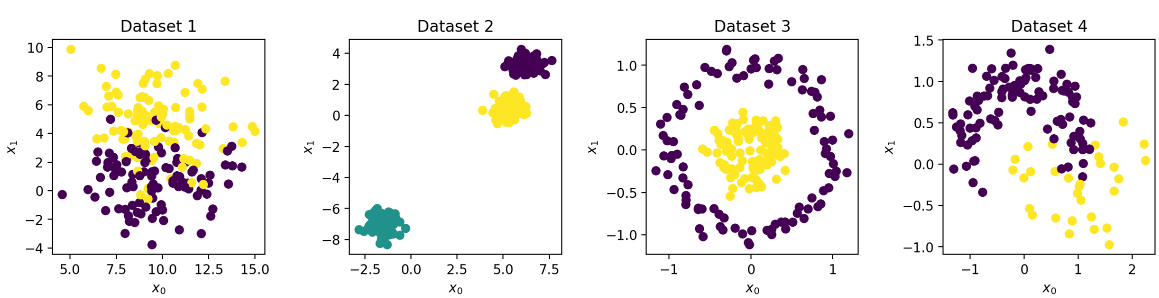
Classification - Assignment 1

Data and Package Import

In [8]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import pylab as plt
```

In [9]:
```
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 * r

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);
```



1. Discrimination Lines

Derive the equation for the line that discriminates between the two classes.

Consider a model of the form:

$\bar{\bar{X}}\vec{w} > 0$ if $y_i = 1$ (class 1)

$\bar{\bar{X}}\vec{w} < 0$ if $y_i = -1$ (class 2)

where $\bar{\bar{X}} = [\overrightarrow{x_0}, \overrightarrow{x_1}, \vec{1}]$ and $\vec{w} = [w_0, w_1, w_2]$.

The equation should be in the form of $x_1 = f(x_0)$. Show your work, and/or explain the process you used to arrive at the answer.

X * w = (x0 , x1 , 1) * (w0 , w1 , w2)

x0*w0* + *x1*w1 + w2 = 0

x1 = f(x0) = (w0/w1) * x0 - w2/w1

where -w0/w1 is slope and the intercept will be -w2/w1

Derive the discrimination line for a related non-linear model

In this case, consider a model defined by:

$y_i = w_0x_0 + w_1x_1 + w_2(x_0^2 + x_1^2)$

where the model predicts class 1 if $y_i > 0$ and predicts class 2 if $y_i \leq 0$.

The equation should be in the form of $x_1 = f(x_0)$. Show your work, and/or explain the process you used to arrive at the answer.

where X = (x0 , x1 , x0^2 + x1^2) and w = (w0 , w1 , w2)

X * w = (x0 , x1 , x0^2 + x1^2) * (w0 , w1 , w2)

x0*w0* + *x1*w1 + w2(x0^2 + x1^2) = 0

x1 = f(x0) = (w0/w1) * x0 - (w2/w1)*(x0^2 + x1^2)

where -w0/w1 is slope and the intercept will be (-w2/w1)*(x0^2 + x1^2)

Briefly describe the nature of this boundary.

What is the shape of the boundary? Is it linear or non-linear?

The boundary for the linear model is linear, while the discrimination line for the non-linear will be non-linear.

2. Assessing Loss Functions

In [10]:
```
def add_intercept(X):
    intercept = np.ones((X.shape[0], 1))
    X_intercept = np.append(intercept, X, 1)
    return X_intercept
```

In [11]:
```
def linear_classifier(X, w):
    X_intercept = add_intercept(X)
    p = np.dot(X_intercept, w)
    return p > 0
```

Write a function that computes the loss function for the perceptron model.

The function should take the followings as arguments:

- weight vector w
- the feature matrix $\bar{\bar{X}}$
- the output vector $\vec{y}$

You may want to use functions above.

In [12]:
```
#Dataset 3 = circles

