig. 5 Plotting the confusion matrix from our new function

```
1 plot_roc(test_labels, target_predicted_binary)
```

[7] ✓ 0.2s Python

```
Sensitivity or TPR: 90.47619047619048%
Specificity or TNR: 70.0%
Precision: 86.36363636363636%
Negative Predictive Value: 77.77777777777779%
False Positive Rate: 30.0%
False Negative Rate: 9.523809523809524%
False Discovery Rate: 13.636363636363635%
Accuracy: 83.87096774193549%
Validation AUC 0.8023809523809523
```

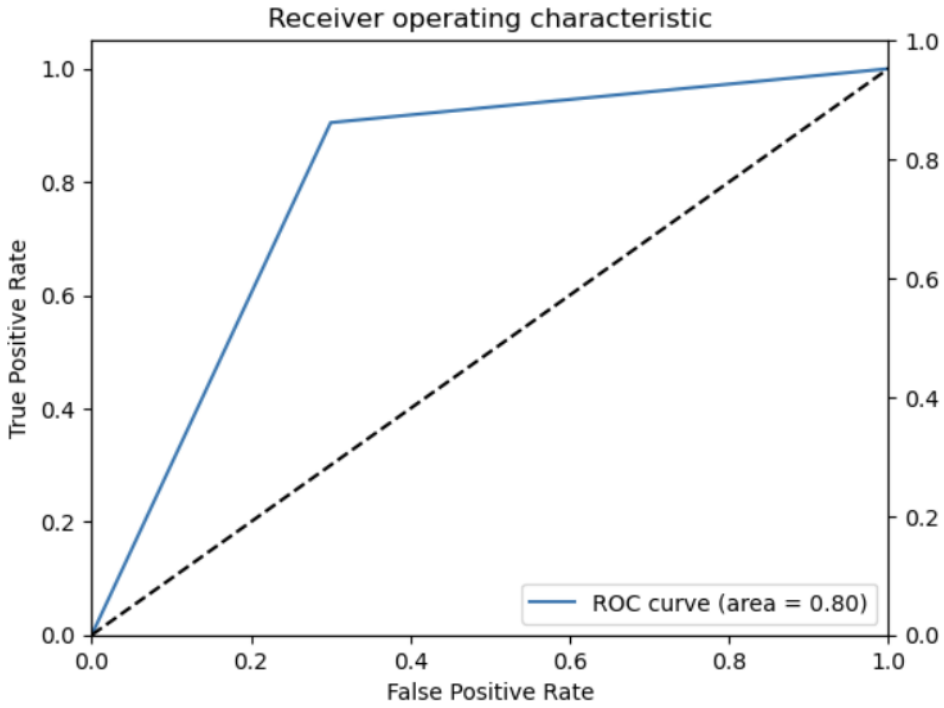

Fig. 6 Plotting the ROC curve from our new function

```
%%time
from sklearn.model_selection import RandomizedSearchCV
# Define your XGBoost model
xgb_model = xgb.XGBClassifier(eval_metric='error@.40',
                              objective='binary:logistic')

# Define hyperparameter ranges
hyperparameter_ranges = {'alpha': [i for i in range(0, 101)],
                         'min_child_weight': [i for i in range(1, 6)],
                         'subsample': [i / 10 for i in range(5, 11)],
                         'eta': [i / 10 for i in range(1, 4)]}

search = RandomizedSearchCV(estimator=xgb_model,
                            param_distributions=hyperparameter_ranges,
                            scoring='neg_mean_squared_error',
                            n_iter=10,  # Number of parameter settings that are sampled
                            cv=5,       # Number of folds for cross-validation
                            verbose=1,
                            n_jobs=1)  # Use all available cores

search.fit(train.drop(['class'], axis = 1).values, train['class'].values)
best_params = search.best_params_
```

[8] ✓ 3.0s Python

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
CPU times: user 44 s, sys: 1.31 s, total: 45.3 s
Wall time: 3.07 s
```

Fig. 6 Creating a hyperparameter tuning job

```
best_params
```

[14] ✓ 0.0s Python

```
{'subsample': 0.8, 'min_child_weight': 2, 'eta': 0.2, 'alpha': 10}
```

Fig. 7 Best parameter after tuning

```python
# Retrieve the results of RandomizedSearchCV
cv_results = pd.DataFrame(search.cv_results_)

# Sort the results by mean_test_score (or other relevant metric)
cv_results.sort_values(by='mean_test_score', ascending=False, inplace=True)

