Project

March 18, 2021

[ ]:
```
%config InlineBackend.figure_format = 'retina'
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split, cross_val_score,␣
 ↪GridSearchCV, KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import make_scorer, accuracy_score, f1_score, roc_auc_score
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier

np.random.seed(42)

def encodeBind(df, features):
    dummies = pd.get_dummies(df[features])
    resDF = pd.concat([df, dummies], axis=1)
    resDF.drop(features, axis=1, inplace = True)
    return(resDF)

def preprocessDF(df, featureEncode, featureScale):
    scaler = StandardScaler()
    dfLabel = df.goal

    df.drop("goal", axis = 1, inplace = True)
    if featureEncode != None:
        df = encodeBind(df, featureEncode)
    df["goal"] = dfLabel

    df[featureScale] = scaler.fit_transform(df[featureScale])

    return(df)
```

[ ]:
```
# Adult dataset
adultNames = ["age", "workclass", "fnlwgt", "education", "education-num",␣
 ↪"marital-status",
              "occupation", "relationship", "race", "sex", "capital-gain",␣
 ↪"capital-loss",
              "hours-per-week", "native-country", "goal"]
adultEncode = ["workclass", "education", "marital-status", "occupation",␣
 ↪"relationship", "race", "sex", "native-country"]
adultScale = ["age", "fnlwgt", "education-num", "capital-gain", "capital-loss",␣
 ↪"hours-per-week"]

adult = pd.read_table("adult.data", names = adultNames, sep = ",\s",␣
 ↪engine='python')
adult = adult[adult["workclass"] != "?"]
adult["goal"] = np.where(adult.goal == ">50K", 1, 0)
```

[ ]:
```
adult[adultScale].hist()
```

[ ]:
```
adult = preprocessDF(adult, adultEncode, adultScale)
adult
```

[ ]:
```
# Covertype dataset
covNames = ["elevation", "aspect", "slope", "hordishydro", "verdishydro",␣
 ↪"hordisroad", "hillam", "hillnoon", "hillpm",
            "hordisfire"] + ["wild" + str(i) for i in range(1,5)] + ["soil" +␣
 ↪str(i) for i in range(1,41)] + ["goal"]
covScale = ["elevation", "aspect", "slope", "hordishydro", "verdishydro",␣
 ↪"hordisroad", "hillam", "hillnoon", "hillpm",
            "hordisfire"]
cov = pd.read_table("covtype.data", sep = ",", names = covNames)
cov.goal = np.where(cov.goal == cov["goal"].value_counts().idxmax(), 1, 0)
```

[ ]:
```
cov[covScale].hist()
```

[ ]:
```
cov = preprocessDF(cov, None, covScale)
cov
```

[ ]:
```
# Letter dataset
letterNames = ["goal", "x-box", "y-box", "width", "height", "onpix", "x-bar",
              "y-bar", "x2bar", "y2bar", "xybar", "x2ybr", "xy2br", "x-ege",
              "xegvy", "y-ege", "yegvx"]
letterScale = ["x-box", "y-box", "width", "height", "onpix", "x-bar",
              "y-bar", "x2bar", "y2bar", "xybar", "x2ybr", "xy2br", "x-ege",
              "xegvy", "y-ege", "yegvx"]
letter = pd.read_table("letter-recognition.data", sep = ",", names =␣
 ↪letterNames)
letterCols = list(letter.columns)
```

```
letterCols[-1], letterCols[0] = letterCols[0], letterCols[-1]
letter = letter[letterCols]
chosenLetter = [chr(i) for i in range(ord('A'), ord('M')+1)]
letter.goal = letter.goal.apply(lambda x: 1 if x in chosenLetter else 0)
```

[ ]:
```
letter[letterScale].hist()
```

[ ]:
```
letter = preprocessDF(letter, None, letterScale)
letter
```

[ ]:
```
# Avila dataset
avilaNames = ["interdis", "upmar", "lowmar", "exploi", "rownum", "modratio",
              "interspace", "weight", "peaknum", "modratio-interspace", "goal"]
avila = pd.read_csv("avila.txt", sep = ",", names = avilaNames)
avila.goal = np.where(avila.goal == "A", 1, 0)
```

[ ]:
```
avila[list(avila.columns)[:-1]].hist()
```

[ ]:
```
avila
```

[ ]:
```
# Bean dataset
#beanNames = [""]
bean = pd.read_excel("Dry_Bean_Dataset.xlsx", engine='openpyxl')
beanScale = list(bean.columns)[:-1]
bean.Class = np.where(bean.Class == bean.Class.value_counts().idxmax(), 1, 0)
bean.columns = [*bean.columns[:-1], 'goal']
bean[beanScale].hist()
```

