PYTHON MACHINE LEARNING

from **Learning Python for Data Analysis and Visualization** by Jose Portilla
https://www.udemy.com/learning-python-for-data-analysis-and-visualization/
Notes by Michael Brothers

Companion to the file *Python for Data Analysis*.

Table of Contents

What is Machine Learning? 3

Types of Machine Learning – Supervised & Unsupervised 3
Supervised Learning 3
Supervised Learning: Regression 3
Supervised Learning: Classification 3
Unsupervised Learning 3

Supervised Learning – LINEAR REGRESSION 4
Getting & Setting Up the Data 4
Quick visualization of the data: 4
Root Mean Square Error 6
Using SciKit Learn to perform multivariate regressions 6
Building Training and Validation Sets using `train_test_split` 7
Predicting Prices 7
Residual Plots 8

Supervised Learning – LOGISTIC REGRESSION 9
Getting & Setting Up the Data 9
Binary Classification using the Logistic Function 9
Dataset Analysis 9
Data Preparation 10
Multicollinearity Consideration 11
Testing and Training Data Sets 11
For more info on Logistic Regression: 12

Supervised Learning – MULTI-CLASS CLASSIFICATION 12
The Iris Flower Data Set 12
Getting & Setting Up the Data 13
Data Visualization 13
Plotting individual histograms: 14
Multi-Class Classification with Sci Kit Learn 14
K-Nearest Neighbors 14

SUPPORT VECTOR MACHINES 16

Supervised Learning using NAÏVE BAYES CLASSIFIERS 19
Bayes' Theorem 19
Naïve Bayes Equation 19
Constructing a classifier from the probability model 19
Gaussian Naïve Bayes 19
For more info on Naïve Bayes: 20

DECISION TREES and RANDOM FORESTS 20
Visualization Function 21
Random Forests 22
Random Forest Regression 23

More resources for Random Forests: 24
Unsupervised Learning – NATURAL LANGUAGE PROCESSING 25
Exploratory Data Analysis (EDA) 25
Feature Engineering 25
Text Pre-processing 26
Vectorization 26
Term Frequency – Inverse Document Frequency (TF-IDF) 27
Training a Model 27
APPENDIX I – SciKit Learn Boston Dataset: 28
APPENDIX II: FOR FURTHER RESEARCH 29

PYTHON MACHINE LEARNING WITH SCIKIT LEARN

ADDITIONAL FREE RESOURCES:

1.) SciKit Learn's own documentation and basic tutorial: SciKit Learn Tutorial
2.) Nice Introduction Overview from Toptal
3.) This free online book by Stanford professor Nils J. Nilsson.
4.) Andrew Ng's Machine Learning Class
 notes
 Coursera Video

What is Machine Learning?

A machine learning program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**.

- We start with data, which we call experience E
- We decide to perform some sort of task or analysis, which we call T
- We then use some validation measure to test our accuracy, which we call performance measure P (determined by splitting up our data set into a training set followed by a testing set to validate the accuracy)

Types of Machine Learning – Supervised & Unsupervised

Supervised Learning

We have a dataset consisting of both features and labels. The task is to construct an estimator which is able to predict the label of an object given the set of features.

Supervised Learning is divided into two categories:
 - Regression
 - Classification

Supervised Learning: Regression

Given some data, the machine assumes that those values come from some sort of function and attempts to find out what the function is. It tries to fit a mathematical function that describes a curve, such that the curve passes as close as possible to all the data points.
 Example: Predicting house prices based on input data

Supervised Learning: Classification

Classification is discrete, meaning an example belongs to precisely one class, and the set of classes covers the whole possible output space.
 Example: Classifying a tumor as either malignant or benign based on input data

Unsupervised Learning

Here data has no labels, and we are interested in finding similarities between the objects in question. In a sense, unsupervised learning is a means of discovering labels from the data itself.

Supervised Learning – LINEAR REGRESSION

Ultimately we want to minimize the difference between our hypothetical model (theta) and the actual,
in an exercise called Gradient Descent (trial and error with different parameter values).
Note that complex gradient descents may be subject to local minimums.

Batch Gradient Descent – stepwise calculations performed over entire training set (i = 0 to m), repeat until convergence

Stochastic Gradient Descent – for j = 1 to m, perform parameter adjustments to the whole based on iterative calculations. In a sense, calculations meander their way toward the minimum without necessarily hitting it exactly, but get there much faster for large data sets.

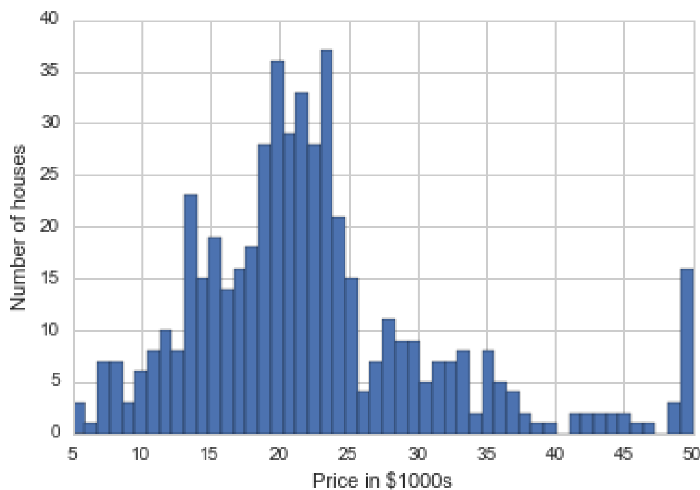
Getting & Setting Up the Data

```
import numpy as np
import pandas as pd
from pandas import Series,DataFrame

import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('whitegrid')
%matplotlib inline

from sklearn.datasets import load_boston
boston = load_boston()
print boston.DESCR
```
provides a detailed description of the 506 Boston dataset records

Quick visualization of the data:

Histogram of prices (this is the target of our dataset)

```
plt.hist(boston.target,bins=50)
plt.xlabel('Price in $1000s')
plt.ylabel('Number of houses')
```
use bins=50, otherwise it defaults to only 10



NOTE: `boston` is NOT a DataFrame. `type(boston)` returns `sklearn.datasets.base.Bunch`

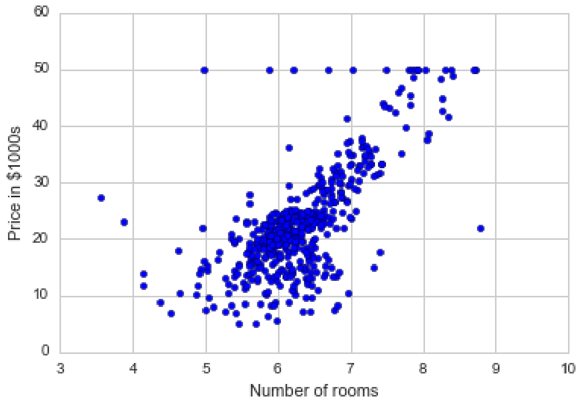
The MEDV (median value of owner-occupied homes in 1000s) column in the data does not appear when cast as a DataFrame – instead, it is accessed using the `.target` method.

Values range from 5.0 to 50.0, with float values in between. Source: 1970 U.S. Census of Population and Housing, Boston Standard Metropolitan Statistical Area (SMSA), section 29, tracts listed in 2 parts.
See http://www.census.gov/prod/www/decennial.html

SO HERE'S MY PROBLEM: all our data is aggregate – we're comparing "average values" in a tract to "average rooms" in a tract, so we're applying the bias that tracts are fairly homogenous. And wouldn’t we want to apply weights to tracts – those with 700 housing units weigh more statistically than those with 70?

Plot the column at the 5 index (Labeled RM)

```
plt.scatter(boston.data[:,5],boston.target)
plt.ylabel('Price in $1000s')
plt.xlabel('Number of rooms')
```



The lecture then builds a DataFrame using features specific to the SciKit boston dataset:

```
boston_df = DataFrame(boston.data)
boston_df.columns = boston.feature_names     to label the columns
boston_df['Price'] = boston.target           adds a column not yet present
boston_df.head()
```

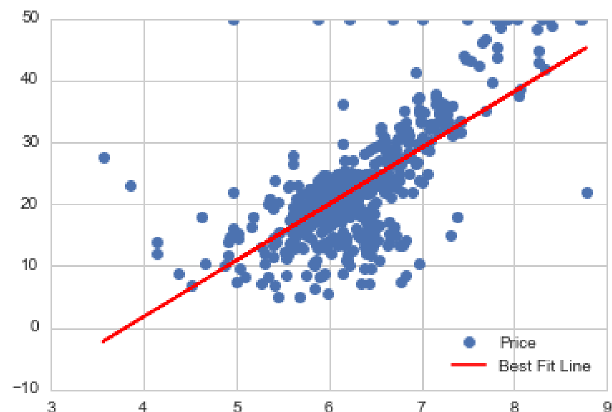
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	Price
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

He then uses Seaborn's lmplot to fit a linear regression:
`sns.lmplot('RM','Price',data = boston_df)`, but it doesn't represent the data well at either extreme.