# Display the top 20 models
top_models = cv_results.head(20)
print(top_models[['params', 'mean_test_score']])

# Get the best hyperparameters
best_params = search.best_params_
print("Best Hyperparameters:", best_params)

# Create an XGBoost model with the best hyperparameters
best_xgb_model = XGBClassifier(objective='binary:logistic', eval_metric='error', **best_params)

# Train the model with the best hyperparameters
best_xgb_model.fit(train.drop(['class'], axis = 1).values, train['class'].values)
```

[10] ✓ 0.3s Python

```
                                              params  mean_test_score
8  {'subsample': 0.8, 'min_child_weight': 2, 'eta...        -0.169306
4  {'subsample': 1.0, 'min_child_weight': 2, 'eta...        -0.237959
1  {'subsample': 0.9, 'min_child_weight': 1, 'eta...        -0.245796
9  {'subsample': 0.8, 'min_child_weight': 2, 'eta...        -0.257959
0  {'subsample': 1.0, 'min_child_weight': 4, 'eta...        -0.677388
2  {'subsample': 0.8, 'min_child_weight': 1, 'eta...        -0.677388
3  {'subsample': 0.9, 'min_child_weight': 3, 'eta...        -0.677388
5  {'subsample': 0.7, 'min_child_weight': 2, 'eta...        -0.677388
6  {'subsample': 0.8, 'min_child_weight': 1, 'eta...        -0.677388
7  {'subsample': 0.5, 'min_child_weight': 4, 'eta...        -0.677388
Best Hyperparameters: {'subsample': 0.8, 'min_child_weight': 2, 'eta': 0.2, 'alpha': 10}
```

```
XGBClassifier(alpha=10, base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eta=0.2, eval_metric='error',
              feature_types=None, gamma=None, gpu_id=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=2, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, ...)
```

Fig. 8 Examining the metrics after tuning

```python
batch_X = test.iloc[:,1:];
predicted_probabilities = best_xgb_model.predict_proba(batch_X)
target_predicted = pd.DataFrame(predicted_probabilities[:, 1], columns=['class'])
def binary_convert(x):
    threshold = 0.5
    if x > threshold:
        return 1
    else:
        return 0

best_target_predicted_binary = target_predicted['class'].apply(binary_convert)
test_labels = test.iloc[:,0]
```

[11] ✓ 0.1s Python

Fig. 9 Getting the predicated target and test labels form the best xgb model

```python
plot_confusion_matrix(test_labels, best_target_predicted_binary)
```

[12] ✓ 0.1s Python

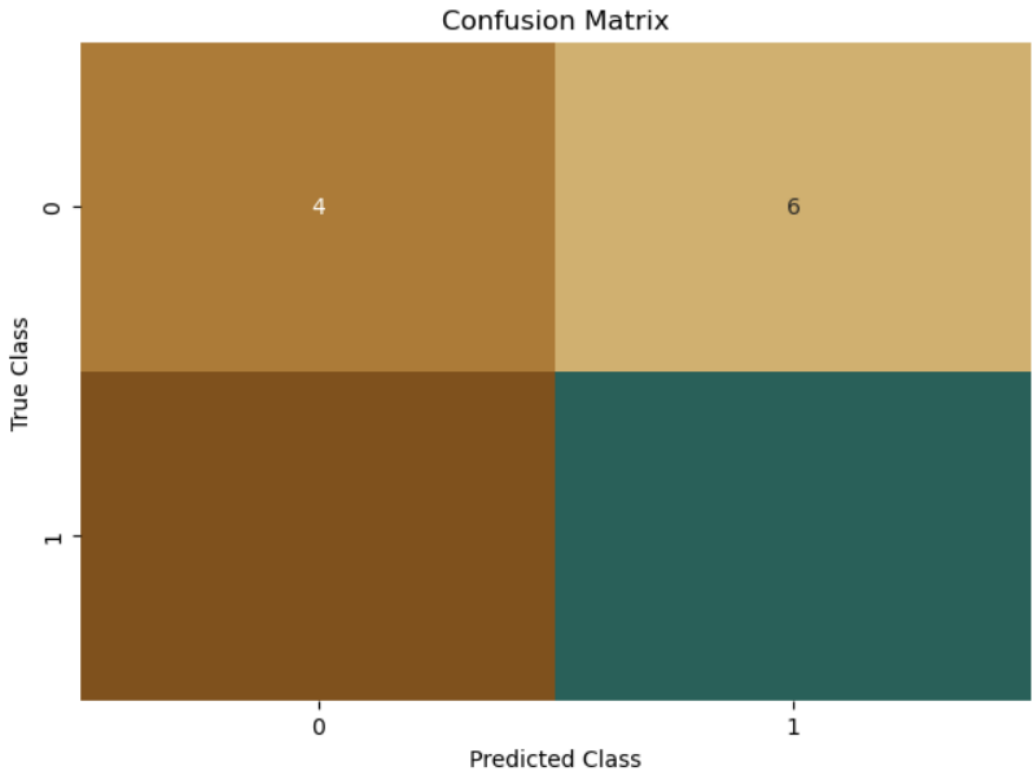

Fig. 10 Confusion matrix of the best xgb model after tuning