#w = np.array([-0.25, -0.4, -1]) #guess for weights vector
w = np.array([-10, -4, -10]) #guess for weights vector
X = X_circles
y = y_circles*2 - 1
```

In [13]:
```
def perceptron(w, X=X, y=y):
    x_int = add_intercept(X)
    x_b = np.dot(x_int, w)
    loss = sum(np.maximum(0, -y*x_b))
    return loss
```

In [14]:
```
result = perceptron(w, X, y)
print(result)
```

```
1003.5220798944558
```

Write a function that computes the loss function for the logistic regression model.

The function should take the followings as arguments:

- weight vector w
- the feature matrix $\bar{\bar{X}}$
- the output vector $\vec{y}$

You may want to use functions above.

In [15]:
```
def log_reg(w, X = X, y=y):
    x_int = add_intercept(X)
    x_b = np.dot(x_int, w)
    expOfYx_b = np.exp(-y * x_b)
    loss = sum(np.log(1 + expOfYx_b))
    return loss
```

In [16]:
```
result2 = log_reg(w, X, y)
print(result2)
```

```
1011.7851643612464
```

Minimize the both loss functions using the Dataset 3 above.

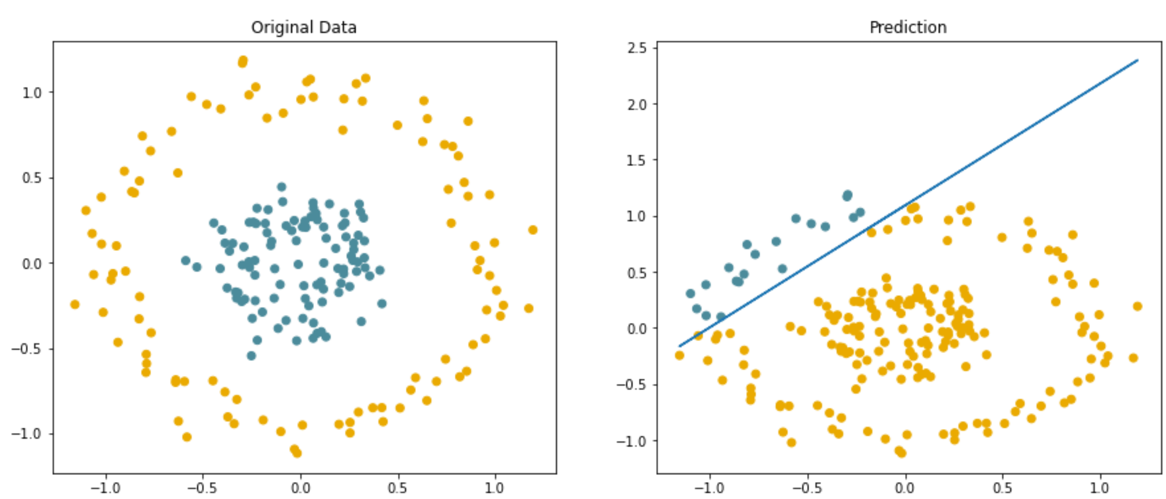
In [17]:
```
clrs = np.array(['#003057', '#EAAA00', '#4B8B9B', '#B3A369', '#377117', '#187
```

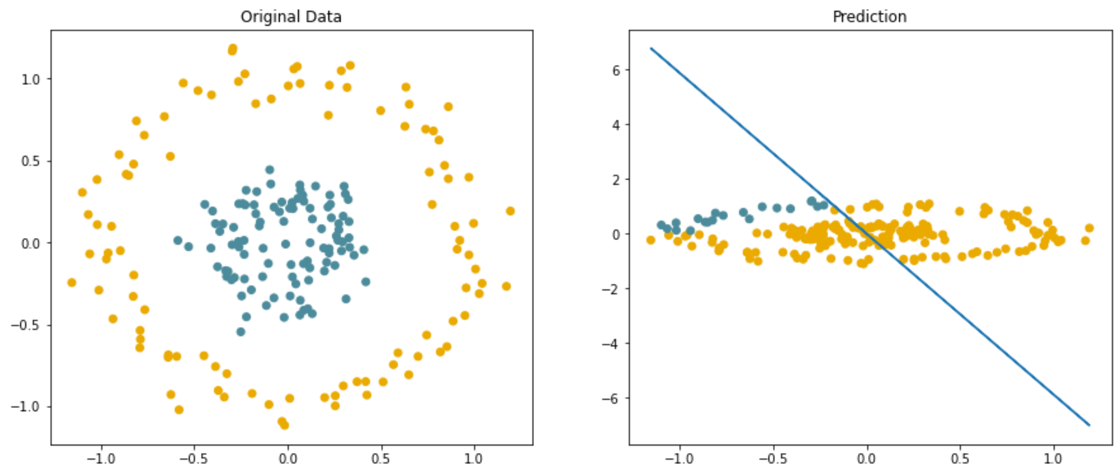
In [18]:

```
from scipy.optimize import minimize

#Perceptron / max cost loss function weights optimized
output = minimize(perceptron, w) # minimize weights vector in the max cost lo
w_perc = output.x
print("Perceptron loss optimized w:{}".format(w_perc))

#plot original vs perceptron loss function
predict_perc = linear_classifier(X, w_perc)
fig, axes = plt.subplots(1, 2, figsize = (15,6))
axes[0].scatter(X[:,0], X[:,1], c = clrs[y_circles + 1])
axes[1].scatter(X[:,0], X[:,1], c = clrs[predict_perc + 1])
m = -w_perc[1] / w_perc[2]
b = -w_perc[0] / w_perc[2]
axes[1].plot(X[:,0], m*X[:,0] + b, ls = '-')
axes[0].set_title('Original Data')
axes[1].set_title('Prediction')

#Logisitic regression loss function weights optimized
output2 = minimize(log_reg, w) # minimize weights vector in the log reg loss
w_log = output2.x
print("Logistic regression loss optimized w:{}".format(w_log))

#plot original vs logisitic regression loss function
predict_log = linear_classifier(X, w_log)
fig, axes = plt.subplots(1, 2, figsize = (15,6))
axes[0].scatter(X[:,0], X[:,1], c = clrs[y_circles + 1])
axes[1].scatter(X[:,0], X[:,1], c = clrs[predict_perc + 1])
m = -w_log[1] / w_log[2]
b = -w_log[0] / w_log[2]
axes[1].plot(X[:,0], m*X[:,0] + b, ls = '-')
axes[0].set_title('Original Data')
axes[1].set_title('Prediction')
```

```
Perceptron loss optimized w:[-1.45732283e-10 -1.44890080e-10  1.33737666e-1
0]
Logistic regression loss optimized w:[-0.00013    -0.06291806 -0.0107254 ]
```

Out[18]: `Text(0.5, 1.0, 'Prediction')`





What is the value of the loss function for the perceptron model after optimization?

In [19]:
```
print(perceptron(w_perc, X, y))
```

```
1.523639858855155e-08
```

What is the value of the loss function for the logistic regression model after optimization?

In [20]:
```
print(log_reg(w_log, X, y))
print('Seems reasonable, much higher than max cost loss')
```

