a1-COMP5318-2025-template

March 27, 2025

1 COMP5318 Assignment 1: Rice Classification

Group number: group-set1 200

Student 1 SID: 540799107

Student 2 SID: 520534353

[1]:
```
# Import all libraries
from sklearn.model_selection import StratifiedKFold,cross_val_score,␣
 ↪train_test_split, GridSearchCV
import pandas as pd
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier,␣
 ↪GradientBoostingClassifier

from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score
```

[2]:
```
# Ignore future warnings
from warnings import simplefilter
simplefilter(action='ignore', category=FutureWarning)
```

[3]:
```
# Load the rice dataset: rice-final2.csv
rice_data = pd.read_csv('rice-final2.csv', na_values='?')
```

[4]:
```
rice_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1400 entries, 0 to 1399
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype
```

```
---  ------              --------------  -----
 0   Area                1396 non-null   float64
 1   Perimiter           1396 non-null   float64
 2   Major_Axis_Length   1395 non-null   float64
 3   Minor_Axis_Length   1397 non-null   float64
 4   Eccentricity        1394 non-null   float64
 5   Convex_Area         1395 non-null   float64
 6   Extent              1398 non-null   float64
 7   class               1400 non-null   object
dtypes: float64(7), object(1)
memory usage: 87.6+ KB
```

[5]:
```
class_column = rice_data.columns[-1]
feature_columns = rice_data.columns[:-1]

# Store all numeric columns into list
numeric_columns = rice_data[feature_columns].select_dtypes(include=['number']).
 ↪columns.tolist()

# Fill missing value with mean value within each column
imputer = SimpleImputer(strategy='mean')
rice_data[numeric_columns] = imputer.fit_transform(rice_data[numeric_columns])

#Normalize all features columns into (0,1)
scaler = MinMaxScaler()
rice_data[numeric_columns] = scaler.fit_transform(rice_data[numeric_columns])

# Repalce class1,class2 with 0,1
rice_data[class_column] = rice_data[class_column].replace({'class1': 0,␣
 ↪'class2': 1})
```

[6]:
```
# Print first ten rows of pre-processed dataset to 4 decimal places as per␣
 ↪assignment spec
# A function is provided to assist

def print_data(X, y, n_rows=10):
    """Takes a numpy data array and target and prints the first ten rows.

    Arguments:
        X: numpy array of shape (n_examples, n_features)
        y: numpy array of shape (n_examples)
        n_rows: numpy of rows to print
    """
    for example_num in range(n_rows):
        for feature in X[example_num]:
            print("{:.4f}".format(feature), end=",")
```

```
        if example_num == len(X)-1:
            print(y[example_num],end="")
        else:
            print(y[example_num])


#4.Print the first 10 rows of the pre-processed dataset.
X = rice_data[numeric_columns].values
y = rice_data[class_column].values

print_data(X, y, n_rows=10)
```

```
0.4628,0.5406,0.5113,0.4803,0.7380,0.4699,0.1196,1
0.4900,0.5547,0.5266,0.5018,0.7319,0.4926,0.8030,1
0.6109,0.6847,0.6707,0.5409,0.8032,0.6253,0.1185,0
0.6466,0.6930,0.6677,0.5961,0.7601,0.6467,0.2669,0
0.6712,0.6233,0.4755,0.8293,0.3721,0.6803,0.4211,1
0.2634,0.2932,0.2414,0.4127,0.5521,0.2752,0.2825,1
0.8175,0.9501,0.9515,0.5925,0.9245,0.8162,0.0000,0
0.3174,0.3588,0.3601,0.3908,0.6921,0.3261,0.8510,1
0.3130,0.3050,0.2150,0.5189,0.3974,0.3159,0.4570,1
0.5120,0.5237,0.4409,0.6235,0.5460,0.5111,0.3155,1
```

1.0.1 Part 1: Cross-validation without parameter tuning

[7]:
```
## Setting the 10 fold stratified cross-validation
cvKFold=StratifiedKFold(n_splits=10, shuffle=True, random_state=0)

# The stratified folds from cvKFold should be provided to the classifiers
```

[8]:
```
# Logistic Regression
def logregClassifier(X, y):
    #Initializes a logistic regression classifier
    clf = LogisticRegression(random_state=0)

    #Conduct cross-validation
    scores = cross_val_score(clf, X, y, cv=cvKFold)

    #Reture the mean results of cross-validation
    return scores.mean()
```

[9]:
```
#Naïve Bayes
def nbClassifier(X, y):
    #Initializes a Naïve Bayes classifier
    clf = GaussianNB()

    scores = cross_val_score(clf, X, y, cv=cvKFold)
```

```
    return scores.mean()
```

