Linear Regression

What is Regression Analysis?

Regression analysis is a form of predictive modelling technique which investigates the relationship between a dependent (target) and independent variable (s) (predictor). This technique is used for forecasting, time series modelling and finding the causal effect relationship between the variables. For example, relationship between rash driving and number of road accidents by a driver is best studied through regression.

Regression analysis is an important tool for modelling and analyzing data. Here, we fit a curve / line to the data points, in such a manner that the differences between the distance of data points from the curve or line is minimized. The topic will be explained in detail in coming sections.

Why do we use Regression Analysis?

As mentioned above,Regression analysis estimates the relationship between two or more variables. Let’s understand this with an easy example:

Let’s say, you want to estimate growth in sales of a company based on current economic conditions. You have the recent company data which indicates that the growth in sales is around two and a half times the growth in the economy. Using this insight, we can predict future sales of the company based on current & past information.

There are multiple benefits of using Regression analysis. They are as follows:

- It indicates the significant relationships between dependent variable and independent variable.
- It indicates the strength of impact of multiple independent variables on dependent variable.

Regression analysis also allows us to compare the effects of variables measured on different scales, such as the effect of price changes and the number of promotional activities. These benefits help Market Researchers / Data Analysts / Data Scientists to eliminate and evaluate the best set of variables to be used for building predictive models.

Linear Regression

It is one of the most widely known modeling technique. Linear regression is usually among the first few topics which people pick while learning predictive modeling. In this technique, the dependent variable is continuous, independent variable(s) can be continuous or discrete, and nature of regression line is linear.

Linear Regression establishes a relationship between dependent variable (Y) and one or more independent variables (X) using a best fit straight line (also known as regression line).

It is represented by an equation Y=a+b*X + e, where a is intercept, b is slope of the line and e is error term. This equation can be used to predict the value of target variable based on given predictor variable(s).

In [19]:
```
# imports
import pandas as pd
import matplotlib.pyplot as plt

# this allows plots to appear directly in the notebook
%matplotlib inline
```

Example: Advertising Data

Let's take a look at some data, ask some questions about that data, and then use Linear regression to answer those questions!

In [20]:
```
# read data into a DataFrame
data = pd.read_csv('http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv', index_co
data.head()
```

Out[20]:

	TV	Radio	Newspaper	Sales
1	230.1	37.8	69.2	22.1
2	44.5	39.3	45.1	10.4
3	17.2	45.9	69.3	9.3
4	151.5	41.3	58.5	18.5
5	180.8	10.8	58.4	12.9

What are the **features**?

- TV: Advertising dollars spent on TV for a single product in a given market (in thousands of dollars)
- Radio: Advertising dollars spent on Radio
- Newspaper: Advertising dollars spent on Newspaper

What is the **response**?

- Sales: sales of a single product in a given market (in thousands of widgets)

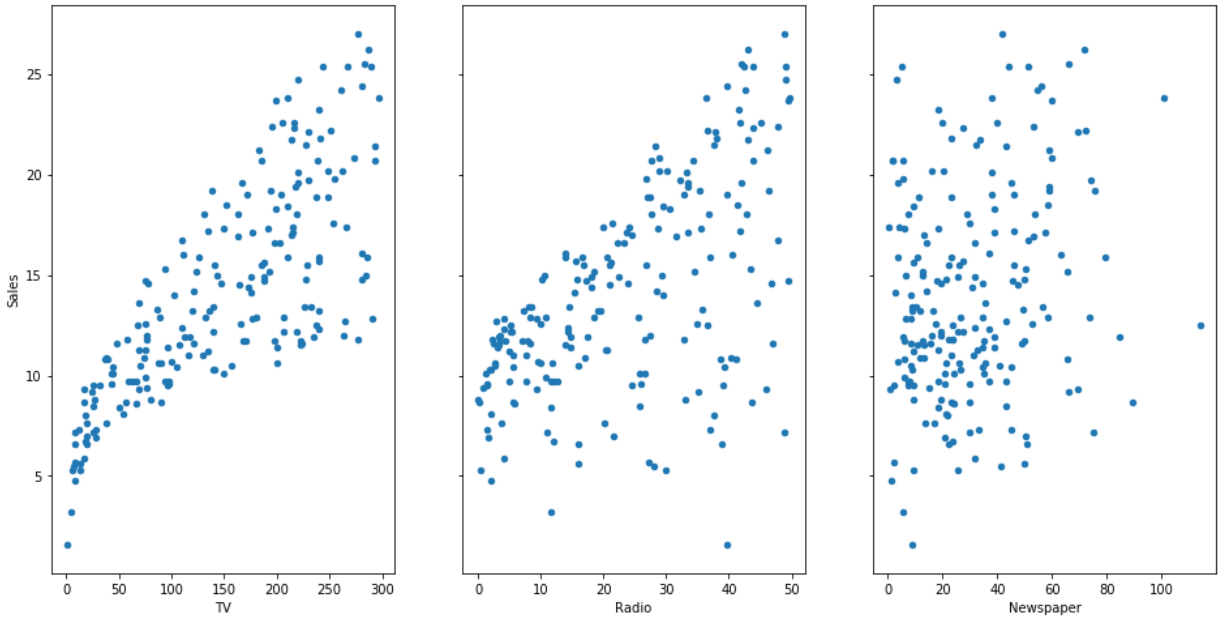
In [21]:
```
# print the shape of the DataFrame
data.shape
```

Out[21]:
```
(200, 4)
```

There are 200 **observations**, and thus 200 markets in the dataset.