```
1 plot_roc(test_labels, best_target_predicted_binary)
```

[13] ✓ 0.3s Python

```
Sensitivity or TPR: 95.23809523809523%
Specificity or TNR: 40.0%
Precision: 76.92307692307693%
Negative Predictive Value: 80.0%
False Positive Rate: 60.0%
False Negative Rate: 4.761904761904762%
False Discovery Rate: 23.076923076923077%
Accuracy: 77.41935483870968%
Validation AUC 0.6761904761904762
```

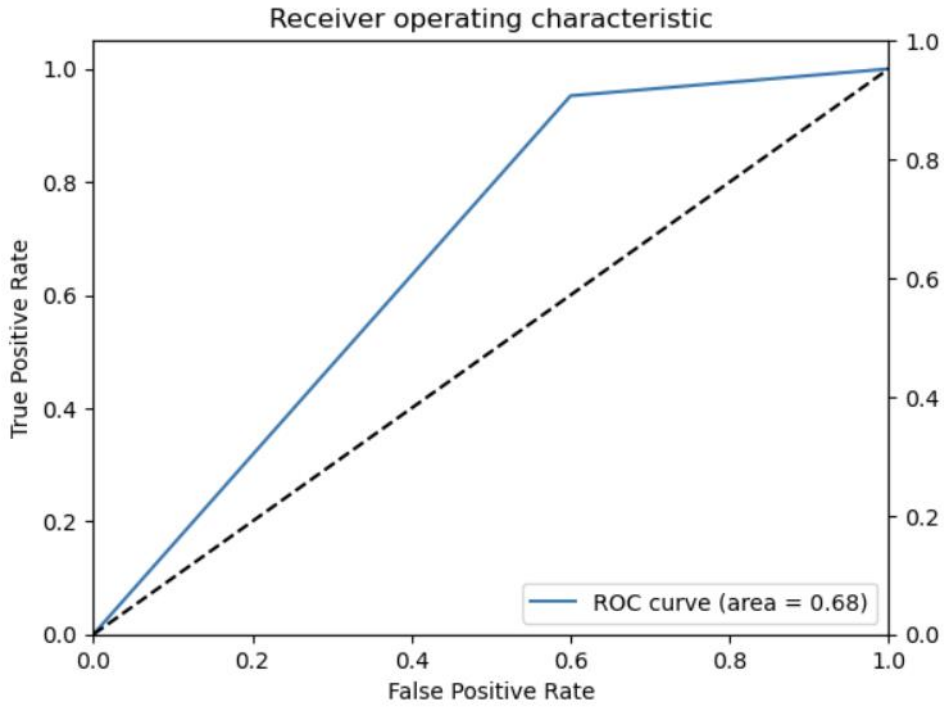

Fig. 11 ROC plot of the best XGBoost model

These results differ significantly from the original results. The new XGBoost model seem to perform worse than the original. Earlier, the ROC curve area was 0.8 while now it is 0.68. From the threshold I could see that the model was going to perform worse but the ROC curve explains it clearly. Another reason for the newer model to perform worse is there is not enough data for the model to be trained on. As we know the larger are ROC curve area, the better the model is. While in this case the new model after tuning the parameter actually seem to perform worse. This could be because of the following reasons:

- The model might already be good from the start
- We do not have a large amount of data for proper training
- We might be using a subset of the hyperparameter tuning ranges

After a little modification of the hyperparameter tuning ranges, I was able to get a better ROC curve.