He explains the math behind the Least Squares Method, then applies numpy to the univariate problem at hand:

```
X = np.vstack(boston_df.RM)                  Use vstack to make X two-dimensional (w/index)
X = np.array([[value,1] for value in X])     pairs each x-value to an attribute number (1)
                                             this feels messy
Y = boston_df.Price                          Set up Y as the target price of the houses.
m, b = np.linalg.lstsq(X, Y)[0]              returns m & b values for the least-squares-fit line
```

```
plt.plot(boston_df.RM,boston_df.Price,'o')   plot with best fit line (entered in one cell)
x = boston_df.RM
plt.plot(x, m*x + b,'r',label='Best Fit Line')
plt.legend(loc='lower right')                unlike Seaborn, pyplot requires a separate legend line
```



Root Mean Square Error

Since we used numpy already, we can obtain the error the same way:

```
result = np.linalg.lstsq(X,Y)
error_total = result[1]
rmse = np.sqrt(error_total/len(X))          this is the root mean square error
print "The root mean square error was %.2f " %rmse
The root mean square error was 6.60
```

Since the root mean square error (RMSE) corresponds approximately to the standard deviation we can now say that the price of a house won't vary more than 2 times the RMSE 95% of the time. Thus we can reasonably expect a house price to be within $13,200 of our line fit.

Using SciKit Learn to perform multivariate regressions

First, import the linear regression library:

```
import sklearn
from sklearn.linear_model import LinearRegression
```

The sklearn.linear_model.LinearRegression class is an estimator. Estimators predict a value based on the observed data. In scikit-learn, all estimators implement the fit() and predict() methods. The former method is used to learn the parameters of a model, and the latter method is used to predict the value of a response variable for an explanatory variable using the learned parameters. It is easy to experiment with different models using scikit-learn because all estimators implement the fit and predict methods.

```
lreg = LinearRegression()     create a Linear Regression object
```

For more info/examples: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Methods available on this type of object are:

`lreg.fit()` which fits a linear model

`lreg.predict()` which is used to predict Y using the linear model with estimated coefficients

`lreg.score()` which returns the *coefficient of determination* (R^2) – a measure of how well observed outcomes are replicated by the model. Values fall between 0 and 1, the higher the better.

We'll start the multi variable regression analysis by seperating our boston dataframe into the data columns and the target columns:

```
X_multi = boston_df.drop('Price',1)       these are our Data Columns
                                          (in order to drop a column you need to pass a 1 index)
Y_target = boston_df.Price                this is our Target Column

lreg.fit(X_multi,Y_target)                Implement the Linear Regression
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
```

Let's go ahead check the intercept and number of coefficients.

```
print 'The estimated intercept coefficient is %.2f' %lreg.intercept_
The estimated intercept coefficient is 36.49
print 'The number of coefficients used was %d' %len(lreg.coef_)
The number of coefficients used was 13
```

`lreg` is now an equation for a line with 13 coefficients.

To see each of these coefficients mapped to their original columns:

```
coeff_df = DataFrame(boston_df.columns)        Set a DataFrame from the Features
coeff_df.columns = ['Features']
```

Set a new column lining up the coefficients from the linear regression

```
coeff_df["Coefficient Estimate"] = pd.Series(lreg.coef_)
coeff_df
```

	Features	Coefficient Estimate
0	CRIM	-0.107171
1	ZN	0.046395
2	INDUS	0.02086
3	CHAS	2.688561
4	NOX	-17.795759
5	RM	3.804752
6	AGE	0.000751
7	DIS	-1.475759
8	RAD	0.305655
9	TAX	-0.012329
10	PTRATIO	-0.953464
11	B	0.009393
12	LSTAT	-0.525467
13	Price	NaN

For more info on interpreting coefficients:
http://www.theanalysisfactor.com/interpreting-regression-coefficients/

SciKit Learn's built-in methods of best feature selection:
http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_regression.html

Jose claims that the highest correlated feature was # of rooms (RM) with a coefficient estimate of 3.8. I see NOX as the highest with a coefficient of -17.79. Related question: how much does the coefficient affect the target value if the variable doesn't change much? ie, a low coefficient on # rooms may have greater effect when rooms can double from 2 to 4 quite easily, where a high coefficient on NOX may not matter much if the variation over our sample set is only 1 or 2 ppm. And what about orders of magnitude? A small change to a big number may outweigh a big change to a small one. What about non-linear relationships? The number of rooms may have diminishing marginal utility.

Building Training and Validation Sets using `train_test_split`

SciKit Learn has a built-in tool for randomly selecting samples from a dataset for training and testing purposes:

```
X_train, X_test, Y_train, Y_test =
                sklearn.cross_validation.train_test_split(X,boston_df.Price)
print X_train.shape, X_test.shape, Y_train.shape, Y_test.shape
(379L, 2L) (127L, 2L) (379L,) (127L,)
```

¾ of the original dataset are allocated to train, ¼ to test

Predicting Prices

```
lreg = LinearRegression()
```

Once again do a linear regression, except only on the training sets this time

```
lreg.fit(X_train,Y_train)
```

Now run predictions on both the X training and testing sets

```
pred_train = lreg.predict(X_train)
pred_test = lreg.predict(X_test)
```

Now obtain the mean square error (these values change with each new train_test_split run)

```
print "Fit a model X_train, and calculate MSE with Y_train: %.2f"
                      % np.mean((Y_train - pred_train) ** 2)
print "Fit a model X_train, and calculate MSE with X_test and Y_test: %.2f"
                      %np.mean((Y_test - pred_test) ** 2)
Fit a model X_train, and calculate MSE with Y_train: 42.95
Fit a model X_train, and calculate MSE with X_test and Y_test: 46.34
```

It looks like our mean square error between our training and testing was pretty close.
But how do we actually visualize this?

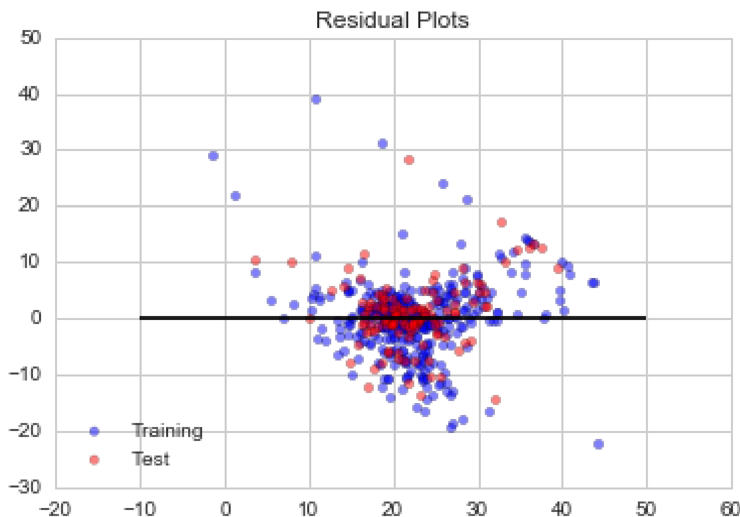
Residual Plots

In regression analysis, the difference between the observed value of the dependent variable (y) and the predicted value ($\hat{y}$) is called the residual (e). Each data point has one residual, so that:

$$Residual = Observed\ value - Predicted\ value$$

You can think of these residuals in the same way as the D value we discussed earlier, in this case however, there were multiple data points considered.
A residual plot is a graph that shows the residuals on the vertical axis and the independent variable on the horizontal axis. If the points in a residual plot are randomly dispersed around the horizontal axis, a linear regression model is appropriate for the data; otherwise, a non-linear model is more appropriate.
Residual plots are a good way to visualize the errors in your data. If you have done a good job then your data should be randomly scattered around line zero. If there is some strucutre or pattern, that means your model is not capturing some thing. There could be an interaction between 2 variables that you're not considering, or may be you are measuring time dependent data. If this is the case go back to your model and check your data set closely.
So now let's go ahead and create the residual plot. For more info on the residual plots check out this great link.

