Assignmen1)

Preprocessing data:

```
# pre-process the data
num_all_samples = original_data.shape[0]                                    # original number of training points
features = original_data.columns[:-1]                                 # original features
feature_sum_percentage = np.sum(original_data.iloc[:, :-1] > 0) / num_all_samples    # fraction of nonzero components for each fea
features_to_drop = features[feature_sum_percentage<0.6]           # features with less than 60% nonzero components
data_1 = original_data.drop (features_to_drop, axis=1)                 # drop those features
sample_min = data_1.min(axis=1)                                   # finding samples with zero values
data_2 = data_1[sample_min != 0]                                 # drop sample points with any zero values
SFE_data = data_2[(data_2.SFE < 35) | (data_2.SFE > 45)]           # get a subset of dataframe with condition


Y = SFE_data.SFE > 40    # if SFE > 40, Y =true , else Y=false
# so Y == True and ~Y==False which represent the labels
```

(a):

```
# Assignmnet 1- a:

split = int(0.2 * SFE_data.shape[0])
train_data , test_data = SFE_data[:split], SFE_data[split:]

X_train = train_data.iloc[:,:-1]
Y_train = train_data.SFE>40   # High SFE will be labled as true (1) and low SFE will be labeled as false (0)
# High SFE = class 1 / low SFE = class 0

X_test = test_data.iloc[:, :-1]
Y_test = test_data.SFE>40

# X_train[Y_train] returns X_train where SFE>40 is true
# X_train[~Y_train] returns X_train where SFE>40 is false
```

(b): filter method using ttest

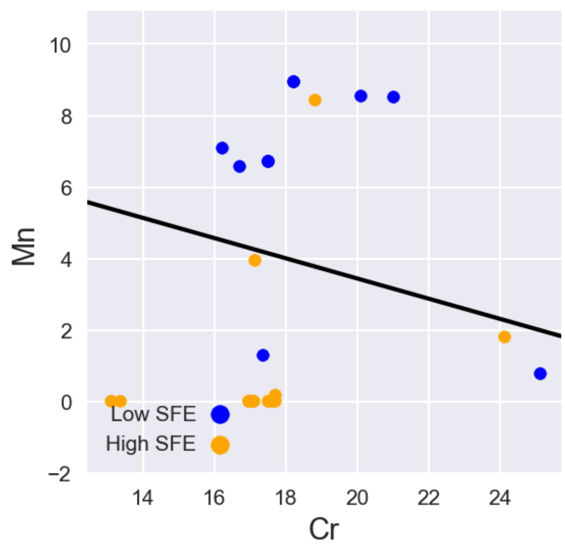
```
# Assignmnet 1- b:

ttest = scipy.stats.ttest_ind(X_train[Y_train], X_train[~Y_train])
ttest_Value = [abs(ele) for ele in ttest]   #note that P_value always >0

cols = {'t_statistics': ttest_Value[0], 'p_value': ttest_Value[1]}

df = pd.DataFrame(cols, columns = cols.keys(),
                  index=['C' , 'N' ,  'Mn' , 'Si', 'Cr', 'Ni' , 'Fe'])

# df_org = pd.DataFrame(cols, columns = ['t_statistics','p_value'],
#                  index=['C' , 'N' ,  'Mn' , 'Si', 'Cr', 'Ni' , 'Fe'])   # is used for assignmnet2 ( not sorted)

df.sort_values(by=['t_statistics'], inplace=True, ascending=False)
df
```

	t_statistics	p_value
Cr	4.917379	0.000064
Mn	4.886601	0.000069
N	3.287128	0.003363
C	2.435094	0.023452
Fe	1.275129	0.215565
Si	1.252222	0.223638
Ni	0.914754	0.370239

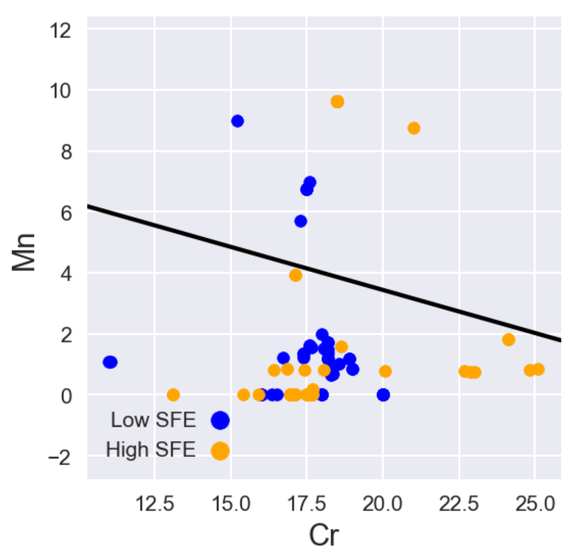
(c):

```
# Assignmnet 1- c:

var1 = 'Cr'
var2 = 'Mn'

Xtrain_var1_var2 = train_data [[var1,var2]]
classifier_2 = LDA()
classifier_2.fit(Xtrain_var1_var2, Y_train.astype(int))

a_2 = classifier_2.coef_[0]
b_2 = classifier_2.intercept_[0]
```

```
# plot classifier
Plot_Classifier (Xtrain_var1_var2, Y_train, var1, var2, a_2, b_2, 'train')
```



Estimate the classification error using the training and test data. What do you observe?



