Table of Contents

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('../settings/plot_style.mplstyle')
```

In [2]:
```
import numpy as np
import pandas as pd

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

Classification Overview

Problem statement and datasets

A model that maps continuous or discrete inputs to a discrete (ordinal, categorical, or integer) output space. In this course we will focus primarily on problems with continuous inputs.

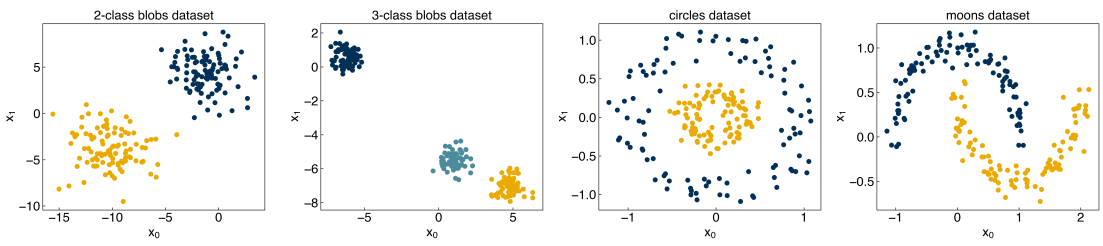
Toy datasets

Since this problem type is significantly different than what we have seen before, we will introduce a few

new datasets in this module. First, we will consider some "toy" datasets that can be generated using `scikit-learn`:

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]]
titles = ['2-class blobs dataset', '3-class blobs dataset', 'circles dataset', 'moons dataset']

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[i].set_title(titles[i])

plt.show()
```

```
/Users/SihoonChoi/opt/anaconda3/lib/python3.7/site-packages/sklearn/utils/deprecation.py:143: FutureWarning: The sklearn.datasets.samples_generator module is  deprecated in version 0.22 and will be removed in version 0.24. The corresponding classes / functions should instead be imported from sklearn.datasets. Anything that cannot be imported from sklearn.datasets is now part of the private API.
  warnings.warn(message, FutureWarning)
```

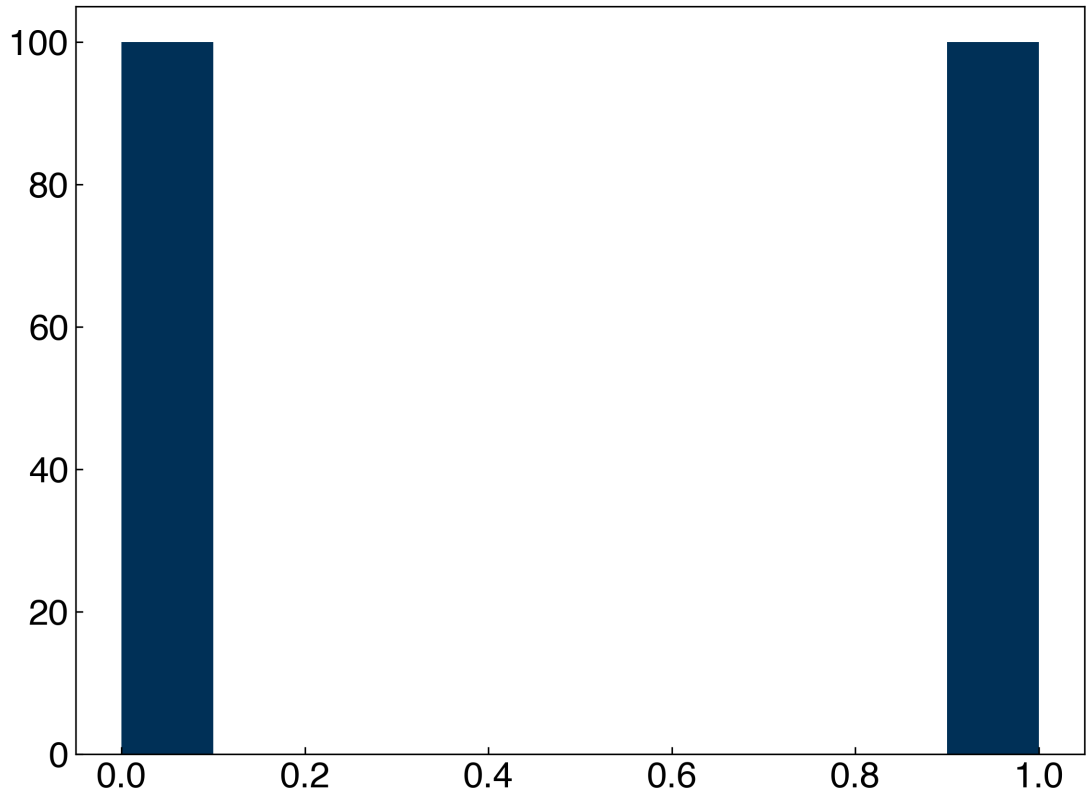


A few things to note:

- The "random seed" is set to 1, so these will be the same every time.
- There is a "noisiness" variable that allows us to easily add more noise to these datasets.

- The outputs, y, are approximately evenly divided between [0, 1] for 2-class datasets, or [0, 1, 2] for the 3 class dataset.

In [4]:
```
fig, ax = plt.subplots()
ax.hist(y_blob)
plt.show()
```



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

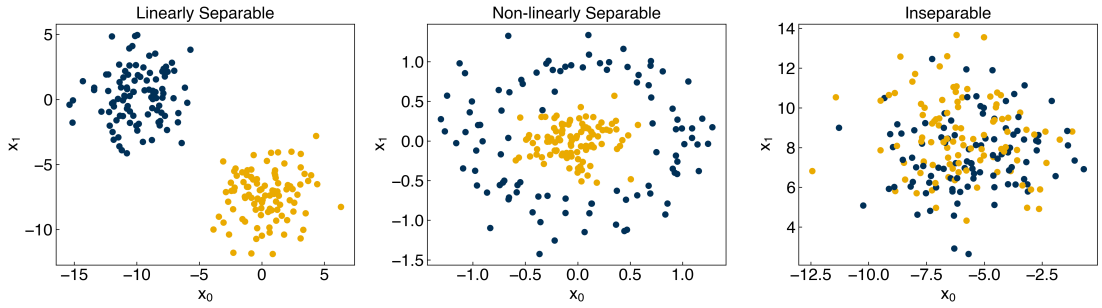
2) Consider the multi-class nature of the problem when deriving the method (e.g. kNN) or determining the cost function (e.g. logistic regression)

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 [5]:
```
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(5)
X_blob2, y_blob2 = make_blobs(n_samples = 200, centers = 2, cluster_std
= 2 * noisiness, n_features = 2)

X_circles1, y_circles1 = make_circles(n_samples = 200, factor = 0.2, no
ise = 0.2 * noisiness)

all_datasets = [[X_blob1, y_blob1], [X_circles1, y_circles1], [X_blob2,
y_blob2]]

fig, axes = plt.subplots(1, 3, figsize=(17, 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('Linearly Separable')
axes[1].set_title('Non-linearly Separable')
axes[2].set_title('Inseparable')
plt.show()
```



Discussion: Which of the datasets are:

- Linearly separable?

  blobs dataset

- Non-linearly separable?

  moons dataset, circles dataset

- Balanced?

  all datasets are balanced

General types of classification models

There are two distinct types of classification models: discriminative and generative. We will focus on discriminative models, but will also discuss and see examples of generative models.

Discriminative models:

These models are most similar to regression. Rather than learning a line\model that best represents the data we want to learn a line\model that best separates (discriminates) between different classes. For a binary classifier we can write this as:

$f(\vec{x})>p$ if class 1

$f(\vec{x})<p$ if class 2

where p is some constant threshold that separates the classes.

Another way to think of this is that we will establish a function that estimates the probability of a point belonging to a particular class, given its features:

$$P(y_i|\vec{x}) = f(\vec{x})$$

Then the classes can be discretized by establishing probability cutoffs. Conceptually, discriminative models separate classes by identifying **differences** between classes, and directly solve the problem of estimating class probability.

Generative models

Generative models are somewhat less intuitive, but can be very powerful. In a generative model the goal is to solve the "inverse problem" of predicting the probability of features given a class label output. Conceptually, you can think of this as identifying **similarities** between points within a given class. Mathematically:

$$P(\vec{x}|y_i) = f(\vec{x})$$

This is counter-intuitive, but the model can then be used in conjunction with Bayes' rule to indirectly solve the classification problem. Bayes rule is:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \rightarrow P(y_i|\vec{x}) = \frac{P(\vec{x}|y_i)P(y_i)}{P(\vec{x})}$$

The $P(y_i)$ term is available from the data (number of times each class appears) and the $P(\vec{x})$ term is a constant so it can be dropped when computing relative probabilities.

Generative models are more difficult to understand, but they have a key advantage: new synthetic data can be generated by using the function $P(\vec{x}|y_i)$. This opens the possibility of iterative training schemes that systematically improve the estimate of $P(\vec{x}|y_i)$ (e.g. Generative Artificial Neural Networks) and can also aid in diagnosing problems in models.

Accuracy metrics and model validation

Assessing the accuracy of a classification model requires different metrics from regression. We will explore a few here.

False positives and false negatives

Since the output of a classification problem is discrete, we can have different types of errors. In particular, there are 2 types of errors for any 2-class problem:

- False positives (Type I error): A point is classified as 1 but should be 0.
- False negatives (Type II error): A point is classified as 0 but should be 1.

Note that in most problems we will work with the definition of 0 and 1 is arbitrary, so these types can be arbitrarily switched. However, they are still distinctly different kinds of failures of the model, and in some cases it can make a big difference.

Discussion: Consider a chemical process where your model is predicting whether or not a reactor is near runaway conditions. What are the implications of a false positive or negative?

Accuracy, Precision, Recall, and F1 scores

The accuracy, precision, and recall are 3 common metrics for evaluating 2-class models:

- Accuracy = (number correct)/(total) = (TP + TN)/(TP + TN + FP + FN)
- Precision = TP/(TP + FP)
- Recall = TP/(TP + FN)

An additional metric, the F1 score, is sometimes used to summarize precision and recall:

- F1 = $\frac{2Precision \times Recall}{Precision + Recall}$

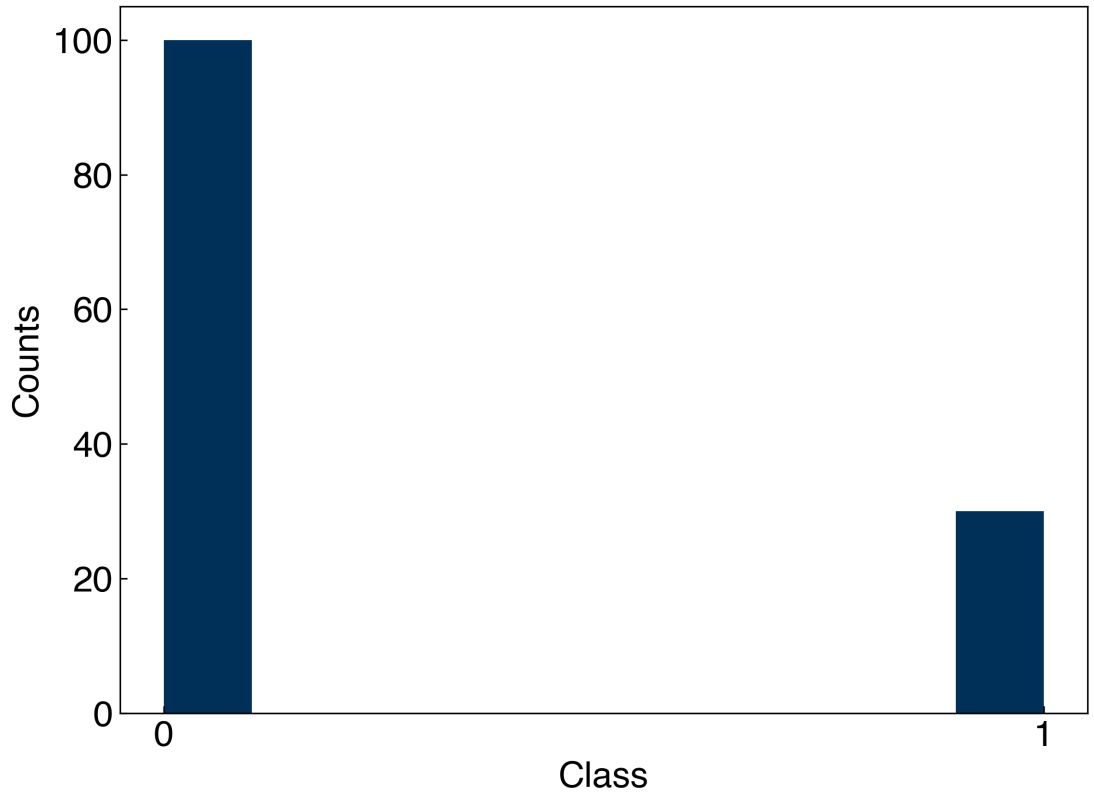
This is the "harmonic mean" of precision and recall and ranges from 0 for a model with 0 precision or recall to 1 for a model with perfec precision and recall.

We can implement this with a simple Python function:

In [6]:
```
def acc_prec_recall(y_model, y_actual):
    TP = np.sum(np.logical_and(y_model == y_actual, y_model == 1))
    TN = np.sum(np.logical_and(y_model == y_actual, y_model == 0))
    FP = np.sum(np.logical_and(y_model != y_actual, y_model == 1))
    FN = np.sum(np.logical_and(y_model != y_actual, y_model == 0))
    acc = (TP + TN) / (TP + TN + FP + FN)
    if TP == 0:
        prec = 0
        recall = 0
    else:
        prec = TP / (TP + FP)
        recall = TP / (TP + FN)
    return acc, prec, recall
```

These metrics depend strongly on the class imbalance! Let's take one of our toy datasets and create a truly bad classifier that always guesses that the class is 0. Then, we can see how the accuracy, precision, and recall of this classifier change as the imbalance shifts:

In [7]:
```
N_include = 30

#only include N_include examples of class 1
y_imbalanced = []
Ni = 0
for i, yi in enumerate(y_moons):
    if yi == 1 and Ni < N_include:
        y_imbalanced.append(yi)
        Ni += 1
    elif yi == 0:
        y_imbalanced.append(yi)

y_imbalanced = np.array(y_imbalanced)
fig, ax = plt.subplots()
ax.hist(y_imbalanced)
ax.set_xticks([0, 1])
ax.set_xlabel('Class')
ax.set_ylabel('Counts');
```

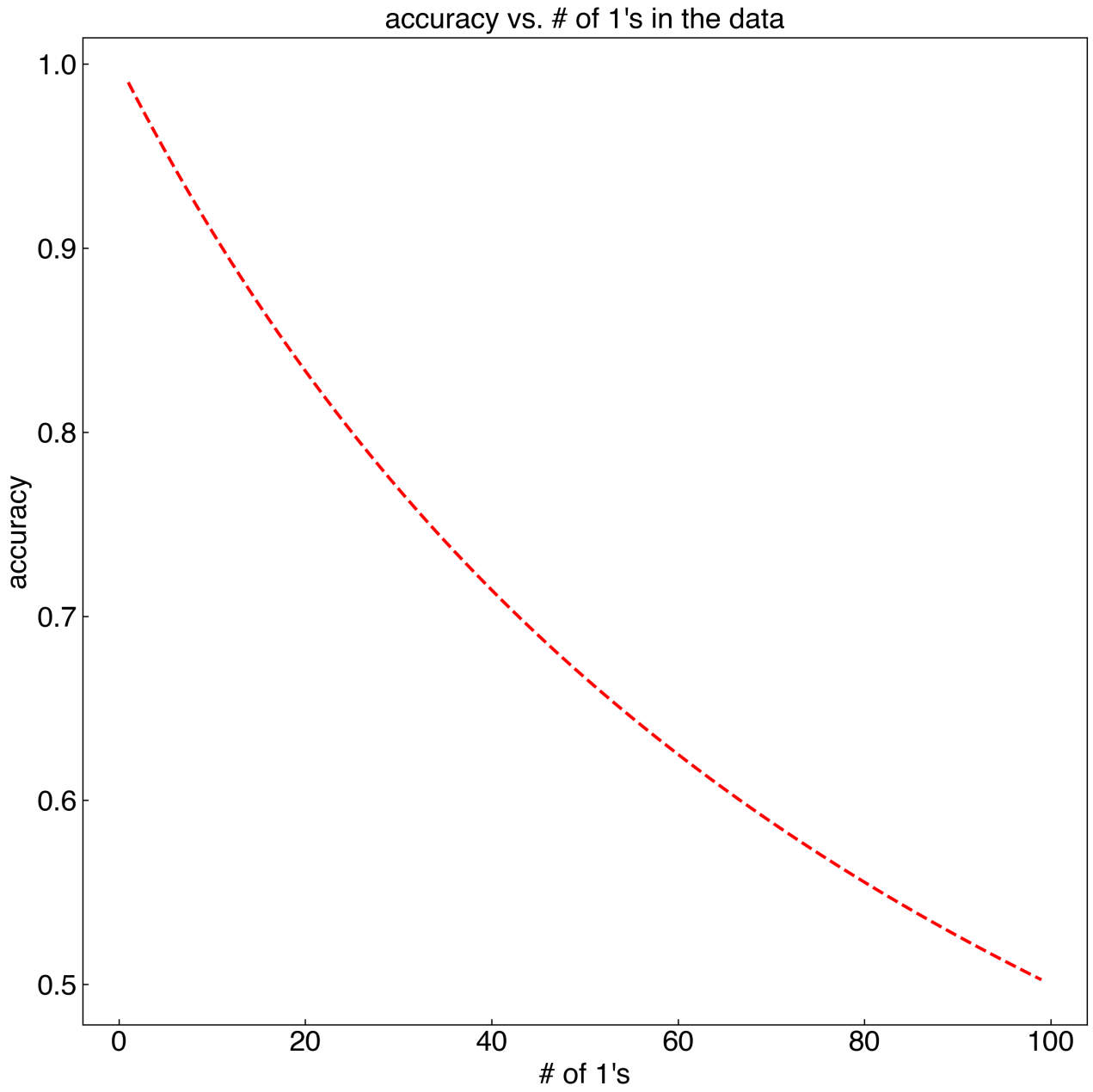


Now we can calculate the accuracy, precision, and recall when our classifier simply guesses that every point is 0:

In [8]:
```
y_guess = np.zeros(len(y_imbalanced))

acc_prec_recall(y_guess, y_imbalanced)
```

Out[8]: (0.7692307692307693, 0, 0)

Exercise: Plot the accuracy of the "guessing zero" model as a function of number of 1's included in the actual data



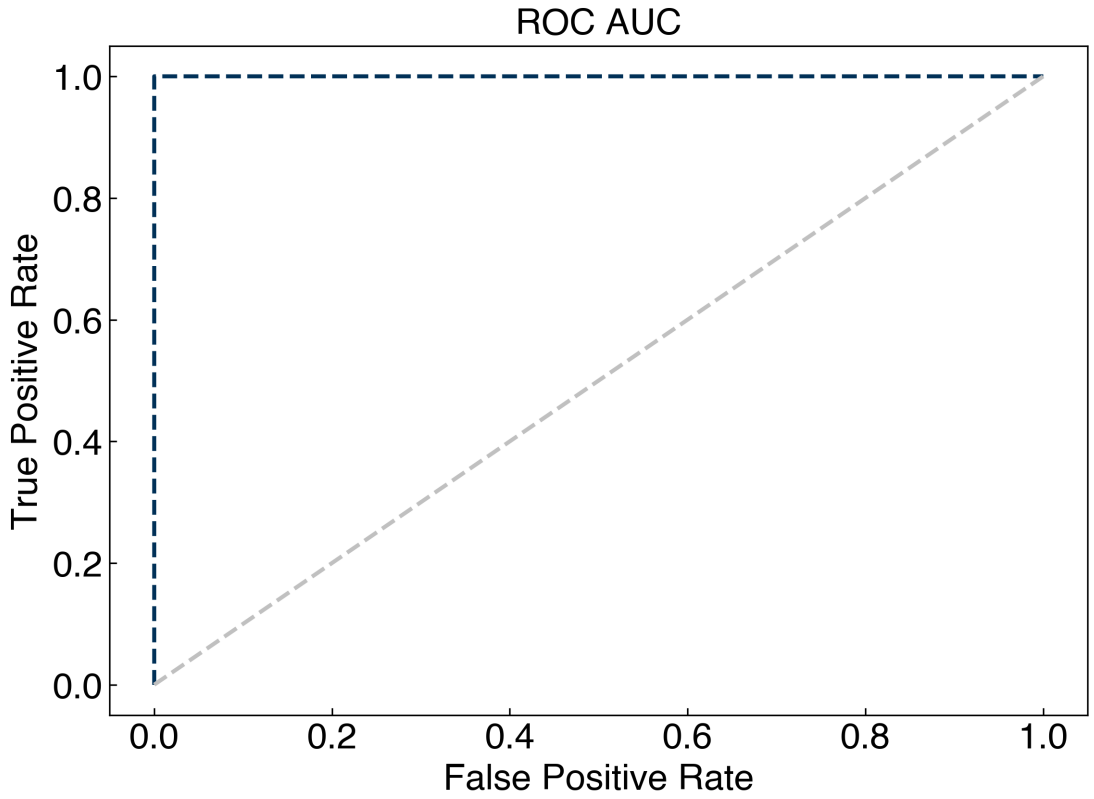
Receiver Operating Characteristic (ROC) curves

The "receiver operating characteristic", or ROC curve, is useful for models where a threshold is used to tune the rate of false positives and false negatives. The area under the curve can be used as a metric for how well the model performs.

We will discuss this metric more once the meaning of a "threshold" has been described.

In [9]:
```
from sklearn.metrics import roc_curve
from sklearn.svm import SVC
from sklearn.linear_model import SGDClassifier
```

```
from sklearn.ensemble import RandomForestClassifier

svc = SVC()
sgd = SGDClassifier()
rf = RandomForestClassifier()

sgd.fit(X_blob2, y_blob2)
y_sgd = sgd.predict(X_blob2)

rf.fit(X_blob2, y_blob2)
y_rf = rf.predict(X_blob2)

fpr, tpr, threshold = roc_curve(y_blob2, y_rf)

fig, ax = plt.subplots()
ax.plot(fpr, tpr)
ax.plot(fpr, fpr, '#C0C0C0')
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.set_title('ROC AUC');
```



Confusion matrices

False positives and false negatives only apply to binary problems. The "confusion matrix" is a multi-class generalization of the concept, and can help identify which classes are "confusing" the algorithm.

In a confusion matrix the diagonal elements correspond to true positives and true negatives, while the off-diagonal elements correspond to false postives and false negatives, with false positives above the diagonal and false negatives below (or vice versa, depending on label definitions).

Cross-validation and Resampling

Similar to the case of regression, cross-validation is an important tool for classification models. The general idea is the same: We hide some of the data from the model when we train it, then we test the model on the hidden data.

However, there are a few wrinkles in classification that arise in the case of class imbalance. When constructing test/train sets, it is important to ensure that all classes are represented in a manner consistent with the overall set. This can be a major challenge if the classes are highly imbalanced. The best approach is to balance classes prior to cross validation. There are a few strategies for doing this:

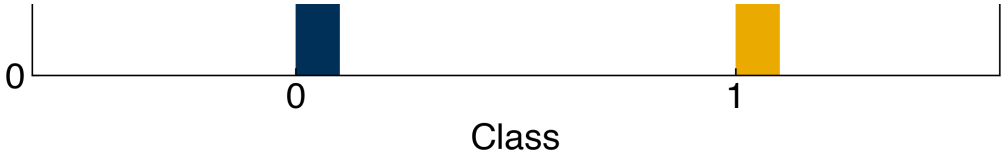
1) Re-balancing the cost function to penalize mis-classification of the under-represented class more. This works well, but requires that you know the relative importance of the 2 classes, and requires modifying the inner working of the algorithms. It also suffers from some of the same disadvantages as over-sampling.

2) Undersampling: discarding information from over-represented class. This is inefficient since not all information is used.

3) Oversampling: add repeates of the under-represented class (very similar to re-balancing the cost function). This can lead to over-fitting of the decision boundary to the few examples of the under-represented class.

4) Resampling: Re-sample from the under-represented class, but add some noise. This is a robust solution, but requires some knowledge of the distribution of the under-represented data (e.g. generative models) or special techniques (e.g. SMOTE).

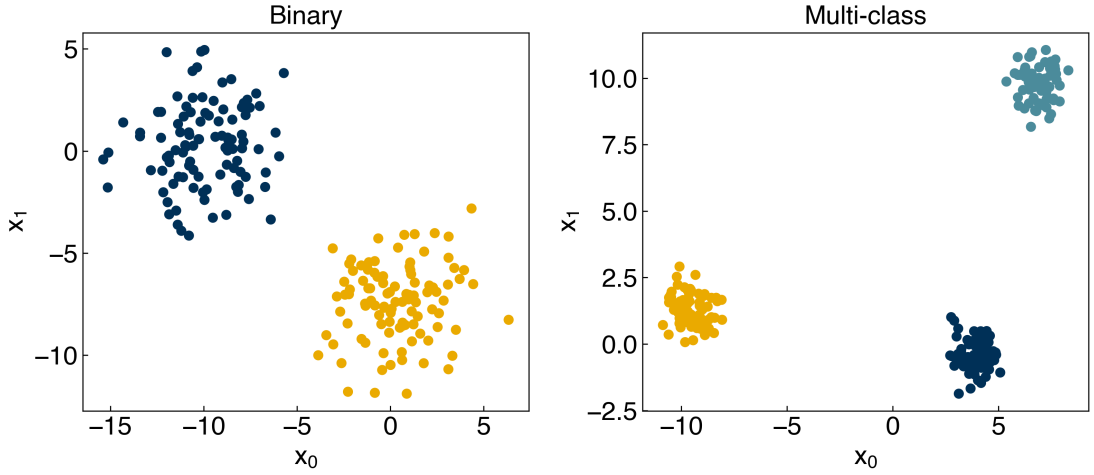
In [10]:
```
fig, ax = plt.subplots()

ax.hist(y_imbalanced[y_imbalanced == 0])
ax.hist(y_imbalanced[y_imbalanced == 1])
ax.set_xticks([0, 1])
ax.set_xlabel('Class')
ax.set_ylabel('Counts');
```



Multi-class classification

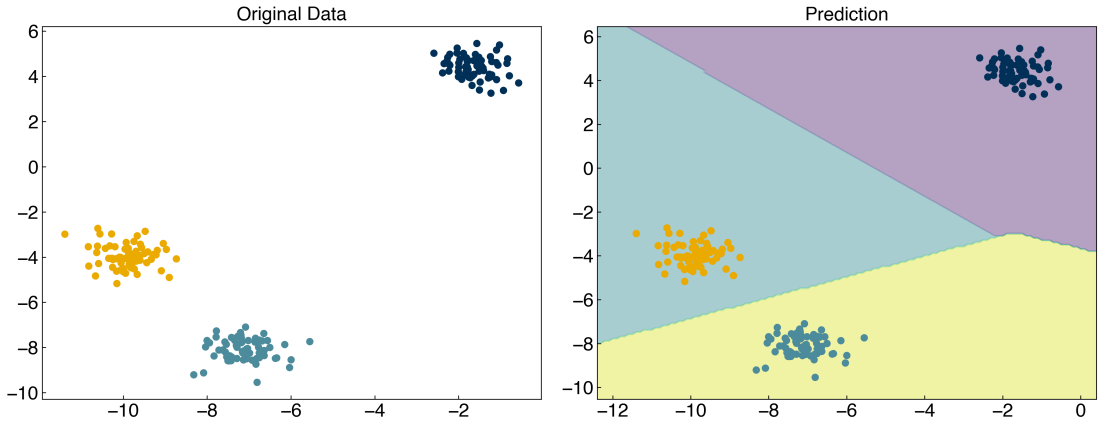
In [11]:
```
np.random.seed(9)
X_blob1, y_blob1 = make_blobs(n_samples = 200, centers = 2, cluster_std
= 2 * noisiness, n_features = 2)

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

In [12]:
```
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 [13]:
```
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');
```



Deriving a loss function for discrimination

So far we have not actually discussed how to create a classification model. We will start by investigating discriminative models, since these are most similar to regression. The difference is that instead of regressing a line that fits the input data, we are regressing a line that discriminates between the two classes. This requires a new loss function. We can derive one such loss function by considering the following mathematical definition of a classification model.

We will start by considering a discrimination problem:

$f(\vec{x})>p$ if class 1

$f(\vec{x})<p$ if class 2

and let $f(\vec{x}) = \bar{\bar{X}}\vec{w}$, where $\bar{\bar{X}} = [\vec{x}, \vec{1}]$ similar to linear regression.

We can use y as the output variable and arbitrarily assign "class 1" to 1 and "class 2" to -1, such that $p = 0$.

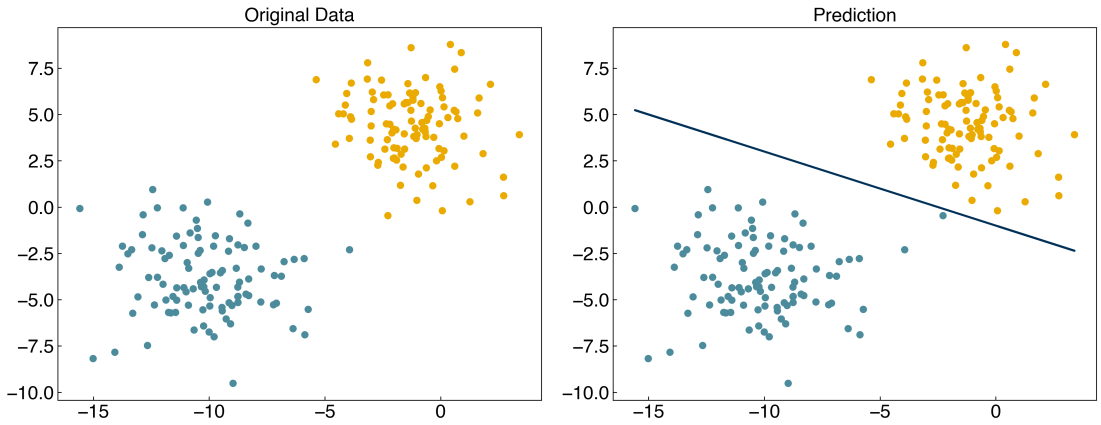
$\bar{\bar{X}}\vec{w}>0$ if $y_i = 1$ (class 1)

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

Let's take a look at this in code with some toy data:

In [14]:
```
def add_intercept(X):
    intercept = np.ones((X.shape[0],1))
    X_intercept = np.append(intercept,X,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');
```



Exercise: Derive the slope and intercept of 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 $\bar{\bar{X}} = [\vec{x_0}, \vec{x_1}, \vec{1}]$ and $\vec{w} = [w_0, w_1, w_2]$.

Then, the equation will be

$$x_0 w_0 + x_1 w_1 + w_2 = 0$$

$$x_1 = -\frac{w_0}{w_1}x_0 - \frac{w_2}{w_1}$$

The slope will be $-\frac{w_0}{w_1}$ and the intercept will be $-\frac{w_2}{w_1}$.

This looks a lot like linear regression, but we still need an **objective function**. This is where things get tricky. Based on the definition of $\pm$1 for classes, and the algebraic rules for inequalities, we can multiply by y_i and re-write this as a single line:

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

Convince yourself that this is true!

Now we can turn this into an equality by taking the maximum:

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

Now we are getting close. If a point y_i is mis-classified then this will give a positive value, but if it is correctly classified it will return zero. Therefore we can get a cost for the entire dataset by summing the function over all data points:

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

and we can find the optimal $\vec{w}$ by minimizing g with respect to $\vec{w}$

This is the "max cost" function, often commonly referred to as the "perceptron" model. We can implement this loss function:

In [15]:
```
def max_cost(w, X, 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
```

Counting loss function

We can also modify the loss function so that we count the number of points that are incorrect by taking the "sign" before summing over the points:

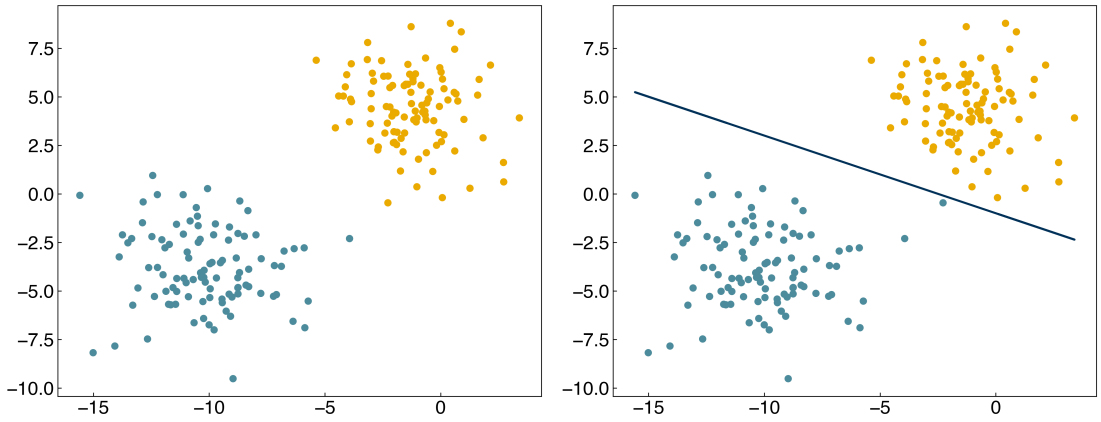
In [16]:
```
def n_wrong(w, X = X, y = y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept,w)
    return sum(np.maximum(0, np.sign(-y*Xb)))

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

```
1.0
```

In principle, we can also minimize this directly:

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

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 + 3) // 2])
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;
```

```
1.0
```



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: What are some differences between these two loss functions?

> The max cost function tells how far the input is from the discrimination line, while the counting loss function only tells the number of misclassification.

Table of Contents

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('../settings/plot_style.mplstyle')
```

In [2]:

```
import numpy as np
import pandas as pd

clrs = np.array(['#003057', '#EAAA00', '#4B8B9B', '#B
3A369', '#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:

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

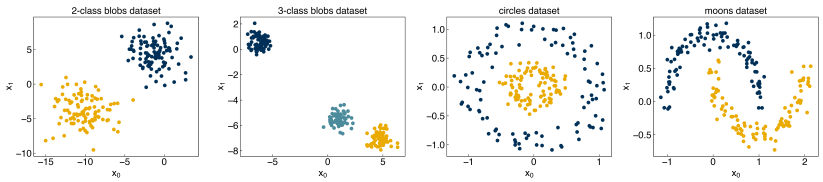
We will apply this to the toy datasets:

In [3]:

```
from sklearn.datasets.samples_generator import make_b
lobs, 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, clu
ster_std=0.5*noisiness, n_features=2)

X_circles, y_circles = make_circles(n_samples=200, fa
ctor=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_ci
rcles, y_circles], [X_moons, y_moons]]
```

```
titles = ['2-class blobs dataset', '3-class blobs dat
aset', 'circles dataset', 'moons dataset']

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[i].set_title(titles[i])

plt.show()
```

```
/Users/aj/anaconda3/envs/OSI/lib/python3.8/site-packa
ges/sklearn/utils/deprecation.py:144: FutureWarning:
The sklearn.datasets.samples_generator module is  dep
recated in version 0.22 and will be removed in versio
n 0.24. The corresponding classes / functions should
instead be imported from sklearn.datasets. Anything t
hat cannot be imported from sklearn.datasets is now p
art of the private API.
  warnings.warn(message, FutureWarning)
```



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
```

However, before applying the model, we need to "re-scale" the class data to fit the assumption we made. By default, `scikit-learn` uses $y = 0$ and $y = 1$ to define the two different classes. However, when we derived the loss function we assumed that the classes were defined by $y = 1$ and $y = -1$, with the discrimination line at $y = 0$.

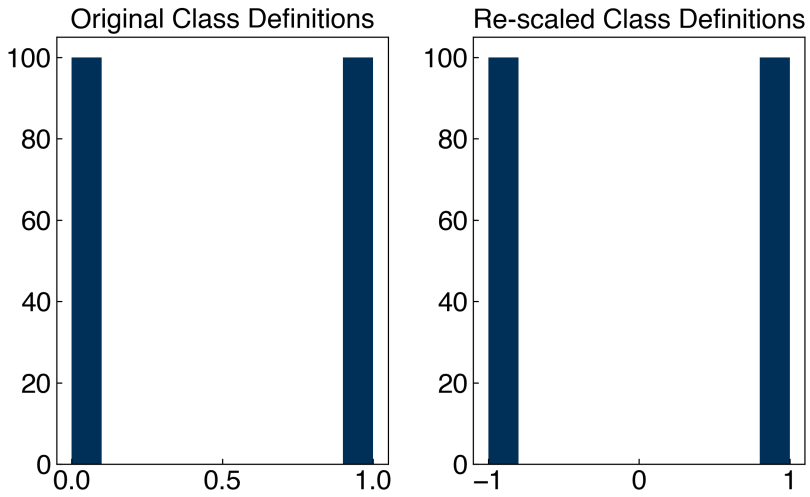
In [5]:

```
fig, axes = plt.subplots(1,2, figsize=(8,5))

X = X_blob
y = y_blob
axes[0].hist(y)
axes[0].set_title('Original Class Definitions')
```

```
y = y_blob*2 - 1 #convert to -1, 1
axes[1].hist(y)
axes[1].set_title('Re-scaled Class Definitions')
```

Out[5]:

```
Text(0.5, 1.0, 're-scaled Class Definitions')
```

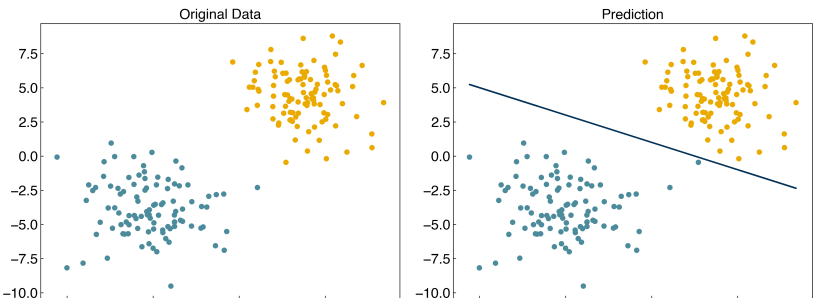


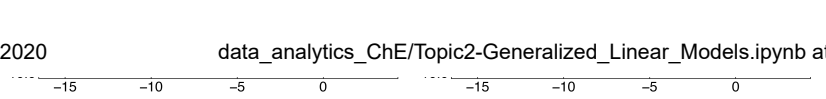
Note that this re-scaling is only necessary if we are using our own model. The `scikit-learn` implementation is clever enough to take care of this automatically.

Now we can select some arbitrary parameters and evaluate the model:

In [6]:

```
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 to compute the cost of a given set of parameters:

In [7]:

```
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 [8]:

```
from scipy.optimize import minimize

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

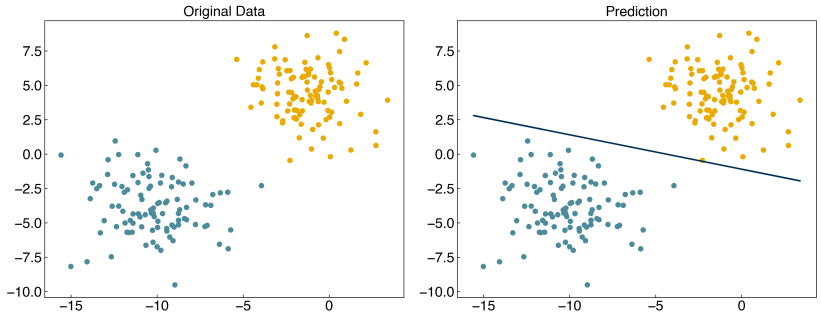
Out[8]:

```
      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,  -9.679
40886])
```

In [9]:

```
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?

- The first derivative of the max cost loss function is non differentiable.
- An initial guess of $\overrightarrow{0}$ makes the max cost zero, which is a trivial solution.

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 $\overrightarrow{w} = 0$.

(2) The max function is not differentiable.

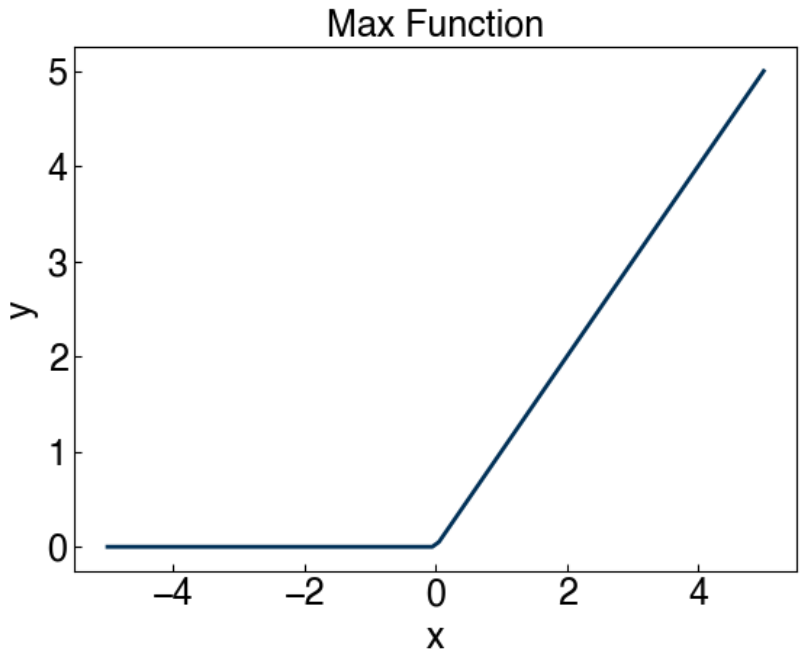
We can overcome the second problem by creating some smooth approximation of the maximum function. This is achieved using the "softmax" function:

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

In [10]:

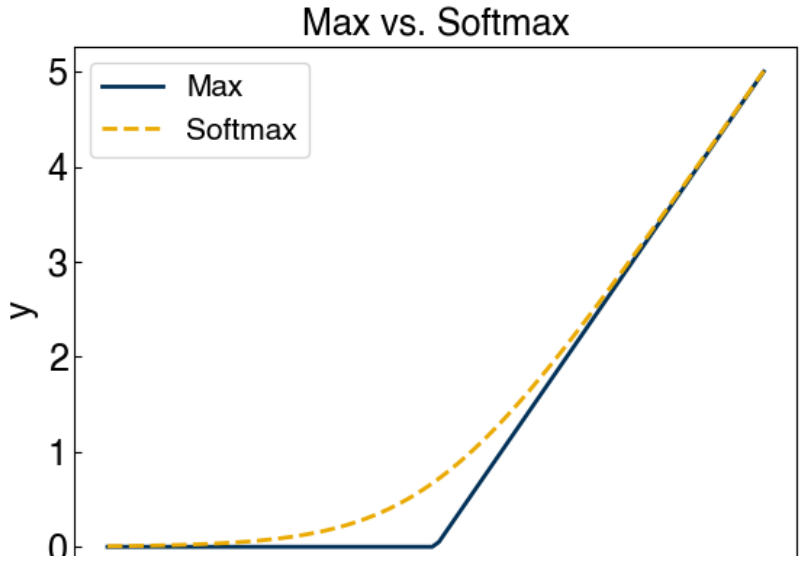
```
x = np.linspace(-5, 5, 100)

fig, ax = plt.subplots(figsize = (6, 5), dpi = 100)
```

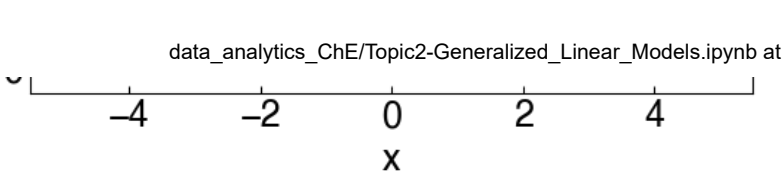
```
ax.plot(x, np.maximum(0, x), ls = '-')

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



In [11]:

```
x = np.linspace(-5, 5, 100)

fig, ax = plt.subplots(figsize = (6, 5), dpi = 100)

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 [12]:

```
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 $\vec{w}$:

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

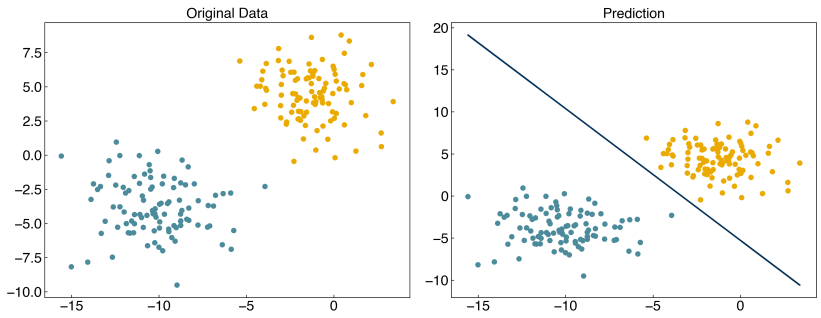
It turns out this problem is not linear, and needs to be solved iteratively using e.g. Newton's method. The math is a little more complex than before, so we won't cover it in lecture, but it is covered in Ch. 4 of "Machine Learning Refined" if you are interested. This approximation is called **logistic 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 [13]:

```
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 "moons" dataset.

In [14]:

```
from scipy.optimize import minimize

y_moons_scaled = 2 * y_moons - 1

w = [-10, 10, 5]

result = minimize(max_cost, w, args = (X_moons, y_moo
ns_scaled))
print('Perceptron loss function: {}'.format(max_cost(
result.x, X_moons, y_moons_scaled)))

result = minimize(softmax_cost, w, args = (X_moons, y
_moons_scaled))
print('Logistic regression loss function: {}'.format(
softmax_cost(result.x, X_moons, y_moons_scaled)))
```

```
Perceptron loss function: 4.5796169252392186e-07
Logistic regression loss function: 53.939495485292476
```

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.

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 [15]:

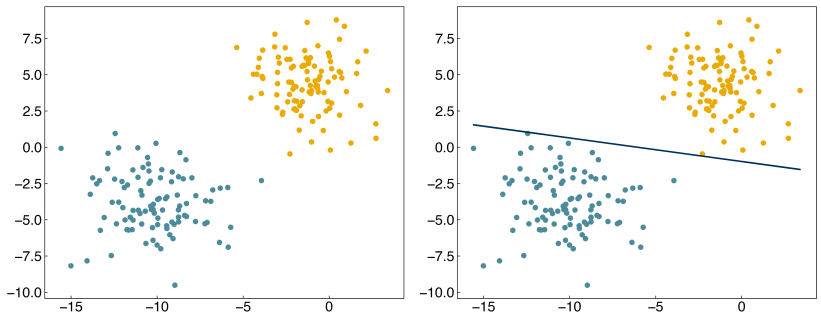
```
def margin_cost(w, X = X, y = y):
    X_intercept = add_intercept(X)
    Xb = np.dot(X_intercept,w)
    return sum(np.maximum(0, 1 - y * Xb))

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

```
13950.826906338369
```

In [16]:

```
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 = '-');
```



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

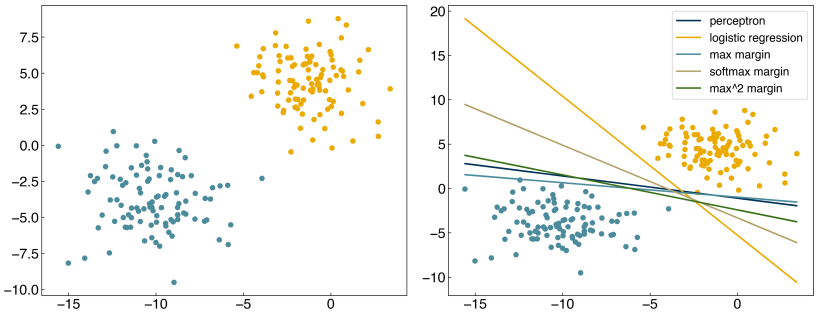
In [17]:

```
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 [18]:

```
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 = label)

labels = ['perceptron', 'logistic regression', 'max m
argin', 'softmax margin', 'max^2 margin']
w_set = [w_perceptron, w_logit,w_opt_margin, w_opt_ma
rgin2, w_opt_softmax]
```

```
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?

> It's hard to tell in this case, but I would choose the softmax margin model.

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

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:

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

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

Let's use the new regularized cost function:

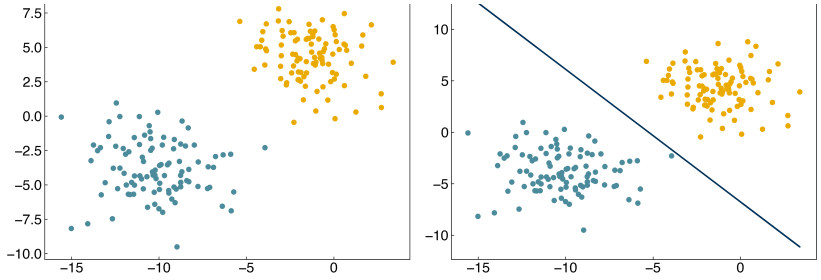
In [19]:

```
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
```

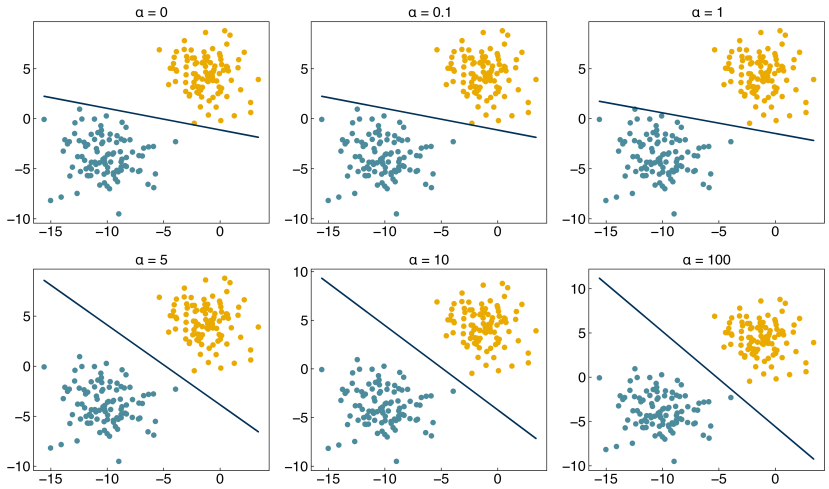
and optimize it with the minimize function:

In [20]:

```
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');
```



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



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.

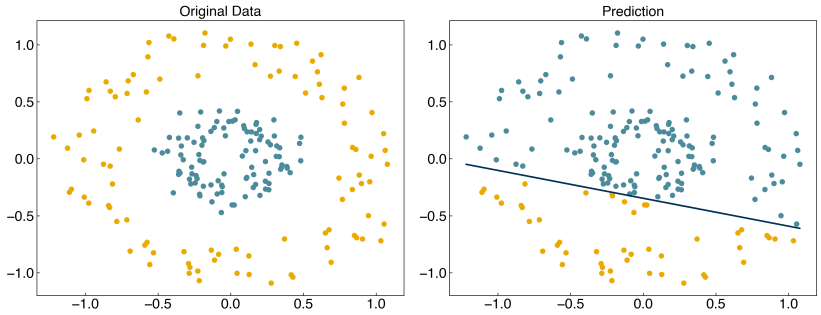
Non-linearity and Kernels

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 [21]:

```
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');
```



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:

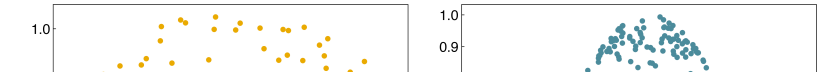
In [22]:

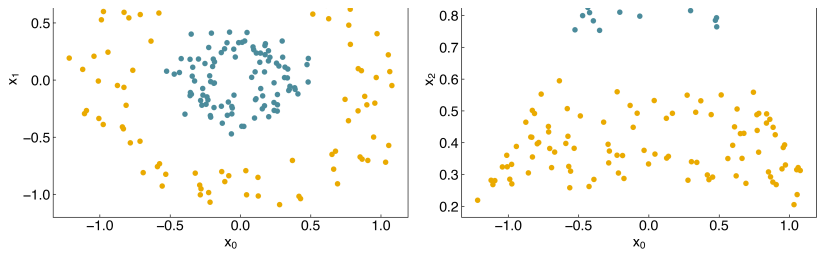
```
X_new = np.exp(-(X[:,0]**2 + X[:,1]**2))
X_new = X_new.reshape(-1, 1)
X_nonlinear = np.append(X, X_new, 1)
print(X_nonlinear.shape)
```

```
(200, 3)
```

In [23]:

```
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('$x_2$');
```



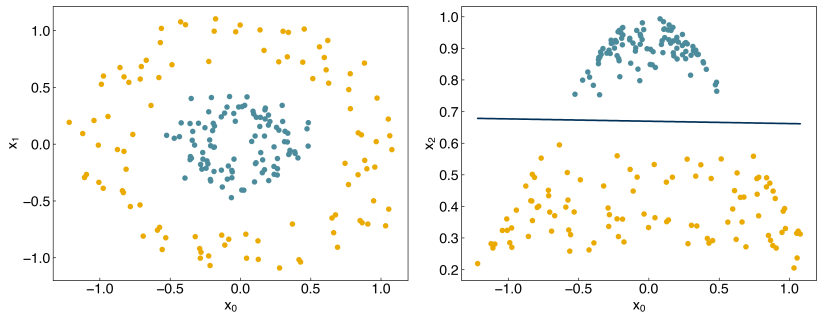


We see that the dataset is now linearly separable in this transformed space!

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

In [24]:

```
w_guess = np.array([-10, -4, 0, -10]) #note that we h
ave an extra parameter 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_circles + 1])
axes[1].scatter(X_nonlinear[:, 0], X_nonlinear[:, 2],
c = clrs[prediction + 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('$x_2$');
```



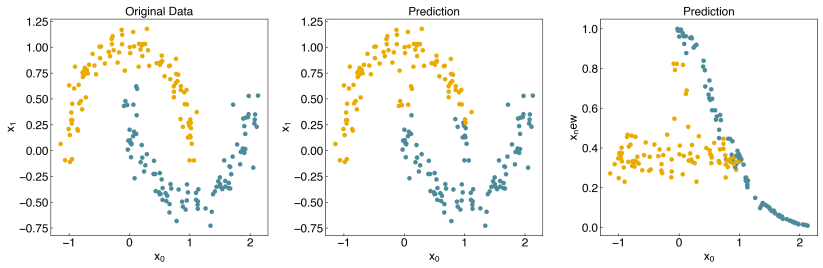
Now the model is able to correctly classify the non-linearly separable dataset! The kernel has created a new, higher-dimensional transformed space:

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

In [25]:

In [25]:

```
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[prediction + 1])
axes[2].scatter(X_nonlinear[:, 0], X_nonlinear[:, 2],
c = clrs[prediction + 1])

axes[0].set_xlabel('$x_0$')
axes[0].set_ylabel('$x_1$')
axes[0].set_title('Original Data')

axes[1].set_xlabel('$x_0$')
axes[1].set_ylabel('$x_1$')
axes[1].set_title('Prediction')

axes[2].set_xlabel('$x_0$')
axes[2].set_ylabel('$x_new$')
axes[2].set_title('Prediction');
```



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":

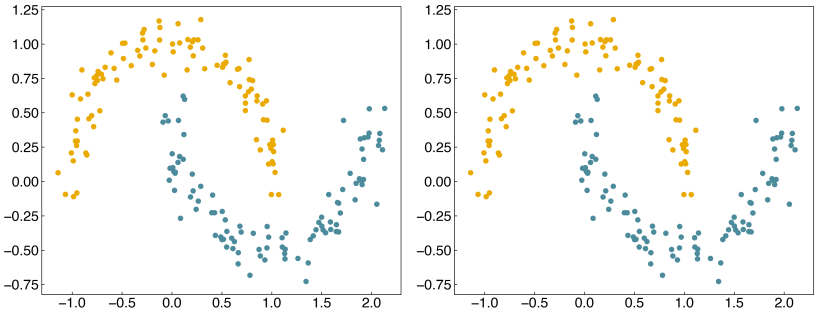
In [26]:

```
from sklearn.metrics.pairwise import rbf_kernel

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

```
(200, 200)
```

In [27]:

```
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[prediction + 1]);
```



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

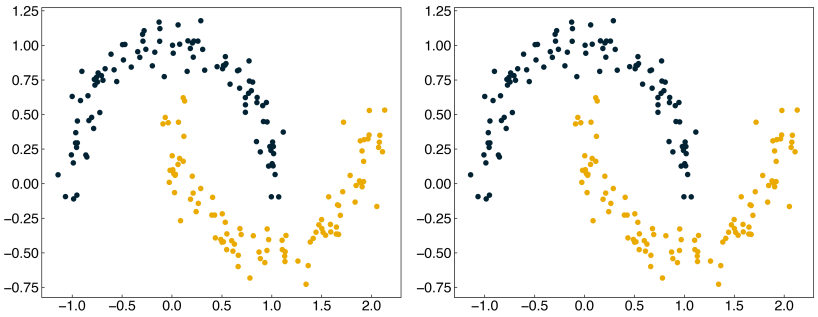
> It is a non-parametric model. Since the RBF kernel matrix takes account of distances between data points, the number of parameters will increase as the data gets bigger. In this case, we have used all of the data to train the model, meaning the number of parameters is equal to the number of data points, and the model is over-fitted. Therefore, it is unlikely to generalize unless we use regularization or cross-validation.

We can make this process easier and add regularization by using the SVM model from `scikit-learn` (note that it is called a support vector "classifier", or SVC). The kernel width is controlled by `gamma`, and the regularization strength is controlled by `C`:

In [28]:

```
from sklearn.svm import SVC # "Support vector classif
ier"

model = SVC(kernel = 'rbf', gamma = 1, 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
]);
```



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.

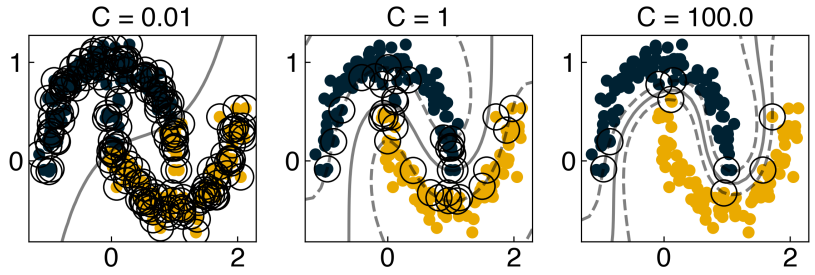
In [29]:

```
def plot_svc_decision_function(model, ax=None, plot_s
upport=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)
```

Let's use this function to see how the decision boundary and the support vectors change as a function of `C`:

In [30]:

```
fig, axes = plt.subplots(1,3, figsize=(8,3))

Cs = [1e-2, 1, 1e2]

for i, Ci in enumerate(Cs):
    model = SVC(kernel = 'rbf', gamma = 1, C = Ci)
    model.fit(X, y)
    y_predict = model.predict(X)
    ax_i = axes[i]
    ax_i.scatter(X[:, 0], X[:, 1], c = clrs[y_predict
], s = 50, cmap = 'RdBu')
    plot_svc_decision_function(model, ax = ax_i)
    ax_i.set_title('C = {}'.format(Ci))
```



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

As C goes to infinity, a small margin is accepted so the decision boundary will be very complex.
The boundary will be complex as well if γ goes to infinity. In this case, the boundaries will enclose just one data point. Look at the plots above and play with the values of `C` and `gamma` to get a better feel for how the boundary changes.

Note that there are more "support vectors" as C gets smaller (or as alpha gets larger). This is because when there is more regularization, more points are allowed to be in the "margins".

Table of Contents

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('../settings/plot_style.mplstyle')
```

In [2]:

```
import numpy as np
import pandas as pd

clrs = np.array(['#003057', '#EAAA00', '#4B8B9B', '#B
3A369', '#377117', '#1879DB', '#8E8B76', '#F5D580',
'#002233'])
```

Alternate classification methods

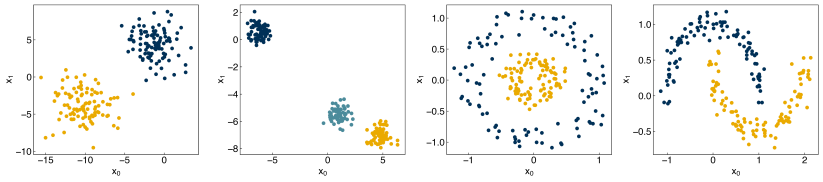
In the prior lectures we have discussed discriminative "generalized linear" models in depth, and derived and studied the "support vector machine" model. In this lecture we will look at some fundamentally different approaches to the classification problem. While we will not derive these methods or study them in depth, we will introduce the key concepts and see how they work for the toy datasets.

In [3]:

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/sklearn/utils/deprecation.py:143: FutureWarning: The sklearn.datasets.samples_generator module is deprecated in version 0.22 and will be removed in version 0.24. The corresponding classes / functions should instead be imported from sklearn.datasets. Anything that cannot be imported from sklearn.datasets is now part of the private API.
  warnings.warn(message, FutureWarning)
```



k-Nearest Neighbors

An alternative non-linear classification method is the k-nearest neighbors algorithm. This operates on a principle that is very easy to understand: democracy.

The class of a point is determined by letting its k-nearest neighbors "vote" on which class it should be in. The point is

assigned to whichever class has the most votes. In the case of a tie, k is decreased by 1 until the tie is broken.

The advantage of democracy is that it is "nonlinear" - we can distinguish classes with very complex structures.

We need 3 functions to implement kNN:

- distance metric - calculate the distance between 2 points. We will use the Euclidean distance.
- get neighbors - find the k points nearest to a given point.
- assign class - poll the neighbors to assign the point to a class

In [4]:

```
def distance(x1, x2):
    # we will use the numpy 2-norm to calculate Eucli
dean distance:
    return np.linalg.norm(x1 - x2, 2) #<- the 2 is op
tional here since 2 is the default.
```

In [5]:

```
def get_neighbor_idxs(x, x_list, k):
    dist_pairs = []
    for i,xi in enumerate(x_list):
        dist = distance(x, xi)
        dist_pairs.append([dist, i, xi]) #<- gives us
the distance for each point
    dist_pairs.sort() #<- sort by distance
    k_dists = dist_pairs[:k] #<- take the k closest p
oints
    kNN_idxs = [i for di, i, xi in k_dists] #<- we wi
ll get the indices of neighbors instead of the point
 itself.
    return kNN_idxs
```

In [6]:

```
from scipy.stats import mode

def assign_class(x, x_list, y_list, k): #<- now we ne
ed to know the responses
    neighbors = get_neighbor_idxs(x, x_list, k)
    y_list = list(y_list) #<- this ensures that index
ing works properly if y_list is a `pandas` object.
    votes = [y_list[i] for i in neighbors]
    assignment = mode(votes)[0][0] #<- we won't deal
 with ties for this simple implementation
    return assignment
```

Now we can "wrap" all of these functions into a single function that predicts the class of an array of points:

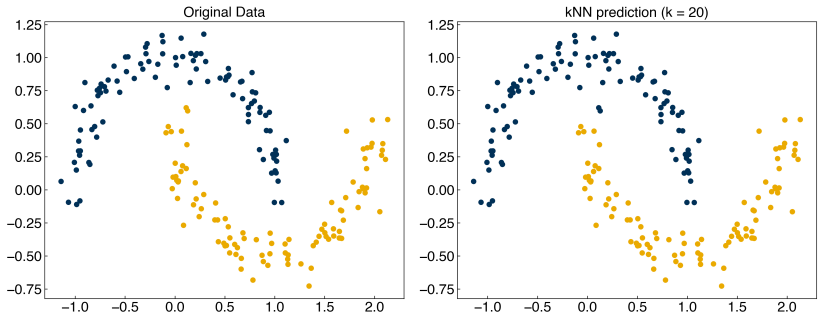
In [7]:

```
def kNN(X, k, X_train, y_train):
    y_out = []
    for xi in X:
        y_out.append(assign_class(xi, X_train, y_train, k))
    y_out = np.array(y_out)
    return y_out
```

Let's see how this works on some of our toy datasets:

In [8]:

```
X = X_moons
y = y_moons

y_knn = kNN(X, 20, X, y)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X[:, 0], X[:, 1], c = clrs[y])
axes[0].set_title('Original Data')
axes[1].scatter(X[:, 0], X[:, 1], c = clrs[y_knn])
axes[1].set_title('kNN prediction (k = 20)');
```



We see that the kNN model predicts the classes perfectly!

Discussion: Do you think this kNN model is reliable for new predictions?

> No. This model may be overfitted to the whole X data.

Let's try to visualize the model:

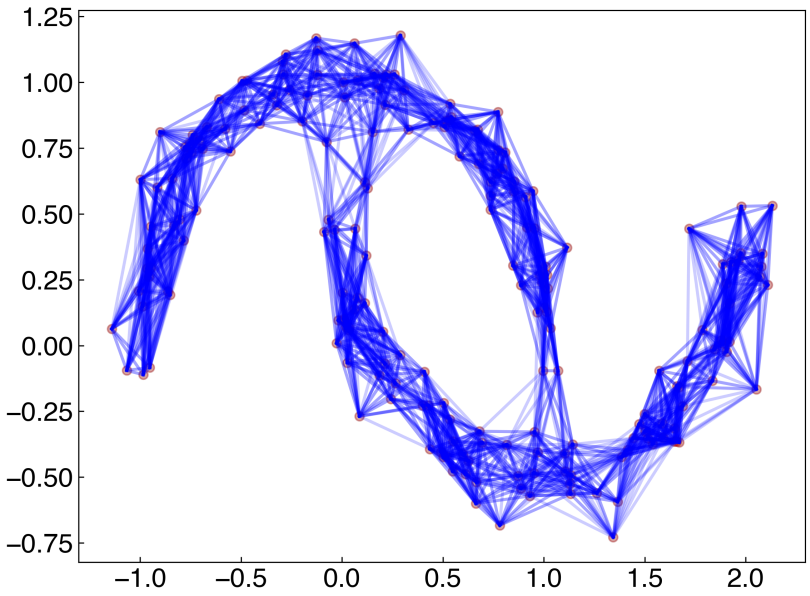
In [9]:

```
def visualize_neighbors(X, X_train, k):
    ## helper function to visualize neighbors
    fig, ax = plt.subplots()
    ax.scatter(X[:, 0], X[:, 1], color = 'k', alpha = 0.2)
    ax.scatter(X_train[:, 0], X_train[:, 1], color = 'r', alpha = 0.2)
    for xi in X:
        neighbors = get_neighbor_idxs(xi, X_train, k)
```

```
        neighbors = get_neighbor_idxs(xi, X_train, k)
        for nj in neighbors:
            xj = X_train[nj]
            ax.plot([xi[0], xj[0]], [xi[1], xj[1]], l
s = '-', color = 'b', alpha = 0.2)
```

In [10]:

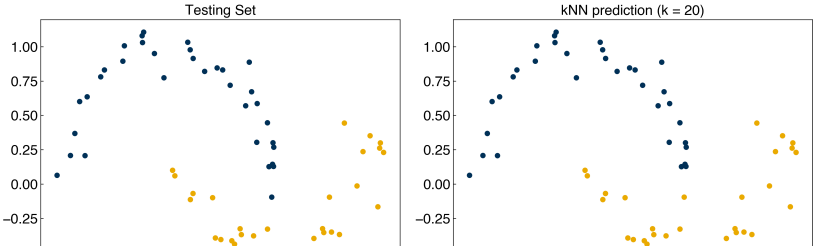
```
visualize_neighbors(X, X, 20)
```

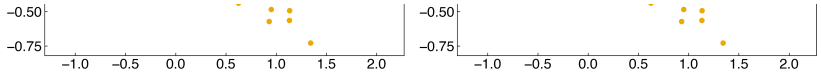


The model works well, but all of the points are training points. This means that the model has "memorized" the classes of each point, and then just uses all the data to predict the class. Let's see what happens if we use a test-train split:

In [11]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X
, y, test_size = 0.33)

y_knn = kNN(X_test, 20, X_train, y_train)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_test[:,0], X_test[:,1], c = clrs[y_
test])
axes[0].set_title('Testing Set')
axes[1].scatter(X_test[:, 0], X_test[:, 1], c = clrs[
y_knn])
axes[1].set_title('kNN prediction (k = 20)');
```



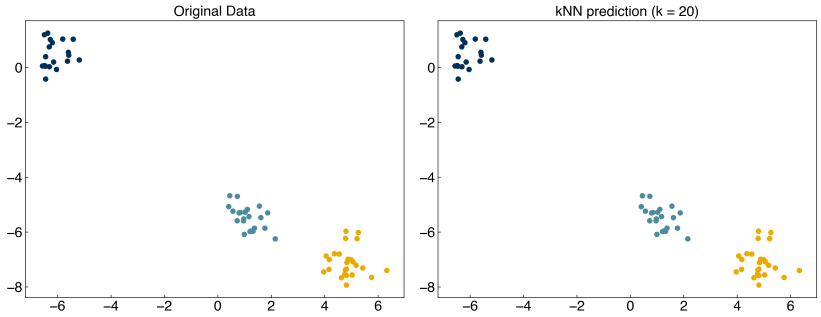


We see that the model still performs very well, even for the testing data.

The kNN approach is very flexible, and can easily be extended to multi-class datasets, although "ties" between different classes may become more likely.

In [12]:

```
X = X_mc
y = y_mc

X_train, X_test, y_train, y_test = train_test_split(X
, y, test_size = 0.33)

y_knn = kNN(X_test, 20, X_train, y_train)

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_test[:, 0], X_test[:, 1], c = clrs[
y_test])
axes[0].set_title('Original Data')
axes[1].scatter(X_test[:, 0], X_test[:, 1], c = clrs[
y_knn])
axes[1].set_title('kNN prediction (k = 20)');
```



Discussion: Which part of the code is training the model? What are the hyperparameters of the model?

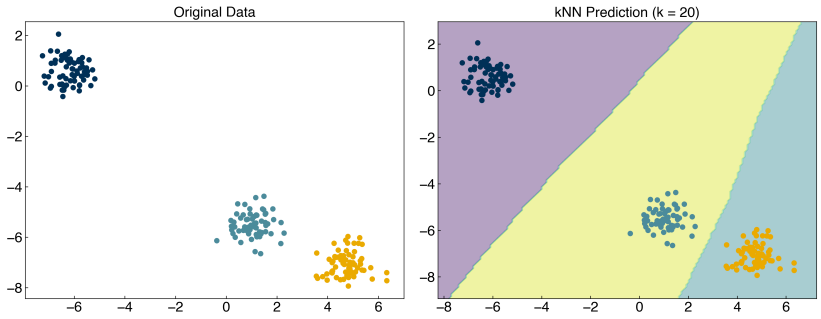
> We actually do not train the model. There is no loss function.
> k and distance metrics are commonly tuned hyperparameters.

We can also use the `scikit-learn` implementation:

In [13]:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors = 20)
```

```
knn = KNeighborsClassifier(n_neighbors = 20)
knn.fit(X, y)
y_predict = knn.predict(X)

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

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

Z = knn.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[:, 0], X[:, 1], c = clrs[y_predict
])
axes[0].set_title('Original Data')
axes[1].set_title('kNN Prediction (k = 20)');
```



This will make it easier to optimize hyperparameters and compare models. It is also generally more efficient.

kNN advantages and disadvantages

Advantages

- Simple to understand/implement
- Intuitive
- Highly non-linear class boundaries

Disadvantages

- Slow for large training sets and/or high dimensions
- Difficult to interpret in high dimensions

Naive Bayes Classification

A totally different approache is the "Naive Bayes" classifier, which is a generative model. The model is "naive" because it naively assumes that the data in each class follows a Gaussian distribution, and that the features are not correlated.

Under these assumptions, the Gaussian distribution for each class

gives the probability function for y (class) as a function of X (features):

$$P(\vec{x}|y_i) \sim \exp\left(\sum_j \frac{(x_j - \mu_i)^2}{2\sigma_i^2}\right)$$

where μ_i is the mean (centroid) of class i, and σ_i is the standard deviation of class i.

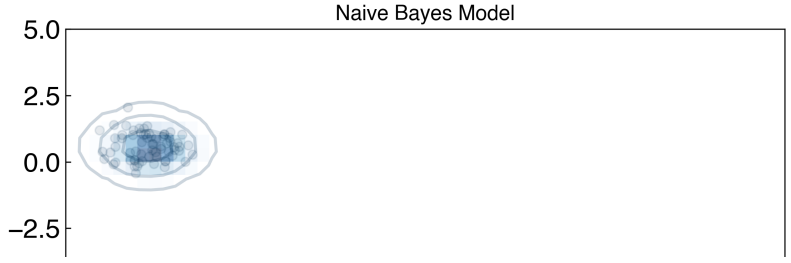
This can be used with Bayes' formula to estimate $P(y_i|\vec{x})$:

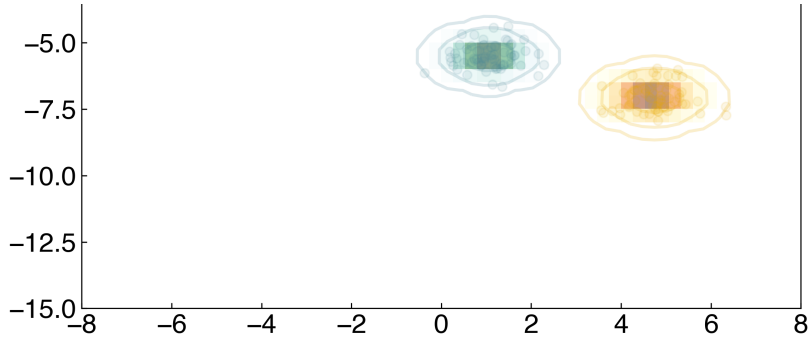
$$P(y_i|\vec{x}) = \frac{P(\vec{x}|y_i)P(y_i)}{P(\vec{x})}$$

The code block below will visualize this. You don't need to understand how it works, but as usual should understand the output:

In [14]:

```
fig, ax = plt.subplots()

ax.set_title('Naive Bayes Model', size = 14)

xlim = (-8, 8)
ylim = (-15, 5)

xg = np.linspace(xlim[0], xlim[1], 60)
yg = np.linspace(ylim[0], ylim[1], 40)
xx, yy = np.meshgrid(xg, yg)
Xgrid = np.vstack([xx.ravel(), yy.ravel()]).T

cmaps = ['Blues', 'YlOrBr', 'BuGn']
## Don't worry if this doesn't exactly make sense
for label, color in enumerate(clrs[:3]):
    mask = y == label
    ax.scatter(X[mask][:, 0], X[mask][:, 1], c = colo
r, alpha = 0.1)
    mu, std = X[mask].mean(0), X[mask].std(0) #<- her
e is the key part: take the mean/stdev of each class.
    P = np.exp(-0.5 * (Xgrid - mu) ** 2 / std ** 2).p
rod(1)
    Pm = np.ma.masked_array(P, P < 0.03)
    ax.pcolorfast(xg, yg, Pm.reshape(xx.shape), alpha
= 0.5, cmap = cmaps[label])
    ax.contour(xx, yy, P.reshape(xx.shape),
               levels = [0.01, 0.1, 0.5, 0.9],
               colors = color, alpha = 0.2)
```



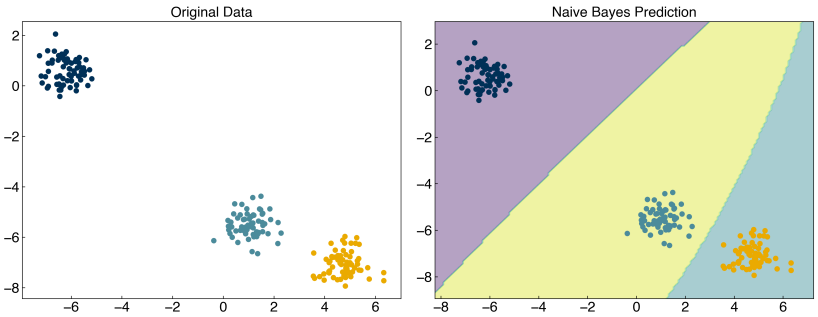


This is a *generative model* because we are now (assuming) we know the distribution from which each class was drawn. We can then use this probability distribution to assess the relative probability that any new point belongs to each class.

We will not implement the model here, but will use the `scikit-learn` implementation:

In [15]:

```
from sklearn.naive_bayes import GaussianNB

NB = GaussianNB()
NB.fit(X, y)
y_predict = NB.predict(X)

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

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

Z = NB.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[:, 0], X[:, 1], c = clrs[y_predict
])
axes[0].set_title('Original Data')
axes[1].set_title('Naive Bayes Prediction');
```



Naive Bayes works well for linearly-separable datasets where the classes follow Gaussian distributions, but will not work well for highly non-linear datasets. There is a "not so naive" extension that

highly non-linear datasets. There is a "not so naive" extension that can be used once more sophisticated models for the underlying class distributions are known, but this is beyond the scope of this lecture.

Discussion: Is Naive Bayes a parametric or non-parametric model?

> It is a parametric model. Naïve Bayes models need the mean and standard deviation for each feature as parameters. Therefore, the number of parameters will depend on the number of features, which will not change due to the size of the dataset.

Naive Bayes advantages and disadvantages:

Advantages:

- Difficult to overfit
- Probabilistic predictions
- Can easily re-sample to correct class imbalance

Disadvantages:

- Decision boundaries are quadratic (often nearly linear in practice)

Decision Trees

Decision trees are a very powerful type of **discriminative** classification algorithm, and they are relatively easy to interpret. They also have the advantage of working well with discrete input variables (e.g. discrete feature spaces). Essentially, a decision tree checks each input variable and attempts to make a discrete "cut" in that variable to decide which class it belongs in. It then repeates this process with other variables until the training data can be separated into the correct classes. In a sense, it breaks the problem into a lot of 1-dimensional classification models and repeats the process recursively. The way that the decision point is determined is slightly different than what we saw for generalized linear models, and is usually based on information theory concepts like gini criteria or information entropy. We will not go into the details here.

The disadvantage of decision trees is that they are very prone to over-fitting, because variables can be used more than once. The "Random forest" approach overcomes this by training an

ensemble of decision trees with subsets of the data (similar to the "bootstrapping" we saw before) and using this ensemble of models to produce an estimate.

We will not go into the theory of decision trees here, but we will show a brief example using the toy datasets with the `scikit-learn` implementation.

In [16]:

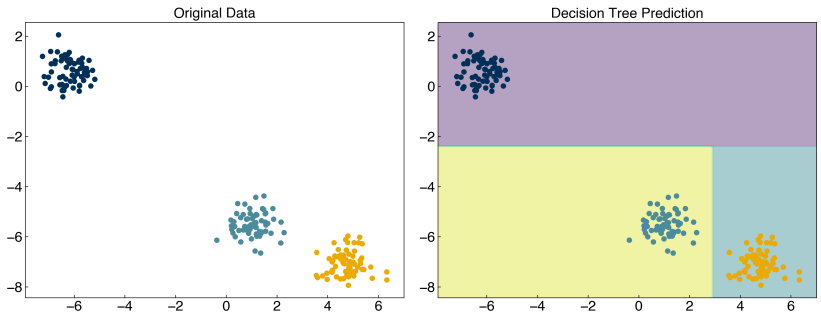
```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier()
```

In [17]:

```
X = X_mc
y = y_mc
```

In [18]:

```
tree.fit(X, y)
y_tree = tree.predict(X)

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

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

Z = tree.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[:, 0], X[:, 1], c = clrs[y_tree])

bottom, top = axes[0].get_ylim()
axes[1].set_ylim(bottom, top)

left, right = axes[0].get_xlim()
axes[1].set_xlim(left, right)

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

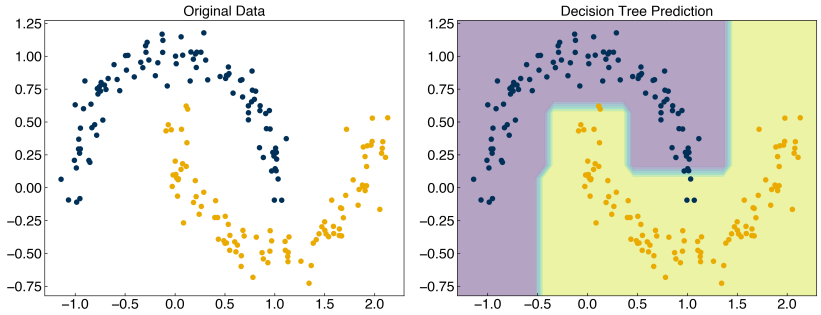


In [19]:

```
X = X_moons
y = y_moons
```

In [20]:

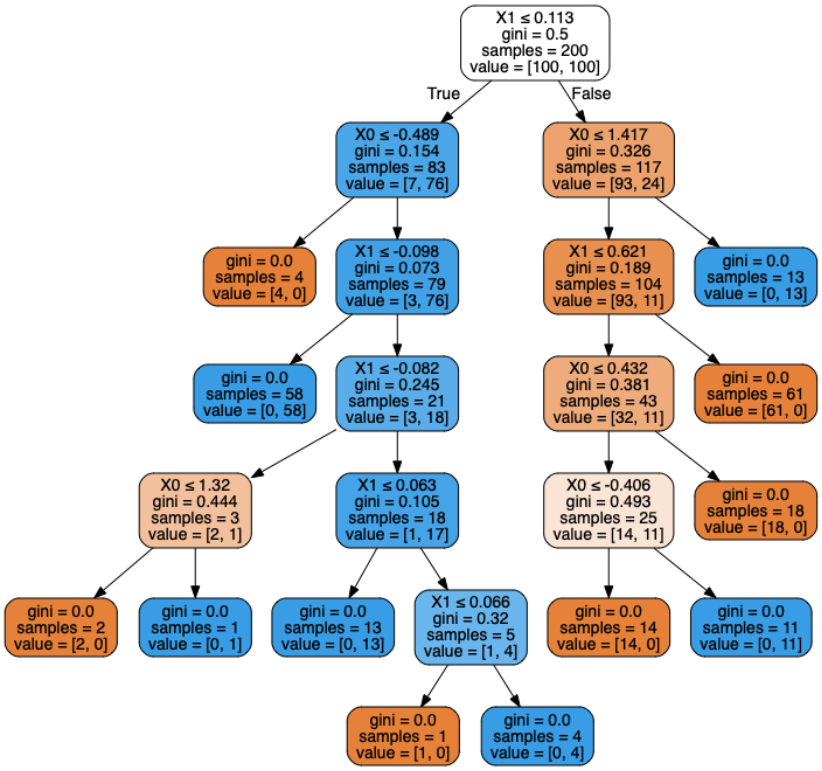
```
tree.fit(X, y)
y_tree = tree.predict(X)

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

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

Z = tree.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[:, 0], X[:, 1], c = clrs[y_tree])

bottom, top = axes[0].get_ylim()
axes[1].set_ylim(bottom, top)

left, right = axes[0].get_xlim()
axes[1].set_xlim(left, right)

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



In [21]:

```
from io import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

dot_data = StringIO()
export_graphviz(tree, out_file = dot_data,
                filled = True, rounded = True,
                special_characters = True)
graph = pydotplus.graph_from_dot_data(dot_data.getval
ue())
Image(graph.create_png())
```

Out[21]:



Decision trees advantages and disadvantages:

Advantages:

- Intuitive interpretation of model
- Able to predict highly non-linear boundaries

Disadvantages:

- Easily over-fit
- Training can be expensive for large or high-dimensional data
- Predictions are discrete rather than continuous (boundaries are not smooth)

Conclusion

There are many types of classification algorithms available, but most are based on concepts similar to these. In practice, it is typical to compare the performance, training time, and prediction time for various classifiers to determine the best model, since it can be difficult to predict which approach will be best.

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('../settings/plot_style.mplstyle')
```

In [2]:

```
import numpy as np
import pandas as pd

clrs = np.array(['#003057', '#EAAA00', '#4B8B9B', '#B
3A369', '#377117', '#1879DB', '#8E8B76', '#F5D580',
'#002233'])
```

Perovskite dataset

We will work with a real dataset, taken from the paper "New tolerance factor to predict the stability of perovskite oxides and halides (https://advances.sciencemag.org/content/5/2/eaav0693)" by Bartel et al. Perovskites are a very useful class of oxide materials that can have very high catalyic activities, and have recently shown promise in solar cells. However, not all combinations of elements will form a perovskite structure, and it is very useful to be able to predict whether or not an elemental composition will form a perovskite. This helps determine whether or not a material can be synthesized before going into the lab.

This dataset contains a list of chemical formulas, along with some chemical features of the elements (radius and oxidation state), and whether or not they will form a stable perovskite crystal structure:

In [3]:

```
df = pd.read_csv('data/perovskite_data.csv')

df.head(10)
```

Out[3]:

	ABX3	A	B	X	nA	nB	nX	rA (Ang)	rB (Ang)	rX (Ang)	t
0	AgBiO3	Ag	Bi	O	1	5	-2	1.28	0.76	1.40	0.88
1	AgBrO3	Ag	Br	O	1	5	-2	1.28	0.31	1.40	1.11
2	AgCaCl3	Ag	Ca	Cl	1	2	-1	1.28	1.00	1.81	0.78
3	AgCdBr3	Ag	Cd	Br	1	2	-1	1.28	0.95	1.96	0.79
4	AgClO3	Ag	Cl	O	1	5	-2	1.28	0.12	1.40	1.25
5	AgCoF3	Ag	Co	F	1	2	-1	1.28	0.74	1.33	0.89

6	AgCuF3	Ag	Cu	F	1	2	-1	1.28	0.73	1.33	0.90
7	AgMgCl3	Ag	Mg	Cl	1	2	-1	1.28	0.72	1.81	0.86
8	AgMgF3	Ag	Mg	F	1	2	-1	1.28	0.72	1.33	0.90
9	AgMnF3	Ag	Mn	F	1	2	-1	1.28	0.83	1.33	0.85

In [4]:

```
feature_columns = ['nA', 'nB', 'nX', 'rA (Ang)', 'rB
 (Ang)', 'rX (Ang)', 't', 'tau']

X_perov = df[feature_columns].values
y_perov = df['exp_label'].values

print(X_perov.shape, y_perov.shape)
```

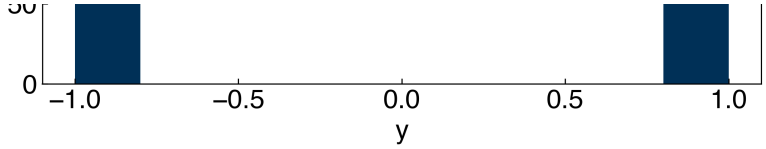
```
(576, 8) (576,)
```

A few things to note on this dataset:

- We only take the continuous variables to form our feature matrix, X.
- The `A`, `B`, `X` columns determine which elements are in the structure, `nA`, `nB`, `nX` columns are the formal oxidation state of each element, and `rA`, `rB`, and `rX` columns are the radii of each element (in Angstrom).
- There are also two additional columns, `t` and `tau`, which are "derived features" that are described in the original publication.
- The outputs, y, are in [-1, 1] and are not perfectly evenly distributed.

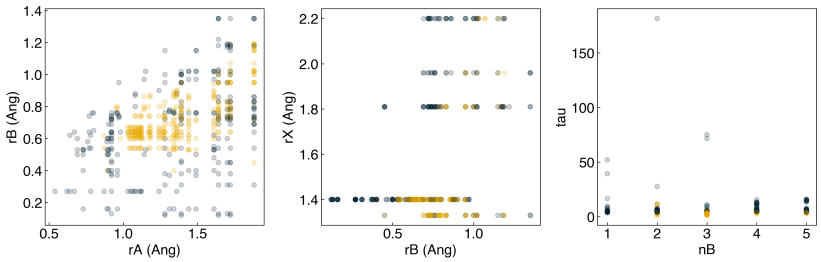
In [5]:

```
fig, axes = plt.subplots()
axes.hist(y_perov)
axes.set_xlabel('y')
axes.set_ylabel('Counts')
plt.show()
```



In [6]:

```
fig, axes = plt.subplots(1, 3, figsize = (15, 5))

axes[0].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[y_perov], alpha = 0.2)
axes[0].set_xlabel(feature_columns[3])
axes[0].set_ylabel(feature_columns[4])

axes[1].scatter(X_perov[:, 4], X_perov[:, 5], c = clr
s[y_perov], alpha = 0.2)
axes[1].set_xlabel(feature_columns[4])
axes[1].set_ylabel(feature_columns[5])

axes[2].scatter(X_perov[:, 1], X_perov[:, 7], c = clr
s[y_perov], alpha = 0.2)
axes[2].set_xlabel(feature_columns[1])
axes[2].set_ylabel(feature_columns[7]);
```



Kernel-based Models

In [7]:

```
from sklearn.metrics.pairwise import rbf_kernel
from sklearn.metrics import accuracy_score, confusion
_matrix
```

RBF Kernel

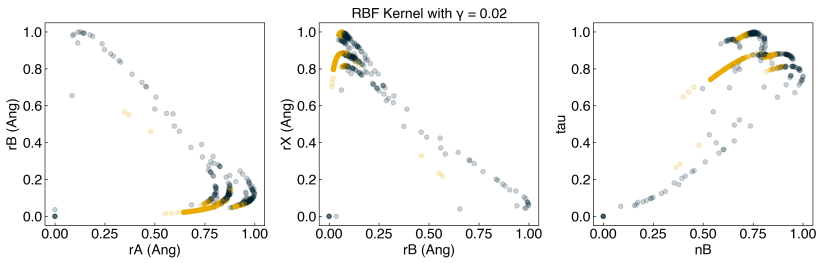
In [8]:

```
X_kernel = rbf_kernel(X_perov, X_perov, gamma = 0.02)
```

In [9]:

```
fig, axes = plt.subplots(1, 3, figsize = (15, 5))

axes[0].scatter(X_kernel[:, 3], X_kernel[:, 4], c = c
lrs[y_perov], alpha = 0.2)
axes[0].set_xlabel(feature_columns[3])
axes[0].set_ylabel(feature_columns[4])
```

```
axes[1].scatter(X_kernel[:, 4], X_kernel[:, 5], c = c
lrs[y_perov], alpha = 0.2)
axes[1].set_xlabel(feature_columns[4])
axes[1].set_ylabel(feature_columns[5])

axes[2].scatter(X_kernel[:, 1], X_kernel[:, 7], c = c
lrs[y_perov], alpha = 0.2)
axes[2].set_xlabel(feature_columns[1])
axes[2].set_ylabel(feature_columns[7])

axes[1].set_title('RBF Kernel with $\gamma$ = 0.02');
```



Support vector machine

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 np.array(p > 0, dtype = int)
```

In [12]:

```
def regularized_cost(w, X = X_perov, y = y_perov, alp
ha = 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
```

In [13]:

```
from scipy.optimize import minimize
```

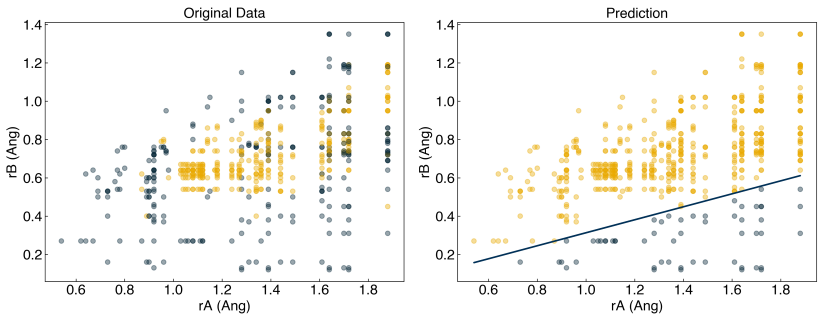
Train SVM with the original data

In [14]:

```
w_guess = np.array([-10, -4, -10])
result = minimize(regularized_cost, w_guess, args = (
X_perov[:, 3:5], y_perov, 1))
w_svm = result.x
```

In [15]:

```
prediction = linear_classifier(X_perov[:, 3:5], w_svm
)
prediction = 2 * prediction - 1

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[y_perov], alpha = .4)
axes[0].set_xlabel(feature_columns[3])
axes[0].set_ylabel(feature_columns[4])

axes[1].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[prediction], alpha = .4)
axes[1].set_xlabel(feature_columns[3])
axes[1].set_ylabel(feature_columns[4])

m = -w_svm[1] / w_svm[2]
b = -w_svm[0] / w_svm[2]
axes[1].plot(X_perov[:, 3], m * X_perov[:, 3] + b, ls
= '-')

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



In [16]:

```
print(accuracy_score(y_perov, prediction))
```

```
0.6302083333333334
```

Train SVM after kernel transformation

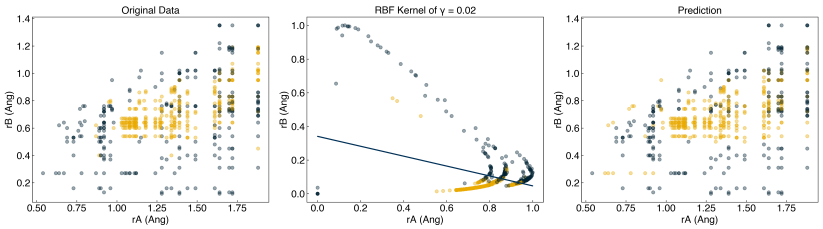
In [17]:

```
w_guess = np.array([-10, -4, -10])
result = minimize(regularized_cost, w_guess, args = (
X_kernel[:, 3:5], y_perov, 1))
w_svm = result.x
```

In [18]:

In [18]:

```
prediction = linear_classifier(X_kernel[:, 3:5], w_sv
m)
prediction = 2 * prediction - 1

fig, axes = plt.subplots(1, 3, figsize = (21, 6))
axes[0].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[y_perov], alpha = .4)
axes[0].set_xlabel(feature_columns[3])
axes[0].set_ylabel(feature_columns[4])

axes[1].scatter(X_kernel[:, 3], X_kernel[:, 4], c = c
lrs[y_perov], alpha = .4)
axes[1].set_xlabel(feature_columns[3])
axes[1].set_ylabel(feature_columns[4])

axes[2].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[prediction], alpha = .4)
axes[2].set_xlabel(feature_columns[3])
axes[2].set_ylabel(feature_columns[4])

m = -w_svm[1] / w_svm[2]
b = -w_svm[0] / w_svm[2]
axes[1].plot(X_kernel[:, 3], m * X_kernel[:, 3] + b,
ls = '-')

axes[0].set_title('Original Data')
axes[1].set_title('RBF Kernel of $\gamma$ = 0.02')
axes[2].set_title('Prediction');
```



In [19]:

```
print(accuracy_score(y_perov, prediction))
```

```
0.8472222222222222
```

scikit-learn SVC

In [20]:

```
from sklearn.svm import SVC
```

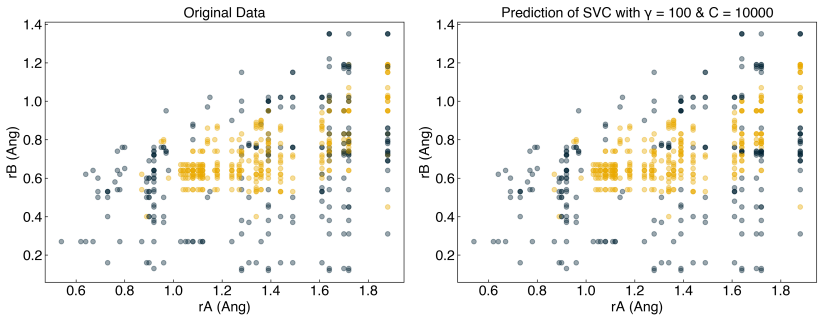
Use only the 4th & 5th feature

In [21]:

```
model = SVC(kernel = 'rbf', gamma = 100, C = 1000)
model.fit(X_perov[:, 3:5], y_perov)
y_predict = model.predict(X_perov[:, 3:5])
```

```
print(model.score(X_perov[:, 3:5], y_perov))

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[y_perov], alpha = .4)
axes[0].set_xlabel(feature_columns[3])
axes[0].set_ylabel(feature_columns[4])

axes[1].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[y_predict], alpha = .4)
axes[1].set_xlabel(feature_columns[3])
axes[1].set_ylabel(feature_columns[4])

axes[0].set_title('Original Data')
axes[1].set_title('Prediction of SVC with $\gamma$ =
 100 & C = 10000');
```

```
0.9253472222222222
```

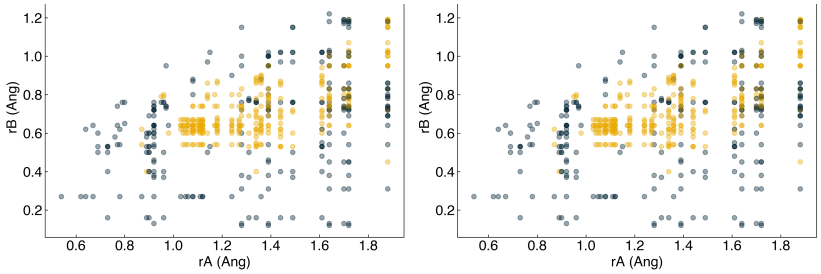


Use the whole features

In [22]:

```
model = SVC(kernel = 'rbf', gamma = 100, C = 1000)
model.fit(X_perov, y_perov)
y_predict = model.predict(X_perov)

print(model.score(X_perov, y_perov))

fig, axes = plt.subplots(1, 2, figsize = (15, 6))
axes[0].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[y_perov], alpha = .4)
axes[0].set_xlabel(feature_columns[3])
axes[0].set_ylabel(feature_columns[4])

axes[1].scatter(X_perov[:, 3], X_perov[:, 4], c = clr
s[y_predict], alpha = .4)
axes[1].set_xlabel(feature_columns[3])
axes[1].set_ylabel(feature_columns[4])

axes[0].set_title('Original Data')
axes[1].set_title('Prediction of SVC with $\gamma$ =
 100 & C = 10000');
```

```
0.9965277777777778
```



Comparing Model Performance

We can use a train/test split to check for over-fitting:

In [23]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X
_perov, y_perov, test_size=0.33)
```

Now, we can work only with the training set to optimize the hyperparameters of the model. Similar to the case of regression, we can take advantage of `GridSearchCV`:

In [24]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.utils import shuffle

X_train, y_train = shuffle(X_train, y_train) #Shuffle
everything just for good measure

sigmas = np.array([1e-3, 1e-2, 1e-1, 1, 10, 100])
gammas = 1. / 2 / sigmas**2

alphas = np.array([1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1
, 1])
Cs = 1 / alphas

parameter_ranges = {'C': Cs, 'gamma': gammas}

svc = SVC(kernel = 'rbf')

svc_search = GridSearchCV(svc, parameter_ranges, cv =
3)
svc_search.fit(X_train, y_train)
svc_search.best_estimator_, svc_search.best_score_
```

Out[24]:

```
(SVC(C=10000.0, gamma=0.005), 0.9376614987080103)
```

Let's investigate the performance of this model on the validation set:

In [25]:

```
best_svc = svc_search.best_estimator_

y_predict = best_svc.predict(X_test)

best_svc.score(X_test, y_test)
```

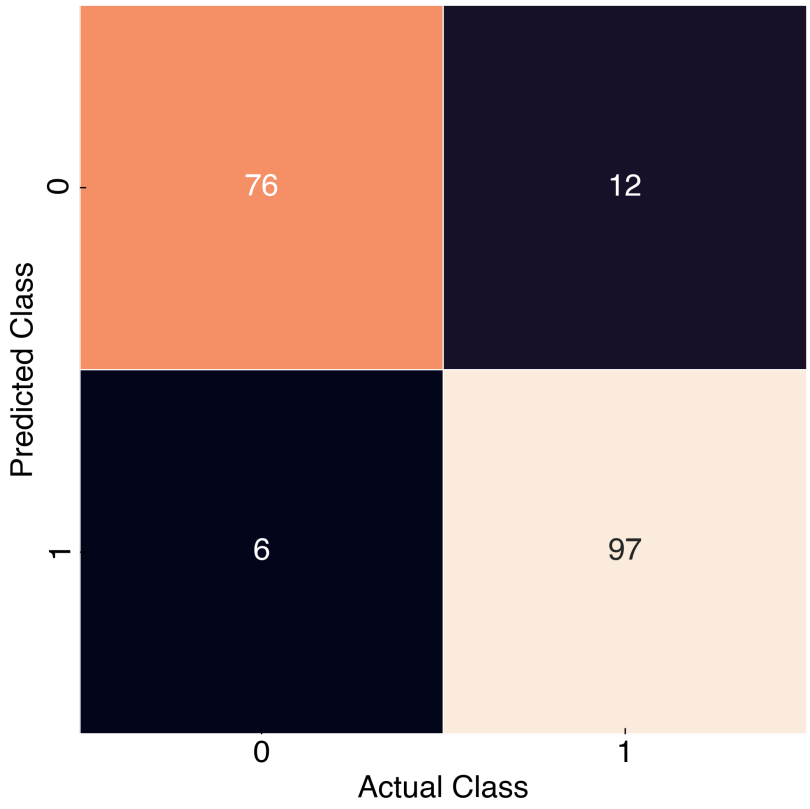
Out[25]:

```
0.9057591623036649
```

We see that it works pretty well. We can also visualize the performance using a confusion matrix:

In [26]:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y_test, y_predict)

fig, ax = plt.subplots(figsize = (7, 7))
sns.heatmap(cm, annot = True, linewidth = .45, cbar = False)
ax.set_xlabel('Actual Class')
ax.set_ylabel('Predicted Class');
```



Exercise: Calculate the accuracy, precision, and recall for the best SVC model on the perovskite dataset.

In [27]:

```
tn, fp, fn, tp = cm.reshape(-1, )

accuracy = (tn + tp) / (tn + fp + fn + tp)
# precision = ?
# recal = ?
```

Decision tree

In [28]:

```
from sklearn.tree import DecisionTreeClassifier

dtree=DecisionTreeClassifier()
dtree.fit(X_train,y_train)
y_predict = dtree.predict(X_train)

cm_train = confusion_matrix(y_train, y_predict)
```

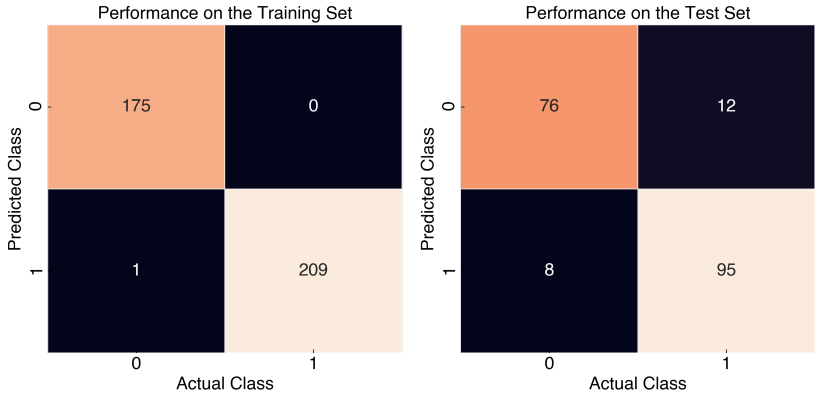
We see that the model performs very well on the training set. However, what we really want to check is the test set:

In [29]:

```
y_predict = dtree.predict(X_test)
cm_test = confusion_matrix(y_test, y_predict)
```

In [30]:

```
fig, axes = plt.subplots(1, 2, figsize = (12, 6))
sns.heatmap(cm_train, annot = True, cbar = False, lin
ewidth = .5, ax = axes[0], fmt = 'd')
sns.heatmap(cm_test, annot = True, cbar = False, line
width = .5, ax = axes[1], fmt = 'd')

axes[0].set_xlabel('Actual Class')
axes[0].set_ylabel('Predicted Class')
axes[0].set_title('Performance on the Training Set')

axes[1].set_xlabel('Actual Class')
axes[1].set_ylabel('Predicted Class')
axes[1].set_title('Performance on the Test Set');
```



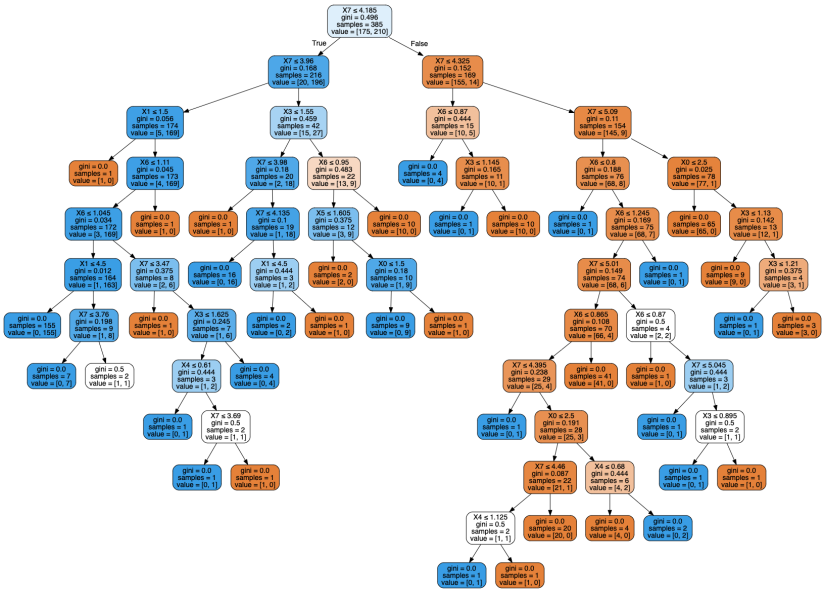
We see that the performance is worse, but not too bad. Let's

visualize the model to get some intuition about how it is working:

In [31]:

```
from io import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

dot_data = StringIO()
export_graphviz(dtree, out_file = dot_data,
                filled = True, rounded = True,
                special_characters = True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

Out[31]:



We see that the tree is very complicated! However, we see that we can "read" the tree and make intuitive sense of how the model works. This is one of the greatest strengths of the decision tree. We can also control the complexity of the tree by limiting its maximum depth:

In [32]:

```
dtree = DecisionTreeClassifier(max_depth = 3)
dtree.fit(X_train,y_train)
y_predict = dtree.predict(X_train)

cm_train = confusion_matrix(y_train, y_predict)
```

We see that the training performance is slightly poorer if the max depth is limited. However, the test performance is comparable:

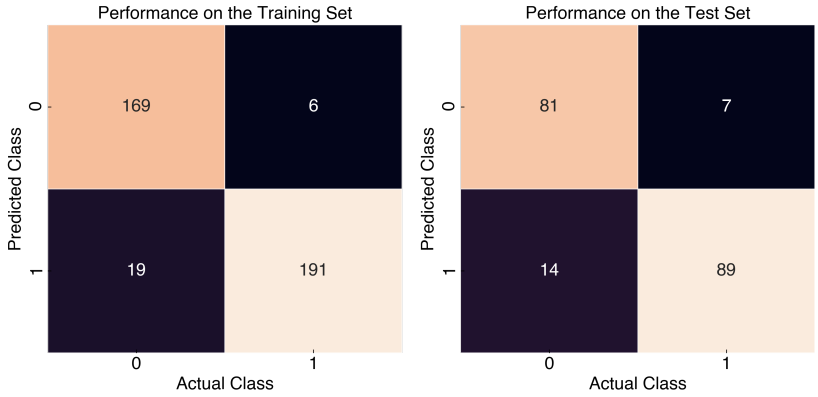
In [33]:

```
y_predict = dtree.predict(X_test)
```

```
cm_test = confusion_matrix(y_test, y_predict)
```

In [34]:

```
fig, axes = plt.subplots(1, 2, figsize = (12, 6))
sns.heatmap(cm_train, annot = True, cbar = False, lin
ewidth = .5, ax = axes[0], fmt = 'd')
sns.heatmap(cm_test, annot = True, cbar = False, line
width = .5, ax = axes[1], fmt = 'd')

axes[0].set_xlabel('Actual Class')
axes[0].set_ylabel('Predicted Class')
axes[0].set_title('Performance on the Training Set')

axes[1].set_xlabel('Actual Class')
axes[1].set_ylabel('Predicted Class')
axes[1].set_title('Performance on the Test Set');
```

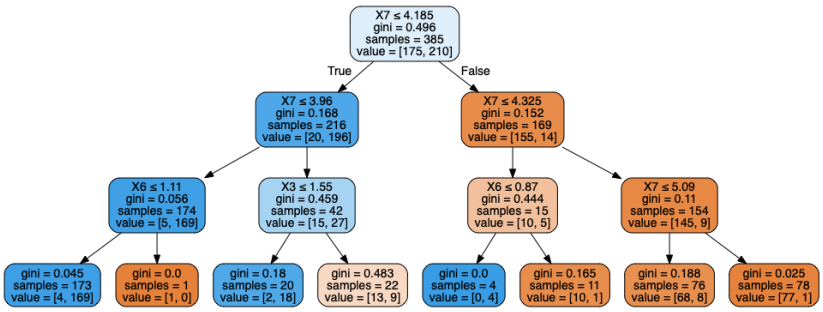


We can also visualize the tree:

In [35]:

```
dot_data = StringIO()
export_graphviz(dtree, out_file = dot_data,
                filled = True, rounded = True,
                special_characters = True)
graph = pydotplus.graph_from_dot_data(dot_data.getval
ue())
Image(graph.create_png())
```

Out[35]:



Discussion: Which variable is most important for determining whether or not a material will form a perovskite?

x_7 appears the most times in this decision tree.