Scatter plot the training data
```
train = plt.scatter(pred_train,(Y_train - pred_train),c='b',alpha=0.5)
```
Scatter plot the testing data
```
test = plt.scatter(pred_test,(Y_test - pred_test),c='r',alpha=0.5)
```
Plot a horizontal axis line at 0
```
plt.hlines(y=0,xmin=-10,xmax=50)
```
Add Labels
```
plt.legend((train,test),('Training','Test'),loc='lower left')
plt.title('Residual Plots')
```



Great! Looks like there aren't any major patterns to be concerned about, (though it may be interesting to check out the line occurring towards the upper right), but overall the majority of the residuals seem to be randomly allocated above and below the horizontal.
NOTE: the line upper right relates to the outlier 50 values from the dataset (same disbursement of 11 values).
For more info: http://scikit-learn.org/stable/modules/linear_model.html#linear-model

Supervised Learning – LOGISTIC REGRESSION

Getting & Setting Up the Data

```
import numpy as np
import pandas as pd
from pandas import Series,DataFrame
import math        this is just to see the logistic function
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('whitegrid')
%matplotlib inline
```

Machine Learning Imports

```
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import train_test_split
```

For evaluating our ML results

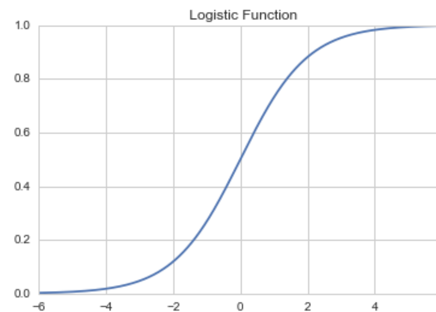
```
from sklearn import metrics
```

Dataset Import

```
import statsmodels.api as sm
```

Binary Classification using the Logistic Function

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



The Logistic Function takes any value from negative to positive infinity and it has always has an output between 0 and 1. Refer to the jupyter notebook for code behind the plot above.

Essentially we're applying a linear regression equation to the logistic function. The goal is to return a probability of "success" or "failure" from our linear regression equation. Since the logistic function outputs a value between 0 and 1, we now have a binary classification between outputs from 0 to 0.5 (failure), and 0.5 to 1 (success).

For more info: Wikipedia, Andrew Ng's Lecture Notes

Dataset Analysis

The dataset is packaged within Statsmodels. It is a data set from a 1974 survey of women by Redbook magazine. Married women were asked if they have had extramarital affairs. The published work on the data set can be found in: Fair, Ray. 1978. "A Theory of Extramarital Affairs," Journal of Political Economy, February, 45-61.

Given certain variables for each woman, can we classify them as either having particpated in an affair, or not participated in an affair?

Standard method of loading Statsmodels datasets into a pandas DataFrame:

Note the name fair stands for the 'affair' dataset.

```
df = sm.datasets.fair.load_pandas().data
```

Now we add a column to hold the binary value "Had Affair":

```
def affair_check(x):
    if x != 0:
        return 1
    else:
        return 0
df['Had_Affair'] = df['affairs'].apply(affair_check)
```

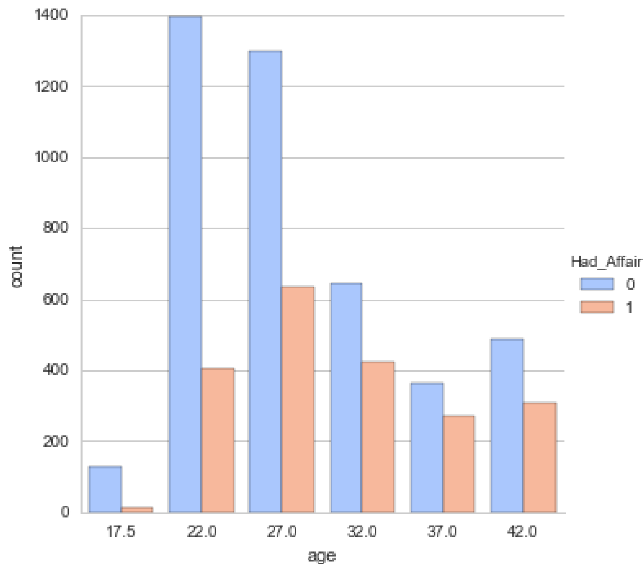
Take a quick look at the Had_Affair column and mean values of all other attributes:

```
df.groupby('Had_Affair').mean()
```

Most of the values are fairly close to one another. There are no obvious correlations between a given parameter and the likelihood of participating in an affair.

The lecture then forms a series of factorplots on various individual parameters:

```
sns.factorplot('age',data=df,hue='Had_Affair',palette='coolwarm');
```



Data Preparation

Most of the columns in our dataset contain parametric data (age, level of education, degree of religiousness, etc.) while Occupation does not. Occupation and Husban's Occupation contain Categorical Variables. We need to apply the pandas `get_dummies` method to split each occupational category into its own column:

Create new DataFrames for the Categorical Variables

```
occ_dummies = pd.get_dummies(df['occupation'])
hus_occ_dummies = pd.get_dummies(df['occupation_husb'])
```

This creates dataframes with rows for each original record, and columns 1.0 to 6.0 for each categorical occupation. (For some reason pandas converted integer values to floats.) Cells are now contain either 1 or 0.

```
occ_dummies.columns = ['occ1','occ2','occ3','occ4','occ5','occ6']
hus_occ_dummies.columns = ['hocc1','hocc2','hocc3','hocc4','hocc5','hocc6']
```

This is just to rename the columns to something more recognizable

Drop the original columns (and the target) and load the new dataframes onto our dataset

```
X = df.drop(['occupation','occupation_husb','Had_Affair'],axis=1)
X = pd.concat([X, occ_dummies, hus_occ_dummies],axis=1)
```

Note: in the lecture, Jose first combined occ_dummies & hus_occ_dummies into a "dummies" dataframe and joined that into X using concat. I chose to do it in one step.

Set up the target data

```
Y = df.Had_Affair
```

Multicollinearity Consideration

Our six dummy occupation categories are highly correlated. Among the six only one will contain a "1" value, so you can always determine the value of one column based on the values of the other five. This will lead to an exaggerated level of accuracy in the regression calculation. To compensate, we drop a column of data, and sacrifice one data point in favor of more realistic regression calculations. While the choice of column is fairly arbitrary, it does affect the final result.
For more info see: https://en.wikipedia.org/wiki/Multicollinearity

Drop one column of each dummy variable set to avoid multicollinearity

```
X = X.drop('occ1',axis=1)
X = X.drop('hocc1',axis=1)
```

Drop the affairs column so Y target makes sense

```
X = X.drop('affairs',axis=1)
```

In order to use the Y with SciKit Learn, we need to set it as a 1-D array. This means we need to "flatten" the array.
Numpy has a built in method for this called ravel:

```
Y = np.ravel(Y)
```

NOTE: Y was a Series to begin with, so `np.array(Y)` *does the same thing!*

Running the Logistic Regression with SciKit Learn

```
log_model = LogisticRegression()        initiate the LogisticRegression model
log_model.fit(X,Y)                      fit our data to the model
log_model.score(X,Y)                    check our accuracy
0.7260446120012567
```

This indicates a 73% accuracy rating.
Compare this to the "null error rate" (simply 1 minus the Y target average):

```
Y.mean()
0.32249450204209867
```

Just guessing "no affair" will be right 68% of the time. Our model doesn't do much better.

Check the coefficients:

```
coeff_df = DataFrame(zip(X.columns, np.transpose(log_model.coef_)))
```

(Refer to the jupyter notebook)
A positive coeffecient corresponds to increasing the likelihood of having an affair while a negative coefficient corresponds to a decreased likelihood of having an affair as the actual data value point increases
As you might expect, an increased marriage rating corresponded to a decrease in the likelihood of having an affair.
Increased religiousness also seems to correspond to a decrease in the likelihood of having an affair.
Since all the dummy variables (the wife and husband occupations) are positive that means the lowest likelihood of having an affair corresponds to the baseline occupation we dropped (1-Student).