In [22]:
```
# visualize the relationship between the features and the response using scatterp
fig, axs = plt.subplots(1, 3, sharey=True)
data.plot(kind='scatter', x='TV', y='Sales', ax=axs[0], figsize=(16, 8))
data.plot(kind='scatter', x='Radio', y='Sales', ax=axs[1])
data.plot(kind='scatter', x='Newspaper', y='Sales', ax=axs[2])
```

Out[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f37b3f59b38>
```



Questions About the Advertising Data

Let's pretend you work for the company that manufactures and markets this widget. The company might ask you the following: On the basis of this data, how should we spend our advertising money in the future?

These general questions might lead you to more specific questions:

1. Is there a relationship between ads and sales?
2. How strong is that relationship?
3. Which ad types contribute to sales?
4. What is the effect of each ad type of sales?
5. Given ad spending in a particular market, can sales be predicted?

We will explore these questions below!

Simple Linear Regression

Simple Linear regression is an approach for predicting a **quantitative response** using a **single feature** (or "predictor" or "input variable"). It takes the following form:

$$y = \beta_0 + \beta_1 x$$

What does each term represent?

- y is the response
- x is the feature
- β_0 is the intercept
- β_1 is the coefficient for x

Together, β_0 and β_1 are called the **model coefficients**. To create your model, you must "learn" the values of these coefficients. And once we've learned these coefficients, we can use the model to predict Sales!

Estimating ("Learning") Model Coefficients

Generally speaking, coefficients are estimated using the **least squares criterion**, which means we find the line (mathematically) which minimizes the **sum of squared residuals** (or "sum of squared errors"):

What elements are present in the diagram?

- The black dots are the **observed values** of x and y.
- The blue line is the **least squares line**.
- The red lines are the **residuals**, which is the distance between the observed values and the least squares line.

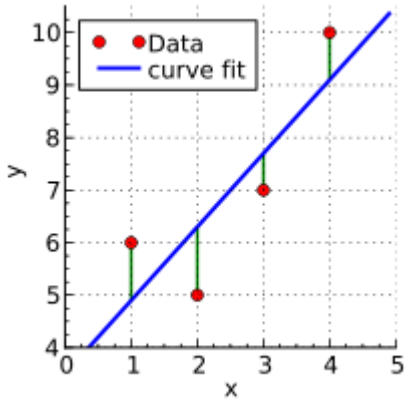
How do the model coefficients relate to the least squares line?

- β_0 is the **intercept** (the value of y when x=0)
- β_1 is the **slope** (the change in y divided by change in x)

Here is a graphical depiction of those calculations:

The mathematics behind the Least Squares Method.

Take a quick look at the plot created. Now consider each point, and know that each of them have a coordinate in the form (X,Y). Now draw an imaginary line between each point and the current "best-fit" line. We'll call the distanace between each point and the current best-fit line as D. To get a quick image of what we're trying to visualize, take a look at the picture below:



Now as before, we're labeling each green line as having a distance D, and each red point as having a coordinate of (X,Y). Then we can define our best fit line as the line having the property were:

$$D_1^2 + D_2^2 + D_3^2 + D_4^2 + \ldots + D_N^2$$

So how do we find this line? The least-square line approximating the set of points:

$$(X, Y)_1, (X, Y)_2, (X, Y)_3, (X, Y)_4, (X, Y)_5,$$

has the equation:

$$Y = a_0 + a_1 X$$

this is basically just a rewritten form of the standard equation for a line:

$$Y = mx + b$$

We can solve for these constants a0 and a1 by simultaneously solving these equations:

$$\Sigma Y = a_0 N + a_1 \Sigma X$$
$$\Sigma XY = a_0 \Sigma X + a_1 \Sigma X^2$$

These are called the normal equations for the least squares line. There are further steps that can be taken in rearranging these equations to solve for y, but we'll let scikit-learn do the rest of the heavy lifting here.

scikit-learn

Introduction

Since its release in 2007, scikit-learn has become one of the most popular open source Machine Learning libraries for Python. scikit-learn provides algorithms for Machine Learning tasks including classification, regression, dimensionality reduction, and clustering. It also provides modules for extracting features, processing data, and evaluating models.

Conceived as an extension to the SciPy library, scikit-learn is built on the popular Python libraries NumPy and matplotlib. NumPy extends Python to support efficient operations on large arrays and multidimensional matrices. matplotlib provides visualization tools, and SciPy provides modules for scientific computing.

scikit-learn is popular for academic research because it has a well-documented, easy-to-use, and versatile API. Developers can use scikit-learn to experiment with different algorithms by changing only a few lines of the code. scikit-learn wraps some popular implementations of machine learning algorithms, such as LIBSVM and LIBLINEAR. Other Python libraries, including NLTK, include wrappers for scikit-learn. scikit-learn also includes a variety of datasets, allowing developers to focus on algorithms rather than obtaining and cleaning data.

Licensed under the permissive BSD license, scikit-learn can be used in commercial applications without restrictions. Many of scikit-learn's algorithms are fast and scalable to all but massive datasets. Finally, scikit-learn is noted for its reliability; much of the library is covered by automated tests.

In [23]:
```
# create X and y
feature_cols = ['TV']
X = data[feature_cols]
y = data.Sales

# follow the usual sklearn pattern: import, instantiate, fit
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(X, y)

# print intercept and coefficients
print(lm.intercept_)
print(lm.coef_)
```

```
7.03259354913
[ 0.04753664]
```

Interpreting Model Coefficients

How do we interpret the TV coefficient (β_1)?

- A "unit" increase in TV ad spending is **associated with** a 0.047537 "unit" increase in Sales.
- Or more clearly: An additional $1,000 spent on TV ads is **associated with** an increase in sales of 47.537 widgets.

Note that if an increase in TV ad spending was associated with a **decrease** in sales, β_1 would be **negative**.

Using the Model for Prediction

Let's say that there was a new market where the TV advertising spend was **$50,000**. What would we predict for the Sales in that market?

$$y = \beta_0 + \beta_1 x$$
$$y = 7.032594 + 0.047537 \times 50$$

In [24]:
```
# manually calculate the prediction
7.032594 + 0.047537*50
```

Out[24]:
```
9.409444
```

Thus, we would predict Sales of **9,409 widgets** in that market.

In [25]:
```
# you have to create a DataFrame since the Statsmodels formula interface expects
X_new = pd.DataFrame({'TV': [50]})
X_new.head()
```

Out[25]:

	TV
0	50

In [26]:
```
# use the model to make predictions on a new value
lm.predict(X_new)
```

Out[26]:
```
array([ 9.40942557])
```

Plotting the Least Squares Line

Let's make predictions for the **smallest and largest observed values of x**, and then use the predicted values to plot the least squares line:

In [27]:
```
# create a DataFrame with the minimum and maximum values of TV
X_new = pd.DataFrame({'TV': [data.TV.min(), data.TV.max()]})
X_new.head()
```

Out[27]:

	TV
0	0.7
1	296.4

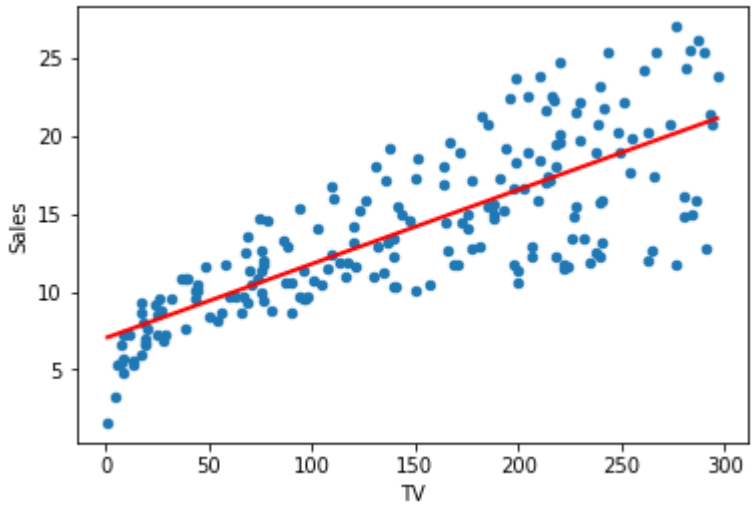
In [28]:
```
# make predictions for those x values and store them
preds = lm.predict(X_new)
preds
```

Out[28]:
```
array([  7.0658692 ,  21.12245377])
```

In [29]:
```
# first, plot the observed data
data.plot(kind='scatter', x='TV', y='Sales')

# then, plot the least squares line
plt.plot(X_new, preds, c='red', linewidth=2)
```

Out[29]:
```
[<matplotlib.lines.Line2D at 0x7f37b47945c0>]
```



Confidence in our Model

Question: Is linear regression a high bias/low variance model, or a low bias/high variance model?

Answer: It's a High bias/low variance model. Under repeated sampling, the line will stay roughly in the same place (low variance), but the average of those models won't do a great job capturing the true relationship (high bias). Note that low variance is a useful characteristic when you don't have a lot of training data!

A closely related concept is **confidence intervals**. Statsmodels calculates 95% confidence intervals for our model coefficients, which are interpreted as follows: If the population from which this sample was drawn was **sampled 100 times**, approximately **95 of those confidence intervals** would contain the "true" coefficient.

In [30]:
```
import statsmodels.formula.api as smf
lm = smf.ols(formula='Sales ~ TV', data=data).fit()
lm.conf_int()
```

Out[30]:

	0	1
Intercept	6.129719	7.935468
TV	0.042231	0.052843

Keep in mind that we only have a **single sample of data**, and not the **entire population of data**. The "true" coefficient is either within this interval or it isn't, but there's no way to actually know. We estimate the coefficient with the data we do have, and we show uncertainty about that estimate by giving a range that the coefficient is **probably** within.

Note that using 95% confidence intervals is just a convention. You can create 90% confidence intervals (which will be more narrow), 99% confidence intervals (which will be wider), or whatever intervals you like.

Hypothesis Testing and p-values

Closely related to confidence intervals is **hypothesis testing**. Generally speaking, you start with a **null hypothesis** and an **alternative hypothesis** (that is opposite the null). Then, you check whether the data supports **rejecting the null hypothesis** or **failing to reject the null hypothesis**.

(Note that "failing to reject" the null is not the same as "accepting" the null hypothesis. The alternative hypothesis may indeed be true, except that you just don't have enough data to show that.)

As it relates to model coefficients, here is the conventional hypothesis test:

- **null hypothesis:** There is no relationship between TV ads and Sales (and thus β_1 equals zero)
- **alternative hypothesis:** There is a relationship between TV ads and Sales (and thus β_1 is not equal to zero)

How do we test this hypothesis? Intuitively, we reject the null (and thus believe the alternative) if the 95% confidence interval **does not include zero**. Conversely, the **p-value** represents the probability that the coefficient is actually zero:

In [31]:
```
# print the p-values for the model coefficients
lm.pvalues
```

Out[31]:
```
Intercept    1.406300e-35
TV           1.467390e-42
dtype: float64
```

If the 95% confidence interval **includes zero**, the p-value for that coefficient will be **greater than 0.05**. If the 95% confidence interval **does not include zero**, the p-value will be **less than 0.05**. Thus, a p-value less than 0.05 is one way to decide whether there is likely a relationship between the feature and the response. (Again, using 0.05 as the cutoff is just a convention.)

In this case, the p-value for TV is far less than 0.05, and so we **believe** that there is a relationship between TV ads and Sales.

Note that we generally ignore the p-value for the intercept.

How Well Does the Model Fit the data?

The most common way to evaluate the overall fit of a linear model is by the **R-squared** value. R-squared is the **proportion of variance explained**, meaning the proportion of variance in the observed data that is explained by the model, or the reduction in error over the **null model**. (The null model just predicts the mean of the observed response, and thus it has an intercept and no slope.)

R-squared is between 0 and 1, and higher is better because it means that more variance is explained by the model. Here's an example of what R-squared "looks like":

You can see that the **blue line** explains some of the variance in the data (R-squared=0.54), the **green line** explains more of the variance (R-squared=0.64), and the **red line** fits the training data even further (R-squared=0.66). (Does the red line look like it's overfitting?)

Let's calculate the R-squared value for our simple linear model:

In [32]:
```
# print the R-squared value for the model
lm.rsquared
```

Out[32]:
```
0.61187505085007099
```

Is that a "good" R-squared value? It's hard to say. The threshold for a good R-squared value depends widely on the domain. Therefore, it's most useful as a tool for **comparing different models**.

Multiple Linear Regression

Simple linear regression can easily be extended to include multiple features. This is called **multiple linear regression**:

$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$

Each x represents a different feature, and each feature has its own coefficient. In this case:

$y = \beta_0 + \beta_1 \times TV + \beta_2 \times Radio + \beta_3 \times Newspaper$

Let's use Statsmodels to estimate these coefficients:

In [34]:
```
# create X and y
feature_cols = ['TV', 'Radio', 'Newspaper']
X = data[feature_cols]
y = data.Sales

lm = LinearRegression()
lm.fit(X, y)

# print intercept and coefficients
print(lm.intercept_)
print(lm.coef_)
```

```
2.93888936946
[ 0.04576465  0.18853002 -0.00103749]
```

How do we interpret these coefficients? For a given amount of Radio and Newspaper ad spending, an **increase of $1000 in TV ad spending** is associated with an **increase in Sales of 45.765 widgets**.

A lot of the information we have been reviewing piece-by-piece is available in the model summary output:

In [35]:
```
lm = smf.ols(formula='Sales ~ TV + Radio + Newspaper', data=data).fit()
lm.conf_int()
lm.summary()
```

Out[35]:

OLS Regression Results

Dep. Variable:	Sales	**R-squared:**	0.897
Model:	OLS	**Adj. R-squared:**	0.896
Method:	Least Squares	**F-statistic:**	570.3
Date:	Thu, 07 Sep 2017	**Prob (F-statistic):**	1.58e-96
Time:	09:23:48	**Log-Likelihood:**	-386.18
No. Observations:	200	**AIC:**	780.4
Df Residuals:	196	**BIC:**	793.6
Df Model:	3		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	2.9389	0.312	9.422	0.000	2.324	3.554
TV	0.0458	0.001	32.809	0.000	0.043	0.049
Radio	0.1885	0.009	21.893	0.000	0.172	0.206
Newspaper	-0.0010	0.006	-0.177	0.860	-0.013	0.011

Omnibus:	60.414	**Durbin-Watson:**	2.084
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	151.241
Skew:	-1.327	**Prob(JB):**	1.44e-33
Kurtosis:	6.332	**Cond. No.**	454.

What are a few key things we learn from this output?

- TV and Radio have significant **p-values**, whereas Newspaper does not. Thus we reject the null hypothesis for TV and Radio (that there is no association between those features and Sales), and fail to reject the null hypothesis for Newspaper.
- TV and Radio ad spending are both **positively associated** with Sales, whereas Newspaper ad spending is **slightly negatively associated** with Sales. (However, this is irrelevant since we have failed to reject the null hypothesis for Newspaper.)
- This model has a higher **R-squared** (0.897) than the previous model, which means that this model provides a better fit to the data than a model that only includes TV.

Feature Selection

How do I decide **what features has to be included** in a linear model? Here's one idea:

- Try different models, and only keep predictors in the model if they have small p-values.
- Check whether the R-squared value goes up when you add new predictors.

What are the **drawbacks** in this approach?

- Linear models rely upon a lot of **assumptions** (such as the features being independent), and if those assumptions are violated (which they usually are), R-squared and p-values are less reliable.
- Using a p-value cutoff of 0.05 means that if you add 100 predictors to a model that are **pure noise**, 5 of them (on average) will still be counted as significant.
- R-squared is susceptible to **overfitting**, and thus there is no guarantee that a model with a high R-squared value will generalize. Below is an example:

In [36]:
```
# only include TV and Radio in the model
lm = smf.ols(formula='Sales ~ TV + Radio', data=data).fit()
lm.rsquared
```

Out[36]:
```
0.89719426108289557
```

In [37]:
```
# add Newspaper to the model (which we believe has no association with Sales)
lm = smf.ols(formula='Sales ~ TV + Radio + Newspaper', data=data).fit()
lm.rsquared
```

Out[37]:
```
0.89721063817895219
```

R-squared will always increase as you add more features to the model, even if they are unrelated to the response. Thus, selecting the model with the highest R-squared is not a reliable approach for choosing the best linear model.

There is alternative to R-squared called **adjusted R-squared** that penalizes model complexity (to control for overfitting), but it generally under-penalizes complexity (http://scott.fortmann-roe.com/docs/MeasuringError.html).

So is there a better approach to feature selection? **Cross-validation.** It provides a more reliable estimate of out-of-sample error, and thus is a better way to choose which of your models will best **generalize** to out-of-sample data. There is extensive functionality for cross-validation in scikit-learn, including automated methods for searching different sets of parameters and different models. Importantly, cross-validation can be applied to any model, whereas the methods described above only apply to linear models.

Handling Categorical Predictors with Two Categories

Up until now, all the predictors have been numeric. What if one of the predictors was categorical?

Let's create a new feature called **Size**, and randomly assign observations to be **small or large**:

In [38]:
```
import numpy as np

# set a seed for reproducibility
np.random.seed(12345)

# create a Series of booleans in which roughly half are True
nums = np.random.rand(len(data))
mask_large = nums > 0.5

# initially set Size to small, then change roughly half to be large
data['Size'] = 'small'
data.loc[mask_large, 'Size'] = 'large'
data.head()
```

Out[38]:

	TV	Radio	Newspaper	Sales	Size
1	230.1	37.8	69.2	22.1	large
2	44.5	39.3	45.1	10.4	small
3	17.2	45.9	69.3	9.3	small
4	151.5	41.3	58.5	18.5	small
5	180.8	10.8	58.4	12.9	large

For scikit-learn, we need to represent all data **numerically**. If the feature only has two categories, we can simply create a **dummy variable** that represents the categories as a binary value:

In [39]:
```
# create a new Series called IsLarge
data['IsLarge'] = data.Size.map({'small':0, 'large':1})
data.head()
```

Out[39]:

	TV	Radio	Newspaper	Sales	Size	IsLarge
1	230.1	37.8	69.2	22.1	large	1
2	44.5	39.3	45.1	10.4	small	0
3	17.2	45.9	69.3	9.3	small	0
4	151.5	41.3	58.5	18.5	small	0
5	180.8	10.8	58.4	12.9	large	1

Let's redo the multiple linear regression and include the **IsLarge** predictor:

In [40]:
```
# create X and y
feature_cols = ['TV', 'Radio', 'Newspaper', 'IsLarge']
X = data[feature_cols]
y = data.Sales

# instantiate, fit
lm = LinearRegression()
lm.fit(X, y)

# print coefficients
zip(feature_cols, lm.coef_)
```

Out[40]:
```
<zip at 0x7f37b3ef8cc8>
```

How do we interpret the **IsLarge coefficient**? For a given amount of TV/Radio/Newspaper ad spending, being a large market is associated with an average **increase** in Sales of 57.42 widgets (as compared to a Small market, which is called the **baseline level**).

What if we had reversed the 0/1 coding and created the feature 'IsSmall' instead? The coefficient would be the same, except it would be **negative instead of positive**. As such, your choice of category for the baseline does not matter, all that changes is your **interpretation** of the coefficient.

Handling Categorical Predictors with More than Two Categories

Let's create a new feature called **Area**, and randomly assign observations to be **rural, suburban, or urban**:

In [41]:
```
# set a seed for reproducibility
np.random.seed(123456)

# assign roughly one third of observations to each group
nums = np.random.rand(len(data))
mask_suburban = (nums > 0.33) & (nums < 0.66)
mask_urban = nums > 0.66
data['Area'] = 'rural'
data.loc[mask_suburban, 'Area'] = 'suburban'
data.loc[mask_urban, 'Area'] = 'urban'
data.head()
```

Out[41]:

	TV	Radio	Newspaper	Sales	Size	IsLarge	Area
1	230.1	37.8	69.2	22.1	large	1	rural
2	44.5	39.3	45.1	10.4	small	0	urban
3	17.2	45.9	69.3	9.3	small	0	rural
4	151.5	41.3	58.5	18.5	small	0	urban
5	180.8	10.8	58.4	12.9	large	1	suburban

We have to represent Area numerically, but we can't simply code it as 0=rural, 1=suburban,

2=urban because that would imply an **ordered relationship** between suburban and urban (and thus urban is somehow "twice" the suburban category).

Instead, we create **another dummy variable**:

In [42]:
```
# create three dummy variables using get_dummies, then exclude the first dummy co
area_dummies = pd.get_dummies(data.Area, prefix='Area').iloc[:, 1:]

# concatenate the dummy variable columns onto the original DataFrame (axis=0 means
data = pd.concat([data, area_dummies], axis=1)
data.head()
```

Out[42]:

	TV	Radio	Newspaper	Sales	Size	IsLarge	Area	Area_suburban	Area_urban
1	230.1	37.8	69.2	22.1	large	1	rural	0	0
2	44.5	39.3	45.1	10.4	small	0	urban	0	1
3	17.2	45.9	69.3	9.3	small	0	rural	0	0
4	151.5	41.3	58.5	18.5	small	0	urban	0	1
5	180.8	10.8	58.4	12.9	large	1	suburban	1	0

Here is how we interpret the coding:

- **rural** is coded as Area_suburban=0 and Area_urban=0
- **suburban** is coded as Area_suburban=1 and Area_urban=0
- **urban** is coded as Area_suburban=0 and Area_urban=1

Why do we only need **two dummy variables, not three?** Because two dummies captures all of the information about the Area feature, and implicitly defines rural as the baseline level. (In general, if you have a categorical feature with k levels, you create k-1 dummy variables.)

If this is confusing, think about why we only needed one dummy variable for Size (IsLarge), not two dummy variables (IsSmall and IsLarge).

Let's include the two new dummy variables in the model:

In [44]:
```
# create X and y
feature_cols = ['TV', 'Radio', 'Newspaper', 'IsLarge', 'Area_suburban', 'Area_urba
X = data[feature_cols]
y = data.Sales

# instantiate, fit
lm = LinearRegression()
lm.fit(X, y)

# print coefficients
print(feature_cols, lm.coef_)
```

```
['TV', 'Radio', 'Newspaper', 'IsLarge', 'Area_suburban', 'Area_urban'] [ 0.0457
4401  0.1878667  -0.0010877   0.07739661 -0.10656299  0.26813802]
```

How do we interpret the coefficients?

- Holding all other variables fixed, being a **suburban** area is associated with an average **decrease** in Sales of 106.56 widgets (as compared to the baseline level, which is rural).
- Being an **urban** area is associated with an average **increase** in Sales of 268.13 widgets (as compared to rural).

A final note about dummy encoding: If you have categories that can be ranked (i.e., strongly disagree, disagree, neutral, agree, strongly agree), you can potentially use a single dummy variable and represent the categories numerically (such as 1, 2, 3, 4, 5).

Assumptions of Linear Regression

Linear regression is an analysis that assesses whether one or more predictor variables explain the dependent (criterion) variable. The regression has five key assumptions:

- Linear relationship
- Multivariate normality
- No or little multicollinearity
- No auto-correlation
- Homoscedasticity

A note about sample size. In Linear regression the sample size rule of thumb is that the Regression analysis requires at least 20 cases per independent variable in the analysis.

Primarily, linear regression needs the relationship between the independent and dependent variables to be linear. It is also important to check for outliers since linear regression is sensitive to outlier effects. The linearity assumption can best be tested with scatter plots, the following two examples depict two cases, where no and little linearity is present.

Secondly, the linear regression analysis requires all variables to be multivariate normal. This assumption can best be checked with a histogram or a Q-Q-Plot. Normality can be checked with a goodness of fit test, e.g., the Kolmogorov-Smirnov test. When the data is not normally distributed a non-linear transformation (e.g., log-transformation) might fix this issue.

Thirdly, linear regression assumes that there is little or no multicollinearity in the data. Multicollinearity occurs when the independent variables are too highly correlated with each other. Multicollinearity may be tested with three central criteria:

1. Correlation matrix – When computing the matrix of Pearson's Bivariate Correlation among all independent variables the correlation coefficients need to be smaller than 1.
2. Tolerance – The tolerance measures the influence of one independent variable on all other independent variables; the tolerance is calculated with an initial linear regression analysis. Tolerance is defined as $T = 1 – R^2$ for these first step regression analysis. With $T < 0.1$ there might be multicollinearity in the data and with $T < 0.01$ there certainly is.
3. Variance Inflation Factor (VIF) – the variance inflation factor of the linear regression is defined as $VIF = 1/T$. With $VIF > 10$ there is an indication that multicollinearity may be present; with $VIF > 100$ there is certainly multicollinearity among the variables.

If multicollinearity is found in the data, centering the data (that is deducting the mean of the variable from each score) might help in solving the problem. However, the simplest way to address the problem is to remove independent variables with high VIF values.

Fourth, linear regression analysis requires that there is little or no autocorrelation in the data. Autocorrelation occurs when the residuals are not independent from each other. For instance, this typically occurs in stock prices, where the price is not independent from the previous price.

4) Condition Index – The condition index is calculated using a factor analysis on the independent variables. Values of 10-30 indicate a mediocre multicollinearity in the linear regression variables, values > 30 indicate strong multicollinearity.

If multicollinearity is found in the data centering the data, that is deducting the mean score might help solve the problem. Other alternatives to tackle the problem is conducting a factor analysis and rotating the factors to insure independence of the factors in the linear regression analysis.

Fourthly, linear regression analysis requires that there is little or no autocorrelation in the data. Autocorrelation occurs when the residuals are not independent from each other. In other words when the value of y(x+1) is not independent from the value of y(x).

The last assumption of the linear regression analysis is homoscedasticity. The scatter plot is a good way to check whether the data is homoscedastic (meaning the residuals are equal across the regression line). The following scatter plots show examples of data that are not homoscedastic (i.e., heteroscedastic):

The Goldfeld-Quandt Test can also be used to test heteroscedasticity. The test splits the data into two groups and tests to see if the variances of the residuals are similar across the groups. If homoscedasticity is present, a non-linear correction might fix the problem.

In [ ]:

Logistic Regression

Researchers are often interested in setting up a model to analyze the relationship between predictors (i.e., independent variables) and it's corresponsing 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 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 individual 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 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 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 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 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, rather it tells 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_{in}.

$$\mathbf{w}_h = \underset{\mathbf{w}}{\operatorname{argmin}} \; E_{\text{in}}(\mathbf{w})$$

The error measure will be used 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: $\underset{h}{\operatorname{argmax}} P(h \mid \mathbf{x})$. The likelihood approach is looking for the hypothesis that makes the data most probable: $\underset{h}{\operatorname{argmax}} P(\mathbf{x} \mid h)$.

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.

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

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

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

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

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

Now maximizing that is the same as minimizing its negative:

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

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

logs:

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

Expanding the logistic function,

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

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:

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

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 sophisticted, closed-form solution like least-squares linear, 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 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 η 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 which describes 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]:

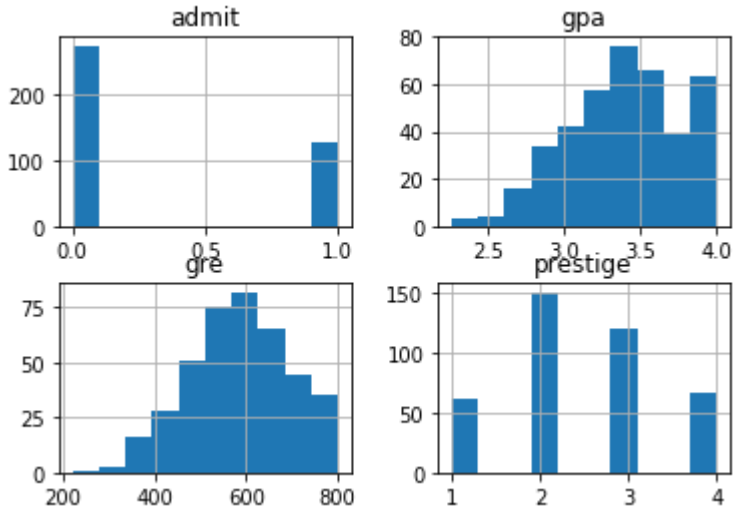
	admit	gre	gpa	prestige
count	400.000000	400.000000	400.000000	400.00000
mean	0.317500	587.700000	3.389900	2.48500
std	0.466087	115.516536	0.380567	0.94446
min	0.000000	220.000000	2.260000	1.00000
25%	0.000000	520.000000	3.130000	2.00000
50%	0.000000	580.000000	3.395000	2.00000
75%	1.000000	660.000000	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 can be 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 with 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 would use requires intercepts/constants to be 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: Deprecat
ionWarning: This module was deprecated in version 0.18 in favor of the model_se
lection module into which all the refactored classes and functions are moved. A
lso note that the interface of the new CV iterators are different 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	occupation	o
affair								
0	4.329701	28.390679	7.989335	1.238813	2.504521	14.322977	3.405286	3
1	3.647345	30.537019	11.152460	1.728933	2.261568	13.972236	3.463712	3

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	occupat
rate_marriage							
1.0	33.823232	13.914141	2.308081	2.343434	13.848485	3.232323	3.83838
2.0	30.471264	10.727011	1.735632	2.330460	13.864943	3.327586	3.76436
3.0	30.008056	10.239174	1.638469	2.308157	14.001007	3.402820	3.79859
4.0	28.856601	8.816905	1.369536	2.400981	14.144514	3.420161	3.83586
5.0	28.574702	8.311662	1.252794	2.506334	14.399776	3.454918	3.89269

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

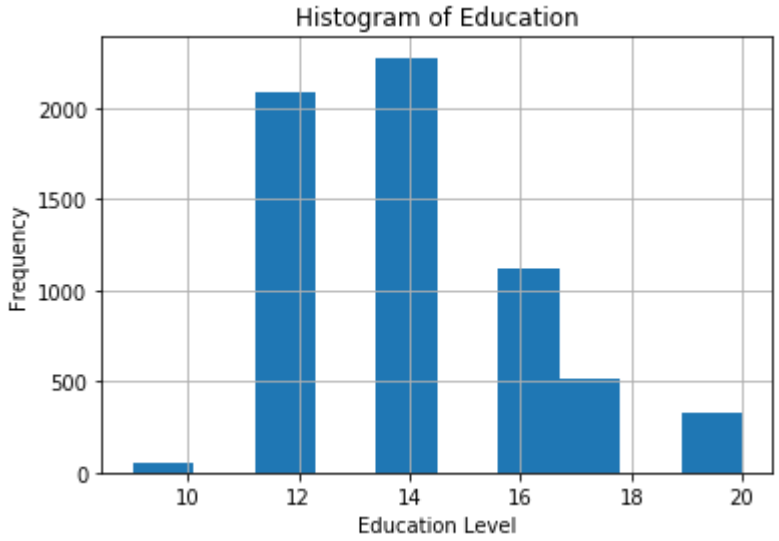
Data Visualization

In [29]:
```
# show plots in the notebook
%matplotlib inline
```

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

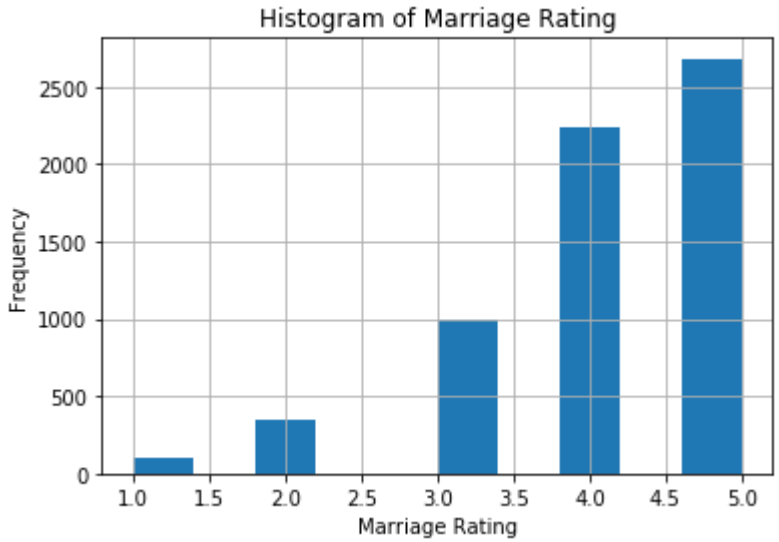
In [30]:
```
# histogram of education
dta.educ.hist()
plt.title('Histogram of Education')
plt.xlabel('Education Level')
plt.ylabel('Frequency')
```

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')
```

