Table of Contents

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib as mpl
from cycler import cycler

plt.style.use('default')

font_size = 12

mpl.rcParams['axes.prop_cycle'] = cycler('color', ['#003057', '#EAAA00
', '#4B8B9B', '#B3A369', '#377117', '#1879DB', '#8E8B76', '#002233', '#
F5D580'])
mpl.rcParams['axes.titlesize'] = font_size
mpl.rcParams['axes.titleweight'] = 'ultralight'
mpl.rcParams['axes.labelsize'] = font_size
mpl.rcParams['axes.labelweight'] = 'ultralight'

mpl.rcParams['xtick.labelsize'] = font_size
mpl.rcParams['xtick.direction'] = 'in'

mpl.rcParams['ytick.labelsize'] = font_size
mpl.rcParams['ytick.left'] = True
mpl.rcParams['ytick.direction'] = 'in'

mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.linestyle'] = '--'
#mpl.rcParams['lines.marker'] = 'o'

mpl.rcParams['figure.titlesize'] = font_size
mpl.rcParams['figure.titleweight'] = 'bold'
mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['figure.dpi'] = 300
mpl.rcParams['figure.autolayout'] = True

mpl.rcParams['savefig.dpi'] = 300
mpl.rcParams['savefig.format'] = 'svg'
mpl.rcParams['savefig.transparent'] = True

mpl.rcParams['font.size'] = font_size
mpl.rcParams['font.family'] = 'sans-serif'
mpl.rcParams['font.sans-serif'] = 'Helvetica'
mpl.rcParams['font.style'] = 'normal'

mpl.rcParams['mathtext.default'] = 'regular'

mpl.rcParams['legend.fontsize'] = font_size - 3
```

In [2]:
```
import numpy as np
clrs = np.array(['#003057', '#EAAA00', '#4B8B9B', '#B3A369', '#377117',
'#1879DB', '#8E8B76', '#F5D580', '#002233'])
```

Generalized Linear Models

In this lecture we will explore a type of discriminative classification model called "generalized linear models". This is slightly different from the "general linear model" we discussed for regression, but there are also some similarities.

Recall the general form of a linear model:

$y_i = \sum_j w_j X_{ij} + \epsilon_i$

or

$\vec{y} = \bar{\bar{X}}\vec{w} + \vec{\epsilon}$

In the case of a "general linear model", we assume that the error, $\vec{\epsilon}$, follows a normal distribution. However, in a generalized linear model the error follows other types of distributions. This is handled by taking a non-linear tranform:

$\vec{y_{GLM}} = \sigma(\bar{\bar{X}}\vec{w}) + \sigma(\vec{\epsilon})$

where $\sigma(\vec{z})$ is a non-linear function that "links" the normal distribution to the distribution of interest. These "link functions" can be derived from probability theory, but we will derive them from the loss function perspective.

Perceptron loss function

Recall the derivation of the "perceptron" loss function from the last lecture. We start with a model that discriminates between two classes:

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

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

Then multiply by y_i to form a single inequality:

$-y_i \bar{\bar{X}}\vec{w} < 0$

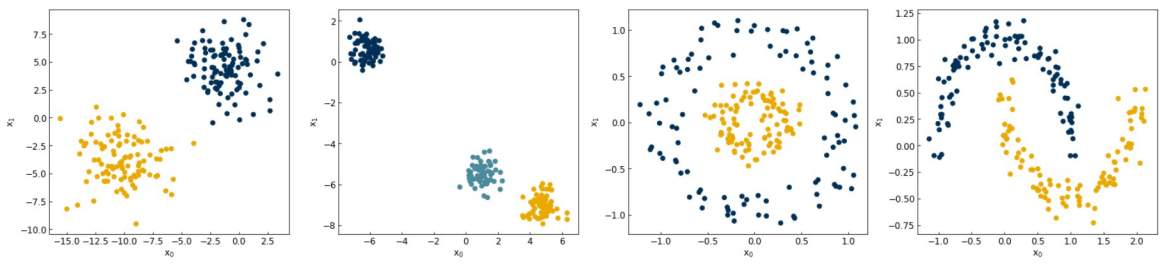
and take the maximum to create an equality:

$max(0, -y_i \bar{\bar{X}}\vec{w}) = 0$

We will apply this to the toy datasets:

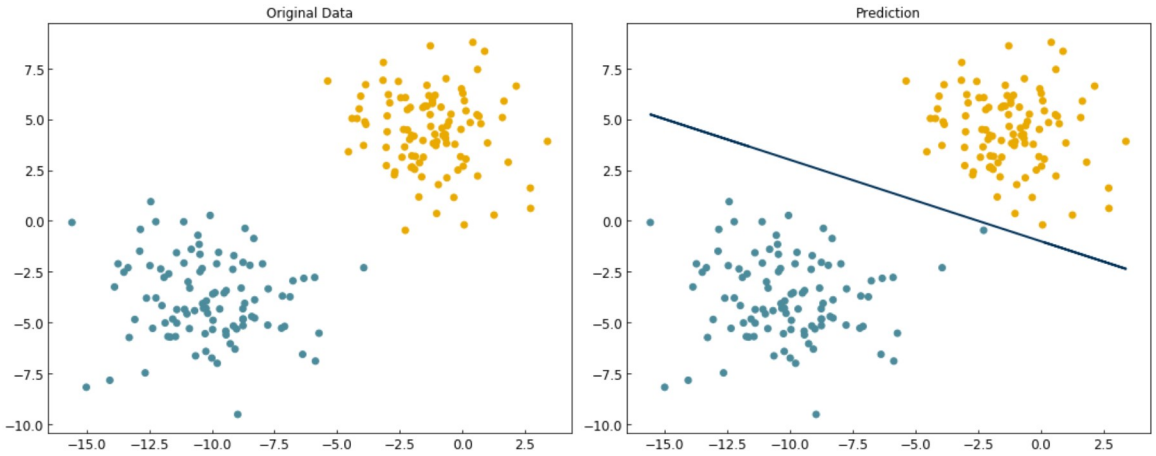
In [3]:
```
from sklearn.datasets.samples_generator import make_blobs, make_moons, make_circles
np.random.seed(1) #make sure the same random samples are generated each time

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, noise=0.1*noisiness)

X_moons, y_moons = make_moons(n_samples=200, noise=0.1*noisiness)

fig, axes = plt.subplots(1,4, figsize=(22, 5))

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

for i, Xy_i in enumerate(all_datasets):
    Xi, yi = Xy_i
    axes[i].scatter(Xi[:,0], Xi[:,1], c = clrs[yi])
    axes[i].set_xlabel('$x_0$')
    axes[i].set_ylabel('$x_1$')

plt.show()
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotlib/font_manager.py:1241: UserWarning: findfont: Font family ['sans-serif'] not found. Falling back to DejaVu Sans.
  (prop.get_family(), self.defaultFamily[fontext]))
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotlib/figure.py:2369: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



We can implement the model:

In [4]:
```
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_intercept = add_intercept(X)
    p = np.dot(X_intercept, w)
    return p > 0

X = X_blob
y = y_blob
y = y_blob*2 - 1 #convert to -1, 1

w = np.array([-10, -4, -10])
prediction = linear_classifier(X, w)

fig, axes = plt.subplots(1,2,figsize=(15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y_blob + 1])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

#plot line
m = -w[1] / w[2]
b = -w[0] / w[2]
axes[1].plot(X[:, 0], m*X[:, 0]+b, ls = '-')

axes[0].set_title('Original Data')
axes[1].set_title('Prediction');
```



and we can implement the max cost loss function:

In [5]:
```
def max_cost(w, X=X, y=y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept,w)
    return sum(np.maximum(0, -y*Xb))

print(max_cost(w,X,y))
```

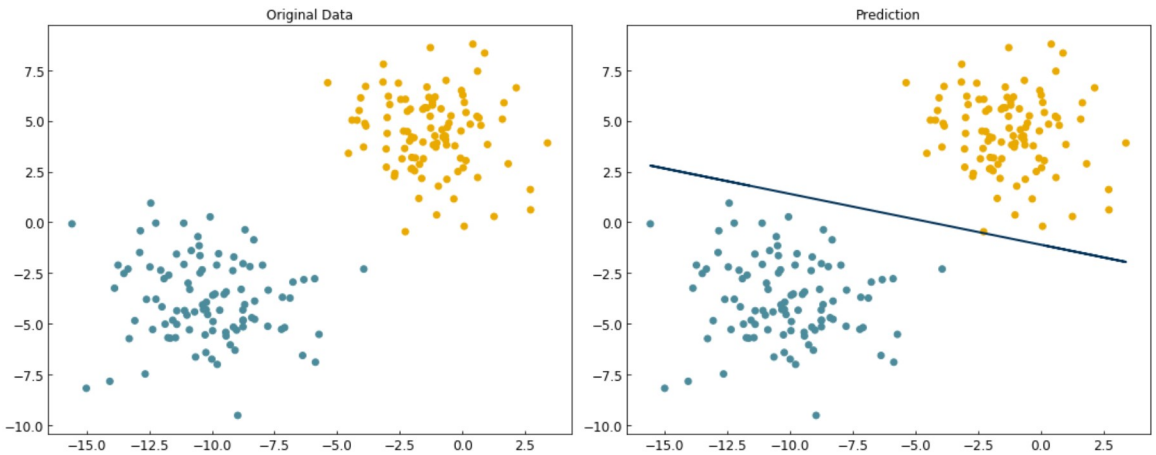
```
3.75103040564972
```

Now, we can solve the model by minimizing the loss function with respect to the parameters:

In [6]:
```
from scipy.optimize import minimize

result = minimize(max_cost, w)
w_perceptron = result.x
result
```

Out[6]:
```
      fun: 0.0
 hess_inv: array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
      jac: array([0., 0., 0.])
  message: 'Optimization terminated successfully.'
     nfev: 20
      nit: 1
     njev: 4
   status: 0
  success: True
        x: array([-10.69214391,  -2.42205481,  -9.67940886])
```

In [7]:
```
prediction = linear_classifier(X, w_perceptron)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y_blob + 1])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

#plot line
m = -w_perceptron[1] / w_perceptron[2]
b = -w_perceptron[0] / w_perceptron[2]
axes[1].plot(X[:, 0], m*X[:, 0] + b, ls = '-')

axes[0].set_title('Original Data')
axes[1].set_title('Prediction');
```



Discussion: What can go wrong with the max cost loss function?

Answer

- First derivative of the max cost loss function is non differentiable.
- Initial guess of [0, 0, 0] leads the max cost to zero, but a trivial.

The perceptron as a neural network

It turns out that the "perceptron", invented by Frank Rosenblatt in 1958, was the original neural network. The structure of the perceptron is similar to a biological neuron which "fires" if the sum of its inputs exceed some threshold:

The "perceptron" is equivalent to a "single layer" neural network with a step activation function. In fact, all the generalized linear models for classification are single layer neural networks, but with slightly different types of activation functions.

Logistic regression

The max cost loss function has two main problems:

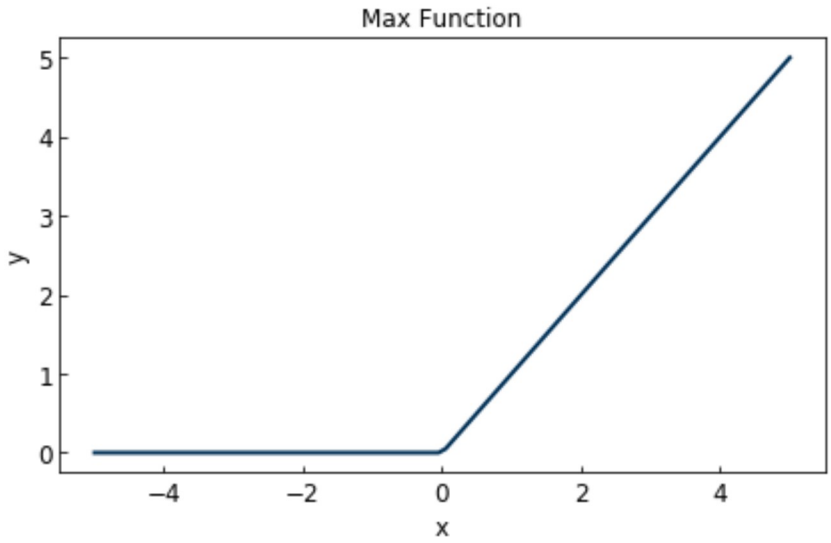
(1) There is a trivial solution at $\vec{w} = 0$.

(2) The max function is not differentiable.

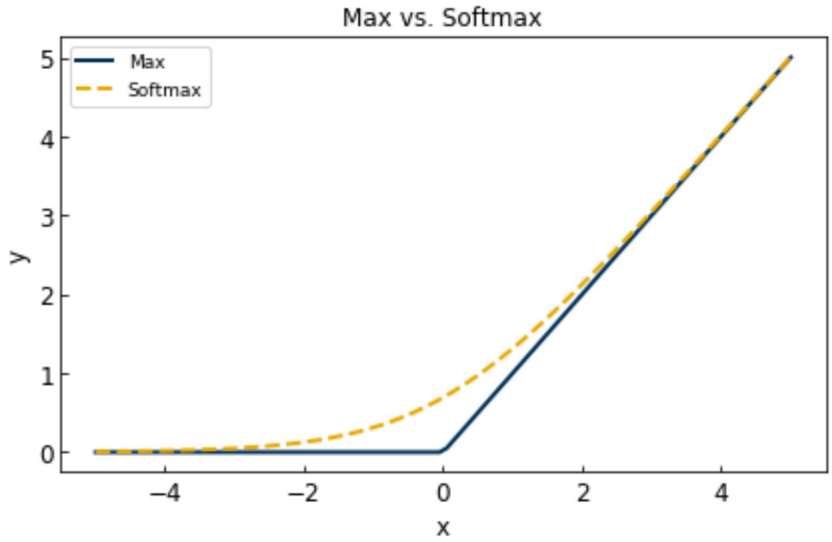
We can overcome the second problem by creating some smooth approximation of the maximum function. This is achieved using the "softmax" function:

$max(x,y) \approx soft(x,y) = log(exp(x) + exp(y))$

In [8]:
```
x = np.linspace(-5, 5, 100)

fig, ax = plt.subplots()

ax.plot(x, np.maximum(0, x), ls = '-')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Max Function');
```



In [9]:
```
x = np.linspace(-5, 5, 100)

fig, ax = plt.subplots()

ax.plot(x, np.maximum(0, x), ls = '-')
ax.plot(x, np.log(np.exp(0) + np.exp(x)), ls = '--')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend(['Max', 'Softmax'])
ax.set_title('Max vs. Softmax');
```



We can see that this also gets rid of the "trivial solution" at $\vec{w}=0$, so our problems are solved!

Now we can write a "softmax" cost function:

$g_{softmax}(\vec{w}) = \sum_i log\left\{1 + exp(-y_i \bar{\bar{X}}\vec{w})\right\}$

Let's implement it:

In [10]:
```
def softmax_cost(w, X = X, y = y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept, w)
    exp_yXb = np.exp(-y * Xb)
    return sum(np.log(1 + exp_yXb))

print(softmax_cost(w, X, y))
```

```
3.7745706457998764
```

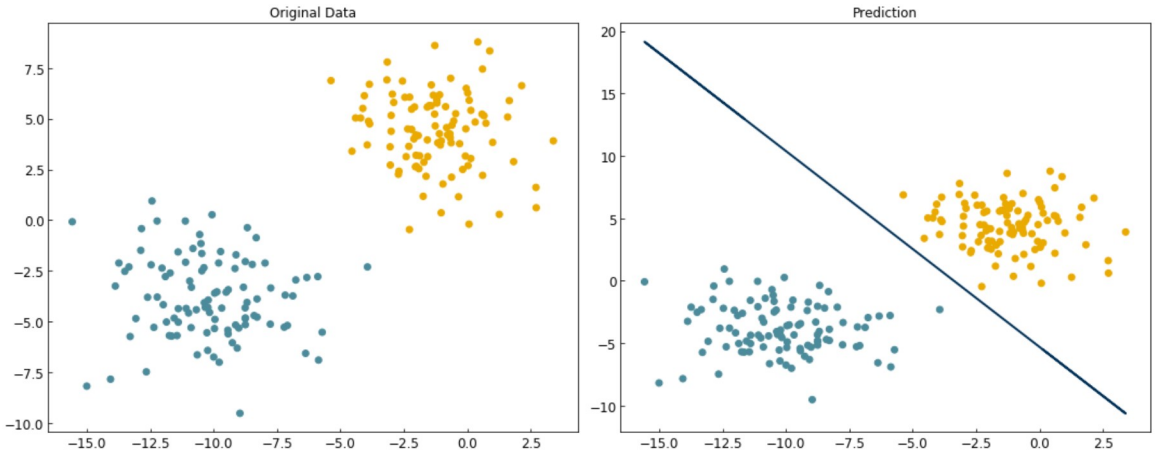
This function is differentiable, so we can minimize this with respect to $\vec{w}$ by setting the derivative equal to zero and solving for $\vec{w}$:

$\frac{\partial g_{softmax}}{\partial \vec{w}} = 0$

It turns out this problem is not linear, and needs to be solved iteratively using e.g. Newton's method. The math is a little more complex than before, so we won't cover it in lecture, but it is covered in Ch. 4 of "Machine Learning Refined" if you are interested. This approximation is called **logistic regression**.

