What is Logistic Regression?

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, the logistic regression is a predictive analysis. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

Type of questions that a binary logistic regression can examine.

- How does the probability of getting lung cancer (yes vs. no) change for every additional pound a person is overweight and for every pack of cigarettes smoked per day?
- Do body weight, calorie intake, fat intake, and age have an influence on the probability of having a heart attack (yes vs. no)?

Binary logistic regression major assumptions:

1. The dependent variable should be dichotomous in nature (e.g., presence vs. absent).
2. There should be no outliers in the data, which can be assessed by converting the continuous predictors to standardized scores, and removing values below -3.29 or greater than 3.29.
3. There should be no high correlations (multicollinearity) among the predictors. This can be assessed by a correlation matrix among the predictors. Tabachnick and Fidell (2013) suggest that as long correlation coefficients among independent variables are less than 0.90 the assumption is met.

At the center of the logistic regression analysis is the task estimating the log odds of an event. Mathematically, logistic regression estimates a multiple linear regression function defined as:

logit(p) $= \log\left(\frac{p(y=1)}{1-(p=1)}\right) = \beta_0 + \beta_1 \cdot x_{i2} + \beta_2 \cdot x_{i2} + \_ + \beta_p \cdot x_{in}$

for i = 1…n .

Overfitting. When selecting the model for the logistic regression analysis, another important consideration is the model fit. Adding independent variables to a logistic regression model will always increase the amount of variance explained in the log odds (typically expressed as R^2). However, adding more and more variables to the model can result in overfitting, which reduces the generalizability of the model beyond the data on which the model is fit.

Reporting the R2. Numerous pseudo-R2 values have been developed for binary logistic regression. These should be interpreted with extreme caution as they have many computational issues which cause them to be artificially high or low. A better approach is to present any of the goodness of fit tests available; Hosmer-Lemeshow is a commonly used measure of goodness of fit based on the Chi-square test.

Introduction

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Unlike linear regression which outputs continuous number values, logistic regression transforms its output using the logistic sigmoid function to return a probability value which can then be mapped to two or more discrete classes.

Comparison to linear regression

Given data on time spent studying and exam scores. Linear Regression and logistic regression can predict different things:

- **Linear Regression** could help us predict the student's test score on a scale of 0 - 100. Linear regression predictions are continuous (numbers in a range).
- **Logistic Regression** could help use predict whether the student passed or failed. Logistic regression predictions are discrete (only specific values or categories are allowed). We can also view probability scores underlying the model's classifications.

Types of logistic regression

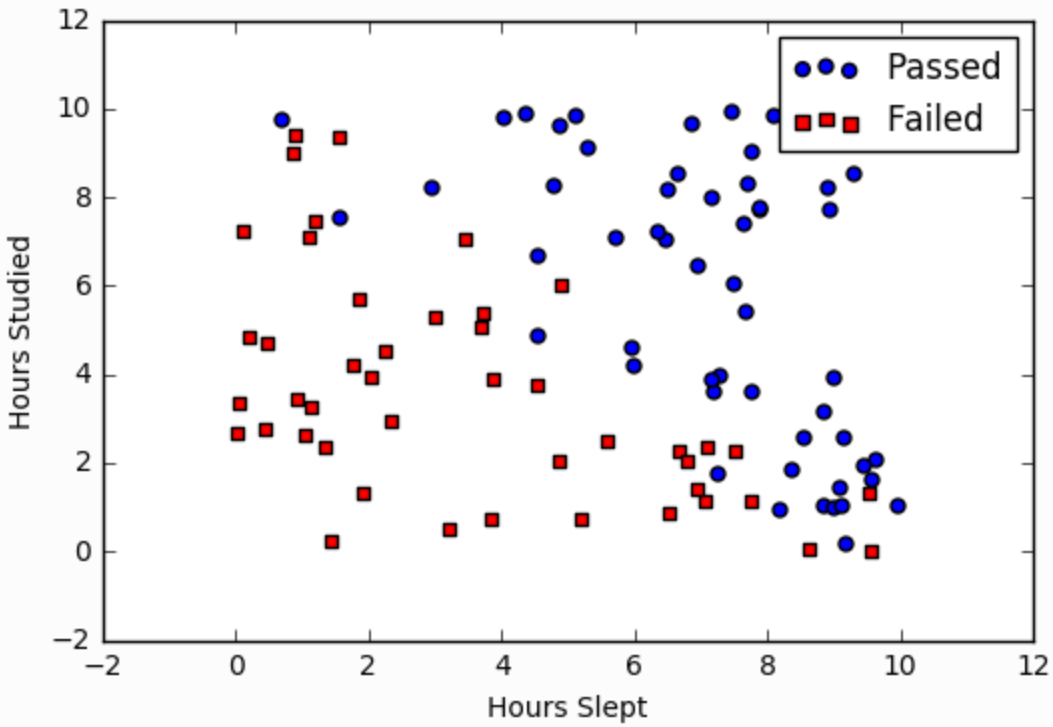
- Binary (Pass/Fail)
- Multi (Cats, Dogs, Sheep)
- Ordinal (Low, Medium, High)

Binary logistic regression

Say we're given data on student exam results and our goal is to predict whether a student will pass or fail based on number of hours slept and hours spent studying. We have two features (hours slept, hours studied) and two classes: passed (1) and failed (0).

Studied	Slept	Passed
4.85	9.63	1
8.62	3.23	0
5.43	8.23	1
9.21	6.34	0

Graphically we could represent our data with a scatter plot.



Sigmoid activation

In order to map predicted values to probabilities, we use the sigmoid function. The function maps any real value into another value between 0 and 1. In machine learning, we use sigmoid to map predictions to probabilities.

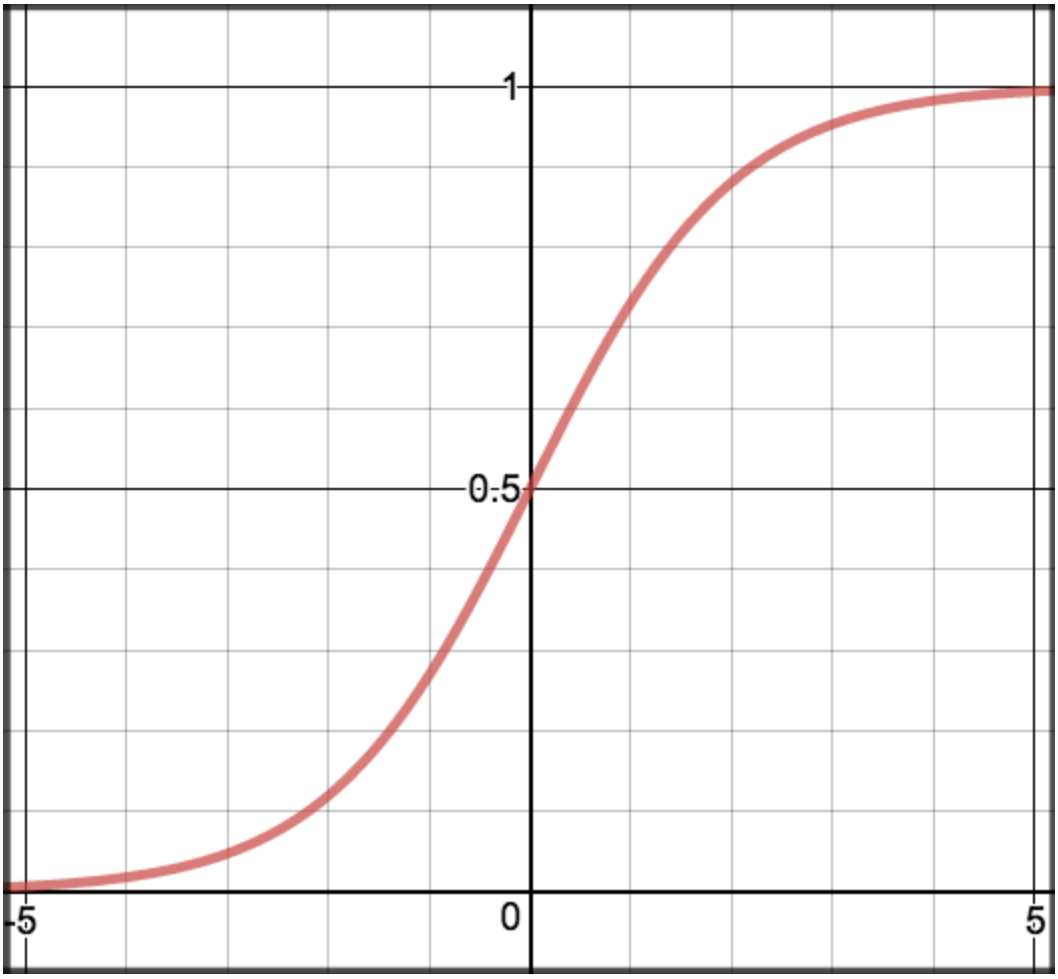
Math

$$S(z)=11+e−zS(z)=11+e−z$$

Note

- $S(z)$s(z) = output between 0 and 1 (probability estimate)
- zz = input to the function (your algorithm's prediction e.g. mx + b)
- ee = base of natural log

Graph



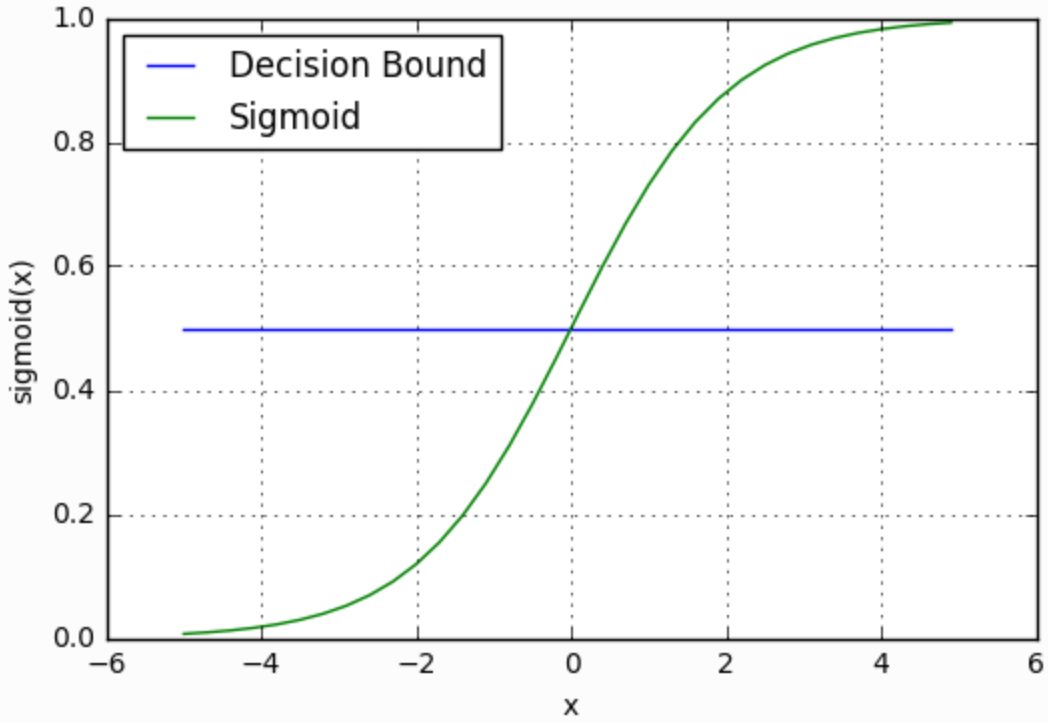
Code

Decision boundary

Our current prediction function returns a probability score between 0 and 1. In order to map this to a discrete class (true/false, cat/dog), we select a threshold value or tipping point above which we will classify values into class 1 and below which we classify values into class 2.

$$p \geq 0.5, class = 1 \quad p < 0.5, class = 0$$

For example, if our threshold was .5 and our prediction function returned .7, we would classify this observation as positive. If our prediction was .2 we would classify the observation as negative. For logistic regression with multiple classes we could select the class with the highest predicted probability.



Making predictions

Using our knowledge of sigmoid functions and decision boundaries, we can now write a prediction function. A prediction function in logistic regression returns the probability of our observation being positive, True, or "Yes". We call this class 1 and its notation is $P(class=1)$P(class=1). As the probability gets closer to 1, our model is more confident that the observation is in class 1.

Math

Let's use the same multiple linear regression equation from our linear regression tutorial.

$$z=W_0+W_1Studied+W_2Slept$$z=W0+W1Studied+W2Slept

This time however we will transform the output using the sigmoid function to return a probability value between 0 and 1.

$$P(class=1)=\frac{1}{1+e^{-z}}$$P(class=1)=11+e−z

If the model returns .4 it believes there is only a 40% chance of passing. If our decision boundary was .5, we would categorize this observation as "Fail.""

Code

We wrap the sigmoid function over the same prediction function we used in multiple linear regression

```
def predict(features, weights):
  '''
  Returns 1D array of probabilities
  that the class label == 1
```

```
    '''
    z = np.dot(features, weights)
    return sigmoid(z)
```

Cost function

Unfortunately we can't (or at least shouldn't) use the same cost function MSE (L2) as we did for linear regression. Why? There is a great math explanation in chapter 3 of Michael Neilson's deep learning book [5], but for now I'll simply say it's because our prediction function is non-linear (due to sigmoid transform). Squaring this prediction as we do in MSE results in a non-convex function with many local minimums. If our cost function has many local minimums, gradient descent may not find the optimal global minimum.

Math

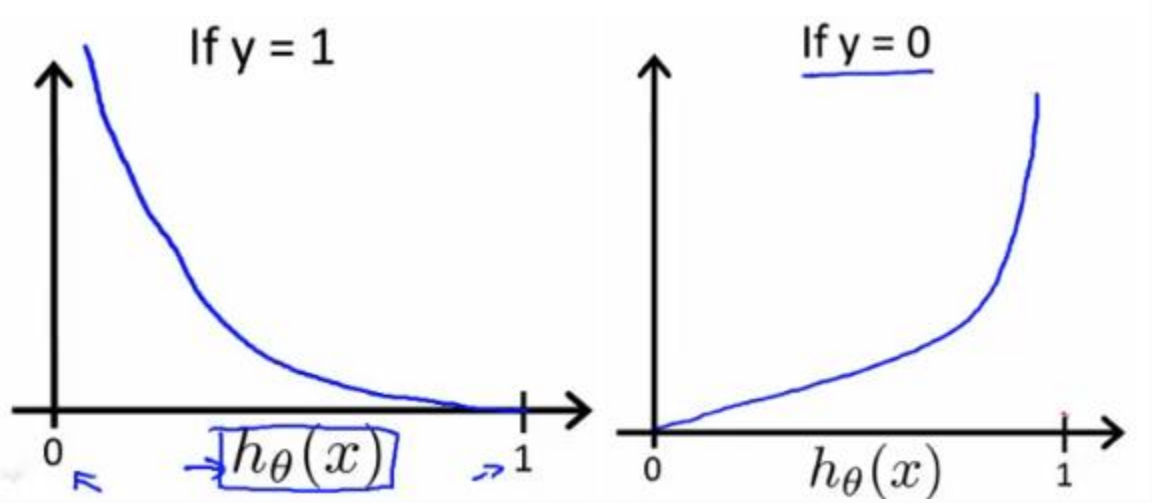
Instead of Mean Squared Error, we use a cost function called Cross-Entropy, also known as Log Loss. Cross-entropy loss can be divided into two separate cost functions: one for $y=1$ and one for $y=0$.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_\theta(x), y) = -\log(h_\theta(x)) \qquad \text{if } y = 1$$

$$\text{Cost}(h_\theta(x), y) = -\log(1 - h_\theta(x)) \qquad \text{if } y = 0$$

The benefits of taking the logarithm reveal themselves when you look at the cost function graphs for y=1 and y=0. These smooth monotonic functions [7] (always increasing or always decreasing) make it easy to calculate the gradient and minimize cost. Image from Andrew Ng's slides on logistic regression [1].



The key thing to note is the cost function penalizes confident and wrong predictions more than it rewards confident and right predictions! The corollary is increasing prediction accuracy (closer to 0 or 1) has diminishing returns on reducing cost due to the logistic nature of our cost function.

Above functions compressed into one

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

Multiplying by yy and $(1-y)$(1−y) in the above equation is a sneaky trick that let's us use the same equation to solve for both y=1 and y=0 cases. If y=0, the first side cancels out. If y=1, the second side cancels out. In both cases we only perform the operation we need to perform.

Vectorized cost function

$$h = g(X\theta)$$
$$J(\theta) = \frac{1}{m}\cdot\left(-y^T\log(h) - (1-y)^T\log(1-h)\right)$$

Code

```
def cost_function(features, labels, weights):
    '''
    Using Mean Absolute Error

    Features:(100,3)
    Labels: (100,1)
    Weights:(3,1)
    Returns 1D matrix of predictions
    Cost = ( log(predictions) + (1-labels)*log(1-predictions) ) / len(labels)
    '''
    observations = len(labels)

    predictions = predict(features, weights)

    #Take the error when label=1
    class1_cost = -labels*np.log(predictions)

    #Take the error when label=0
    class2_cost = (1-labels)*np.log(1-predictions)

    #Take the sum of both costs
    cost = class1_cost - class2_cost

    #Take the average cost
    cost = cost.sum()/observations

    return cost
```

Gradient descent

To minimize our cost, we use Gradient Descent just like before in Linear Regression. There are other more sophisticated optimization algorithms out there such as conjugate gradient like BFGS, but you don't have to worry about these. Machine learning libraries like Scikit-learn hide their implementations so you can focus on more interesting things!

Math

One of the neat properties of the sigmoid function is its derivative is easy to calculate. If you're curious, there is a good walk-through derivation on stack overflow [6]. Michael Neilson also covers the topic in chapter 3 of his book.

$$s'(z)=s(z)(1-s(z))s'(z)=s(z)(1-s(z))$$

Which leads to an equally beautiful and convenient cost function derivative:

$$C'=x(s(z)-y)C'=x(s(z)-y)$$

Note

- $C'C'$ is the derivative of cost with respect to weights
- yy is the actual class label (0 or 1)
- $s(z)s(z)$ is your model's prediction
- xx is your feature or feature vector.

Notice how this gradient is the same as the MSE (L2) gradient, the only difference is the hypothesis function.

Pseudocode

```
Repeat {

  1. Calculate gradient average
  2. Multiply by learning rate
  3. Subtract from weights

}
```

Code

```
def update_weights(features, labels, weights, lr):
    '''
    Vectorized Gradient Descent

    Features:(200, 3)
    Labels: (200, 1)
    Weights:(3, 1)
    '''
    N = len(features)

    #1 - Get Predictions
    predictions = predict(features, weights)

    #2 Transpose features from (200, 3) to (3, 200)
    # So we can multiply w the (200,1)  cost matrix.
    # Returns a (3,1) matrix holding 3 partial derivatives --
    # one for each feature -- representing the aggregate
    # slope of the cost function across all observations
    gradient = np.dot(features.T,  predictions - labels)

    #3 Take the average cost derivative for each feature
    gradient /= N

    #4 - Multiply the gradient by our learning rate
    gradient *= lr
```

```
    #5 - Subtract from our weights to minimize cost
    weights -= gradient

    return weights
```

Mapping probabilities to classes

The final step is assign class labels (0 or 1) to our predicted probabilities.

Decision boundary

```
def decision_boundary(prob):
  return 1 if prob >= .5 else 0
```

Convert probabilities to classes

```
def classify(preds):
  '''
  input  - N element array of predictions between 0 and 1
  output - N element array of 0s (False) and 1s (True)
  '''
  decision_boundary = np.vectorize(decision_boundary)
  return decision_boundary(predictions).flatten()
```

Example output

```
Probabilities = [ 0.967, 0.448, 0.015, 0.780, 0.978, 0.004]
Classifications = [1, 0, 0, 1, 1, 0]
```

Training

Our training code is the same as we used for linear regression.

```
def train(features, labels, weights, lr, iters):
    cost_history = []

    for i in range(iters):
        weights = update_weights(features, labels, weights, lr)

        #Calculate error for auditing purposes
        cost = cost_function(features, labels, weights)
        cost_history.append(cost)

        # Log Progress
        if i % 1000 == 0:
            print "iter: "+str(i) + " cost: "+str(cost)

    return weights, cost_history
```

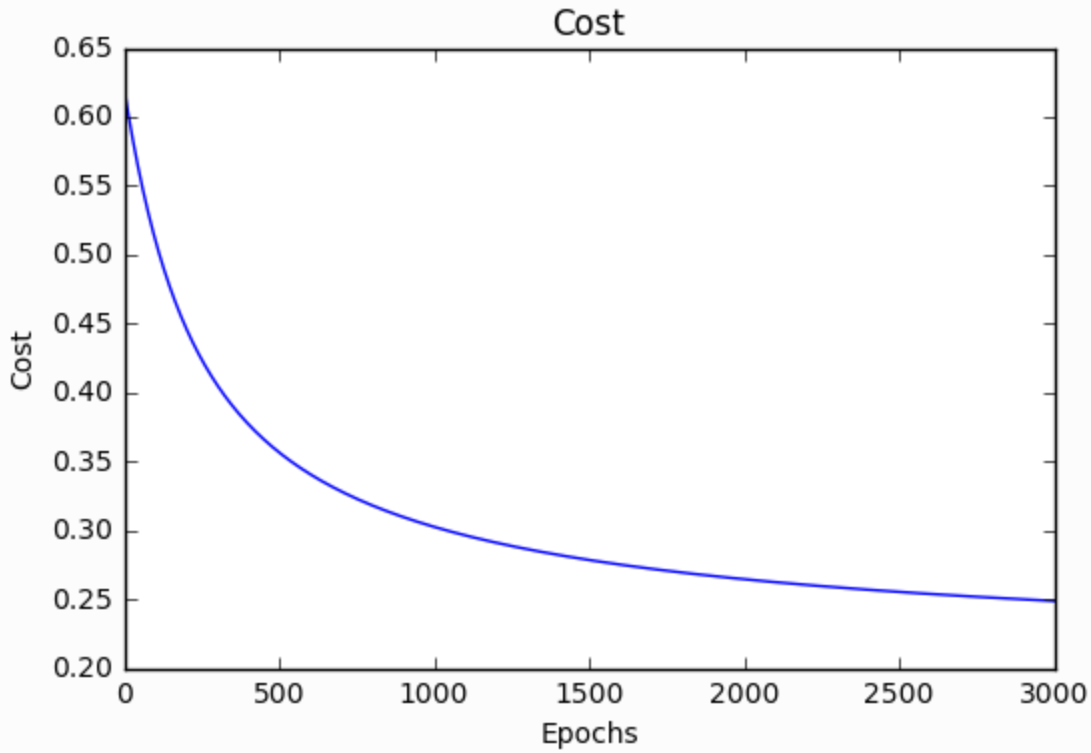
Model evaluation

If our model is working, we should see our cost decrease after every iteration.

```
iter: 0 cost: 0.635
iter: 1000 cost: 0.302
iter: 2000 cost: 0.264
```

Final cost: 0.2487. **Final weights:** [-8.197, .921, .738]

Cost history



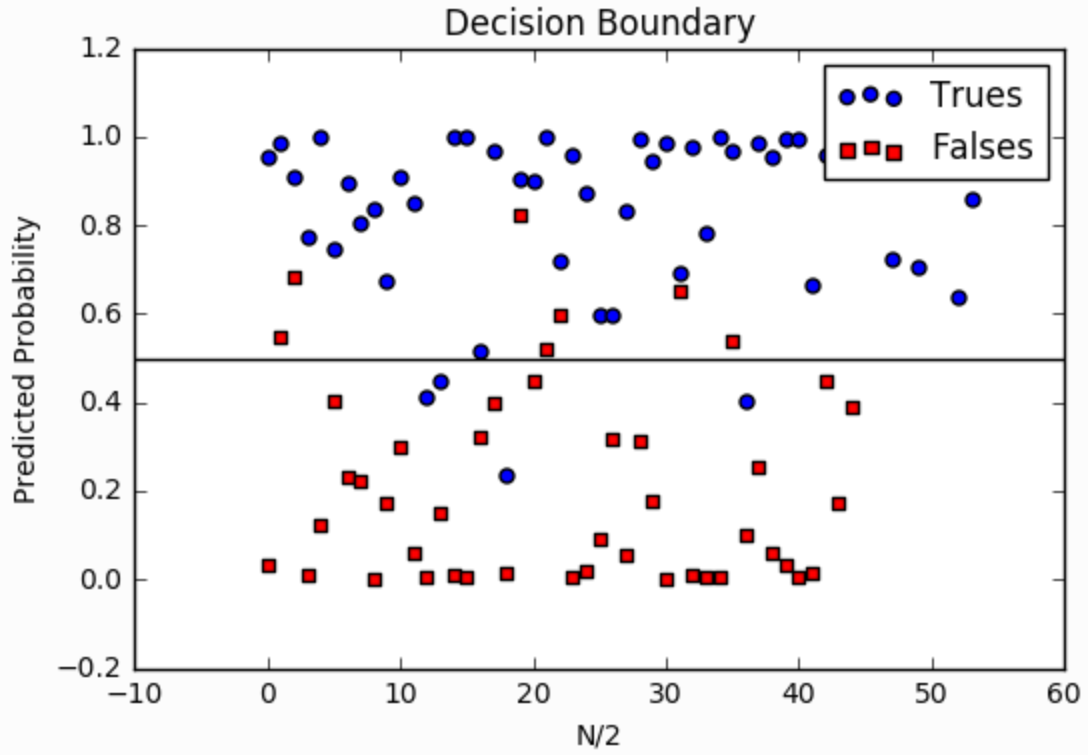
Accuracy

Accuracy measures how correct our predictions were. In this case we simple compare predicted labels to true labels and divide by the total.

```
def accuracy(predicted_labels, actual_labels):
    diff = predicted_labels - actual_labels
    return 1.0 - (float(np.count_nonzero(diff)) / len(diff))
```

Decision boundary

Another helpful technique is to plot the decision boundary on top of our predictions to see how our labels compare to the actual labels. This involves plotting our predicted probabilities and coloring them with their true labels.



Code to plot the decision boundary

```
def plot_decision_boundary(trues, falses):
    fig = plt.figure()
    ax = fig.add_subplot(111)

    no_of_preds = len(trues) + len(falses)

    ax.scatter([i for i in range(len(trues))], trues, s=25, c='b', marker="o", label='Trues')
    ax.scatter([i for i in range(len(falses))], falses, s=25, c='r', marker="s", label='Falses')

    plt.legend(loc='upper right');
    ax.set_title("Decision Boundary")
    ax.set_xlabel('N/2')
    ax.set_ylabel('Predicted Probability')
    plt.axhline(.5, color='black')
    plt.show()
```

Multiclass logistic regression

Instead of $y=0,1$ we will expand our definition so that $y=0,1...n$. Basically we re-run binary classification multiple times, once for each class.

Procedure

1. Divide the problem into n+1 binary classification problems (+1 because the index starts at 0?).
2. For each class…
3. Predict the probability the observations are in that single class.
4. prediction = <math>max(probability of the classes)

For each sub-problem, we select one class (YES) and lump all the others into a second class (NO). Then we take the class with the highest predicted value.

Softmax activation

something about softmax here…

Scikit-Learn example

Let's compare our performance to the `LogisticRegression` model provided by scikit-learn [8].

```
import sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import train_test_split

# Normalize grades to values between 0 and 1 for more efficient computation
normalized_range = sklearn.preprocessing.MinMaxScaler(feature_range=(-1,1))

# Extract Features + Labels
labels.shape =  (100,) #scikit expects this
features = normalized_range.fit_transform(features)

# Create Test/Train
features_train,features_test,labels_train,labels_test = train_test_split(features,labels,test_size=0.4)

# Scikit Logistic Regression
scikit_log_reg = LogisticRegression()
scikit_log_reg.fit(features_train,labels_train)

#Score is Mean Accuracy
scikit_score = clf.score(features_test,labels_test)
print 'Scikit score: ', scikit_score

#Our Mean Accuracy
observations, features, labels, weights = run()
probabilities = predict(features, weights).flatten()
classifications = classifier(probabilities)
our_acc = accuracy(classifications,labels.flatten())
print 'Our score: ',our_acc
```

Scikit score: 0.88. **Our score:** 0.89