```
138.6007017052387
Seems reasonable, much higher than max cost loss
```

What are the two main challenges of the perceptron loss function?

The solution for the optimum weights for the perceptron loss function is not unique. There are many suitable and valid solutions (if linearly separable data). Another challenge for this loss function is that it is difficult to differentiate and optimize.

3. Support Vector Machine

Write a function that computes the loss function of the support vector machine model.

This functions should take the followings as arguments:

- weight vector w
- the feature matrix $\bar{\bar{X}}$
- the output vector $\vec{y}$
- regularization strength α

You may want to use `add_intercept` and `linear_classifier` functions from the Problem 2.

In [21]:
```
#Dataset 1 = blob
X = X_blob
y = y_blob
w = np.array([-5, -2, -5])
alphas = [0, 1, 2, 10, 100]
```

In [22]:
```
def supVecMac(w, X=X, y=y, alpha = alphas):
    x_int = add_intercept(X)
    x_b = np.dot(x_int, w)
    loss = sum(np.maximum(0, 1 - y*x_b))
    loss += alpha*np.linalg.norm(w[1:],2)
    return loss
```

Evaluate the effect of regularization strength.

Optimize the SVM model for **Dataset 1**.

Search over α = [0, 1, 2, 10, 100] and assess the loss function of the SVM model.

In [25]:

```
#Dataset 1 = blob
X1 = X_blob
y1 = y_blob * 2 - 1 #scale y dataset (convert to -1,1)
w = np.array([-5, -2, -5])

alphas = [0, 1, 2, 10, 100]
w_SVM_array = np.zeros((5,3)) #will use to catch weights for alpha iterations
loss_array = np.zeros((5,1))

#iterate of alphas vector to get loss and optimize
for i in range(len(alphas)):
    print('alpha = {}'.format(alphas[i]))
    loss = supVecMac(w, X = X1, y = y1, alpha = alphas[i])
    print('Loss for original data with a = {} is: {}'.format(alphas[i],loss))
    outSVM = minimize(supVecMac, w, args = (X1, y1, alphas[i])) # minimize we
    w_SVM = outSVM.x
    w_SVM_array[i,:] = w_SVM #collect weights in array for plotting below
    print("SVM loss opt weights with a= {} is:{}".format(alphas[i], w_SVM))
    loss_array[i,0] = supVecMac(w_SVM, X1, y1, alphas[i])
    print("SVM loss opt value with a = {} is:{}".format(alphas[i], supVecMac(
    print('')

print(w_SVM_array)
print('')
print(loss_array)
```

```
alpha = 0
Loss for original data with a = 0 is: 4863.630015541278
SVM loss opt weights with a= 0 is:[-2.87496262  0.07464577  0.79805647]
SVM loss opt value with a = 0 is:74.18343801930003

alpha = 1
Loss for original data with a = 1 is: 4869.015180348412
SVM loss opt weights with a= 1 is:[-2.79544972  0.06720912  0.79179116]
SVM loss opt value with a = 1 is:74.96714182388266

alpha = 2
Loss for original data with a = 2 is: 4874.400345155547
SVM loss opt weights with a= 2 is:[-2.8106515   0.06996952  0.7886809 ]
SVM loss opt value with a = 2 is:75.76548028590945

alpha = 10
Loss for original data with a = 10 is: 4917.481663612623
SVM loss opt weights with a= 10 is:[-2.46724497  0.0581116   0.68156037]
SVM loss opt value with a = 10 is:81.69827343363832

alpha = 100
Loss for original data with a = 100 is: 5402.146496254729
SVM loss opt weights with a= 100 is:[-1.45446461  0.03127142  0.40177368]
SVM loss opt value with a = 100 is:127.0042101078237

[[-2.87496262  0.07464577  0.79805647]
 [-2.79544972  0.06720912  0.79179116]
 [-2.8106515   0.06996952  0.7886809 ]
 [-2.46724497  0.0581116   0.68156037]
 [-1.45446461  0.03127142  0.40177368]]
```

```
[[ 74.18343802]
 [ 74.96714182]
 [ 75.76548029]
 [ 81.69827343]
 [127.00421011]]
```

In [26]:
```
print('As alpha or regularization strength increases, loss increases.')
```