The key concept to understand is that $\vec{w}$ is determined by minimizing the softmax cost function. We can do this numerically for our toy model:

In [11]:
```
from scipy.optimize import minimize

result = minimize(softmax_cost, w, args = (X, y))
w_logit = result.x

prediction = linear_classifier(X, w_logit)
fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y_blob + 1])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

#plot line
m = -w_logit[1] / w_logit[2]
b = -w_logit[0] / w_logit[2]
axes[1].plot(X[:, 0], m*X[:, 0] + b, ls = '-')

axes[0].set_title('Original Data')
axes[1].set_title('Prediction');
```



Note: There are other ways to derive "logistic regression". See Ch. 4 of ML refined for an alternative derivation.

Exercise: Compare the loss function for the perceptron and logistic regression after optimization for the "blobs" and "moons" datasets.

Margin loss function

Recall the two problems with the max cost function:

1) There is a "trivial solution" at $\vec{w} = 0$

2) The cost function is not differentiable at all points

Logistic regression uses a smooth approximation of the maximum to ensure differentiability, and the "trivial solution" goes away as a side effect.

An alternative approach is to directly eliminate the trivial solution by introducing a "margin" cost function, where we recognize that there will be some "buffer zone" between the classes:

We can write this mathematically as:

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

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

by using the same trick of multiplying by y_i and taking a maximum we can write this as an equality:

$max(0, 1 -y_i \bar{\bar{X}}\vec{w}) = 0$

and the corresponding cost/objective function:

$g_{margin}(\vec{w}) = \sum_i max(0, 1-y_i \bar{\bar{X}}\vec{w})$

Note that this is very similar to the cost function for the perceptron, but now there is no trivial solution at $\vec{w} = 0$. However, we can solve this with a few approaches:

1) Use derivative-free numerical approximations

2) Replax max with a differentiable function like $softmax$ or max^2

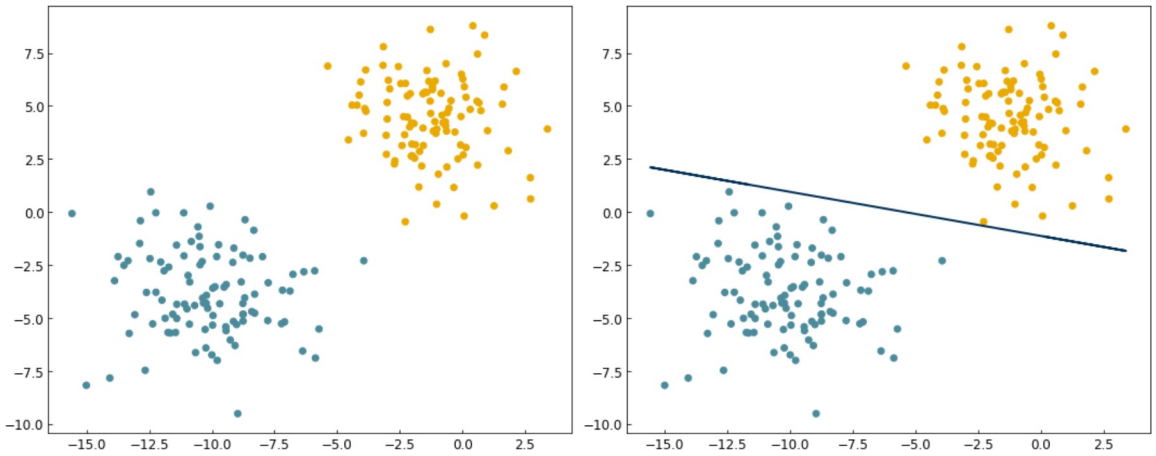
Let's see what happens with strategy 1:

In [12]:
```
def margin_cost(w, X = X, y = y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept,w)
    return sum(np.maximum(0, 1 - y * Xb))

print(margin_cost(w,X,y))
```

```
4.75103040564972
```

In [13]:
```
result = minimize(margin_cost, w)

w_opt_margin = result.x

prediction = linear_classifier(X, w_opt_margin)
fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y_blob + 1])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

#plot line
m = - w_opt_margin[1] / w_opt_margin[2]
b = - w_opt_margin[0] / w_opt_margin[2]
axes[1].plot(X[:, 0], m * X[:, 0] + b, ls = '-')
```

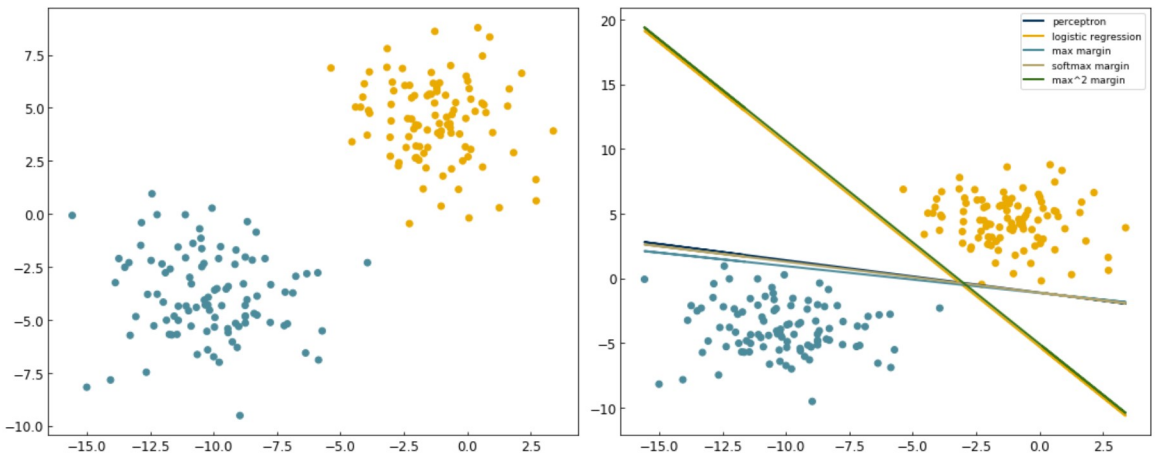
Out[13]:
```
[<matplotlib.lines.Line2D at 0x1a19840350>]
```



It works, but we get a different solution from logistic regression. Let's see how this compares to the max^2 and $softmax$ approximations:

In [14]:
```
def margin_cost_squared(beta, X = X, y = y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept, beta)
    return sum(np.maximum(0, 1 - y * Xb)**2)

def margin_cost_softmax(beta, X = X, y = y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept,beta)
    exp_yXb = np.exp(1 - y * Xb)
    return sum(np.log(1 + exp_yXb))
```

In [15]:
```
result = minimize(margin_cost_squared, w)
w_opt_margin2 = result.x

result = minimize(margin_cost_softmax, w)
w_opt_softmax = result.x

prediction = linear_classifier(X, w_opt_softmax)
fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y_blob + 1])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

#plot lines
def plot_line(ax, color, w, X, label):
    m = -w[1] / w[2]
    b = -w[0] / w[2]
    ax.plot(X[:, 0], m*X[:, 0] + b, ls = '-', color = color, label = la
bel)

labels = ['perceptron', 'logistic regression', 'max margin', 'softmax m
argin', 'max^2 margin']
w_set = [w_perceptron, w_logit,w_opt_margin, w_opt_margin2, w_opt_softm
ax]

for w_i, color, label in zip(w_set, clrs[:5], labels):
    plot_line(axes[1], color, w_i, X, label)

axes[1].legend();
```



Discussion: Which of these models is the best?

There are infinitely many lines that have equal cost for a linearly-separable dataset. The line that you find will depend on the approximation used, and can also depend on the initial guesses for the parameter $\vec{w}$. As we will see, additional constraints can be added to the loss function to alleviate this issue.

Support Vector Machine

From margins to support vectors

One of the most powerful classification models, "support vector machines", are very closely related to the margin cost function:

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

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

Multiply by y and convert to an equality:

$max(0, 1 -y_i \bar{\bar{X}}\vec{w}) = 0$

and sum over all points to get the loss function:

$g_{margin}(\vec{w}) = \sum_i max(0, 1-y_i \bar{\bar{X}}\vec{w})$

We can visualize this geometrically as:

The distance between the discrimination line and the closest points is called the "margin" of the model, and the points that define the margin are called the "support vectors". It can be shown with geometric arguments that the width of the margins is inversely proportional to the size of the weight vector (without the intercept term):

For support vector machines, the goal is to maximize the margins between the "support vectors". This is achieved by minimizing the value of the weights, $\vec{w}$. This can be done by "regularization", as we discussed in the regression lectures. Specifically, support vector machines use L_2 regularization:

$g_{SVM}(\vec{w}) = \sum_i max(0, 1-y_i \bar{\bar{X}}\vec{w}) + \alpha ||\vec{\tilde{w}}||_2 $

where $\vec{\tilde{w}}$ are the weights with the intercept omitted.

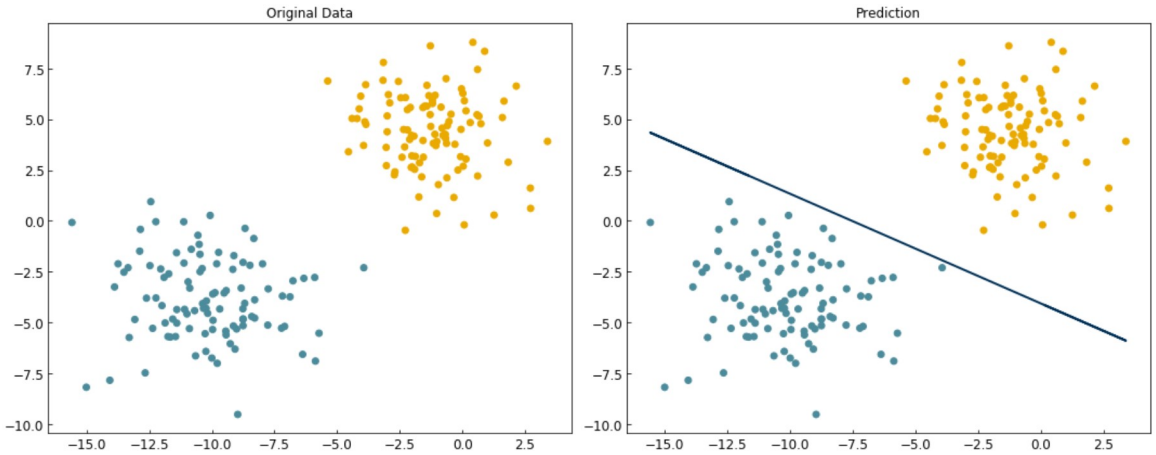
Let's use the new regularized cost function:

In [16]:
```
X = X_blob
y = y_blob * 2 - 1

def regularized_cost(w, X = X, y = y, alpha = 1):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept, w)
    cost = sum(np.maximum(0, 1 - y*Xb))
    cost += alpha*np.linalg.norm(w[1:], 2)
    return cost
```

and optimize it with the minimize function:

In [17]:
```
from scipy.optimize import minimize

w_guess = np.array([-10, -4, -10])
result = minimize(regularized_cost, w_guess, args = (X, y, 1))
w_svm = result.x

prediction = linear_classifier(X, w_svm)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y_blob + 1])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

#plot line
m = -w_svm[1] / w_svm[2]
b = -w_svm[0] / w_svm[2]
axes[1].plot(X[:, 0], m*X[:, 0] + b, ls = '-')
axes[0].set_title('Original Data')
axes[1].set_title('Prediction');
```

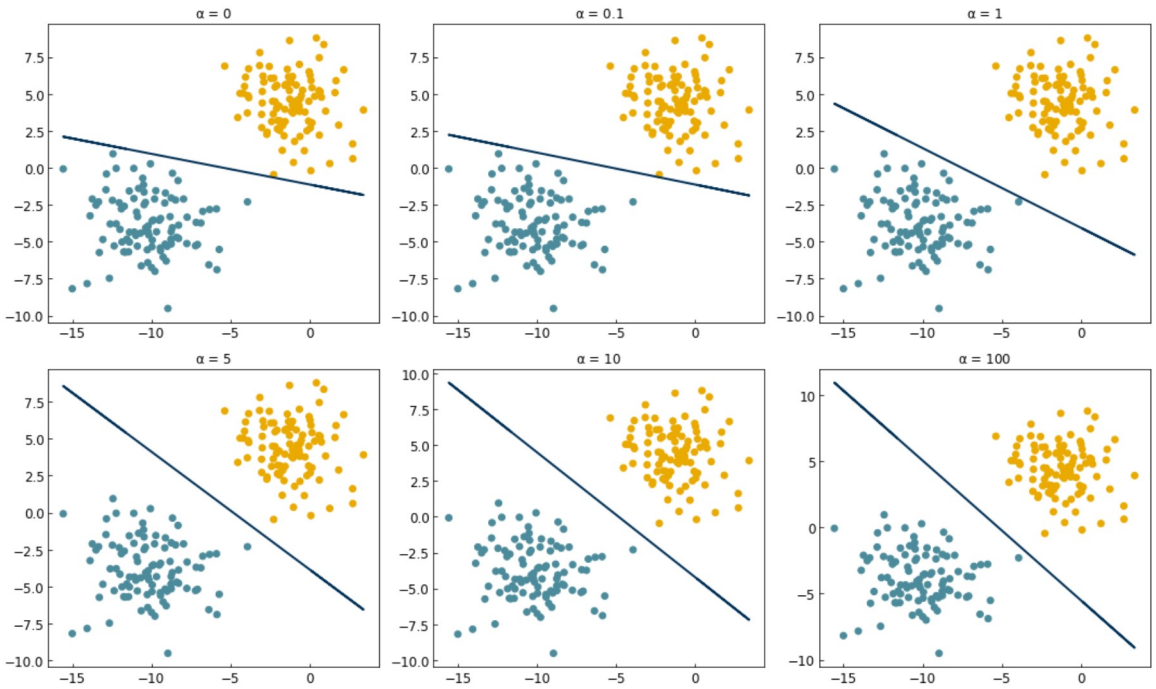
```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



Exercise: Plot the discrimination line for α = [0, 1, 2, 10, 100].

In [18]:
```
alphas = [0, 0.1, 1, 5, 10, 100]
fig, axes = plt.subplots(2, 3, figsize=(15, 9))
#axes[0].scatter(X[:,0],X[:,1],c=y)

axes = axes.ravel()

for i, alpha in enumerate(alphas):
    w_guess = np.array([-10, -4, -10])
    result = minimize(regularized_cost, w_guess, args = (X, y, alpha))
    w_svm = result.x

    prediction = linear_classifier(X, w_svm)
    axes[i].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

    #plot line
    m = -w_svm[1] / w_svm[2]
    b = -w_svm[0] / w_svm[2]
    axes[i].plot(X[:, 0], m*X[:, 0] + b, ls = '-')
    axes[i].set_title(r'$\alpha$ = {}'.format(alpha))
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



Support vector machines may sound scary, but as you can see above they are really just a very minor modification to ridge regression (least-squares regression regularized by the L_2 norm:

(1) The loss function is the "margin" loss function instead of the sum of squares.

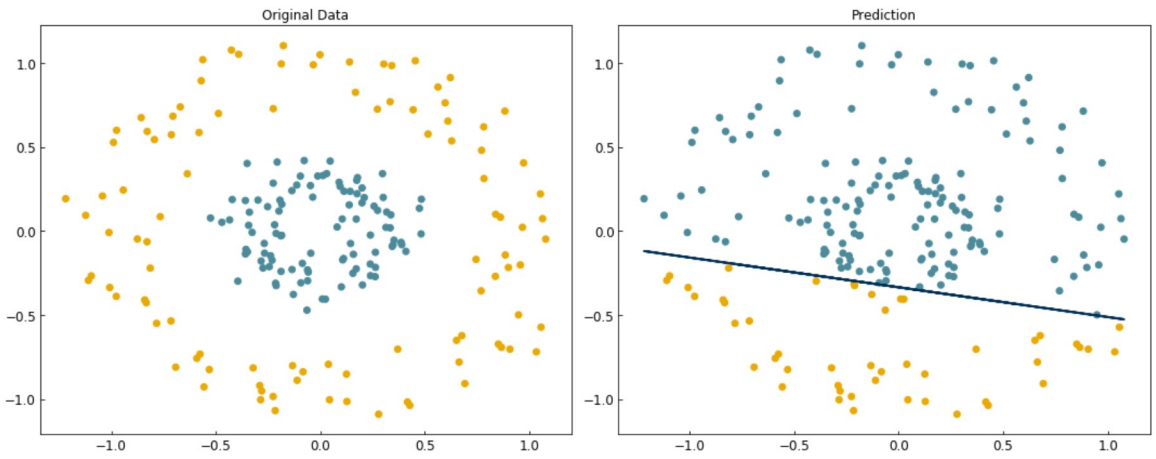
(2) The model must be solved numerically because it is non-linear.

Non-linearity and Kernels

We have seen lots of ways to find discrimination lines for linearly separable datasets, but they do not work well for non-linearly separable datasets:

In [19]:
```
X = X_circles
y = y_circles*2 - 1

w_guess = np.array([-10, -4, -10])
result = minimize(regularized_cost, w_guess, args=(X, y, 1))
w_svm = result.x

prediction = linear_classifier(X, w_svm)

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[prediction + 1])

#plot line
m = -w_svm[1] / w_svm[2]
b = -w_svm[0] / w_svm[2]
axes[1].plot(X[:, 0], m*X[:,0]+b, ls = '-')
axes[0].set_title('Original Data')
axes[1].set_title('Prediction');
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



For the case of general linear regression, we saw that we could endow a model with non-linear behavior by transforming the input features using polynomials, Gaussians, or other non-linear transforms. We can do something similar here, but it is slightly trickier since there are two variables. We can use a Gaussian transform as before:

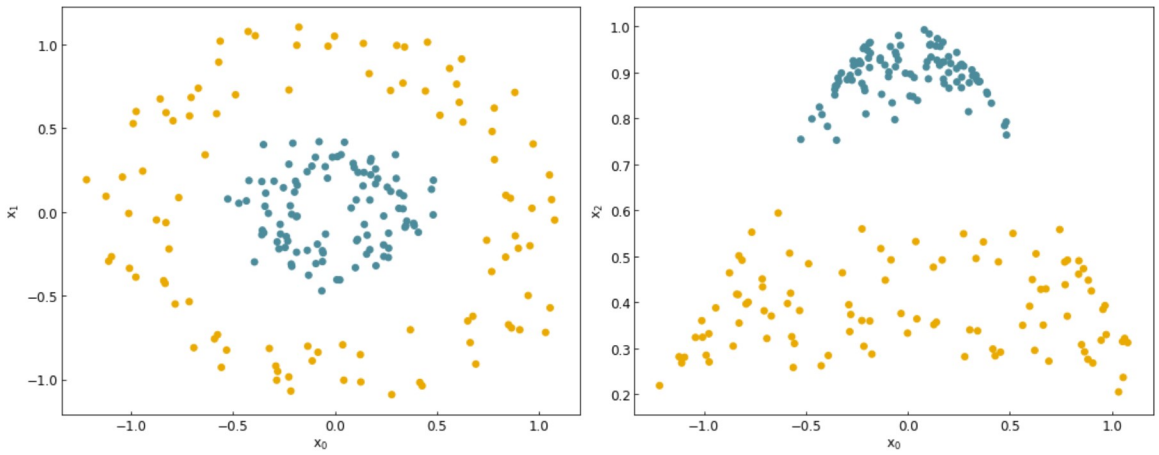
$x_{nonlinear} = exp(-(x_0^2 + x_1^2))$

where we have arbitrarily set the standard deviation to 1. We can add this as a third feature:

In [20]:
```
X_new = np.exp(-(X[:,0]**2 + X[:,1]**2))
X_new = X_new.reshape(-1, 1)
X_nonlinear = np.append(X, X_new, 1)
print(X_nonlinear.shape)
```

```
(200, 3)
```

In [21]:
```
fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_nonlinear[:, 0],X_nonlinear[:, 1],c = clrs[y_circles
+ 1])
axes[0].set_xlabel('$x_0$')
axes[0].set_ylabel('$x_1$')
axes[1].scatter(X_nonlinear[:, 0],X_nonlinear[:, 2],c = clrs[y_circles
+ 1])
axes[1].set_xlabel('$x_0$')
axes[1].set_ylabel('$x_2$');
```

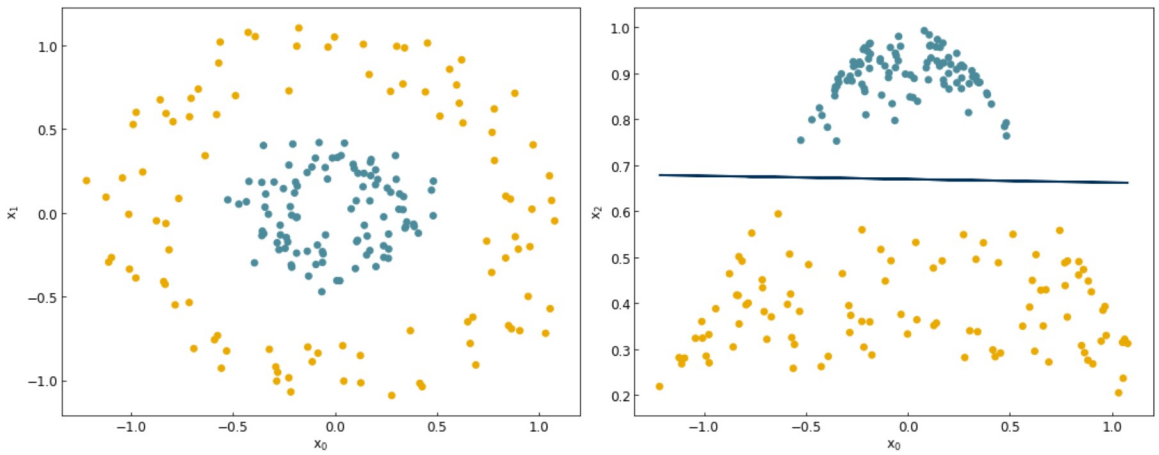


We see that the dataset is now linearly separable in this transformed space!

Let's see what happens if we use this new matrix as input to the SVM:

In [22]:
```
w_guess = np.array([-10, -4, 0, -10]) #note that we have an extra param
eter now

result = minimize(regularized_cost, w_guess, args = (X_nonlinear, y,
1))
w_svm = result.x

prediction = linear_classifier(X_nonlinear, w_svm)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_nonlinear[:, 0], X_nonlinear[:, 1], c = clrs[y_circle
s + 1])
axes[1].scatter(X_nonlinear[:, 0], X_nonlinear[:, 2], c = clrs[predicti
on + 1])

#plot line
m = -w_svm[1] / w_svm[3]
b = -w_svm[0] / w_svm[3]
axes[1].plot(X[:, 0], m*X[:,0] + b, ls = '-')
axes[0].set_xlabel('$x_0$')
axes[0].set_ylabel('$x_1$')
axes[1].set_xlabel('$x_0$')
axes[1].set_ylabel('$x_2$');
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```

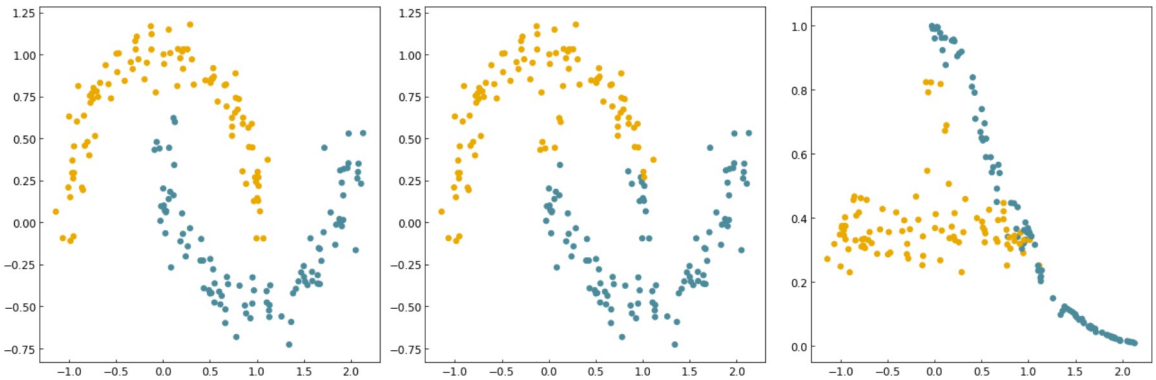


Now the model is able to correctly classify the non-linearly separable dataset! The kernel has created a new, higher-dimensional transformed space:

Let's see how it works for the "moons" dataset:

In [23]:
```
X = X_moons
y = y_moons*2 - 1

X_new = np.exp(-(X[:, 0]**2 + X[:, 1]**2))
X_new = X_new.reshape(-1, 1)
X_nonlinear = np.append(X, X_new, 1)

result = minimize(regularized_cost, w_guess, args = (X_nonlinear, y,
1))
w_svm = result.x

prediction = linear_classifier(X_nonlinear, w_svm)

fig, axes = plt.subplots(1, 3, figsize = (18, 6))
axes[0].scatter(X_nonlinear[:, 0], X_nonlinear[:, 1], c = clrs[y_moons
+ 1])
axes[1].scatter(X_nonlinear[:, 0], X_nonlinear[:, 1], c = clrs[predicti
on + 1])
axes[2].scatter(X_nonlinear[:, 0], X_nonlinear[:, 2], c = clrs[predicti
on + 1]);
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



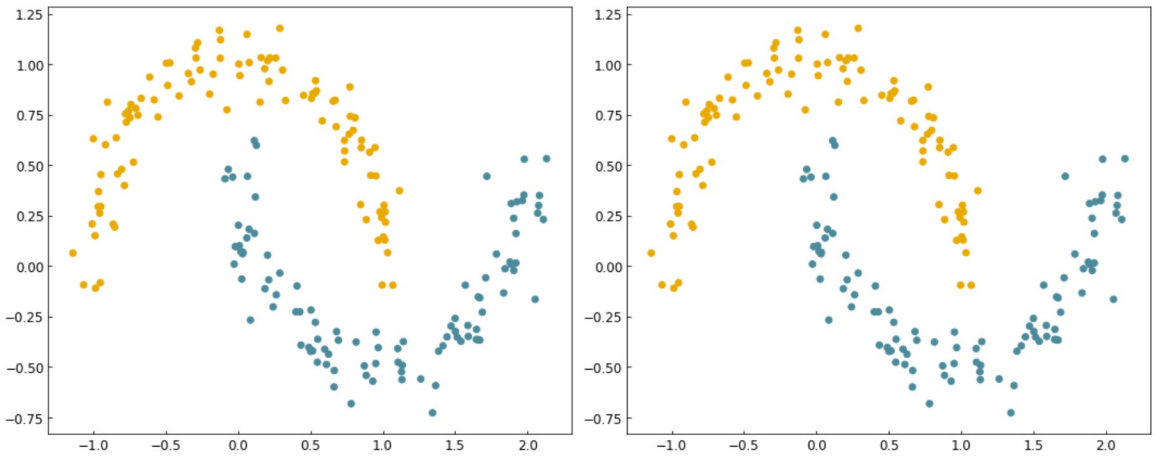
We see that it is an improvement, but not perfect, because the data is not linearly separable in the transformed space. To make this more general can use the "kernel" idea from "kernel ridge regression", and construct a new "kernel matrix":

In [24]:
```
from sklearn.metrics.pairwise import rbf_kernel

X_kernel = rbf_kernel(X, X, gamma=1)
print(X_kernel.shape)
```

```
(200, 200)
```

In [25]:
```
w_guess = np.zeros(X.shape[0] + 1)

result = minimize(regularized_cost, w_guess, args=(X_kernel, y, 1))
w_svm = result.x

prediction = linear_classifier(X_kernel, w_svm)

fig, axes = plt.subplots(1, 2, figsize=(15, 6))
axes[0].scatter(X_nonlinear[:, 0], X_nonlinear[:, 1], c = clrs[y_moons
+ 1])
axes[1].scatter(X_nonlinear[:, 0], X_nonlinear[:, 1], c = clrs[predicti
on + 1]);
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



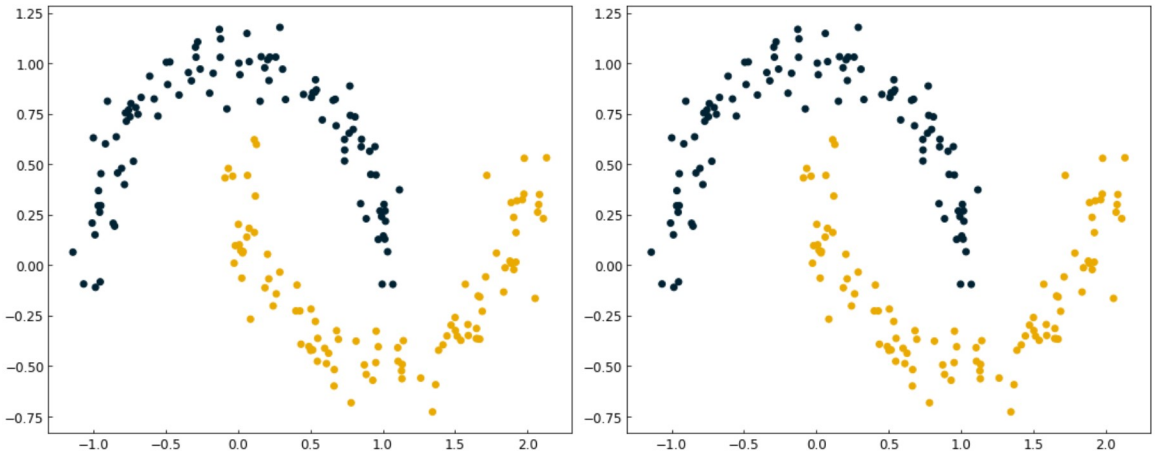
Discussion: Is this a parametric or non-parametric model? Do you think it will generalize?

In [26]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.5)

w_guess = np.zeros(X_train.shape[0] + 1)

X_train_kernel = rbf_kernel(X_train, X_train, gamma = 1)
X_test_kernel = rbf_kernel(X_test, X_train, gamma = 1)
```

We can make this process much easier by using the SVM model from `scikit-learn` (note that it is called a support vector "classifier", or SVC):

In [27]:
```
from sklearn.svm import SVC # "Support vector classifier"

model = SVC(kernel = 'rbf', gamma = 1e5, C = 1000)
model.fit(X, y)
y_predict = model.predict(X)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[y_predict]);
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



Note that there is a slight difference between the regularization strength in the SVC model and ridge regression. In the SVC model, the parameter `C` is **inversely** proportional to the regularization strength:

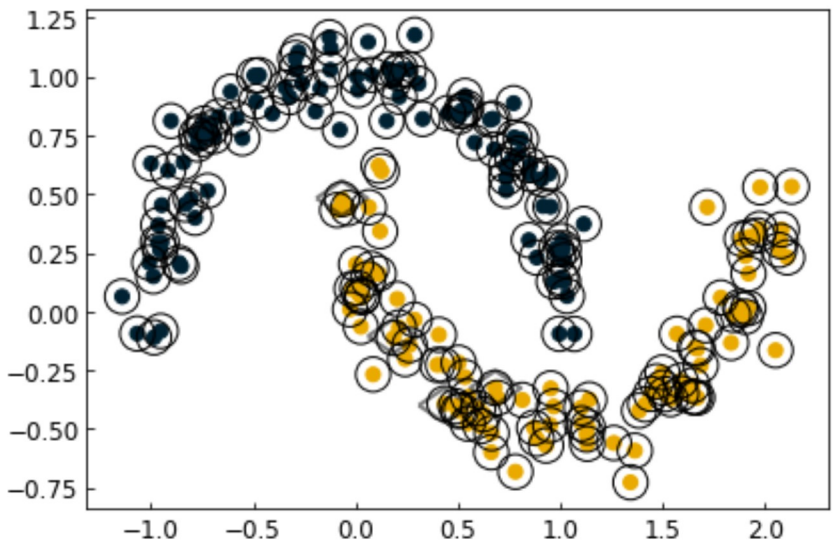
$g_{SVM}(\vec{w}) = \sum_i max(0, 1-y_i \bar{\bar{X}}\vec{w}) + \frac{1}{C} ||\vec{\tilde{w}}||_2 $

The function below will allow visualization of the decision boundary. You don't need to understand how it works, but should understand its output.