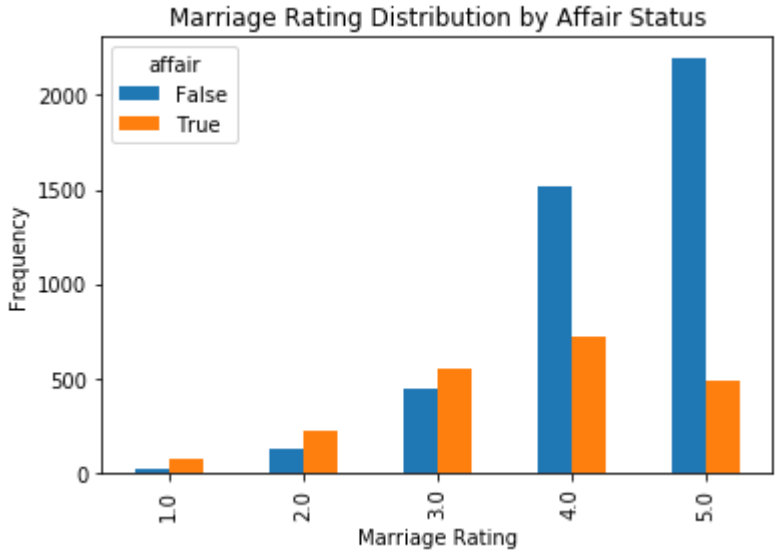
Out[31]: <matplotlib.text.Text at 0x7f0da7a6d160>



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')
```

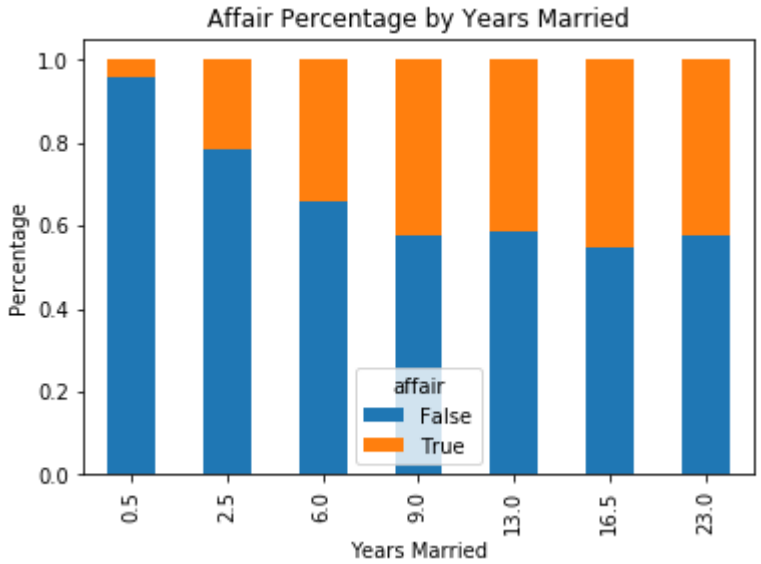
Out[32]: <matplotlib.text.Text at 0x7f0da7b59198>



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=
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_husb_2',
        'occ_husb_3', 'occ_husb_4', 'occ_husb_5', 'occ_husb_6', 'rate_marriag
e',
        '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, 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_st
model2 = LogisticRegression()
model2.fit(X_train, y_train)
```

Out[42]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          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 from college, has been married for 3 years. She 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%.