```
Click here to ask Blackbox to help you code faster
%%time
from sklearn.model_selection import RandomizedSearchCV
# Define your XGBoost model
xgb_model = xgb.XGBClassifier(eval_metric='error@.40',
                              objective='binary:logistic')

# Define hyperparameter ranges
hyperparameter_ranges = {'alpha': [i for i in range(0, 101)],
                         'min_child_weight': [i for i in range(1, 12)],
                         'subsample': [i / 10 for i in range(1, 15)],
                         'eta': [i / 10 for i in range(1, 8)]}

search = RandomizedSearchCV(estimator=xgb_model,
                            param_distributions=hyperparameter_ranges,
                            scoring='neg_mean_squared_error',
                            n_iter=10,  # Number of parameter settings that are sampled
                            cv=5,       # Number of folds for cross-validation
                            verbose=1,
                            n_jobs=1)  # Use all available cores

search.fit(train.drop(['class'], axis = 1).values, train['class'].values)
best_params = search.best_params_
```

```
plot_roc(test_labels, best_target_predicted_binary)
```

[22] ✓ 0.3s Python

```
Sensitivity or TPR: 95.23809523809523%
Specificity or TNR: 60.0%
Precision: 83.33333333333334%
Negative Predictive Value: 85.71428571428571%
False Positive Rate: 40.0%
False Negative Rate: 4.761904761904762%
False Discovery Rate: 16.666666666666664%
Accuracy: 83.87096774193549%
Validation AUC 0.7761904761904761
```

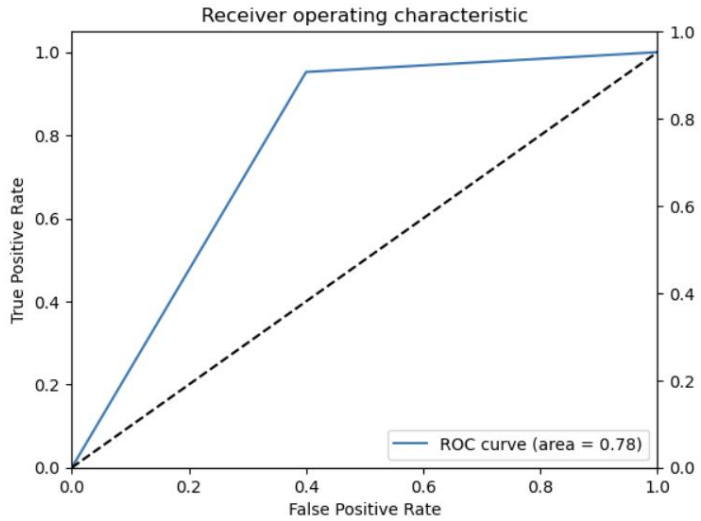

Fig. 12 New ROC curve

Conclusion

In Lab 7, I was part of a healthcare provider team that focused on utilizing machine learning to enhance the detection of abnormalities in orthopedic patients. The dataset, curated by Dr. Henrique da Mota, included biomechanical features for classification tasks such as identifying normal or abnormal conditions like Disk Hernia and Spondylolisthesis. By creating a hyperparameter tuning job, I aimed to optimize the model's performance by adjusting parameters like alpha, min_child_weight, subsample, and eta. The goal was to improve the model's accuracy in predicting abnormalities, ultimately benefiting patient care and diagnosis. Through this iterative process, I sought to fine-tune the model to achieve better results and contribute to more effective healthcare solutions.

By delving into the intricacies of hyperparameter tuning, I was able to explore different combinations of parameters to enhance the model's ability to classify orthopedic conditions accurately. This process involved tweaking the hyperparameters to strike a balance between bias and variance, ensuring that the model generalizes well to unseen data. The iterative nature of hyperparameter tuning allowed me to experiment with

various settings and evaluate their impact on the model's performance metrics.

As I delved deeper into the hyperparameter tuning process, I closely monitored the changes in the model's behavior and performance. By fine-tuning the hyperparameters, I aimed to address any shortcomings in the initial model and enhance its predictive capabilities. The iterative adjustments made during the tuning process were guided by a desire to achieve a more robust and reliable model for detecting abnormalities in orthopedic patients.

Through this lab work with the healthcare provider team, I was able to leverage the power of machine learning to contribute meaningfully to the field of orthopedic diagnostics. The insights gained from the hyperparameter tuning exercise not only improved the model's accuracy but also provided valuable learnings for future projects in healthcare analytics. Overall, the experience of fine-tuning the model underscored the importance of continuous improvement and optimization in machine learning applications within the healthcare domain.