[10]:
```
# Decision Tree
def dtClassifier(X, y):
    #Initializes a Decision Tre classifier and using entropy as criterion
    clf = DecisionTreeClassifier(random_state=0,criterion='entropy')

    scores = cross_val_score(clf, X, y, cv=cvKFold)
    return scores.mean()
```

[11]:
```
# Ensembles: Bagging, Ada Boost and Gradient Boosting
def bagDTClassifier(X, y, n_estimators, max_samples, max_depth):

    base_dt = DecisionTreeClassifier(random_state=0, criterion='entropy',␣
 ↪max_depth=max_depth)

    bag_clf = BaggingClassifier(estimator=base_dt,
                                n_estimators=n_estimators,
                                max_samples=max_samples,
                                random_state=0)

    scores = cross_val_score(bag_clf, X, y, cv=cvKFold)

    return scores.mean()

def adaDTClassifier(X, y, n_estimators, learning_rate, max_depth):
    base_dt = DecisionTreeClassifier(random_state=0, criterion='entropy',␣
 ↪max_depth=max_depth)

    ada_clf = AdaBoostClassifier(estimator=base_dt,
                                 n_estimators=n_estimators,
                                 learning_rate=learning_rate,
                                 random_state=0)

    scores = cross_val_score(ada_clf, X, y, cv=cvKFold)

    return scores.mean()

def gbClassifier(X, y, n_estimators, learning_rate):

    gb_clf = GradientBoostingClassifier(n_estimators=n_estimators,
                                        learning_rate=learning_rate,
                                        random_state=0)

    scores = cross_val_score(gb_clf, X, y, cv=cvKFold)

    return scores.mean()
```

1.0.2 Part 1 Results

[12]:
```
# Parameters for Part 1:

#Bagging
bag_n_estimators = 50
bag_max_samples = 100
bag_max_depth = 5

#AdaBoost
ada_n_estimators = 50
ada_learning_rate = 0.5
ada_bag_max_depth = 5

#GB
gb_n_estimators = 50
gb_learning_rate = 0.5

# Print results for each classifier in part 1 to 4 decimal places here:
print("LogR average cross-validation accuracy: {:.4f}".
 ↪format(logregClassifier(X, y)))
print("NB average cross-validation accuracy: {:.4f}".format(nbClassifier(X, y)))
print("DT average cross-validation accuracy: {:.4f}".format(dtClassifier(X, y)))
print("Bagging average cross-validation accuracy: {:.4f}".
 ↪format(bagDTClassifier(X, y, bag_n_estimators, bag_max_samples,␣
 ↪bag_max_depth)))
print("AdaBoost average cross-validation accuracy: {:.4f}".
 ↪format(adaDTClassifier(X, y, ada_n_estimators, ada_learning_rate,␣
 ↪ada_bag_max_depth)))
print("GB average cross-validation accuracy: {:.4f}".format(gbClassifier(X, y,␣
 ↪gb_n_estimators, gb_learning_rate)))
```

```
LogR average cross-validation accuracy: 0.9386
NB average cross-validation accuracy: 0.9264
DT average cross-validation accuracy: 0.9179
Bagging average cross-validation accuracy: 0.9414
AdaBoost average cross-validation accuracy: 0.9250
GB average cross-validation accuracy: 0.9300
```

1.0.3 Part 2: Cross-validation with parameter tuning

[13]:
```
# KNN
k = [1, 3, 5, 7, 9]
p = [1, 2]


def bestKNNClassifier(X, y):
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,␣
↪random_state=0)

    param_grid = {
        'n_neighbors': k,
        'p': p
    }

    knn = KNeighborsClassifier()
    # Conduct grid search
    grid = GridSearchCV(estimator=knn, param_grid=param_grid, cv=cvKFold,␣
↪scoring='accuracy')

    grid.fit(X_train, y_train)

    best_params = grid.best_params_
    best_cv_accuracy = grid.best_score_

    test_accuracy = grid.score(X_test, y_test)

    return best_params, best_cv_accuracy, test_accuracy
```

[14]:
```
# SVM
# You should use SVC from sklearn.svm with kernel set to 'rbf'
C = [0.01, 0.1, 1, 5]
gamma = [0.01, 0.1, 1, 10]

def bestSVMClassifier(X, y):

    X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,␣
↪random_state=0)

    param_grid = {
        'C': [0.01, 0.1, 1, 5],
        'gamma': [0.01, 0.1, 1, 10]
    }

    svm = SVC(kernel='rbf', random_state=0)

    grid = GridSearchCV(estimator=svm, param_grid=param_grid, cv=cvKFold,␣
↪scoring='accuracy')
    grid.fit(X_train, y_train)

    best_params = grid.best_params_
    best_cv_accuracy = grid.best_score_

    test_accuracy = grid.score(X_test, y_test)
```

```
    return best_params, best_cv_accuracy, test_accuracy
```

[15]:
```
# Random Forest
# You should use RandomForestClassifier from sklearn.ensemble with information␣
 ↪gain and max_features set to 'sqrt'.
n_estimators = [10, 30, 60, 100]
max_leaf_nodes = [6, 12]

def bestRFClassifier(X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,␣
 ↪random_state=0)

    param_grid = {
        'n_estimators': [10, 30, 60, 100],
        'max_leaf_nodes': [6, 12]
    }

    rf = RandomForestClassifier(criterion='entropy', max_features='sqrt',␣
 ↪random_state=0)

    grid = GridSearchCV(estimator=rf, param_grid=param_grid, cv=cvKFold)
    grid.fit(X_train, y_train)

    best_params = grid.best_params_
    best_cv_accuracy = grid.best_score_

    y_pred = grid.predict(X_test)
    test_accuracy = accuracy_score(y_test, y_pred)
    macro_f1 = f1_score(y_test, y_pred, average='macro')
    weighted_f1 = f1_score(y_test, y_pred, average='weighted')

    return best_params, best_cv_accuracy, test_accuracy, macro_f1, weighted_f1
```

1.0.4 Part 2: Results

[16]:
```
best_params_KNN, best_cv_acc_KNN, test_acc_KNN = bestKNNClassifier(X, y)
best_params_SVM, best_cv_acc_SVM, test_acc_SVM = bestSVMClassifier(X, y)
best_params_RF, best_cv_acc_RF, test_acc_RF, macro_f1, weighted_f1 =␣
 ↪bestRFClassifier(X, y)
# Results of KNN
print("KNN best k: {}".format(best_params_KNN['n_neighbors']))
print("KNN best p: {}".format(best_params_KNN['p']))
print("KNN cross-validation accuracy: {:.4f}".format(best_cv_acc_KNN))
print("KNN test set accuracy: {:.4f}".format(test_acc_KNN))
print()
```

```
# Reults of SVM
print("SVM best C: {:.4f}".format(best_params_SVM['C']))
print("SVM best gamma: {:.4f}".format(best_params_SVM['gamma']))
print("SVM cross-validation accuracy: {:.4f}".format(best_cv_acc_SVM))
print("SVM test set accuracy: {:.4f}".format(test_acc_SVM))
print()

#Results of RF
print("RF best n_estimators: {}".format(best_params_RF['n_estimators']))
print("RF best max_leaf_nodes: {}".format(best_params_RF['max_leaf_nodes']))
print("RF cross-validation accuracy: {:.4f}".format(best_cv_acc_RF))
print("RF test set accuracy: {:.4f}".format(test_acc_RF))
print("RF test set macro average F1: {:.4f}".format(macro_f1))
print("RF test set weighted average F1: {:.4f}".format(weighted_f1))
```

```
KNN best k: 9
KNN best p: 1
KNN cross-validation accuracy: 0.9390
KNN test set accuracy: 0.9314

SVM best C: 5.0000
SVM best gamma: 1.0000
SVM cross-validation accuracy: 0.9457
SVM test set accuracy: 0.9343

RF best n_estimators: 30
RF best max_leaf_nodes: 12
RF cross-validation accuracy: 0.9390
RF test set accuracy: 0.9371
RF test set macro average F1: 0.9355
RF test set weighted average F1: 0.9370
```

1.0.5 Part 3: Reflection

Write one paragraph describing the most important thing that you have learned throughout this assignment.

Student 1: When I finished this assignment, I realized something I hadn't noticed before. Pre-processing data and choosing the right parameters is really important. Before starting, I was mostly focused on finding the best algorithm, assuming that a better method would yield better results. However, after dealing with missing values, adjusting class labels, completing normalization, and finding the best parameters through grid search, I saw that these small steps actually improved my results significantly,sometimes even more than using advanced or complicated algorithms. From now on, I will pay more attention to cleaning data and fine-tuning settings, rather than just relying on selecting a sophisticated algorithm. This experience taught me that a combination of proper data preparation and parameter optimization is key to achieving optimal performance.

Student 2:Through this assignment, we went through a complete machine learning pipeline — from the initial stage of data cleaning, to applying different classification models, and finally using cross-validation to evaluate model performance and tune hyperparameters. This process helped us gain a deeper understanding of machine learning. Through hands-on implementation, we observed that even when working on the same task, using different models or different parameters can lead to significantly different outcomes. My specific responsibility was data cleaning and implementing and evaluating the performance of Bagging, AdaBoost, and Gradient Boosting models. By completing this part, I gained a deeper understanding of ensemble methods. I now better understand how bootstrap aggregation and boosting manipulate the training data, and I also learned how bagging, AdaBoost, and Gradient Boosting are implemented in code.