Testing and Training Data Sets

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y)
log_model2 = LogisticRegression()       make a new log_model
log_model2.fit(X_train, Y_train)        fit the new model
```

Predict the classes of the testing data set

```
class_predict = log_model2.predict(X_test)
```

Compare the predicted classes to the actual test classes

```
print metrics.accuracy_score(Y_test,class_predict)
0.726130653266            and this is about the same as our previous score
```

For more info on Logistic Regression:
So what could we do to try to further improve our Logistic Regression model?
We could try some regularization techniques or using a non-linear model.
A great post on how to do logistic regression analysis using Statsmodels from yhat!
The SciKit learn Documentation includes several examples at the bottom of the page.
DataRobot has a great overview of Logistic Regression
Fantastic resource from aimotion.blogspot on the Logistic Regression and the Mathmatics of how it relates to the cost function and gradient!

Supervised Learning – MULTI-CLASS CLASSIFICATION
The Iris Flower Data Set
For this series of lectures, we will be using the famous Iris flower data set. The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by Sir Ronald Fisher in the 1936 as an example of discriminant analysis.
The set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor), so 150 total samples. Four features were measured from each sample: the length and the width of the sepals and petals, in cm.

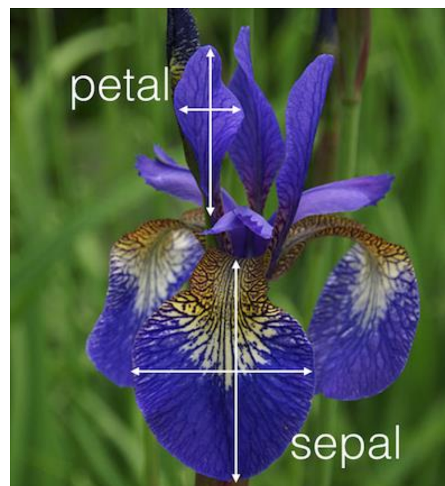
Iris Setosa

Iris Versicolour

Iris Virginica

The three classes in the Iris dataset:
- Iris-setosa (n=50)
- Iris-versicolour (n=50)
- Iris-virginica (n=50)

The four features of the Iris dataset:
- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm



In this section we will learn how to use multi-class classification with SciKit Learn to seperate data into multiple classes.
We will first use SciKit Learn to implement a strategy known as **one vs. all** (sometimes called one vs. rest) to perform multi-class classification. This method works by basically performing a logistic regression for binary classification for each possible class. The class that is then predicted with the highest confidence is assigned to that data point.

For a great visual explanation of this, here is Andrew Ng's quick explanation of how one-vs-rest works:
```
from IPython.display import YouTubeVideo
YouTubeVideo("Zj403m-fjqg")
```

After we use the one-vs-all logistic regression method, we'll use the **k nearest neighbors** method to classify the data.

Getting & Setting Up the Data

```
from sklearn import linear_model
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
Y = iris.target
```

Put into a pandas DataFrame:

```
iris_data = DataFrame(X, columns=['Sepal Length','Sepal Width',
                      'Petal Length','Petal Width'])
iris_target = DataFrame(Y,columns=['Species'])
```

If we look at the iris_target data, we'll notice that the Species are still defined as either 0, 1, or 2.

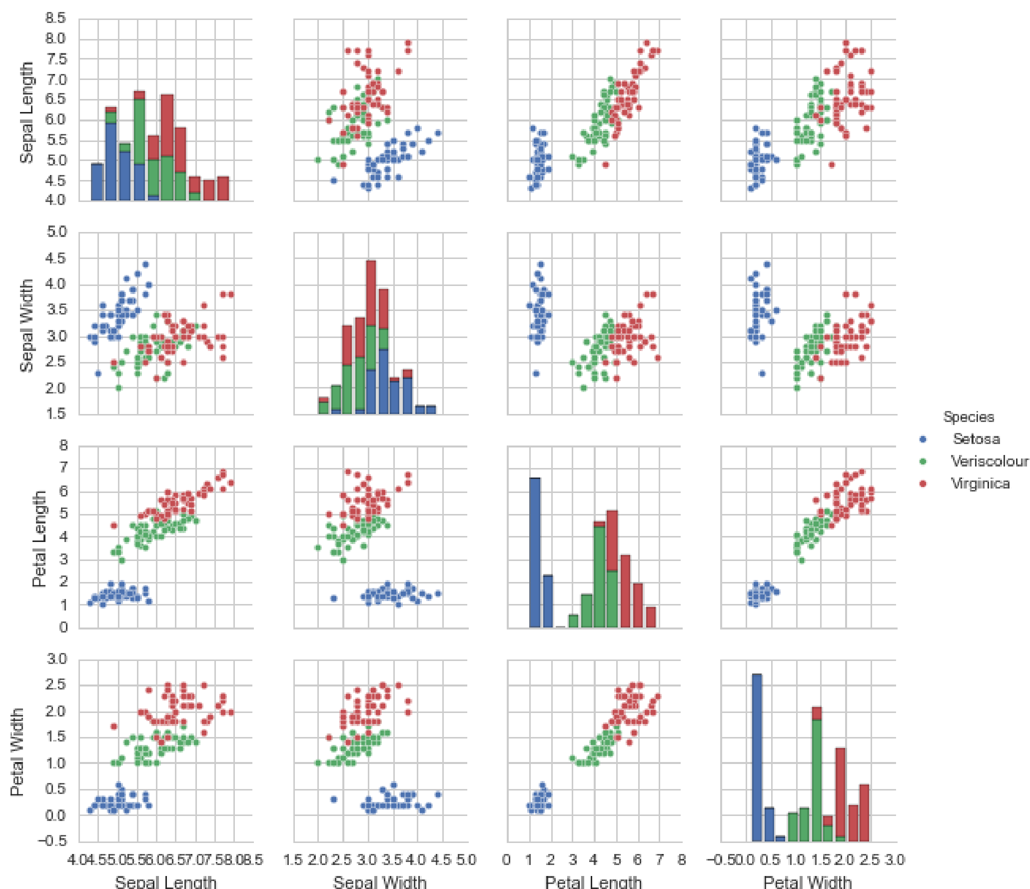
Let's go ahead and use apply() to split the column, apply a naming function, and then combine it back together:

```
def flower(num):
    ''' Takes in a numerical class, returns a flower name'''
    if num == 0:
        return 'Setosa'
    elif num == 1:
        return 'Veriscolour'
    else:
        return 'Virginica'
iris_target['Species'] = iris_target['Species'].apply(flower)
```

Data Visualization

We can get a quick birds eye view with seaborn's pairplot:

```
iris = pd.concat([iris_data, iris_target],axis=1)   first create a combined DataFrame
sns.pairplot(iris, hue='Species', size=2)
```



Note that one class is linearly separable from the other 2; the latter are NOT linearly separable from each other

For our purposes we can ignore the "species vs. species" plots along the diagonal.

Plotting individual histograms:

Use factorplot to view individual parameters (but you need to sort them first):

```
xorder = np.apply_along_axis(sorted, 0, iris['Petal Length'].unique())
sns.factorplot('Petal Length', data=iris, order=xorder, size = 10, hue='Species',
                              kind='count');
```
(see the full plot in the file Python Data Visualizations)

Multi-Class Classification with Sci Kit Learn

We already have X and Y defined as the Data Features and Target so let's go ahead and continue with those arrays. We will then have to split the data into Testing and Training sets. I'll pass a test_size argument to have the testing data be 40% of the total data set. I'll also pass a random seed number.

```
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import train_test_split
logreg = LogisticRegression()
```

Split the data into Trainging and Testing sets

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4,
                              random_state=3)
```

Train the model with the training set

```
logreg.fit(X_train, Y_train)
```

Now that we've trained our model with a training set, let's test our accuracy with the testing set.
We'll make a prediction using our model and then check its accuracy.

Import testing metrics from SciKit Learn

```
from sklearn import metrics                        Import testing metrics from SciKit Learn
Y_pred = logreg.predict(X_test)                    Prediction from X_test
print metrics.accuracy_score(Y_test,Y_pred)        Check accuracy
0.933333333333
```

It looks like our model had a 93% accuracy (this could change from run to run due to the random splitting).
Should we trust this level of accuracy? I encourage you to figure out ways to intuitively understand this result.
Try looking at the PairPlot again and check to see how separate the data features initially were. Also try changing the test_size parameter and check how that affects the outcome. In conclusion, given how clean the data is and how separated some of the features are, we should expect pretty high accuracy.

K-Nearest Neighbors

Now we'll use *k*-nearest neighbors to implement Multi-Class Classification. The premise of this algorithm is simple. Given an object to be assigned to a class in a feature space, select the class that is "nearest" to the neighbors in the training set. This "nearness" is a distance metric, which is usually a Euclidean distance.
The k-nearest neighbor (kNN) algorithm is well explained in the following two videos:
How kNN algorithm works by Thales Sehn Körting
MIT OpenCourseWare 10. Introduction to Learning, Nearest Neighbors lecture by Patrick Winston
To leverage the algorithm the hard way, compute the distance between your object and every neighbor, and make your selection based on the closest ones. The value of k determine how many neighbors are considered. For binary classifications, always choose an odd number to avoid ties.

