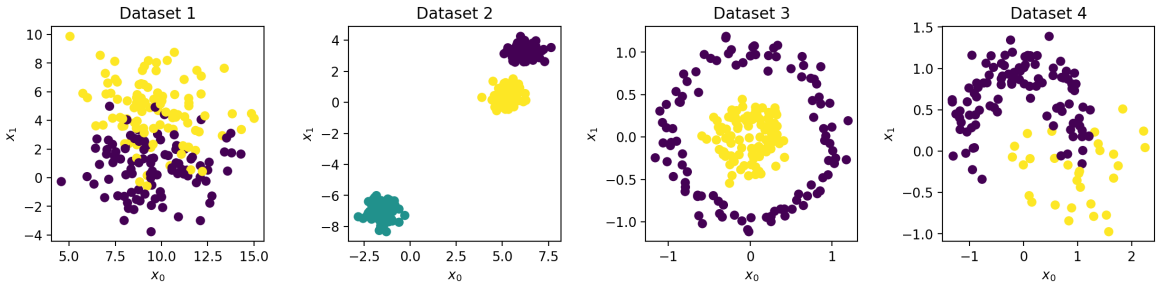
Classification - Assignment 2

Data and Package Import

In [172]:

```
%matplotlib inline
import numpy as np
import pandas as pd
import pylab as plt

clrs = np.array(['#003057', '#EAAA00', '#4B8B9B', '#B3A369', '#377117', '#187
```

In [173]:
```
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 * n

X_mc, y_mc = make_blobs(n_samples = 200, centers = 3, cluster_std = 0.5 * noi

X_circles, y_circles = make_circles(n_samples = 200, factor = 0.3, noise = 0.

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

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_mc

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 [174]:
```
df = pd.read_csv('data/perovskite_data.csv')
X_perov = df[['nA', 'nB', 'nX', 'rA (Ang)', 'rB (Ang)', 'rX (Ang)', 't', 'tau
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 for this type of nearest neighbors model would be 100% because the algorithm picks the nearest neighbor for each data point. The closest neighbor for each point is the point itself, so we always get 100% accuracy since we are testing on the exact same data. It memorizes the training set and always makes the same prediction.

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

Briefly explain your answer.

This is not reliable for the testing data because it has memorized the training data. K = 1 generally implies overfitting.

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 [175]:
```
def distance(x1, x2):
    return np.linalg.norm(x1 - x2, 2)
```

In [176]:
```
def get_neighbor(x, x_list):
    dist_pairs = []
    for i, xi in enumerate(x_list):
        dist = distance(x, xi)
        dist_pairs.append([dist, i])
    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 [178]:
```
def assign_class(x, x_list, y_list):
    y_curr = 0

    for i, xj in enumerate(x_list):
        xDist = distance(x, xj)
        if xDist == 0:
            y_new = y_list[i]
        y_new = y_list[i]/xDist
        y_curr += y_new

    if y_curr < 0:
        assignment = -1
    if y_curr >= 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 [179]:
```
def weighted_neighbors(X, X_train, y_train):
    y_out = []
    for xi in X:
        y_out.append(assign_class(xi, X_train, y_train))
    y_out = np.array(y_out)
    return y_out
```

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

In [180]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.75)

y_weighted = weighted_neighbors(X_perov, X_train, y_train)
```

```
<ipython-input-178-5e087a13e0dc>:8: RuntimeWarning: divide by zero encounte
red in true_divide
  y_new = y_list[i]/xDist
```

Compute the accuracy and precision of the prediction.

In [181]:
```
from sklearn.metrics import accuracy_score, precision_score

print(accuracy_score(y_perov, y_weighted))
print(precision_score(y_perov,y_weighted))
```

```
0.9149305555555556
0.8771428571428571
```

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

In [182]:
```
from sklearn.neighbors import KNeighborsClassifier

t_KNN = KNeighborsClassifier(n_neighbors = 5)
t_KNN.fit(X_perov, y_perov)
y_pred = t_KNN.predict(X_perov)
```

Compute the accuracy and precision.

In [183]:
```
print(accuracy_score(y_perov, y_pred))
print(precision_score(y_perov,y_pred))
```

```
0.9444444444444444
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 [184]:
```
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 linear_classifier(X,w):
    X_int = add_intercept(X)
    p = np.dot(X_int, w)
    return p > 0

def softmax(w,X,y):
    X_int = add_intercept(X)
    Xb = np.dot(X_int, w)
    exp_yXb = np.exp(-y*Xb)
    loss = sum(np.log(1 + exp_yXb))
    return loss

newXperov = np.delete(X_perov, 7, axis = 1)
w = [-1, 0.44, -.81, 0.1, 2, 0.3, -4, 2]

#resultLogReg = softmax(w, newXperov, y_perov)
new_resultLogReg = minimize(softmax, w, args = (newXperov,y_perov))
#print(resultLogReg)
##print(new_resultLogReg)

w_log = new_resultLogReg.x
loss_log = softmax(w_log, newXperov, y_perov)
print(loss_log)

prediction = linear_classifier(newXperov, w_log)

prediction_fixed = np.zeros(np.size(prediction))

for i, x in enumerate(prediction_fixed):
    if prediction[i]:
        prediction_fixed[i] = 1
    else:
        prediction_fixed[i] = -1
```

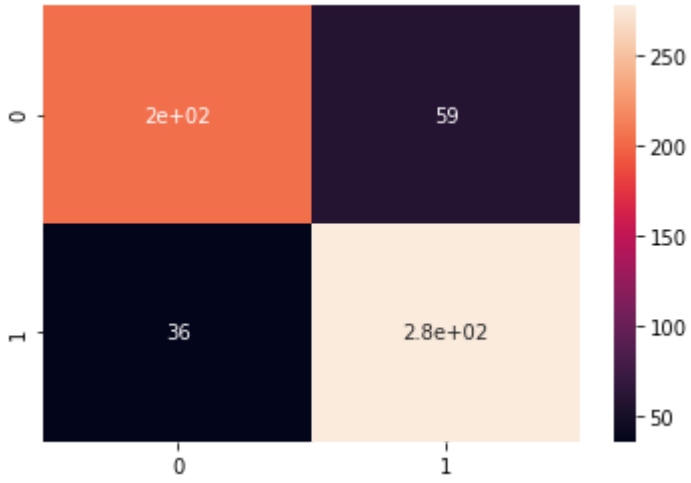
```
243.53583886314038
```

Plot the confusion matrix.

In [185]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y_perov, prediction_fixed)
print(cm)
sns.heatmap(cm, annot = True)
```

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

Out[185]: <matplotlib.axes._subplots.AxesSubplot at 0x24b35782f10>



Compute the accuracy, precision and recall.

In [186]:
```
from sklearn.metrics import recall_score

print(accuracy_score(y_perov, prediction_fixed))
print(precision_score(y_perov, prediction_fixed, average = 'micro'))
print(recall_score(y_perov, prediction_fixed, average = 'micro'))
```

```
0.8350694444444444
0.8350694444444444
0.8350694444444444
```

6745 Only: Customizing non-linear boundaries --- I am in 4745

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)`.

In [ ]:
```
n/a
```

Build a baseline model based on logistic regression.

Report the accuracy and precision of the baseline model.

In [ ]:
```
n/a
```

Plot the prediction of the baseline model.

In [ ]:
```
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)`.

In [ ]:
```
n/a
```

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

Report the accuracy and precision.

In [ ]:
```
n/a
```

Plot the result of your new model.

In [ ]:
```
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 [187]:
```
X = X_perov
y = y_perov
```

In [188]:
```
# Model 1: SVC
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Model 2: KNN
from sklearn.neighbors import KNeighborsClassifier

# Model 3: Decision Tree Classifer
from sklearn.tree import DecisionTreeClassifier
```

Make a hyperparameter grid for each model.

You should optimize at least one hyperparameter for each model.

In [192]:
```
# Model 1: SVC

alphas = [1,2,10,50,100]
cS = np.divide(1,alphas)
sigmas = np.linspace(5,50,10)
gammas = 1./sigmas

param_gridSVC = {'gamma': gammas, 'C': cS}

# Model 2: KNN
neighborsVec = [5, 10, 20, 25, 30]
param_gridKNN = {'n_neighbors': neighborsVec}

# Model 3: Decision Tree Classifer
param_gridDTC = {'max_depth':[1, 2, 3, 4, 5, 6]}
```

Optimize hyperparameters.

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

In [193]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.33)

# Model 1: SVC
svc = SVC(kernel = 'rbf')
svm_search = GridSearchCV(svc, param_gridSVC, cv = 3)
svm_search.fit(X_train,y_train)
opt_C = svm_search.best_estimator_.C
opt_gamma = svm_search.best_estimator_.gamma

print('Model 1 SVC')
print('Optimal C: {}'.format(opt_C))
print('Optimal gamma: {}'.format(opt_gamma))

# Model 2: KNN
k_range = neighborsVec
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
knn_search = GridSearchCV(knn, param_gridKNN, cv = 3)
knn_search.fit(X_train,y_train)
optN = knn_search.best_estimator_.n_neighbors

print('')
print('Model 2 KNN')
print('Optimal N: {}'.format(optN))

# Model 3: Decision Tree Classifer
dtc = DecisionTreeClassifier()
dtc_search = GridSearchCV(dtc, param_gridDTC, cv = 3)
dtc_search.fit(X_train, y_train)
opt_depth = dtc_search.best_estimator_.max_depth

print('')
print('Model 3 DTC')
print('Optimal Depth: {}'.format(opt_depth))
```

```
Model 1 SVC
Optimal C: 0.5
Optimal gamma: 0.03333333333333333

Model 2 KNN
Optimal N: 5

Model 3 DTC
Optimal Depth: 1
```

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

In [194]:
```
print('Model 1 SVC Accuracy: {}'.format(accuracy_score(y_perov, svm_search.be
print('')
print('Model 2 KNN Accuracy: {}'.format(accuracy_score(y_perov, knn_search.be
print('')
print('Model 3 DTC Accuracy: {}'.format(accuracy_score(y_perov, dtc_search.be
```

```
Model 1 SVC Accuracy: 0.9010416666666666

Model 2 KNN Accuracy: 0.9322916666666666

Model 3 DTC Accuracy: 0.9166666666666666
```

Briefly describe your conclusions based on the results.

The most accurate model (93.22%) for this dataset is K nearest neighbors with 5 neighbors.

In [ ]:
```

```