In [ ]:

```
pip install Orange3
```

In [ ]:

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn.model_selection import train_test_split
# import category encoders
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import accuracy_score
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold
from sklearn.model_selection import RepeatedKFold
from sklearn import svm
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import precision_score
from sklearn import metrics
from sklearn.metrics import precision_recall_curve
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_classification
import xgboost as xgb
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from mlxtend.evaluate import paired_ttest_5x2cv
from sklearn.model_selection import RepeatedStratifiedKFold
from numpy import mean
from numpy import std
from sklearn.model_selection import cross_val_score
from scipy import stats as stats
from scipy.stats import rankdata
#from orange3.evaluation import compute_CD, graph_ranks
import Orange as ora
import matplotlib.pyplot as plt
# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

Comparing multiple classifiers over multiple datasets

In [ ]:

```
data_antibody = pd.read_csv("/kaggle/input/data-antibody/data_antibody.csv")
data_antibody=data_antibody.astype(int)
data_pcr= pd.read_csv("/kaggle/input/pcr-data/pcr_data.csv")
```

```
data_pcr=data_pcr.astype(int)
data_both= pd.read_csv("/kaggle/input/both-covid-data/both_covid_data.csv")
```

In [ ]:

```
def RandomForest_classif(x_train,y_train):
    #Classification
    clf= RandomForestClassifier()
    clf=clf.fit(x_train, y_train)
    return clf

def Kneighbors_classif(x_train,y_train):
    #Classification
    clf= KNeighborsClassifier(n_neighbors=3)
    clf= clf.fit(x_train, y_train)
    return clf

def DecisionTree_classif(x_train,y_train):
    #Classification
    clf = tree.DecisionTreeClassifier()
    clf = clf.fit(x_train,y_train)
    return clf

def mpl_classif(x_train,y_train):
    clf =  MLPClassifier(max_iter=300,solver='lbfgs', alpha=1e-5, random_state=42)
    clf=clf.fit(x_train,y_train)
    return clf

def gb_classif(x_train,y_train):
    param_dist = {'n_estimators':500,'max_depth':5}
    clf=GradientBoostingClassifier(**param_dist)
    clf=clf.fit(x_train, y_train)
    return clf

def xgb_classif(x_train,y_train):
    param_dist = {'n_estimators':300,'max_depth':9,'min_child_weight': 2}

    clf = xgb.XGBClassifier(**param_dist)

    return clf.fit(x_train, y_train)

def svc_classif(x_train,y_train):
    regr = svm.SVC()
    regr=regr.fit(x_train, y_train)
    return regr
```

In [ ]:

```
def calculate_metrics(x_train, x_test,y_train, y_test,clf):

        #prediction
        y_pred=clf.predict(x_test)

        #accuracy score
        #acc=accuracy_score(y_test,y_pred)*100

        #confusion matrix
        tn, fp, fn, tp = confusion_matrix(y_test,y_pred).ravel()

        """
        #precision
        data.iloc[k:k+1,:1]=(tp/(tp+fp))*100

        data.iloc[k:k+1,1:2]=acc
        """
        #recall
        recall=(tp/(tp+fn))*100


        return recall
```

In [ ]:

```
def calculate_prediction(x,y):

    dt=pd.DataFrame(columns=['DT','RF','GBM','XGBoost','Mlp','SVM','KNN'],index=range(50
))


    n=0
    for k in range(50):

        x_train, x_test, y_train, y_test = train_test_split(
        x, y, test_size=0.3,random_state=n,stratify=y)

        clf_svm=svc_classif(x_train,y_train)
        dt.iloc[k:k+1,5:6]=calculate_metrics(x_train, x_test,y_train, y_test,clf_svm)


        clf_knn=Kneighbors_classif(x_train,y_train)
        dt.iloc[k:k+1,6:]=calculate_metrics(x_train, x_test,y_train, y_test,clf_knn)

        clf_dt=DecisionTree_classif(x_train,y_train)
        dt.iloc[k:k+1,:1]=calculate_metrics(x_train, x_test,y_train, y_test,clf_dt)

        clf_rf=RandomForest_classif(x_train,y_train)
        dt.iloc[k:k+1,1:2]=calculate_metrics(x_train, x_test,y_train, y_test,clf_rf)

        clf_mlp=mpl_classif(x_train,y_train)
        dt.iloc[k:k+1,4:5]=calculate_metrics(x_train, x_test,y_train, y_test,clf_mlp)

        clf_gbm=gb_classif(x_train,y_train)
        dt.iloc[k:k+1,2:3]=calculate_metrics(x_train, x_test,y_train, y_test,clf_gbm)

        clf_xgboost=xgb_classif(x_train,y_train)
        dt.iloc[k:k+1,3:4]=calculate_metrics(x_train, x_test,y_train, y_test,clf_xgboost
)

        n+=1


    return dt
```

In [ ]:

```
dt=calculate_prediction(data_both.iloc[:,0:10],data_both['Class'])
dt
```

In [ ]:

```
# Then, we extract the performances as a numpy.ndarray.
performances_array =  dt.iloc[:,:].values
algorithms_names=dt.columns.values
# Finally, we apply the Friedman test.
t, p =stats.friedmanchisquare(*performances_array)
```

In [ ]:

```
# summarize
print('P-value:' , p, 't-Statistic: %.3f' % (t))
# interpret the result
if p <= 0.1:
    print('Difference between mean performance is probably real')
else:
    print('Algorithms probably have the same performance')
```

In [ ]:

```
# Calculating the ranks of the algorithms for each dataset. The value of p is multipled b
y -1
# because the rankdata method ranks from the smallest to the greatest performance values.
# Since we are considering Recall as our performance measure, we want larger values to be
```

```
best ranked.
ranks = np.array([rankdata(-p) for p in performances_array])
# Calculating the average ranks.
average_ranks = np.mean(ranks, axis=0)
print('\n'.join('{} average rank: {}'.format(a, r) for a, r in zip(algorithms_names, ave
rage_ranks)))
```

In [ ]:

```
# This method computes the critical difference for Nemenyi test with alpha=0.1.
# For some reason, this method only accepts alpha='0.05' or alpha='0.1'.
cd =  ora.evaluation.compute_CD(average_ranks,
n=len(dt),
alpha='0.1',
test='nemenyi')
# This method generates the plot.
ora.evaluation.graph_ranks(average_ranks,
names=algorithms_names,
cd=cd,
width=10,
textspace=1.5,
reverse=True)
plt.savefig('model_evaluation.jpg')
plt.show()
```

In [ ]:

```
# This method computes the critical difference for Bonferroni-Dunn test with alpha=0.1.
# For some reason, this method only accepts alpha='0.05' or alpha='0.1'.
cd = ora.evaluation.compute_CD(average_ranks,
n=len(dt),
alpha='0.1',
test='bonferroni-dunn')
# This method generates the plot.
ora.evaluation.graph_ranks(average_ranks,
names=algorithms_names,
cd=cd,
cdmethod=0,
width=10,
textspace=1.5,
reverse=True)
plt.show()
```