Using SciKit Learn with k=6:

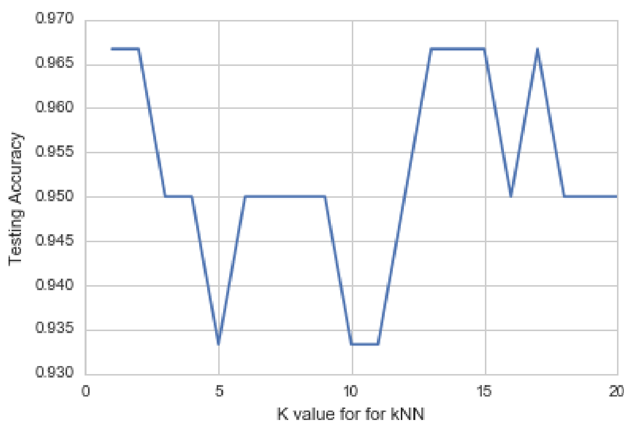
```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 6)     create an instance and set k=6
knn.fit(X_train,Y_train)                        fit our data to the instance
Y_pred = knn.predict(X_test)                    run a prediction
print metrics.accuracy_score(Y_test,Y_pred)     check accuracy
0.95
```

Trying the same thing with k=1:

```
knn = KNeighborsClassifier(n_neighbors = 1)
knn.fit(X_train,Y_train)
Y_pred = knn.predict(X_test)
print metrics.accuracy_score(Y_test,Y_pred)
0.966666666667
```

You can cycle through 20 possible k-values to find the most accurate:

```
k_range = range(1, 21)
accuracy = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, Y_train)
    Y_pred = knn.predict(X_test)
    accuracy.append(metrics.accuracy_score(Y_test, Y_pred))
plt.plot(k_range, accuracy)
plt.xlabel('K value for for kNN')
plt.ylabel('Testing Accuracy');
```



Interesting! Try changing the way Sci Kit Learn split the training and Testing data sets and try re-running this analysis. What changed?

A better classification method is to first divide your feature space based on the existing data (done so by tracing all possible perpendicular bisectors to draw "decision boundaries"), then assign a class to the object by the space it occupies. In some cases, Euclidian distances don't apply, so we use vector angles (a mechanism for comparing parameter ratios)

For more info:
1.) Wikipedia on Multiclass Classification
2.) MIT Lecture Slides on MultiClass Classification
3.) Sci Kit Learn Documentation
4.) DataRobot on Classification Techniques

SUPPORT VECTOR MACHINES

Support Vector Machines (SVM) are a method that uses points in a transformed problem space that best separate classes into two groups. Classification for multiple classes is then supported by a one-vs-all method (just like we previously did for Logistic Regression for Multi Class Classification).

Formal Explanation:

In machine learning, support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. Given a set of training examples, each marked for belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

The advantages of support vector machines are:

Effective in high dimensional spaces.
Still effective in cases where number of dimensions is greater than the number of samples.
Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

If the number of features is much greater than the number of samples, the method is likely to give poor performances.
SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).

Key to the success of Support Vector Machines is computation of a *hyperplane* dividing the classes. If the plane is curved, then a kernel trick may be employed to cast the feature space into 3 dimensions for slicing. Nice video here.

Import numpy and matplotlib (but not Seaborn), and the Iris dataset as seen above.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
Y = iris.target
```

Next, import the SVC (Support Vector Classification) from the SVM library of Sci Kit Learn and set up our model:

```
from sklearn.svm import SVC
model = SVC()
from sklearn.cross_validation import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y)
model.fit(X_train,Y_train)
```

Now test the model:

```
from sklearn import metrics
predicted = model.predict(X_test)
expected = Y_test
print metrics.accuracy_score(expected,predicted)
0.973684210526
```

this number can vary due to the random outcome of train_test_split

Using this basic SVM method we achieved a 97.4% accuracy

Now that we've gone through a basic implementation of SVM lets go ahead and quickly explore the various kernel types we can use for classification. We can do this by plotting out the boundaries created by each kernel type! We'll start with some imports and by setting up the data.
The four methods we will explore are two linear models, a Gaussian Radial Basis Function, and a SVC with a polynomial (3rd Degree) kernel.
The linear models LinearSVC() and SVC(kernel='linear') yield slightly different decision boundaries. This can be a consequence of the following differences:

- LinearSVC minimizes the squared hinge loss while SVC minimizes the regular hinge loss.
- LinearSVC uses the One-vs-All multiclass reduction while SVC uses the One-vs-One multiclass reduction.

Import all SVM

```
from sklearn import svm
```

We'll use all the data and not bother with a split between training and testing. We'll also only use two features.

```
X = iris.data[:,:2]
Y = iris.target
C = 1.0                     SVM regularization parameter
```

SVC with a Linear Kernel (our original example)

```
svc = svm.SVC(kernel='linear', C=C).fit(X, Y)
```

Gaussian Radial Bassis Function

```
rbf_svc = svm.SVC(kernel='rbf', gamma=0.7, C=C).fit(X, Y)
```

SVC with 3rd degree poynomial

```
poly_svc = svm.SVC(kernel='poly', degree=3, C=C).fit(X, Y)
```

SVC Linear

```
lin_svc = svm.LinearSVC(C=C).fit(X,Y)
```

Now that we have fitted the four models, we will go ahead and begin the process of setting up the visual plots.
Note: This example is taken from the Sci Kit Learn Documentation.

First we define a mesh to plot in. We define the max and min of the plot for the y and x axis by the smallest and larget features in the data set. We can use numpy's built in meshgrid method to construct our plot:

```
h = 0.02                          Set the step size
x_min=X[:, 0].min() - 1           Set the X-axis min and max
x_max =X[:, 0].max() + 1
y_min = X[:, 1].min() - 1         Set the Y-axis min and max
y_max = X[:, 1].max() + 1
```

Finally, numpy can create a meshgrid

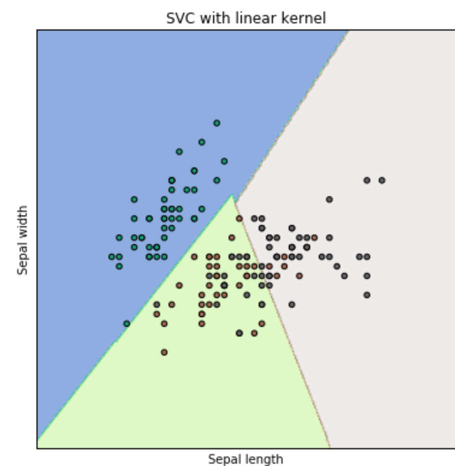
```
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),np.arange(y_min, y_max, h))
```

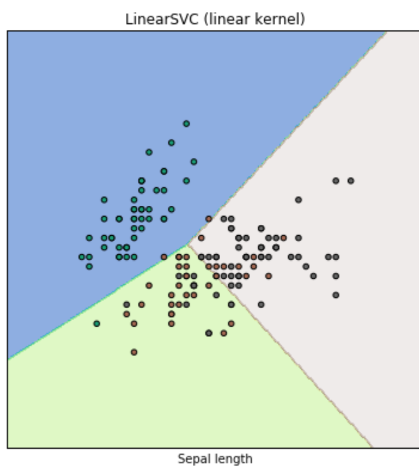
Add titles for the plots

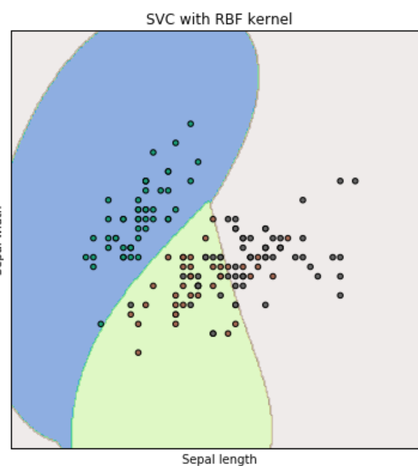
```
titles = ['SVC with linear kernel',
          'LinearSVC (linear kernel)',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel']
```

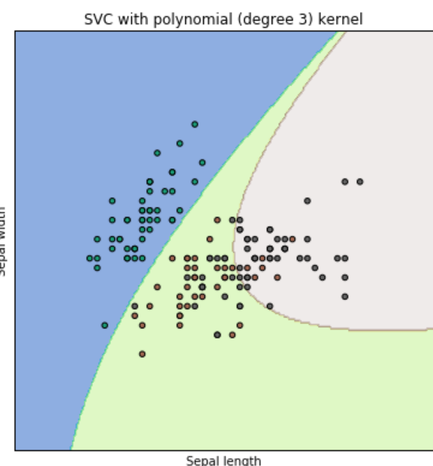
Finally we will go through each model, set its position as a subplot, then scatter the data points and draw a countour of the decision boundaries.

