# Logistic Regression

Researchers are often interested in setting up a model to analyze the relationship between some predictors (i.e., independent variables) and a response (i.e., dependent variable). Linear regression is commonly used when the response variable is continuous. One assumption of linear models is that the residual errors follow a normal distribution. This assumption fails when the response variable is categorical, so an ordinary linear model is not appropriate. This newsletter presents a regression model for a response variable that is dichotomous–having two categories. Examples are common: whether a plant lives or dies, whether a survey respondent agrees or disagrees with a statement, or whether an at-risk child graduates or drops out from high school.

In ordinary linear regression, the response variable (Y) is a linear function of the coefficients (B0, B1, etc.) that correspond to the predictor variables (X1, X2, etc.). A typical model would look like:

```
Y = B0 + B1*X1 + B2*X2 + B3*X3 + … + E
```

For a dichotomous response variable, we could set up a similar linear model to predict individuals' category memberships if numerical values are used to represent the two categories. Arbitrary values of 1 and 0 are chosen for mathematical convenience. Using the first example, we would assign Y = 1 if a plant lives and Y = 0 if a plant dies.

This linear model does not work well for a few reasons. First, the response values, 0 and 1, are arbitrary, so modeling the actual values of Y is not exactly of interest. Second, it is really the probability that each individual in the population responds with 0 or 1 that we are interested in modeling. For example, we may find that plants with a high level of a fungal infection (X1) fall into the category "the plant lives" (Y) less often than those plants with low level of infection. Thus, as the level of infection rises, the probability of a plant living decreases.

Thus, we might consider modeling P, the probability, as the response variable. Again, there are problems. Although the general decrease in probability is accompanied by a general increase in infection level, we know that P, like all probabilities, can only fall within the boundaries of 0 and 1. Consequently, it is better to assume that the relationship between X1 and P is sigmoidal (S-shaped), rather than a straight line.

It is possible, however, to find a linear relationship between X1 and a function of P. Although a number of functions work, one of the most useful is the logit function. It is the natural log of the odds that Y is equal to 1, which is simply the ratio of the probability that Y is 1 divided by the probability that Y is 0. The relationship between the logit of P and P itself is sigmoidal in shape. The regression equation that results is:

```
ln[P/(1-P)] = B0 + B1*X1 + B2*X2 + …
```

Although the left side of this equation looks intimidating, this way of expressing the probability results in the right side of the equation being linear and looking familiar to us. This helps us understand the meaning of the regression coefficients. The coefficients can easily be transformed so that their interpretation makes sense.

The logistic regression equation can be extended beyond the case of a dichotomous response variable to the cases of ordered categories and polytymous categories (more than two categories).

# Mathematics behind Logistic Regression

# Notation

The problem structure is the classic classification problem. Our data set $\mathcal{D}$ is composed of $N$ samples. Each sample is a tuple containing a feature vector and a label. For any sample $n$ the feature vector is a $d+1$ dimensional column vector denoted by $\mathbf{x}_n$ with $d$ real-valued components known as features. Samples are represented in homogeneous form with the first component equal to $1$: $x_0 = 1$. Vectors are bold-faced. The associated label is denoted $y_n$ and can take on only two values: $+1$ or $-1$.

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\} \mathbf{x}_n = \begin{bmatrix} 1 & x_1 & \ldots & x_d \end{bmatrix}^T$$

# Derivation

Despite the name logistic *regression* this is actually a probabilistic classification model. It is also a linear model which can be subjected to nonlinear transforms.

All linear models make use of a "signal" $s$ which is a linear combination of the input vector $\mathbf{x}$ components weighed by the corresponding components in a weight vector $\mathbf{w}$.

$$\mathbf{w} = \begin{bmatrix} w_0 & w_1 & \ldots & w_d \end{bmatrix}^T s = w_0 + w_1 x_1 + \ \ldots \ + w_d x_d = \sum_{i=0}^{d} w_i x_i = \mathbf{w} \cdot \mathbf{x} = \mathbf{w}^T \mathbf{x}$$

Note that the homogeneous representation (with the $1$ at the first component) allows us to include a constant offset using a more compact vector-only notation (instead of $\mathbf{w}^T \mathbf{x} + b$).

