Import Libraries

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import random
import copy
from sklearn.model_selection import LeaveOneOut, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import matthews_corrcoef, confusion_matrix, accuracy_score, classification_report
from sklearn.feature_selection import SelectKBest, f_classif
from scipy.stats import fisher_exact
from sklearn.preprocessing import StandardScaler
```

Read In Data

```
gene_df = pd.read_excel('/kaggle/input/pomeroy-et-al-brain-data/Dataset_C_MD_outcome2.gct (6).xlsx')
```

Create a list of tuples formated ('patientIDs', 'patientStatus') where patientIDs is Brain_MD_X and patientStatus is 0 (dead) or 1 (alive)

+ Code + Text

```
# Brin_MD_X
patientIDs = gene_df.columns[2:].to_list()
# dead is 0, alive is 1
patientStatus = [0 if i <= 21 else 1 for i in range(1, 61)]

# Create the list of tuples
patients = list(zip(patientIDs, patientStatus))

#print(patients)
```

Create Training and Testing Datasets.

```
goodData = False # boolean to see if the data is good. About half of the dead
                 # patients and half of the alive patients should be in the data
                 # set of 30

attemptNumber = 0 # number of attempts to get good data

# empty lists that will contain all the patientIDs and their status for a training and testing dataset
patients_trainingSet = [None] * 30
patients_testingSet = [None] * 30


while not goodData:

    # add 1 to number of attempts to get good data
    attemptNumber +=1

    # shuffle around patients
    random.shuffle(patients)

    # pick 30 random patients
    temp_trainingSet = patients[:30]

    # set number of dead patients in set variable to 0
    dead, alive = 0, 0

    # Count how many dead patients were selected for the set
    for patient in temp_trainingSet:
        if patient[1] == 0:
            dead+=1
        else:
            alive+=1

    # To stop infinite loops
    if attemptNumber == 10:
        break

    # If about half of the 39 alive patients are in this data set, populate the training and testing datasets, set goodData to True and break
    if 17 <= alive <= 22:
        print(alive, 'patients who responded and ', dead, 'patients who did not respond in testing dataset\n\n')
```

```
            patients_trainingSet = patients[:30]
            patients_testingSet = patients[30:]
            goodData = True
        else:
            # If attempt fails
            print("Failed attempt", attemptNumber)

# print training and datasets
#print('Training Dataset:\n', patients_trainingSet,'\n\n')
#print('Testing Dataset:\n', patients_testingSet)
```

```
 Failed attempt 1
 18 patients who responded and  12 patients who did not respond in testing dataset
```

Ensure data sets are good

1. Ensure the correct number of patients are in the datasets
2. Ensure there is no overlap between data sets (no duplicate patients)

```
if len(patients_trainingSet) == 30:
    if len(patients_testingSet) == 30:
        print('The correct number of patients per dataset')
    else:
        print('panic', len(patients_trainingSet), len(patients_testingSet))

for train_patient in patients_trainingSet:
    for test_patient in patients_testingSet:
        if train_patient[0] == test_patient[0]:
            print('Panic, ', test_patiet[0], ' is duplicated')
            sys.exit()

print("No duplicate patients across training and testing datasets")
```

```
 The correct number of patients per dataset
 No duplicate patients across training and testing datasets
```

K-Nearest Neighbor analysis

Craete a input

1. outcome -> a list of the outcomes for all the patients in the training data set
2. expressionLevel -> a list of lists. For every probe, each patient's expression level for that probe is consolidated into a list.

```
# Create a copy of the entire dataset
train = gene_df.copy()

# Get a list of patients that will be removed to create the training dataset
patTrain = [lisTrain[0] for lisTrain in patients_testingSet]

# for each patient that is in the testing set, drop it out of the training set
for patient in patTrain:
    train = train.drop(columns=[patient])

# A list of the outcomes (dead/alive) for all the patients in the training data set
trainOutcomes = [lis[1] for lis in patients_trainingSet]

# A list of lists. For every probe, each patient's expression level for that probe is consolidated into a list
train_expressionLevel = train.drop(train.iloc[:, 0:2], axis=1)
train_expressionLevel = train_expressionLevel.transpose()
```

```
# Create a copy of the entire dataset
test = gene_df.copy()

# Get a list of patients that will be removed to create the training dataset
patTest = [lisTest[0] for lisTest in patients_trainingSet]

# for each patient that is in the testing set, drop it out of the training set
for patient in patTest:
    test = test.drop(columns=[patient])

# A list of the outcomes (dead/alive) for all the patients in the training data set
testOutcomes = [lisTest[1] for lisTest in patients_testingSet]

# A list of lists. For every probe, each patient's expression level for that probe is consolidated into a list
test_expressionLevel = test.drop(train.iloc[:, 0:2], axis=1)
test_expressionLevel = test_expressionLevel.transpose()
```

Select the best features from train

```
kFeatures = 50

def best(X_train, y_train, X_test, y_test, kFeatures):
    selector = SelectKBest(f_classif, k=kFeatures)

    # Fit the feature selector on the training data and transform both training and test data
    train_selected = selector.fit_transform(X_train, y_train)
    test_selected = selector.transform(X_test)

    return train_selected, test_selected, y_train

train_expressionLevel, test_expressionLevel, trainOutcomes = best(train_expressionLevel, trainOutcomes, test_expressionLevel, testOutcomes,
print(train_expressionLevel, trainOutcomes)
```

```
 [[ 175 -166   79 ... -137  -93 -221]
  [ -18  226  -33 ... -476  134 -149]
  [ 229 -195  624 ...  187   10 -202]
  ...
  [ 507 -233  383 ...  -79  -74 -124]
  [ 378 -101  126 ...   82  -27 -215]
  [ 159   31  455 ...  -76   76 -153]] [0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]
```

Apply the classifier to the testing set and evaluate the accuracy

```
def knn_leave_one_out(X, y):
    loo = LeaveOneOut()

    # Define parameter grid for grid search
    param_grid = {'n_neighbors': range(1, 11)}  # This can be adjusted up to the number of patients in the dataset. Lower is better, too low

    # Initialize KNN classifier
    knn = KNeighborsClassifier()

    # Perform grid search with leave-one-out cross-validation
    grid_search = GridSearchCV(knn, param_grid, scoring='accuracy', cv=loo)
    grid_search.fit(X, y)

    # Get the best parameters
    best_params = grid_search.best_params_

    # Train the best KNN classifier on the entire dataset
    best_knn = KNeighborsClassifier(n_neighbors=best_params['n_neighbors'])
    best_knn.fit(X, y)

    return best_knn, best_params

# Perform leave-one-out testing
best_knn, best_params = knn_leave_one_out(train_expressionLevel, trainOutcomes)
print("Best Parameters:", best_params)
print(best_knn)
```

```
 Best Parameters: {'n_neighbors': 6}
 KNeighborsClassifier(n_neighbors=6)
```

```
# Use the trained KNN classifier to make predictions on the test data
testPredictions = best_knn.predict(test_expressionLevel)

# Calculate the confusion matrix
confMatrix = confusion_matrix(testOutcomes, testPredictions)

# Fisher's exact test
oddsRatio, pValue = fisher_exact(confMatrix)
print("Fisher's exact test p-value:", pValue)

#Assuming you have predictions and true labels

mccPhi = matthews_corrcoef(testOutcomes, testPredictions)
print("Matthews' correlation coefficient Phi:", mccPhi)
```

```
Fisher's exact test p-value: 0.07099911582670201
Matthews' correlation coefficient Phi: -0.3611575592573076
```