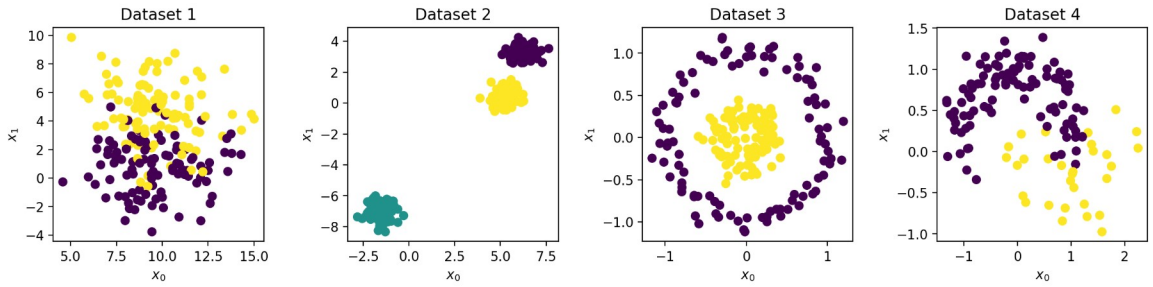
Classification - imahmood7_HW07

Data and Package Import

In [3]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import pylab as plt
```

In [4]:
```
from sklearn.datasets import make_blobs, make_moons, make_circles
np.random.seed(4)

noisiness = 1

X_blob, y_blob = make_blobs(n_samples = 200, centers = 2, cluster_std =
2 * noisiness, n_features = 2)

X_mc, y_mc = make_blobs(n_samples = 200, centers = 3, cluster_std = 0.5
* noisiness, n_features = 2)

X_circles, y_circles = make_circles(n_samples = 200, factor = 0.3, nois
e = 0.1 * noisiness)

X_moons, y_moons = make_moons(n_samples = 200, noise = 0.25 * noisines
s)

N_include = 30
idxs = []
Ni = 0
for i, yi in enumerate(y_moons):
    if yi == 1 and Ni < N_include:
        idxs.append(i)
        Ni += 1
    elif yi == 0:
        idxs.append(i)

y_moons = y_moons[idxs]
X_moons = X_moons[idxs]

fig, axes = plt.subplots(1, 4, figsize = (15, 3), dpi = 200)

all_datasets = [[X_blob, y_blob], [X_mc, y_mc], [X_circles, y_circles],
[X_moons, y_moons]]

labels = ['Dataset 1', 'Dataset 2', 'Dataset 3', 'Dataset 4']
for i, Xy_i in enumerate(all_datasets):
    Xi, yi = Xy_i
    axes[i].scatter(Xi[:, 0], Xi[:, 1], c = yi)
    axes[i].set_title(labels[i])
    axes[i].set_xlabel('$x_0$')
    axes[i].set_ylabel('$x_1$')

fig.subplots_adjust(wspace = 0.4);
```



In [5]:
```
df = pd.read_csv('data/perovskite_data.csv')
X_perov = df[['nA', 'nB', 'nX', 'rA (Ang)', 'rB (Ang)', 'rX (Ang)', 't
', 'tau']].values
y_perov = df['exp_label'].values
```

1. k-nearest Neighbors Model

1-NN

Calculate the accuracy of a 1-nearest Neighbors model for the training data.

A 1-nearest Neighbors model considers a point as its own nearest neighbors.

Hint: the block below is not a code block.

The accuracy is 1, since for the training data if we say the point itself is the nearest neighbor, then it will just make the model based on exactly the training data, making the accuracy 1, which is up for suspicion.

Will this be a reliable indicator of its accuracy for testing data?

Briefly explain your answer.

No, anytime we have an accuracy of 1, we should be skeptical when applying it to the testing data, and in this case it is not which we will find out if we cross-validate it. It has overfitted for the training data.

Weighted Neighbors Classification

Instead of selecting the k-nearest neighbors to vote, we could design an algorithm where all neigbors get to vote, but their vote is weighted inversely to their distance from the point of interest:

$y_i = \sum_j y_j/(||x_i - x_j||)$

where j is an index over all training points.

The class will be assigned as follows:

- class 1 if $y_i \geq 0$
- class -1 if $y_i < 0$

In [6]:
```
#This function finds the distance between two points using the L2 norm
def distance(x1, x2):
    return np.linalg.norm(x1 - x2, 2)
```

In [7]:
```
#This function finds the neighbors for point x
def get_neighbor(x, x_list): #point x, and x_list is a list of training
points
    dist_pairs = []
    for i, xi in enumerate(x_list):
        dist = distance(x, xi)
        dist_pairs.append([dist, i]) #every run appends the distance wi
th that index
    return dist_pairs
```

Write a function that assigns a class to a point.

The function should take the followings as arguments:

- a single point `x`
- a list of training points `x_list`
- a list of training labels `y_list`

You may want to use functions above. You will also need to add a statement to avoid dividing by zero if the point is in the training set. If the distance between 2 points is zero, then the label from the same point in the training set should be used (e.g. if $x_i = x_j$ then $y_i = y_j$).

In [8]:
```
def assign_class(x, x_list, y_list):
    yi = 0
    for i, xj in enumerate(x_list):
        x_xj_dist = distance(x,xj)
        if x_xj_dist == 0:
            yi_now = y_list[i]
        yi_now = y_list[i]/x_xj_dist
        yi += yi_now