Linear classification passes the signal through a harsh threshold:

$$h(\mathbf{x}) = \text{sign}(s)$$

Linear regression uses the signal directly without modification:

$$h(\mathbf{x}) = s$$

Logistic regression passes the signal through the logistic/sigmoid but then treats the result as a probability:

$$h(\mathbf{x}) = \theta(s)$$

The logistic function (http://en.wikipedia.org/wiki/Logistic_function) is

$$\theta(s) = \frac{e^s}{1 + e^s} = \frac{1}{1 + e^{-s}}$$

There are many other formulas that can achieve a soft threshold such as the hyperbolic tangent, but this function results in some nice simplification.

We say that the data is generated by a noisy target.

$$P(y \mid \mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{for } y = +1 \\ 1 - f(\mathbf{x}) & \text{for } y = -1 \end{cases}$$

With this noisy target we want to learn a hypothesis $h(\mathbf{x})$ that best fits the above noisy target according to some error function.

$$h(\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x}) \approx f(\mathbf{x})$$

It's important to note that the data does not tell you the probability of a label but rather what label the sample has after being generated by the target distribution.

# Error Measure

To learn a good hypothesis we want to find a hypothesis parameterization $\mathbf{w}$ (the weight vector) that minimizes some in-sample error measure $E_{\text{in}}$.

$$\mathbf{w}_h = \text{argmin}_{\mathbf{w}} E_{\text{in}}(\mathbf{w})$$

The error measure we will use is both plausible and nice. It is based on likelihood which is the probability of generating the data given a model.

If our hypothesis is close to our target distribution ($h \approx f$) then we expect that probability of generating the data to be high.

There is some controversy with using likelihood. We are really looking for the most probable hypothesis given the data: $\text{argmax}_h P(h \mid \mathbf{x})$. The likelihood approach is looking for the hypothesis that makes the data most probable: $\text{argmax}_h P(\mathbf{x} \mid h)$.

The Bayesian approach tackles this issue using Bayes' Theorem but introduces other issues such as choosing priors.

To determine the likelihood we assume the data was generated with our hypothesis $h$:

$$P(y \mid \mathbf{x}) = \begin{cases} h(\mathbf{x}) & \text{for } y = +1 \\ 1 - h(\mathbf{x}) & \text{for } y = -1 \end{cases}$$

where $h(\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x})$.

We don't want to deal with cases so we take advantage of a nice property of the logistic function: $\theta(-s) = 1 - \theta(s)$.

$$\text{if } y = +1 \text{ then } h(\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x}) = \theta(y\ \mathbf{w}^T\mathbf{x}) \text{if } y = -1 \text{ then } 1 - h(\mathbf{x}) = 1 - \theta(\mathbf{w}^T\mathbf{x}) = \theta(-\mathbf{w}^T\mathbf{x}) = \theta(y\ \mathbf{w}^T\mathbf{x})$$

Using this simplification,

$$P(y \mid \mathbf{x}) = \theta(y\ \mathbf{w}^T\mathbf{x})$$

The likelihood is defined for a data set $\mathcal{D}$ with $N$ samples given a hypothesis (denoted arbitrarily $g$ here):

$$L(\mathcal{D} \mid g) = \prod_{n=1}^{N} P(y_n \mid \mathbf{x}_n) = \prod_{n=1}^{N} \theta(y_n\ \mathbf{w}_g^T\mathbf{x}_n)$$

Now finding a good hypothesis is a matter of finding a hypothesis parameterization $\mathbf{w}$ that maximizes the likelihood.

$$\mathbf{w}_h = \text{argmax}_\mathbf{w}\ L(\mathcal{D} \mid h) = \text{argmax}_\mathbf{w}\ \theta(y_n\ \mathbf{w}^T\mathbf{x}_n)$$

Maximizing the likelihood is equivalent to maximizing the log of the function since the natural logarithm is a monotonically increasing function:

$$\text{argmax}_\mathbf{w}\ \ln\left(\prod_{n=1}^{N} \theta(y_n\ \mathbf{w}^T\mathbf{x}_n)\right)$$

We can maximize the above proportional to a constant as well so we'll tack on a $\frac{1}{N}$:

$$\text{argmax}_\mathbf{w}\ \frac{1}{N}\ln\left(\prod_{n=1}^{N} \theta(y_n\ \mathbf{w}^T\mathbf{x}_n)\right)$$

Now maximizing that is the same as minimizing its negative:

$$\text{argmin}_\mathbf{w}\left[-\frac{1}{N}\ln\left(\prod_{n=1}^{N} \theta(y_n\ \mathbf{w}^T\mathbf{x}_n)\right)\right]$$

If we move the negative into the log and the log into the product we turn the product into a sum of logs:

$$\text{argmin}_\mathbf{w}\ \frac{1}{N}\sum_{n=1}^{N} \ln\left(\frac{1}{\theta(y_n\ \mathbf{w}^T\mathbf{x}_n)}\right)$$

Expanding the logistic function,

$$\text{argmin}_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{y_n \mathbf{w}^T \mathbf{x}_n}\right)$$

Now we have a much nicer form for the error measure known as the "cross-entropy" error.

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{-y_n \mathbf{w}^T \mathbf{x}_n}\right)$$

This is nice because it can be interpreted as the average point error where the point error function is

$$e(h(\mathbf{x}_n), y_n) = \ln\left(1 + e^{-y_n \mathbf{w}^T \mathbf{x}_n}\right) E_{\text{in}}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} e(h(\mathbf{x}_n), y_n)$$

So to learn a hypothesis we'll want to perform the following optimization:

$$\mathbf{w}_h = \text{argmin}_{\mathbf{w}} E_{\text{in}}(\mathbf{w}) = \text{argmin}_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{y_n \mathbf{w}^T \mathbf{x}_n}\right)$$

# Learning Algorithm

The learning algorithm is how we search the set of possible hypotheses (hypothesis space $\mathcal{H}$) for the best parameterization (in this case the weight vector $\mathbf{w}$). This search is an optimization problem looking for the hypothesis that optimizes an error measure.

There is no nice, closed-form solution like with least-squares linear regression so we will use gradient descent instead. Specifically we will use batch gradient descent which calculates the gradient from all data points in the data set.

Luckily, our "cross-entropy" error measure is convex so there is only one minimum. Thus the minimum we arrive at is the global minimum.

Gradient descent is a general method and requires twice differentiability for smoothness. It updates the parameters using a first-order approximation of the error surface.

$$\mathbf{w}_{i+1} = \mathbf{w}_i + \nabla E_{\text{in}}(\mathbf{w}_i)$$

To learn we're going to minimize the following error measure using batch gradient descent.

$$e(h(\mathbf{x}_n), y_n) = \ln\left(1 + e^{-y_n \mathbf{w}^T \mathbf{x}_n}\right) E_{\text{in}}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} e(h(\mathbf{x}_n), y_n) = \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{-y_n \mathbf{w}^T \mathbf{x}_n}\right)$$

We'll need to the derivative of the point loss function and possibly some abuse of notation.

$$\frac{d}{d\mathbf{w}} e(h(\mathbf{x}_n), y_n) = \frac{-y_n \, \mathbf{x}_n \, e^{-y_n \mathbf{w}^T \mathbf{x}_n}}{1 + e^{-y_n \mathbf{w}^T \mathbf{x}_n}} = -\frac{y_n \, \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^T \mathbf{x}_n}}$$

With the point loss derivative we can determine the gradient of the in-sample error:

$$\begin{aligned}
\nabla E_{\text{in}}(\mathbf{w}) &= \frac{d}{d\mathbf{w}} \left[ \frac{1}{N} \sum_{n=1}^{N} e(h(\mathbf{x}_n), y_n) \right] \\
&= \frac{1}{N} \sum_{n=1}^{N} \frac{d}{d\mathbf{w}} e(h(\mathbf{x}_n), y_n) \\
&= \frac{1}{N} \sum_{n=1}^{N} \left( -\frac{y_n \, \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^T \mathbf{x}_n}} \right) \\
&= -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \, \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^T \mathbf{x}_n}}
\end{aligned}$$

Our weight update rule per batch gradient descent becomes

$$\begin{aligned}
\mathbf{w}_{i+1} &= \mathbf{w}_i - \eta \, \nabla E_{\text{in}}(\mathbf{w}_i) \\
&= \mathbf{w}_i - \eta \left( -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \, \mathbf{x}_n}{1 + e^{y_n \mathbf{w}_i^T \mathbf{x}_n}} \right) \\
&= \mathbf{w}_i + \eta \left( \frac{1}{N} \sum_{n=1}^{N} \frac{y_n \, \mathbf{x}_n}{1 + e^{y_n \mathbf{w}_i^T \mathbf{x}_n}} \right)
\end{aligned}$$

where $\eta$ is our learning rate.

### Enough with the theory, now jump to the implimentation. We will look at 2 libraries for the same.

# Logistic Regression with statsmodel

We'll be using the same dataset as UCLA's Logit Regression 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 prestige 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.

In [1]:

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

In [12]:

```
# read the data in
df = pd.read_csv("https://stats.idre.ucla.edu/stat/data/binary.csv")
```

In [14]:

```
df.head()
```

Out[14]:

| | 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 |

In [15]:

```
# rename the 'rank' column because there is also a DataFrame method called 'rank'
df.columns = ["admit", "gre", "gpa", "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. There's also function for calculating the standard deviation, std.

A feature I really like in pandas is the pivot_table/crosstab aggregations. crosstab makes it really easy to do multidimensional frequency tables. You might want to play around with this to look at different cuts of the data.

In [16]:

```
df.describe()
```

Out[16]:

| | 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 |

In [18]:

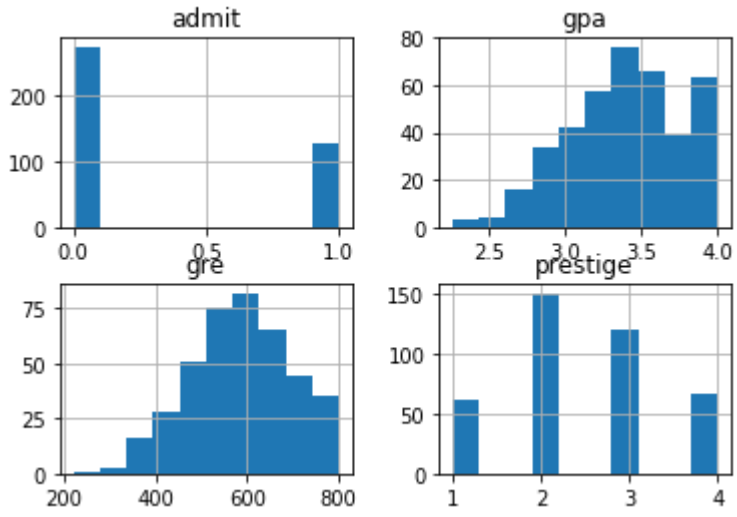
```
# frequency table cutting presitge and whether or not someone was admitted
pd.crosstab(df['admit'], df['prestige'], rownames=['admit'])
```

Out[18]:

| **prestige** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| **admit** | | | | |
| **0** | 28 | 97 | 93 | 55 |
| **1** | 33 | 54 | 28 | 12 |

In [19]:

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



## 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.

In [20]:

```
# dummify rank
dummy_ranks = pd.get_dummies(df['prestige'], prefix='prestige')
dummy_ranks.head()
```

Out[20]:

| | 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 |

In [21]:

```
# 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':])
data.head()
```

Out[21]:

| | 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 |

In [22]:

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

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 need.

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/constants are specified explicitly.

## Performing the regression

Actually doing the Logistic 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.

In [23]:

```
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()
```

```
Optimization terminated successfully.
         Current function value: 0.573147
         Iterations 6
```

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.

In [24]:

```
result.summary()
```

Out[24]:

Logit Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | admit | **No. Observations:** | 400 |
| **Model:** | Logit | **Df Residuals:** | 394 |
| **Method:** | MLE | **Df Model:** | 5 |
| **Date:** | Sat, 09 Sep 2017 | **Pseudo R-squ.:** | 0.08292 |
| **Time:** | 20:24:18 | **Log-Likelihood:** | -229.26 |
| **converged:** | True | **LL-Null:** | -249.99 |
| | | **LLR p-value:** | 7.578e-08 |

| | **coef** | **std err** | **z** | **P>\|z\|** | **[0.025** | **0.975]** |
|---|---|---|---|---|---|---|
| **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 |

# Logistic Regression with scikit-learn

## Dataset

The dataset I chose is the affairs dataset (http://statsmodels.sourceforge.net/stable/datasets/generated/fair.html) that comes with Statsmodels (http://statsmodels.sourceforge.net/). It was derived from a survey of women in 1974 by Redbook magazine, in which married women were asked about their participation in extramarital affairs. More information about the study is available in a 1978 paper (http://fairmodel.econ.yale.edu/rayfair/pdf/1978a200.pdf) from the Journal of Political Economy.

## Description of Variables

The dataset contains 6366 observations of 9 variables:

- `rate_marriage`: woman's rating of her marriage (1 = very poor, 5 = very good)
- `age`: woman's age
- `yrs_married`: number of years married
- `children`: number of children
- `religious`: woman's rating of how religious she is (1 = not religious, 4 = strongly religious)
- `educ`: level of education (9 = grade school, 12 = high school, 14 = some college, 16 = college graduate, 17 = some graduate school, 20 = advanced degree)
- `occupation`: woman's occupation (1 = student, 2 = farming/semi-skilled/unskilled, 3 = "white collar", 4 = teacher/nurse/writer/technician/skilled, 5 = managerial/business, 6 = professional with advanced degree)
- `occupation_husb`: husband's occupation (same coding as above)
- `affairs`: time spent in extra-marital affairs

## Problem Statement

I decided to treat this as a classification problem by creating a new binary variable `affair` (did the woman have at least one affair?) and trying to predict the classification for each woman.

## Import modules

In [25]:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
from patsy import dmatrices
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import train_test_split
from sklearn import metrics
from sklearn.cross_validation import cross_val_score
```

```
/usr/local/lib/python3.5/dist-packages/sklearn/cross_validation.py:44: Dep
recationWarning: This module was deprecated in version 0.18 in favor of th
e model_selection module into which all the refactored classes and functio
ns are moved. Also note that the interface of the new CV iterators are dif
ferent from that of this module. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

# Data Pre-Processing

First, let's load the dataset and add a binary `affair` column.

In [26]:

```
# load dataset
dta = sm.datasets.fair.load_pandas().data

# add "affair" column: 1 represents having affairs, 0 represents not
dta['affair'] = (dta.affairs > 0).astype(int)
```

# Data Exploration

In [27]:

```
dta.groupby('affair').mean()
```

Out[27]:

| | rate_marriage | age | yrs_married | children | religious | educ | occup |
|---|---|---|---|---|---|---|---|
| **affair** | | | | | | | |
| **0** | 4.329701 | 28.390679 | 7.989335 | 1.238813 | 2.504521 | 14.322977 | 3.405 |
| **1** | 3.647345 | 30.537019 | 11.152460 | 1.728933 | 2.261568 | 13.972236 | 3.463 |

We can see that on average, women who have affairs rate their marriages lower, which is to be expected. Let's take another look at the `rate_marriage` variable.

In [28]:

```
dta.groupby('rate_marriage').mean()
```

Out[28]:

| | age | yrs_married | children | religious | educ | occupation | oc |
|---|---|---|---|---|---|---|---|
| **rate_marriage** | | | | | | | |
| **1.0** | 33.823232 | 13.914141 | 2.308081 | 2.343434 | 13.848485 | 3.232323 | 3. |
| **2.0** | 30.471264 | 10.727011 | 1.735632 | 2.330460 | 13.864943 | 3.327586 | 3. |
| **3.0** | 30.008056 | 10.239174 | 1.638469 | 2.308157 | 14.001007 | 3.402820 | 3. |
| **4.0** | 28.856601 | 8.816905 | 1.369536 | 2.400981 | 14.144514 | 3.420161 | 3. |
| **5.0** | 28.574702 | 8.311662 | 1.252794 | 2.506334 | 14.399776 | 3.454918 | 3. |

An increase in `age`, `yrs_married`, and `children` appears to correlate with a declining marriage rating.

# Data Visualization

In [29]:

```
# show plots in the notebook
%matplotlib inline
```

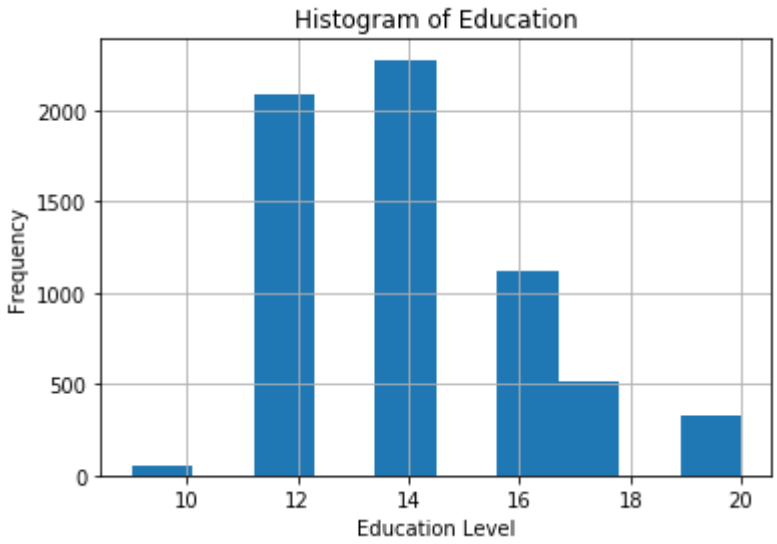
Let's start with histograms of education and marriage rating.

In [30]:

```
# histogram of education
dta.educ.hist()
plt.title('Histogram of Education')
plt.xlabel('Education Level')
plt.ylabel('Frequency')
```

Out[30]:

```
<matplotlib.text.Text at 0x7f0da48ff278>
```

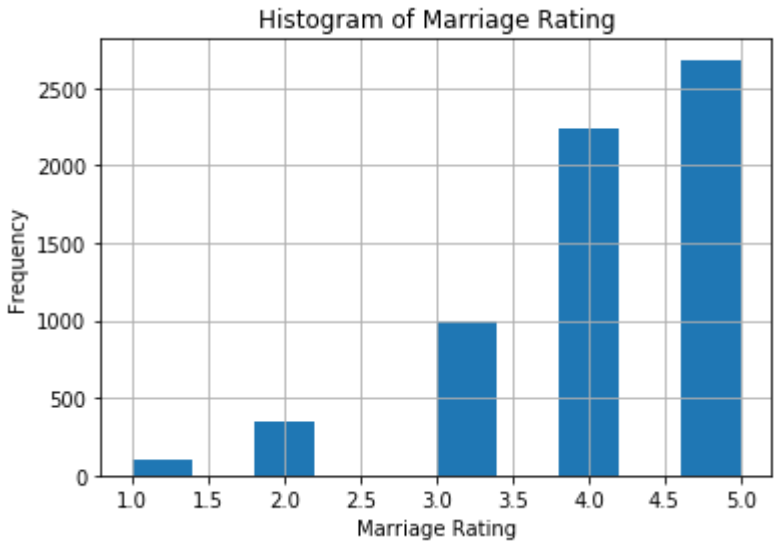


In [31]:

```
# histogram of marriage rating
dta.rate_marriage.hist()
plt.title('Histogram of Marriage Rating')
plt.xlabel('Marriage Rating')
plt.ylabel('Frequency')
```

Out[31]:

```
<matplotlib.text.Text at 0x7f0da7a6d160>
```



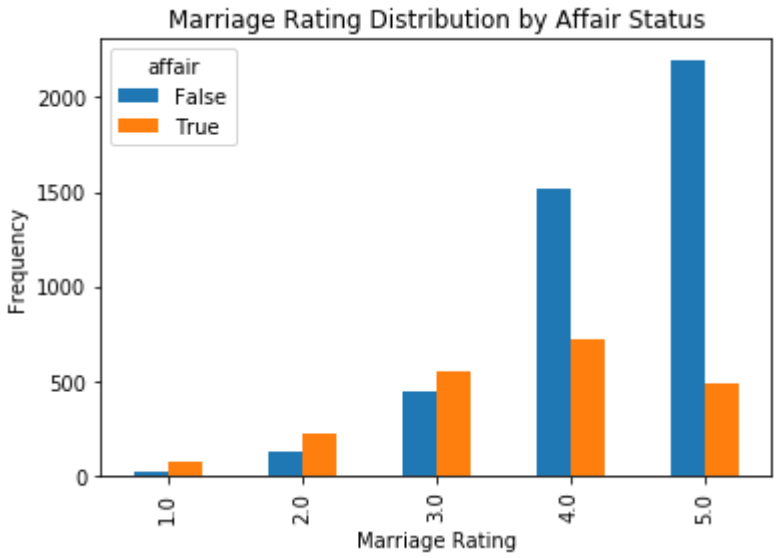

Let's take a look at the distribution of marriage ratings for those having affairs versus those not having affairs.

In [32]:

```
# barplot of marriage rating grouped by affair (True or False)
pd.crosstab(dta.rate_marriage, dta.affair.astype(bool)).plot(kind='bar')
plt.title('Marriage Rating Distribution by Affair Status')
plt.xlabel('Marriage Rating')
plt.ylabel('Frequency')
```

Out[32]:

```
<matplotlib.text.Text at 0x7f0da7b59198>
```



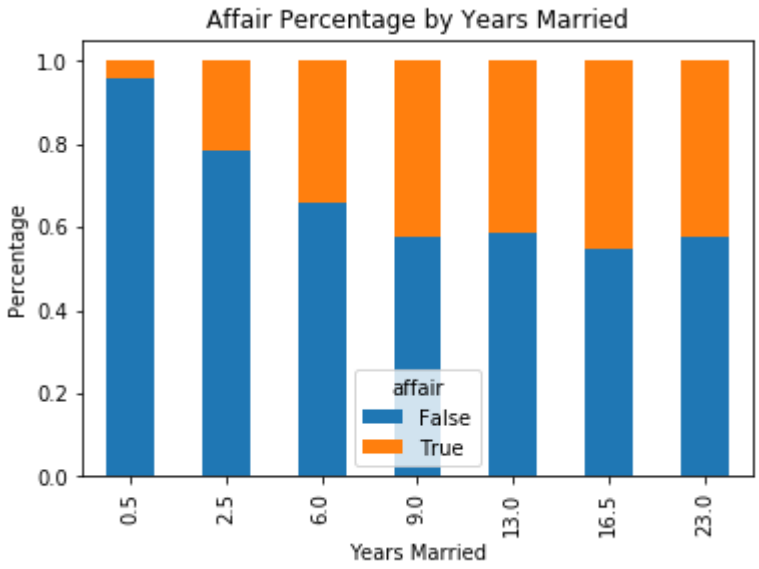

Let's use a stacked barplot to look at the percentage of women having affairs by number of years of marriage.

In [33]:

```
affair_yrs_married = pd.crosstab(dta.yrs_married, dta.affair.astype(bool))
affair_yrs_married.div(affair_yrs_married.sum(1).astype(float), axis=0).plot(kind='bar'
, stacked=True)
plt.title('Affair Percentage by Years Married')
plt.xlabel('Years Married')
plt.ylabel('Percentage')
```

Out[33]:

```
<matplotlib.text.Text at 0x7f0da7b00da0>
```




# Prepare Data for Logistic Regression

To prepare the data, I want to add an intercept column as well as dummy variables for `occupation` and `occupation_husb`, since I'm treating them as categorial variables. The dmatrices function from the patsy module (http://patsy.readthedocs.org/en/latest/) can do that using formula language.

In [35]:

```
# create dataframes with an intercept column and dummy variables for
# occupation and occupation_husb
y, X = dmatrices('affair ~ rate_marriage + age + yrs_married + children + \
                  religious + educ + C(occupation) + C(occupation_husb)',
                  dta, return_type="dataframe")
X.columns
```

Out[35]:

```
Index(['Intercept', 'C(occupation)[T.2.0]', 'C(occupation)[T.3.0]',
       'C(occupation)[T.4.0]', 'C(occupation)[T.5.0]', 'C(occupation)[T.6.
0]',
       'C(occupation_husb)[T.2.0]', 'C(occupation_husb)[T.3.0]',
       'C(occupation_husb)[T.4.0]', 'C(occupation_husb)[T.5.0]',
       'C(occupation_husb)[T.6.0]', 'rate_marriage', 'age', 'yrs_married',
       'children', 'religious', 'educ'],
      dtype='object')
```

The column names for the dummy variables are ugly, so let's rename those.

In [36]:

```
# fix column names of X
X = X.rename(columns = {'C(occupation)[T.2.0]':'occ_2',
                        'C(occupation)[T.3.0]':'occ_3',
                        'C(occupation)[T.4.0]':'occ_4',
                        'C(occupation)[T.5.0]':'occ_5',
                        'C(occupation)[T.6.0]':'occ_6',
                        'C(occupation_husb)[T.2.0]':'occ_husb_2',
                        'C(occupation_husb)[T.3.0]':'occ_husb_3',
                        'C(occupation_husb)[T.4.0]':'occ_husb_4',
                        'C(occupation_husb)[T.5.0]':'occ_husb_5',
                        'C(occupation_husb)[T.6.0]':'occ_husb_6'})
```

We also need to flatten y into a 1-D array, so that scikit-learn will properly understand it as the response variable.

In [37]:

```
# flatten y into a 1-D array
y = np.ravel(y)
```

# Logistic Regression

Let's go ahead and run logistic regression on the entire data set, and see how accurate it is!

In [38]:

```
# instantiate a logistic regression model, and fit with X and y
model = LogisticRegression()
model = model.fit(X, y)

# check the accuracy on the training set
model.score(X, y)
```

Out[38]:

```
0.72588752748978946
```

73% accuracy seems good, but what's the null error rate?

In [39]:

```
# what percentage had affairs?
y.mean()
```

Out[39]:

```
0.32249450204209867
```

Only 32% of the women had affairs, which means that you could obtain 68% accuracy by always predicting "no". So we're doing better than the null error rate, but not by much.

Let's examine the coefficients to see what we learn.

In [41]:

```
# examine the coefficients
X.columns, np.transpose(model.coef_)
```

Out[41]:

```
(Index(['Intercept', 'occ_2', 'occ_3', 'occ_4', 'occ_5', 'occ_6', 'occ_hus
b_2',
       'occ_husb_3', 'occ_husb_4', 'occ_husb_5', 'occ_husb_6', 'rate_marr
iage',
       'age', 'yrs_married', 'children', 'religious', 'educ'],
      dtype='object'), array([[ 1.48986215],
       [ 0.18804152],
       [ 0.49891971],
       [ 0.25064099],
       [ 0.83897693],
       [ 0.8340083 ],
       [ 0.19057989],
       [ 0.29777984],
       [ 0.16135349],
       [ 0.18771784],
       [ 0.19394847],
       [-0.70311586],
       [-0.05841769],
       [ 0.10567657],
       [ 0.01692027],
       [-0.37113376],
       [ 0.00401557]]))
```

Increases in marriage rating and religiousness correspond to a decrease in the likelihood of having an affair. For both the wife's occupation and the husband's occupation, the lowest likelihood of having an affair corresponds to the baseline occupation (student), since all of the dummy coefficients are positive.

# Model Evaluation Using a Validation Set

So far, we have trained and tested on the same set. Let's instead split the data into a training set and a testing set.

In [42]:

```
# evaluate the model by splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0
)
model2 = LogisticRegression()
model2.fit(X_train, y_train)
```

Out[42]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

We now need to predict class labels for the test set. We will also generate the class probabilities, just to take a look.

In [44]:

```
# predict class labels for the test set
predicted = model2.predict(X_test)
predicted
```

Out[44]:

```
array([ 1.,  0.,  0., ...,  0.,  0.,  0.])
```

In [45]:

```
# generate class probabilities
probs = model2.predict_proba(X_test)
probs
```

Out[45]:

```
array([[ 0.35148525,  0.64851475],
       [ 0.90955539,  0.09044461],
       [ 0.72569407,  0.27430593],
       ...,
       [ 0.55730636,  0.44269364],
       [ 0.81211049,  0.18788951],
       [ 0.74732836,  0.25267164]])
```

As you can see, the classifier is predicting a 1 (having an affair) any time the probability in the second column is greater than 0.5.

Now let's generate some evaluation metrics.

In [46]:

```
# generate evaluation metrics
print(metrics.accuracy_score(y_test, predicted))
print(metrics.roc_auc_score(y_test, probs[:, 1]))
```

```
0.729319371728
0.745948078253
```

The accuracy is 73%, which is the same as we experienced when training and predicting on the same data.

We can also see the confusion matrix and a classification report with other metrics.

In [47]:

```
print(metrics.confusion_matrix(y_test, predicted))
print(metrics.classification_report(y_test, predicted))
```

```
[[1169  134]
 [ 383  224]]
             precision    recall  f1-score   support

        0.0       0.75      0.90      0.82      1303
        1.0       0.63      0.37      0.46       607

avg / total       0.71      0.73      0.71      1910
```

# Model Evaluation Using Cross-Validation

Now let's try 10-fold cross-validation, to see if the accuracy holds up more rigorously.

In [48]:

```
# evaluate the model using 10-fold cross-validation
scores = cross_val_score(LogisticRegression(), X, y, scoring='accuracy', cv=10)
scores, scores.mean()
```

Out[48]:

```
(array([ 0.72100313,  0.70219436,  0.73824451,  0.70597484,  0.70597484,
         0.72955975,  0.7327044 ,  0.70440252,  0.75157233,  0.75      ]),
 0.7241630685514876)
```

Looks good. It's still performing at 73% accuracy.

# Predicting the Probability of an Affair

Just for fun, let's predict the probability of an affair for a random woman not present in the dataset. She's a 25-year-old teacher who graduated college, has been married for 3 years, has 1 child, rates herself as strongly religious, rates her marriage as fair, and her husband is a farmer.

In [50]:

```
model.predict_proba(np.array([[1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 3, 25, 3, 1, 4,
                               16]]))
```

Out[50]:

```
array([[ 0.77472403,  0.22527597]])
```

The predicted probability of an affair is 23%.