```
# Use enumerate for a count
for i, clf in enumerate((svc, lin_svc, rbf_svc, poly_svc)):

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, m_max]x[y_min, y_max].
    plt.figure(figsize=(15,15))
    # Set the subplot position (Size = 2 by 2, position deifined by i count
    plt.subplot(2, 2, i + 1)

    # Subplot spacing
    plt.subplots_adjust(wspace=0.4, hspace=0.4)

    # Define Z as the prediction, note the use of ravel to format the arrays
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)

    # Contour plot (filled with contourf)
    plt.contourf(xx, yy, Z, cmap=plt.cm.terrain, alpha=0.5)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Dark2)

    # Labels and Titles
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title(titles[i])

plt.show()
```









Additional Support Vector Machine Resources:

1.) Microsoft Research Paper SVM Tutorial
2.) StatSoft Online Textbook
3.) Sci Kit Learn Documentation
4.) Wikipedia
5.) Columbia Lecture Slides
6.) Andrew Ng's Class Notes

Supervised Learning using NAÏVE BAYES CLASSIFIERS

This section requires understanding of mathematical notation.

Capital pi is used to show the product of sequences:

$$\prod_{i=1}^{4} i = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

arg max f(x) the argument of the maximum (arg max or argmax) is the set of *inputs* that correspond to the maximum outputs of a function. The set may be empty, have one element, or have multiple elements.

Bayes' Theorem

Assesses the likelihood of A given B when you know the overall likelihood of A, of B, and the likelihood of B given A.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

When you don't know the overall likelihood of B, use

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|notA)P(notA)}$$

Naïve Bayes Equation

$$P(y|x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, \ldots, x_n)}$$

In other words, the probability of y given a specific set of conditions x is equal to the overall probability of y times the product of all the individual, *independent* conditions x given y divided by the overall probability of the conditions.

Constructing a classifier from the probability model (and ultimately, a decision rule)

The goal here is to use the Naïve Bayes probability model to drive a decision model that selects the most probable class for y based on given attributes x. Picking the hypothesis that is most probable is known as the *maximum a posteriori* or MAP decision rule. Selecting the appropriate classifier depends on our assumption of the *distributions* of attributes x.

Gaussian Naïve Bayes

Mostly out of convenience, we'll assume that each variable x follows a continuous, normal distribution. In the case of the Iris flower data set, this means that petal and sepal lengths and widths are all continuous, normally distributed quantities independent of one another.

Gaussian Naïve Bayes with SciKit Learn

Note: we only need SciKit Learn for the following operations (not numpy, pandas or matplotlib)

```
from sklearn import datasets
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
iris = datasets.load_iris()
X = iris.data
Y = iris.target
model = GaussianNB()
from sklearn.cross_validation import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y)
model.fit(X_train,Y_train)
predicted = model.predict(X_test)
expected = Y_test
print metrics.accuracy_score(expected, predicted)
0.947368421053
```

Here we scored a 94.7% accuracy using Naive Bayes! (In my runs, I always scored 100%)

For more info on Naïve Bayes:

1.) [SciKit Learn Documentation]
2.) [Naive Bayes with NLTK]
3.) [Wikipedia on Naive Bayes]
4.) [Andrew Ng's Class Notes]
5.) [Andrew Ng's Video Lecture on Naive Bayes]
6.) [UC Berkeley Lecture by Pieter Abbeel]

DECISION TREES and RANDOM FORESTS

Begin by visiting Jose Portilla's "Enchanted Random Forest" blog post:
https://medium.com/@josemarcialportilla/enchanted-random-forest-b08d418cb411#.v6u2qqm5f

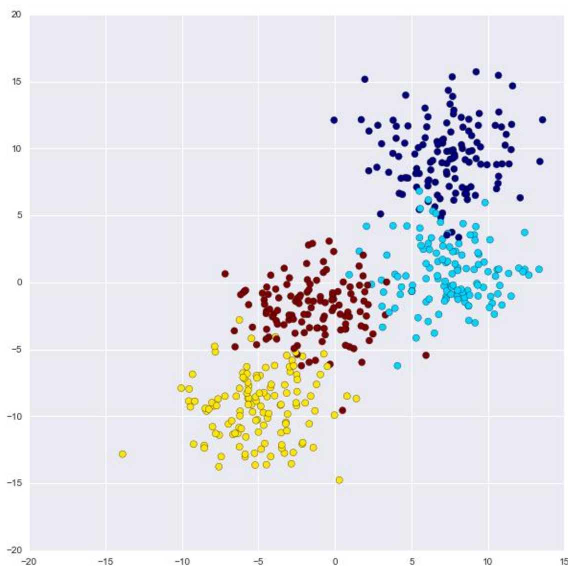
Random Forests are a classic example of an "ensemble learner", made up of many weak learners (decision trees).

Decision Trees using SciKit Learn:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn

from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=500, centers=4, random_state=8, cluster_std=2.4)
```

Scatter plot the points

```
plt.figure(figsize=(10,10))
plt.scatter(X[:,0], X[:,1], c=y, s=50, cmap='jet');
```




```
from sklearn.tree import DecisionTreeClassifier
```

Before we begin implementing the Decision Tree, lets create a nice function to plot out the decision boundaries using mesh grid (a technique common to the Sci-Kit Learn documentation).

Visualization Function

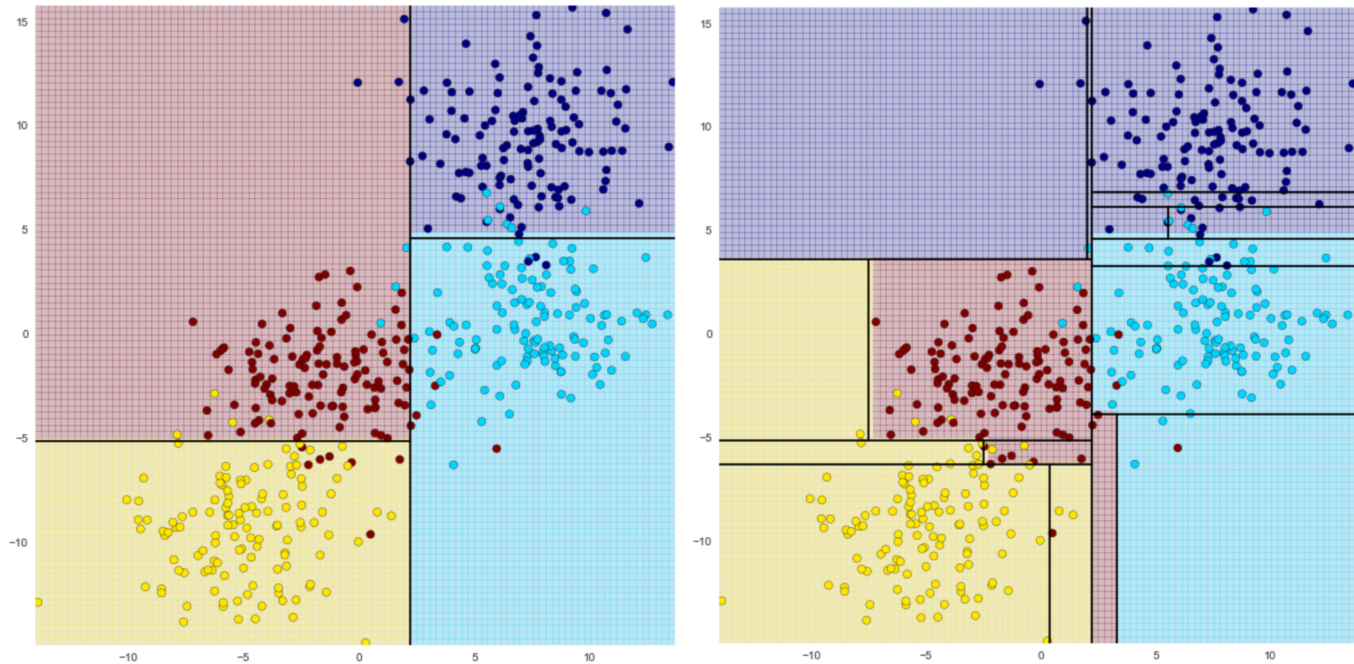
```
def visualize_tree(classifier, X, y, boundaries=True,xlim=None, ylim=None):
    '''
    Visualizes a Decision Tree.
    INPUTS: Classifier Model, X, y, optional x/y limits.
    OUTPUTS: Meshgrid visualization for boundaries of the Decision Tree
    '''
    # Fit the X and y data to the tree
    classifier.fit(X, y)

    # Automatically set the x and y limits to the data (+/- 0.1)
    if xlim is None:
        xlim = (X[:, 0].min() - 0.1, X[:, 0].max() + 0.1)
    if ylim is None:
        ylim = (X[:, 1].min() - 0.1, X[:, 1].max() + 0.1)

    # Assign the variables
    x_min, x_max = xlim
    y_min, y_max = ylim

    # Create a mesh grid
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))

    # Define the Z by the predictions (this will color in the mesh grid)
    Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])

    # Reshape based on meshgrid
    Z = Z.reshape(xx.shape)

    # Plot the figure (use)
    plt.figure(figsize=(10,10))
    plt.pcolormesh(xx, yy, Z, alpha=0.2, cmap='jet')

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')

    #Set Limits
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)

    def plot_boundaries(i, xlim, ylim):
        ''' Plots the Decision Boundaries '''
        if i < 0:
            return

        # Shorter variable name
        tree = classifier.tree_

        # Recursively go through nodes of tree to plot boundaries.
        if tree.feature[i] == 0:
            plt.plot([tree.threshold[i], tree.threshold[i]], ylim, '-k')
            plot_boundaries(tree.children_left[i],
                            [xlim[0], tree.threshold[i]], ylim)
            plot_boundaries(tree.children_right[i],
                            [tree.threshold[i], xlim[1]], ylim)

        elif tree.feature[i] == 1:
            plt.plot(xlim, [tree.threshold[i], tree.threshold[i]], '-k')
            plot_boundaries(tree.children_left[i], xlim,
                            [ylim[0], tree.threshold[i]])
            plot_boundaries(tree.children_right[i], xlim,
                            [tree.threshold[i], ylim[1]])

    # Random Forest vs Single Tree
    if boundaries:
        plot_boundaries(0, plt.xlim(), plt.ylim())
```

Set model variable

```
clf = DecisionTreeClassifier(max_depth=2,random_state=0)  ON LEFT
clf = DecisionTreeClassifier(max_depth=4,random_state=0)  ON RIGHT
```

Show Boundaries

```
visualize_tree(clf,X,y)
```



Notice how changing the depth of the decision causes the boundaries to change substantially! If we pay close attention to the second model we can begin to see evidence of over-fitting. This basically means that if we were to try to predict a new point the result would be influenced more by the noise than the signal.
So how do we address this issue? The answer is by creating an ensemble of decision trees.

Random Forests

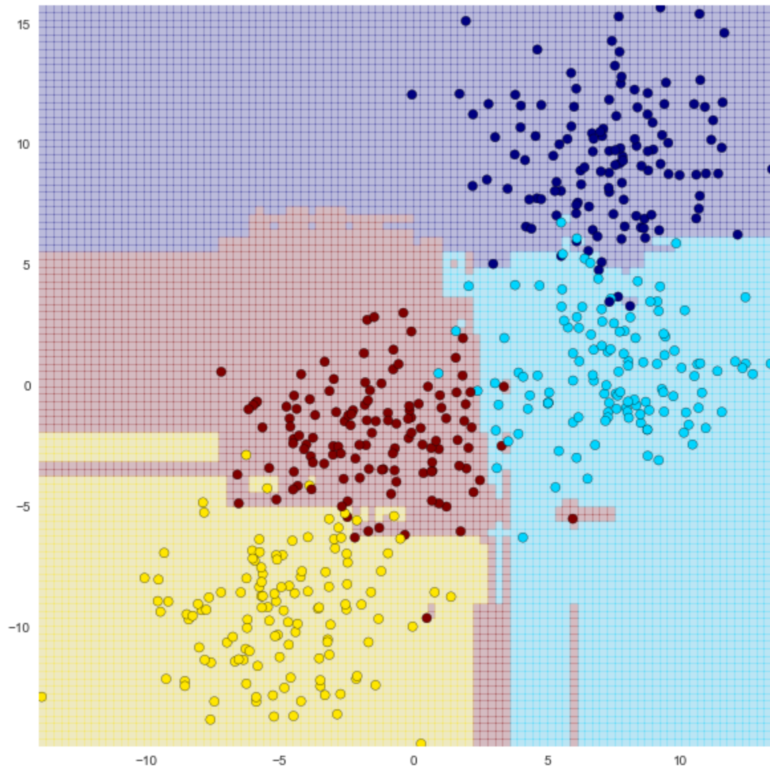
Ensemble Methods essentially average the results of many individual estimators which over-fit the data. The resulting estimates are much more robust and accurate than the individual estimates which make them up! One of the most common ensemble methods is the Random Forest, in which the ensemble is made up of many decision trees which are in some way perturbed. Lets see how we can use Sci-Kit Learn to create a random forest (its actually very simple!)
Note that n_estimators stands for the numerb of trees to use. You would intuitively know that using more decision trees would be better, but after a certain amount of trees (somewhere between 100-400 depending on your data) the benefits in accuracy of adding more estimators significantly decreases and just becomes a load on your CPU.

```
from sklearn.ensemble import RandomForestClassifier
```

n_estimators

```
clf = RandomForestClassifier(n_estimators=100,random_state=0)
```

Get rid of boundaries to avoid error
```
visualize_tree(clf,X,y,boundaries=False)
```



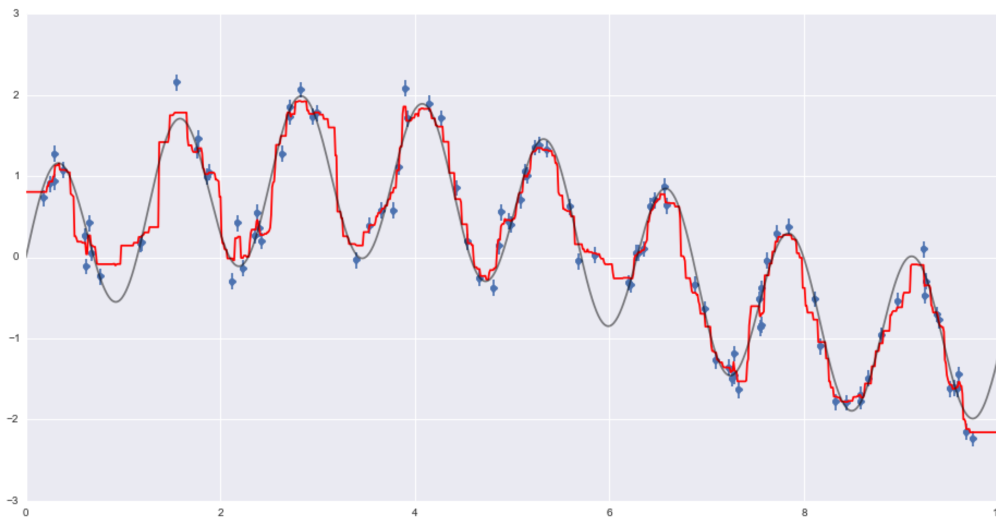
You can see that the random forest has been able to pick up features that the Decision Tree was not able to (although we must be careful of over-fitting with Random Forests too!)
While a visual is nice, a better way to evaluate our model would be with train test split if we had real data!

Random Forest Regression
We can also use Random Forests for Regression! Let's see a quick example!
Let's imagine we have some sort of weather data that's sinusoidal in nature with some noise.
It has a slow oscillation component, a fast oscillation component, and then a random noise component.

```
from sklearn.ensemble import RandomForestRegressor
x = 10 * np.random.rand(100)
def sin_model(x, sigma=0.2):
    '''
    Generate random sinusoidal data for regression analysis. Does SciKit-Learn have this?
    '''
    noise = sigma * np.random.randn(len(x))
    return np.sin(5 * x) + np.sin(0.5 * x) + noise
# Call y for data with x
y = sin_model(x)
# Plot x vs y
plt.figure(figsize=(16,8))
plt.errorbar(x, y, 0.1, fmt='o')
```
Shows a sinusoidal plot of data points

Now lets use a Random Forest Regressor to create a fitted regression, obviously a standard linear regression approach wouldn't work here. And if we didn't know anything about the true nature of the model, polynomial or sinusoidal regression would be tedious.