We Observe that the although the classifier works not bad on the training data, it does not classify the test data well. This shows that the 0.2% of the data as train can not represent the whole data.

Error on train and test:

```
# Estimate the classification error using the training and test data

y_pred_train = classifier_2.predict(Xtrain_var1_var2)
error_train = np.mean(y_pred_train != Y_train.astype(int))

metrics.accuracy_score()

y_pred_test = classifier_2.predict(Xtest_var1_var2)
y_test = Y_test.astype(int).values

error_test = np.mean(y_pred_test != y_test)

print(error_train)
print (error_test)
```

```
0.125
0.5454545454545454
```

(d): Repeat for the top three, four, and five predictors. Estimate the errors on the training and testing data (there is no need to plot the classifiers). How do the training and testing errors behave?

```
# Assignmnet 1- d:
# generallize the method to a function that compute classifier and classification error for the top 2,3,4, and 5 predictors

def get_classifier(train_data, Y_train, feature):
    xtrain = train_data[feature]
    ytrain = Y_train.astype(int)
    classifier = LDA()
    classifier.fit(xtrain, ytrain)
    return classifier

def get_classification_error(train_data, Y_train, X_test, Y_test):
    idx = 2
    test_error_list = []
    train_error_list = []
    columns = df.index

    while (idx <= 5):    # idx <= len(columns) if you desire to reach top 7 (all features)
        feature = columns[:idx]

        classifier = get_classifier(train_data, Y_train, feature)

        xtrain = train_data[feature]
        y_pred_train = classifier.predict(xtrain)
        ytrain = Y_train.astype(int)
        error_train = np.mean(y_pred_train != ytrain)
        train_error_list.append(error_train)

        xtest = X_test [feature]
        y_pred_test = classifier.predict(xtest)
        ytest = Y_test.astype(int)
        error_test = np.mean(y_pred_test != ytest)
        test_error_list.append(error_test)
        idx+=1

    return  train_error_list, test_error_list
```

```
train_error_list, test_error_list = get_classification_error (train_data, Y_train, X_test, Y_test)

print ( 'train_error_list= ', np.round(train_error_list, 4))
print ('test_error_list= ', np.round(test_error_list, 4))
```

```
train_error_list=  [0.125 0.125 0.125 0.125]
test_error_list=  [0.5455 0.5455 0.5152 0.101 ]
```

The error does not improve on the training data and this shows that number of samples for training data is too small. The error on test data shows that it the classifier almost classifies randomly ~0.5. The last error suddenly has reduced which shows the combination of 5 features contribute to classifier. It also shows that maybe the fifth feature itself has significant impact on classifier.

Assignment2:

(a): Wrapper method :

1) Exhaustive search

```
## Assignmnet 2: Classification using wrapper feature selection

# 1) exhausive search (for 1 to 5 variables)

def get_combination(iterable, k_features):
    comb_list = []

    for subset in itertools.combinations(iterable, k_features):
        comb_list.append(subset)
    comb_feature = [list(comb_list[i]) for i in range(len(comb_list))]
    return comb_feature

def get_classifier(train_data, feature):
    xtrain = train_data[feature]
    Y_train = train_data.SFE > 40
    ytrain = Y_train.astype(int)
    classifier = LDA()
    classifier.fit(xtrain, ytrain)
    return classifier

def get_accuracy(data, feature, classifier):
    x = data[feature]
    y1 = data.SFE > 40
    y_org = y1.astype(int)
    y_pred = classifier.predict(x)
    acc = metrics.accuracy_score(y_org, y_pred)
    return acc
```

```
def ExhaustiveSearchFeatureSelection(train_data, test_data, k_features):
    df_acc = pd.DataFrame(columns=['features', 'accuracy_train', 'accuracy_test'])

    X_train = train_data.iloc[:, :-1]
    iterable = X_train.columns.values
    comb_feature = get_combination(iterable, k_features)

    for feature in comb_feature:
        classifier = get_classifier(train_data, feature)
        acc_train = get_accuracy(train_data, feature, classifier)
        #acc = np.round(acc, 4)
        df_acc = df_acc.append({'features': feature, 'accuracy_train': acc_train}, ignore_index=True)

    train_acc = max(df_acc.accuracy_train)
    max_acc_features = df_acc.loc[df_acc['accuracy_train'] == train_acc, 'features'].tolist()
    selected_features = max_acc_features[0]

    #getting accuracy on test using the classifier of selected features
    cls = get_classifier(train_data, selected_features)
    test_acc = get_accuracy(test_data,selected_features,cls)

    return train_acc, test_acc, selected_features
```

```python
# get lists of the results
def get_featuresets(train_data, min_feature, max_feature):
    train_acc_list = []
    test_acc_list =[]
    featuresets_list = []

    for i in range(min_feature, max_feature + 1):
        train_acc, test_acc, selected_features = ExhaustiveSearchFeatureSelection(train_data, test_data, k_features=i)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        featuresets_list.append(selected_features)

    return train_acc_list, test_acc_list, featuresets_list


train_acc_list, test_acc_list, featuresets_list = get_featuresets(train_data, min_feature=1, max_feature=5)


efs_cols = {'EFS/n_variable':[1,2,3,4,5], 'selected_features': featuresets_list,
            'error on train': [1-i for i in train_acc_list], 'error on test':[1- i for i in test_acc_list]}

efs_DataFrame = pd.DataFrame(efs_cols)
```

```python
efs_DataFrame
```