```
As alpha or regularization strength increases, loss increases.
```

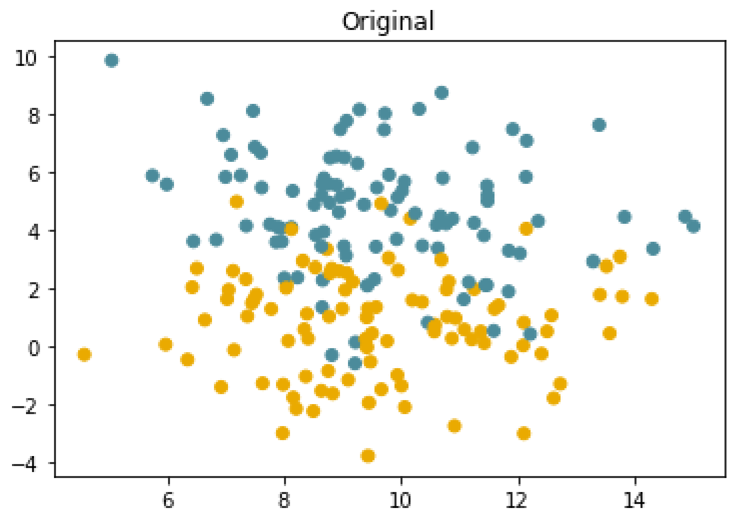
Plot the discrimination lines for α = [0, 1, 2, 10, 100].

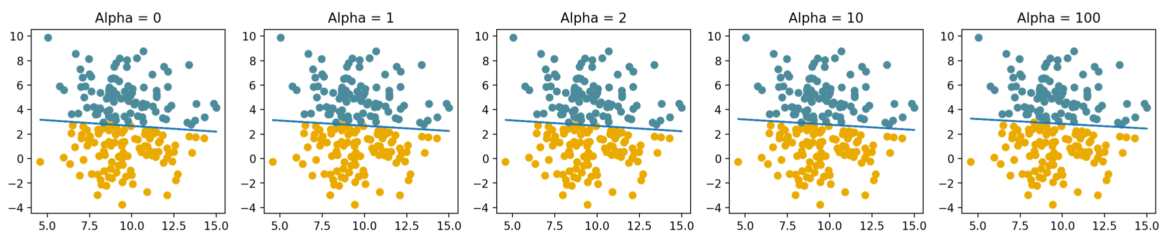
In [27]:

```
#Plot of original data
fig, ax = plt.subplots()
ax.scatter(X1[:,0], X1[:,1], c = clrs[y_blob + 1])
ax.set_title('Original')

#Prediction Graph with Various Alphas Values
fig, axes = plt.subplots(1, 5, figsize = (18, 3), dpi = 200)

#Alpha 1
predict1 = linear_classifier(X1, w_SVM_array[0,:]) #predict
axes[0].scatter(X1[:,0], X1[:,1], c = clrs[predict1 + 1]) #scatter plot
m = -w_SVM_array[0,1] / w_SVM_array[0,2] #slope
b = -w_SVM_array[0,0] / w_SVM_array[0,2] #y intercept
axes[0].plot(X1[:,0], m*X1[:,0] + b, ls = '-')
axes[0].set_title('Alpha = {}'.format(alphas[0]))

#Alpha 2
predict2 = linear_classifier(X1, w_SVM_array[1,:])
axes[1].scatter(X1[:,0], X1[:,1], c = clrs[predict2 + 1]) #scatter
m = -w_SVM_array[1,1] / w_SVM_array[1,2] #slope
b = -w_SVM_array[1,0] / w_SVM_array[1,2] #y intercept
axes[1].plot(X1[:,0], m*X1[:,0] + b, ls = '-')
axes[1].set_title('Alpha = {}'.format(alphas[1]))

#Alpha 3
predict3 = linear_classifier(X1, w_SVM_array[2,:])
axes[2].scatter(X1[:,0], X1[:,1], c = clrs[predict3 + 1]) #scatter
m = -w_SVM_array[2,1] / w_SVM_array[2,2] #slope
b = -w_SVM_array[2,0] / w_SVM_array[2,2] #y intercept
axes[2].plot(X1[:,0], m*X1[:,0] + b, ls = '-')
axes[2].set_title('Alpha = {}'.format(alphas[2]))

