In [1]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

Classification

This notebook covers classification metrics and introduces Logistic Regression.

Classification is the general term for supervised learning problems where we try to predict the value of a categorical variable. That is, we want to build a model f that approximates the relationship between features X and labels y so that

$$f(X_j) = y_j$$

for most observations (X_j, y_j). Here the y_j come from a finite set of labels, and we often refer to the groups defined by these labels as classes. We say that a classification problem is **binary** if there are two classes, and **multiclass** otherwise.

To do machine learning, we need to be more explicit and replace "most observations" with a concrete metric. The most obvious metric is **accuracy**, which is defined as the number of correct predictions divided by the total number of observations. However, there are many alternatives which are more suitable in different circumstances.

Precision and Recall

The biggest problem with accuracy is that it will let us down in situations where one class occurs with much higher frequency than another. For example, if we're trying to build an e-mail spam detector, and it turns out that 99% of all e-mails aren't spam, then we can get a model with 99% accuracy by ignoring all features and labeling all e-mails as not spam. Obviously this isn't helpful.

Instead, we can focus on the uncommon class and ask questions like "if my model marks an e-mail as spam, how likely is it to actually be spam?" or "what percent of spam e-mails are correctly labeled by my model?" These are the concepts of **precision** and **recall**, respectively.

More generally, whenever we have **unbalanced classes**, we can pick one class to be the **positive class** and then organize our data as follows:

	Observation Positive	**Observation Negative**
Prediction Positive	True Positive	False Positive (Type I)
Prediction Negative	False Negative (Type II)	True Negative

- The **Precision** is true positives divided by all positive *predictions*
- The **Recall** is true positives divided by all positive *observations*.

Questions:

1. What's the interpretation of precision and recall?
2. When would you want high precision? High recall?
3. Is Harvard's admission's process high precision or high recall? (positive class: applicant is qualified)
4. What about Sir Blackstone's aphorism "Better that ten guilty persons escape than that one innocent suffer" with Captain Louis Renault's order to "Round up the Usual Suspects" in the film "Casablanca"? (positive class: person/suspect is guilty)

Other Classification Metrics

- There is also **F-beta** score which gives a weighted harmonic mean between the precision and recall (as a function of β) and the **F-1** score is the special case when $\beta = 1$.
- The **Jaccard Similarity Coefficient** is in general the intersection of the predicted and actual label set divided by the union. This is equivalent to the accuracy score for most classification problems.

In [2]:
```
from sklearn import metrics

y_obs  = [0, 0, 1, 1, 0, 1, 0, 1]
y_pred = [0, 0, 1, 1, 0, 0, 0, 1]

print("Accuracy:", metrics.accuracy_score(y_obs, y_pred))
print("Precision:", metrics.precision_score(y_obs, y_pred))
print("Recall:", metrics.recall_score(y_obs, y_pred))
print("F1:", metrics.f1_score(y_obs, y_pred))
print("Jaccard:", metrics.jaccard_similarity_score(y_obs, y_pred))
```

```
Accuracy: 0.875
Precision: 1.0
Recall: 0.75
F1: 0.8571428571428571
Jaccard: 0.875
```

In [3]:
```
# summary report
# http://scikit-learn.org/stable/modules/model_evaluation.html#classification-report
from sklearn.metrics import classification_report
print(classification_report(y_obs, y_pred))
```

```
             precision    recall  f1-score   support

          0       0.80      1.00      0.89         4
          1       1.00      0.75      0.86         4

avg / total       0.90      0.88      0.87         8
```

Probabilistic Models

Some classification models output probabilities instead of predicting labels directly. For example, given a binary classification problem with labels 0 and 1, we may build a model f whose output we interpret as

$$f(X_j) = P(y_j = 1)$$

for each observation (X_j, y_j).

Precision-Recall Tradeoffs

We can make class predictions by comparing probabilities to some threshold p. We predict $y_j = 1$ when $f(X_j) > p$ and $y_j = 0$ when $f(X_j) \leq p$. We can compute a precision and recall score for each choice of p, so varying p gives us a family (or curve) or precision-recall pairs.

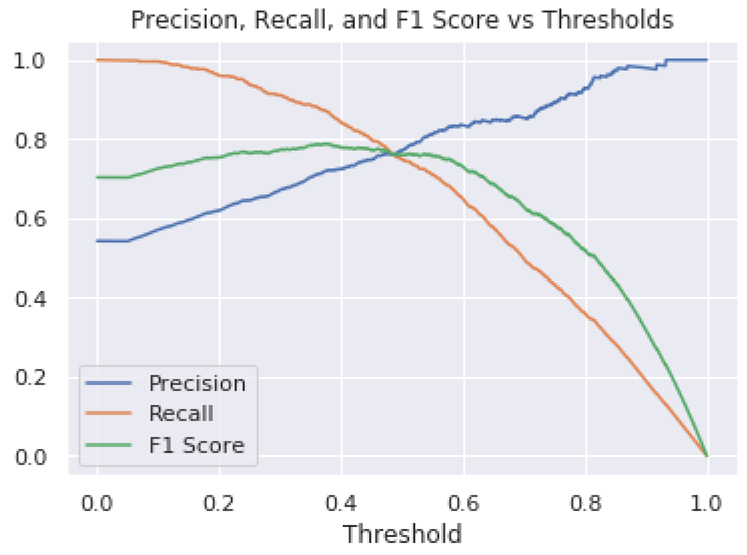
Questions (Part 1):

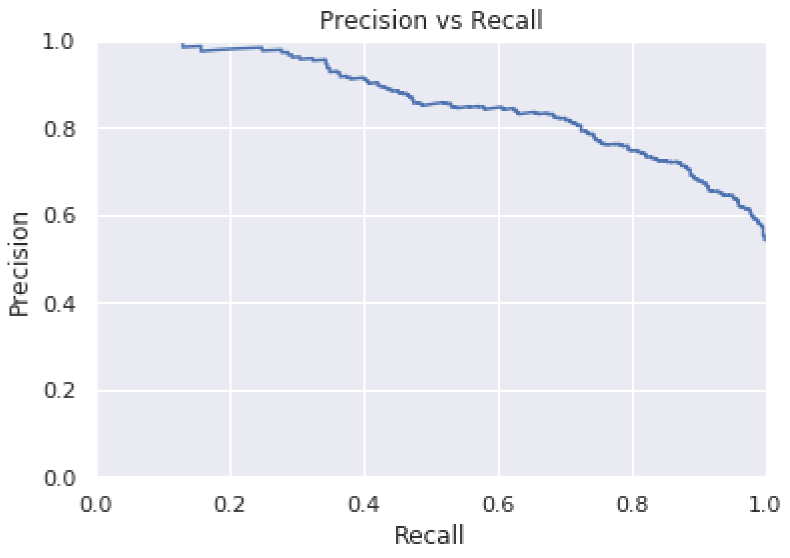
1. How does increasing p affect the Precision or Recall? What assumption do you have to make in order to answer this?
2. How does the curve vary depending on the quality of the estimator f? What if the estimator were perfect? What if it were guessing at random?
3. What if f were a reasonably good estimator but you used $1 - f$ as your estimator?
4. How do you decide how to make the tradeoff between precision and recall? How does this relate to the cost of a false positive versus false negative?

In [4]:
```
import numpy as np
from numpy import random
import matplotlib.pylab as plt
from ipywidgets import interact

random.seed(42)
x = np.r_[0:1:1000j]
y = random.binomial(1, x)
```

In [5]:
```
def plot_threshold(threshold=0.5):
    true_pos = (x > threshold) * (y > 0)
    plt.plot(x[true_pos], y[true_pos], '.', label="True Positive")
    false_pos = (x > threshold) * (y == 0)
    plt.plot(x[false_pos], y[false_pos], '.', label="False Positive")
    true_neg = (x <= threshold) * (y == 0)
    plt.plot(x[true_neg], y[true_neg], '.', label="True Negative")
    false_neg = (x <= threshold) * (y > 0)
    plt.plot(x[false_neg], y[false_neg], '.', label="False Negative")
    plt.axvline(threshold, c='k')
    plt.ylim(-0.5, 1.5)
    plt.legend()

    try:
        precision = 1.0 * sum(true_pos) / (sum(true_pos) + sum(false_pos))
    except ZeroDivisionError:
        precision = 1
    recall = 1.0 * sum(true_pos) / (sum(true_pos) + sum(false_neg))
    plt.title('Precision: %0.2f, Recall: %0.2f' % (precision, recall))

interact(plot_threshold, threshold=(0, 1, 0.1));
```

In [6]:
```
# compute the relevant stats
precisions, recalls, thresholds = metrics.precision_recall_curve(y, x)
thresholds = np.hstack([[0.], thresholds])  # n precisions but n-1 thresholds
f1s = 2 * (precisions * recalls) / (precisions + recalls)

fig = plt.figure()
plt.plot(thresholds, precisions, label='Precision')
plt.plot(thresholds, recalls, label='Recall')
plt.plot(thresholds, f1s, label='F1 Score')
plt.legend(loc='lower left')
plt.xlabel("Threshold"); plt.ylabel(" ")
plt.title("Precision, Recall, and F1 Score vs Thresholds")

fig = plt.figure()
plt.plot(recalls, precisions)
#plt.fill_between(recalls, precisions, alpha=0.2)  # AUC value
plt.xlim([0., 1.]); plt.ylim([0., 1.])
plt.xlabel("Recall"); plt.ylabel("Precision")
plt.title("Precision vs Recall");
```





Given that we have a precision and recall tradeoff for probabilistic estimators, we usually report statistics like "Precision at .6", which means "Precision when the threshold is set at $p = .6$" and vice versa for "Recall at .8".

Questions (Part 2):

1. Would you want high precision or high recall process for email spam detection?
2. What about drug approvals?
3. Let's suppose the NSA has an estimator for "likely to be a terrorist" which they use to determine who should be surveilled. How do enhanced national security versus fourth-amendment protections map onto precision and recall?

Single-valued Probabilistic Metrics

While a modeler can decide the appropriate threshold once given a precision-recall curve, it is hard to optimize for and it doesn't necessarily make sense to optimize for "Precision at .6" (why not "Precision at .7"?). We need a single-valued metric that is independent of threshold. Fortunately, there are several common ones:

1. The **Area Under the Curve** or **AUC** computes the area under the Precision Recall curve.
2. There is a **Receiver Operating Characteristic**, which is similar to the Precision-Recall curve. The area under this curve is itself a metric called **ROC-AUC**. The definition isn't hard, but it's beyond the scope of this course. You can find out more on Wikipedia (https://en.wikipedia.org/wiki/Receiver_operating_characteristic) or develop some geometric intuition from this blog post (https://shapeofdata.wordpress.com/2015/01/05/precision-recall-aucs-and-rocs/).
3. The **Log-Loss** or **Cross Entropy** is another characteristic. It is related to the notion of Entropy in Thermodynamics and Shannon-Entropy.

Entropy

Probabilistic models are inherently uncertain. In other words, they give us partial information. If a model tells us that $P(y_j = 1) = 0.9$, then we may strongly suspect that $y_j = 1$, but we would have strictly more information if we could measure or observe the label value y_j for sure.

Entropy and cross entropy are both ways to measure the information difference between predicted probabilities and true labels. If we assume that the true and predicted labels have the same distribution, then we can compute entropy as:

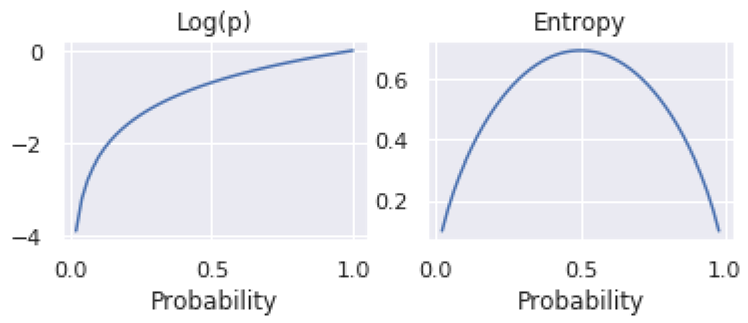
$$-\sum_j \left[p_j \log(p_j) + (1 - p_j)\log(1 - p_j)\right]$$

where p_j is the predicted probability that observation j is in the positive class (in a binary classification setup). This is essentially a penalty for uncertainty.

The value of each summand will lie somewhere on the curve below. Notice how the curve is symmetric about its maximum, $p = 0.5$. Entropy is highest when the probability of correctly identifying a class is a toss-up ($p = 0.5$) and lowest when the probability of correctly identifying a class is high ($p \approx 1.0$) or incorrectly identifying a class is low ($p \approx 0.0$).

In [7]:
```
# Plot entropy
p = np.linspace(0,1)
y_log = np.log(p)
y_entropy = (-1)*(p*np.log(p) + (1-p)*np.log(1-p))

fig = plt.figure()
plt.subplot(2,2,1)
plt.plot(p,y_log)
plt.title('Log(p)')
plt.xlabel("Probability"); plt.ylabel(" ")

plt.subplot(2,2,2)
plt.plot(p,y_entropy)
plt.title('Entropy')
plt.xlabel("Probability"); plt.ylabel(" ");
```

```
/opt/conda/envs/data3/lib/python3.6/site-packages/ipykernel_launcher.py:3: Ru
ntimeWarning: divide by zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports u
ntil
/opt/conda/envs/data3/lib/python3.6/site-packages/ipykernel_launcher.py:4: Ru
ntimeWarning: divide by zero encountered in log
  after removing the cwd from sys.path.
/opt/conda/envs/data3/lib/python3.6/site-packages/ipykernel_launcher.py:4: Ru
ntimeWarning: invalid value encountered in multiply
  after removing the cwd from sys.path.
```



Cross Entropy

When the probability distributions of the true and predicted classes are not the same, we use cross entropy. In this case, the true probabilities are either 0 or 1 while the predicted probabilities can be values on the interval $(0, 1)$. For a binary class problem where y_j is either 0 or 1, the cross entropy or log loss is given by:

$$-\sum_j \left[y_j \log(p_j) + (1 - y_j) \log(1 - p_j)\right]$$

Now we are simultaneously penalizing uncertainty and incorrect predictions. Note that incorrect predictions that are made with high confidence contribute the largest penalties to the sum.

Cross entropy is a good metric to use for training probabilistic models, but it can take a bit of effort to interpret. (Is an average cross entropy of 0.8 good or bad?) For this reason, cross entropy is often paired with a second metric, which is used for model evaluation. For example, when training neural network classifiers, it is common to report cross entropy and accuracy after each batch of training steps.

Questions:

1. Can you generalize the entropy formula from a two-class metric to an m-class metric? What about for the other metrics?
2. In Scikit Learn, `metrics.auc` (computed from precision and recall numbers) and `metrics.average_precision_score` (computed from observations and predictions) returns the same score. Why is average precision the same as AUC?
3. Which of these metrics do you want to increase, and which do you want to decrease?

Logistic Regression

Logistic regression is a classification algorithm that builds a probabilistic model by rescaling the output of a linear function. In the basic setup, we are trying to solve a binary classification problem where all of the label values y_j are equal to 0 or 1. Our prediction is

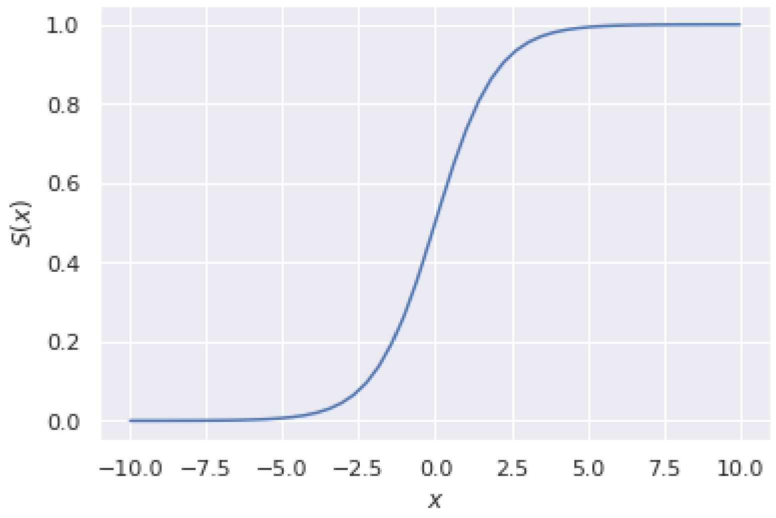
$$P(y_j = 1) = f(X_j) = S\left(X_j\beta + \beta_0\right) = S\left(\sum_i X_{ji}\beta_i + \beta_0\right)$$

where β is a column vector of coefficients and β_0 is an intercept term as in linear regression, and S denotes the sigmoid function

$$S(X) = \frac{1}{1 + e^{-x}}$$

which maps $\mathbb{R}$ to $(0, 1)$

In [8]:
```
lx = np.linspace(-10, 10)
ly = np.exp(lx) / (1 + np.exp(lx))
plt.plot(lx, ly)
plt.xlabel(r'$x$')
plt.ylabel(r'$S(x)$');
```



The cost function minimized in training is

$$C(\beta) = -\sum_j \left[y_j \log(f(X_j)) + (1 - y_j) \log(1 - f(X_j))\right] \, .$$

This is just the cross entropy function from before. C can be minimized using gradient descent together with the equations

$$\frac{\partial C}{\partial \beta_i} = \sum_j (f(X_j) - y_j) X_{ji} \, .$$

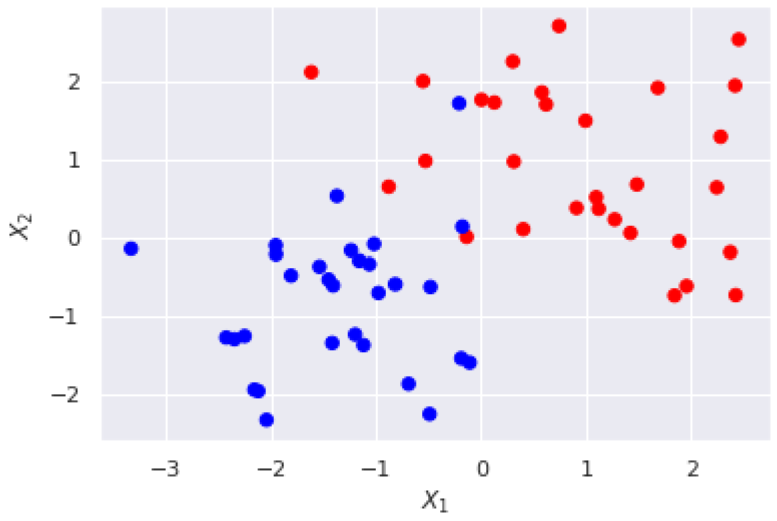
For those interested, the derivation is worked out here (https://stats.stackexchange.com/questions/278771/how-is-the-cost-function-from-logistic-regression-derivated).

Questions:

1. There's a `weight` parameter to Scikit Learn's `LogisticRegression` that lets you reweight different training examples. When might you want to do this?
2. Enumerate the similarities and differences between linear regression and logistic regression.

Let's look at a simple example of Scikit-Learn's `LogisticRegression` (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) classifier using artificial data. We will predict the class of points based on their coordinates in two dimensional space.

In [9]:
```
np.random.seed(30)

X0 = np.hstack((np.random.normal(1, 0.9, (2, 30)), np.random.normal(-1, 0.9, (
2, 30))))
y = np.hstack((np.ones((30)), np.zeros((30))))

plt.scatter(*X0, c=y, s=40, cmap=plt.cm.bwr)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```

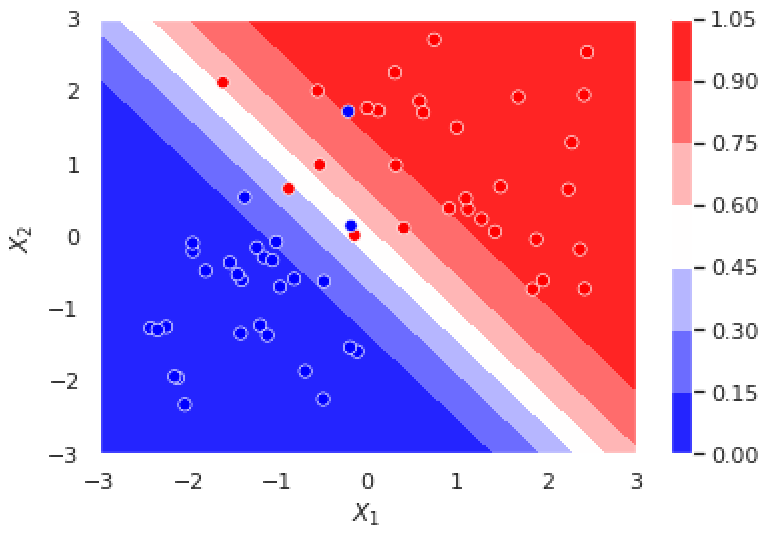


In [10]:
```
from sklearn.linear_model import LogisticRegression

X = X0.T

logistic_classifier = LogisticRegression()

logistic_classifier.fit(X,y)

print("Training Accuracy:", logistic_classifier.score(X,y))
```

```
Training Accuracy: 0.9333333333333333
```

In addition to predicting labels, we can get the raw probabilities that the model predicts for each class using the `.predict_proba` method of `LogisticRegression`. Take a look at how the model assigns probabilities to different regions:

In [11]:
```
x_min = -3
x_max = 3

# create a mesh to plot for probability values
h = .005  # step size in the mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(x_min, x_max, h))

# predicted values
zz = logistic_classifier.predict_proba(np.c_[xx.ravel(), yy.ravel()]).T[1].res
hape(xx.shape)
plt.contourf(xx, yy, zz, cmap=plt.cm.bwr)
plt.colorbar()

plt.scatter(*X0, c=y, s=40, cmap=plt.cm.bwr, edgecolors='white', linewidth =
0.6)

plt.xlim([x_min,x_max])
plt.ylim([x_min,x_max])
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```



Regardless of what threshold we choose, Logistic Regression will always make predictions by comparing inputs to some linear boundary. This is not surprising since Logistic Regression is built using a linear function.

Multiclass classification problems

So far we have talked about Two-Class Classification in the context of Logistic Regression. But what if we have more than two classes? There are generally two strategies to "bootstrap" a binary classifier to a multi-class classifier:

1. **One-versus-All**: For each class $k = 1, \ldots, K$, build a binary classifier for all points with label $y = k$ versus $y \neq k$.
2. **All-versus-All**: For each class $k \neq k'$, construct a binary classifier to distinguish between class k and k'. There's also the notion of Error-Correcting Output Codes

Scikit (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) uses One-versus-All for Multi-class Logistic Regression. If $f_k(x)$ is the predictor for class k, the probability of class k is just the normalized predictions,

$$p_k = \frac{f_k(x)}{\sum_k f_k(x)}$$

Scikit provides a way to do other multiclass-from-binary-classifier strategies in Scikit-Documentation (http://scikit-learn.org/stable/modules/multiclass.html).

Spoilers

Answers

Logistic Regression

1. One answer is if one class is very common (e.g. 99% of the data), you can downsample it it and then use the weight to give an unbiased estimate. A classic example is click-prediction in advertising, where clicks are rare compared to non-clicks.
2. Both multiply the features by a weight vector. Linear regression just uses those values; logistic regression maps into the range (0, 1).