ŷhat

PRODUCTS ABOUT BLOG RSS DOCS Follow @YhatHQ

« SQL FOR PANDAS DATAFRAMES YHAT IS GOING TO PYCON »

Logistic Regression in Python

by yhat

March 3, 2013

Subscribe Tweet 85

Logistic Regression is a statistical technique capable of predicting a binary outcome. It's a well-known strategy, widely used in disciplines ranging from credit and finance to medicine to criminology and other social sciences. Logistic regression is fairly intuitive and very effective; you're likely to find it among the first few chapters of a machine learning or applied statistics book and it's usage is covered by many stats courses.

It's not hard to find quality logistic regression examples using R. This tutorial, for example, published by UCLA, is a great resource and one that I've consulted many times. Python is one of the most popular languages for machine learning, and while there are bountiful resources covering topics like Support Vector Machines and text classification using Python, there's far less material on logistic regression.

This is a post about using logistic regression in Python.

Introduction

Introduction

We'll use a few libraries in the code samples. Make sure you have these installed before you run through the code on your machine.

- `numpy`: a language extension that defines the numerical array and matrix
- `pandas`: primary package to handle and operate directly on data.
- `statsmodels`: statistics & econometrics package with useful tools for parameter estimation & statistical testing
- `pylab`: for generating plots

Check out our post on Setting Up Scientific Python if you're missing one or more of these.

Example Use Case for Logistic Regression

We'll be using the same dataset as UCLA's Logit Regression in R tutorial to explore logistic regression in Python. Our goal will be to identify the various factors that may influence admission into graduate school.

The dataset contains several columns which we can use as predictor variables:

- `gpa`
- `gre` score
- `rank` or presitge of an applicant's undergraduate alma mater

The fourth column, `admit`, is our binary target variable. It indicates whether or not a candidate was admitted our not.

Load the data

Load the data using `pandas.read_csv`. We now have a `DataFrame` and can explore the data.

```
import pandas as pd
import statsmodels.api as sm
import pylab as pl
import numpy as np
```

```
# read the data in
df = pd.read_csv("http://www.ats.ucla.edu/stat/data/binary.csv")

# take a look at the dataset
print df.head()
#    admit  gre   gpa  rank
# 0      0  380  3.61     3
# 1      1  660  3.67     3
# 2      1  800  4.00     1
# 3      1  640  3.19     4
# 4      0  520  2.93     4

# rename the 'rank' column because there is also a DataFrame method calle
df.columns = ["admit", "gre", "gpa", "prestige"]
print df.columns
# array([admit, gre, gpa, prestige], dtype=object)
```

logistic_load_data.py hosted with ❤ by **GitHub** **view raw**

Notice that one of the columns is called "`rank`." This presents a problem since `rank` is also the name of a method belonging to pandas `DataFrame` (`rank` calculates the ordered rank (1 through n) of a `DataFrame` / `Series`). To make things easier, I renamed the rank column to "prestige".

Summary Statistics & Looking at the data

Now that we've got everything loaded into Python and named appropriately let's take a look at the data. We can use the `pandas` function `describe` to give us a summarized view of everything-- `describe` is analogous to `summary` in R. There's also function for calculating the standard deviation, `std`. I've included it here to be consistent UCLA's tutorial, but the standard deviation is also included in `describe`.