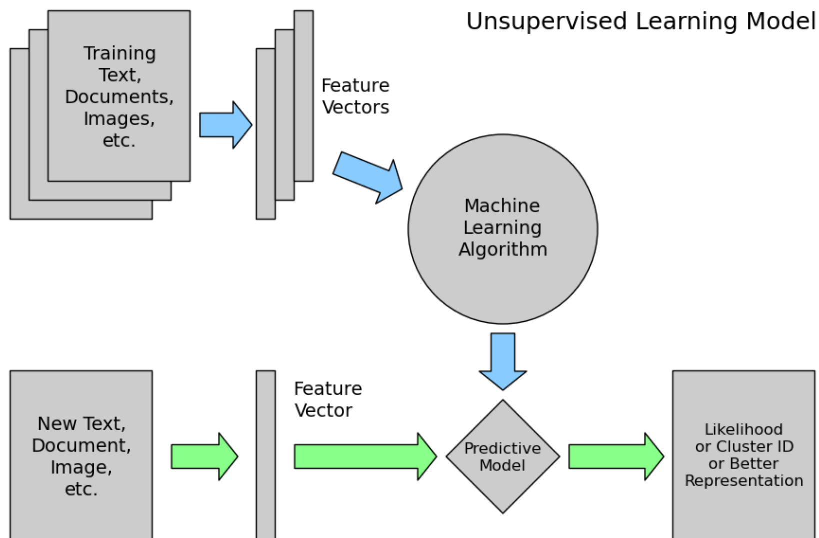
```
# X points
xfit = np.linspace(0, 10, 1000)
# Model
rfr = RandomForestRegressor(100)
# Fit Model (Format array for y with [:,None])
rfr.fit(x[:, None], y)
# Set predicted points
yfit = rfr.predict(xfit[:, None])
# Set real poitns (the model function)
ytrue = sin_model(xfit, 0)
# Plot
plt.figure(figsize=(16,8))
plt.errorbar(x, y, 0.1, fmt='o')
plt.plot(xfit, yfit, '-r');
plt.plot(xfit, ytrue, '-k', alpha=0.5);
```



As you can see, the non-parametric random forest model is flexible enough to fit the multi-period data, without us even specifying a multi-period model!
This is a tradeoff between simplicity and thinking about what your data actually is.

More resources for Random Forests:
Extensive Article on Wikipedia
Kaggle Overview
A whole webpage form the inventors themselves! Leo Breiman and Adele Cutler
It's strange to think Random Forests is actually trademarked!

Unsupervised Learning – NATURAL LANGUAGE PROCESSING




```
import nltk
```

Then, download the SMS spam collection dataset from the UCI datasets:

```
messages = [line.rstrip() for line in
                      open('smsspamcollection/SMSSpamCollection')]
print len(messages)
5574

for message_no, message in enumerate(messages[:10]):
    print message_no, message
    print '\n'
```

Displays the first 10 messages. Shows "ham" or "spam", tab separated.

Exploratory Data Analysis (EDA)

```
import pandas
messages = pandas.read_csv('smsspamcollection/SMSSpamCollection', sep='\t',
                           names=["label", "message"])
messages.head()
```

	label	message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

use `messages.groupby('label').describe()` to see that 4825 messages are "ham", 747 are "spam"

Feature Engineering

Drilling down on specific features of our dataset. First we'll tackle length:

```
messages['length'] = messages['message'].apply(len)      adds a "length" column
import matplotlib.pyplot as plt
%matplotlib inline
messages['length'].plot(bins=50, kind='hist');
```

Shows that most messages fall betwn 0-200 characters, but some may be as large as 1000.

```
messages.length.describe()
```

Tells us the longest message is 910 characters

```
messages[messages['length'] == 910]['message'].iloc[0]
```
Shows us a particularly verbose love note.

Let's plot lengths for each label separately:
```
messages.hist(column='length', by='label', bins=50,figsize=(10,4));
```
Shows us that "ham" peaks before 150 characters, while "spam" has a peak between 150-200.

Text Pre-processing

In order to classify the corpus we need to convert text content into some sort of numerical feature vector.
A simple method is the bag-of-words approach, where each unique word in a text will be represented by one number.
To massage the data we'll first strip punctuation using the string module, then the stopwords using one of NLTK's libraries:
```
def text_process(mess):
    """
    Takes in a string of text, then performs the following:
    1. Remove all punctuation (Note: spaces are NOT removed)
    2. Remove all stopwords
    3. Returns a list of the cleaned text
    """
    nopunc = [char for char in mess if char not in string.punctuation]
    nopunc = ''.join(nopunc)
    return [word for word in nopunc.split() if word.lower() not in
                          stopwords.words('english')]
```

Continuing Normalization

Removing punctuation and stop words are steps toward normalizing the data. We can continue normalizing with tools such as stemming (reducing words to their roots to improve counts, for example "traveling" == "travel") or distinguishing by part of speech. For more info: http://www.nltk.org/book/

Vectorization

We have converted the messages into lists of token words (lemmas). Now we need to convert the lists into numerical vectors. We'll do that in three steps using the bag-of-words model:

1. Count how many times does a word occur in each message (known as term frequency)
2. Weigh the counts, so that frequent tokens get lower weight (inverse document frequency)
3. Normalize the vectors to unit length, to abstract from the original text length (L2 norm)

```
from sklearn.feature_extraction.text import CountVectorizer
bow_transformer = CountVectorizer(analyzer=text_process).fit(messages['message'])
```
Returns a sparse matrix where each unique word is a row, and each message a column. Cells contain word counts.
```
print len(bow_transformer.vocabulary_)
11444
```
there are 11,444 unique words in this corpus

Term Frequency – Inverse Document Frequency (TF-IDF)

The tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

One of the simplest ranking functions is computed by summing the tf-idf for each query term; many more sophisticated ranking functions are variants of this simple model.

Typically, the tf-idf weight is composed by two terms: the first computes the normalized Term Frequency (TF), aka. the number of times a word appears in a document, divided by the total number of words in that document; the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

Example:

Consider a document containing 100 words wherein the word cat appears 3 times.

The term frequency (tf) for cat is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word cat appears in one thousand of these. Then, the inverse document frequency (idf) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-idf weight is the product of these quantities: 0.03 * 4 = 0.12.

Doing this in SciKit Learn:

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer().fit(messages_bow)
tfidf4 = tfidf_transformer.transform(bow4)
print tfidf4
  (0, 9570)     0.538562626293
  (0, 7197)     0.438936565338
  (0, 6232)     0.318721689295
  (0, 6214)     0.299537997237
  (0, 5270)     0.297299574059
  (0, 4638)     0.266198019061
  (0, 4073)     0.408325899334
```

To find the IDF of "u" and of "university":

```
print tfidf_transformer.idf_[bow_transformer.vocabulary_['u']]
print tfidf_transformer.idf_[bow_transformer.vocabulary_['university']]
3.28005242674
8.5270764989
```

Training a Model

APPENDIX I – SciKit Learn Boston Dataset:

```
print boston.DESCR
Boston House Prices dataset

Notes
------
Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
http://archive.ics.uci.edu/ml/datasets/Housing
```

About the dataset (from http://www.colorado.edu/ibs/crs/workshops/R_1-11-2012/root/Harrison_1978.pdf)

David Harrison, Jr. and Daniel L. Rubinfeld published "Hedonic Housing Prices and the Demand for Clean Air" in the Journal of Environmental Economics and Management (vol5, pp81-102, 1978)

"This paper investigates the methodological problems associated with the use of housing market data to measure the willingness to pay for clean air. With the use of a hedonic housing price model and data for the Boston metropolitan area, quantitative estimates of the willingness to pay for air quality improvements are generated. Marginal air pollution damages (as revealed in the housing market) are found to increase with the level of air pollution and with household income. The results are relatively sensitive to the specification of the hedonic housing price equation, but insensitive to the specification of the air quality demand equation."

"This paper investigates the methodological problems associated with the housing market approach. While several studies have used this methodology to estimate the demand for air quality improvements, they have paid little attention to the sensitivity of the results to the assumptions embedded in the procedures. Using data for the Boston housing market, we generate quantitative estimates of the willingness to pay for air quality improvements and test the sensitivity of those results to alternative specifications of the basic building blocks in the procedure. Our data base is superior to others because it contains a large number of neighborhood variables (necessary to isolate the independent influence of air pollution) and more reliable air pollution data."

Investopedia defines *hedonic pricing* as "A model identifying price factors according to the premise that price is determined both by internal characteristics of the good being sold and external factors affecting it."

APPENDIX II: FOR FURTHER RESEARCH

Value of coefficients:

1. What determines the strength of a coefficient? absolute value? p-value?
2. How much does the coefficient affect the target value if the associated variable doesn't change much?
   ie, a low coefficient on # rooms may have greater effect when rooms can double from 2 to 4 quite easily, where a high coefficient on NOX may not matter much if the variation over our sample set is only 1 or 2 ppm.
3. What about orders of magnitude? A small change to a big number may outweigh a big change to a small one.
4. What about non-linear functions? There should be a greater effect between 3 and 4 room houses than between 7 and 8 rooms.

Multicollinearity:

Need to play around with this some more (perhaps with a SciKit Learn dataset instead of statsmodel). How best to choose which category to drop. In the affair example, we arbitrarily dropped the only negative-coefficient category (student). What happens if we chose a different one / multiples / none?