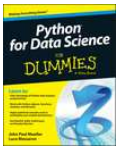
dummies
A Wiley Brand

USING LOGISTIC REGRESSION IN PYTHON FOR DATA SCIENCE



RELATED BOOK
Python for Data Science For Dummies

By **John Paul Mueller, Luca Massaron**

You can use logistic regression in Python for data science. Linear regression is well suited for estimating values, but it isn't the best tool for predicting the class of an observation. In spite of the statistical theory that advises against it, you can actually try to classify a binary class by scoring one class as 1 and the other as 0. The results are disappointing most of the time, so the statistical theory wasn't wrong!

The fact is that linear regression works on a continuum of numeric estimates. In order to classify correctly, you need a more suitable measure, such as the probability of class ownership. Thanks to the following formula, you can transform a linear regression numeric estimate into a probability that is more apt to describe how a class fits an observation:

probability of a class = exp(r) / (1+exp(r))

r is the regression result (the sum of the variables weighted by the coefficients) and exp is the exponential function. exp(r) corresponds to Euler's number e elevated to the power of r. A linear regression using such a formula (also called a link function) for transforming its results into probabilities is a logistic regression.

APPLYING LOGISTIC REGRESSION

Logistic regression is similar to linear regression, with the only difference being the y data, which should contain integer values indicating the class relative to the observation. Using the Iris dataset from the Scikit-learn datasets module, you can use the values 0, 1, and 2 to denote three classes that correspond to three species:

```
from sklearn.datasets import load_iris
iris = load_iris()
X, y = iris.data[:-1,:], iris.target[:-1]
```

To make the example easier to work with, leave a single value out so that later you can use this value to test the efficacy of the logistic regression model on it.

```
from sklearn.linear_model import LogisticRegression
logistic = LogisticRegression()
logistic.fit(X,y)
print 'Predicted class %s, real class %s' % (
 logistic.predict(iris.data[-1,:]),iris.target[-1])
print 'Probabilities for each class from 0 to 2: %s'
 % logistic.predict_proba(iris.data[-1,:])
Predicted class [2], real class 2
Probabilities for each class from 0 to 2:
 [[ 0.00168787 0.28720074 0.71111138]]
```

Contrary to linear regression, logistic regression doesn't just output the resulting class (in this case, the class 2), but it also estimates the probability of the observation's being part of all three classes. Based on the observation used for prediction, logistic regression estimates a probability of 71 percent of its being from class 2 — a high probability, but not a perfect score, therefore leaving a margin of uncertainty.

Using probabilities lets you guess the most probable class, but you can also order the predictions with respect to being part of that class. This is especially useful for medical purposes: Ranking a prediction in terms of likelihood with respect to others can reveal what patients are at most risk of getting or already having a disease.

CONSIDERING WHEN CLASSES ARE MORE

The previous problem, logistic regression, automatically handles a multiple class problem (it started with three iris species to guess). Most algorithms provided by Scikit-learn that predict probabilities or a score for class can automatically handle multiclass problems using two different strategies:

- **One versus rest:** The algorithm compares every class with all the remaining classes, building a model for every class. If you have ten classes to guess, you have ten models. This approach relies on the OneVsRestClassifier class from Scikit-learn.
- **One versus one:** The algorithm compares every class against every individual remaining class, building a number of models equivalent to n * (n-1) / 2, where n is the number of classes. If you have ten classes, you have 45 models. This approach relies on the OneVsOneClassifier class from Scikit-learn.

In the case of logistic regression, the default multiclass strategy is the one versus rest. This example shows how to use both the strategies with the handwritten digit dataset, containing a class for numbers from 0 to 9. The following code loads the data and places it into variables.

```
from sklearn.datasets import load_digits
digits = load_digits()
X, y = digits.data[:1700,:], digits.target[:1700]
tX, ty = digits.data[1700:,:], digits.target[1700:]
```

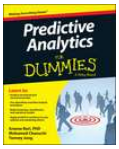
The observations are actually a grid of pixel values. The grid's dimensions are 8 pixels by 8 pixels. To make the data easier to learn by machine-learning algorithms, the code aligns them into a list of 64 elements. The example reserves a part of the available examples for a test.

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.multiclass import OneVsOneClassifier
OVR = OneVsRestClassifier(LogisticRegression()).fit(X,y)
OVO = OneVsOneClassifier(LogisticRegression()).fit(X,y)
print 'One vs rest accuracy: %.3f' % OVR.score(tX,ty)
print 'One vs one accuracy: %.3f' % OVO.score(tX,ty)
One vs rest accuracy: 0.938
One vs one accuracy: 0.969
```

The two multiclass classes OneVsRestClassifier and OneVsOneClassifier operate by incorporating the estimator (in this case, LogisticRegression). After incorporation, they usually work just like any other learning algorithm in Scikit-learn. Interestingly, the one-versus-one strategy obtained the best accuracy thanks to its high number of models in competition.

When working with Anaconda and Python version 3.4, you may receive a deprecation warning when working with this example. You're safe to ignore the deprecation warning — the example should work as normal. All the deprecation warning tells you is that one of the features used in the example is due for an update or will become unavailable in a future version of Python.

HOW TO CREATE A SUPERVISED LEARNING MODEL WITH LOGISTIC REGRESSION



RELATED BOOK

Predictive Analytics For Dummies

By **Anasse Bari, Mohamed Chaouchi, Tommy Jung**

After you build your first classification predictive model for analysis of the data, creating more models like it is a really straightforward task in scikit. The only real difference from one model to the next is that you may have to tune the parameters from algorithm to algorithm.

HOW TO LOAD YOUR DATA

This code listing will load the iris dataset into your session:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
```

HOW TO CREATE AN INSTANCE OF THE CLASSIFIER

The following two lines of code create an instance of the classifier. The first line imports the logistic regression library. The second line creates an instance of the logistic regression algorithm.

```
>>> from sklearn import linear_model
>>> logClassifier = linear_model.LogisticRegression(C=1,   random_state=111)
```

Notice the parameter (regularization parameter) in the constructor. The *regularization parameter* is used to prevent overfitting. The parameter isn't strictly necessary (the constructor will work fine without it because it will default to C=1). Creating a logistic regression classifier using C=150 creates a better plot of the decision surface. You can see both plots below.

HOW TO RUN THE TRAINING DATA

You'll need to split the dataset into training and test sets before you can create an instance of the logistic regression classifier. The following code will accomplish that task:

```
>>> from sklearn import cross_validation
>>> X_train, X_test, y_train, y_test =   cross_validation.train_test_split(iris.data,   iris.target, test_size=0.10, random_state=111)
>>> logClassifier.fit(X_train, y_train)
```

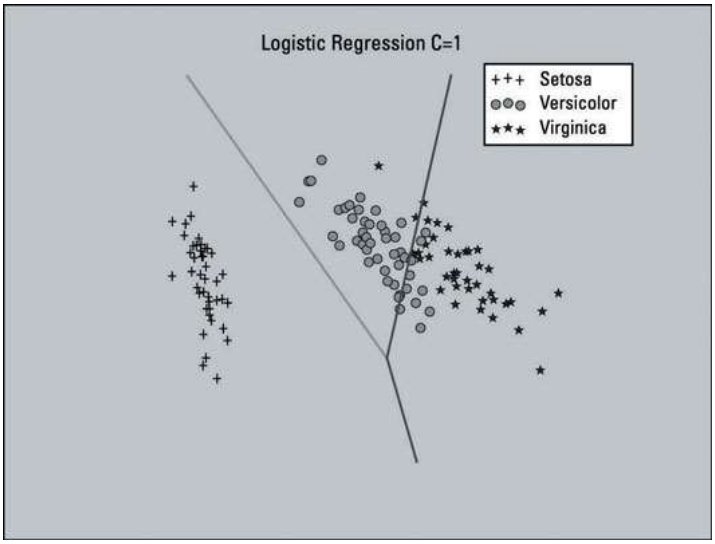
Line 1 imports the library that allows you to split the dataset into two parts.

Line 2 calls the function from the library that splits the dataset into two parts and assigns the now-divided datasets to two pairs of variables.

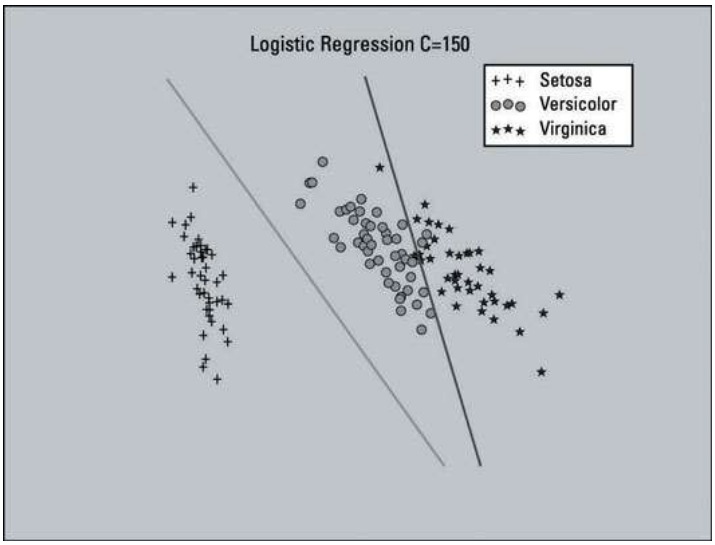
Line 3 takes the instance of the logistic regression classifier you just created and calls the fit method to train the model with the training dataset.

HOW TO VISUALIZE THE CLASSIFIER

Looking at the decision surface area on the plot, it looks like some tuning has to be done. If you look near the middle of the plot, you can see that many of the data points belonging to the middle area (Versicolor) are lying in the area to the right side (Virginica).



This image shows the decision surface with a C value of 150. It visually looks better, so choosing to use this setting for your logistic regression model seems appropriate.



HOW TO RUN THE TEST DATA

In the following code, the first line feeds the test dataset to the model and the third line displays the output:

```
>>> predicted = logClassifier.predict(X_test)
>>> predictedarray([0, 0, 2, 2, 1, 0, 0, 2, 2, 1, 2, 0, 2, 2, 2])
```

HOW TO EVALUATE THE MODEL

You can cross-reference the output from the prediction against the y_test array. As a result, you can see that it predicted all the test data points correctly. Here's the code:

```
>>> from sklearn import metrics
>>> predictedarray([0, 0, 2, 2, 1, 0, 0, 2, 2, 1, 2, 0, 2, 2, 2])
>>> y_testarray([0, 0, 2, 2, 1, 0, 0, 2, 2, 1, 2, 0, 2, 2, 2])
>>> metrics.accuracy_score(y_test, predicted)1.0   # 1.0 is 100 percent accuracy
>>> predicted == y_testarray([ True, True, True, True, True, True, True,  True, True, True, True, True, True, True,  True], dtype=bool)
```

So how does the logistic regression model with parameter C=150 compare to that? Well, you can't beat 100 percent. Here is the code to create and evaluate the logistic classifier with C=150:

```
>>> logClassifier_2 = linear_model.LogisticRegression(  C=150, random_state=111)
>>> logClassifier_2.fit(X_train, y_train)
>>> predicted = logClassifier_2.predict(X_test)
>>> metrics.accuracy_score(y_test, predicted)0.93333333333333335
>>> metrics.confusion_matrix(y_test, predicted)array([[5, 0, 0],  [0, 2, 0],  [0, 1, 7]])
```

We expected better, but it was actually worse. There was one error in the predictions. The result is the same as that of the Support Vector Machine (SVM) model.

Here is the full listing of the code to create and evaluate a logistic regression classification model with the default parameters:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import linear_model
>>> from sklearn import cross_validation
>>> from sklearn import metrics
>>> iris = load_iris()
>>> X_train, X_test, y_train, y_test =   cross_validation.train_test_split(iris.data,   iris.target, test_size=0.10, random_state=111)
>>> logClassifier = linear_model.LogisticRegression(,   random_state=111)
>>> logClassifier.fit(X_train, y_train)
>>> predicted = logClassifier.predict(X_test)
>>> predictedarray([0, 0, 2, 2, 1, 0, 0, 2, 2, 1, 2, 0, 2, 2, 2])
>>> y_testarray([0, 0, 2, 2, 1, 0, 0, 2, 2, 1, 2, 0, 2, 2, 2])
>>> metrics.accuracy_score(y_test, predicted)1.0   # 1.0 is 100 percent accuracy
>>> predicted == y_testarray([ True, True, True, True, True, True, True,  True, True, True, True, True, True, True,  True], dtype=bool)
```