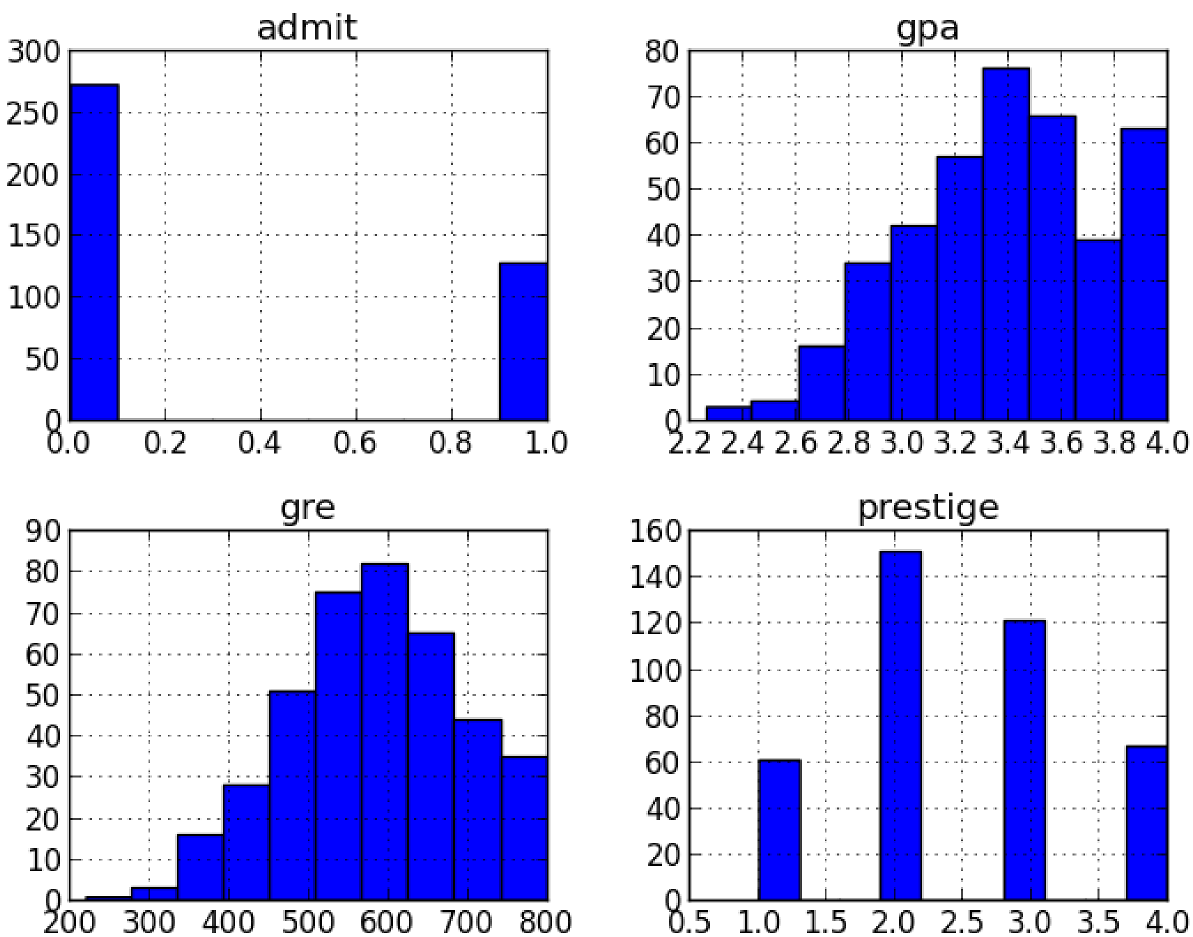
A feature I really like in `pandas` is the `pivot_table/crosstab` aggregations.

`crosstab` makes it really easy to do multidimensional frequency tables (sort of like `table` in R). You might want to play around with this to look at different cuts of the data.

```
# summarize the data
print df.describe()
#              admit         gre         gpa   prestige
# count  400.000000  400.000000  400.000000  400.00000
# mean     0.317500  587.700000    3.389900    2.48500
# std      0.466087  115.516536    0.380567    0.94446
# min      0.000000  220.000000    2.260000    1.00000
# 25%      0.000000  520.000000    3.130000    2.00000
# 50%      0.000000  580.000000    3.395000    2.00000
# 75%      1.000000  660.000000    3.670000    3.00000
# max      1.000000  800.000000    4.000000    4.00000

# take a look at the standard deviation of each column
print df.std()
# admit      0.466087
# gre      115.516536
# gpa        0.380567
# prestige   0.944460

# frequency table cutting presitge and whether or not someone was admitte
print pd.crosstab(df['admit'], df['prestige'], rownames=['admit'])
# prestige   1   2   3   4
# admit
# 0         28  97  93  55
# 1         33  54  28  12

# plot all of the columns
df.hist()
pl.show()
```

logistic_looking_at_the_data.py hosted with ❤ by **GitHub** **view raw**

Histograms are often one of the most helpful tools you can use during the exploratory phase of any data analysis project. They're normally pretty easy to plot, quick to interpret, and they give you a nice visual representation of your problem.



dummy variables

`pandas` gives you a great deal of control over how categorical variables are represented. We're going dummify the "prestige" column using `get_dummies`.

`get_dummies` creates a new `DataFrame` with binary indicator variables for each category/option in the column specified. In this case, `prestige` has four levels: 1, 2, 3 and 4 (1 being most prestigious). When we call `get_dummies`, we get a dataframe with four columns, each of which describes one of those levels.

```
# dummify rank
dummy_ranks = pd.get_dummies(df['prestige'], prefix='prestige')
print dummy_ranks.head()
#    prestige_1  prestige_2  prestige_3  prestige_4
# 0           0           0           1           0
# 1           0           0           1           0
# 2           1           0           0           0
# 3           0           0           0           1
# 4           0           0           0           1

# create a clean data frame for the regression
cols_to_keep = ['admit', 'gre', 'gpa']
data = df[cols_to_keep].join(dummy_ranks.ix[:, 'prestige_2':])
print data.head()
#    admit  gre   gpa  prestige_2  prestige_3  prestige_4
# 0      0  380  3.61           0           1           0
# 1      1  660  3.67           0           1           0
# 2      1  800  4.00           0           0           0
# 3      1  640  3.19           0           0           1
# 4      0  520  2.93           0           0           1

# manually add the intercept
data['intercept'] = 1.0
```

logistic_prepping.py hosted with ❤ by **GitHub** **view raw**

Once that's done, we merge the new dummy columns into the original dataset and get rid of the `prestige` column which we no longer neeed.

Lastly we're going to add a constant term for our Logistic Regression. The `statsmodels` function we're going to be using requires that intercepts/contsants are specified explicitely.

Performing the regression

Performing the regression

Acutally doing the Logsitic Regression is quite simple. Specify the column containing the variable you're trying to predict followed by the columns that the model should use to make the prediction.

In our case we'll be predicting the `admit` column using `gre`, `gpa`, and the prestige dummy variables `prestige_2`, `prestige_3` and `prestige_4`. We're going to treat `prestige_1` as our baseline and exclude it from our fit. This is done to prevent multicollinearity, or the dummy variable trap caused by including a dummy variable for every single category.

```
train_cols = data.columns[1:]
# Index([gre, gpa, prestige_2, prestige_3, prestige_4], dtype=object)

logit = sm.Logit(data['admit'], data[train_cols])

# fit the model
result = logit.fit()
```

logistic_do_regression.py hosted with ❤ by GitHub view raw

Since we're doing a logistic regression, we're going to use the `statsmodels` Logit function. For details on other models available in `statsmodels`, check out their docs here.

Interpreting the results

One of my favorite parts about `statsmodels` is the summary output it gives. If you're coming from R, I think you'll like the output and find it very familiar too.

```
# cool enough to deserve it's own gist
print result.summary()
```

logistic_results.py hosted with ❤ by GitHub view raw

```
                          Logit Regression Results
```

```
        ==============================================================================
        Dep. Variable:                  admit   No. Observations:                  400
        Model:                          Logit   Df Residuals:                      394
        Method:                           MLE   Df Model:                            5
        Date:                Sun, 03 Mar 2013   Pseudo R-squ.:                 0.08292
        Time:                        12:34:59   Log-Likelihood:                -229.26
        converged:                       True   LL-Null:                       -249.99
                                                LLR p-value:                 7.578e-08
        ==============================================================================
                         coef    std err          z      P>|z|      [95.0% Conf. Int.]
        ------------------------------------------------------------------------------
        gre            0.0023      0.001      2.070      0.038         0.000     0.004
        gpa            0.8040      0.332      2.423      0.015         0.154     1.454
        prestige_2    -0.6754      0.316     -2.134      0.033        -1.296    -0.055
        prestige_3    -1.3402      0.345     -3.881      0.000        -2.017    -0.663
        prestige_4    -1.5515      0.418     -3.713      0.000        -2.370    -0.733
        intercept     -3.9900      1.140     -3.500      0.000        -6.224    -1.756
```

```
  -1.756
        ===============================================================
========
```

You get a great overview of the coeffecients of the model, how well those coeffecients fit, the overall fit quality, and several other statistical measures.

The result object also lets you to isolate and inspect parts of the model output. The confidence interval gives you an idea for how robust the coeffecients of the model are.

```
# look at the confidence interval of each coeffecient
print result.conf_int()
#                    0         1
# gre         0.000120  0.004409
# gpa         0.153684  1.454391
# prestige_2 -1.295751 -0.055135
# prestige_3 -2.016992 -0.663416
# prestige_4 -2.370399 -0.732529
# intercept  -6.224242 -1.755716
```

logistic_conf_int.py hosted with ❤ by GitHub view raw

In this example, we're very confident that there is an inverse relationship between the probability of being admitted and the prestige of a candidate's undergraduate school.

In other words, the probability of being accepted into a graduate program is higher for students who attended a top ranked undergraduate college (`prestige_1==True`) as opposed to a lower ranked school with, say, `prestige_4==True` (remember, a prestige of 1 is the most prestigious and a prestige of 4 is the least prestigious.

odds ratio

Take the exponential of each of the coeffecients to generate the odds ratios. This tells you how a 1 unit increase or decrease in a variable affects the odds of being admitted. For example, we can expect the odds of being admitted to decrease by about 50% if the prestige of a school is 2. UCLA gives a more in depth explanation of the odds ratio here.

```
# odds ratios only
print np.exp(result.params)
# gre           1.002267
# gpa           2.234545
# prestige_2    0.508931
# prestige_3    0.261792
# prestige_4    0.211938
# intercept     0.018500
```

logistic_odds_ratio.py hosted with ❤ by GitHub view raw

We can also do the same calculations using the coeffecients estimated using the confidence interval to get a better picture for how uncertainty in variables can impact the admission rate.

```
# odds ratios and 95% CI
params = result.params
conf = result.conf_int()
conf['OR'] = params
conf.columns = ['2.5%', '97.5%', 'OR']
print np.exp(conf)
#                   2.5%     97.5%        OR
# gre           1.000120  1.004418  1.002267
# gpa           1.166122  4.281877  2.234545
# prestige_2    0.273692  0.946358  0.508931
# prestige_3    0.133055  0.515089  0.261792
# prestige_4    0.093443  0.480692  0.211938
# intercept     0.001981  0.172783  0.018500
```

logistic_ci_and_est.py hosted with ❤ by GitHub view raw

Digging a little deeper

Digging a little deeper

As a way of evaluating our classifier, we're going to recreate the dataset with every logical combination of input values. This will allow us to see how the predicted probability of admission increases/decreases across different variables. First we're going to generate the combinations using a helper function called `cartesian` which I originally found here.

We're going to use `np.linspace` to create a range of values for "gre" and "gpa". This creates a range of linearly spaced values from a specified min and maximum value--in our case just the min/max observed values.

```
# instead of generating all possible values of GRE and GPA, we're going
# to use an evenly spaced range of 10 values from the min to the max
gres = np.linspace(data['gre'].min(), data['gre'].max(), 10)
print gres
# array([ 220.        ,  284.44444444,  348.88888889,  413.33333333,
#         477.77777778,  542.22222222,  606.66666667,  671.11111111,
#         735.55555556,  800.        ])
gpas = np.linspace(data['gpa'].min(), data['gpa'].max(), 10)
print gpas
# array([ 2.26      ,  2.45333333,  2.64666667,  2.84      ,  3.03333333,
#         3.22666667,  3.42      ,  3.61333333,  3.80666667,  4.        ]


# enumerate all possibilities
combos = pd.DataFrame(cartesian([gres, gpas, [1, 2, 3, 4], [1.]]))
# recreate the dummy variables
combos.columns = ['gre', 'gpa', 'prestige', 'intercept']
dummy_ranks = pd.get_dummies(combos['prestige'], prefix='prestige')
dummy_ranks.columns = ['prestige_1', 'prestige_2', 'prestige_3', 'prestige

# keep only what we need for making predictions
cols_to_keep = ['gre', 'gpa', 'prestige', 'intercept']
combos = combos[cols_to_keep].join(dummy_ranks.ix[:, 'prestige_2':])
```

```
# make predictions on the enumerated dataset
combos['admit_pred'] = result.predict(combos[train_cols])

print combos.head()
#    gre       gpa  prestige  intercept  prestige_2  prestige_3  prestige_
# 0  220  2.260000         1          1           0           0
# 1  220  2.260000         2          1           1           0
# 2  220  2.260000         3          1           0           1
# 3  220  2.260000         4          1           0           0
# 4  220  2.453333         1          1           0           0
```

logistic_cartesian.py hosted with ❤ by GitHub view raw

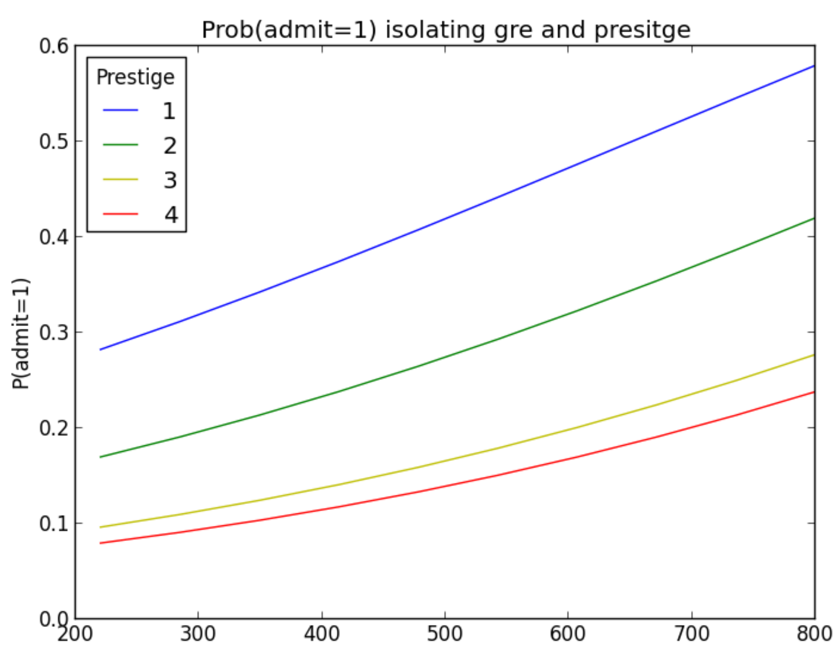
Now that we've generated our predictions, let's make some plots to visualize the results. I created a small helper function called `isolate_and_plot` which allows you to compare a given variable with the different prestige levels and the mean probability for that combination. To isolate presitge and the other variable I used a `pivot_table` which allows you to easily aggregate the data.

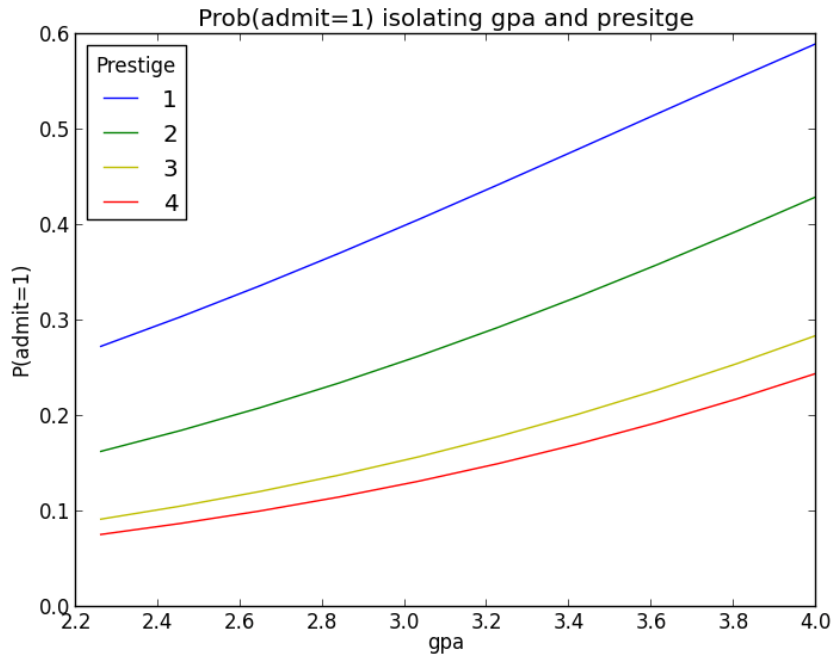
```
def isolate_and_plot(variable):
    # isolate gre and class rank
    grouped = pd.pivot_table(combos, values=['admit_pred'], rows=[variabl
                             aggfunc=np.mean)

    # in case you're curious as to what this looks like
    # print grouped.head()
    #                      admit_pred
    # gre        prestige
    # 220.000000 1           0.282462
    #            2           0.169987
    #            3           0.096544
    #            4           0.079859
    # 284.444444 1           0.311718
```

```
    # make a plot
    colors = 'rbgyrbgy'
    for col in combos.prestige.unique():
        plt_data = grouped.ix[grouped.index.get_level_values(1)==col]
        pl.plot(plt_data.index.get_level_values(0), plt_data['admit_pred'
                color=colors[int(col)])

    pl.xlabel(variable)
    pl.ylabel("P(admit=1)")
    pl.legend(['1', '2', '3', '4'], loc='upper left', title='Prestige')
    pl.title("Prob(admit=1) isolating " + variable + " and presitge")
    pl.show()

isolate_and_plot('gre')
isolate_and_plot('gpa')
```

logistic_isolate_and_plot.py hosted with ❤ by **GitHub** **view raw**

The resulting plots shows how gre, gpa, and prestige affect the admission levels. You can see how the probability of admission gradually increases as gre and gpa increase and that the different presitge levels yield drastic probabilities of admission (particularly the most/least prestigious schools).





Takeaways

Logistic Regression is an excellent algorithm for classification. Even though some of the sexier, black box classification algorithms like SVM and RandomForest can perform better in some cases, it's hard to deny the value in knowing exactly what your model is doing. Often times you can get by using RandomForest to select the features of your model and then rebuild the model with Logistic Regression using the best features.

Other resources

- UCLA Tutorial in R
- scikit-learn docs
- Pure Python implementation
- Basic examples w/ interactive tutorial

« SQL FOR PANDAS DATAFRAMES YHAT IS GOING TO PYCON »