    if yi < 0:
        assignment = -1
    if yi >= 0:
        assignment = 1

    return assignment
```

Write a function that returns the prediction for a given list of testing points.

The function should take the followings as arguments:

- a list of testing points `X`
- a list of training points `X_train`
- a list of training labels `y_train`

In [9]:
```
def weighted_neighbors(X, X_train, y_train):
    y_hat = []
    for x in X:
        y_now = assign_class(x, X_train, y_train)
        y_hat.append(y_now)
    return y_hat
```

Train the model for the perovskite dataset using a random selection of 75% of the data as training data.

In [10]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_perov, y_perov, t
est_size = 0.75)
y_hat = weighted_neighbors(X_perov, X_train, y_train) #In office hours
Sihoon mentioned
# to use the entire X_perov when doing the y_hat
```

```
<ipython-input-8-fe3ac42a026f>:7: RuntimeWarning: divide by zero enco
untered in true_divide
  yi_now = y_list[i]/x_xj_dist
```

Compute the accuracy and precision of the prediction.

In [11]:
```
# def acc_prec_recall(y_model, y_actual): I had tried defining a functi
on myself, but
# in office hours it did not work well for me so I will be using the bu
ilt in function
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score

print("The accuracy is")
print(accuracy_score(y_perov,y_hat))
print("The precision is")
print(precision_score(y_perov,y_hat))
```

```
The accuracy is
0.9149305555555556
The precision is
0.8771428571428571
```

In [ ]:
```

```

Train a 5-NN model using the same training data.

In [12]:
```
# def kNN(X, k, X_train, y_train):
#     y_out = []  #this is analogous to the y_hat in the weighted neigh
bors func
#     for xi in X:
#         y_out.append(assign_class(xi, X_train, y_train, k))
#     y_out = np.array(y_out)
#     return y_out

# y_hat_kNN = kNN(X_perov,5,X_train,y_train)

from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_perov,y_perov)
y_hat_knn = knn.predict(X_perov)
```

Compute the accuracy and precision.

In [13]:
```
print("The accuracy is")
print(accuracy_score(y_perov,y_hat_knn))
print("The precision is")
print(precision_score(y_perov,y_hat_knn))
```

```
The accuracy is
0.9444444444444444
The precision is
0.934984520123839
```

2. Multi-dimensional Classification

Simple logistic regression

Train a logistic regression model using all columns except the `tau` column of the perovskite dataset.

You may use some functions that have been already built in the previous assignments.

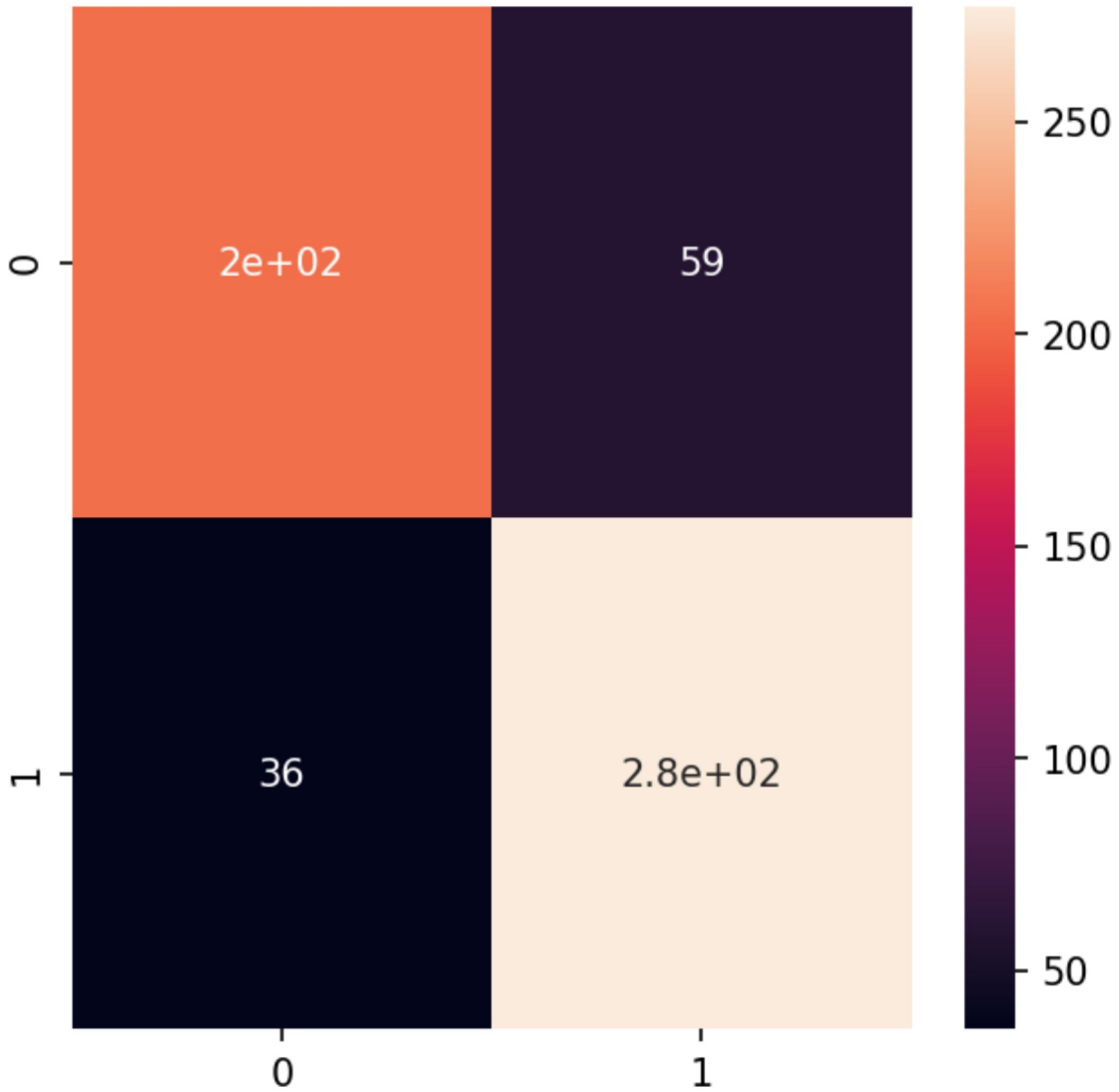
In [14]:
```
from scipy.optimize import minimize
def add_intercept(X):
    intercept = np.ones((X.shape[0],1))
    X_intercept = np.append(intercept,X,1)
    return X_intercept

def log_reg(w, X, y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept, w)
    exp_yXb = np.exp(-y * Xb)
    loss = sum(np.log(1 + exp_yXb))
    return loss
w = [1, 2, 3, 4, 5, 6 ,7, 8]

X_cut = np.delete(X_perov,7,axis = 1)


result = minimize(log_reg,w, args = (X_cut,y_perov))
w_logit = result.x


def linear_classifier(X, w):
    X_intercept = add_intercept(X)
    p = np.dot(X_intercept, w)
    return p > 0

prediction = linear_classifier(X_cut,w_logit)

new_predict = np.zeros(np.size(prediction))
for i, x in enumerate(new_predict): #need to convert the prediction mat
rix from true/false to -1 and 1 for the classes
    if prediction[i]:
        new_predict[i] = 1
    else:
        new_predict[i] = -1
```

Plot the confusion matrix.

In [15]:
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
import seaborn as sns


cm = confusion_matrix(y_perov,new_predict)
print(cm)
fig, ax = plt.subplots(figsize = (5,5),dpi = 150)
sns.heatmap(cm,annot = True)
```

```
[[204  59]
 [ 36 277]]
```

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x251e19b7f70>



Compute the accuracy, precision and recall.

In [16]:
```
from sklearn.metrics import recall_score
print("The accuracy is")
print(accuracy_score(y_perov,new_predict))
print("The precision is")
print(precision_score(y_perov,new_predict))
print("The recall is")
print(recall_score(y_perov,new_predict))
```

```
The accuracy is
0.8350694444444444
The precision is
0.8244047619047619
The recall is
0.8849840255591054
```

6745 Only: Customizing non-linear boundaries

In this problem, you will create a single custom feature that improves the separation performance as much as possible.

Plot the `y_perov` as a function of `rA (Ang)` and `rB (Ang)` .

N/A. I am not taking 6745.

Build a baseline model based on logistic regression.

Report the accuracy and precision of the baseline model.

N/A

Plot the prediction of the baseline model.

N/A

Create a new feature based on a non-linear combination of `rA (Ang)` and `rB (Ang)` .

Plot the new feature as a function of `rA (Ang)` .

N/A

Build a new model that includes `rA (Ang)` , `rB (Ang)` and your new feature.

Report the accuracy and precision.

N/A

Plot the result of your new model.

N/A

Briefly explain how you decided on the feature.

N/A

3. Comparison of Classification Model

In this problem, you will compare the classification performance of three different models using the perovskite dataset.

Choose three different classification models and import them.

These could be models discussed in the lectures, or others that you have learned about elsewhere.

In [52]:
```
#choosing SVC, choose an arbitrary hyperparam for each
from sklearn.svm import SVC
svc = SVC(kernel = 'rbf',gamma = 100, C = 1000)
svc.fit(X_perov, y_perov)
y_predict_svc = svc.predict(X_perov)
print('The accuracy of SVC before optimization is:{}'.format(accuracy_s
core(y_perov,y_predict)))

#choosing Decision Tree
from sklearn.tree import DecisionTreeClassifier
tree=DecisionTreeClassifier()
tree.fit(X_perov,y_perov)
y_predict = tree.predict(X_perov)
print('The accuracy of Decision Tree before optimization is:{}'.format
(accuracy_score(y_perov,y_predict)))


#choosing kNN
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 20)
knn.fit(X_perov, y_perov)
y_predict = knn.predict(X_perov)
print('The accuracy of kNN before optimization is:{}'.format(accuracy_s
core(y_perov,y_predict)))
```

```
The accuracy of SVC before optimization is:0.9236111111111112
The accuracy of Decision Tree before optimization is:0.99652777777777
78
The accuracy of kNN before optimization is:0.9236111111111112
```

Make a hyperparameter grid for each model.

You should optimize at least one hyperparameter for each model.

In [32]:
```
#param for svc
gammas = 1./np.linspace(5,50,10)
Cs = 1./np.array([1,2,10,50,100])
param_grid_svc = {'gamma': gammas,'C': Cs}


#param for decision tree
depths = np.array([1,2,3,4,5,6,7,8,9,10])
param_grid_tree={'max_depth':depths}

#param for k neighbor
depths = np.array([1,2,3,4,5,6,7,8,9,10])
param_grid_knn = {'n_neighbors': depths}
```

Optimize hyperparameters.

First, you select a validation set using hold-out (`train_test_split`). Optimize hyperparameters using `GridSearchCV` on the training set.

In [49]:
```
from sklearn.model_selection import GridSearchCV
# svm hyperparameters
X_train, X_test, y_train, y_test = train_test_split(X_perov, y_perov, t
est_size = 0.75)

svcg = SVC(kernel = 'rbf')
svm_search = GridSearchCV(svcg,param_grid_svc,cv=3)
svm_search.fit(X_train,y_train)
opt_C = svm_search.best_estimator_.C
opt_gamma = svm_search.best_estimator_.gamma
print('SVC:----------')
print('Optimized C for SVC:{}'.format(opt_C))
print('Optimized Gamma for SVC:{}'.format(opt_gamma))
print('With Accuracy of:{}'.format(svm_search.best_score_))

#decision tree
dtree = DecisionTreeClassifier()
dt_search = GridSearchCV(dtree, param_grid_tree, cv=3)
dt_search.fit(X_train,y_train)
#opt_neighbor = dt_search.best_estimator_.n_neighbors
dt_search.best_estimator_, dt_search.best_score_
print("Decision Tree:------------")
print('Optimized Depth for Decistion Tree:{}'.format(dt_search.best_est
imator_.max_depth),'With Accuracy of:{}'.format(dt_search.best_score_))

#knn
knng = KNeighborsClassifier()
knn_search = GridSearchCV(knng,param_grid_knn,cv=3)
knn_search.fit(X_train,y_train)
knn_search.best_estimator_, dt_search.best_score_
print("K Nearest Neighbors:-------------")
print('Optimized Number of Neighbors for kNN:{}'.format(dt_search.best_
estimator_.max_depth),'With Accuracy of:{}'.format(dt_search.best_score
_))
```

```
SVC:----------
Optimized C for SVC:1.0
Optimized Gamma for SVC:0.2
With Accuracy of:0.8125
Decision Tree:------------
Optimized Depth for Decistion Tree:1 With Accuracy of:0.8958333333333
334
K Nearest Neighbors:-------------
Optimized Number of Neighbors for kNN:1 With Accuracy of:0.8958333333
333334
```

Compare the accuracy by predicting the results of the validation set.

This was calculated in the code block above this one, as you can see the accuracy for each cross-validated model is lower than it was for the arbitrarily made ones.

Briefly describe your conclusions based on the results.

I think the accuracy score of my models are higher before cross validating it with the train_test_split and then optimizing the hyperparameters, and with a high accuracy, like the one seen close to 99% is a cause for skepticism. Therefore this shows that it is necessary to validate a model to check if that is a real accuracy metric. This means the models were originally overfitted, and then after doing the test_train_split and GridSearchCV, they were optimized and corrected.

In [ ]: