Logistic Regression

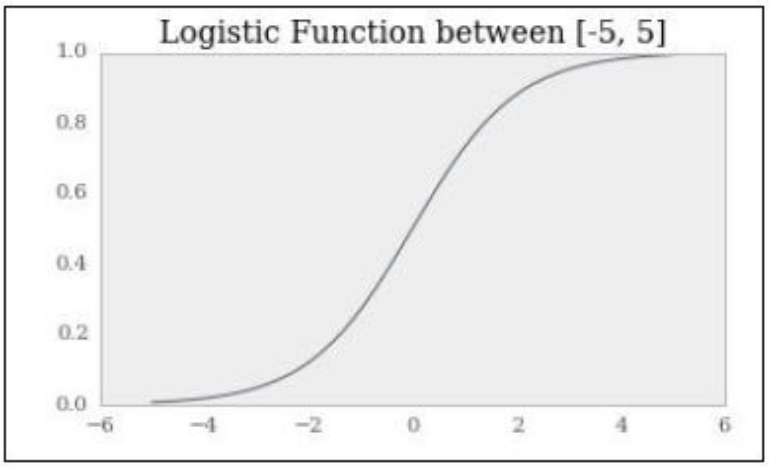
Linear models can actually be used for classification tasks. This involves fitting a linear model to the probability of a certain class, and then using a function to create a threshold at which we specify the outcome of one of the classes.

The Logistic Function

The function used here is typically the logistic function. The logistic function $f(t)$ is defined as follows:

$$f(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}},$$

Visually, it looks like the following:



Here the range of values is $[-5, 5]$, the transition region. In practice most values of t will be outside this region.

$$f(8) = \frac{1}{1 + e^{-8}} = \frac{1}{1 + 0.00034} = 0.99966 \approx 1$$

Worked Example : Create some Data

Let's use the `make_classification` method, create a dataset. We will have 1000 cases with 4 features. This data is contained in X. The targets are 0s and 1s , contained in Y.

```
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4)
```

Implementation

The `LogisticRegression` object works in the same way as the other linear models:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression()
```

Here we have an "empty" model called `lr`, ready to be trained.

Splitting the Data Set

- We will use the last 200 samples to test the trained model on.
- Since this is a random dataset, it's fine to hold out the last 200.
- If you're dealing with structured data, don't do this (*for example, if you deal with time series data*):

```
>>> X_train = X[:-200]
>>> X_test = X[-200:]
>>> y_train = y[:-200]
>>> y_test = y[-200:]
```

- Now, we need to fit the model with logistic regression.
- We can use the trained model to make predictions on both the testing data set, and the training data set.
- Remark: Often, you'll be better on the train set; it's a matter of how much worse you are on the test set:

```
>>> #  Train the Model
>>> lr.fit(X_train, y_train)
>>>
>>> # Lets try it out
>>> y_test_pred = lr.predict(X_test)
>>> y_train_pred = lr.predict(X_train)
```

Model Evaluation

- Now that we have the predictions, let's take a look at how good our predictions were.
- Here, we'll simply look at the number of times we were correct.
- The calculation is simple; it's the number of times we were correct over the total sample:

```
>>> # Number of Cases
>>> y_train.shape[0]
1000

>>> (y_train_pred == y_train).sum().astype(float)/1000

0.8662499
>>> # 86.6 % success rate on training data
```

Similarly for the test sample:

```
>>> (y_test_pred == y_test).sum().astype(float) / 1000

0.900000
>>> # 90 % success rate on training data
```

Class Imbalance

- There will be a situation where one class is weighted differently from the other classes; for example, one class may be 99 percent of cases.
- This situation will occur regularly in real world data science.
- The canonical example is fraud detection, where most transactions aren't fraud, but the cost associated with misclassification is asymmetric between classes.

Let's create a classification problem with 95 percent imbalance and see how the basic stock logistic regression handles this case:

```
>>> X, y = make_classification(n_samples=5000,
n_features=4,
weights=[.95])
>>>
>>>  #to confirm the class imbalance
>>> sum(y) / (len(y)*1.)
0.0555
```

Create the train and test sets, and then fit logistic regression:

```
>>> X_train = X[:-500]
>>> X_test = X[-500:]
>>> y_train = y[:-500]
>>> y_test = y[-500:]
>>>
>>> lr.fit(X_train, y_train)
>>> y_train_pred = lr.predict(X_train)
>>> y_test_pred = lr.predict(X_test)
```

Now, to see how well our model fits the data, do the following:

```
>>> (y_train_pred == y_train).sum().astype(float) / y_train.shape[0]

0.96977
>>> (y_test_pred== y_test).sum().astype(float) / y_test.shape[0]

0.97999
```

- At first, it looks like we did well, but it turns out that when we always guessed that a transaction was not fraud (or class 0 in general) we were right around 95 percent of the time.
- If we look at how well we did in classifying the 1 class, it's not nearly as good:

```
>>> (y_test[y_test==1] == y_test_pred[y_test==1])
.sum().astype(float) / y_test[y_test==1].shape[0]

0.583333
```

- Hypothetically, we might care more about identifying fraud cases than non-fraud cases; this could be due to a business rule, so we might alter how we weigh the correct and incorrect values.
- By default, the classes are weighted (and thus resampled) in accordance with the inverse of the class weights of the training set. However, because we care more about fraud cases, let's oversample the fraud relative to nonfraud cases.
- We know that our relative weighting right now is 95 percent nonfraud; let's change this to overweight fraud cases:

```
>>> lr = LogisticRegression(class_weight={0:.15, 1:.85})
>>> lr.fit(X_train, y_train)
```

Let's predict the outputs again:

```
>>> y_train_pred = lr.predict(X_train)
>>> y_test_pred = lr.predict(X_test)
```

We can see that we did a much better job on classifying the fraud cases:

```
>>> (y_test[y_test==1] == y_test_pred[y_test==1]).sum().
astype(float) / y_test[y_test==1].shape[0]
0.875
```

At what expense do we do this? To find out, use the following command:

```
>>> (y_test_pred == y_test).sum().astype(float) / y_test.shape[0]
0.967999
```

- Here, there's only about 1 percent less accuracy. Whether thatis acceptable depends on your problem.
- Put in the context of the problem, if the estimated cost associated with fraud is sufficiently large, it can eclipse the cost associated with tracking fraud.

Skip This

- The question then changes to how to move on from the logistic function to a method by which we can classify groups.
- First, recall the linear regression hopes offending the linear function that fits the expected value of Y, given the values of X; this is $E(Y|X) = X\beta$. Here, the Y values are the probabilities of the classes.
- Therefore, the problem we're trying to solve is $E(p|X) = X\beta$. Then, once the threshold is applied, this becomes $\text{Logit}(p) = X\beta$.
- The idea expanded is how other forms of regression work, for example, Poisson.