[ ]:
```
bean = preprocessDF(bean, None, beanScale)
bean
```

[ ]:
```
def getTrialTrainTest(pipe, param, score_function, X_train, X_test, Y_train,␣
 ↪Y_test):
    pipe.set_params(**param)
    pipe.fit(X_train, Y_train)
    y_pred_train = pipe.predict(X_train)
    y_pred_test = pipe.predict(X_test)
    trialTrain = score_function(Y_train, y_pred_train)
    trialTest = score_function(Y_test, y_pred_test)

    return trialTrain, trialTest

def spitoutres(data, pipe, param):
    aucTrialTrain = []
    accTrialTrain = []
    f1TrialTrain = []
    aucTrialTest = []
```

```
    accTrialTest = []
    f1TrialTest = []
    for trial in range(5):
        print(trial)
        X, Y = data.iloc[:,:-1], data.iloc[:,-1]
        X_train, X_test, Y_train, Y_test = train_test_split(X, Y,␣
↪train_size=5000, shuffle=True)
        gs = GridSearchCV(pipe, param_grid = param, cv = StratifiedKFold(),␣
↪n_jobs = -1,
                          scoring = scoring, refit = False)
        gs.fit(X_train, Y_train)
        results = gs.cv_results_

        auc_best_index = np.argmin(results['rank_test_AUC'])
        auc_best_param = results['params'][auc_best_index]

        accuracy_best_index = np.argmin(results['rank_test_Accuracy'])
        accuracy_best_param = results['params'][accuracy_best_index]

        f1_best_index = np.argmin(results['rank_test_F1'])
        f1_best_param = results['params'][f1_best_index]

        aucTr, aucTest = getTrialTrainTest(pipe, auc_best_param, roc_auc_score,␣
↪X_train, X_test, Y_train, Y_test)
        accTr, accTest = getTrialTrainTest(pipe, accuracy_best_param,␣
↪accuracy_score, X_train, X_test, Y_train, Y_test)
        f1Tr, f1Test = getTrialTrainTest(pipe, f1_best_param, f1_score,␣
↪X_train, X_test, Y_train, Y_test)

        aucTrialTrain.append(aucTr)
        accTrialTrain.append(accTr)
        f1TrialTrain.append(f1Tr)
        aucTrialTest.append(aucTest)
        accTrialTest.append(accTest)
        f1TrialTest.append(f1Test)

    return aucTrialTrain, accTrialTrain, f1TrialTrain, aucTrialTest,␣
↪accTrialTest, f1TrialTest

def runPerAlgo(algo, datasets, param):
    aucTrainMean, accTrainMean, f1TrainMean, aucTestMean, accTestMean,␣
↪f1TestMean = [], [], [], [], [], []
    aucTrainRaw, accTrainRaw, f1TrainRaw, aucTestRaw, accTestRaw, f1TestRaw =␣
↪[], [], [], [], [], []
    for data in datasets:
```

```
        aucTrain, accTrain, f1Train, aucTest, accTest, f1Test =␣
↪spitoutres(data, algo, param)

        accTrainRaw.append(accTrain)
        accTestRaw.append(accTest)

        aucTrainMean.append(np.mean(aucTrain))
        accTrainMean.append(np.mean(accTrain))
        f1TrainMean.append(np.mean(f1Train))
        aucTestMean.append(np.mean(aucTest))
        accTestMean.append(np.mean(accTest))
        f1TestMean.append(np.mean(f1Test))

    print("Raw train score")

    accTrainDF = pd.DataFrame(accTrainRaw, columns = ["Trial1", "Trial2",␣
↪"Trial3", "Trial 4", "Trial 5"], dtype = float)
    print(accTrainDF)

    print("Raw test score")

    accTestDF = pd.DataFrame(accTestRaw, columns = ["Trial1", "Trial2",␣
↪"Trial3", "Trial 4", "Trial 5"], dtype = float)
    print(accTestDF)

    theDFTest = pd.DataFrame(list(zip(aucTestMean, accTestMean, f1TestMean)),␣
↪columns = ["AUC", "ACC", "F1"], dtype = float)
    theDFTrain = pd.DataFrame(list(zip(aucTrainMean, accTrainMean,␣
↪f1TrainMean)), columns = ["AUC", "ACC", "F1"], dtype = float)

    print("Test over problems")
    print(theDFTest.mean(axis = 1))
    print("Test over metrics")
    print(theDFTest.mean(axis = 0))

    print("Train over problems")
    print(theDFTrain.mean(axis = 1))
    print("Train over metrics")
    print(theDFTrain.mean(axis = 0))
```

[ ]:
```
scoring = {'AUC': 'roc_auc', 'Accuracy': make_scorer(accuracy_score), 'F1':␣
↪make_scorer(f1_score)}

knnParam = {'knn__n_neighbors': np.arange(1,106,4),
    'knn__weights': ["uniform", "distance"]}
nnParam = [{
    "nn__hidden_layer_sizes": [(1,), (2,), (4,), (8,), (32,), (128,)],
```

```
    "nn__solver": ["adam"]
}, {
    "nn__hidden_layer_sizes": [(1,), (2,), (4,), (8,), (32,), (128,)],
    "nn__solver": ["sgd"],
    "nn__momentum": [0, 0.2, 0.5, 0.9]
}]
rfParam = {'rf__max_features': ["auto", "log2", None, 1, 2, 4, 6, 8, 10]}
lrParam = [{
    'lr__solver': ['saga'],
    'lr__penalty': ['l1', 'l2'],
    'lr__C': [10**i for i in range(-8,5)]
    }, {
    'lr__solver': ['lbfgs'],
    'lr__penalty': ['l2'],
    'lr__C': [10**i for i in range(-8,5)]
    }, {
    'lr__solver': ['lbfgs','saga'],
    'lr__penalty': ['none'],
}]

knnPipe = Pipeline([("knn", KNeighborsClassifier())])
nnPipe = Pipeline([("nn", MLPClassifier(max_iter=500))])
rfPipe = Pipeline([("rf", RandomForestClassifier(n_estimators = 1024, n_jobs =␣
 ↪-1))])
lrPipe = Pipeline([("lr", LogisticRegression(max_iter = 5000))])

params = [knnParam, nnParam, rfParam, lrParam]
pipes = [knnPipe, nnPipe, rfPipe, lrPipe]
datasets = [adult, cov, letter, avila, bean]
```

[ ]:
```
### KNN model
runPerAlgo(knnPipe, datasets, knnParam)
```

[ ]:
```
### NN model
runPerAlgo(nnPipe, datasets, nnParam)
```

[ ]:
```
### RF Model
runPerAlgo(rfPipe, datasets, rfParam)
```

[ ]:
```
### LR Model
runPerAlgo(lrPipe, datasets, lrParam)
```