n-assignment-f23-vangari-prashanth

November 15, 2023

0.1 Student name: Vangari Prashanth

0.2 Student Id: 11645119

0.3 Assignment: Model evaluation

[8]:
```
"""
Read the data stored in your local machine https://www.kaggle.com/datasets/
 ↪andrewmvd/fetal-health-classification
"""

import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('fetal_health.csv')
X = df.drop(columns='fetal_health')
y = df['fetal_health']
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,␣
 ↪test_size=0.2, random_state=1)
```

0.3.1 Part 1: Perform classification task using 5 different models (75 pts)

For each model (except Naive Bayes), use GridSearchCV() to tune the hyperparameters (note: testing ranges are specified in the assignment description)

Logistic Regression with L1 penalty (Lasso) (15 pts)

[8]:
```

```

Decision Tree (15 pts)

[8]:
```

```

KNN (15 pts)

[8]:
```

```

SVC (15 pts)

[8]:
```

```

SGD (15 pts)

[8]:

0.3.2 Part 2: Compare 5 different models' accuracies (25 pts)

Use the best hyperparameters returned from GridSearchCV to re-train the models, and compare the accuracies of all 5 models. The followings include hyperparameters of mentioned models: * Logistic Regression: * C: from 0.1 to 1, step = 0.3 * multi_class: auto, ovr, multinomial * solver: newton-cg * Decision Tree: * criterion: gini, entropy, log_loss * max_features: sqrt, log2 * max_depth: 2 to 5 * KNN: * n_neighbors: 3 to 7 * weights: uniform, distance * SVC: * degree: 2 to 5 * C: 0.1 to 1, step = 0.3 * kernel: poly, rbf * SGD: * loss: hinge, log_loss, modified_huber

[9]:

```
'''
This code uses grid search cross-validation and hyperparameter tuning to␣
 ↪optimize a logistic regression model.
A hypergrid parameters is used . To find the ideal set of hyperparameters, such␣
 ↪as regularization strength (C), multi-class strategy, and solver algorithm,␣
 ↪GridSearchCV is utilized. Following a fitting of the model to the training␣
 ↪set, the optimal hyperparameters are reported.
These ideal hyperparameters are used to instantiate a new logistic regression␣
 ↪model, which is then trained using the training set. The test data's target␣
 ↪variable is predicted using the final model, and the model's accuracy is␣
 ↪computed and reported, offering an evaluation of the logistic regression␣
 ↪model's performance.
'''

#1. Logistic Regression

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score

logRegModel = LogisticRegression()

pg = {
    'C': [0.1, 0.4, 0.7, 1.0],
    'multi_class': ['auto', 'ovr', 'multinomial'],
    'solver': ['newton-cg']
}

# make a GridSearchCV attribute example
grd_srch = GridSearchCV(estimator=logRegModel, param_grid=pg, cv=5, n_jobs=-1)

# Fit the GridSearchCV object to the training data
grd_srch.fit(X_train, y_train)
```

```
# Print the best hyperparameters found by GridSearchCV
print('best hyperparameters:', grd_srch.best_params_)


# Create a new Logistic Regression model with the best hyperparameters
fine_logRegModel = LogisticRegression(**grd_srch.best_params_)


# Fit the Logistic Regression model with the best hyperparameters to the␣
 ↪training data
fine_logRegModel.fit(X_train, y_train)

# Use the best model to predict the target variable for the test data
y_pred = fine_logRegModel.predict(X_test)

# Calculate and print the accuracy of the best Logistic Regression model on the␣
 ↪test data
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy of best Logistic Regression:', accuracy)
```

```
best hyperparameters: {'C': 1.0, 'multi_class': 'ovr', 'solver': 'newton-cg'}
Accuracy of best Logistic Regression: 0.8943661971830986
```

[10]:
```
#2.Decision tree
from sklearn.tree import DecisionTreeClassifier
import warnings

# Suppress DeprecationWarnings
warnings.filterwarnings('ignore')

# Define a function to ignore DeprecationWarnings
def fxn():
    warnings.warn("deprecated", DeprecationWarning)

# Use the function within a context manager to catch and ignore␣
 ↪DeprecationWarnings
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()

# Create a Decision Tree classifier
dec_tree = DecisionTreeClassifier()

# Define the hyperparameter grid to search
pg = {
    'criterion': ['gini', 'entropy', 'log_loss'],
```

```
    'max_features': ['sqrt', 'log2'],
    'max_depth': range(2, 6)
}

# Create a GridSearchCV object with Decision Tree classifier, hyperparameter␣
 ↪grid, and 5-fold cross-validation
gd_sch_cv2 = GridSearchCV(dec_tree, pg, cv=5)

# Fit the GridSearchCV object to the training data
gd_sch_cv2.fit(X_train, y_train)

# Print the best hyperparameters found by GridSearchCV
print("best hyperparameters:", gd_sch_cv2.best_params_)

# Create a new Decision Tree model with the best hyperparameters
fine_dt = DecisionTreeClassifier(criterion=gd_sch_cv2.best_params_['criterion'],
                                 max_features=gd_sch_cv2.
 ↪best_params_['max_features'],
                                 max_depth=gd_sch_cv2.best_params_['max_depth'])
# Fit the Decision Tree model with the best hyperparameters to the training data
fine_dt.fit(X_train, y_train)

# Use the best model to predict the target variable for the test data
y_pred = fine_dt.predict(X_test)

# Calculate and print the accuracy of the best Decision Tree model on the test␣
 ↪data
acc1 = accuracy_score(y_test, y_pred)

print("Accuracy of the best Decision Tree:", acc1)
```

```
best hyperparameters: {'criterion': 'entropy', 'max_depth': 5, 'max_features':
'sqrt'}
Accuracy of the best Decision Tree: 0.892018779342723
```