In [28]:
```
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # plot decision boundary and margins
    ax.contour(X, Y, P, colors = 'k',
               levels=[-1, 0, 1], alpha = 0.5,
               linestyles=['--', '-', '--'])
    if plot_support:
        # plot support vectors
        ax.scatter(model.support_vectors_[:, 0],
               model.support_vectors_[:, 1],
               s = 300, linewidth = 1, facecolors = 'none', edgecolors
= 'k');

    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```

In [29]:
```
fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c = clrs[y_predict], s = 50, cmap = 'RdBu
')
plot_svc_decision_function(model, ax = ax)
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



Discussion: How does the decision boundary change with C and γ?

Multi-class classification

Types Classification Datasets

There are a few different things to consider when examining a classification dataset:

- **Linearly separable**: A problem where it is possible to exactly separate the classes with a straight line (or plane) in the feature space.

- **Binary vs. Multi-class**: A binary classification problem has only 2 classes, while a multi-class problem has more than 2 classes.

There are two approaches to dealing with multi-class problems:

1) Convert multi-class problems to binary problems using a series of "one vs. the rest" binary classifiers

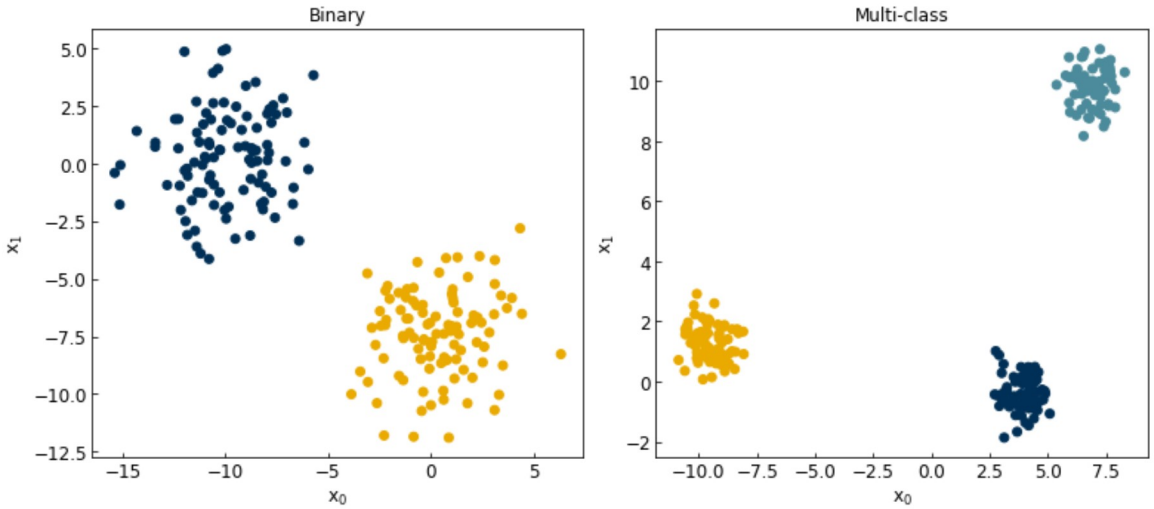
2) Consider the multi-class nature of the problem when deriving the method (e.g. kNN) or determining the cost function (e.g. logistic regression)

In the end, the difference between these approaches tend to be relatively minor, although the training procedures can be quite different. One vs. the rest is more efficient for parallel training, while multi-class objective functions are more efficient in serial.

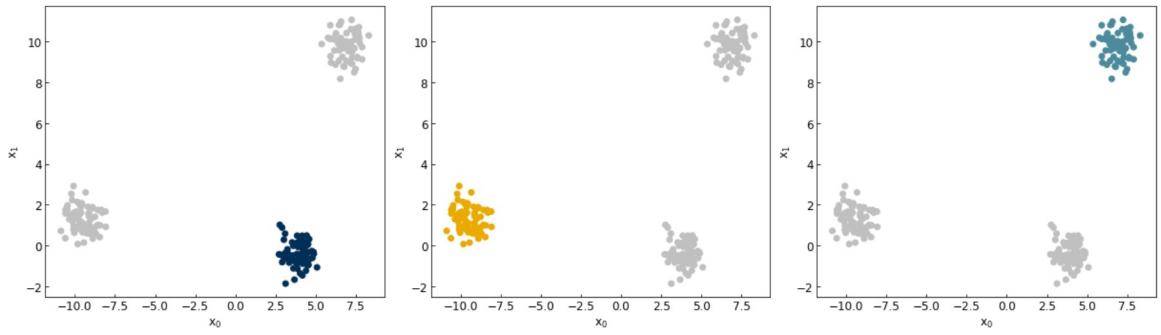
- **Balanced vs. Imbalanced**: A balanced problem has roughly equal numbers of examples in all classes, while an imbalanced problem has an (typically significantly) higher number of examples of some classes. Strategies for overcoming class imbalance will be briefly discussed in subsequent lectures.

In [30]:
```
np.random.seed(9)
X_blob1, y_blob1 = make_blobs(n_samples = 200, centers = 2, cluster_std
= 2 * noisiness, n_features = 2)

np.random.seed(248)
X_blob3, y_blob3 = make_blobs(n_samples = 200, centers = 3, cluster_std
= .6 * noisiness, n_features = 2)
```

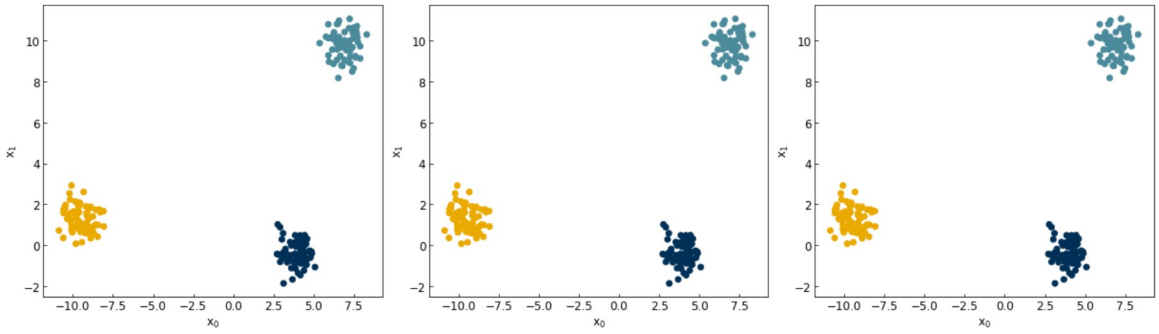
In [31]:
```
all_datasets = [[X_blob1, y_blob1], [X_blob3, y_blob3]]
fig, axes = plt.subplots(1, 2, figsize=(11, 5))

for i, Xy_i in enumerate(all_datasets):
    Xi, yi = Xy_i
    axes[i].scatter(Xi[:,0], Xi[:,1], c = clrs[yi])
    axes[i].set_xlabel('$x_0$')
    axes[i].set_ylabel('$x_1$')

axes[0].set_title('Binary')
axes[1].set_title('Multi-class')
plt.show()
```



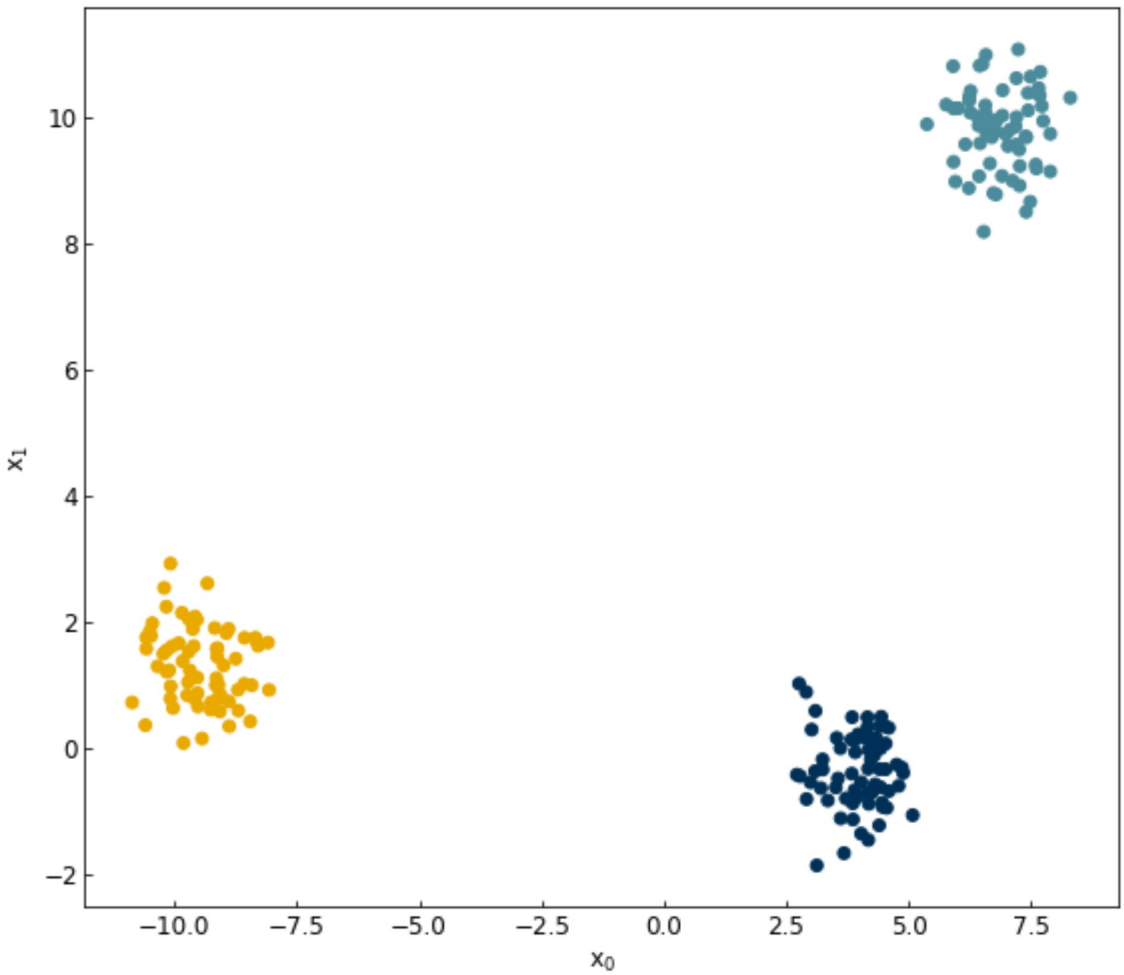
In [32]:
```
fig, axes = plt.subplots(1, 3, figsize=(17, 5))

for i in range(3):
    axes[i].scatter(X_blob3[:, 0], X_blob3[:, 1], c = [clrs[i] if yi ==
i else '#C0C0C0' for yi in y_blob3])
    axes[i].set_xlabel('$x_0$')
    axes[i].set_ylabel('$x_1$')

plt.show()
```



In [33]:
```
fig, axes = plt.subplots(1, 3, figsize=(17, 5))

for i in range(3):
    axes[i].scatter(X_blob3[:, 0], X_blob3[:, 1], c = clrs[y_blob3])
    axes[i].set_xlabel('$x_0$')
    axes[i].set_ylabel('$x_1$')

plt.show()
```

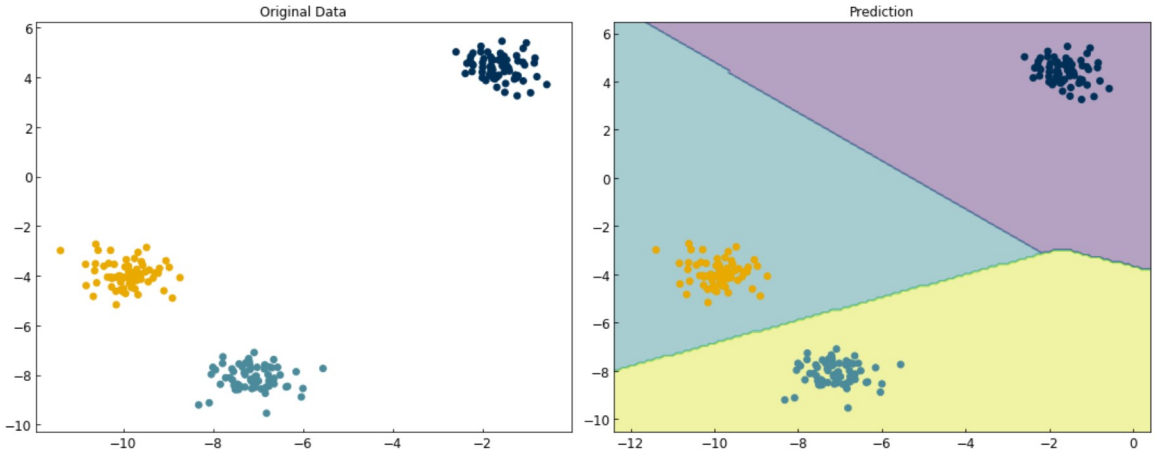


In [34]:
```
fig, ax = plt.subplots(figsize = (8, 7))

ax.scatter(X_blob3[:, 0], X_blob3[:, 1], c = clrs[y_blob3])
ax.set_xlabel('$x_0$')
ax.set_ylabel('$x_1$')

plt.show()
```



In [35]:
```
np.random.seed(1)
X_mc, y_mc = make_blobs(n_samples = 200, centers = 3, cluster_std = 0.5
*noisiness, n_features = 2)

model = SVC(kernel = 'linear', C = 1, decision_function_shape = 'ovr')

model.fit(X_mc, y_mc)
y_mc_hat = model.predict(X_mc)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_mc[:, 0], X_mc[:, 1], c = clrs[y_mc])

x_min, x_max = X_mc[:, 0].min() - 1, X_mc[:, 0].max() + 1
y_min, y_max = X_mc[:, 1].min() - 1, X_mc[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_m
ax, 0.1))

Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

axes[1].contourf(xx, yy, Z, alpha = 0.4)
axes[1].scatter(X_mc[:, 0], X_mc[:, 1], c = clrs[y_mc_hat])
axes[0].set_title('Original Data')
axes[1].set_title('Prediction');
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/matplotli
b/figure.py:2369: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



Counting loss function

Recall from the prior lecture that we can also set up a loss function that counts the total number of mis-classified points:

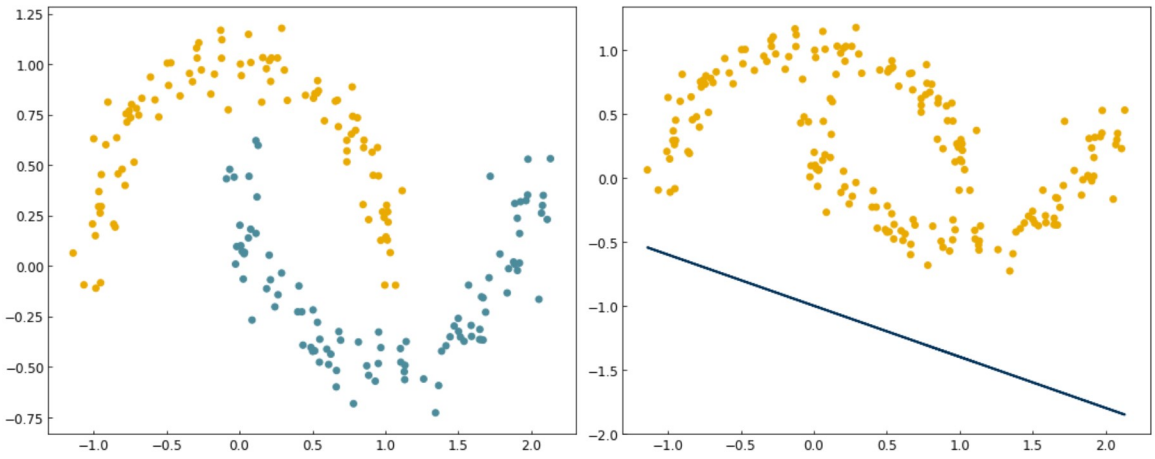
In [36]:
```
def n_wrong(w, X = X, y = y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept, w)
    return sum(np.maximum(0, np.sign(-y * Xb)))

print(n_wrong(w_logit))
```

```
100.0
```

In principle, we can also minimize this directly:

In [37]:
```
result = minimize(n_wrong, w)

w_count = result.x
print(n_wrong(w_count))

prediction = linear_classifier(X, w_count)
fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y_moons + 1])
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[prediction + 1])

#plot line
m = -w_count[1] / w_count[2]
b = -w_count[0] / w_count[2]
axes[1].plot(X[:, 0], m*X[:, 0] + b, ls = '-')
result;
```

```
100.0
```



The problem is that the "sign" function is not differentiable! This makes it a bad loss function. In general, we expect that minimizing the loss functions should also minimize the number of incorrect points, but this isn't always the case.

Discussion: How will the different cost functions respond to outliers?