	EFS/n_variable	selected_features	error on train	error on test
0	1	[Ni]	0.125000	0.151515
1	2	[C, Ni]	0.125000	0.191919
2	3	[C, N, Ni]	0.125000	0.303030
3	4	[C, N, Fe, Cr]	0.083333	0.101010
4	5	[C, N, Fe, Si, Cr]	0.083333	0.111111

2) Sequential forward search

```
def SequentialForwardSearch(train_data, test_data, max_feat):
    feat_list = []
    train_acc_list = []
    test_acc_list = []
    iterable = X_train.columns.values

    efs= ExhaustiveSearchFeatureSelection(train_data, test_data, k_features=1)
    train_acc_list.append(efs[0])
    test_acc_list.append(efs[1])
    selected_features = efs[2]

    feat_list.append(selected_features)
    initial_list = selected_features  # selected_features will be overwritten in the while loop

    counter = 0
    while (counter < max_feat-1):
        comb_feature = get_combination([i for i in iterable if not i in initial_list], k_features=1)
        new_set = [initial_list + i for i in comb_feature]

        df_acc = pd.DataFrame(columns=['features', 'accuracy'])
        for set in new_set:
            classifier = get_classifier(train_data, feature= set)
            acc = get_accuracy(train_data, feature=set, classifier = classifier)
            acc = np.round(acc, 4)
            df_acc = df_acc.append({'features': set, 'accuracy': acc}, ignore_index=True)

        acc_max = max(df_acc.accuracy)
        max_acc_features = df_acc.loc[df_acc['accuracy'] == acc_max, 'features'].tolist()
        selected_features = max_acc_features[0]
        feat_list.append(selected_features)
        train_acc_list.append(acc_max)
        initial_list = selected_features

        # getting accuracy on test using the classifier of selected features
        cls = get_classifier(train_data, selected_features)
        test_acc = get_accuracy(test_data, selected_features, cls)
        test_acc_list.append(test_acc)

        counter += 1

    return train_acc_list, test_acc_list, feat_list
```

```
train_acc_list, test_acc_list, feat_list = SequentialForwardSearch (train_data, test_data, max_feat=5)

sfs_cols = {'SFS/n_variable':[1,2,3,4,5], 'selected_features': feat_list,
            'error on train': [1-i for i in train_acc_list], 'error on test':[1- i for i in test_acc_list]}

sfs_DataFrame = pd.DataFrame(sfs_cols)
sfs_DataFrame
```

	SFS/n_variable	selected_features	error on train	error on test
0	1	[Ni]	0.125	0.151515
1	2	[Ni, C]	0.125	0.191919
2	3	[Ni, C, N]	0.125	0.303030
3	4	[Ni, C, N, Fe]	0.125	0.393939
4	5	[Ni, C, N, Fe, Mn]	0.125	0.222222

Conclusion:

The filtering method shows that ‘Cr’ is the best predictor based on ttest however through the exhaustive search ‘Ni’ is the best feature for 1-feature selection. The contradiction is because ttest does not depends on the classifier. Filtering is specifically unreliable if number of samples for training is low.

The exhaustive search shows a better result on the test sets specifically compare to sequential forward test. This is because in the **SFS** the features will be fixed in each search sequence and this will cause **finite horizon problem.** For example although ‘Ni’ is the best feature for 1-feature selection, it has not been selected in the 4-variable combination. This shows the privilege of efs over sfs because the features do not get fixed in exhaustive search. although the **exhaustive search** in general shows a better result it is **computationally expensive specially** when the data size increase.