[11]:
```
#3 KNN

'''
Scikit-learn's `KNeighborsClassifier` is used in the machine learning workflow␣
 ↪to implement the K-Nearest Neighbors (KNN) algorithm. Optimizing the␣
 ↪classifier through hyperparameter tuning to improve model performance is an␣
 ↪essential step.
```

```
The method used to achieve this is a grid search over a predefined parameter␣
 ↪grid with two weighting strategies ('distance' and 'uniform') and various␣
 ↪values for the number of neighbors. To find the best configurations, the␣
 ↪search, assisted by {GridSearchCV}, runs a cross-validated assessment for␣
 ↪every set of parameters. A new KNN classifier is instantiated with these␣
 ↪best-found parameters and fitted to the training data after the optimal␣
 ↪hyperparameters have been determined. A different test dataset is then used␣
 ↪to assess the effectiveness of the model, and the accuracy metric offers a␣
 ↪numerical representation of the classifier's predictive capability.
'''


from sklearn.neighbors import KNeighborsClassifier


knn = KNeighborsClassifier()

# Finding best Hyperparameters
pg = {'n_neighbors': [3, 4, 5, 6, 7], 'weights': ['uniform', 'distance']}


grd_srchcv3 = GridSearchCV(knn, pg, cv=5)


grd_srchcv3.fit(X_train, y_train)

# Deliver the best hyperparameters
print("best hyperparameters: ", grd_srchcv3.best_params_)

# to fit the model utilize the best hyperparameters
fne_knn = KNeighborsClassifier(n_neighbors=grd_srchcv3.
 ↪best_params_['n_neighbors'], weights=grd_srchcv3.best_params_['weights'])
fne_knn.fit(X_train, y_train)


y_pred = fne_knn.predict(X_test)

# finding Accuracy
acc2 = accuracy_score(y_test, y_pred)

print("Accuracy of best KNN: ", acc2)
```

```
best hyperparameters:  {'n_neighbors': 3, 'weights': 'distance'}
Accuracy of best KNN:  0.9084507042253521
```

[12]:
```
#4.SVC
```

```
'''
We describe here how to use the `SVC` module from Scikit-learn to fine-tune a␣
 ↪Support Vector Classifier (SVC). The SVC model is very flexible, and its␣
 ↪ideal configuration for a given dataset can be found by adjusting a number␣
 ↪of hyperparameters. Using both polynomial and radial basis function (RBF)␣
 ↪kernels, a grid search, carried out via `GridSearchCV}, investigates a range␣
 ↪of values for the polynomial degree, the regularization parameter {C}, and␣
 ↪the kernel type. The combinations of these parameters are cross-validated in␣
 ↪this exhaustive search to determine the best model settings for the training␣
 ↪data.
After that, a new SVC instance is configured using the optimal parameters, and␣
 ↪it is retrained using the training set.
'''

from sklearn.svm import SVC


svc = SVC()

pm = {'degree': [2, 3, 4, 5], 'C': [0.1, 0.4, 0.7, 1.0], 'kernel': ['poly',␣
 ↪'rbf']}


clf = GridSearchCV(svc, pm)


clf.fit(X_train, y_train)

# Best hyperparameters
fin_deg = clf.best_params_['degree']
f_clf = clf.best_params_['C']
f_krnl = clf.best_params_['kernel']

print("best Hyperparameters: degree={}, C={}, kernel={}".format(fin_deg, f_clf,␣
 ↪f_krnl))

# create a new instance of the SVC model with the best hyperparameters
f_svc_mdl = SVC(degree=fin_deg, C=f_clf, kernel=f_krnl)


f_svc_mdl.fit(X_train, y_train)


y_pred = f_svc_mdl.predict(X_test)

# Accuracy
acc3 = accuracy_score(y_test, y_pred)
```

```
print("Accuracy of best SVC: ", acc3)
```

```
best Hyperparameters: degree=5, C=0.7, kernel=poly
Accuracy of best SVC:  0.8849765258215962
```

[13]:
```
#5.SGD

'''
The Stochastic Gradient Descent (SGD) Classifier is a linear classifier that␣
 ↪can handle large amounts of data effectively. The documentation explains how␣
 ↪to use it. During the model optimization process, Scikit-learn's␣
 ↪`GridSearchCV` is used to evaluate several loss functions, including␣
 ↪'hinge', 'log', and'modified_huber', in order to determine which one works␣
 ↪best for the given dataset. This grid search is carried out in parallel to␣
 ↪speed up computation and uses cross-validation to guarantee robustness␣
 ↪against overfitting. Following the identification and reporting of the␣
 ↪optimal hyperparameters, the classifier is redesigned using these parameters␣
 ↪to create the final SGD model.
Predictions on the test set are then made using this refined model.
'''



from sklearn.linear_model import SGDClassifier


sgd = SGDClassifier()

# Finding best Hyperparameters
pms = {'loss': ['hinge', 'log', 'modified_huber']}

# create an instance of GridSearchCV and fit it on the training data
grd_srchcv5 = GridSearchCV(sgd, pms, cv=5, n_jobs=-1)
grd_srchcv5.fit(X_train, y_train)


print('best hyperparameters:', grd_srchcv5.best_params_)

# predict the target variable using the best model
f_sgd = grd_srchcv5.best_estimator_
y_pred = f_sgd.predict(X_test)

# Accuracy
acc4 = accuracy_score(y_test, y_pred)
print('Accuracy of best SGD:', acc4)
```

```
best hyperparameters: {'loss': 'hinge'}
```

```
Accuracy of best SGD: 0.8356807511737089
```

[14]:
```
# Best Hyperparameters for Logistic Regression

'''
With hyperparameter optimization, the Logistic Regression model shows improved␣
 ↪predictive power on the test set. As a measure of the model's performance␣
 ↪and potential for generalization, the accuracy score quantifies the model's␣
 ↪predictions.

The maximum features, depth parameters, and best-found criterion are␣
 ↪specifically tailored into the Decision Tree Classifier. The predictive␣
 ↪accuracy of the model is calculated after retraining with the optimized␣
 ↪hyperparameters, offering information about how well the model performs the␣
 ↪classification task.

The number of neighbors and weighting scheme for the K-Nearest Neighbors (KNN)␣
 ↪classifier are calibrated for best results. The accuracy of the trained␣
 ↪model is then evaluated, providing insight into how well the KNN approach␣
 ↪works in the particular situation.

The kernel type, regularization parameter, and ideal degree of the Support␣
 ↪Vector Classifier (SVC) are set up. The SVC's performance in␣
 ↪high-dimensional spaces is demonstrated when the model is subsequently␣
 ↪trained and its accuracy is assessed.

Finally, the best estimator is used to implement the Stochastic Gradient␣
 ↪Descent (SGD) Classifier. The model's ability to perform linear␣
 ↪classification, particularly on large datasets, is demonstrated by the␣
 ↪accuracy of its predictions.
'''


fine_logRegModel = LogisticRegression(**grd_srch.best_params_)


fine_logRegModel.fit(X_train, y_train)


y_pred = fine_logRegModel.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)
print('Best Logistic Regression accuracy:', accuracy)

# Decision Tree with best hyperparameters
dt_best = DecisionTreeClassifier(criterion=gd_sch_cv2.best_params_['criterion'],
```

```
                                max_features=gd_sch_cv2.
 ↪best_params_['max_features'],
                                max_depth=gd_sch_cv2.best_params_['max_depth'])
dt_best.fit(X_train, y_train)


y_pred = dt_best.predict(X_test)

acc1 = accuracy_score(y_test, y_pred)
# Print accuracy for Decision Tree
print("Best Decision Tree accuracy:", acc1)

# K-Nearest Neighbors with best hyperparameters
fne_knn = KNeighborsClassifier(n_neighbors=grd_srchcv3.
 ↪best_params_['n_neighbors'], weights=grd_srchcv3.best_params_['weights'])
fne_knn.fit(X_train, y_train)


y_pred = fne_knn.predict(X_test)


acc2 = accuracy_score(y_test, y_pred)
# Print accuracy for KNN
print("Best KNN accuracy: ", acc2)



# SVC with best Hyperparameters

# create a new instance of the SVC model with the best hyperparameters
f_svc_mdl = SVC(degree=fin_deg, C=f_clf, kernel=f_krnl)

# fit the f_svc_mdl on the training data
f_svc_mdl.fit(X_train, y_train)

# Using the best hyperparameters, a decision tree
y_pred = f_svc_mdl.predict(X_test)

# calculate the accuracy score for f_svc_mdl
acc3 = accuracy_score(y_test, y_pred)

# print accuracy
print("Best SVC Accuracy: ", acc3)


#Predict the target variable for the test data using the best model.
f_sgd = grd_srchcv5.best_estimator_
```

```
y_pred = f_sgd.predict(X_test)

# Accuracy score is calculated
acc4 = accuracy_score(y_test, y_pred)
print('Best SGD accuracy:', acc4)
```

```
Best Logistic Regression accuracy: 0.8943661971830986
Best Decision Tree accuracy: 0.8779342723004695
Best KNN accuracy:  0.9084507042253521
Best SVC Accuracy:  0.8849765258215962
Best SGD accuracy: 0.8356807511737089
```

Observation: Based on the above, accuracy score, we can conclude that KNN has the highest accuracy score with 0.9084507042253521 where as SGD has least accuracy score with 0.8356807511737089