#Alpha 4
predict4 = linear_classifier(X1, w_SVM_array[3,:])
axes[3].scatter(X1[:,0], X1[:,1], c = clrs[predict4 + 1]) #scatter
m = -w_SVM_array[3,1] / w_SVM_array[3,2] #slope
b = -w_SVM_array[3,0] / w_SVM_array[3,2] #y intercept
axes[3].plot(X1[:,0], m*X1[:,0] + b, ls = '-')
axes[3].set_title('Alpha = {}'.format(alphas[3]))

#Alpha 5
predict5 = linear_classifier(X1, w_SVM_array[4,:])
axes[4].scatter(X1[:,0], X1[:,1], c = clrs[predict5 + 1]) #scatter
m = -w_SVM_array[4,1] / w_SVM_array[4,2] #slope
b = -w_SVM_array[4,0] / w_SVM_array[4,2] #y intercept
axes[4].plot(X1[:,0], m*X1[:,0] + b, ls = '-')
axes[4].set_title('Alpha = {}'.format(alphas[4]))
```

Out[27]:

```
Text(0.5, 1.0, 'Alpha = 100')
```





Find the optimal set of hyperparameters for an SVM model with Dataset 1.

Use `GridSearchCV` and find the optimal value of α and γ.

In [51]:
```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics.pairwise import rbf_kernel

#X and y values are X1 and y1 which is for the blob or Dataset 1
sigVec_options = np.array([1, 5, 7, 15, 20]) #possible sigma values that coul

alphas_new = [0.01, 1, 2, 10, 100] #need to throw out 0 as an option because
cS = np.divide(1,alphas_new) #give SVC proper input which uses inverse alpha
param_grid = {'C':cS}

findSigma = []
for i in range(len(sigVec_options)):
    findAlpha = []
    for j in range(len(alphas_new)):
        gamVec_options = 1./(2*sigVec_options[i]**2)
        svc = SVC(kernel = 'rbf', gamma = gamVec_options)
        xtrain = rbf_kernel(X1, X1, gamVec_options)
        svc_search = GridSearchCV(svc, param_grid, cv = 3) #3 fold cross vali
        svc_search.fit(xtrain,y1)
        r2 = svc_search.best_score_
        findAlpha.append(r2)


    optAlphaIndex = findAlpha.index(max(findAlpha))
    suboptAlpha = sigVec_options[optAlphaIndex]
    submaxR2 = max(findAlpha)

    findSigma.append((submaxR2, suboptAlpha))

optSigmaIndex = findSigma.index(max(findSigma))
optSigma = sigVec_options[optSigmaIndex]
optAlpha = findSigma[optSigmaIndex][1] #best alpha for every sigma iteration
maxR2 = findSigma[optSigmaIndex][0] #highest r2 value at every full iteration

print(findSigma) #prints vector with best r2 and best sigma from each iterati
print('')
print('Optimal sigma:{}'.format(optSigma))
print('Optimal gamma:{}'.format(1./(2*optSigma**2)))
print('Optimal alpha:{}'.format(optAlpha))
```

```
[(0.8346901854364542, 1), (0.854741444293683, 1), (0.8596411879993969, 1),
(0.8596411879993969, 1), (0.854741444293683, 1)]

Optimal sigma:7
Optimal gamma:0.01020408163265306
Optimal alpha:1
```

Calculate the accruacy, precision, and recall for the best model.

You can write your own function that calculates the metrics or you may use built-in functions.

In [52]:
```
def APR (y_set, y_real):
    TP = np.sum(np.logical_and(y_set == y_real, y_set == 1)) #true positive
    TN = np.sum(np.logical_and(y_set == y_real, y_set == 0)) #true negative
    FP = np.sum(np.logical_and(y_set != y_real, y_set == 1)) #false positive
    FN = np.sum(np.logical_and(y_set != y_real, y_set == 0)) #false negative

    accuracy = (TP + TN) / (TP + TN + FP + FN)

    if TP == 0:
        precision = 0
        recall = 0
    else:
        precision = TP / (TP + FP)
        recall = TP / (TP + FN)
    return accuracy, precision, recall

#Calculate values on dataset 1
svc = SVC(kernel = 'rbf', gamma = gamVec_options)
svc.fit(X1, y1)
y_predict = svc.predict(X1)

APR(y_predict, y1)
```

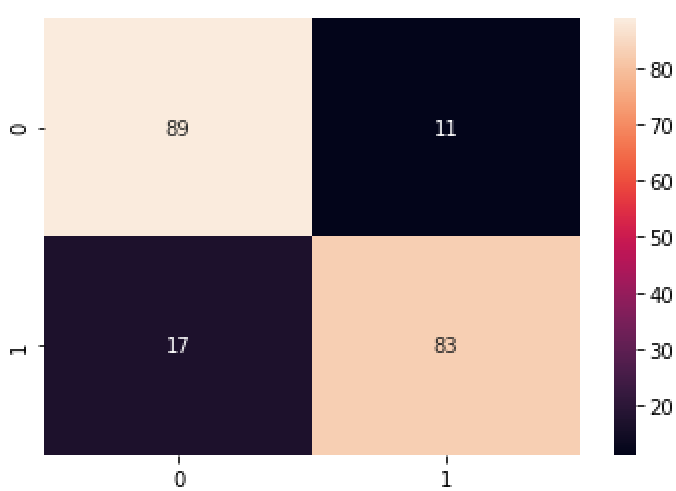
Out[52]:
```
(0.8829787234042553, 0.8829787234042553, 1.0)
```

Plot the confusion matrix.

In [60]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y1,y_predict) #visualize model vs prediction from svc f
print(cm)
sns.heatmap(cm, annot=True)
```

```
[[89 11]
 [17 83]]
```

Out[60]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1c5c313d970>
```



What happens to the decision boundary as α goes to ∞?

As alpha gets larger, C (the inverse) gets smaller. As C gets smaller, there will be more support vectors which will cause the decision boundary to be less effective in showing a distinction between the classes.

What happens to the decision boundary as γ goes to 0?

As gamma approaches 0, the decision boundary will become less complex and effective in showing a distinction between the classes.

I am taking 4745 = n/a

4. 6745 Only: Analytical Derivation

Derive an analytical expression for the gradient of the softmax function with respect to $\vec{w}$.

The **softmax** loss function is defined as:

$$g(\vec{w}) = \sum_i log(1 + \exp(-y_i \vec{x}_i^T \vec{w}))$$

where $\vec{x}_i$ is the i-th row of the input matrix $\bar{\bar{X}}$.

Hint 1: The function $g(\vec{w})$ can be expressed as $f(r(s(\vec{w})))$ where r and s are arbitrary functions and the chain rule can be applied.

Hint 2: You may want to review Ch. 4 of "Machine Learning Refined, 1st Ed."

I am taking 4745 - n/a

Optional: Logistic regression from the regression perspective

An alternate interpretation of classification is that we are performing non-linear regression to fit a **step function** to our data (because the output is whether 0 or 1). Since step functions are not differentiable at the step, a smooth approximation with non-zero derivatives must be used. One such approximation is the *tanh* function:

$$\tanh(x) = \frac{2}{1+\exp(-x)} - 1$$

This leads to a reformulation of the classification problem as:

$$\vec{y} = \tanh(\bar{\bar{X}}\vec{w})$$

Show that this is mathematically equivalent to **logistic regression**, or minimization of the **softmax** cost function.

I am taking 4745 - n/a

In [ ]: