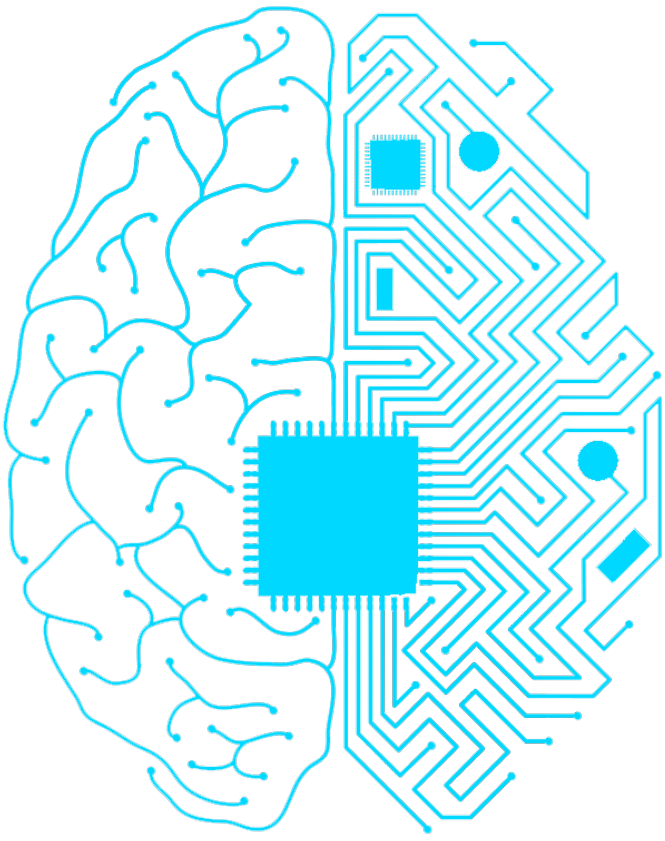
Machine Learning, Data Science and Deep Learning

Introductory Course Using Python



Shav Vimalendiran

25 October 2020

Contents

1 Introduction **1**

2 Python Basics **1**
2.1 Whitespace Is Important . . . 1
2.2 Importing Modules . . . 2
2.3 Lists . . . 2
2.4 Tuples . . . 3
2.5 Dictionaries . . . 4
2.6 Functions . . . 4
2.7 Boolean Expressions . . . 5
2.8 Looping . . . 5

3 Introducing Pandas **6**
3.1 Pandas . . . 6
3.2 NumPy . . . 6
3.3 Scikit_Learn . . . 6
3.4 Examples . . . 6

4 Statistics and Probability Refresher **11**
4.1 Mean vs. Median . . . 11
4.2 Mode . . . 12
4.3 Variance . . . 13
4.4 Standard Deviation . . . 14
4.5 Population vs. Sample . . . 14
4.5.1 Python Scripting Standard Deviation Variance . . . 14
4.6 Probability Density Functions . . . 15
4.7 Probability Mass Functions . . . 15
4.8 Percentiles and Moments . . . 16
4.9 Moments: Mean, Variance, Skew, Kurtosis . . . 17
4.9.1 Computation of Moments in Python . . . 18
4.10 MatPlotLib Basics . . . 19
4.10.1 Drawing a line graph . . . 20
4.10.2 Mutiple Plots on One Graph . . . 20
4.10.3 Save it to a File . . . 21
4.10.4 Adjust the Axes . . . 22
4.10.5 Add a Grid . . . 22
4.10.6 Change Line Types and Colors . . . 23
4.10.7 Labeling Axes and Adding a Legend . . . 24
4.10.8 XKCD Style :) . . . 24
4.10.9 Pie Chart . . . 25
4.10.10 Bar Chart . . . 26
4.10.11 Scatter Plot . . . 27
4.10.12 Histogram . . . 27
4.10.13 Box & Whisker Plot . . . 28
4.11 Covariance and Correlation . . . 29
4.12 Covariance in Numpy . . . 30
4.13 Conditional Probability . . . 33
4.13.1 Conditional Probability Activity & Exercise . . . 33
4.13.2 Looking at Independence . . . 35

4.14 Bayes' Theorem . . . 35

5 Predictive Models **36**
5.1 Regression Analysis . . . 36
5.2 Linear Regression . . . 36
5.3 Measuring error with R-squared . . . 36
5.4 Polynomial Regression . . . 38
5.4.1 Polynomial Regression using NumPy . . . 39
5.5 Multivariate Regression (Multiple Regression) . . . 40
5.6 Multi-Level Models . . . 44

6 Machine Learning with Python **44**
6.1 Supervised vs Unsupervised Learning . . . 44
6.2 Train-Test Polynomials in Python . . . 45
6.3 Bayesian Methods . . . 49
6.4 Naive Bayes In Python . . . 50
6.5 K-Means Clustering . . . 52
6.6 Limitations of K-Means Clustering . . . 53
6.7 K-Means Clustering Example . . . 53
6.8 Entropy . . . 55
6.9 Decison Trees . . . 56
6.10 Decison Trees in Python . . . 56
6.11 Ensemble learning: using a random forest . . . 59
6.12 Ensemble Learning . . . 60
6.13 Advanced Ensemble Learning . . . 60
6.14 Support Vector Machines . . . 60

7 Recommender Systems **63**
7.1 User-Based Collaborative Filtering . . . 63
7.2 Item-Based Collaborative Filtering . . . 64
7.3 Finding Similar Movies In Python . . . 64
7.4 Item-Based Collaborative Filtering . . . 70
7.5 Improvements on Item-Based Collaborative Filtering . . . 78

8 More Data Mining and Machine Learning Techniques **79**
8.1 K-Nearest Neighbor . . . 79
8.2 KNN (K-Nearest-Neighbors) . . . 80
8.3 The Curse of Dimensionality . . . 83
8.4 Principal Component Analysis . . . 84
8.5 ETL and ELT . . . 86
8.6 Reinforcement Learning . . . 87
8.7 Q-Learning . . . 88
8.8 The exploration problem . . . 88
8.9 Pac Man Recap . . . 89

9 Dealing with Real-World Data **89**
9.1 The Bias / Variance Tradeoff . . . 89
9.2 K-Fold Cross Validation . . . 90
9.3 K-Fold Cross Validation In Practice . . . 91
9.4 Cleaning Your Input Data . . . 92
9.5 Cleaning Weblog data . . . 93
9.6 Normalizing Numerical Data . . . 103

9.7 Dealing with Outliers . . . 103

10 Apache Spark Machine Learning on Big Data 105
10.1 Spark Introduction . . . 105
10.1.1 Python vs. Scala . . . 106
10.2 Resilient Distributed Dataset (RDD) . . . 107
10.2.1 Transformations . . . 107
10.2.2 Actions . . . 108
10.3 Introducing MLLib . . . 108
10.4 Decision Trees in Spark . . . 109
10.5 TF-IDF . . . 110
10.6 Create a Working, Scaleable Search Algorithm using MLLib in Spark . . . 111
10.7 Data Sets Data Frames . . . 112

11 Experimental Design 112
11.1 A/B Tests . . . 112
11.2 T-Tests and P-Values . . . 113
11.3 T-Tests and P-Values in Python . . . 113
11.4 How Long Do I Run an Experiment? . . . 114
11.5 Drawbacks of A/B Testing . . . 115
11.5.1 Correlation does not imply causation! . . . 115
11.5.2 Novelty Effects . . . 115
11.5.3 Seasonal Effects . . . 115
11.5.4 Selection Bias . . . 115
11.5.5 Data Pollution . . . 115
11.5.6 Attribution Errors . . . 115

12 Deep Learning and Neural Networks 116
12.1 Pre-Requisites . . . 116
12.2 Introducing Artificial Neural Networks . . . 116
12.2.1 The First Artificial Neurons . . . 117
12.2.2 The Linear Threshold Unit (LTU) . . . 117
12.2.3 The Perceptron . . . 117
12.2.4 Multi-Layer Perceptrons . . . 117
12.2.5 A Modern Deep Neural Network . . . 117
12.3 playground.tensorflow.org . . . 117
12.4 Deep Learning . . . 118
12.4.1 Backpropagation . . . 118
12.4.2 Activation Functions (aka Rectifier) . . . 118
12.4.3 Optimization Functions . . . 118
12.4.4 Avoiding Overfitting . . . 118
12.4.5 Tuning Your Topology . . . 119
12.5 Tensorflow . . . 119
12.5.1 Creating a Neural Network with Tensorflow . . . 119

1 Introduction

Data Science, Machine Learning AI are the most valuable technical skills to have right now. This course covers topics like Bayes theorem, regressions, clustering techniques, experimental design, decision tress etc.

We will build Artificial Neural Networks using Tensorflow and Keras. Implemented a movie recommendation using movie ratings data, spam detection using Bayesian methods, search engine from Wikipedia data, scaling up to big data and a Hadoop cluster, handwriting recognition sentiment analysis using a variety of AI techniques.

To install packages, I used "conda install pydotplus" and to check my python version I used "python –version" all in the CMD prompt.

$userRatings.to_csv(r'C:$
5.$RecommenderSystemsaName.csv', index = False)$ to save a df as a csv

too add index back in: userRatings = userRatings.reset$_index()$

2 Python Basics

2.1 Whitespace Is Important

[4]:
```
listOfNumbers = [1, 2, 3, 4, 5, 6]

for number in listOfNumbers:
    print(number)
    if (number % 2 == 0):
        print("is even")
    else:
        print("is odd")

print ("YES All done.")
```

```
1
is odd
2
is even
3
is odd
4
is even
5
is odd
6
is even
YES All done.
```

Language here uses tab. It is very important in telling python what is in what block of code notice the **colons** - they are important. the funtion here. “% 2” is the number modulus

2.2 Importing Modules

Here we inport external packages. You can name it whatever you want. Here we import the packages into python

[15]:
```
import numpy as na

A = na.random.normal(25.0, 5.0, 10)
print (A)
```

```
[27.12990721 16.609007   17.24225133 27.34120284 28.53214603 29.27160648
 31.64856167 25.53030759 21.8205691  16.77277521]
```

This is the function for the normal distribution of numbers

2.3 Lists

[8]:
```
x = [1, 2, 3, 4, 5, 6, 8, 9]
print(len(x))
```

```
8
```

Above we see the syntax for a pyth0n list. Len determines the **length** of the list. You should be aware of how most languages sart counting from 0. Element 0 is 1 here, and element is 1 is 2 here.

Below is a slice, and gives you the first numbers from a list. It gives you up until, **and not including**, the 3rd element.

[17]:
```
x[:3]
```

[17]:
```
[1, 2, 3]
```

This ones gives everything after, **including**, the 3rd element in the list

[16]:
```
x[3:]
```

[16]:
```
[4, 5, 6, 8, 9]
```

This negative syntax tells you that you want the last 2 elements of a list.

[18]:
```
x[-2:]
```

[18]:
```
[8, 9]
```

This adds a new list to our list.

[19]:
```
x.extend([7,8])
x
```

[19]:
```
[1, 2, 3, 4, 5, 6, 8, 9, 7, 8]
```

This adds a single number to the end of our list.

[10]:
```
x.append(9)
x
```

[10]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In Python, it is okay to have complex data structures. Here we can have 2 lists in a lists.

[20]:
```
y = [10, 11, 12]
listOfLists = [x, y]
listOfLists
```

[20]:
```
[[1, 2, 3, 4, 5, 6, 8, 9, 7, 8], [10, 11, 12]]
```

This is the code we use to reference a single element of a lsit. Remember how elements are indexed

[12]:
```
y[1]
```

[12]:
```
11
```

Sort function. You can also reverse sort.

[22]:
```
z = [3, 2, 1, 8, 5]
z.sort()
z
```

[22]:
```
[1, 2, 3, 5, 8]
```

[23]:
```
z.sort(reverse=True)
z
```

[23]:
```
[8, 5, 3, 2, 1]
```

2.4 Tuples

Like lists, but immutable. You can't change them like lists. cant extend, sort etc. Tuples are just immutable lists. Use () instead of []

[24]:
```
x = (1, 2, 3)
len(x)
```

[24]:
```
3
```

[26]:
```
y = (4, 5, 6)
y[2]
```

[26]:
```
6
```

[17]:
```
listOfTuples = [x, y]
listOfTuples
```

[17]:
```
[(1, 2, 3), (4, 5, 6)]
```

[18]:
```
(age, income) = "32,120000".split(',')
print(age)
print(income)
```

```
32
120000
```

In Data Science, Tuples are commonly used for assigning variables to input data as it is read in. For example, lets say we have a comma sperated value file containing ages and commas for income. The split funciton splits the comma'ed information into a 1x2 Tuple and then assign it to two variables all at once.

2.5 Dictionaries

Last data Structure we will see a lot in python. Here we have a mini database.

[29]:
```
# Like a map or hash table in other languages
captains = {}
captains["Enterprise"] = "Kirk"
captains["Enterprise D"] = "Picard"
captains["Deep Space Nine"] = "Sisko"
captains["Voyager"] = "Janeway"

print(captains["Voyager"])
```

```
Janeway
```

Now essentially you have a lookout table that will associate a captain with a ship name. Useful for lookups. What if you lookup something that doesnt exist? get funciton gives it back safely. This will give you "none" back, instead if an error

[20]:
```
print(captains.get("Enterprise"))
```

```
Kirk
```

[21]:
```
print(captains.get("NX-01"))
```

```
None
```

Iterates through every entry in dictioanry. Will give you back the lookup values for each one.

[22]:
```
for ship in captains:
    print(ship + ": " + captains[ship])
```

```
Deep Space Nine: Sisko
Enterprise: Kirk
Enterprise D: Picard
Voyager: Janeway
```

2.6 Functions

Lets you repeat a set of operations over and over again for differnet parameters. Rememeber, unlike MATLAB, you need whitespace. No curly brackets, etc

[32]:
```
def SquareIt(x):
    return x * x

print(SquareIt(2))
```

```
4
```

Some cool things you can do, is pass functions around as parameters.

[24]:
```
def DoSomething(f, x):
    return f(x)

print(DoSomething(SquareIt, 3))
```

```
9
```

Kinda unique to Python. Here we inline simple functions into a function. This way you can just do the fucntion here, the function exists right now, its trnsitory, etc.

[25]:
```
#Lambda functions let you inline simple functions
print(DoSomething(lambda x: x * x * x, 3))
```

```
27
```

2.7 Boolean Expressions

== tests for equality.

[26]:
```
print(1 == 3)
```

```
False
```

[55]:
```
print(True or False)
```

```
True
```

“is” is similar to “==”, and is more of a python way of doing things.

[ ]:
```
print(1 is 3)
```

[59]:
```
if 1 == 3:
    print("How did that happen?")
elif 1 > 3:
    print("Yikes")
else:
    print("All is well with the world")
```

```
All is well with the world
```

2.8 Looping

range operator automatically defines a list of numebers in the list.

[72]:
```
for x in range(10):
    print(x),
```

```
0 1 2 3 4 5 6 7 8 9
```

You can choose to stop the iteration of a loop prematurely if certain conditions are met.

[75]:
```
for x in range(10):
    if (x == 1):
        continue
```

```
    if (x > 5):
        break
    print(x),
```

```
0 2 3 4 5
```

This one starts with x.

[71]:
```
x = 0
while (x < 10): #while x is less than 10 print out, and increment by 1
    print(x)
    x += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

3 Introducing Pandas

Pandas is a Python library that makes handling tabular data easier. Since we're doing data science - this is something we'll use from time to time!

It's one of three libraries you'll encounter repeatedly in the field of data science:

3.1 Pandas

Introduces "Data Frames" and "Series" that allow you to slice and dice rows and columns of information.

3.2 NumPy

Usually you'll encounter "NumPy arrays", which are multi-dimensional array objects. It is easy to create a Pandas DataFrame from a NumPy array, and Pandas DataFrames can be cast as NumPy arrays. NumPy arrays are mainly important because of...

3.3 Scikit_Learn

The machine learning library we'll use throughout this course is scikit_learn, or sklearn, and it generally takes NumPy arrays as its input.

So, a typical thing to do is to load, clean, and manipulate your input data using Pandas. Then convert your Pandas DataFrame into a NumPy array as it's being passed into some Scikit_Learn function. That conversion can often happen automatically.

3.4 Examples

Let's start by loading some comma-separated value data using Pandas into a DataFrame:

For example, you might use Pandas to input your data, manipulate it, clean it and understand it, and then translate it into a NumPy array and put it into scikit learn.

[13]:
```
%matplotlib inline
import numpy as np #inline means the graph comes up here and not in a seperate␣
 ↪window
import pandas as pd #create an alias
```

```
df = pd.read_csv("PastHires.csv") #past hires is a comma seperated values file, and␣
 ↪the pandas function makes a data frame out of it and assigns it
df.head() #this visualises the first 5 rows of that data frame
```

[13]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	Y	4	BS	N	N	Y
1	0	N	0	BS	Y	Y	Y
2	7	N	6	BS	N	N	N
3	2	Y	1	MS	Y	N	Y
4	20	N	2	PhD	Y	N	N

head() is a handy way to visualize what you've loaded. You can pass it an integer to see some specific number of rows at the beginning of your DataFrame:

[2]:
```
df.head(10)
```

[2]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	Y	4	BS	N	N	Y
1	0	N	0	BS	Y	Y	Y
2	7	N	6	BS	N	N	N
3	2	Y	1	MS	Y	N	Y
4	20	N	2	PhD	Y	N	N
5	0	N	0	PhD	Y	Y	Y
6	5	Y	2	MS	N	Y	Y
7	3	N	1	BS	N	Y	Y
8	15	Y	5	BS	N	N	Y
9	0	N	0	BS	N	N	N

You can also view the end of your data with tail():

[3]:
```
df.tail(4)
```

[3]:
```
    Years Experience Employed?  Previous employers Level of Education  \
9                  0         N                   0                BS
10                 1         N                   1               PhD
11                 4         Y                   1                BS
12                 0         N                   0               PhD

   Top-tier school Interned Hired
9                N        N     N
10               Y        N     N
11               N        Y     Y
12               Y        N     Y
```

We often talk about the “shape” of your DataFrame. This is just its dimensions. This particular CSV file has 13 rows with 7 columns per row:

[4]:
```
df.shape
```

[4]:
```
(13, 7)
```

The total size of the data frame is the rows * columns:

[5]:
```
df.size
```

[5]:
```
91
```

The len() function gives you the number of rows in a DataFrame:

[6]:
```
len(df)
```

[6]:
```
13
```

If your DataFrame has named columns (in our case, extracted automatically from the first row of a .csv file,) you can get an array of them back:

[7]:
```
df.columns
```

[7]:
```
Index(['Years Experience', 'Employed?', 'Previous employers',
       'Level of Education', 'Top-tier school', 'Interned', 'Hired'],
      dtype='object')
```

Extracting a single column from your DataFrame looks like this - this gives you back a “Series” in Pandas:

[8]:
```
df['Hired']
```

[8]:
```
0     Y
1     Y
2     N
3     Y
4     N
5     Y
6     Y
7     Y
8     Y
```

```
9     N
10    N
11    Y
12    Y
Name: Hired, dtype: object
```

You can also extract a given range of rows from a named column, like so:

[28]:
```
df['Hired'][:5]
```

[28]:
```
0    Y
1    Y
2    N
3    Y
4    N
Name: Hired, dtype: object
```

Or even extract a single value from a specified column / row combination:

[10]:
```
df['Hired'][5]
```

[10]:
```
'Y'
```

To extract more than one column, you pass in an array of column names instead of a single one:

[11]:
```
df[['Years Experience', 'Hired']]
```

[11]:

	Years Experience	Hired
0	10	Y
1	0	Y
2	7	N
3	2	Y
4	20	N
5	0	Y
6	5	Y
7	3	Y
8	15	Y
9	0	N
10	1	N
11	4	Y
12	0	Y

You can also extract specific ranges of rows from more than one column, in the way you'd expect:

[12]:
```
df[['Years Experience', 'Hired']][:5]
```

[12]:

	Years Experience	Hired
0	10	Y
1	0	Y
2	7	N
3	2	Y
4	20	N

Sorting your DataFrame by a specific column looks like this:

[13]:
```
df.sort_values(['Years Experience'])
```

[13]:
```
    Years Experience Employed?  Previous employers Level of Education  \
1                  0         N                    0                BS
5                  0         N                    0               PhD
9                  0         N                    0                BS
12                 0         N                    0               PhD
10                 1         N                    1               PhD
3                  2         Y                    1                MS
7                  3         N                    1                BS
11                 4         Y                    1                BS
6                  5         Y                    2                MS
2                  7         N                    6                BS
0                 10         Y                    4                BS
8                 15         Y                    5                BS
4                 20         N                    2               PhD

   Top-tier school Interned Hired
1                Y        Y     Y
5                Y        Y     Y
9                N        N     N
12               Y        N     Y
10               Y        N     N
3                Y        N     Y
7                N        Y     Y
11               N        Y     Y
6                N        Y     Y
2                N        N     N
0                N        N     Y
8                N        N     Y
4                Y        N     N
```

You can break down the number of unique values in a given column into a Series using value_counts() - this is a good way to understand the distribution of your data:

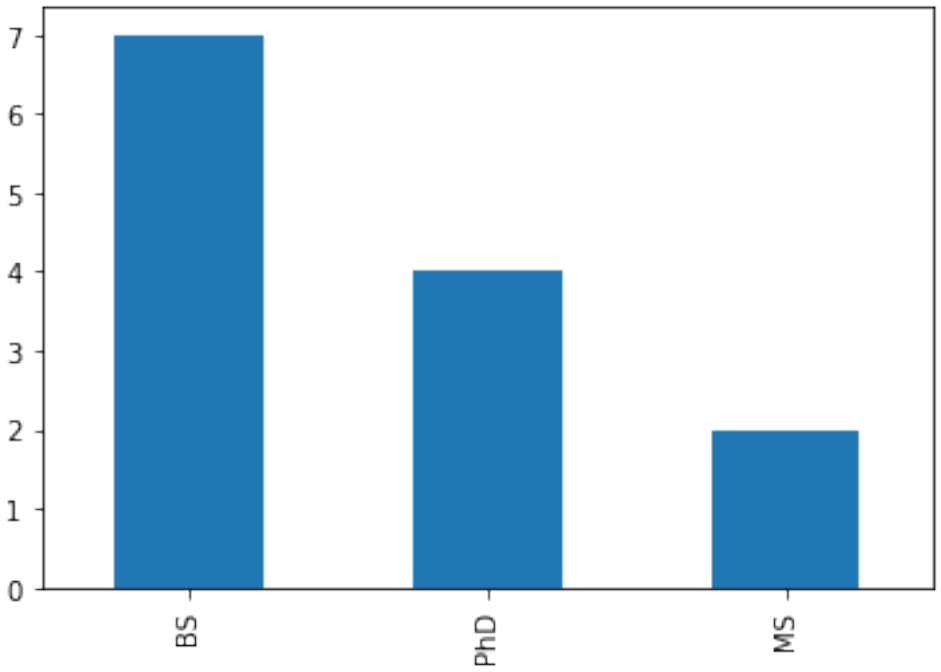
[8]:
```
degree_counts = df['Level of Education'].value_counts()
degree_counts
```

[8]:
```
BS     7
PhD    4
MS     2
Name: Level of Education, dtype: int64
```

Pandas even makes it easy to plot a Series or DataFrame - just call plot():

[9]:
```
degree_counts.plot(kind='bar')
```

[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x21382db9040>
```



4 Statistics and Probability Refresher

4.1 Mean vs. Median

Let's create some fake income data, centered around 27,000 with a normal distribution and standard deviation of 15,000, with 10,000 data points. (We'll discuss those terms more later, if you're not familiar with them.)

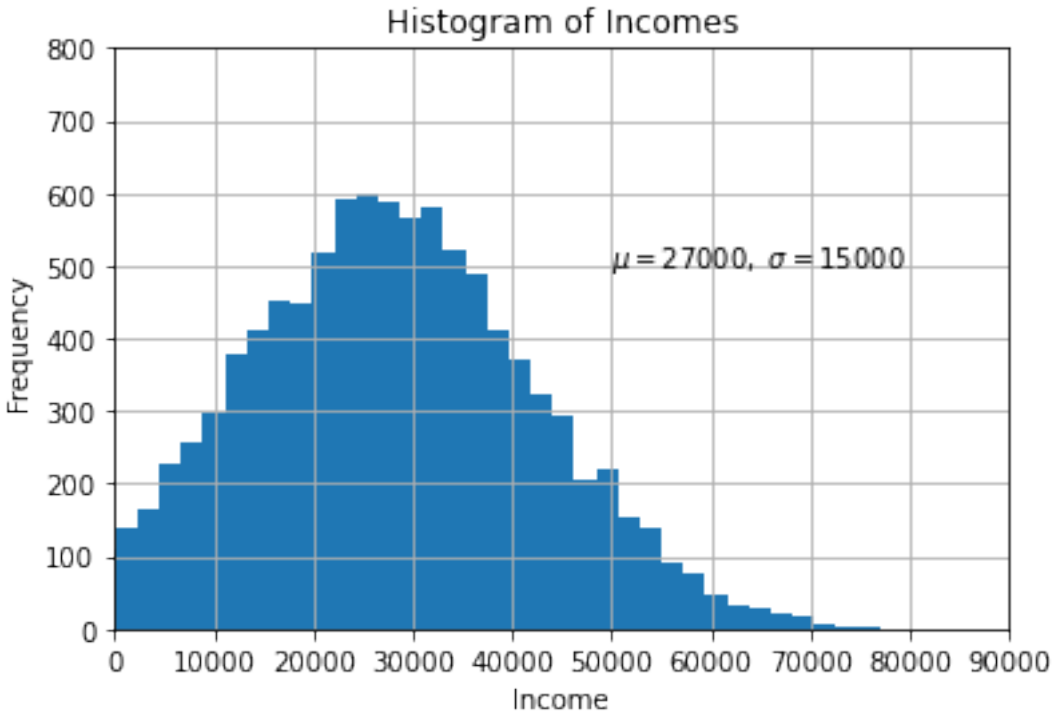
Then, compute the mean (average) - it should be close to 27,000:

[27]:
```
import numpy as np

incomes = np.random.normal(27000, 15000, 10000)
np.mean(incomes)
```

[27]:
```
27308.198701844085
```

We can segment the income data into 50 buckets, and plot it as a histogram:

[28]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.hist(incomes, 50) # in 50 differnt groups/ buckets
plt.xlabel('Income')
plt.ylabel('Frequency')
plt.title('Histogram of Incomes')
plt.text(50000, 500, r'$\mu=27000,\ \sigma=15000$')
plt.xlim(0, 90000)
plt.ylim(0, 800)
plt.grid(True)
plt.show()
```



Now compute the median - since we have a nice, even distribution it too should be close to 27,000:

[3]:
```
np.median(incomes)
```

[3]:
```
27195.153719614136
```

Now we'll add Donald Trump into the mix. Darn income inequality!

[29]:
```
incomes = np.append(incomes, [1000000000])
```

The median won't change much, but the mean does:

[30]:
```
np.median(incomes)
```

[30]:
```
27222.885666270337
```

[32]:
```
np.mean(incomes)
```

[32]:
```
127295.46915492858
```

Great example of how the ouliers can greatly skew the mean

4.2 Mode

Next, let's generate some fake age data for 500 people:

[34]:
```
ages = np.random.randint(18, high=90, size=500)
ages
```

[34]:
```
array([87, 31, 32, 68, 39, 73, 59, 86, 73, 65, 76, 45, 51, 68, 86, 59, 34,
       79, 33, 74, 72, 42, 40, 25, 21, 27, 84, 18, 41, 67, 41, 38, 63, 77,
       19, 84, 82, 65, 36, 57, 79, 59, 80, 21, 23, 84, 70, 70, 55, 56, 74,
       44, 70, 47, 87, 36, 59, 72, 67, 76, 19, 30, 56, 50, 28, 61, 25, 28,
       68, 22, 60, 36, 49, 39, 34, 22, 43, 83, 39, 21, 83, 50, 28, 87, 55,
       73, 67, 44, 63, 72, 49, 71, 64, 58, 47, 35, 53, 47, 42, 75, 66, 86,
       38, 37, 54, 82, 56, 75, 83, 73, 85, 42, 23, 63, 49, 37, 80, 70, 20,
       18, 45, 80, 72, 61, 40, 50, 25, 51, 39, 51, 28, 50, 75, 30, 34, 42,
       73, 62, 36, 39, 76, 40, 60, 63, 58, 62, 28, 43, 64, 83, 63, 32, 67,
       78, 74, 66, 73, 32, 52, 27, 51, 31, 27, 44, 25, 52, 18, 58, 55, 80,
       42, 64, 18, 37, 27, 80, 36, 38, 49, 43, 18, 50, 64, 72, 24, 60, 71,
       36, 26, 83, 69, 25, 35, 28, 27, 85, 71, 45, 48, 30, 63, 69, 87, 89,
       62, 34, 73, 71, 39, 41, 56, 79, 71, 18, 34, 39, 73, 65, 46, 49, 69,
       47, 59, 24, 66, 84, 68, 84, 84, 42, 80, 83, 84, 67, 85, 34, 25, 89,
       33, 33, 76, 61, 30, 74, 59, 69, 63, 42, 46, 41, 83, 82, 74, 87, 81,
       89, 23, 44, 38, 60, 85, 42, 63, 79, 73, 22, 78, 46, 56, 87, 53, 30,
       52, 70, 70, 57, 78, 81, 43, 48, 78, 35, 71, 75, 86, 29, 56, 34, 18,
       80, 22, 32, 18, 39, 41, 42, 47, 28, 41, 74, 25, 86, 39, 71, 35, 51,
       23, 49, 41, 67, 67, 89, 58, 26, 27, 48, 41, 60, 44, 28, 23, 66, 27,
       73, 69, 77, 88, 76, 44, 30, 62, 80, 74, 67, 42, 57, 38, 24, 56, 56,
       25, 34, 37, 22, 54, 76, 36, 71, 52, 36, 75, 80, 88, 21, 59, 53, 42,
       72, 50, 40, 78, 33, 48, 39, 61, 74, 37, 79, 53, 62, 40, 57, 33, 73,
       37, 67, 34, 65, 43, 71, 21, 22, 79, 78, 18, 59, 23, 79, 55, 82, 62,
       59, 83, 77, 48, 34, 50, 57, 76, 49, 41, 20, 66, 25, 19, 80, 89, 20,
       21, 23, 37, 51, 81, 78, 58, 29, 67, 46, 74, 39, 46, 34, 53, 27, 50,
       77, 50, 56, 51, 71, 45, 22, 60, 35, 42, 62, 35, 34, 36, 67, 19, 68,
       18, 28, 26, 80, 65, 86, 41, 37, 65, 38, 67, 56, 28, 25, 25, 58, 86,
       75, 64, 33, 31, 52, 71, 43, 46, 19, 84, 68, 21, 65, 49, 38, 84, 75,
       70, 27, 79, 55, 62, 41, 73, 39, 86, 22, 84, 78, 42, 65, 21, 45, 27,
       79, 52, 72, 83, 74, 40, 57])
```

[35]:
```
from scipy import stats
stats.mode(ages)
```

[35]:
```
ModeResult(mode=array([42]), count=array([13]))
```

4.3 Variance

Variance measures how "spread-out" the data is. Variance (σ) is simply the **average of the squared differences from the mean**

Example: What is the variance of the data set (1, 4, 5, 4, 8)?

1. First find the mean: (1+4+5+4+8)/5 = 4.4
2. Now find the differences from the mean: (-3.4, -0.4, 0.6, -0.4, 3.6)
3. Find the squared differences: (11.56, 0.16, 0.36, 0.16, 12.96)
4. Find the average of the squared differences:
5. σ^2= (11.56 + 0.16 + 0.36 + 0.16 + 12.96) / 5 = 5.04

4.4 Standard Deviation

Standard Deviation σ is just the square root of the variance.

This is usually used as a way to identify outliers. Data points that lie more than one standard deviation from the mean can be considered **unusual**.

You can talk about how extreme a data point is by talking about "how many sigmas/standard deviations" away from the mean it is.

4.5 Population vs. Sample

If you're working with a sample of data instead of an entire data set (the entire population) then you want to use the "sample variance" instead of the "population variance"

For N samples, you just divide the squared variances by N-1 instead of N. This is due to probability.

So, in our example, we computed the population variance like this:

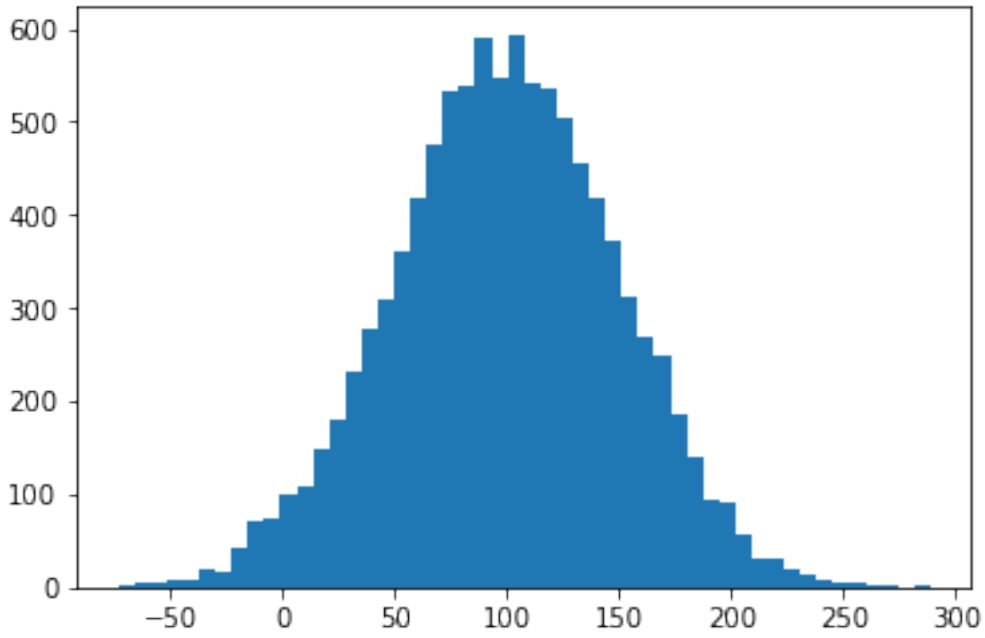
σ^2 = (11.56 + 0.16 + 0.36 + 0.16 + 12.96) / 5 = 5.04

But the sample variance would be:

S^2 = (11.56 + 0.16 + 0.36 + 0.16 + 12.96) / 4 = 6.3

4.5.1 Python Scripting Standard Deviation Variance

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

incomes = np.random.normal(100.0, 50.0, 10000)

plt.hist(incomes, 50)
plt.show()
```



Numpy makes it extremely easy to compute the std and variance, as follows. This can be done as the python list has extra stuff tagged onto it, so we can just use the std function like this:

[2]:
```
incomes.std()
```

[2]:
```
49.77851466593912
```

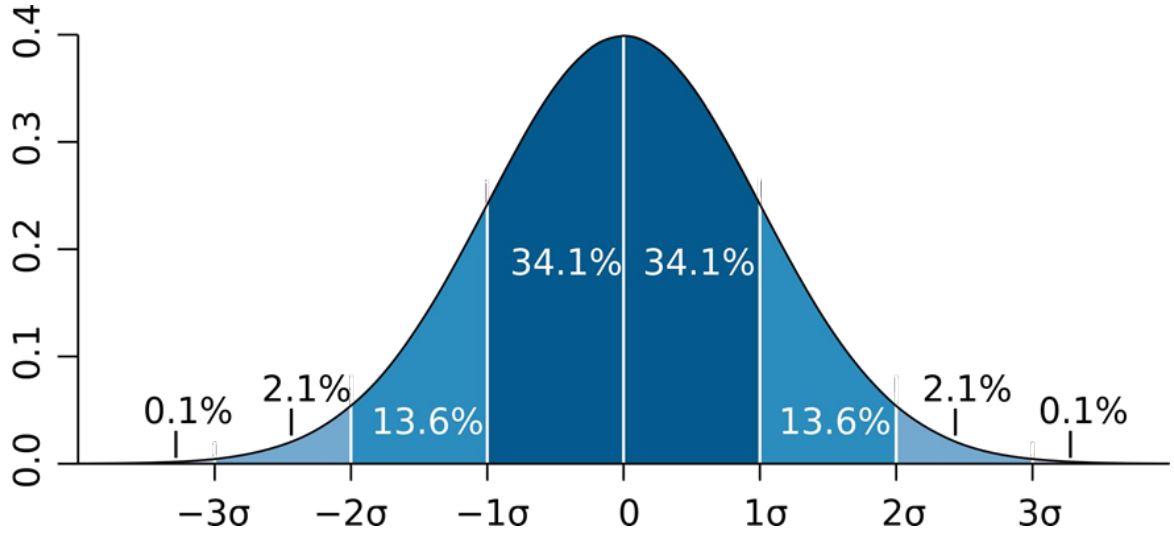
[3]:
```
incomes.var()
```

[3]:
```
2477.9005223471163
```

This works how we would expect. Std is the root of the Var.

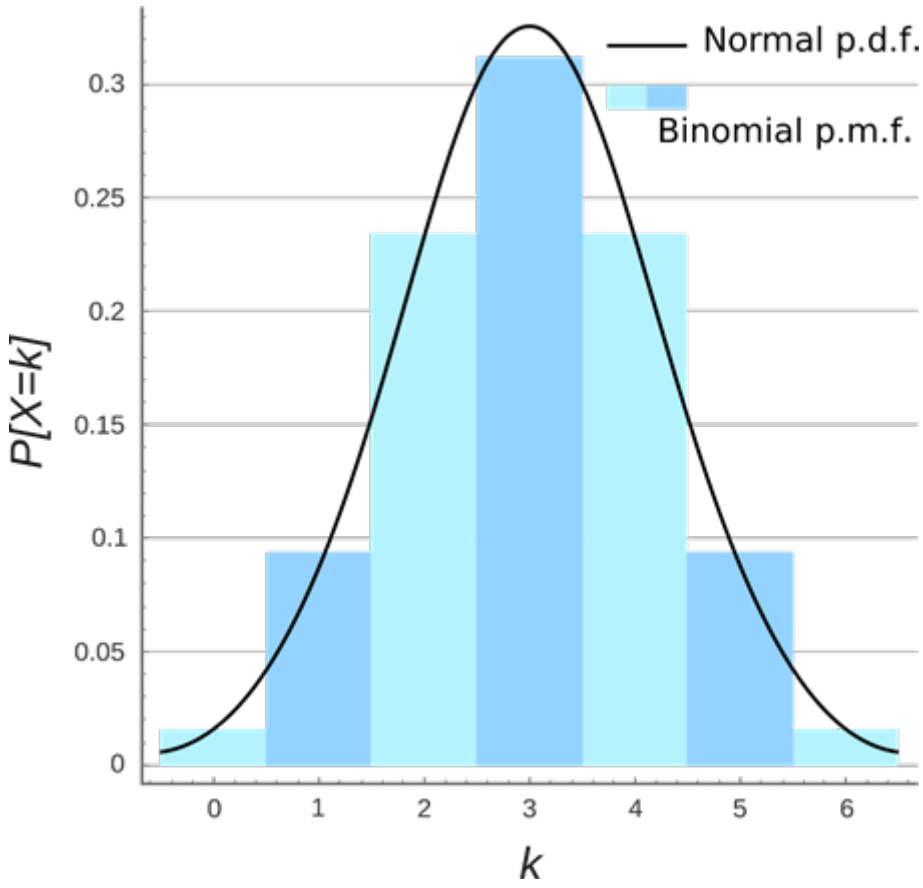
4.6 Probability Density Functions

Gives you the probability of a data point falling within some given range of a given value. For example, with a normal distribution



4.7 Probability Mass Functions

This is used for discrete data sets. You have exact bins something will occur in a data set.

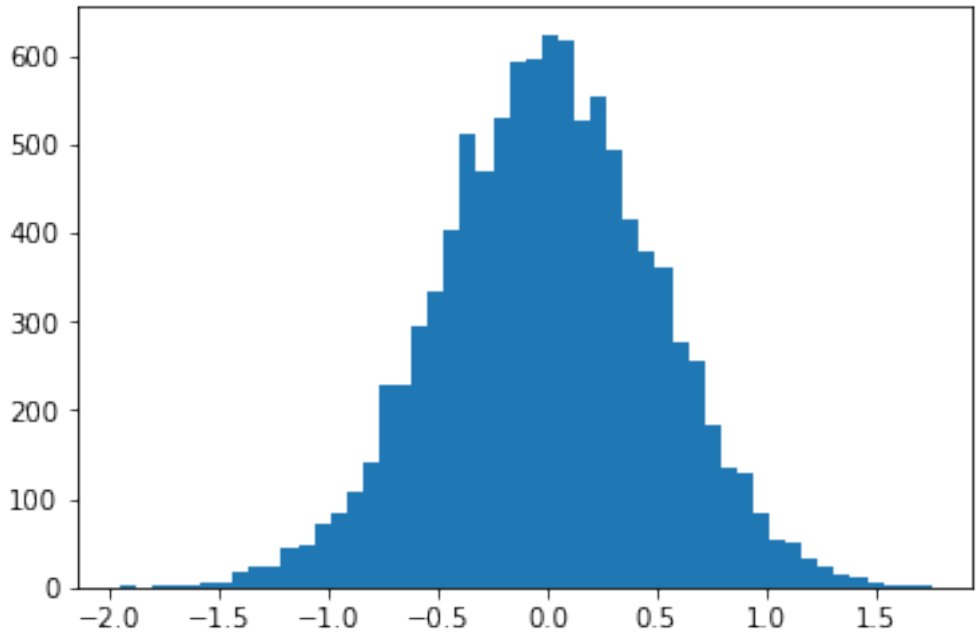


4.8 Percentiles and Moments

In a data set, what's the point at which X% of the values are less than that value? Quartiles contain 25% of a data set. IQR contains the middle 20%.

This example produces normally distributed random data.

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

vals = np.random.normal(0, 0.5, 10000)

plt.hist(vals, 50)
plt.show()
```



This is used to give you the 50th percentile. This is the same thing as the **median**

[2]:
```
np.percentile(vals, 50)
```

[2]: 0.0021422057356575478

[3]:
```
np.percentile(vals, 90)
```

[3]: 0.64070301059941248

[4]:
```
np.percentile(vals, 20)
```

[4]: -0.41207346812009299

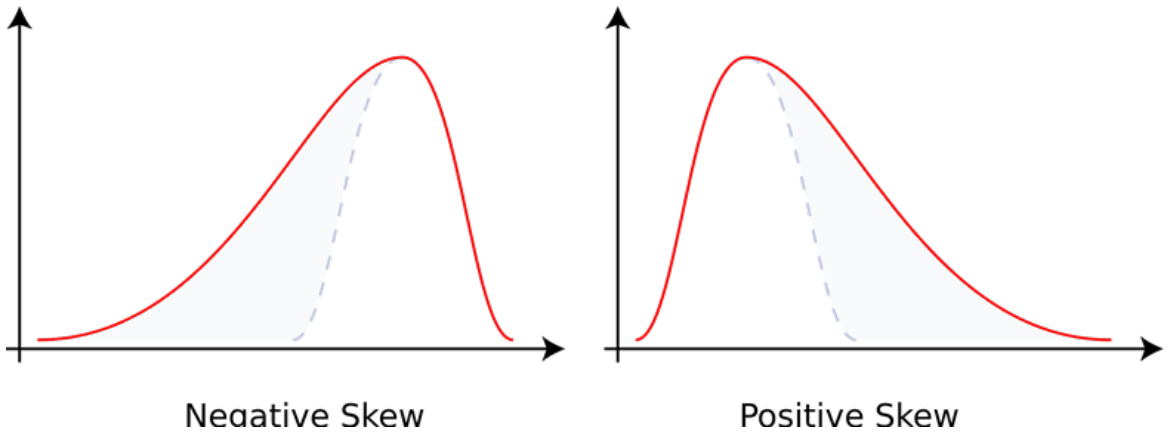
4.9 Moments: Mean, Variance, Skew, Kurtosis

Moments are a quantitative measures of the shape of a probability density function. Mathematically:

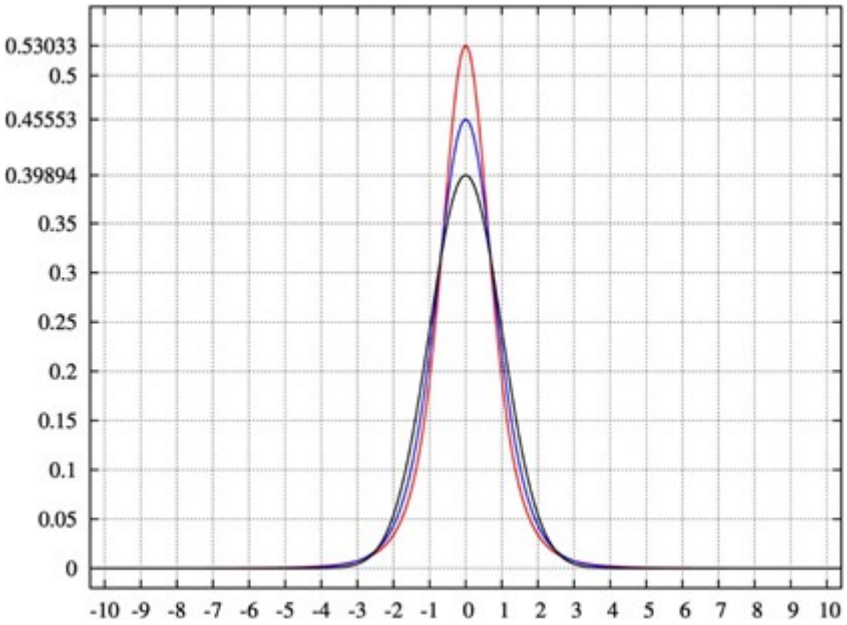
$$\mu_n = \int_{-\infty}^{\infty} (x - c)^n f(x)\, dx$$

But intuitively, it's a lot simpler in statistics. The first moment is the mean. The second moment is the variance. The third moment is "skew" (γ)

This tells you how "lopsided" the distribution is. A distribution with a longer tail on the left will be skewed left, and have a negative skew.



The fourth moment is “kurtosis”. How thick is the tail, and how sharp is the peak, compared to a normal distribution? Example: higher peaks have higher kurtosis

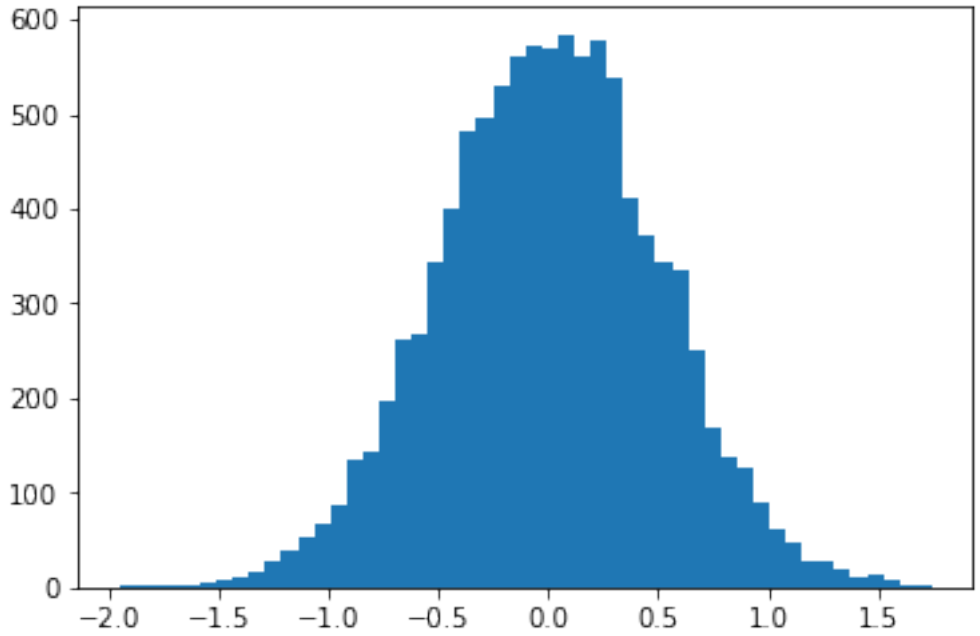


4.9.1 Computation of Moments in Python

These 4 moments can be computed in python.

Create a roughly normal-distributed random set of data:

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

vals = np.random.normal(0, 0.5, 10000)

plt.hist(vals, 50)
plt.show()
```



The first moment is the mean; this data should average out to about 0:

[2]:
```
np.mean(vals)
```

[2]:
```
0.0030761613438204053
```

The second moment is the variance:

[3]:
```
np.var(vals)
```

[3]:
```
0.24962106559173047
```

The third moment is skew - since our data is nicely centered around 0, it should be almost 0:

[4]:
```
import scipy.stats as sp
sp.skew(vals)
```

[4]:
```
-0.018249260513551597
```

The fourth moment is “kurtosis”, which describes the shape of the tail. For a normal distribution, this is 0:

[5]:
```
sp.kurtosis(vals)
```

[5]:
```
0.04795034855803815
```

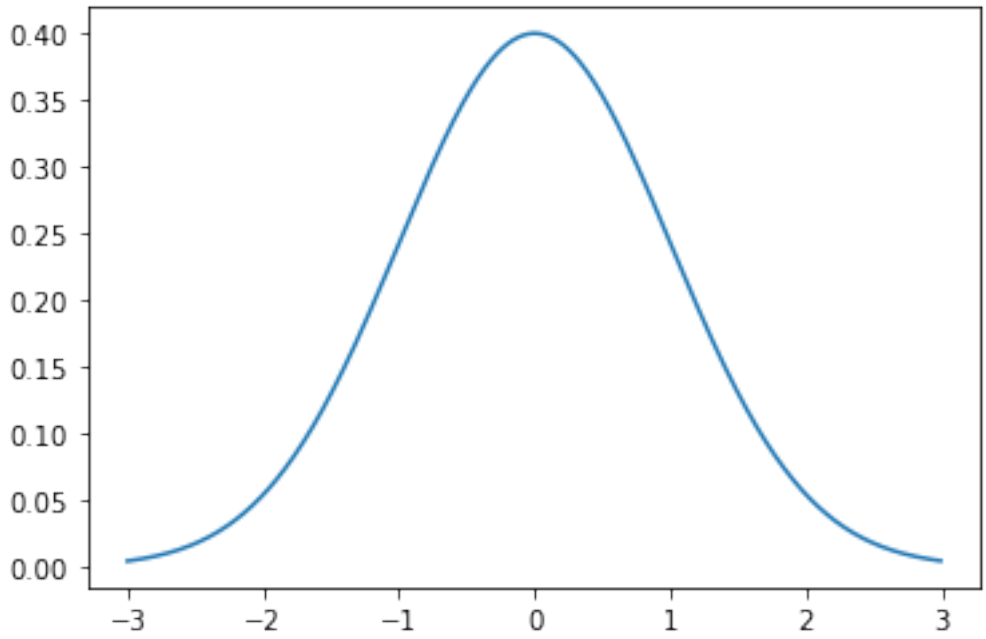
The kurtosis and skew are near 0 here because we are just looking at a normal distribution.

4.10 MatPlotLib Basics

Matplotlib is a library you can use in Python to make pretty graphs.

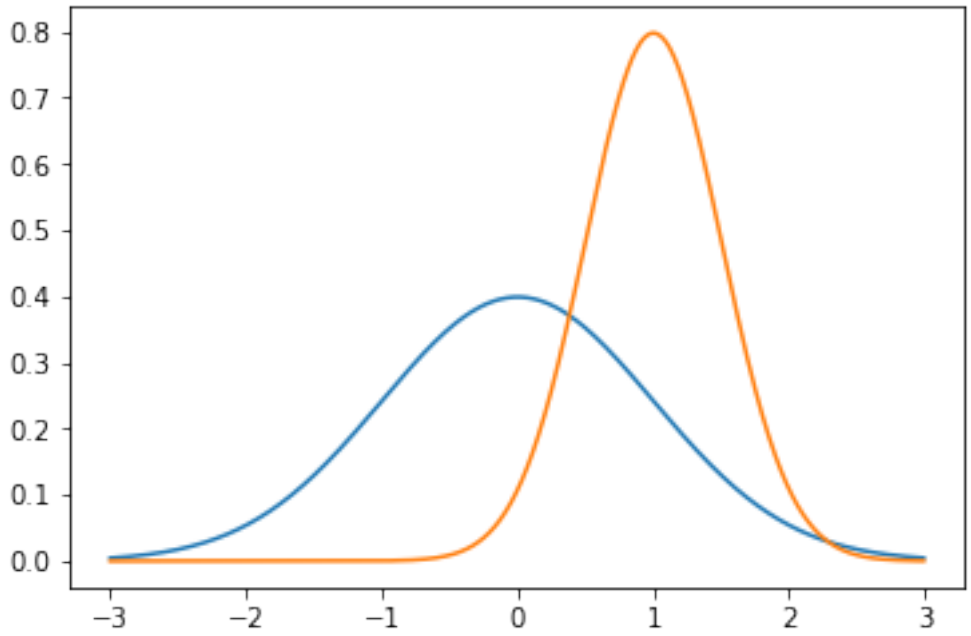
4.10.1 Drawing a line graph

[4]:
```
%matplotlib inline
from scipy.stats import norm
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-3, 3, 0.01) #a-range gives you all the numbers in an interval with a␣
 ↪specifed interval between

plt.plot(x, norm.pdf(x)) # y function is a normal distribution based on the x␣
 ↪values
plt.show()
```



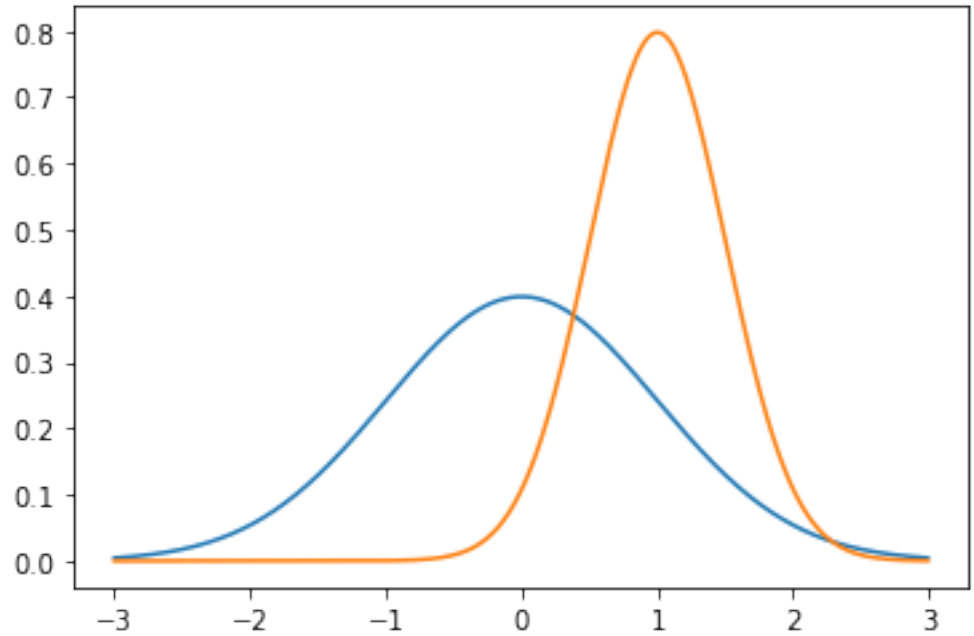
4.10.2 Mutiple Plots on One Graph

[2]:
```
plt.plot(x, norm.pdf(x))
plt.plot(x, norm.pdf(x, 1.0, 0.5)) # another nomral distribution of mean of 1 and␣
 ↪stdev of 0.5
plt.show()
```



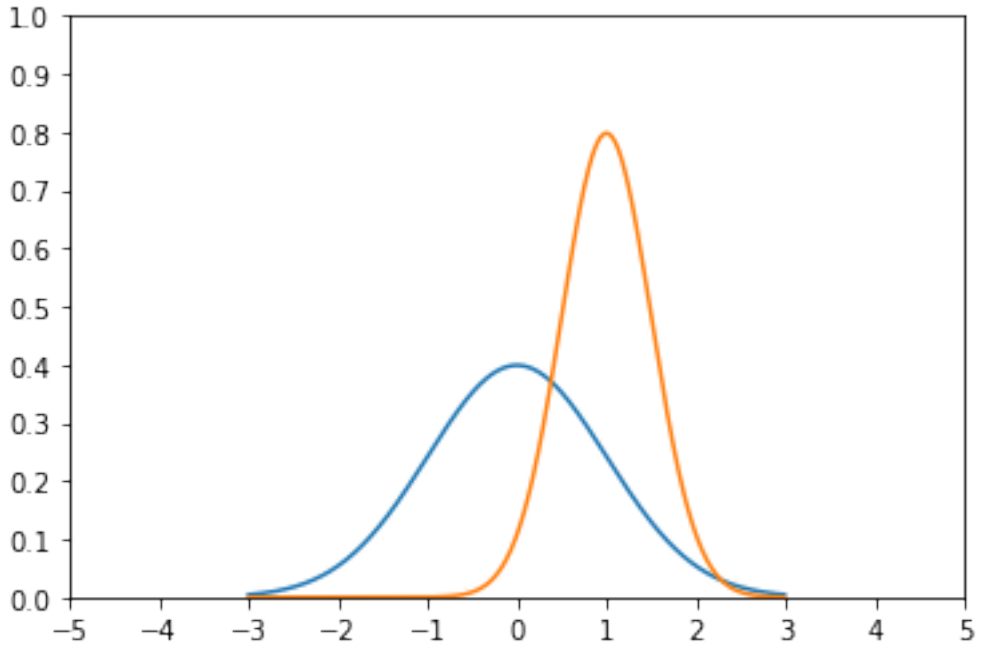
4.10.3 Save it to a File

[2]:
```
plt.plot(x, norm.pdf(x))
plt.plot(x, norm.pdf(x, 1.0, 0.5))
plt.savefig('C:\\Users\\Shav\\MyPlot.png', format='png')
```



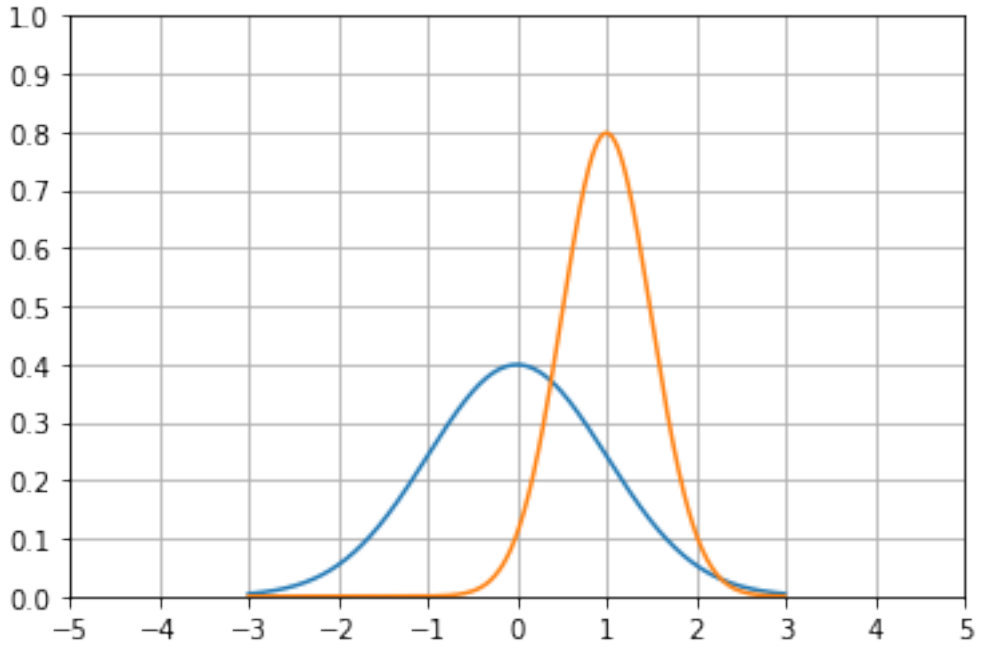
4.10.4 Adjust the Axes

[4]:
```
axes = plt.axes() #once you have axes as objects you can adjust them
axes.set_xlim([-5, 5])
axes.set_ylim([0, 1.0])
axes.set_xticks([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
axes.set_yticks([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]) #could use␣
 ↪the a-range function to make this easier
plt.plot(x, norm.pdf(x))
plt.plot(x, norm.pdf(x, 1.0, 0.5))
plt.show()
```



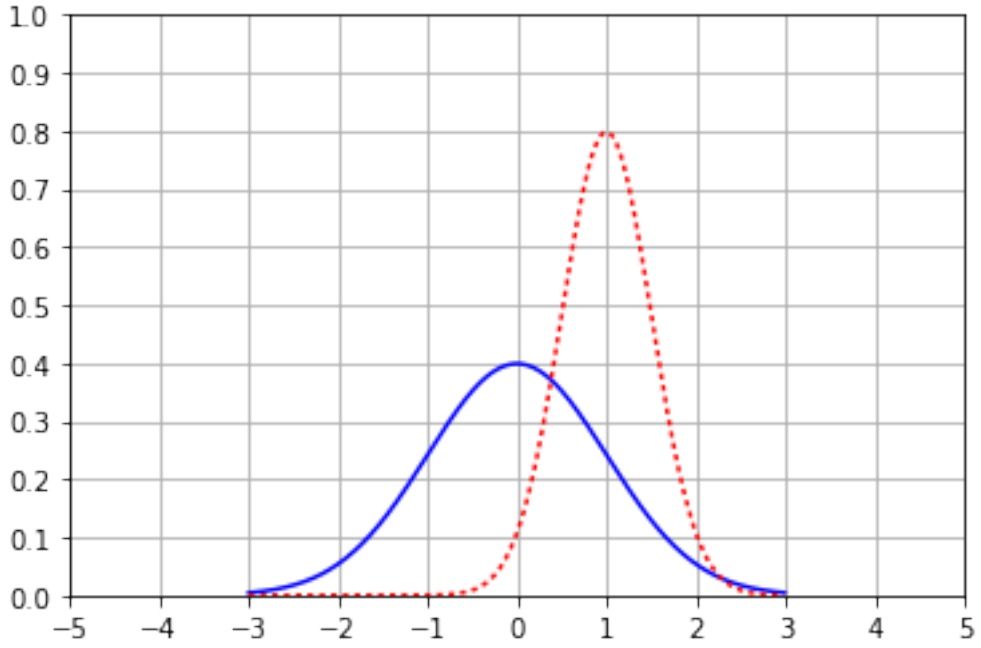
4.10.5 Add a Grid

[5]:
```
axes = plt.axes()
axes.set_xlim([-5, 5])
axes.set_ylim([0, 1.0])
axes.set_xticks([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
axes.set_yticks([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
axes.grid()
plt.plot(x, norm.pdf(x))
plt.plot(x, norm.pdf(x, 1.0, 0.5))
plt.show()
```



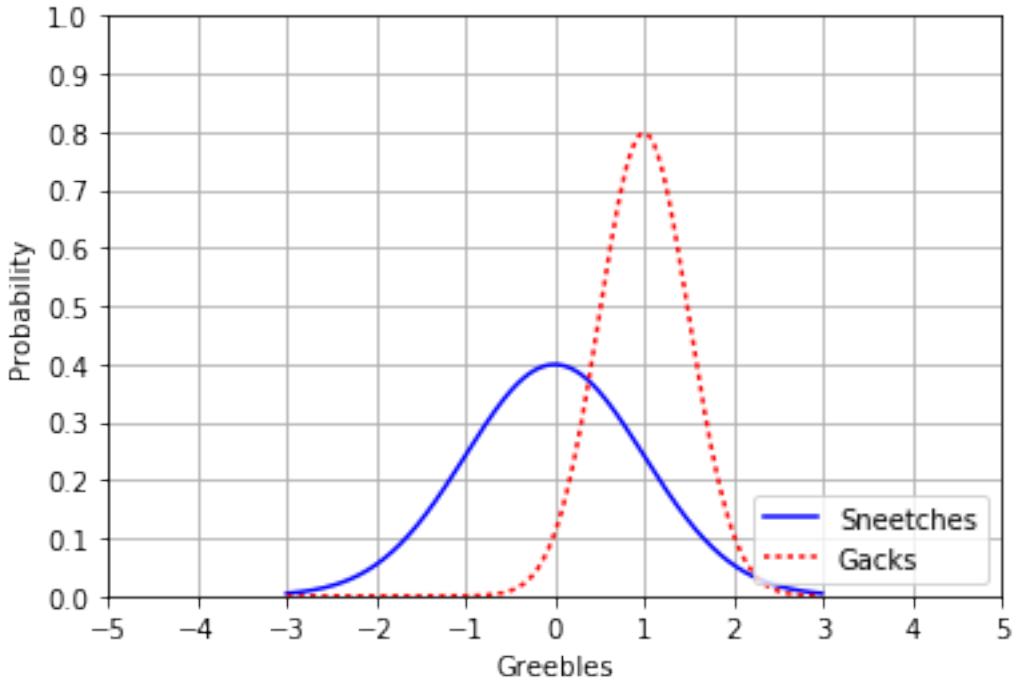
4.10.6 Change Line Types and Colors

[6]:
```
axes = plt.axes()
axes.set_xlim([-5, 5])
axes.set_ylim([0, 1.0])
axes.set_xticks([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
axes.set_yticks([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
axes.grid()
plt.plot(x, norm.pdf(x), 'b-') #blue solid line
plt.plot(x, norm.pdf(x, 1.0, 0.5), 'r:')
plt.show()
```



4.10.7 Labeling Axes and Adding a Legend

[7]:
```
axes = plt.axes()
axes.set_xlim([-5, 5])
axes.set_ylim([0, 1.0])
axes.set_xticks([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
axes.set_yticks([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
axes.grid()
plt.xlabel('Greebles')
plt.ylabel('Probability')
plt.plot(x, norm.pdf(x), 'b-')
plt.plot(x, norm.pdf(x, 1.0, 0.5), 'r:')
plt.legend(['Sneetches', 'Gacks'], loc=4) #loc is the location
plt.show()
```

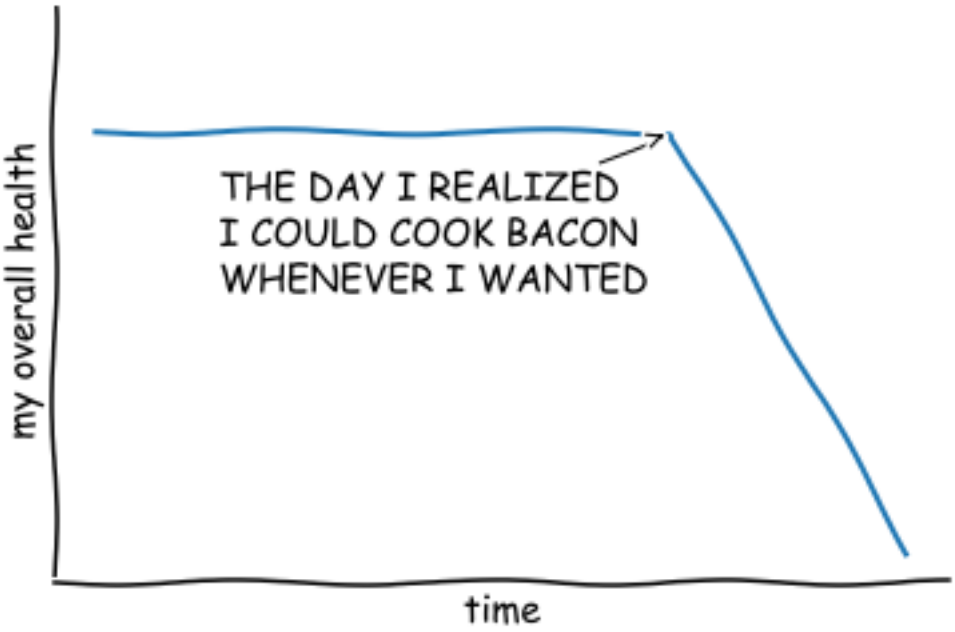


4.10.8 XKCD Style :)

[8]:
```
plt.xkcd() #bit of an easter egg

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
plt.xticks([])
plt.yticks([])
ax.set_ylim([-30, 10])
```

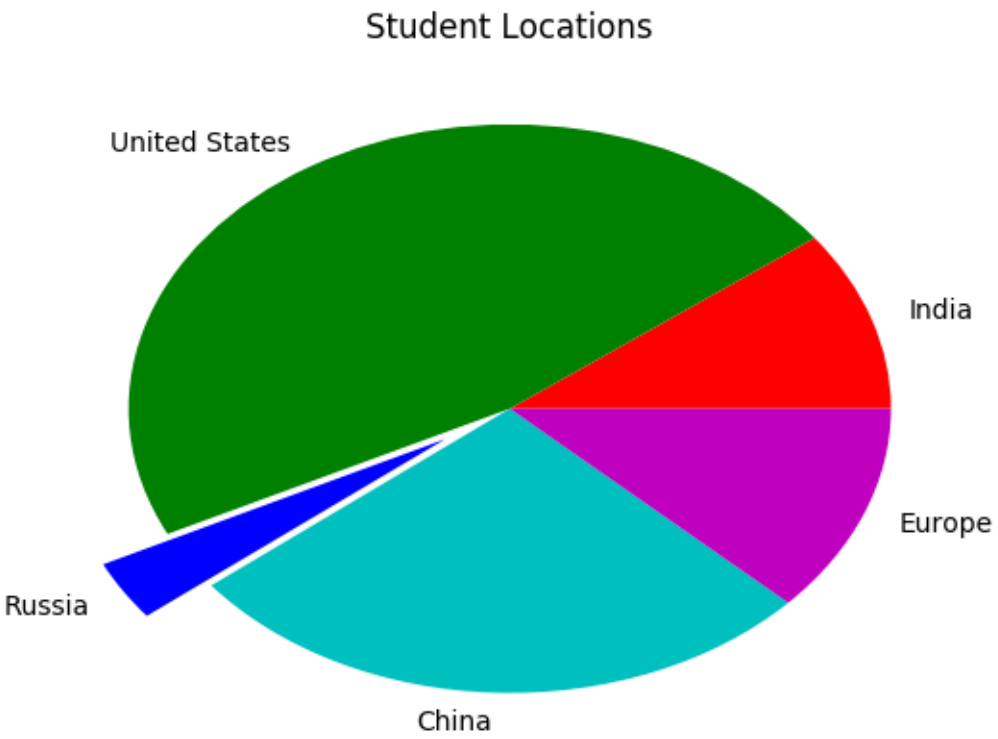
```
data = np.ones(100)
data[70:] -= np.arange(30) #intersting use of what we have learnt.

plt.annotate(
    'THE DAY I REALIZED\nI COULD COOK BACON\nWHENEVER I WANTED',
    xy=(70, 1), arrowprops=dict(arrowstyle='->'), xytext=(15, -10))

plt.plot(data)

plt.xlabel('time')
plt.ylabel('my overall health')
```

[8]: <matplotlib.text.Text at 0x28fdde72828>



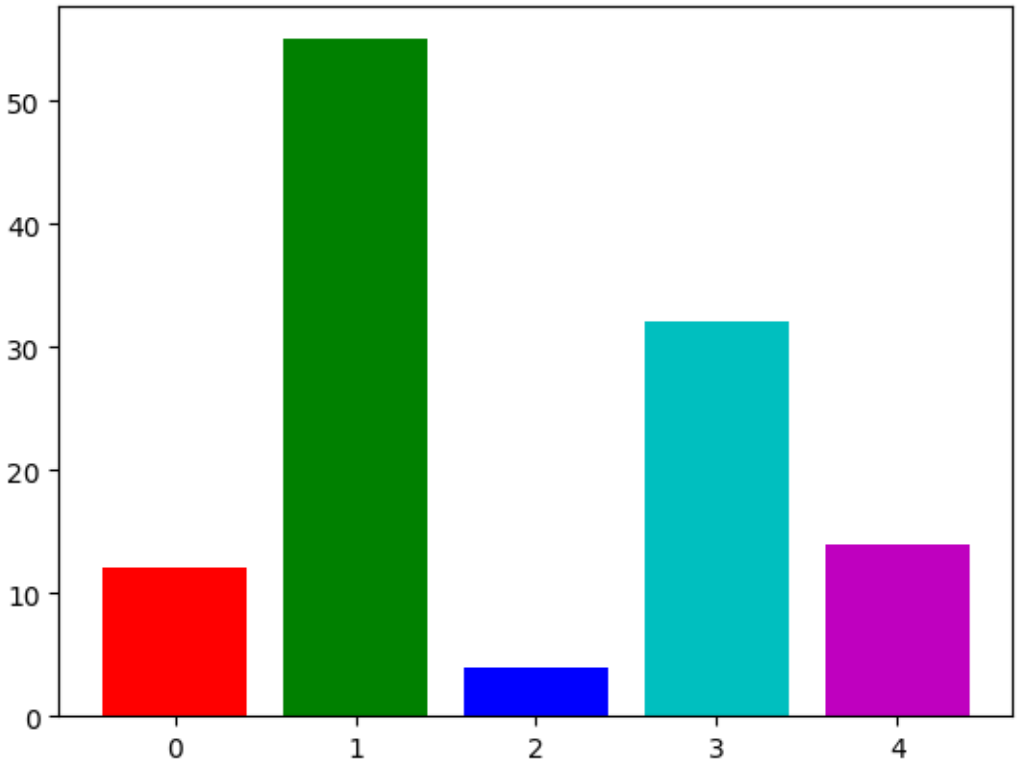
4.10.9 Pie Chart

[9]:
```
# Remove XKCD mode:
plt.rcdefaults()

values = [12, 55, 4, 32, 14]
colors = ['r', 'g', 'b', 'c', 'm']
explode = [0, 0, 0.2, 0, 0]
labels = ['India', 'United States', 'Russia', 'China', 'Europe']
plt.pie(values, colors= colors, labels=labels, explode = explode)
plt.title('Student Locations')
plt.show()
```



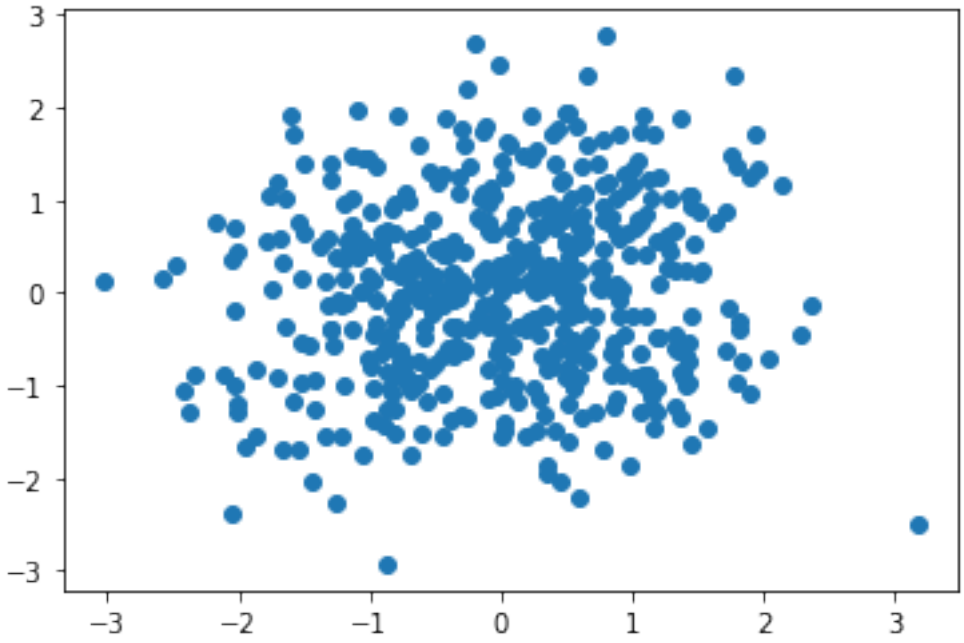
4.10.10 Bar Chart

[10]:
```
values = [12, 55, 4, 32, 14]
colors = ['r', 'g', 'b', 'c', 'm']
plt.bar(range(0,5), values, color= colors)
plt.show()
```



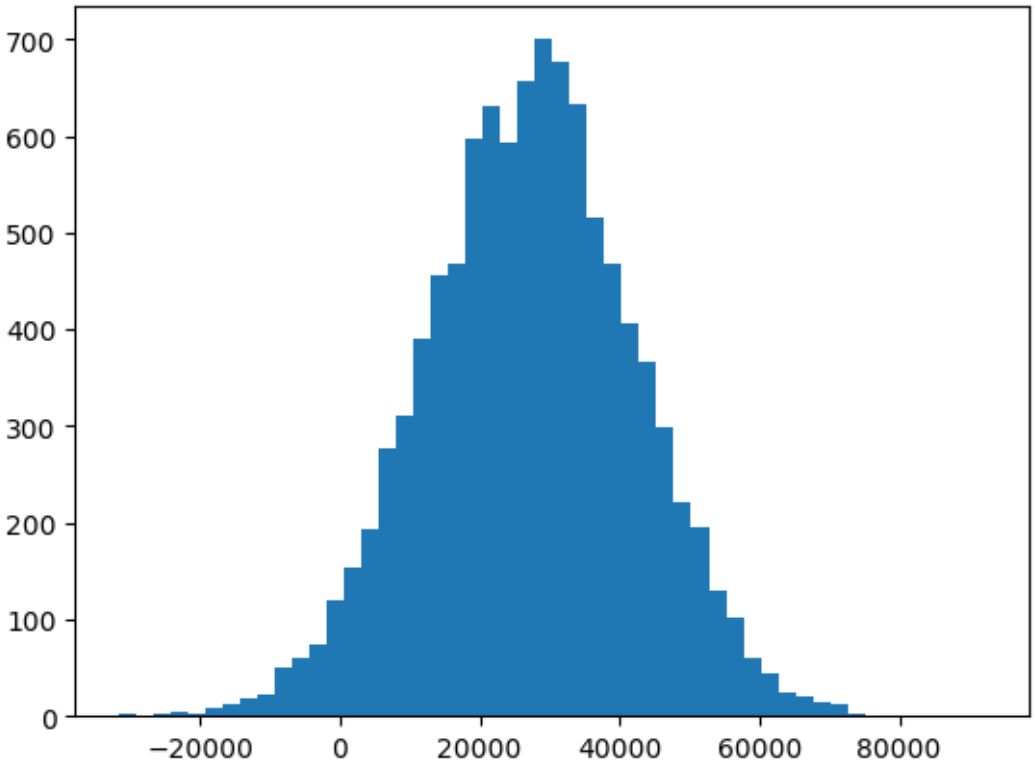
4.10.11 Scatter Plot

[9]:
```
from pylab import randn

X = randn(500)
Y = randn(500)
plt.scatter(X,Y)
plt.show()
```



4.10.12 Histogram

[12]:
```
incomes = np.random.normal(27000, 15000, 10000)
plt.hist(incomes, 50)
plt.show()
```



4.10.13 Box & Whisker Plot

Useful for visualizing the spread & skew of data.

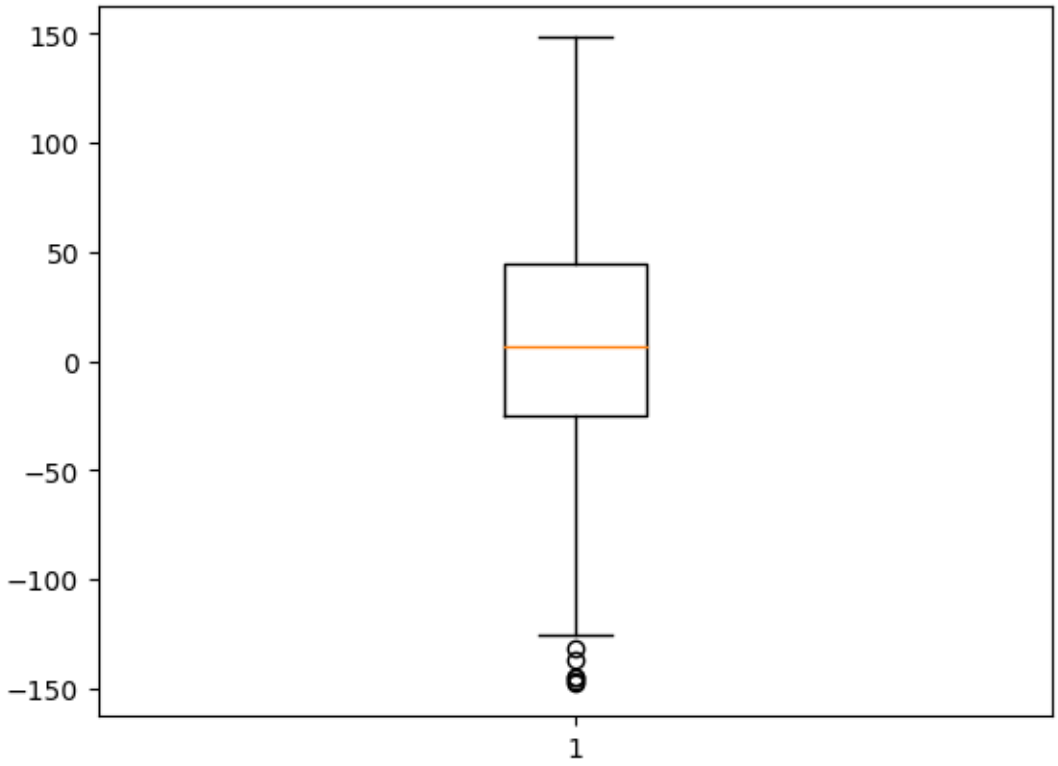
The red line represents the median of the data, and the box represents the bounds of the 1st and 3rd quartiles.

So, half of the data exists within the box.

The dotted-line “whiskers” indicate the range of the data - except for outliers, which are plotted outside the whiskers. Outliers are 1.5X or more the interquartile range.

This example below creates uniformly distributed random numbers between -40 and 60, plus a few outliers above 100 and below -100:

[13]:
```
uniformSkewed = np.random.rand(100) * 100 - 40
high_outliers = np.random.rand(10) * 50 + 100
low_outliers = np.random.rand(10) * -50 - 100
data = np.concatenate((uniformSkewed, high_outliers, low_outliers))
plt.boxplot(data)
plt.show()
```



4.11 Covariance and Correlation

Covariance measures how two variables vary in tandem from their means i.e is there a correlation?

how do we measure it? We think of the data sets for the two variables as high-dimensional vectors and convert these to vectors of variances from the mean. We then. We then take the dot product (cosine of the angle between them) of the two vectors and divide by the sample size

For example, let's say we work for an e-commerce company, and they are interested in finding a correlation between page speed (how fast each web page renders for a customer) and how much a customer spends. If dot product is small, these vectors are moving together i.e correlated.

Interpreting covariance is hard. We know a small covariance, close to 0, means there isn't much correlation between the two variables and thay large covariances – that is, far from 0 (could be negative for inverse relationships), mean there is a correlation. But how large is "large"?

That's where correlation comes in! Just divide the covariance by the standard deviations of both variables, and that normalizes things.

- correlation of -1 means a perfect inverse correlation
- Correlation of 0: no correlation
- Correlation 1: perfect correlation

Remember though, correlation does **not** imply causation. Only a controlled, randomized experiment can give you insights on causation. Use correlation to decide what experiments to conduct! E.G page load times and money spent, could be due to people with high speed internet have more money to spend.

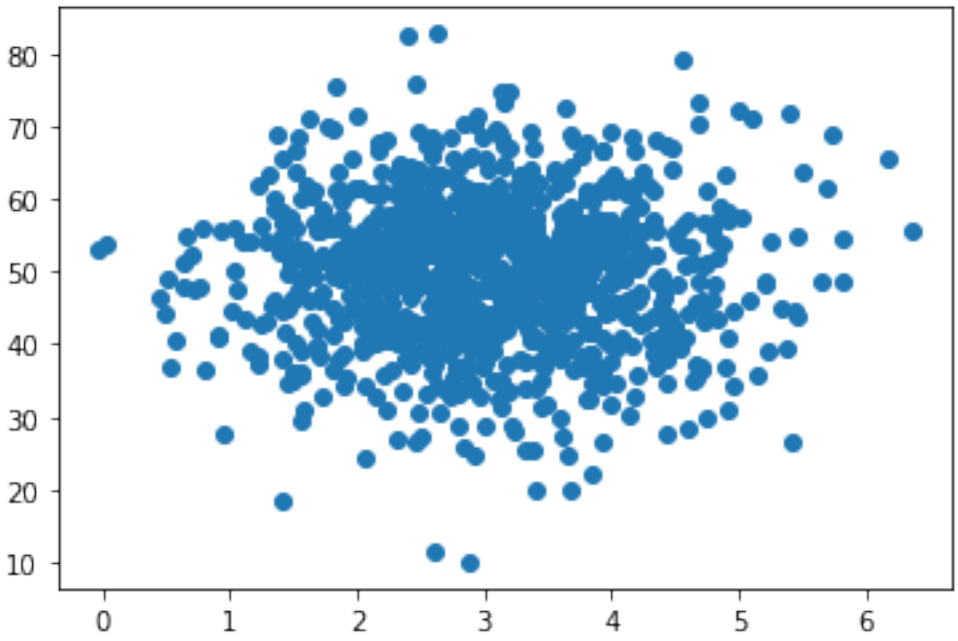
4.12 Covariance in Numpy

numpy offers covariance methods, but we'll do it the "hard way" to show what happens under the hood. Basically we treat each variable as a vector of deviations from the mean, and compute the "dot product" of both vectors. Geometrically this can be thought of as the angle between the two vectors in a high-dimensional space, but you can just think of it as a measure of similarity between the two variables.

First, let's just make page speed and purchase amount totally random and independent of each other; a very small covariance will result as there is no real correlation:

[9]:
```
%matplotlib inline

import numpy as np
from pylab import *

#defining our own functions

def de_mean(x):
    xmean = mean(x)
    return [xi - xmean for xi in x] #xi is each data point in the list

def covariance(x, y):
    n = len(x)
    return dot(de_mean(x), de_mean(y)) / (n-1) #this is the sample rememebr so -1

pageSpeeds = np.random.normal(3.0, 1.0, 1000)
purchaseAmount = np.random.normal(50.0, 10.0, 1000)

scatter(pageSpeeds, purchaseAmount)

covariance (pageSpeeds, purchaseAmount)
```

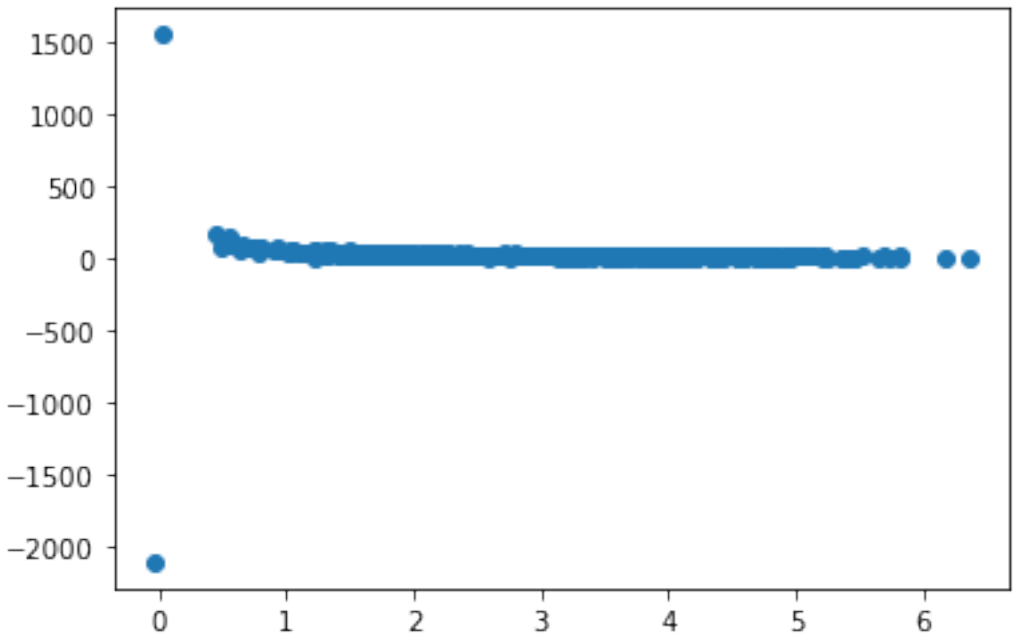
[9]:
```
-0.000709281062864929
```



Now we'll make our fabricated purchase amounts an actual function of page speed, making a very real correlation. The negative value indicates an inverse relationship; pages that render in less time result in more money spent:

[57]:
```
purchaseAmount = np.random.normal(50.0, 10.0, 1000) / pageSpeeds

scatter(pageSpeeds, purchaseAmount)

covariance (pageSpeeds, purchaseAmount)
```

[57]:
```
-6.262497637596617
```



But, what does this value mean? Covariance is sensitive to the units used in the variables, which makes it difficult to interpret. Correlation normalizes everything by their standard deviations, giving you an easier to understand value that ranges from -1 (for a perfect inverse correlation) to 1 (for a perfect positive correlation):

[46]:
```
def correlation(x, y):
    stddevx = x.std()
    stddevy = y.std()
    return covariance(x,y) / stddevx / stddevy  #In real life you'd check for␣
↪divide by zero here

correlation(pageSpeeds, purchaseAmount)
```

[46]:
```
-0.1426360840460116
```

numpy can do all this for you with numpy.corrcoef. It returns a matrix of the correlation coefficients between every combination of the arrays passed in:

[47]:
```
np.corrcoef(pageSpeeds, purchaseAmount)
```

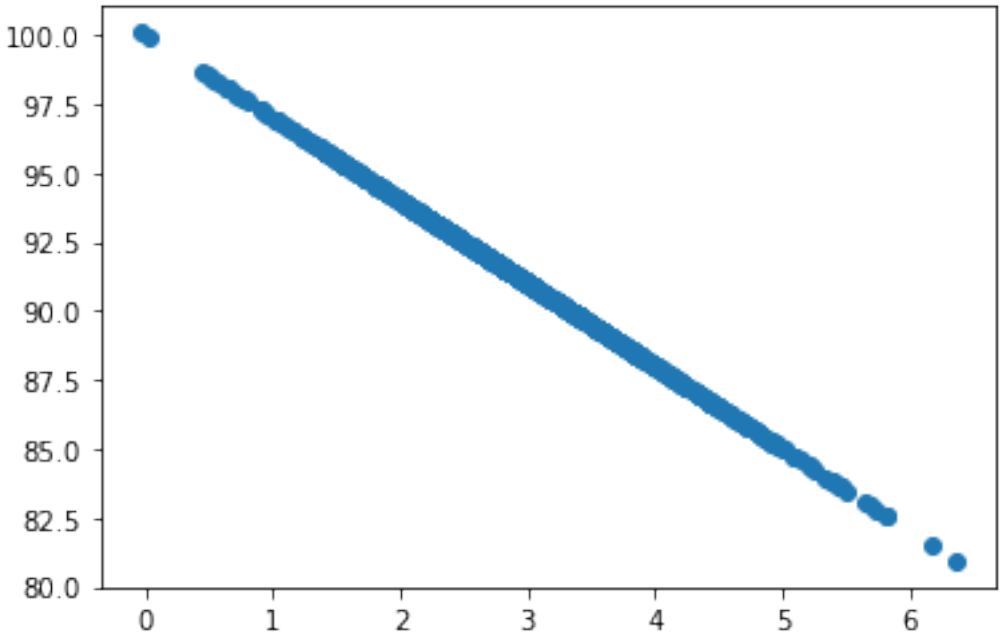
[47]:
```
array([[ 1.        , -0.14249345],
       [-0.14249345,  1.        ]])
```

(It doesn't match exactly just due to the math precision available on a computer.) To read this, the top left value is the correlation between page speed and page speed, so 1, and the top right is purchas amount and page speeds etc.

We can force a perfect correlation by fabricating a totally linear relationship (again, it's not exactly -1 just due to precision errors, but it's close enough to tell us there's a really good correlation here):

[53]:
```
purchaseAmount = 100 - pageSpeeds * 3

scatter(pageSpeeds, purchaseAmount)

correlation (pageSpeeds, purchaseAmount)
```

[53]:
```
-1.001001001001001
```



Remember, correlation does not imply causality!

numpy also has a numpy.cov function that can compute Covariance for you. Try using it for the pageSpeeds and purchaseAmounts data above. Interpret its results, and compare it to the results from our own covariance function above.

[58]:
```
import numpy as np
np.cov(pageSpeeds, purchaseAmount)
```

[58]:
```
array([[ 9.90812794e-01, -6.26249764e+00],
       [-6.26249764e+00,  6.99810064e+03]])
```

The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i.

4.13 Conditional Probability

This is the probablitiy of something occuring giving smething else occured that it depends on. In real life, e.g amazon's "people who bought this also bought this" or "people who viewed this also viewed". Conditional Probability is asking "if I have two events that depend on each other, what's the probability that both will occur?". Notation is as P(A,B). This is the probability of A and B both occurring. P(B|A) is the probability of B given that A has occurred. We also know:

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

For example: I give my students two tests. 60% of my students passed both tests, but the first test was easier – 80% passed that one. What percentage of students who passed the first test also passed the second?

- A = passing the first test
- B = passing the second test
- we are asking for P(B|A) – the probability of B given A

Using the above equation, 0.6/0.8 = 0.75 so 75% of students who passed the first test passed the second.

4.13.1 Conditional Probability Activity & Exercise

Below is some code to create some fake data on how much stuff people purchase given their age range.

It generates 100,000 random "people" and randomly assigns them as being in their 20's, 30's, 40's, 50's, 60's, or 70's.

It then assigns a lower probability for young people to buy stuff.

In the end, we have two Python dictionaries:

"totals" contains the total number of people in each age group. "purchases" contains the total number of things purchased by people in each age group. The grand total of purchases is in totalPurchases, and we know the total number of people is 100,000.

Let's run it and have a look:

[4]:
```
from numpy import random
random.seed(0) #initialise randomizer

totals = {20:0, 30:0, 40:0, 50:0, 60:0, 70:0}
purchases = {20:0, 30:0, 40:0, 50:0, 60:0, 70:0}
totalPurchases = 0
for _ in range(100000): #generate 100,000 people
    ageDecade = random.choice([20, 30, 40, 50, 60, 70]) #assign to a different age
    purchaseProbability = float(ageDecade) / 100.0
    totals[ageDecade] += 1
    if (random.random() < purchaseProbability): #radomly decide if they bought or␣
 ↪not.
        totalPurchases += 1
        purchases[ageDecade] += 1
```

[5]:
```
totals
```

[5]:
```
{20: 16576, 30: 16619, 40: 16632, 50: 16805, 60: 16664, 70: 16704}
```

Number of people in each category

[6]:
```
purchases
```

[6]:
```
{20: 3392, 30: 4974, 40: 6670, 50: 8319, 60: 9944, 70: 11713}
```

[7]:
```
totalPurchases
```

[7]:
```
45012
```

Let's play with conditional probability.

First let's compute P(E|F), where E is "purchase" and F is "you're in your 30's". The probability of someone in their 30's buying something is just the percentage of how many 30-year-olds bought something:

[8]:
```
PEF = float(purchases[30]) / float(totals[30])
print('P(purchase | 30s): ' + str(PEF))
```

```
P(purchase | 30s): 0.29929598652145134
```

P(F) is just the probability of being 30 in this data set:

[9]:
```
PF = float(totals[30]) / 100000.0
print("P(30's): " +  str(PF))
```

```
P(30's): 0.16619
```

And P(E) is the overall probability of buying something, regardless of your age:

[10]:
```
PE = float(totalPurchases) / 100000.0
print("P(Purchase):" + str(PE))
```

```
P(Purchase):0.45012
```

If E and F were independent, then we would expect P(E | F) to be about the same as P(E). But they're not; PE is 0.45, and P(E|F) is 0.3. So, that tells us that E and F are dependent (which we know they are in this example.). i.e probs some dependence

What is P(E)P(F)?

[11]:
```
print("P(30's)P(Purchase)" + str(PE * PF))
```

```
P(30's)P(Purchase)0.07480544280000001
```

P(E,F) is different from P(E|F). P(E,F) would be the probability of both being in your 30's and buying something, out of the total population - not just the population of people in their 30's:

[12]:
```
print("P(30's, Purchase)" + str(float(purchases[30]) / 100000.0))
```

```
P(30's, Purchase)0.04974
```

P(E,F) = P(E)P(F) is **only true if there is an independednt relationship** i.e Two events A and B are called independent if P(A|B)=P(A). They are pretty close in this example. But because E and F are actually dependent on each other, and the randomness of the data we're working with, it's not quite the same.

We can also check that P(E|F) = P(E,F)/P(F) and sure enough, it is:

[13]:
```
print((purchases[30] / 100000.0) / PF)
```

```
0.29929598652145134
```

4.13.2 Looking at Independence

Here the code has been modified such that the purchase probability does NOT vary with age, making E and F actually independent.

Then, it has been confirm that P(E|F) is about the same as P(E), showing that the conditional probability of purchase for a given age is not any different than the a-priori probability of purchase regardless of age.

First we'll modify the code to have some fixed purchase probability regardless of age, say 40%:

[1]:
```
from numpy import random
random.seed(0)

totals = {20:0, 30:0, 40:0, 50:0, 60:0, 70:0}
purchases = {20:0, 30:0, 40:0, 50:0, 60:0, 70:0}
totalPurchases = 0
for _ in range(100000):
    ageDecade = random.choice([20, 30, 40, 50, 60, 70])
    purchaseProbability = 0.4
    totals[ageDecade] += 1
    if (random.random() < purchaseProbability):
        totalPurchases += 1
        purchases[ageDecade] += 1
```

Next we will compute P(E|F) for some age group, let's pick 30 year olds again:

[2]:
```
PEF = float(purchases[30]) / float(totals[30])
print("P(purchase | 30s): " + str(PEF))
```

```
P(purchase | 30s): 0.3987604549010169
```

Now we'll compute P(E)

[3]:
```
PE = float(totalPurchases) / 100000.0
print("P(Purchase):" + str(PE))
```

```
P(Purchase):0.4003
```

P(E|F) is pretty darn close to P(E), so we can say that E and F are likely indepedent variables.

4.14 Bayes' Theorem

Now that you understand conditional probability, you can understand Bayes' Theorem:

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

In English – the probability of A given B, is the probability of A times the probability of B given A over the probability of B. The key insight is that the **probability of something that depends on B depends very much on the base probability of B and A**. People ignore this all the time.

Drug testing is a common example. Even a “highly accurate” drug test can produce more false positives than true positives.

You can say that the probability of a test detecting a user can be high, but the probability of being a user given that you tested positive can be very differnt. Bayes theorum allows you to quantify that differnece.

Let’s say we have a drug test that can accurately identify users of a drug 99% of the time, and accurately has a negative result for 99% of non- users. But only 0.3% of the overall population actually uses this drug in question.

Lets say Event A = Is a user of the drug, Event B = tested positively for the drug.

We can work out from that information that P(B) is 1.3% (0.99 * 0.003 + 0.01 * 0.997 – the probability of testing positive if you do use, plus the probability of testing positive if you don’t)

$$P(A|B) = \frac{P(A)P(A|B)}{P(B)} = \frac{0.003*0.99}{0.013} = 22.8\%$$

So the odds of someone being an actual user of the drug given that they tested positive is only 22.8%! Even though P(B|A) is high (99%), it doesn’t mean P(A|B) is high.

Always take these things into consideration. Probability of a drug test being accurate depends a lot on the actually probability of being a drug user in the population, not just the accuracy of the test.

5 Predictive Models

5.1 Regression Analysis

This involves fitting a function to some observations. You then use this function to predict unobserved values.

5.2 Linear Regression

Simplest form of regression. Fit a line to a data set of observations. We normally use the “least squares” method. This minimizes the squared-error between each point and the line.

Under the hood we y=mx+b. The slope is the correlation between the two variables times the standard deviation in Y, all divided by the standard deviation in X.

The intercept is the mean of Y minus the slope times the mean of X

This is the same as maximizing the likelihood of the observed data if you start thinking of the problem in terms of probabilities and probability distribution functions. This is sometimes called “maximum likelihood estimation”

There are more ways to do it. Gradient Descent is an alternate method to least squares. Basically, it iterates to find the line that best follows the contours defined by the data. This can make sense when dealing with 3D data. It is easy to try in Python and just compare the results to least squares. Usually least squares is a perfectly good choice.

5.3 Measuring error with R-squared

This is used to measure how well our line fits our data. R-squared (aka coefficient of determination) measures the fraction of the total variation in Y that is captured by the model. R-squared can be computed by:

$$1 - \frac{sum of squared errors}{sum of squared variation from mean}$$

Interpreting R-squared is simple. It ranges from 0 to 1. 0 is bad (none of the variance is captured), 1 is good (all of the variance is captured).

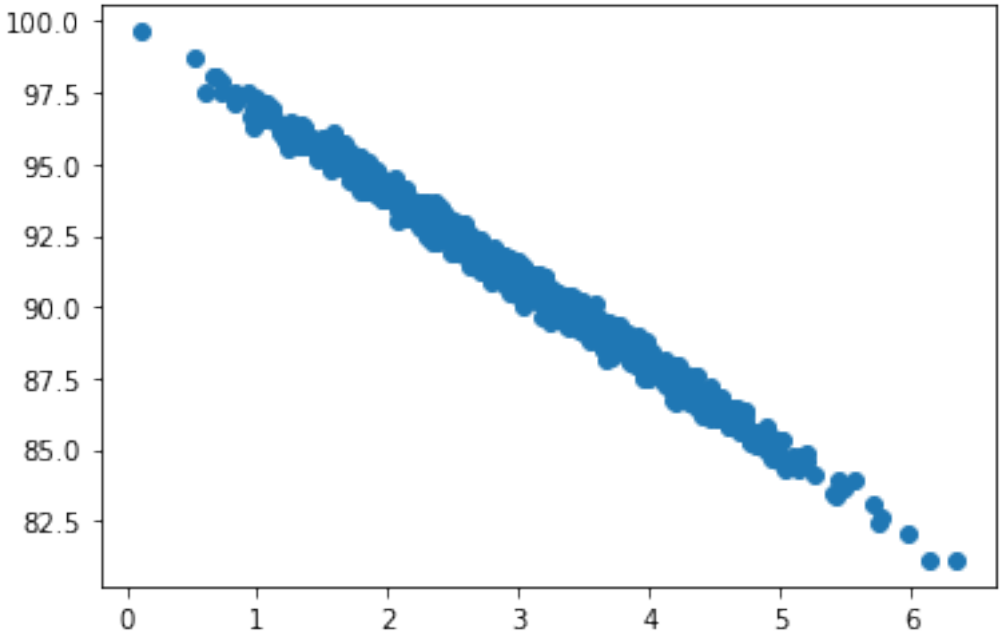
We will see various regrssion models, and r-squared is a good quantitaive measure of how good a model is.

Linear Regression in Python

Let's fabricate some data that shows a roughly linear relationship between page speed and amount purchased:

[4]:
```
%matplotlib inline
import numpy as np
from pylab import *

pageSpeeds = np.random.normal(3.0, 1.0, 1000)
purchaseAmount = 100 - (pageSpeeds + np.random.normal(0, 0.1, 1000)) * 3

#see that purchase amount is a linear function of that

scatter(pageSpeeds, purchaseAmount)
```

[4]:
```
<matplotlib.collections.PathCollection at 0x1adacf2c430>
```



As we only have two features, we can keep it simple and just use scipy.state.linregress:

[5]:
```
from scipy import stats

slope, intercept, r_value, p_value, std_err = stats.linregress(pageSpeeds,␣
 ↪purchaseAmount)
```

Not surprisngly, our R-squared value shows a really good fit:

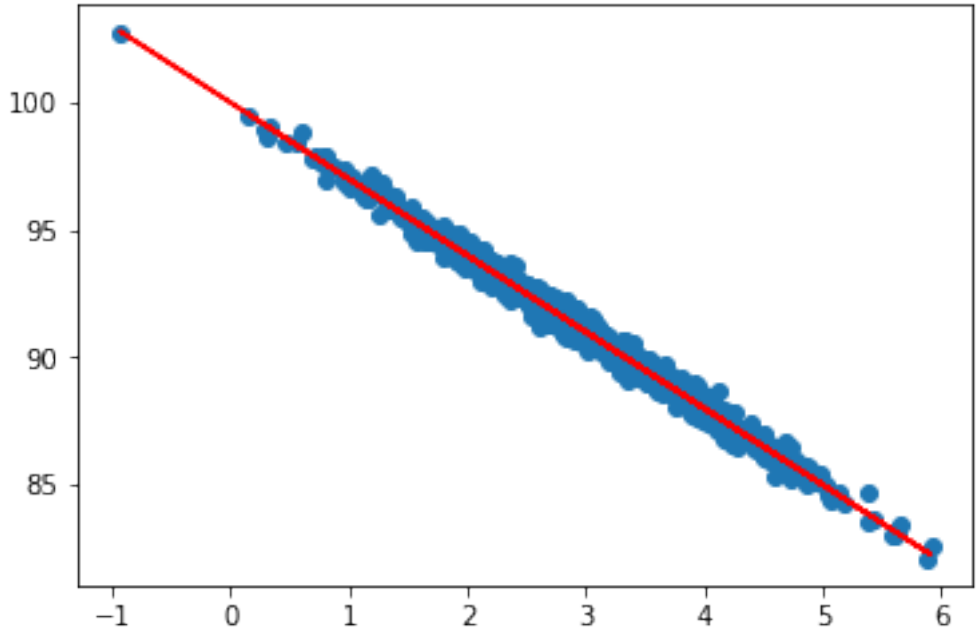
[6]:
```
r_value ** 2
```

[6]: 0.9895893375910493

This is a good model!

Let's use the slope and intercept we got from the regression to plot predicted values vs. observed:

[4]:
```
import matplotlib.pyplot as plt

de f predict(x):
    return slope * x + intercept

fitLine = predict(pageSpeeds)

plt.scatter(pageSpeeds, purchaseAmount)
plt.plot(pageSpeeds, fitLine, c='r')
plt.show()
```



5.4 Polynomial Regression

What if your data doesn't look linear at all? Why limit ourselves to straight lines? Linear formula: $y = mx + b$. This is a "first order" or "first degree" polynomial, as the power of x is 1. Second order polynomial: $y = ax^2 + bx + c$, third order: $y = ax^3 + bx^2 + cx + d$. Higher orders produce more complex curves.

Be aware though, **more degrees isnt always better**. Beware overfitting. Don't use more degrees than you need. Ther is usually a natural relationship in your data that isnt really all that complicated.

If you have a lot of data points, with a lot a varaince, you can easily fall into the trap of using a curve with a high degree to fit the varaince. However, that probably doesnt represent the intrinsic relationship of the data, and thus, predicting new values.

Visualize your data first to see how complex of a curve there might really be. Visualize the fit – is your curve going out of its way to accommodate outliers?

You can always compute the r-squared value to check the fit. A high r-squared simply means your curve fits your training data well; but it may not be a good predictor. This is separate from the abiity to accurately predict data going forward.

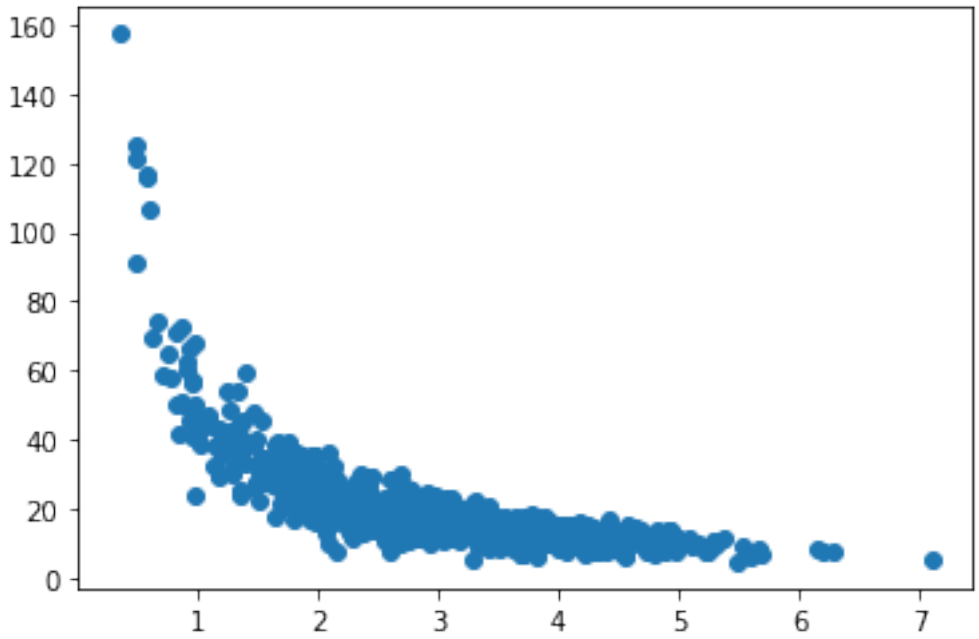
Later we'll talk about more principled ways to detect overfitting (train/test).

5.4.1 Polynomial Regression using NumPy

numpy.polyfit() makes it easy. Let's look at some more realistic-looking page speed / purchase data:

[32]:
```
%matplotlib inline
from pylab import *
import numpy as np

np.random.seed(2) #creates a rondom seed value. Means subsequent random operations␣
 ↪will be deterministic
#this means every time we run this, we get the same result. So we can compare!
#i.e start with same initial points
pageSpeeds = np.random.normal(3.0, 1.0, 1000)
purchaseAmount = np.random.normal(50.0, 10.0, 1000) / pageSpeeds

scatter(pageSpeeds, purchaseAmount)
```

[32]:
```
<matplotlib.collections.PathCollection at 0x1f801d2afa0>
```

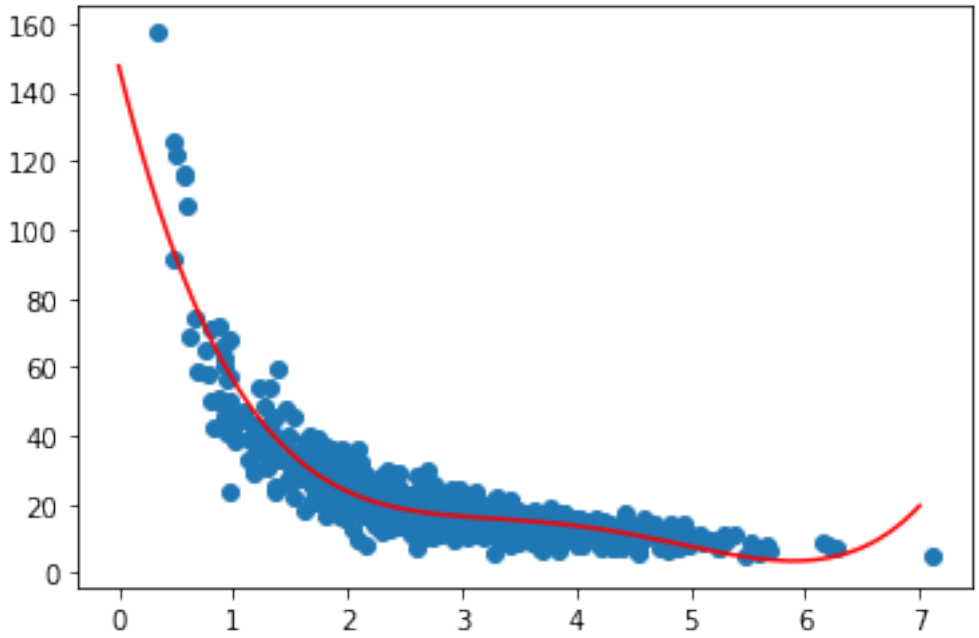


numpy has a handy **polyfit function** we can use, to let us construct an nth-degree polynomial model of our data that minimizes squared error. Let's try it with a 4th degree polynomial:

[38]:
```
x = np.array(pageSpeeds)
y = np.array(purchaseAmount)

p4 = np.poly1d(np.polyfit(x, y, 4)) #i.e 4th degree polynomial
```

We'll visualize our original scatter plot, together with a plot of our predicted values using the polynomial for page speed times ranging from 0-7 seconds:

[39]:
```
import matplotlib.pyplot as plt

xp = np.linspace(0, 7, 100)
plt.scatter(x, y)
plt.plot(xp, p4(xp), c='r')
plt.show()
```



Looks pretty good! Let's measure the r-squared error:

[40]:
```
from sklearn.metrics import r2_score

r2 = r2_score(y, p4(x)) #good function. Compares data to prediction.

print(r2)
```

```
0.8293766396303072
```

5.5 Multivariate Regression (Multiple Regression)

What if more than one variable influences the one you're interested in? An example would be predicting a price for a car based on its many attributes (body style, brand, mileage, etc.).

We still use least squares (OLS for ordinalry least squyares in our notebook), but instead of a single coeffieicent fo a single varuiable, we have multiple terms with multiple variables.

$$price = \alpha + \beta_1 mileage + \beta_2 age + \beta_3 doors$$

These coefficients imply how important each factor is (if the data is all normalized!). Get rid of ones that don't matter. For this however, we need to assume the different factors are not themselves dependent on each other.

The statsmodel package makes it easy. Let's grab a small little data set of Blue Book car values:

[1]:
```
import pandas as pd

df = pd.read_excel('http://cdn.sundog-soft.com/Udemy/DataScience/cars.xls')
#import as pandas dataframe
```

[2]:
```
df.head()
```

[2]:

	Price	Mileage	Make	Model	Trim	Type	Cylinder	Liter	\
0	17314.103129	8221	Buick	Century	Sedan 4D	Sedan	6	3.1	
1	17542.036083	9135	Buick	Century	Sedan 4D	Sedan	6	3.1	
2	16218.847862	13196	Buick	Century	Sedan 4D	Sedan	6	3.1	
3	16336.913140	16342	Buick	Century	Sedan 4D	Sedan	6	3.1	
4	16339.170324	19832	Buick	Century	Sedan 4D	Sedan	6	3.1	

	Doors	Cruise	Sound	Leather
0	4	1	1	1
1	4	1	1	0
2	4	1	1	0
3	4	1	0	0
4	4	1	0	1

We can use pandas to split up this matrix into the feature vectors we're interested in, and the value we're trying to predict.

Note how we are avoiding the make and model; regressions don't work well with **categorical values**, unless you can convert them into some numerical order that makes sense somehow.

Let's scale our feature data into the same range so we can easily compare the coefficients we end up with.

[4]:
```
import statsmodels.api as sm
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

X = df[['Mileage', 'Cylinder', 'Doors']]
y = df['Price'] # what i am predicting

X[['Mileage', 'Cylinder', 'Doors']] = scale.fit_transform(X[['Mileage', 'Cylinder',␣
 ↪'Doors']].as_matrix())
#the scale fit transform fits the data from a -1 to 1 range. This way i can compare␣
 ↪the coefficents in a meaningfil way
print (X)

est = sm.OLS(y, X).fit()

est.summary()
```

	Mileage	Cylinder	Doors
0	-1.417485	0.527410	0.556279
1	-1.305902	0.527410	0.556279
2	-0.810128	0.527410	0.556279
3	-0.426058	0.527410	0.556279
4	0.000008	0.527410	0.556279
5	0.293493	0.527410	0.556279
6	0.335001	0.527410	0.556279
7	0.382369	0.527410	0.556279
8	0.511409	0.527410	0.556279
9	0.914768	0.527410	0.556279
10	-1.171368	0.527410	0.556279
11	-0.581834	0.527410	0.556279
12	-0.390532	0.527410	0.556279
13	-0.003899	0.527410	0.556279
14	0.430591	0.527410	0.556279
15	0.480156	0.527410	0.556279
16	0.509822	0.527410	0.556279
17	0.757160	0.527410	0.556279
18	1.594886	0.527410	0.556279
19	1.810849	0.527410	0.556279
20	-1.326046	0.527410	0.556279
21	-1.129860	0.527410	0.556279
22	-0.667658	0.527410	0.556279
23	-0.405792	0.527410	0.556279
24	-0.112796	0.527410	0.556279
25	-0.044552	0.527410	0.556279
26	0.190700	0.527410	0.556279
27	0.337442	0.527410	0.556279
28	0.566102	0.527410	0.556279
29	0.660837	0.527410	0.556279
..	...	...	...
774	-0.161262	-0.914896	0.556279
775	-0.089234	-0.914896	0.556279
776	-0.040523	-0.914896	0.556279
777	0.002572	-0.914896	0.556279
778	0.236603	-0.914896	0.556279
779	0.249666	-0.914896	0.556279
780	0.357220	-0.914896	0.556279
781	0.365521	-0.914896	0.556279
782	0.434131	-0.914896	0.556279
783	0.517269	-0.914896	0.556279
784	0.589908	-0.914896	0.556279
785	0.599186	-0.914896	0.556279
786	0.793052	-0.914896	0.556279
787	1.033554	-0.914896	0.556279
788	1.045762	-0.914896	0.556279
789	1.205567	-0.914896	0.556279
790	1.541414	-0.914896	0.556279
791	1.561070	-0.914896	0.556279
792	1.725026	-0.914896	0.556279

```
793  1.851502 -0.914896  0.556279
794 -1.709871  0.527410  0.556279
795 -1.474375  0.527410  0.556279
796 -1.187849  0.527410  0.556279
797 -1.079929  0.527410  0.556279
798 -0.682430  0.527410  0.556279
799 -0.439853  0.527410  0.556279
800 -0.089966  0.527410  0.556279
801  0.079605  0.527410  0.556279
802  0.750446  0.527410  0.556279
803  1.932565  0.527410  0.556279

[804 rows x 3 columns]
```

[4]:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Price   R-squared:                       0.064
Model:                            OLS   Adj. R-squared:                  0.060
Method:                 Least Squares   F-statistic:                     18.11
Date:                Mon, 15 May 2017   Prob (F-statistic):           2.23e-11
Time:                        10:31:39   Log-Likelihood:                -9207.1
No. Observations:                 804   AIC:                         1.842e+04
Df Residuals:                     801   BIC:                         1.843e+04
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Mileage    -1272.3412    804.623     -1.581      0.114   -2851.759     307.077
Cylinder    5587.4472    804.509      6.945      0.000    4008.252    7166.642
Doors      -1404.5513    804.275     -1.746      0.081   -2983.288     174.185
==============================================================================
Omnibus:                      157.913   Durbin-Watson:                   0.008
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              257.529
Skew:                           1.278   Prob(JB):                     1.20e-56
Kurtosis:                       4.074   Cond. No.                         1.03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

The table of coefficients above gives us the values to plug into an equation of form: B0 + B1 * Mileage + B2 * model_ord + B3 * doors

In this example, it's pretty clear that the number of cylinders is more important than anything based on the coefficients.

Could we have figured that out earlier? Yes, wothout using a complicated regression model, we could hav eaily just calculated the mean for each category e.g number of doors.

[5]:
```
y.groupby(df.Doors).mean()
```

[5]:
```
Doors
2    23807.135520
4    20580.670749
Name: Price, dtype: float64
```

Surprisingly, more doors does not mean a higher price! (Maybe it implies a sport car in some cases?) So it's not surprising that it's pretty useless as a predictor here. This is a very small data set however, so we can't really read much meaning into it.

5.6 Multi-Level Models

The concept is that some effects happen at various levels in a hierachy. For example, your health depends on a hierarchy of the health of your cells, which might be a function of the health of the organs theyre inside, which depend on you as a whole, which depends on your family, then your city, and then the world you live in. The state of medical technology etc

Your wealth depends on your own work, what your parents did, what your grandparents did, etc. Multi-level models attempt to model and account for these interdependencies. A hierachy of effects whcih effect each other at larger and larger effects.

Challenge is to identify the factors that affect the outcome you're trying to predict at each level.

For example – SAT scores might be predicted based on the genetics of individual children, the home environment of individual children, the crime rate of the neighborhood they live in, the quality of the teachers in their school, the funding of their school district, and the education policies of their state.

Some of these factors affect more than one level. For example, crime rate might influence the home environment too.

Doing this is hard. Be aware of the concept, as multi-level models showed up on some data science job requirements. Entire advanced statistics and modeling courses exist on this one topic alone. Youre not ready yet.

6 Machine Learning with Python

6.1 Supervised vs Unsupervised Learning

The concept of train/test is a fundamental concept of ML. This lets us cleverly evaluate how good a model we make with ML is.

What is machine learning? "Algorithms that can learn from observational data and can make predictions based on it" it sounds fancy, but it is in reality, these techniques are pretty simple. In regression, we had observational data and we fit a line to it, and we used this line to make predictions.

Unsupervised Learning is when the model is not given any "answers" to learn from; it must make sense of the data just given the observations themselves. No additional information is given. An example would be to group (cluster) some objects together into 2 different sets based on some similarity metric. But I don't tell you what the "right" set is for any object ahead of time. It has no cheat sheet that it can learn from. You don't tell it the "correct" categorisation, it has to infer those categories on its own based on the data alone. The issue is that you don't know what the algorithm will come up with.

Unsupervised learning sounds awful! Why use it? Sometimes you'll see clusters that are surprising that you didn't expect to see. Maybe you don't know what you're looking for – you're looking for **latent variables** i.e. classifications you didn't even know were there.

Example: clustering users on a dating site based on their information and behaviour. Perhaps you'll find there are groups of people that emerge that don't conform to your known stereotypes. Cluster movies based on their properties. Perhaps our current concepts of genre are outdated? Analyse the text of product descriptions to find the terms that carry the most meaning for a certain category. We might not know ahead of time, but using unsupervised learning, we can tease out that meaning.

In supervised learning, the data the algorithm "learns" from comes with the "correct" answers. The training data is where it learns from. It infers relationships between features and categories we want, and then apply that to unseen new values and predict the answer for new, unknown values.

Example: You can train a model for predicting car prices based on car attributes using historical sales data. That model can then predict the optimal price for new cars that haven't been sold before.

How do you evaluate supervised learning? If you have a set of training data that includes the value you're trying to predict – you don't have to guess if the resulting model is good or not. If you have enough training data, you can split it into two parts: a training set and a test set. You then train the model using only the training set and then measure (using r-squared or some other metric) the model's accuracy by asking it to predict values for the test set, and compare that to the known, true values.

There are some caveats however. You need to ensure both sets are large enough to contain representatives of all the variations and outliers in the data you care about. The data sets must be selected randomly, as there could be some pattern sequentially you don't know about. Train/test is a great way to guard against overfitting.

Train/Test is not Infallible. There can still be misleading results. Maybe your sample sizes are too small or due to random chance your train and test sets look remarkably similar and overfitting can still happen.

One way to further protect against overfitting is K-fold cross validation i.e. split your data many times. It sounds complicated. But it's a simple idea:

- Split your data into K randomly-assigned segments
- Reserve one segment as your test data
- Train on each of the remaining K-1 segments and measure their performance against the test set
- Take the average of the K-1 r-squared scores

This way you train on different slices of data, and measure on same test set. This averages out overfitting.

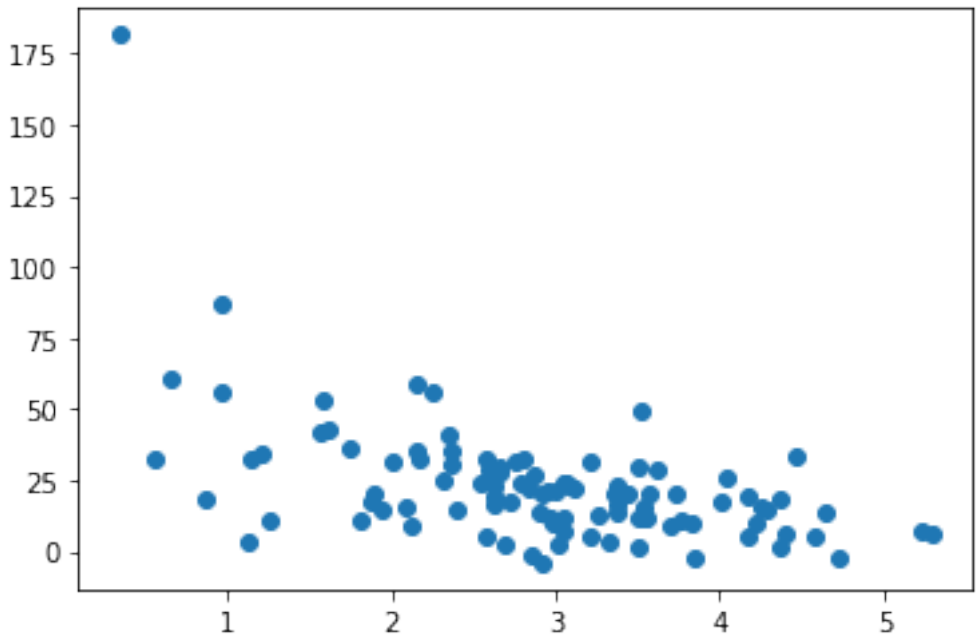
6.2 Train-Test Polynomials in Python

We'll start by creating some data set that we want to build a model for (in this case a polynomial regression):

[11]:
```
%matplotlib inline
import numpy as np
from pylab import *

np.random.seed(2)
```

```
pageSpeeds = np.random.normal(3.0, 1.0, 100)
purchaseAmount = np.random.normal(50.0, 30.0, 100) / pageSpeeds


scatter(pageSpeeds, purchaseAmount)
```

[11]: <matplotlib.collections.PathCollection at 0x20e394aa3d0>



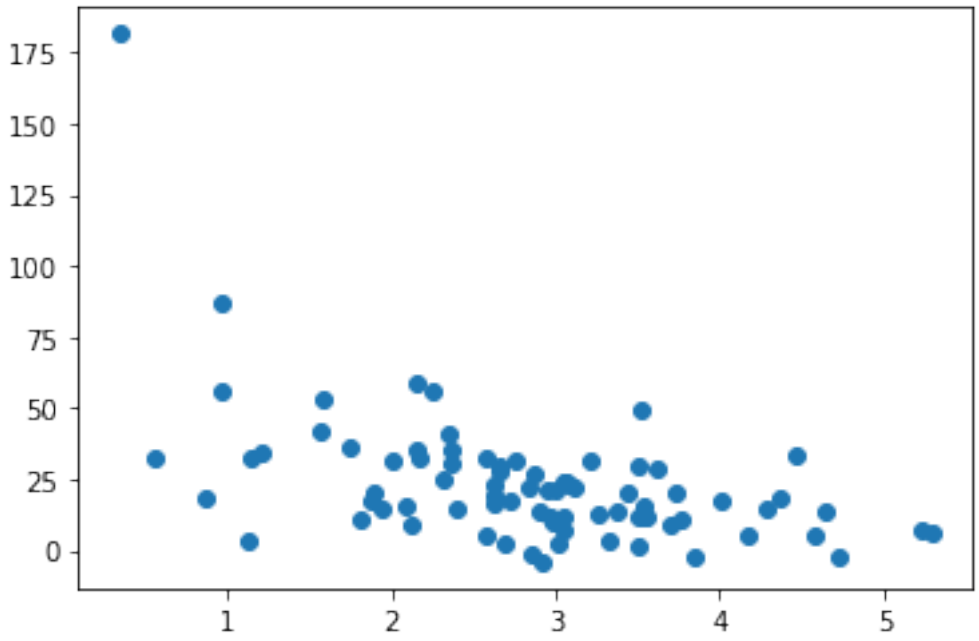
Now we'll split the data in two - 80% of it will be used for "training" our model, and the other 20% for testing it. This way we can avoid overfitting.

[12]:
```
trainX = pageSpeeds[:80]
testX = pageSpeeds[80:]
#its okay to slice here, as the data is random anyway
#pandas has some functions for automatically training/test data
trainY = purchaseAmount[:80]
testY = purchaseAmount[80:]
```

Here's our training dataset:

[13]:
```
scatter(trainX, trainY)
```

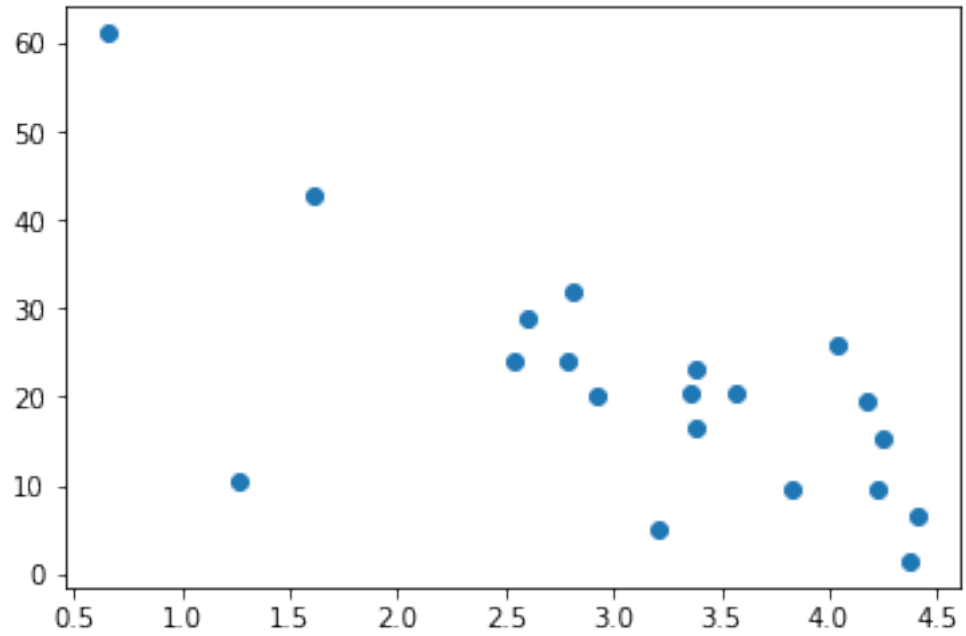
[13]: <matplotlib.collections.PathCollection at 0x20e394fd6a0>



And our test dataset:

[4]:
```
scatter(testX, testY)
```

[4]:
```
<matplotlib.collections.PathCollection at 0x1c445be8320>
```



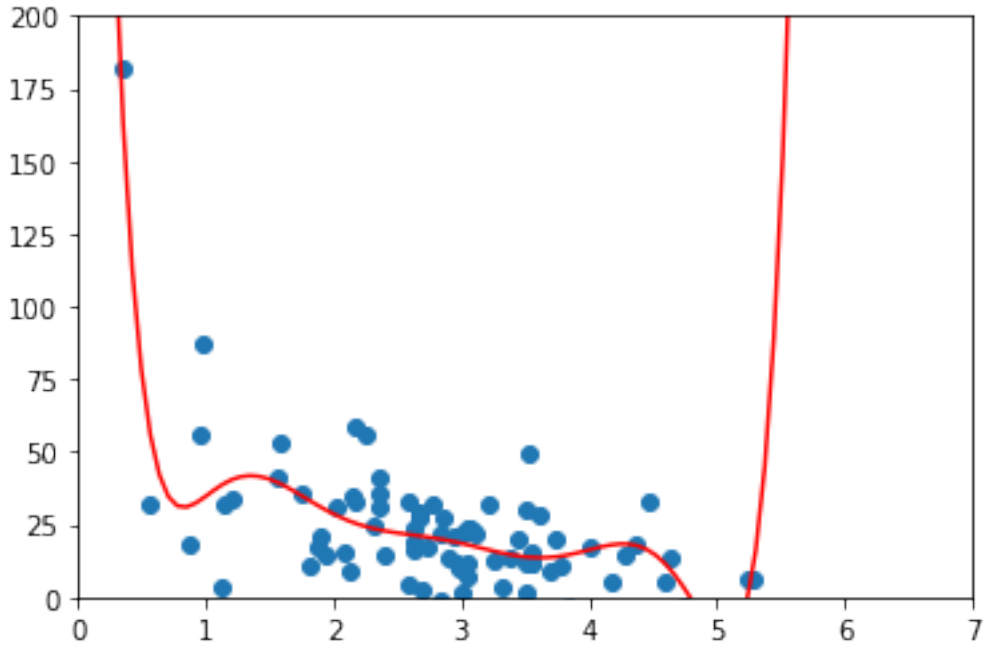
Now we'll try to fit an 8th-degree polynomial to this data (which is almost certainly overfitting, given what we know about how it was generated!)

[19]:
```
x = np.array(trainX)
y = np.array(trainY)
```

```
p4 = np.poly1d(np.polyfit(x, y, 8))
```

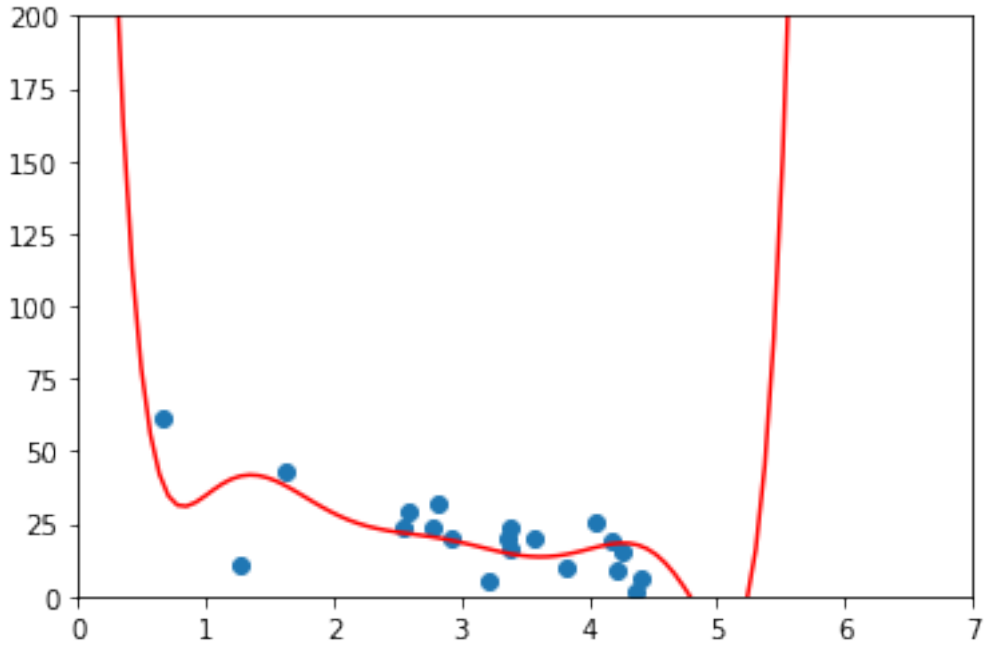
Let's plot our polynomial against the training data:

[20]:
```
import matplotlib.pyplot as plt

xp = np.linspace(0, 7, 100)
axes = plt.axes()
axes.set_xlim([0,7])
axes.set_ylim([0, 200])
plt.scatter(x, y)
plt.plot(xp, p4(xp), c='r')
plt.show()

#great example of overfitting. It would do well with this data, but crap for␣
 ↪predicting
```



And against our test data:

[21]:
```
testx = np.array(testX)
testy = np.array(testY)

axes = plt.axes()
axes.set_xlim([0,7])
axes.set_ylim([0, 200])
plt.scatter(testx, testy)
plt.plot(xp, p4(xp), c='r')
plt.show()
```



Doesn't look that bad when you just eyeball it, but the r-squared score on the test data is kind of horrible! This tells us that our model isn't all that great...

[22]:
```
from sklearn.metrics import r2_score

r2 = r2_score(testy, p4(testx))

print(r2)
```

```
0.30018168613415663
```

...even though it fits the training data better:

[23]:
```
from sklearn.metrics import r2_score

r2 = r2_score(np.array(trainY), p4(np.array(trainX)))

print(r2)
```

```
0.6427069514691345
```

Of course it is higher for the training daa (we trained it on that data).

If you're working with a Pandas DataFrame (using tabular, labeled data,) scikit-learn has built-in train_test_split functions to make this easy to do.

Later we'll talk about even more robust forms of train/test, like K-fold cross-validation - where we try out multiple different splits of the data, to make sure we didn't just get lucky with where we split it.

6.3 Bayesian Methods

Recall Bayes Theorem:

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

Let's use it for machine learning! I want a spam classifier. You can create a spam classifier i.e. An algorithm to analyse a set of spam emails, and a set of non-spam emails and predict if new emails are spam or not.

Example: how would we express the probability of an email being spam if it contains the word "free"?

$$P(Spam|Free) = \frac{P(Spam)P(Free|Spam)}{P(Free)}$$

i.e probablity of it being spam given that it has the word "free in the email" is calculated with the numerator as the probability of a message being spam multiplied by the probability of a an email contining the word "free" given that it is spam.

The numerator can therefore be thought of as the the probability of a message being spam and containing the word "free" (this is subtly different from what we're looking for). Thats the odds of the complete data set, not just what contains the word free.

The denominator is the overall probability of an email containing the word "free". That information isn't always available to us, so it can be calculated by:

$$P(Free|Spam)P(Spam) + P(Free|NotSpam)P(NotSpam)$$

So together – this ratio is the % of emails with the word "free" that are spam.

What about all the other words? We can construct $P(Spam|Word)$ for every (meaningful) word we encounter during training and then multiply these together when analyzing a new email to get the probability of it being spam.

However, this assumes the presence of different words are independent of each other – this is one reason this is called "Naïve Bayes".

Sounds like a lot of work, but Scikit-learn makes it pretty easy to do in python. The **CountVectorizer** lets us operate on lots of words at once by splitting an email into component words and process the words individually, and **MultinomialNB** does all the heavy lifting on Naïve Bayes for us.

We'll train it on known sets of spam and "ham" (non-spam) emails. So this is supervised learning!

6.4 Naive Bayes In Python

We'll cheat by using sklearn.naive_bayes to train a spam classifier! Most of the code is just loading our training data into a pandas DataFrame that we can play with. The ML part is just a couple lines of code. This is pretty typical.

Massaging and cleaning up your data is normally the longest part!

[4]:
```
import os
import io
import numpy
from pandas import DataFrame
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

def readFiles(path): #this function iterates through every file in the directory␣
 ↪and builds up the whole path for each file
    for root, dirnames, filenames in os.walk(path): #walk goes through
```

```
        for filename in filenames:
            path = os.path.join(root, filename)

            inBody = False
            lines = []
            f = io.open(path, 'r', encoding='latin1')
            for line in f:
                if inBody:
                    lines.append(line)
                elif line == '\n': #skips the header by searching for the the first␣
 ↪blank line
                    inBody = True
            f.close()
            message = '\n'.join(lines)
            yield path, message #path and body of data


def dataFrameFromDirectory(path, classification): #define the function
    rows = []
    index = []
    for filename, message in readFiles(path):
        rows.append({'message': message, 'class': classification})
        index.append(filename)

    return DataFrame(rows, index=index)

data = DataFrame({'message': [], 'class': []}) #pandas datafram object 2 empty␣
 ↪lists

data = data.append(dataFrameFromDirectory('C:/Users/Shav/Documents/Python Scripts/
 ↪DataScience-Python3/emails/spam', 'spam'))
data = data.append(dataFrameFromDirectory('C:/Users/Shav/Documents/Python Scripts/
 ↪DataScience-Python3/emails/ham', 'ham'))
```

Let's have a look at that DataFrame:

[5]:
```
data.head()
```

[5]:
```
                                                 message  \
C:/Users/Shav/Documents/Python Scripts/DataScie...  <!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Tr...
C:/Users/Shav/Documents/Python Scripts/DataScie...  1) Fight The Risk of
Cancer!\n\nhttp://www.adc...
C:/Users/Shav/Documents/Python Scripts/DataScie...  1) Fight The Risk of
Cancer!\n\nhttp://www.adc...
C:/Users/Shav/Documents/Python Scripts/DataScie...
##############################################...
C:/Users/Shav/Documents/Python Scripts/DataScie...  I thought you might like
these:\n\n1) Slim Dow...

                                                   class
```

```
C:/Users/Shav/Documents/Python Scripts/DataScie...  spam
C:/Users/Shav/Documents/Python Scripts/DataScie...  spam
C:/Users/Shav/Documents/Python Scripts/DataScie...  spam
C:/Users/Shav/Documents/Python Scripts/DataScie...  spam
C:/Users/Shav/Documents/Python Scripts/DataScie...  spam
```

Now we will use a CountVectorizer to split up each message into its list of words, and throw that into a MultinomialNB classifier. Call fit() and we've got a trained spam filter ready to go! It's just that easy.

[6]:
```
vectorizer = CountVectorizer()
counts = vectorizer.fit_transform(data['message'].values) #this tokenises the words␣
 ↪in 'message' (assigns an index to each word) and counts each word
#represent each word as differnt values in a sparse matrix - each word as a␣
 ↪numerical index in an array
classifier = MultinomialNB()
targets = data['class'].values
classifier.fit(counts, targets) #two inputs - actual data and the targets. i.e list␣
 ↪of words in each email, and how many times it occurs
#targets is the classification for each email that ive encounrterd
```

[6]:
```
MultinomialNB()
```

Let's try it out:

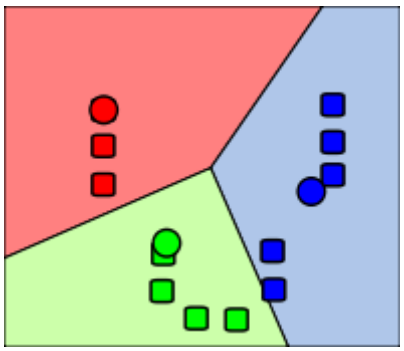
[16]:
```
examples = ['Free Viagra now!!!', "Hi Bob, how about a game of golf tomorrow"]
example_counts = vectorizer.transform(examples) #use the same format you trained␣
 ↪the model on - use vectoriser to convert each message into a list of words and␣
 ↪their frequencies where the words are represented as a postion in an array
predictions = classifier.predict(example_counts)
predictions
```

[16]:
```
array(['spam', 'ham'], dtype='<U4')
```

6.5 K-Means Clustering

This is an *unsupervised learning* technique that attempts to split data into K groups that are closest to K centroids. It is unsupervised learning – so it uses only the positions of each data point itself.

Here, the squares are the data pints and the circles are the "centroids". It finds which centroid a point is closest to in a scatter plot and assigns a cluster. K of 3 is visualised as:



It can uncover interesting groupings of people / things / behavior that you didnt know were there - *latent values*. For example: Where do millionaires live? What genres of music / movies / etc naturally fall out of data? Create your own stereotypes from demographic data?

K-Means Clustering sounds fancy but actually how it works is really simple:

Randomly pick K centroids (k-means) Assign each data point to the centroid it's closest to Recompute the centroids based on the average position of each centroid's points i.e calculate the average location of those data points in the cluster Iterate until points stop changing assignment to centroids

If you want to predict the cluster for new points, just find the centroid they're closest to.

Gaphical Example:

6.6 Limitations of K-Means Clustering

Choosing K value. The principle way of choosing K is to try increasing K values until you stop getting large reductions in squared error (distances from each point to their centroids).

Avoiding local minima. The random choice of initial centroids can yield different results. Run it a few times just to make sure your initial results aren't wacky. This is called "ensambled learning".

Labeling the clusters. K-Means does not attempt to assign any meaning to the clusters you find. It's up to you to dig into the data and try to determine that. This is the hard part.

6.7 K-Means Clustering Example

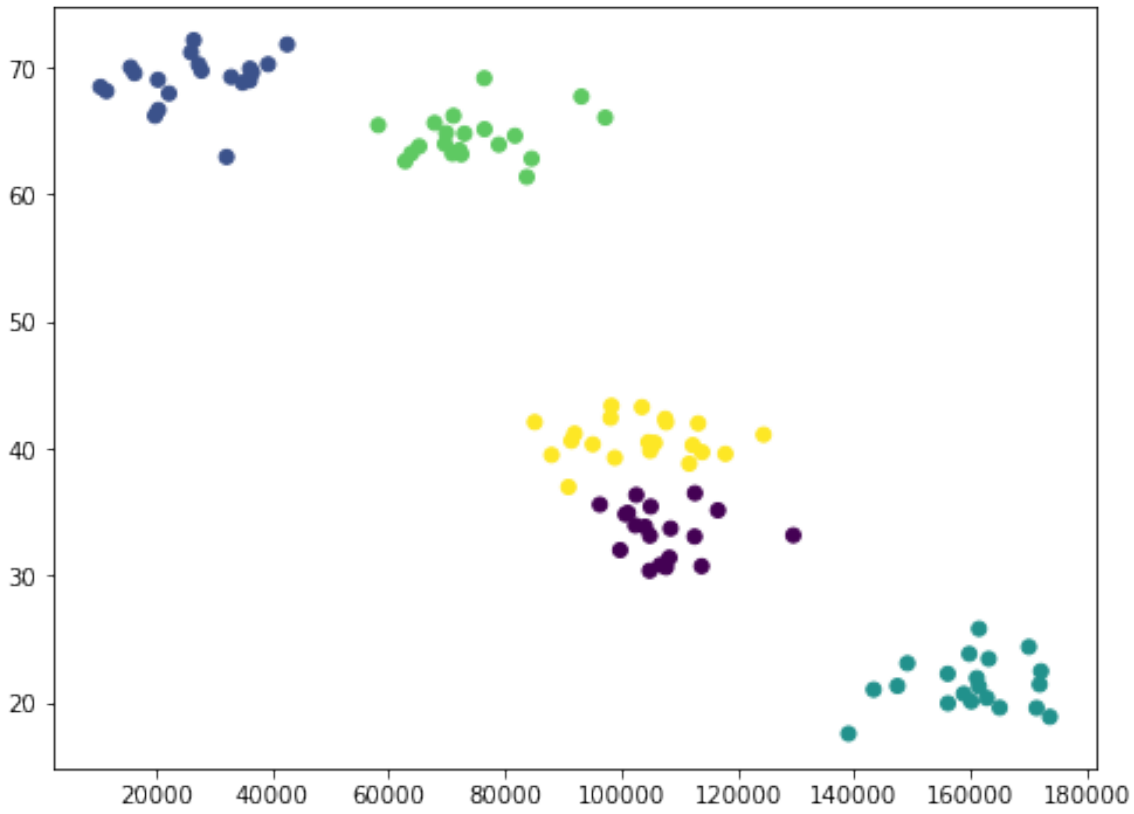
Let's make some fake data that includes people clustered by income and age, randomly:

[7]:
```
from numpy import random, array

#Create fake income/age clusters for N people in k clusters
def createClusteredData(N, k): #n people in k clusters
    random.seed(10)
    pointsPerCluster = float(N)/k
    X = []
    for i in range (k):
        incomeCentroid = random.uniform(20000.0, 200000.0)
        ageCentroid = random.uniform(20.0, 70.0)
        for j in range(int(pointsPerCluster)):
            X.append([random.normal(incomeCentroid, 10000.0), random.
 ↪normal(ageCentroid, 2.0)])
    X = array(X)
    return X
```

We'll use k-means to rediscover these clusters in unsupervised learning:

[11]:
```
%matplotlib inline

from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
from numpy import random, float

data = createClusteredData(100, 5)

model = KMeans(n_clusters=5) #we picked a k value, irl need to converge on one␣
 ↪yourself

# Note I'm scaling the data to normalize it! Important for good results. incomes >>␣
 ↪age. This allows you to compare them.
model = model.fit(scale(data))

# We can look at the clusters each data point was assigned to
print(model.labels_)

# And we'll visualize it:
plt.figure(figsize=(8, 6))
plt.scatter(data[:,0], data[:,1], c=model.labels_.astype(float)) #arbritary colours␣
 ↪assigned
plt.show()
```

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4]
```



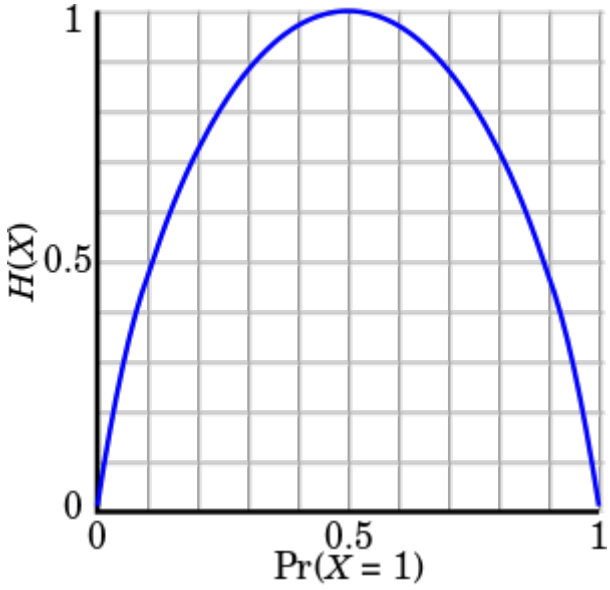
6.8 Entropy

A measure of a data set's disorder – how same or different it is. If we classify a data set into N different classes (example: a data set of animal attributes and their species). The entropy is 0 if all of the classes in the data are the same (everyone is an iguana). The entropy is high if they're all different.

Again, a fancy word for a simple concept. Entropy is just a way of quantifying that sameness or diffrent-ness in your data.

$$H(S) = p_1 ln(p_1) - \ldots - p_n ln(p_n)$$

p_i represents the proportion of the data labeled as that class. Each term looks like this:



We will have it for each class. Only these things in the middle contribute entropy to the class.

6.9 Decison Trees

We can get python to actually generate a flowchart for us to make a decision in real life. A decision tree is a flowchart to help you decide a classification for something with machine learning.

A decision that is based on many variables, a decision tree could be a good choice. E.g. should i go outside, it will look at humidity, temperature etc. It is a form of supervised learning. Give it some sample data and the resulting classifications, and out comes a tree!

Decision Tree example, you want a system to filter out resumes based on historical hiring data. You have a database of some important attributes of job candidates, and you know which ones were hired and which ones weren't. You can train a decision tree on this data and arrive at a system for predicting whether a candidate will get hired based on it. Dependent variable is "hired".

How do Decision Trees Work? At each step, find the attribute we can use to partition the data set to minimize the entropy of the data at the next step. A fancy term for this simple algorithm is ID3.

It is a *"greedy algorithm"* – as it goes down the tree, it just picks the decision that reduce entropy the most at that stage. That might not actually result in an optimal tree, but it works.

One problem with decision tree is that they are susceptible to overfitting. To fight this, we can construct several alternate decision trees and let them "vote" on the final classification.

Fight this by using "random forests". Randomly re-sample the input data for each tree (fancy term for this: bootstrap aggregating or bagging). They then "vote". Another thing random forests can do is restrict the numbers it can choose between at each stage i.e. use a subset of the attributes each step is allowed to choose from. This gives us a variety of algorithms, so they can compete with each other and vote.

This is another example of ensembled learning.

Its crazy that you can actually create a flowchart that really works in python using a few lines of code.

6.10 Decison Trees in Python

First we'll load some fake data on past hires I made up. Note how we use pandas to convert a csv file into a DataFrame:

[26]:
```
import numpy as np
import pandas as pd
from sklearn import tree
import six
import sys
sys.modules['sklearn.externals.six'] = six

input_file = "C:/Users/Shav/Documents/Python Scripts/DataScience-Python3/PastHires.
 ↪csv"
df = pd.read_csv(input_file, header = 0)
```

[12]:
```
df.head()
```

[12]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	Y	4	BS	N	N	Y
1	0	N	0	BS	Y	Y	Y
2	7	N	6	BS	N	N	N
3	2	Y	1	MS	Y	N	Y
4	20	N	2	PhD	Y	N	N

scikit-learn needs everything to be numerical for decision trees to work. So, we'll map Y,N to 1,0 and levels of education to some scale of 0-2. In the real world, you'd need to think about how to deal with unexpected or missing data! By using map(), we know we'll get NaN for unexpected values.

[27]:
```
d = {'Y': 1, 'N': 0} #dictionaty that converts y to 1 and n to 0
df['Hired'] = df['Hired'].map(d)
df['Employed?'] = df['Employed?'].map(d)
df['Top-tier school'] = df['Top-tier school'].map(d)
df['Interned'] = df['Interned'].map(d)
d = {'BS': 0, 'MS': 1, 'PhD': 2}
df['Level of Education'] = df['Level of Education'].map(d)
df.head()
```

[27]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	1	4	0	0	0	1
1	0	0	0	0	1	1	1
2	7	0	6	0	0	0	0
3	2	1	1	1	1	0	1
4	20	0	2	2	1	0	0

```
3                 1          0       1
4                 1          0       0
```

Next we need to separate the features from the target column that we're trying to bulid a decision tree for.

[28]:
```
features = list(df.columns[:6]) #attributes were predicting from
features
```

[28]:
```
['Years Experience',
 'Employed?',
 'Previous employers',
 'Level of Education',
 'Top-tier school',
 'Interned']
```

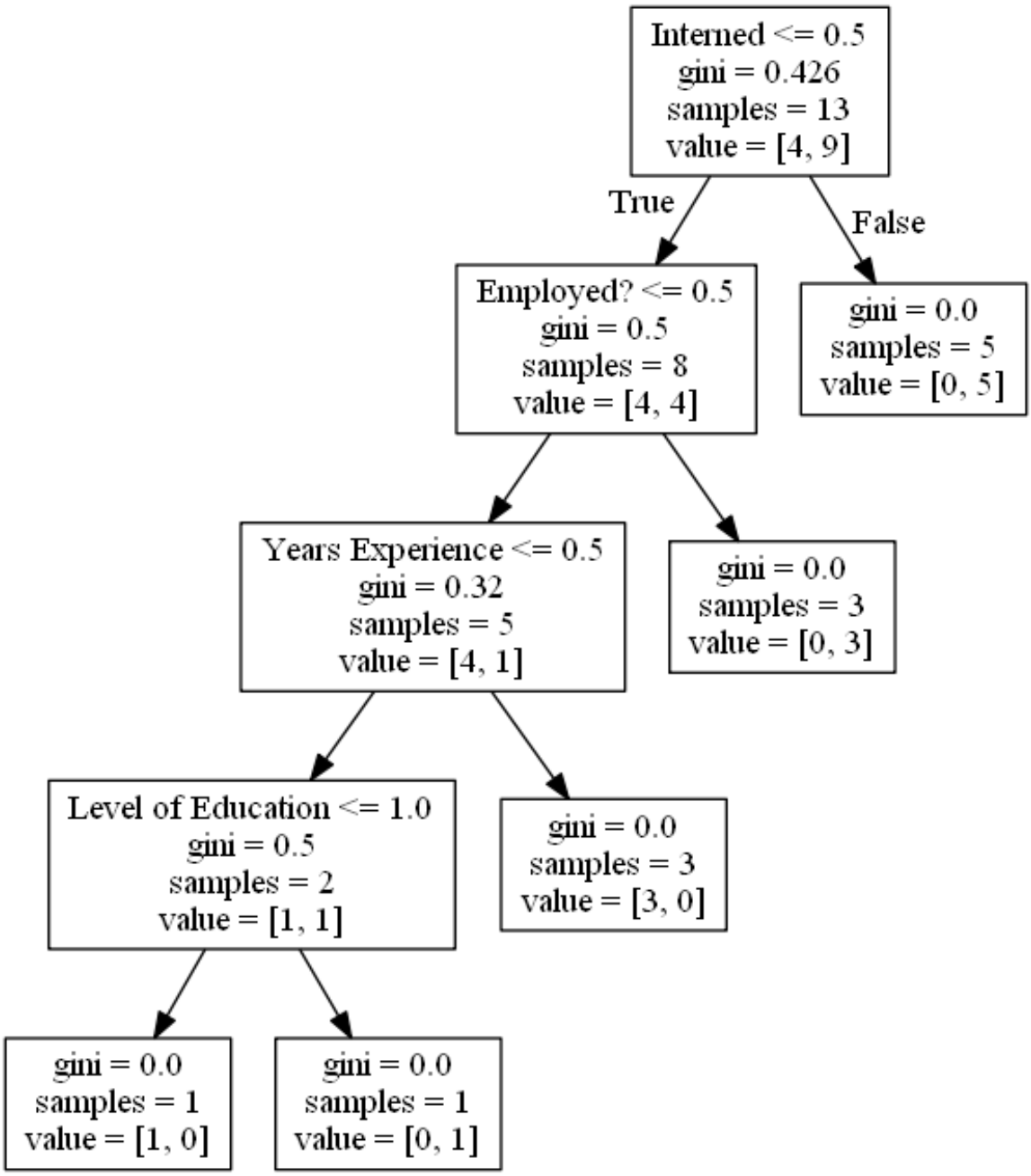
Now actually construct the decision tree:

[29]:
```
y = df["Hired"] #extracts hired column
X = df[features] #all the attributes were predicting form
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X,y)
```

... and display it. Note you need to have pydotplus installed for this to work. (!pip install pydotplus)

To read this decision tree, each condition branches left for "true" and right for "false". When you end up at a value, the value array represents how many samples exist in each target value. So value = [0. 5.] mean there are 0 "no hires" and 5 "hires" by the tim we get to that point. value = [3. 0.] means 3 no-hires and 0 hires.

[30]:
```
from IPython.display import Image
from six import StringIO
import pydotplus

dot_data = StringIO()
tree.export_graphviz(clf, out_file=dot_data,
                         feature_names=features)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())  #note that I edited the code as sklearn.externals.six is␣
 ↪discontinued. Use import six instead.
```



How do i read this? 1. was the person an intern? all our data is 0 or one. If employment value is 0, i.e. 0, you go left. If the value is 1, you go right. The gini score is the measure of entropy it is using at each step. The "value=" shows if they were hired or not. For example, if they have less than 0.5 years experience, they were not hired - [3,0].

6.11 Ensemble learning: using a random forest

Were worried about overfitting our data. We'll use a random forest of 10 decision trees to predict employment of specific candidate profiles:

[145]:
```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_estimators=10)
clf = clf.fit(X, y)
#dont want to walk through tree by hand, you wouuld want to redict
```

```
#Predict employment of an employed 10-year veteran
print (clf.predict([[10, 1, 4, 0, 0, 0]])) #numerical values for the headings we␣
 ↪had before.
#...and an unemployed 10-year veteran
print (clf.predict([[10, 0, 4, 0, 0, 0]]))
```

```
[1]
[0]
```

You get different results sometimes. This is due to the random nature of the forests.

6.12 Ensemble Learning

Random Forests was an example of ensemble learning. It just means we use multiple models to try and solve the same problem and let them vote on the results. This way we returm a better model than any single model could come up with. K-means with different centroid numbers was another example.

Random Forests uses *bagging* (short for bootstrap aggregating) to implement ensemble learning. In simple terms, it means we took random subsamples of our training data and fed them into different versions of the same model, and let them vote on the final result. Many models are built by training on randomly-drawn subsets of the data.

Boosting is an alternate technique where each subsequent model in the ensemble boosts attributes that address data mis-classified by the previous model. i.e what did it get wrong, amplify focus on those weak points, and refine iteratively.

A bucket of models trains several, entirely different, models using training data, and picks the one that works best with the test data. K means, regression and decision trees for example.

Stacking runs multiple models at once on the data, and combines the results together. This is how the Netflix prize was won!

6.13 Advanced Ensemble Learning

A whole field of research on optimising ensemble learning. All have weak points, however. At the end of the day, it is more practical to use the simpler ones.

Bayes Optimal Classifier is theoretically the best – but almost always impractical. Computationally prohibitive to do it. *Bayesian Parameter Averaging* attempts to make BOC practical – but it's still misunderstood, susceptible to overfitting, and often outperformed by the simpler bagging approach. *Bayesian Model Combination* tries to address all of those problems, but in the end, it's about the same as using cross-validation (stacking) to find the best combination of models.

6.14 Support Vector Machines

Advanced way of clustering/classifying higher dimensional data i.e. multiple features you want to predict based off. Results can be crazy good. K means is good for 2 dimensions, but this is for many features you want to predict based off of.

It finds higher-dimensional support vectors (higher dimensional planes to split the data) across which to divide the data (mathematically, these support vectors define hyperplanes).

Under the hood it is good to understand that it uses the *kernel trick* to represent data in higher-dimensional spaces to find hyperplanes that might not be apparent in lower dimensions.

The important point is that SVM's employ some advanced mathematical trickery to cluster data, and it can handle data sets with lots of features. It's also fairly expensive – the "kernel trick" is the only thing that makes it possible.

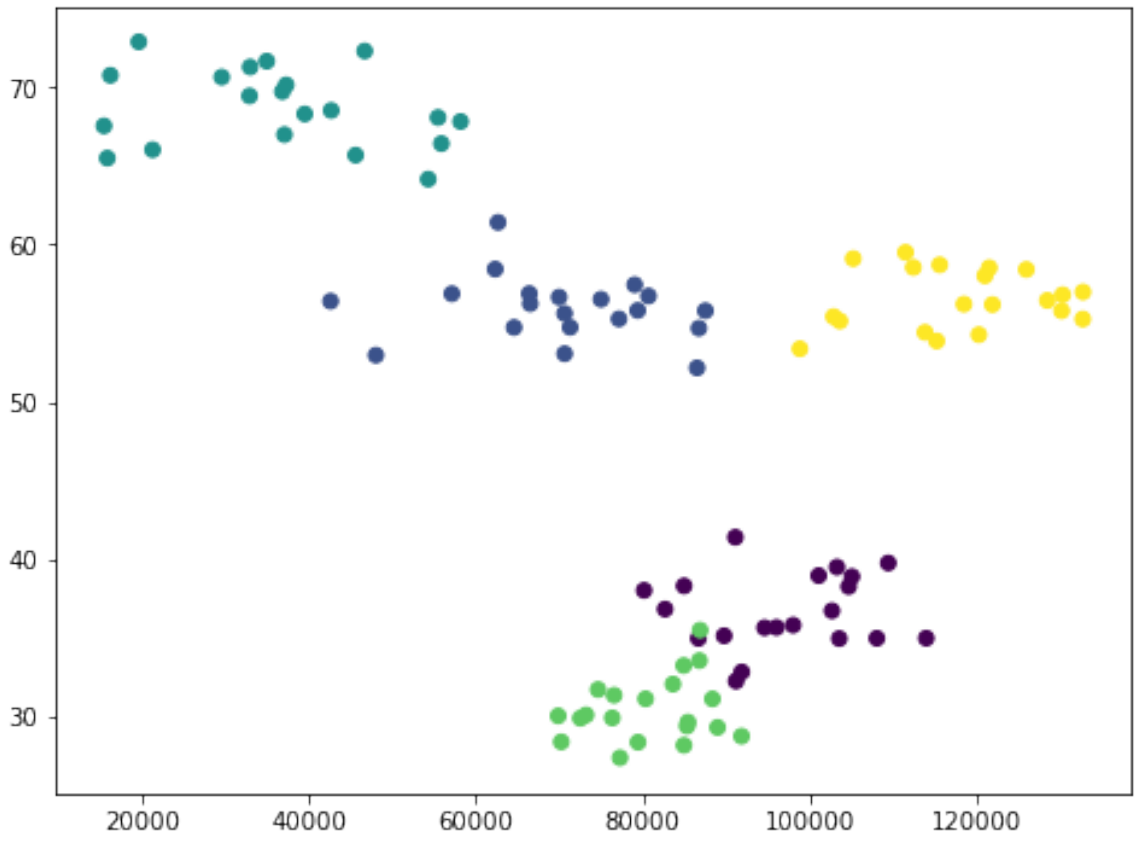
This is also *supervised* by the way, unlike K-means which wasn't. You need to specify the categories.

One example you see is "support vector classification" to classify data using SVM. You can use different "kernels" with SVC. Some will work better than others for a given data set. e.g. polynomial Kernels. Remember, more complexity can actually yield more misleading results.

Let's create the same fake income / age clustered data that we used for our K-Means clustering example:

[2]:
```
import numpy as np

#Create fake income/age clusters for N people in k clusters
def createClusteredData(N, k):
    pointsPerCluster = float(N)/k
    X = []
    y = []
    for i in range (k):
        incomeCentroid = np.random.uniform(20000.0, 200000.0)
        ageCentroid = np.random.uniform(20.0, 70.0)
        for j in range(int(pointsPerCluster)):
            X.append([np.random.normal(incomeCentroid, 10000.0), np.random.
 ↪normal(ageCentroid, 2.0)])
            y.append(i)
    X = np.array(X) #feature array
    y = np.array(y) #what we are predicting for
    return X, y
```

[13]:
```
%matplotlib inline
from pylab import *

(X, y) = createClusteredData(100, 5)

plt.figure(figsize=(8, 6))
plt.scatter(X[:,0], X[:,1], c=y.astype(np.float)) # colour is the number of cluster␣
 ↪shorthand
plt.show()
```

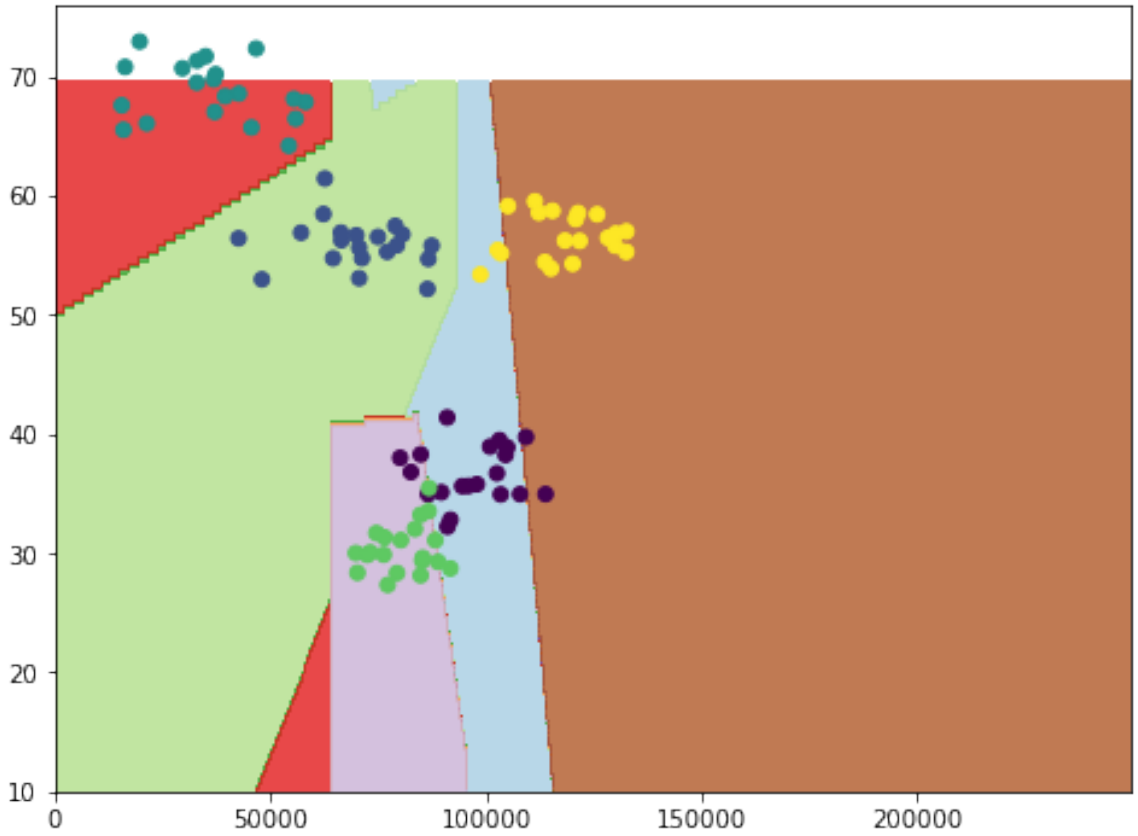


Now we'll use linear SVC to partition our graph into clusters:

[15]:
```
from sklearn import svm, datasets

C = 1.0
svc = svm.SVC(kernel='linear', C=C).fit(X, y) #linear kernel
```

By setting up a dense mesh of points in the grid and classifying all of them, we can render the regions of each cluster as distinct colors:

[16]:
```
def plotPredictions(clf):
    xx, yy = np.meshgrid(np.arange(0, 250000, 10),
                         np.arange(10, 70, 0.5))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    plt.figure(figsize=(8, 6))
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)
    plt.scatter(X[:,0], X[:,1], c=y.astype(np.float))
    plt.show()

plotPredictions(svc)
```



Or just use predict for a given point:

[17]:
```
print(svc.predict([[200000, 40]])) #this way you dont need a graph
```

[4]

[18]:
```
print(svc.predict([[50000, 65]]))
```

[2]

"Linear" is one of many kernels scikit-learn supports on SVC. Look up the documentation for scikit-learn online to find out what the other possible kernel options are. Do any of them work well for this data set?

7 Recommender Systems

Systems that recommend stuff to people based on what everyone else did. There are 2 types: User-Based & Item based Collaborative Filtering. "recommended for you" and "people who bought also bought" type of thing. The latter is based on just what you've bought, or viewed, whereas the former takes into consideration all of your past behaviour. Movie platforms, Netflix etc. all use this technology.

7.1 User-Based Collaborative Filtering

Collaborative filtering is just a fancy way of saying "recommending stuff based on the combination of what you did based on what everyone else did". You start by building a matrix of things each user

bought/viewed/rated (or any signal of interest). Use this matrix to compute similarity scores between users, and recommend stuff that one user has bought/viewed/rated that the other hasn't yet.

Problems with User-Based CF however, include

- People are fickle, tastes change. People might be in a "phase" etc. Peoples taste change over time, so comparing people to people isn't always the best idea. if someone went through a sci fi phase then got into Romcoms, you don't want to recommend someone else who is into sci-fi films Romcoms.
- There are usually many more people than things in your system. Computational problem of finding all the similarities between all of the users in your system is greater than the problem of finding similarities between all the items in your system.
- People do bad things. Economic incentive to make sure your product or item is recommended to people. There are people who try to play the system. You can easily set up a bunch of users/ fake personas in the system by creating new users and have them do a sequence of events, like liking popular items then liking your item too. This is called a **shilling attack**. A better approach would be to use a method that is not susceptible to gaming the system.

7.2 Item-Based Collaborative Filtering

What if we based recommendations on relationships between things instead of people? A movie will always be the same movie – it doesn't change. These relationships are more permanent, and you can make a more direct comparison between items. Also, there are usually fewer things than people (you save computational resources), so your recommendations are more up to date, run more frequently, you can use more complicated algorithms etc. and it is harder to game the system with Item Based CF. Harder to make fake connections between items. Also, make sure people are voting with money. Make voting based on money, so you get better more reliable results. This is what amazon does.

It works very similarly to User based CF but were looking at items instead. First, find every pair of movies that were watched by the same person. Measure the similarity of their ratings across all users who watched both. Then sort by movie, then by similarity strength. (This is just one way to do it!) i.e. relationships between movies are found by relationships of people who watched every pair of movies.

7.3 Finding Similar Movies In Python

Python can be used to this with a surprisingly little amount of code to create real "movie similarities" using the real MovieLens data set. In addition to being important for item-based collaborative filtering, these results are valuable in themselves – think "people who liked X also liked Y".

It's real world data, and we'll encounter real world problems. Then we'll use those results to create movie recommendations for individuals

We'll start by loading up the real dataset from MovieLens project. Using Pandas, we can very quickly load the rows of the u.data and u.item files that we care about, and merge them together so we can work with movie names instead of ID's. (In a real production job, you'd stick with ID's and worry about the names at the display layer to make things more efficient. But this lets us understand what's going on better for now.)

[18]:
```
import pandas as pd
engine='python'

r_cols = ['user_id', 'movie_id', 'rating']
```

```
ratings = pd.read_csv('c:/Users/Shav/Documents/Python Scripts/DataScience-Python3/
 ↪ml-100k/u.data2.csv', names=r_cols, usecols=range(3), encoding="ISO-8859-1",␣
 ↪engine='python')

m_cols = ['movie_id', 'title']
movies = pd.read_csv('c:/Users/Shav/Documents/Python Scripts/DataScience-Python3/
 ↪ml-100k/u.item', sep='|', names=m_cols, usecols=range(2), encoding="ISO-8859-1",␣
 ↪engine='python')


ratings = pd.merge(movies, ratings)
```

[19]:
```
ratings.head()
```

[19]:
```
   movie_id             title user_id  rating
0         1  Toy Story (1995)     308       4
1         1  Toy Story (1995)     287       5
2         1  Toy Story (1995)     148       4
3         1  Toy Story (1995)     280       4
4         1  Toy Story (1995)      66       3
```

Now the amazing pivot_table function on a DataFrame will construct a user / movie rating matrix. Note how NaN indicates missing data - movies that specific users didn't rate.

[20]:
```
movieRatings = ratings.
 ↪pivot_table(index=['user_id'],columns=['title'],values='rating')
movieRatings.head()
```

[20]:
```
title    'Til There Was You (1997)  1-900 (1994)  101 Dalmatians (1996)  \
user_id
0                              NaN           NaN                    NaN
1                              NaN           NaN                    2.0
10                             NaN           NaN                    NaN
100                            NaN           NaN                    NaN
101                            NaN           NaN                    3.0

title    12 Angry Men (1957)  187 (1997)  2 Days in the Valley (1996)  \
user_id
0                        NaN         NaN                          NaN
1                        5.0         NaN                          NaN
10                       5.0         NaN                          NaN
100                      NaN         NaN                          NaN
101                      NaN         NaN                          NaN

title    20,000 Leagues Under the Sea (1954)  2001: A Space Odyssey (1968)  \
user_id
0                                        NaN                           NaN
1                                        3.0                           4.0
10                                       NaN                           5.0
100                                      NaN                           NaN
101                                      NaN                           NaN
```

```
title    3 Ninjas: High Noon At Mega Mountain (1998)  39 Steps, The (1935)  \
user_id
0                                                NaN                  NaN
1                                                NaN                  NaN
10                                               NaN                  4.0
100                                              NaN                  NaN
101                                              NaN                  NaN

title    ...  Yankee Zulu (1994)  Year of the Horse (1997)  \
user_id  ...
0        ...                 NaN                      NaN
1        ...                 NaN                      NaN
10       ...                 NaN                      NaN
100      ...                 NaN                      NaN
101      ...                 NaN                      NaN

title    You So Crazy (1994)  Young Frankenstein (1974)  Young Guns (1988)  \
user_id
0                        NaN                        NaN               NaN
1                        NaN                        5.0               3.0
10                       NaN                        NaN               NaN
100                      NaN                        NaN               NaN
101                      NaN                        NaN               NaN

title    Young Guns II (1990)  Young Poisoner's Handbook, The (1995)  \
user_id
0                         NaN                                   NaN
1                         NaN                                   NaN
10                        NaN                                   NaN
100                       NaN                                   NaN
101                       NaN                                   NaN

title    Zeus and Roxanne (1997)  unknown  Á köldum klaka (Cold Fever) (1994)
user_id
0                            NaN      NaN                                 NaN
1                            NaN      4.0                                 NaN
10                           NaN      NaN                                 NaN
100                          NaN      NaN                                 NaN
101                          NaN      NaN                                 NaN

[5 rows x 1664 columns]
```

You can see now that we can very easily extract vectors of every movie that our user watched, and also of every user that watched/rated every movie. Usedful for both item and user based CF. Let's extract a Series of users who rated Star Wars:

[21]:
```
starWarsRatings = movieRatings['Star Wars (1977)']
starWarsRatings.head()
```

[21]:
```
user_id
0      NaN
1      5.0
10     5.0
100    NaN
101    4.0
Name: Star Wars (1977), dtype: float64
```

Preserve the missing values so you can directly compare columns from differnt movies.

Pandas' corrwith function makes it really easy to compute the **pairwise correlation** of Star Wars' vector of user rating with every other movie (i.e. column)! It gives you the correlation score with every other movie. After that, we'll drop any results that have no data, and construct a new DataFrame of movies and their correlation score (similarity) to Star Wars:

[22]:
```
similarMovies = movieRatings.corrwith(starWarsRatings)
similarMovies = similarMovies.dropna() #drop missing results
df = pd.DataFrame(similarMovies)
df.head(10)
```

```
C:\Users\Shav\anaconda3\lib\site-packages\numpy\lib\function_base.py:2526:
RuntimeWarning: Degrees of freedom <= 0 for slice
  c = cov(x, y, rowvar)
C:\Users\Shav\anaconda3\lib\site-packages\numpy\lib\function_base.py:2455:
RuntimeWarning: divide by zero encountered in true_divide
  c *= np.true_divide(1, fact)
```

[22]:

title	0
'Til There Was You (1997)	0.872872
1-900 (1994)	-0.645497
101 Dalmatians (1996)	0.211132
12 Angry Men (1957)	0.184289
187 (1997)	0.027398
2 Days in the Valley (1996)	0.066654
20,000 Leagues Under the Sea (1954)	0.289768
2001: A Space Odyssey (1968)	0.230884
39 Steps, The (1935)	0.106453
8 1/2 (1963)	-0.142977

(That warning is safe to ignore.) Let's sort the results by similarity score, and we should have the movies most similar to Star Wars! Except... we don't. These results make no sense at all! This is why it's important to know your data - clearly we missed something important.

[23]:
```
similarMovies.sort_values(ascending=False)
```

[23]:
```
title
Hollow Reed (1996)                    1.0
Man of the Year (1995)                1.0
Stripes (1981)                        1.0
Full Speed (1996)                     1.0
Golden Earrings (1947)                1.0
```

```
                                        ...
Theodore Rex (1995)                     -1.0
I Like It Like That (1994)              -1.0
Two Deaths (1995)                       -1.0
Roseanna's Grave (For Roseanna) (1997)  -1.0
Frankie Starlight (1995)                -1.0
Length: 1410, dtype: float64
```

Our results are probably getting messed up by obscure movies that have only been viewed by a **small number** of people who also happened to like Star Wars. So we need to get rid of movies that were only watched by a few people that are producing spurious results.

We need a confidence in our similarity levels based on a minimum bound of how many people watched/reviewed a movie.

Let's construct a new DataFrame that counts up how many ratings exist for each movie, and also the average rating while we're at it - that could also come in handy later.

[24]:
```
import numpy as np
movieStats = ratings.groupby('title').agg({'rating': [np.size, np.mean]})
movieStats.head()
```

[24]:

	rating	
	size	mean
title		
'Til There Was You (1997)	9	2.333333
1-900 (1994)	5	2.600000
101 Dalmatians (1996)	109	2.908257
12 Angry Men (1957)	125	4.344000
187 (1997)	41	3.024390

Let's get rid of any movies rated by fewer than 100 people, and check the top-rated ones that are left. (yes this is intuitive way of choosing, but there are more principle ways of doing train/test experiments on differnt threshold values to find which one performs the best)

[26]:
```
popularMovies = movieStats['rating']['size'] >= 100
movieStats[popularMovies].sort_values([('rating', 'mean')], ascending=False)[:15]
```

[26]:

	rating	
	size	mean
title		
Close Shave, A (1995)	112	4.491071
Schindler's List (1993)	298	4.466443
Wrong Trousers, The (1993)	118	4.466102
Casablanca (1942)	243	4.456790
Shawshank Redemption, The (1994)	283	4.445230
Rear Window (1954)	209	4.387560
Usual Suspects, The (1995)	267	4.385768
Star Wars (1977)	584	4.359589
12 Angry Men (1957)	125	4.344000
Citizen Kane (1941)	198	4.292929
To Kill a Mockingbird (1962)	219	4.292237

```
One Flew Over the Cuckoo's Nest (1975)    264  4.291667
Silence of the Lambs, The (1991)          390  4.289744
North by Northwest (1959)                 179  4.284916
Godfather, The (1972)                     413  4.283293
```

100 might still be too low, but these results look pretty good as far as "well rated movies that people have heard of." Let's join this data with our original set of similar movies to Star Wars:

[27]:
```
df = movieStats[popularMovies].join(pd.DataFrame(similarMovies,␣
↪columns=['similarity']))
```

```
C:\Users\Shav\anaconda3\lib\site-packages\pandas\core\reshape\merge.py:618:
UserWarning: merging between different levels can give an unintended result (2
levels on the left, 1 on the right)
  warnings.warn(msg, UserWarning)
```

[28]:
```
df.head()
```

[28]:

title	(rating, size)	(rating, mean)	similarity
101 Dalmatians (1996)	109	2.908257	0.211132
12 Angry Men (1957)	125	4.344000	0.184289
2001: A Space Odyssey (1968)	259	3.969112	0.230884
Absolute Power (1997)	127	3.370079	0.085440
Abyss, The (1989)	151	3.589404	0.203709

And, sort these new results by similarity score. That's more like it!

[29]:
```
df.sort_values(['similarity'], ascending=False)[:15] #take a look at first 15␣
↪results
```

[29]:

title	(rating, size)	\
Star Wars (1977)	584	
Empire Strikes Back, The (1980)	368	
Return of the Jedi (1983)	507	
Raiders of the Lost Ark (1981)	420	
Austin Powers: International Man of Mystery (1997)	130	
Sting, The (1973)	241	
Indiana Jones and the Last Crusade (1989)	331	
Pinocchio (1940)	101	
Frighteners, The (1996)	115	
L.A. Confidential (1997)	297	
Wag the Dog (1997)	137	
Dumbo (1941)	123	
Bridge on the River Kwai, The (1957)	165	
Philadelphia Story, The (1940)	104	
Miracle on 34th Street (1994)	101	

title	(rating, mean)	similarity
Star Wars (1977)	4.359589	1.000000

```
Empire Strikes Back, The (1980)                    4.206522    0.747981
Return of the Jedi (1983)                          4.007890    0.672556
Raiders of the Lost Ark (1981)                     4.252381    0.536117
Austin Powers: International Man of Mystery (1997) 3.246154    0.377433
Sting, The (1973)                                  4.058091    0.367538
Indiana Jones and the Last Crusade (1989)          3.930514    0.350107
Pinocchio (1940)                                   3.673267    0.347868
Frighteners, The (1996)                            3.234783    0.332729
L.A. Confidential (1997)                           4.161616    0.319065
Wag the Dog (1997)                                 3.510949    0.318645
Dumbo (1941)                                       3.495935    0.317656
Bridge on the River Kwai, The (1957)               4.175758    0.316580
Philadelphia Story, The (1940)                     4.115385    0.314272
Miracle on 34th Street (1994)                      3.722772    0.310921
```

Ideally we'd also filter out the movie we started from - of course Star Wars is 100% similar to itself. But otherwise these results aren't bad.

7.4 Item-Based Collaborative Filtering

As before, we'll start by importing the MovieLens 100K (big data near 1 million pushes the limits of what 1 machine can do) data set into a pandas DataFrame:

[44]:
```
import pandas as pd


r_cols = ['user_id', 'movie_id', 'rating']
ratings = pd.read_csv('c:/Users/Shav/Documents/Python Scripts/DataScience-Python3/
 ↪ml-100k/u.data2.csv', names=r_cols, usecols=range(3), encoding="ISO-8859-1",␣
 ↪engine='python')

m_cols = ['movie_id', 'title']
movies = pd.read_csv('c:/Users/Shav/Documents/Python Scripts/DataScience-Python3/
 ↪ml-100k/u.item', sep='|', names=m_cols, usecols=range(2), encoding="ISO-8859-1",␣
 ↪engine='python')

ratings = pd.merge(movies, ratings)

ratings.head()
```

[44]:

	movie_id	title	user_id	rating
0	1	Toy Story (1995)	308	4
1	1	Toy Story (1995)	287	5
2	1	Toy Story (1995)	148	4
3	1	Toy Story (1995)	280	4
4	1	Toy Story (1995)	66	3

Now we'll pivot this table to construct a nice matrix of users and the movies they rated. NaN indicates missing data, or movies that a given user did not watch:

[45]:
```
userRatings = ratings.
 ↪pivot_table(index=['user_id'],columns=['title'],values='rating')
userRatings.head()
```

[45]:
```
title    'Til There Was You (1997)  1-900 (1994)  101 Dalmatians (1996)  \
user_id
0                              NaN           NaN                    NaN
1                              NaN           NaN                    2.0
10                             NaN           NaN                    NaN
100                            NaN           NaN                    NaN
101                            NaN           NaN                    3.0

title    12 Angry Men (1957)  187 (1997)  2 Days in the Valley (1996)  \
user_id
0                        NaN         NaN                          NaN
1                        5.0         NaN                          NaN
10                       5.0         NaN                          NaN
100                      NaN         NaN                          NaN
101                      NaN         NaN                          NaN

title    20,000 Leagues Under the Sea (1954)  2001: A Space Odyssey (1968)  \
user_id
0                                        NaN                           NaN
1                                        3.0                           4.0
10                                       NaN                           5.0
100                                      NaN                           NaN
101                                      NaN                           NaN

title    3 Ninjas: High Noon At Mega Mountain (1998)  39 Steps, The (1935)  \
user_id
0                                                NaN                   NaN
1                                                NaN                   NaN
10                                               NaN                   4.0
100                                              NaN                   NaN
101                                              NaN                   NaN

title    ...  Yankee Zulu (1994)  Year of the Horse (1997)  \
user_id  ...
0        ...                 NaN                       NaN
1        ...                 NaN                       NaN
10       ...                 NaN                       NaN
100      ...                 NaN                       NaN
101      ...                 NaN                       NaN

title    You So Crazy (1994)  Young Frankenstein (1974)  Young Guns (1988)  \
user_id
0                        NaN                        NaN                NaN
1                        NaN                        5.0                3.0
10                       NaN                        NaN                NaN
100                      NaN                        NaN                NaN
```

```
101                        NaN                            NaN                 NaN

title    Young Guns II (1990)  Young Poisoner's Handbook, The (1995)  \
user_id
0                         NaN                                   NaN
1                         NaN                                   NaN
10                        NaN                                   NaN
100                       NaN                                   NaN
101                       NaN                                   NaN

title    Zeus and Roxanne (1997)  unknown  Á köldum klaka (Cold Fever) (1994)
user_id
0                            NaN      NaN                                 NaN
1                            NaN      4.0                                 NaN
10                           NaN      NaN                                 NaN
100                          NaN      NaN                                 NaN
101                          NaN      NaN                                 NaN

[5 rows x 1664 columns]
```

Now we have this useful matrix. We can do correlations between columns, or rows.

Now the magic happens - pandas has a built-in corr() method that will compute a correlation score for every column pair in the matrix! This gives us a correlation score between every pair of movies (where at least one user rated both movies - otherwise NaN's will show up.) That's amazing!

[46]:
```
corrMatrix = userRatings.corr()
corrMatrix.head()
```

[46]:
```
title                     'Til There Was You (1997)  1-900 (1994)  \
title
'Til There Was You (1997)                        1.0           NaN
1-900 (1994)                                     NaN           1.0
101 Dalmatians (1996)                           -1.0           NaN
12 Angry Men (1957)                             -0.5           NaN
187 (1997)                                      -0.5           NaN

title                     101 Dalmatians (1996)  12 Angry Men (1957)  \
title
'Til There Was You (1997)              -1.000000            -0.500000
1-900 (1994)                                 NaN                  NaN
101 Dalmatians (1996)                   1.000000            -0.049890
12 Angry Men (1957)                    -0.049890             1.000000
187 (1997)                              0.269191             0.666667

title                     187 (1997)  2 Days in the Valley (1996)  \
title
'Til There Was You (1997)   -0.500000                    0.522233
1-900 (1994)                      NaN                         NaN
101 Dalmatians (1996)        0.269191                    0.048973
12 Angry Men (1957)          0.666667                    0.256625
```

```
187 (1997)                   1.000000                    0.596644

title                    20,000 Leagues Under the Sea (1954)  \
title
'Til There Was You (1997)                                 NaN
1-900 (1994)                                              NaN
101 Dalmatians (1996)                                0.266928
12 Angry Men (1957)                                  0.274772
187 (1997)                                                NaN

title                    2001: A Space Odyssey (1968)  \
title
'Til There Was You (1997)                     -0.426401
1-900 (1994)                                  -0.981981
101 Dalmatians (1996)                         -0.043407
12 Angry Men (1957)                            0.178848
187 (1997)                                    -0.554700

title                    3 Ninjas: High Noon At Mega Mountain (1998)  \
title
'Til There Was You (1997)                                          NaN
1-900 (1994)                                                       NaN
101 Dalmatians (1996)                                              NaN
12 Angry Men (1957)                                                NaN
187 (1997)                                                         NaN

title                    39 Steps, The (1935)  ...  Yankee Zulu (1994)  \
title                                          ...
'Til There Was You (1997)                 NaN  ...                 NaN
1-900 (1994)                              NaN  ...                 NaN
101 Dalmatians (1996)                0.111111  ...                 NaN
12 Angry Men (1957)                  0.457176  ...                 NaN
187 (1997)                           1.000000  ...                 NaN

title                    Year of the Horse (1997)  You So Crazy (1994)  \
title
'Til There Was You (1997)                       NaN                 NaN
1-900 (1994)                                    NaN                 NaN
101 Dalmatians (1996)                     -1.000000                 NaN
12 Angry Men (1957)                             NaN                 NaN
187 (1997)                                0.866025                  NaN

title                    Young Frankenstein (1974)  Young Guns (1988)  \
title
'Til There Was You (1997)                       NaN                NaN
1-900 (1994)                              -0.944911                NaN
101 Dalmatians (1996)                      0.158840           0.119234
12 Angry Men (1957)                        0.096546           0.068944
187 (1997)                                 0.455233          -0.500000
```

```
title                       Young Guns II (1990)  \
title
'Til There Was You (1997)                    NaN
1-900 (1994)                                 NaN
101 Dalmatians (1996)                   0.680414
12 Angry Men (1957)                    -0.361961
187 (1997)                              0.500000

title                       Young Poisoner's Handbook, The (1995)  \
title
'Til There Was You (1997)                                     NaN
1-900 (1994)                                                  NaN
101 Dalmatians (1996)                                    0.000000
12 Angry Men (1957)                                      0.144338
187 (1997)                                               0.475327

title                       Zeus and Roxanne (1997)  unknown  \
title
'Til There Was You (1997)                       NaN      NaN
1-900 (1994)                                    NaN      NaN
101 Dalmatians (1996)                      0.707107      NaN
12 Angry Men (1957)                        1.000000      1.0
187 (1997)                                      NaN      NaN

title                       Á köldum klaka (Cold Fever) (1994)
title
'Til There Was You (1997)                                  NaN
1-900 (1994)                                               NaN
101 Dalmatians (1996)                                      NaN
12 Angry Men (1957)                                        NaN
187 (1997)                                                 NaN

[5 rows x 1664 columns]
```

However, we want to avoid spurious results, like before, that happened from just a handful of users that happened to rate the same pair of movies i.e. using a small amount of behaviour information. In order to restrict our results to movies that lots of people rated together - and also give us more popular results that are more easily recongnizable - we'll use the min_periods argument to throw out results where fewer than 100 users rated a given movie pair:

[47]:
```
corrMatrix = userRatings.corr(method='pearson', min_periods=100) #use pearson score␣
 ↪correlation method, and the min periods. Backed up by > 100 people i.e.
corrMatrix.head()
```

[47]:
```
title                       'Til There Was You (1997)  1-900 (1994)  \
title
'Til There Was You (1997)                         NaN           NaN
1-900 (1994)                                      NaN           NaN
101 Dalmatians (1996)                             NaN           NaN
12 Angry Men (1957)                               NaN           NaN
187 (1997)                                        NaN           NaN
```

```
title                    101 Dalmatians (1996)  12 Angry Men (1957)  \
title
'Til There Was You (1997)                   NaN                  NaN
1-900 (1994)                                NaN                  NaN
101 Dalmatians (1996)                       1.0                  NaN
12 Angry Men (1957)                         NaN                  1.0
187 (1997)                                  NaN                  NaN

title                    187 (1997)  2 Days in the Valley (1996)  \
title
'Til There Was You (1997)        NaN                          NaN
1-900 (1994)                     NaN                          NaN
101 Dalmatians (1996)            NaN                          NaN
12 Angry Men (1957)              NaN                          NaN
187 (1997)                       NaN                          NaN

title                    20,000 Leagues Under the Sea (1954)  \
title
'Til There Was You (1997)                                 NaN
1-900 (1994)                                              NaN
101 Dalmatians (1996)                                     NaN
12 Angry Men (1957)                                       NaN
187 (1997)                                                NaN

title                    2001: A Space Odyssey (1968)  \
title
'Til There Was You (1997)                          NaN
1-900 (1994)                                       NaN
101 Dalmatians (1996)                              NaN
12 Angry Men (1957)                                NaN
187 (1997)                                         NaN

title                    3 Ninjas: High Noon At Mega Mountain (1998)  \
title
'Til There Was You (1997)                                        NaN
1-900 (1994)                                                     NaN
101 Dalmatians (1996)                                            NaN
12 Angry Men (1957)                                              NaN
187 (1997)                                                       NaN

title                    39 Steps, The (1935)  ...  Yankee Zulu (1994)  \
title                                          ...
'Til There Was You (1997)                  NaN  ...                 NaN
1-900 (1994)                               NaN  ...                 NaN
101 Dalmatians (1996)                      NaN  ...                 NaN
12 Angry Men (1957)                        NaN  ...                 NaN
187 (1997)                                 NaN  ...                 NaN

title                    Year of the Horse (1997)  You So Crazy (1994)  \
```

```
title
'Til There Was You (1997)                      NaN                 NaN
1-900 (1994)                                   NaN                 NaN
101 Dalmatians (1996)                          NaN                 NaN
12 Angry Men (1957)                            NaN                 NaN
187 (1997)                                     NaN                 NaN

title                     Young Frankenstein (1974)  Young Guns (1988)  \
title
'Til There Was You (1997)                       NaN               NaN
1-900 (1994)                                    NaN               NaN
101 Dalmatians (1996)                           NaN               NaN
12 Angry Men (1957)                             NaN               NaN
187 (1997)                                      NaN               NaN

title                     Young Guns II (1990)  \
title
'Til There Was You (1997)                  NaN
1-900 (1994)                               NaN
101 Dalmatians (1996)                      NaN
12 Angry Men (1957)                        NaN
187 (1997)                                 NaN

title                     Young Poisoner's Handbook, The (1995)  \
title
'Til There Was You (1997)                                    NaN
1-900 (1994)                                                 NaN
101 Dalmatians (1996)                                        NaN
12 Angry Men (1957)                                          NaN
187 (1997)                                                   NaN

title                     Zeus and Roxanne (1997)  unknown  \
title
'Til There Was You (1997)                     NaN      NaN
1-900 (1994)                                  NaN      NaN
101 Dalmatians (1996)                         NaN      NaN
12 Angry Men (1957)                           NaN      NaN
187 (1997)                                    NaN      NaN

title                     Á köldum klaka (Cold Fever) (1994)
title
'Til There Was You (1997)                                NaN
1-900 (1994)                                             NaN
101 Dalmatians (1996)                                    NaN
12 Angry Men (1957)                                      NaN
187 (1997)                                               NaN

[5 rows x 1664 columns]
```

See 1-900 was watched by fewer than 100 poeple, so there werent even 100 people who liked itself,

so has no correlation with itself.

Now let's produce some movie recommendations for user ID 0, who I manually added to the data set as a test case. This guy really likes Star Wars and The Empire Strikes Back, but hated Gone with the Wind. I'll extract his ratings from the userRatings DataFrame, and use dropna() to get rid of missing data (leaving me only with a Series of the movies I actually rated:)

[48]:
```
userRatings = userRatings.reset_index()
myRatings = userRatings.loc[0].dropna()
myRatings = myRatings[1:]
myRatings
```

[48]:
```
title
Empire Strikes Back, The (1980)    5
Gone with the Wind (1939)          1
Star Wars (1977)                   5
Name: 0, dtype: object
```

Now, let's go through each movie I rated one at a time, and build up a list of possible recommendations based on the movies similar to the ones I rated.

So for each movie I rated, I'll retrieve the list of similar movies from our correlation matrix. I'll then scale those correlation scores by how well I rated the movie they are similar to, so movies similar to ones I liked count more than movies similar to ones I hated:

[49]:
```
simCandidates = pd.Series()
for i in range(0, len(myRatings.index)):
    print ("Adding sims for " + myRatings.index[i] + "...")
    # Retrieve similar movies to this one that I rated
    sims = corrMatrix[myRatings.index[i]].dropna()
    # Now scale its similarity by how well I rated this movie
    sims = sims.map(lambda x: x * myRatings[i])
    # Add the score to the list of similarity candidates
    simCandidates = simCandidates.append(sims)

#Glance at our results so far:
print ("sorting...")
simCandidates.sort_values(inplace = True, ascending = False)
print (simCandidates.head(10))
```

```
Adding sims for Empire Strikes Back, The (1980)...
Adding sims for Gone with the Wind (1939)...
Adding sims for Star Wars (1977)...
sorting...
Empire Strikes Back, The (1980)                         5.000000
Star Wars (1977)                                        5.000000
Empire Strikes Back, The (1980)                         3.741763
Star Wars (1977)                                        3.741763
Return of the Jedi (1983)                               3.606146
Return of the Jedi (1983)                               3.362779
Raiders of the Lost Ark (1981)                          2.693297
Raiders of the Lost Ark (1981)                          2.680586
Austin Powers: International Man of Mystery (1997)      1.887164
```

```
Sting, The (1973)                                   1.837692
dtype: float64

<ipython-input-49-2e46217d9732>:1: DeprecationWarning: The default dtype for
empty Series will be 'object' instead of 'float64' in a future version. Specify
a dtype explicitly to silence this warning.
  simCandidates = pd.Series()
```

This is starting to look like something useful! Note that some of the same movies came up more than once, because they were similar to more than one movie I rated. We'll use groupby() to add together the scores from movies that show up more than once, so they'll count more:

[50]:
```
simCandidates = simCandidates.groupby(simCandidates.index).sum()
```

[51]:
```
simCandidates.sort_values(inplace = True, ascending = False)
simCandidates.head(10)
```

[51]:
```
Empire Strikes Back, The (1980)             8.877450
Star Wars (1977)                            8.870971
Return of the Jedi (1983)                   7.178172
Raiders of the Lost Ark (1981)              5.519700
Indiana Jones and the Last Crusade (1989)    3.488028
Bridge on the River Kwai, The (1957)        3.366616
Back to the Future (1985)                   3.357941
Sting, The (1973)                           3.329843
Cinderella (1950)                           3.245412
Field of Dreams (1989)                      3.222311
dtype: float64
```

The last thing we have to do is filter out movies I've already rated, as recommending a movie I've already watched isn't helpful:

[52]:
```
filteredSims = simCandidates.drop(myRatings.index)
filteredSims.head(10)
```

[52]:
```
Return of the Jedi (1983)                   7.178172
Raiders of the Lost Ark (1981)              5.519700
Indiana Jones and the Last Crusade (1989)    3.488028
Bridge on the River Kwai, The (1957)        3.366616
Back to the Future (1985)                   3.357941
Sting, The (1973)                           3.329843
Cinderella (1950)                           3.245412
Field of Dreams (1989)                      3.222311
Wizard of Oz, The (1939)                    3.200268
Dumbo (1941)                                2.981645
dtype: float64
```

There we have it!

7.5 Improvements on Item-Based Collaborative Filtering

Also, it looks like some movies similar to Gone with the Wind - which were lowly rated - made it through to the final list of recommendations. Perhaps movies similar to ones the user rated poorly

should actually be penalized, instead of just scaled down?

There are also probably some outliers in the user rating data set - some users may have rated a huge amount of movies and have a disporportionate effect on the results. Go back to earlier topics to learn how to identify these outliers, and see if removing them improves things.

Perhaps a different min_periods value on the correlation computation would produce more intersting results. If you lower that value, youll get more obscure movies. If you make it higher, you'll get more blockbusters. You need to think of the balance between new movie discovery vs confidence in the recommender system, seeing movies they have heard of.

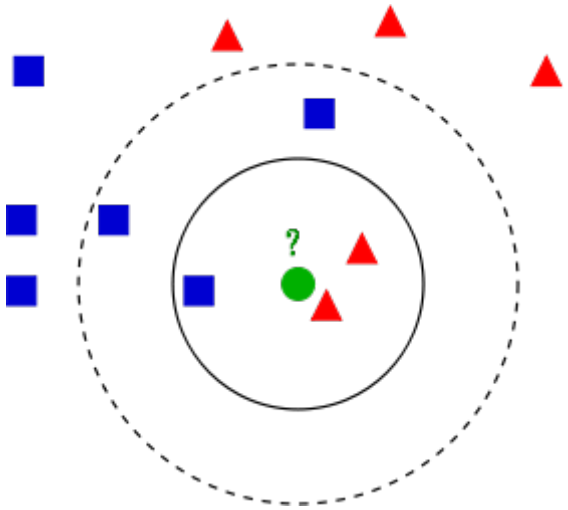
For an even bigger project: we're evaluating the result qualitatively here, but we could actually apply train/test and measure our ability to predict user ratings for movies they've already watched. Whether that's actually a measure of a "good" recommendation is debatable, though! Finding what movies poeple have watched is very different from movies they might like.

8 More Data Mining and Machine Learning Techniques

8.1 K-Nearest Neighbor

Supervised machine learning technique. Say you have a scatter plot; this technique is used to classify new data points based on "distance" to known data. Find the K nearest neighbours, based on your distance metric, and let them all vote on the classification. That is it!

Say this scatter plot is movies, and blue squares are sci-fi, and triangles are Romcoms, and the axes are anything you want e.g. ratings and popularity.



If we have a new point that we don't know the genre for, we can take K as 3, look at the K nearest neighbours, and they can all vote on the classification of this new data point.

If you were to expand this circle to 5, you get a different classification The choice of K can be very important. K should be small enough that you don't have to go too far and pick up irrelevant neighbours, but should be big enough to get a meaningful sample. Usually you can use train-test, or a similar technique, to determine K for a certain data set. At the start though, you use your intuition.

This does qualify as supervised data, as the known classifications inform the classification of the new point. Although it's one of the simplest machine learning models there is – it still qualifies as "supervised learning". But let's do something more complex with it; movie similarities just based on metadata!

8.2 KNN (K-Nearest-Neighbors)

KNN is a simple concept: define some distance metric between the items in your dataset, and find the K closest items. You can then use those items to predict some property of a test item, by having them somehow "vote" on it.

As an example, let's look at the MovieLens data. We'll try to *guess the rating of a movie by looking at the 10 movies that are closest to it in terms of genres and popularity.*

To start, we'll load up every rating in the data set into a Pandas DataFrame:

[12]:
```
import pandas as pd
#First define a distance metric based purely off metadata i.e here Genere␣
 ↪classification & overall popularity by number of people who rated it, and rating
r_cols = ['user_id', 'movie_id', 'rating']
rr_cols = ['user_id', 'movie_id', 'rating']
ratings = pd.read_csv('c:/Users/Shav/Documents/Python Scripts/DataScience-Python3/
 ↪ml-100k/u.data2.csv', names=r_cols, usecols=range(3),␣
 ↪encoding="ISO-8859-1",engine='python')
ratings = ratings.iloc[1:]
ratings.head()
```

[12]:

	user_id	movie_id	rating
1	0	172	5
2	0	133	1
3	0	50	5
4	186	302	3
5	22	377	1

Now, we'll group everything by movie ID, and compute the total number of ratings (each movie's popularity) and the average rating for every movie:

[13]:
```
import numpy as np

movieProperties = ratings.groupby('movie_id').agg({'rating': [np.size, np.mean]})
movieProperties.head()
```

[13]:

	rating	
	size	mean
movie_id		
1	452	3.878319
2	131	3.206107
3	90	3.033333
4	209	3.550239
5	86	3.302326

The raw number of ratings isn't very useful for computing distances between movies, so we'll create a new DataFrame that contains the normalized number of ratings. So, a value of 0 means nobody rated it, and a value of 1 will mean it's the most popular movie there is.

[14]:
```
movieNumRatings = pd.DataFrame(movieProperties['rating']['size'])
#apply function to entire data set . here we noramlise to range
```

```
movieNormalizedNumRatings = movieNumRatings.apply(lambda x: (x - np.min(x)) / (np.
 ↪max(x) - np.min(x)))
movieNormalizedNumRatings.head()
```

[14]:
```
              size
movie_id
1         0.773585
2         0.222985
3         0.152659
4         0.356775
5         0.145798
```

Now, let's get the genre information from the u.item file. The way this works is there are 19 fields, each corresponding to a specific genre - a value of '0' means it is not in that genre, and '1' means it is in that genre. A movie may have more than one genre associated with it.

While we're at it, we'll put together everything into one big Python dictionary called movieDict. Each entry will contain the movie name, list of genre values, the normalized popularity score, and the average rating for each movie:

[15]:
```
movieDict = {}
with open(r"C:\Users\Shav\Documents\Python Scripts\DataScience-Python3\ml-100k\u.
 ↪item", encoding = "ISO-8859-1") as f:
    temp = ''
    for line in f:
        #line.decode("ISO-8859-1")
        fields = line.rstrip('\n').split('|') #pipe delimiter
        movieID = int(fields[0])
        name = fields[1]
        genres = fields[5:25]
        genres = map(int, genres)
        movieDict[movieID] = (name, np.array(list(genres)),␣
 ↪movieNormalizedNumRatings.loc[movieID].get('size'), movieProperties.loc[movieID].
 ↪rating.get('mean'))
```

For example, here's the record we end up with for movie ID 1, "Toy Story":

[17]:
```
print(movieDict[1])
```

```
('Toy Story (1995)', array([0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]), 0.7735849056603774, 3.8783185840707963)
```

Now let's define a function that computes the "distance" between two movies based on how similar their genres are, and how similar their popularity is. Just to make sure it works, we'll compute the distance between movie ID's 2 and 4:

We essentially compute how similar the vector of genres is between differnt movies

[22]:
```
from scipy import spatial

def ComputeDistance(a, b):
    genresA = a[1]
    genresB = b[1]
```

```
    genreDistance = spatial.distance.cosine(genresA, genresB)
    popularityA = a[2]
    popularityB = b[2]
    popularityDistance = abs(popularityA - popularityB) #take absolute diff
    return genreDistance + popularityDistance

ComputeDistance(movieDict[2], movieDict[4])
```

[22]: 0.8004574042309892

Remember the higher the distance, the less similar the movies are. Let's check what movies 2 and 4 actually are - and confirm they're not really all that similar:

[23]:
```
print(movieDict[2])
print(movieDict[4])
```

```
('GoldenEye (1995)', array([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0]), 0.22298456260720412, 3.2061068702290076)
('Get Shorty (1995)', array([0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]), 0.3567753001715266, 3.550239234449761)
```

Now, we just need a little code to compute the distance between some given test movie (Toy Story, in this example) and all of the movies in our data set. When the sort those by distance, and print out the K nearest neighbors:

[33]:
```
import operator
#find k nearest neighbours for a given movie
def getNeighbors(movieID, K):
    distances = []
    for movie in movieDict:
        if (movie != movieID): #if a diffenet movie, it will compute diff
            dist = ComputeDistance(movieDict[movieID], movieDict[movie])
            distances.append((movie, dist))
    distances.sort(key=operator.itemgetter(1))
    neighbors = []
    for x in range(K):
        neighbors.append(distances[x][0]) #pluck off k nearest neighbours
    return neighbors

K = 10 #arbitrary number here for K
avgRating = 0
neighbors = getNeighbors(1, K)
for neighbor in neighbors:
    avgRating += movieDict[neighbor][3]
    print (movieDict[neighbor][0] + " " + str(movieDict[neighbor][3]))

avgRating /= K
```

```
Liar Liar (1997) 3.156701030927835
Aladdin (1992) 3.8127853881278537
Willy Wonka and the Chocolate Factory (1971) 3.6319018404907975
Monty Python and the Holy Grail (1974) 4.0664556962025316
```

```
Full Monty, The (1997) 3.926984126984127
George of the Jungle (1997) 2.685185185185185
Beavis and Butt-head Do America (1996) 2.7884615384615383
Birdcage, The (1996) 3.4436860068259385
Home Alone (1990) 3.0875912408759123
Aladdin and the King of Thieves (1996) 2.8461538461538463
```

While we were at it, we computed the average rating of the 10 nearest neighbors to Toy Story:

[34]:
```
avgRating
```

[34]:
```
3.3445905900235564
```

How does this compare to Toy Story's actual average rating?

[35]:
```
movieDict[1]
```

[35]:
```
('Toy Story (1995)',
 array([0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 0.7735849056603774,
 3.8783185840707963)
```

Not too bad!

Our choice of 10 for K was arbitrary - what effect do different K values have on the results?

Our distance metric was also somewhat arbitrary - we just took the cosine distance between the genres and added it to the difference between the normalized popularity scores. Can you improve on that?

Can we use train test to find a value for K that is better at predicting avg. rating. Can we weigh parameters differently?

8.3 The Curse of Dimensionality

Usually means Principal Component Analysis or more particularly as a technique, Singular Value Decomposition (SVD).

What is the curse of dimensionality? Many problems can be thought of as having a huge number of "dimesions". For example, in recommending movies, the ratings vector for each movie may represent a dimension – every movie is its own dimension! That makes your head hurt. It's tough to visualize. We are only used to 3. Another example would be 4 different measurements of a flower. Those 4 features can represent 4 dimentiosns, which again, is hard to visualise.

Dimensionality reduction attempts to distill higher-dimensional data down to a smaller number of dimensions, while preserving as much of the variance in the data as possible. Useful for compressing data, or feature extraction.

A simple example would be K-means clustering. This is an example of a dimensionality reduction algorithm. It reduces data down to K dimensions using K centroids and distance to those centroids.

Normally though when people talk about dimentionality reduction, they are talking about PCA. Involves fancy math, but at a high level, it finds "eigenvectors" in the higher dimensional data. These define hyperplanes (planes in higer dimentions) that split the data while preserving the most variance in it. The data gets projected onto these hyperplanes, which represent the lower dimensions

you specied that you want to represent. A popular implementation of this is called Singular Value Decomposition (SVD).

Also, this is really useful for things like image compression and facial recognition. e.g. if i had a data set of faces, maybe each face reresents a thord dimention of 2D images. SVD and PCA can identify the features that really count, that are neccesary for preserving variance in that data set.

8.4 Principal Component Analysis

PCA is a dimensionality reduction technique; it lets you distill multi-dimensional data down to fewer dimensions, selecting new dimensions that preserve variance in the data as best it can.

We're not talking about Star Trek stuff here; let's make it real - a black & white image for example, contains three dimensions of data: X position, Y position, and brightness at each point. Distilling that down to two dimensions can be useful for things like image compression and facial recognition, because it distills out the information that contributes most to the variance in the data set.

Let's do this with a simpler example: the Iris data set that comes with scikit-learn. It's just a small collection of data that has four dimensions of data for three different kinds of Iris flowers: The length and width of both the petals and sepals of many individual flowers from each species. PCA lets us visualize this in 2 dimensions instead of 4, while still preserving variance. Let's load it up and have a look:

[1]:
```
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
import pylab as pl
from itertools import cycle

iris = load_iris()

numSamples, numFeatures = iris.data.shape
print(numSamples)
print(numFeatures)
print(list(iris.target_names))
```

```
150
4
['setosa', 'versicolor', 'virginica']
```

So, this tells us our data set has 150 samples (individual flowers) in it. It has 4 dimensions - called features here, and three distinct Iris species that each flower is classified into.

While we can visualize 2 or even 3 dimensions of data pretty easily, visualizing 4D data isn't something our brains can do. So let's distill this down to 2 dimensions, and see how well it works:

[3]:
```
X = iris.data
pca = PCA(n_components=2, whiten=True).fit(X) # n is the desried dimentions whiten␣
 ↪is to noramlise data and make it all comparable
X_pca = pca.transform(X)
```

What we have done is distill our 4D data set down to 2D, by projecting it down to two orthogonal 4D vectors that make up the basis of our new 2D projection. We can see what those 4D vectors are, although it's not something you can really wrap your head around (these are the eigen vectors):

[5]:
```
print(pca.components_)
```

```
[[ 0.36138659 -0.08452251  0.85667061  0.3582892 ]
 [ 0.65658877  0.73016143 -0.17337266 -0.07548102]]
```

Let's see how much information we've managed to preserve:

[6]:
```
print(pca.explained_variance_ratio_)
print(sum(pca.explained_variance_ratio_)) #i.e. how much of variance in orgianl␣
 ↪data was preserved as you reduced dimentions
#2 items. 1st dimention preserved 92% and second dimention only gave us an␣
 ↪additonal 5%
```

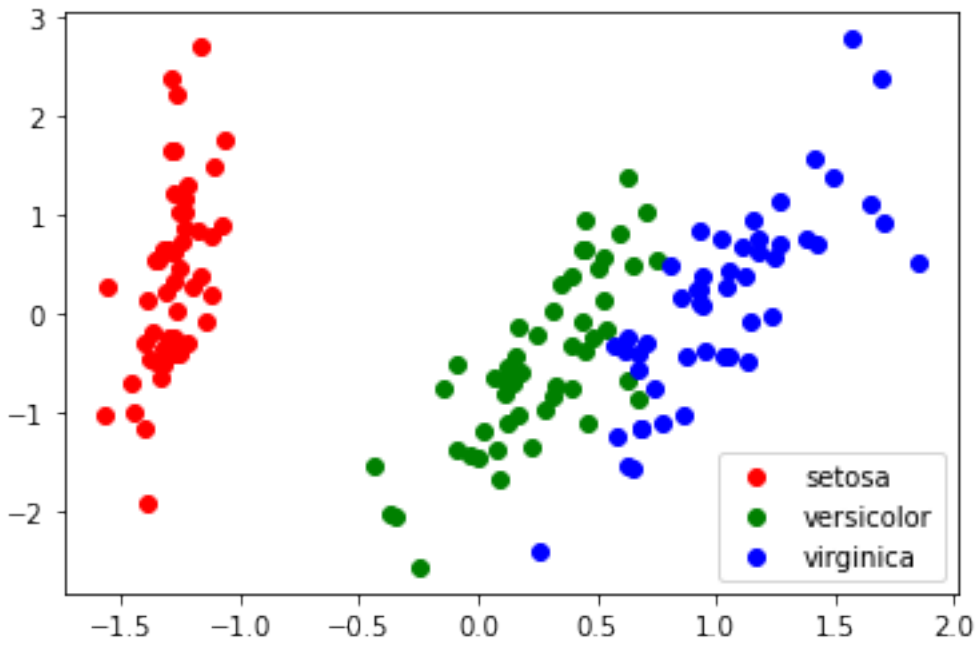
```
[0.92461872 0.05306648]
0.977685206318795
```

That's pretty cool. Although we have thrown away two of our four dimensions, PCA has chosen the remaining two dimensions well enough that we've captured 92% of the variance in our data in a single dimension alone! The second dimension just gives us an additional 5%; altogether we've only really lost less than 3% of the variance in our data by projecting it down to two dimensions.

Why is this? maybe it is only the ratio of the length to width. Maybe its the overall size? some of these things might move together.

As promised, now that we have a 2D representation of our data, we can plot it:

[7]:
```
%matplotlib inline
from pylab import *

colors = cycle('rgb')
target_ids = range(len(iris.target_names))
pl.figure()
for i, c, label in zip(target_ids, colors, iris.target_names): #iterate though 3␣
 ↪and associate colour with it
    pl.scatter(X_pca[iris.target == i, 0], X_pca[iris.target == i, 1],
        c=c, label=label)
pl.legend()
pl.show()
```



You can see the three different types of Iris are still clustered pretty well. If you think about it, this probably works well because the overall size of an individual flower probably makes both the petal and sepal sizes increase by a similar amount. Although the actual numbers on this graph have no intuitive meaning, what we're probably seeing is measure of the ratio of width to height for petals and sepals - and PCA distilled our data down to that on its own.

Our results suggest we could actually distill this data down to a single dimension and still preserve most of its variance. Try it! Do a PCA down to one component, and measure the results.

8.5 ETL and ELT

Data Warehousing Introduction in general. What is Data Warehousing? it is a large, centralized database that contains information from many sources and ties them together.

It is often used for business analysis in large corporations or organizations. e.g. in an e-commerce business, could be used for tying in data from the web server logs, customer service systems etc.

Data warehouse has the challenge of taking data from many sources, transforming them into some sort of schema that allows us to query data sources simultaneously, and make insights using data analysis.

Queried via SQL or tools (i.e. graphical Tableau). Data analyst vs Data scientist. Data analyst query large data sets, data scientist write code that border on AI.

Often entire departments are dedicated to maintaining a data warehouse. Challenges include data normalization is tricky – how does all of this data relate to each other? What views do people need? Maintaining the data feeds is a lot of work and scaling is tricky (esp when it is a very large data set).

ETL: Extract, Transform, Load. ETL and ELT refer to how data gets into a data warehouse. Traditionally, the flow was Extract, Transform, Load. Raw data from operational systems is first periodically extracted e.g. from web logs. Then, the data is transformed into the schema needed by the DW and finally, the data is loaded into the data warehouse, already in the structure needed.

But what if we're dealing with "big data"? That transform step can turn into a big problem.

ELT: Extract, Load, Transform. This is a newer technique that says, "what if we don't use a huge oracle instance". Use a Hadoop cluster?

Today, a huge Oracle instance isn't the only choice for a large data warehouse. Things like Hive let you host massive databases on a Hadoop cluster. Or, you might store it in a large, distributed NoSQL data store and query it using things like Spark or MapReduce.

i.e. use the power of the repository itself to do the transformation in place. Instead of an offline transformation, we use hadoop so we can query it.

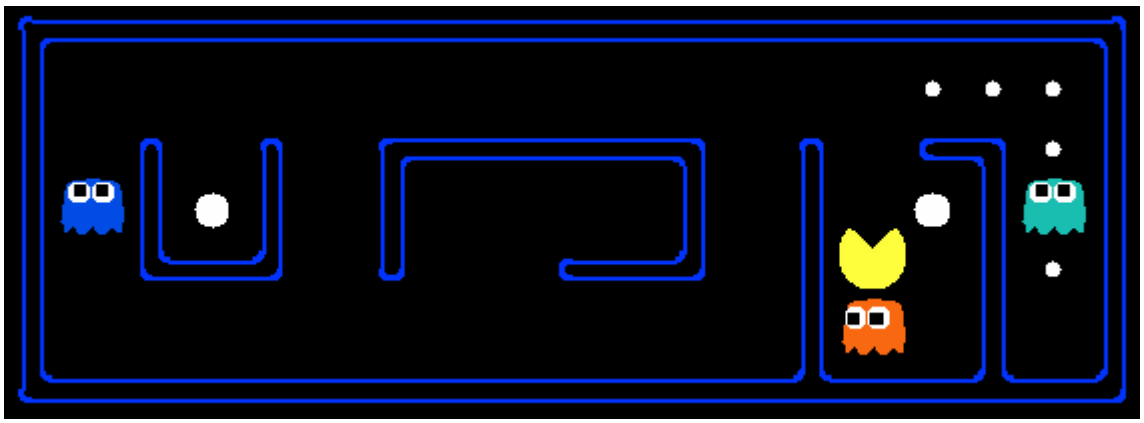
The scalability of Hadoop lets you flip the loading process on its head. The idea is that instead of a monolithic database for a data warehouse, you instead use something built on top of Hadoop or some sort of a cluster. This can not only scale up the processing and querying of data, but also the transformation of data.

You can extract raw data as before, load it in as-is and then use the power of Hadoop to transform it in-place.

Lots more to explore. Data warehousing is a discipline in itself, too big too cover here. Check out other courses on Big Data, Spark, and MapReduce.

We will cover Spark in more depth later. SparkSQL in particular is relevant. Also Hive and Map reduce are relevant here. The overall concept is that if you move from a monolithic database built on Oracle or MySQL to a modern distributed databases built on Hadoop, you can take the transform stage and do it after you load in the raw data. This is simpler, more scalable and takes advantage of the power of large clouds of computing clusters.

8.6 Reinforcement Learning



Teach computer to play Pac-Man. You have some sort of agent that "explores" some space. As it goes, it learns the value of different state changes in different conditions. For example, the state of Pac-man might be what Pac-man has in each surrounding tile. Those values inform subsequent behavior of the agent.

Examples: Pac-Man, Cat & Mouse game

The benefit of this technique is that once you've explored the entire set of possible states your agent can be in, you can very quickly have a good performance when you run iterations of it. i.e. it yields fast on-line performance once the space has been explored

8.7 Q-Learning

This is a specific implementation of reinforcement learning. You have:

- A set of environmental states, *s*
- A set of possible actions in those states, *a*
- A value of each state/action pair, *Q*

Start off with Q values of 0 for every possible state. As Pac-man explores the space, and as bad things happen after a given state/action, reduce its Q. As rewards happen after a given state/action, increase its Q.

What are some state/actions here for Pac-man? Pac-man has a wall to the West, Pac-man dies if he moves one step South, Pac-man just continues to live if going North or East.

You can "look ahead" more than one step by using a discount factor when computing Q (here s is previous state, s' is current state)

$$Q(s,a) += discount * (reward(s,a) + max(Q(s')) - Q(s,a))$$

Or

$$Q(s,a) += alpha * (reward(s,a) + max(Q(s') - Q(s,a))$$

where s is the previous state, a is the previous action, s' is the current state, and alpha is the discount factor (set to .5 here).

Discount factor can builds in "memory" into the system. Factor that into the Q value for the previous state.

8.8 The exploration problem

How do we efficiently explore all of the possible states? The simple approach is to just always choose the action for a given state with the highest Q. If there's a tie, choose at random.

But that's really inefficient, and you might miss a lot of paths that way.

A better way would be to introduce an epsilon term. This introduces some random variation into your actions as you explore. If a random number is less than epsilon, don't follow the highest Q, but choose at random.

That way, exploration never totally stops, and a wider range of actions is explored. However, choosing epsilon can be tricky

This is called the *Markov Decision Process*. This is, from Wikipedia, "Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker"

Sound familiar? MDP's are just a way to describe what we just did using mathematical notation. States are still described as s and s'. State transition functions are described as $P_a(s,s')$, our "Q" values are described as a reward function $R_a(s,s')$

An MDP is a "discrete time stochastic control process".

Dynamic Programming: From Wikipedia: dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions - ideally, using a memory-based data structure. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.

This also describes the process outlined here. each state is a simpler sub-problem. The stored solutions are the Q values, and are associated with states somehow.

8.9 Pac Man Recap

You can make an intelligent Pac-Man in a few steps.

1. Have it semi-randomly explore different choices of movement (actions) given different conditions (states)
2. Keep track of the reward or penalty associated with each choice for a given state/action (Q)
3. Use those stored Q values to inform its future choices

This is a pretty simple concept. But hey, now you can say you understand reinforcement learning, Q-learning, Markov Decision Processes, and Dynamic Programming!

9 Dealing with Real-World Data

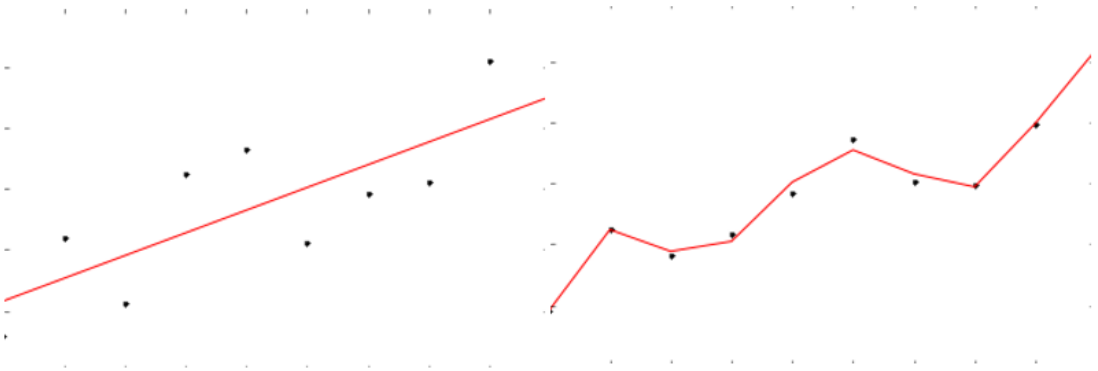
9.1 The Bias / Variance Tradeoff

One of the basic issues we run into when dealing with real world data is overfitting vs underfitting your models/regressions'predictions to that data. Normally we can talk about that in the context of bias and variance.

Bias is how far removed the mean of your predicted values is from the "real" answer. *Variance* is how scattered your predicted values are from the "real" answer. i.e. it is accuracy vs precison.

Here the top left has high variance but low bias. Top right has low variance but high bias. Bottom left has high bias and high variacne. The lower rigt has low bias and low variance.

In reality, you often need to choose between bias and variance. It comes down to overfitting vs underfitting your data.

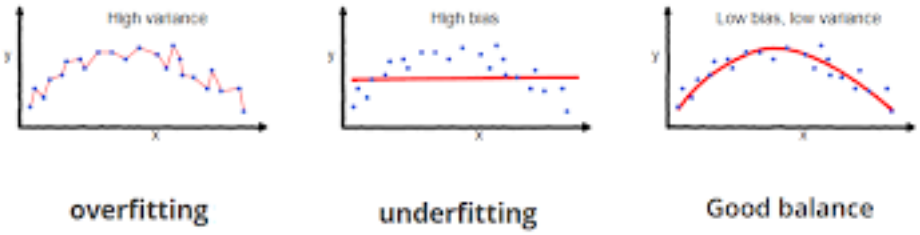


On the left you have a straight line with low variace relative to the observations. Bias (error) from each individual point is high however. On the right, we have high variance, but low bias as each individual point is where it should be. Example of where we have trded off variance for bias.

But what you really care about is error at the end of the day, not just bias and variance specifically. Bias and variance both contribute to error:

$$Error = Bias^2 + Variance$$

An overly complex model will have high variance and low bias. A too-simple model will have low variance and high bias, but both may have the same error – the optimal complexity is in the middle.



Tying it to earlier lessons, increasing K in K-Nearest-Neighbors decreases variance and increases bias (by averaging together more neighbors).

A single decision tree is prone to overfitting – high variance. But a random forest decreases that variance by having mutiple trees and averages solutions.

9.2 K-Fold Cross Validation

Train test can predict how well your predictions based off models are performing. Another tool to fight over fitting is "K-Fold Cross Validation". This makes tain test even better.

Recall from train/test; separate your data and train model in the training data set and then test it on the remaining data to evaluate its performance. This prevents over fitting to data as we test it against data it's never seen before. However train test still has limitations, like over fitting to your specific train test split, and it may not be representative of the entire data set.

The idea is to instead of splitting our data into 2 buckets, we divide into K buckets randomly-assigned segments. Reserve one segment as your **test data** for evaluating the results of the model. Train on

each of the remaining K-1 segments and measure their performance against the test set to evaluate how well we did, and take the average of the K-1 r-squared scores.

What if the test data set isn't representative? We can randomly assign buckets to prevent this. Fortunately, scikit-learn makes this really easy. Even easier than just a single train/test split. In practice, you need to try different variations of your model you want to tune and measure the mean accuracy using K-Fold Cross validation until you find a sweet spot.

9.3 K-Fold Cross Validation In Practice

Let's revisit the Iris data set:

[6]:
```
import numpy as np
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn import datasets
from sklearn import svm

iris = datasets.load_iris()
```

A single train/test split is made easy with the train_test_split function in the cross_validation library:

[7]:
```
# Split the iris data into train/test data sets with 40% reserved for testing
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,␣
 ↪test_size=0.4, random_state=0)
#iris.data here is the feature data, and iris.taret is the target. test size is 0.4␣
 ↪for testing purpose, and 0.6 for training purposes
# Guves you 4 data sets i.e train and test for both feature data and target data.

# Build an SVC model for predicting iris classifications using training data
clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train) #SVC model

# Now measure its performance with the test data
clf.score(X_test, y_test)
```

[7]:
```
0.9666666666666667
```

The score functon tells us we were right 96% of the time. These are stil small daa sets so lets use K-Fold Validation to protect against overfitting.

K-Fold cross validation is just as easy; let's use a K of 5:

[13]:
```
# We give cross_val_score a model, the entire data set and its "real" values, and␣
 ↪the number of folds:
scores = cross_val_score(clf, iris.data, iris.target, cv=5)
#call cross val score on cross validation package. Pass in all the dats. CV is␣
 ↪cross validation folds num score (5 training data set and reserves one for␣
 ↪testing, and reserves 1 for testing.)
# Print the accuracy for each fold:
print(scores)

# And the mean accuracy of all 5 folds:
print(scores.mean())
```

```
[0.96666667 1.         0.96666667 0.96666667 1.        ]
0.9800000000000001
```

We now see 98% accuracy.

Our model is even better than we thought! Can we do better? Let's try a different kernel (polynomial instead of linear):

[48]:
```
clf = svm.SVC(kernel='poly', degree=2, C=1).fit(X_train, y_train)
scores = cross_val_score(clf, iris.data, iris.target, cv=5)
print(scores)
print(scores.mean())
```

```
[0.96666667 1.         1.         0.96666667 1.        ]
0.9866666666666667
```

The more complex polynomial kernel produced higher accuracy than a simple linear kernel. The polynomial kernel is not overfitting (for this polynomial degree. It is for degree>4). But we couldn't have told that with a single train/test split:

[35]:
```
# Build an SVC model for predicting iris classifications using training data
clf = svm.SVC(kernel='poly', C=1).fit(X_train, y_train)

# Now measure its performance with the test data
clf.score(X_test, y_test)
```

[35]: 0.9

That's the same score we got with a single train/test split on the linear kernel.

9.4 Cleaning Your Input Data

How well you can do this has missive implications on how useful your results are. The reality is, much of your time as a data scientist will be spent preparing and "cleaning" your data. A lot of your time is spent dealing with:

- Outliers - behave strangely, and you shouldn't base this behaviour data informing your models. Missing Data - Weblog, might or might not have a refer. Create a new classification? Throw out the line entirely?
- Malicious Data - People may be trying to game your system. People might try to fabricate behaviour data to promote new item.
- Erroneous Data - Software bug writing out the wrong values. Dig into data that doesn't make sense to you. Are session ID's continuous for example.
- Irrelevant Data - Is the data relevant to what you're doing?
- Inconsistent Data - HUGE problem. e.g. Addresses, abbreviate street or not, people can input things in various ways. Books/movies may have different names in different countries.
- Formatting - Dates for example. Phone numbers, dashes, etc. Social Security Data. Make sure they're not treated as different entities.

Remember, Garbage In, Garbage Out. Your model is only as good as the data you give it. Look at your data! Examine it! Question your results! And always do this – not just when you don't get a result that you like!

Example. Let's analyze some web log data. All I want is the most-popular pages on a news website.

9.5 Cleaning Weblog data

Let's take a web access log, and figure out the most-viewed pages on a website from it! Sounds easy, right?

Let's set up a regex that lets us parse an Apache access log line:

[8]:
```
import re #good for pattern matching on large string

format_pat= re.compile(
    r"(?P<host>[\d\.]+)\s"
    r"(?P<identity>\S*)\s"
    r"(?P<user>\S*)\s"
    r"\[(?P<time>.*?)\]\s"
    r'"(?P<request>.*?)"\s'
    r"(?P<status>\d+)\s"
    r"(?P<bytes>\S*)\s"
    r'"(?P<referer>.*?)"\s'
    r'"(?P<user_agent>.*?)"\s*'
)
```

Here's the full path to the log file I'm analyzing. It is an an actual http log from Apache. Change this if you want to run this stuff yourself:

[9]:
```
logPath = r'C:\Users\Shav\Documents\Python Scripts\DataScience-Python3\access_log.
 ↪txt'
```

Now we'll whip up a little script to extract the URL in each access, and use a dictionary to count up the number of times each one appears. Then we'll sort it and print out the top 20 pages. What could go wrong?

[14]:
```
URLCounts = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            request = access['request']
            (action, URL, protocol) = request.split()
            if URLCounts.has_key(URL):
                URLCounts[URL] = URLCounts[URL] + 1
            else:
                URLCounts[URL] = 1

results = sorted(URLCounts, key=lambda i: int(URLCounts[i]), reverse=True)

for result in results[:20]:
    print(result + ": " + str(URLCounts[result]))
```

```
      ␣
↪--------------------------------------------------------------------------
```

```
AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-14-67b42e74cbab> in <module>
      8             request = access['request']
      9             (action, URL, protocol) = request.split()
---> 10            if URLCounts.has_key(URL):
     11                URLCounts[URL] = URLCounts[URL] + 1
     12            else:


AttributeError: 'dict' object has no attribute 'has_key'
```

Hm. The 'request' part of the line is supposed to look something like this:

GET /blog/ HTTP/1.1

Have some reqst fields that contain something else than action, URL, protocol.

There should be an HTTP action, the URL, and the protocol. But it seems that's not always happening. Let's print out requests that don't contain three items:

[13]:
```
URLCounts = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            request = access['request']
            fields = request.split()
            if (len(fields) != 3):
                print(fields)

                #this prints out requests where we dont have the required 3 fields.
```

```
['_\\xb0ZP\\x07tR\\xe5']
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
```

Huh. In addition to empty fields, there's one that just contains garbage. Well, let's modify our script to check for that case:

[15]:
```
URLCounts = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            request = access['request']
            fields = request.split()
            if (len(fields) == 3): #We throw out each entry which doesnt have 3␣
 ↪fields.
                URL = fields[1]
                if URL in URLCounts:
                    URLCounts[URL] = URLCounts[URL] + 1
                else:
                    URLCounts[URL] = 1

results = sorted(URLCounts, key=lambda i: int(URLCounts[i]), reverse=True)

for result in results[:20]:
    print(result + ": " + str(URLCounts[result]))
```

```
/xmlrpc.php: 68494
/wp-login.php: 1923
/: 440
/blog/: 138
/robots.txt: 123
/sitemap_index.xml: 118
/post-sitemap.xml: 118
/page-sitemap.xml: 117
/category-sitemap.xml: 117
/orlando-headlines/: 95
/san-jose-headlines/: 85
http://51.254.206.142/httptest.php: 81
/comics-2/: 76
/travel/: 74
/entertainment/: 72
/business/: 70
/national/: 70
/national-headlines/: 70
/world/: 70
/weather/: 70
```

It worked! But, the results don't really make sense. What we really want is pages accessed by real humans looking for news from our little news site. What the heck is xmlrpc.php? This is a news site, so this seems odd. A look at the log itself turns up a lot of entries like this:

php are pearl scripts. :440 is the home page. This is because it was actually under malicious attack, and the xmlrpc was them attempting to guess at the passwords. Trying to login using the log in script. This is an example of malicious data.

46.166.139.20 - - [05/Dec/2015:05:19:35 +0000] "POST /xmlrpc.php HTTP/1.0" 200 370 "-" "Mozil-

la/4.0 (compatible: MSIE 7.0; Windows NT 6.0)”

By looking you can see that not only was it php files, but it was also trying to execute code on the website, so it says “POST”. We want to gind GETTING requests, i.e people requesting to view a web page.

I’m not entirely sure what the script does, but it points out that we’re not just processing GET actions. We don’t want POSTS, so let’s filter those out:

[16]:
```
URLCounts = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            request = access['request']
            fields = request.split()
            if (len(fields) == 3):
                (action, URL, protocol) = fields
                if (action == 'GET'):
                    if URL in URLCounts:
                        URLCounts[URL] = URLCounts[URL] + 1
                    else:
                        URLCounts[URL] = 1

results = sorted(URLCounts, key=lambda i: int(URLCounts[i]), reverse=True)

for result in results[:20]:
    print(result + ": " + str(URLCounts[result]))
```

```
/: 434
/blog/: 138
/robots.txt: 123
/sitemap_index.xml: 118
/post-sitemap.xml: 118
/page-sitemap.xml: 117
/category-sitemap.xml: 117
/orlando-headlines/: 95
/san-jose-headlines/: 85
http://51.254.206.142/httptest.php: 81
/comics-2/: 76
/travel/: 74
/entertainment/: 72
/business/: 70
/national/: 70
/national-headlines/: 70
/world/: 70
/weather/: 70
/about/: 69
/defense-sticking-head-sand/: 69
```

That’s starting to look better. But, this is a news site - are people really reading the little blog on it

instead of news pages? That doesn't make sense. Let's look at a typical /blog/ entry in the log:

54.165.199.171 - - [05/Dec/2015:09:32:05 +0000] "GET /blog/ HTTP/1.0" 200 31670 "-" "-"

Hm. Why is the user agent blank? Seems like some sort of malicious scraper or something. Let's figure out what user agents we are dealing with:

[17]:
```
UserAgents = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            agent = access['user_agent']
            if agent in UserAgents:
                UserAgents[agent] = UserAgents[agent] + 1
            else:
                UserAgents[agent] = 1

results = sorted(UserAgents, key=lambda i: int(UserAgents[i]), reverse=True)

for result in results:
    print(result + ": " + str(UserAgents[result]))
```

```
Mozilla/4.0 (compatible: MSIE 7.0; Windows NT 6.0): 68484
-: 4035
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0): 1724
W3 Total Cache/0.9.4.1: 468
Mozilla/5.0 (compatible; Baiduspider/2.0;
+http://www.baidu.com/search/spider.html): 278
Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html): 248
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36: 158
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0) Gecko/20100101 Firefox/40.0: 144
Mozilla/5.0 (iPad; CPU OS 8_4 like Mac OS X) AppleWebKit/600.1.4 (KHTML, like
Gecko) Version/8.0 Mobile/12H143 Safari/600.1.4: 120
Mozilla/5.0 (Linux; Android 5.1.1; SM-G900T Build/LMY47X) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/46.0.2490.76 Mobile Safari/537.36: 47
Mozilla/5.0 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm): 43
Mozilla/5.0 (compatible; MJ12bot/v1.4.5; http://www.majestic12.co.uk/bot.php?+):
41
Opera/9.80 (Windows NT 6.0) Presto/2.12.388 Version/12.14: 40
Mozilla/5.0 (compatible; YandexBot/3.0; +http://yandex.com/bots): 27
Ruby: 15
Mozilla/5.0 (Linux; Android 5.1.1; SM-G900T Build/LMY47X) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/47.0.2526.76 Mobile Safari/537.36: 15
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36: 13
Mozilla/5.0 (compatible; AhrefsBot/5.0; +http://ahrefs.com/robot/): 11
Mozilla/5.0 (Windows NT 5.1; rv:6.0.2) Gecko/20100101 Firefox/6.0.2: 11
MobileSafari/600.1.4 CFNetwork/711.4.6 Darwin/14.0.0: 10
```

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.73 Safari/537.36: 9
Mozilla/5.0 (compatible; YandexImages/3.0; +http://yandex.com/bots): 9
Mozilla/5.0 (compatible; linkdexbot/2.0; +http://www.linkdex.com/bots/): 7
Mozilla/5.0 (iPhone; CPU iPhone OS 8_3 like Mac OS X) AppleWebKit/600.1.4 (KHTML, like Gecko) Version/8.0 Mobile/12F70 Safari/600.1.4 (compatible; Googlebot/2.1; +http://www.google.com/bot.html): 6
Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp): 6
Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.28) Gecko/20120306 Firefox/3.6.28 (.NET CLR 3.5.30729): 4
Mozilla/5.0 zgrab/0.x: 4
Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/30.0.1599.66 Safari/537.36: 4
Mozilla/5.0 (compatible; SeznamBot/3.2; +http://fulltext.sblog.cz/): 4
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1): 4
Mozilla/5.0: 3
Mozilla/5.0 (Windows NT 6.1; rv:34.0) Gecko/20100101 Firefox/34.0: 3
Opera/9.80 (Windows NT 5.1; U; ru) Presto/2.9.168 Version/11.50: 3
Mozilla/5.0 (compatible; spbot/4.4.2; +http://OpenLinkProfiler.org/bot): 3
Mozilla/4.0 (compatible: FDSE robot): 3
Mozilla/4.0 (compatible; MSIE 9.0; Windows NT 6.1; 2Pac; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022): 3
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:31.0) Gecko/20100101 Firefox/31.0: 3
Mozilla/5.0 (Windows NT 6.2; WOW64; rv:36.0) Gecko/20100101 Firefox/36.0: 2
Mozilla/5.0 (Windows NT 5.1; rv:36.0) Gecko/20100101 Firefox/36.0: 2
Mozilla/5.0 (Windows NT 6.1; rv:28.0) Gecko/20100101 Firefox/28.0: 2
Mozilla/5.0 (Windows NT 5.1; rv:2.0.1) Gecko/20100101 Firefox/5.0: 2
Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36: 2
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36: 2
Googlebot-Image/1.0: 2
netEstate NE Crawler (+http://www.website-datenbank.de/): 2
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.5) Gecko/20041107 Firefox/1.0: 2
Mozilla/5.0 (X11; U; FreeBSD x86_64; en-US) AppleWebKit/534.16 (KHTML, like Gecko) Chrome/10.0.648.204 Safari/534.16: 2
Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2228.0 Safari/537.36: 2
Opera/9.80 (Windows NT 6.1); U) Presto/2.10.289 Version/12.02: 2
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 5.1; Trident/5.0; .NET CLR 2.0.50727; .NET CLR 3.5.30729): 2
Mozilla/5.0 (Windows NT 6.2; rv:24.0) Gecko/20100101 Firefox/24.0: 2
Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/5.0; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E): 2
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0: 2
Mozilla/5.0 (Windows NT 5.1; rv:28.0) Gecko/20100101 Firefox/28.0: 2
Mozilla/5.0 (Windows NT 6.0; rv:29.0) Gecko/20120101 Firefox/29.0: 2
Mozilla/5.0 (Windows NT 6.0; rv:31.0) Gecko/20100101 Firefox/31.0: 2
Mozilla/5.0 (Windows NT 6.2; rv:31.0) Gecko/20100101 Firefox/31.0: 2

```
Mozilla/4.0 (compatible; Netcraft Web Server Survey): 2
Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; MAGWJS; rv:11.0) like Gecko: 1
Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/37.0.2062.120 Safari/537.36: 1
Mozilla/5.0 (X11; U; Linux i686; pl-PL; rv:1.9.0.2) Gecko/20121223 Ubuntu/9.25
(jaunty) Firefox/3.8: 1
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:42.0) Gecko/20100101 Firefox/42.0: 1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/41.0.2251.0 Safari/537.36: 1
Opera/9.80 (Windows NT 6.2; WOW64); U) Presto/2.12.388 Version/12.14: 1
Mozilla/5.0 (Windows NT 6.1; rv:33.0) Gecko/20100101 Firefox/33.0: 1
Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.9.0.13) Gecko/2009073022
Firefox/3.0.13 (.NET CLR 3.5.30729): 1
Scrapy/1.0.3 (+http://scrapy.org): 1
Mozilla/4.0 (compatible; MSIE 6.0; MSIE 5.5; Windows 95) Opera 7.03 [de]: 1
Telesphoreo: 1
Mozilla/5.0 (Windows NT 6.1; rv:31.0) Gecko/20100101 Firefox/31.0: 1
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/47.0.2526.73 Safari/537.36: 1
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.3.1.1) Gecko/20101203
Firefox/3.6.12 (.NET CLR 3.5.30309): 1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/46.0.2490.86 Safari/537.36 Scanning for research
(researchscan.comsys.rwth-aachen.de): 1
Mozilla/5.0 (Windows NT 6.1; rv:22.0) Gecko/20130405 Firefox/22.0: 1
Mozilla/5.0 (Windows; U; Windows NT 5.0; fr-FR; rv:0.9.4) Gecko/20011019
Netscape6/6.2: 1
Mozilla/5.0 (Windows NT 5.1; rv:32.0) Gecko/20100101 Firefox/31.0: 1
facebookexternalhit/1.1 (+http://www.facebook.com/externalhit_uatext.php): 1
Mozilla/3.0 (compatible; Indy Library): 1
Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:42.0) Gecko/20100101 Firefox/42.0: 1
Mozilla/5.0 (Windows NT 5.1; rv:5.0) Gecko/20100101 Firefox/5.0: 1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/534.59.10 (KHTML,
like Gecko) Version/5.1.9 Safari/534.59.10: 1
Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko: 1
NokiaE5-00/SymbianOS/9.1 Series60/3.0 3gpp-gba: 1
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/45.0.2454.101 Safari/537.36: 1
Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/45.0.2454.101 Safari/537.36: 1
```

Yikes! In addition to '-', there are also a million different web robots accessing the site and polluting my data. Things like spiders, from web crawlers. Yandex, Googlebot, Bandu for search engine purposes. These shouldnt count towoards the inteded purpose of analysis. Filtering out all of them is really hard, but getting rid of the ones significantly polluting my data in this case should be a matter of getting rid of '-', anything containing "bot" or "spider", and W3 Total Cache.

[18]:
```
URLCounts = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
```

```
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            agent = access['user_agent']
            if (not('bot' in agent or 'spider' in agent or
                    'Bot' in agent or 'Spider' in agent or
                    'W3 Total Cache' in agent or agent =='-')): #refine script to␣
 ↪take out user agnts that look fishy
                request = access['request']
                fields = request.split()
                if (len(fields) == 3):
                    (action, URL, protocol) = fields
                    if (action == 'GET'):
                        if URL in URLCounts:
                            URLCounts[URL] = URLCounts[URL] + 1
                        else:
                            URLCounts[URL] = 1

results = sorted(URLCounts, key=lambda i: int(URLCounts[i]), reverse=True)

for result in results[:20]:
    print(result + ": " + str(URLCounts[result]))
```

```
/: 77
/orlando-headlines/: 36
/?page_id=34248: 28
/wp-
content/cache/minify/000000/M9AvyUjVzUstLy7PLErVz8lMKkosqtTPKtYvTi7KLCgpBgA.js:
27
/wp-content/cache/minify/000000/M9bPKixNLarUy00szs8D0Zl5AA.js: 27
/wp-content/cache/minify/000000/lY7dDoIwDIVfiG0KxkfxfnbdKO4HuxICTy-
it8Zw15PzfSftzPCckJem-x4qUWArqBPl5mygZLEgyhdOaoxToGyGaiALiOfUnIz0qDLOdSZGE-nOlpc
3kopDzrSyavVVt_veb5qSDVhjsQ6dHh_B_eE_z2pYIGJ7iBWKeEio_eT9UQe4xHhDll27mGRryVu_pRc
.js: 27
/wp-content/cache/minify/000000/fY45DoAwDAQ_FMvkRQgFA5ZyWLajiN9zNHR0O83MRkyt-
pIctqYFJPedKyYzfHg2PzOFiENAzaD07AxcpKmTolORvDjZt8KEfhBUGjZYCf8Fb0fvA1TXCw.css:
25
/?author=1: 21
/wp-content/cache/minify/000000/hcrRCYAwDAXAhXyEjiQ1YKAh4SVSx3cE7_uG7ASr4M9qg3kG
Wyk1adklK84LHtRj_My6YOPfqcz-AA.js: 20
/wp-content/uploads/2014/11/nhn1.png: 19
/wp-includes/js/wp-emoji-release.min.js?ver=4.3.1: 17
/wp-content/cache/minify/000000/BcGBCQAgCATAiUSaKYSERPk3avzuht4SkBJnt4tHJdqgnPBq
KldesTcN1R8.js: 17
/wp-login.php: 16
/comics-2/: 12
/world/: 12
/favicon.ico: 10
/wp-content/uploads/2014/11/violentcrime.jpg: 6
/robots.txt: 6
```

```
/wp-content/uploads/2014/11/garfield.jpg: 6
/wp-content/uploads/2014/11/babyblues.jpg: 6
```

Now, our new problem is that we're getting a bunch of hits on things that aren't web pages, like scripts. We're not interested in those, so let's filter out any URL that doesn't end in / (all of the pages on my site are accessed in that manner - again this is applying knowledge about my data to the analysis!)

[19]:
```
URLCounts = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            agent = access['user_agent']
            if (not('bot' in agent or 'spider' in agent or
                    'Bot' in agent or 'Spider' in agent or
                    'W3 Total Cache' in agent or agent =='-')):
                request = access['request']
                fields = request.split()
                if (len(fields) == 3):
                    (action, URL, protocol) = fields
                    if (URL.endswith("/")): #needds to end in a slash
                        if (action == 'GET'):
                            if URL in URLCounts:
                                URLCounts[URL] = URLCounts[URL] + 1
                            else:
                                URLCounts[URL] = 1

results = sorted(URLCounts, key=lambda i: int(URLCounts[i]), reverse=True)

for result in results[:20]:
    print(result + ": " + str(URLCounts[result]))
```

```
/: 77
/orlando-headlines/: 36
/comics-2/: 12
/world/: 12
/weather/: 4
/australia/: 4
/about/: 4
/national-headlines/: 3
/feed/: 2
/sample-page/feed/: 2
/science/: 2
/technology/: 2
/entertainment/: 1
/san-jose-headlines/: 1
/business/: 1
/travel/feed/: 1
```

This is starting to look more believable! But if you were to dig even deeper, you'd find that the /feed/ pages are suspect, and some robots are still slipping through. However, it is accurate to say that Orlando news, world news, and comics are the most popular pages accessed by a real human on this day.

The moral of the story is - know your data! And always question and scrutinize your results before making decisions based on them. If your business makes a bad decision because you provided an analysis of bad source data, you could get into real trouble.

Be sure the decisions you make while cleaning your data are justifiable too - don't strip out data just because it doesn't support the results you want!

These results still aren't perfect; URL's that include "feed" aren't actually pages viewed by humans. Modify this code further to strip out URL's that include "/feed". Even better, extract some log entries for these pages and understand where these views are coming from.

[39]:
```
URLCounts = {}

with open(logPath, "r") as f:
    for line in (l.rstrip() for l in f):
        match= format_pat.match(line)
        if match:
            access = match.groupdict()
            agent = access['user_agent']
            if (not('bot' in agent or 'spider' in agent or
                    'Bot' in agent or 'Spider' in agent or
                    'W3 Total Cache' in agent or agent =='-')):
                request = access['request']
                fields = request.split()
                if (len(fields) == 3):
                    (action, URL, protocol) = fields
                    if (URL.endswith("/" or "/feed/")) and not (URL.endswith("/feed/
 ↪")): #needds to end in a slash
                        if (action == 'GET'):
                            if URL in URLCounts:
                                URLCounts[URL] = URLCounts[URL] + 1
                            else:
                                URLCounts[URL] = 1

results = sorted(URLCounts, key=lambda i: int(URLCounts[i]), reverse=True)

for result in results[:20]:
    print(result + ": " + str(URLCounts[result]))
```

```
/: 77
/orlando-headlines/: 36
/comics-2/: 12
/world/: 12
/weather/: 4
/australia/: 4
/about/: 4
/national-headlines/: 3
```

```
/science/: 2
/technology/: 2
/entertainment/: 1
/san-jose-headlines/: 1
/business/: 1
```

9.6 Normalizing Numerical Data

Sometimes it is useful to normalise or "whiten" your data before inputting it into your algorithm. Sometimes various input feature data should be on the same scale and comparable.

Sometimes models are based on differnt numerical attributes. Like multivariant models, e.g. differnet attributes of a car. Ages vs Income.

Example: ages may range from 0-100, and incomes from 0-billions. Some models may not perform well when different attributes are on very different scales. It can result in some attributes counting more than others. This means that bias in the attributes can also be a problem, and makes your data skewed.

Scikit-learn's PCA implementation has a "whiten" option that does this for you. Use it. Scikit-learn has a preprocessing module with handy normalize and scale functions. Your data may have "yes" and "no" that needs to be converted to "1" and "0"

Read the documentation. Most data mining and machine learning techniques work fine with raw, un-normalized data, but double check the one you're using before you start. Don't forget to re-scale your results when you're done!

9.7 Dealing with Outliers

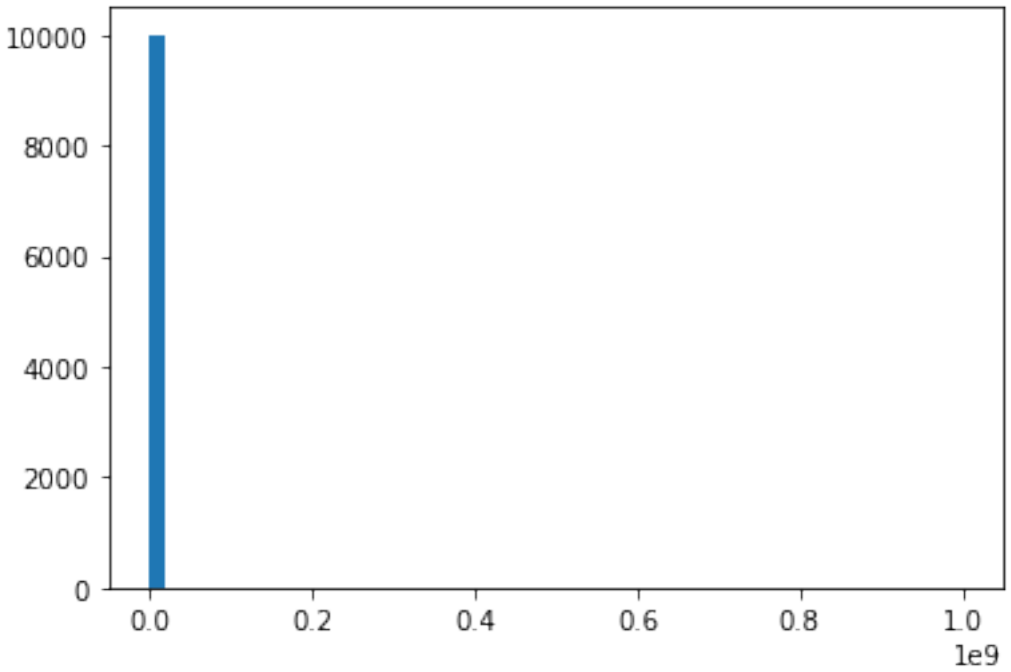
You'll always have atypical users. Sometimes it's appropriate to remove outliers from your training data. Do this responsibly! Understand why you are doing this. For example: in collaborative filtering, a single user who rates thousands of movies could have a big effect on everyone else's ratings. That may not be desirable. (inordinate influence on your system).

Another example: in web log data, outliers may represent bots or other agents that should be discarded. But if someone really wants the mean income of US citizens for example, don't toss out Donald Trump just because you want to.

Our old friend **standard deviation** provides a principled way to classify outliers. Find data points more than some multiple of a standard deviation in your training data. What multiple? You just have to use common sense.

Sometimes outliers can mess up an analysis; you usually don't want a handful of data points to skew the overall results. Let's revisit our example of income data, with Donald Trump thrown in:

[1]:
```
%matplotlib inline
import numpy as np

incomes = np.random.normal(27000, 15000, 10000)
incomes = np.append(incomes, [1000000000]) #outlier at 1 billion

import matplotlib.pyplot as plt
plt.hist(incomes, 50)
plt.show()
```



That's not very helpful to look at. One billionaire ended up squeezing everybody else into a single line in my histogram. Plus it skewed my mean income significantly:

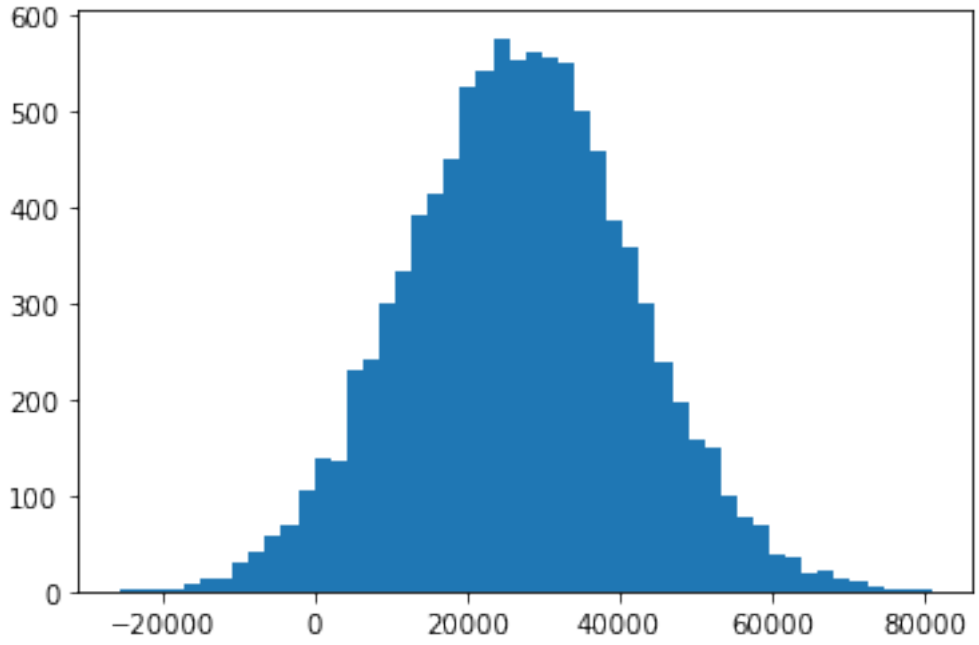
[2]:
```
incomes.mean()
```

[2]:
```
126911.6881323548
```

It's important to dig into what is causing your outliers, and understand where they are coming from. You also need to think about whether removing them is a valid thing to do, given the spirit of what it is you're trying to analyze. If I know I want to understand more about the incomes of "typical Americans", filtering out billionaires seems like a legitimate thing to do. It is important to be aware why you are throwing out outliers.

Here's something a little more robust than filtering out billionaires - it filters out anything beyond two standard deviations of the median value in the data set:

[3]:
```
def reject_outliers(data):
    u = np.median(data)
    s = np.std(data)
    filtered = [e for e in data if (u - 2 * s < e < u + 2 * s)]
    return filtered

filtered = reject_outliers(incomes)

plt.hist(filtered, 50)
plt.show()
```



That looks better. And, our mean is more, well, meangingful now as well:

[4]:
```
np.mean(filtered)
```

[4]:
```
26726.214626383888
```

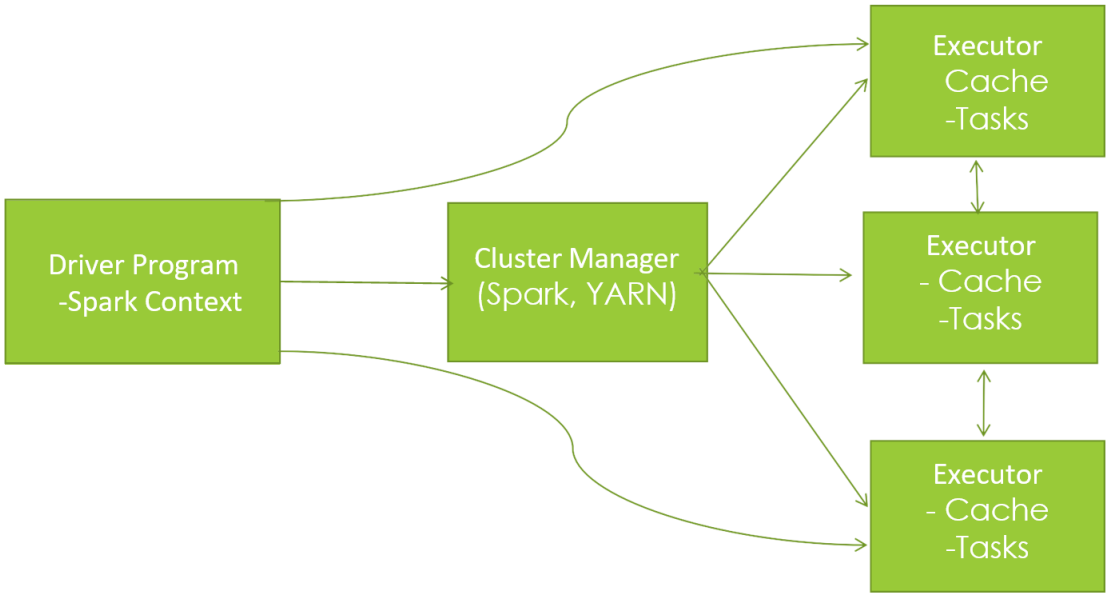
10 Apache Spark Machine Learning on Big Data

10.1 Spark Introduction

"Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Scala, Java, Python, and R, and an optimized engine that supports general computation graphs for data analysis. It also supports a rich set of higher-level tools including Spark SQL for SQL and DataFrames, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for stream processing."

This is quite generic. Spark essentially is a framework for writing jobs/scripts that can process very large amounts of data and it manages distributing that processing across a cluster of computing for you. It allows you to load your data in to RDD (resilient distributed data stores) and automatically transform them. The beauty of it is that it will optimally spread the processing out about a cluster of computers. You're no longer limited to what a single machines memory can do.

How is it scaleable?



Driver program is a script that uses the spark library. This is what you write. You define the spark context, the root object you work within when developing in Spark. The spark framework then takes over and distributes things for you.

Spark actually has its own cluster manager built in, so you can use it without Hadoop installed. YARN is a component of Hadoop, that is just separating out the entire cluster management piece of Hadoop. Spark can interface with YARN to optimally distribute the processing among the resource's available to the cluster.

Withing the cluster, you may have executor tasks running, and they may be running on different computers, or different cores of the same computers. Driver program (Spark context) and the Cluster Manager (YARN) work together to coordinate the effort and return a result to you.

Whats the big deal? Spark is *fast*. "Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk". It also has a DAG Engine (directed acyclic graph) which optimizes workflows. i.e. Nothing actually happens until you actually perform an action on the data. It waits until you actually ask it to produce a result, and only then does it try to figure out how to produce those results.

Spark is hot. Companies like Amazon, Ebay (log analysis and aggregation), NASA (JPL: Deep Space Network), Groupon, TripAdviser & Yahoo all claim to use it.

You have the choice to code in Python, Java, or Scala. Built around one main concept: the Resilient Distributed Dataset (RDD). There are many components of Spark. (other than spark core, which can by itself do quite a bit)

- Spark streaming - allow you to process data in real time. Data may be flowing into a server continuously real time, say from weblogs, and spark can process that in real time.
- SparkSQL - allows you to treat data as SQL database and perform SQL queries of it.
- MLLib - is a machine learning library allowing you to perform common machine learning algorithms with spark under the hood to distribute the processing across a cluster.
- GraphX - refers to graph in network theory sense e.g social network.

10.1.1 Python vs. Scala

Why Python? A lot of people develop in Scala. The reason for Python is because it is easier; No need to compile, manage dependencies, etc. There is less coding overhead.

Plus you already know Python. However, Scala is probably a more popular choice with Spark. Spark is built in Scala, so coding in Scala is "native" to Spark. Also, new features, libraries tend to be Scala-first.

No need to worry though, as Python and Scala look very similar in Spark.

10.2 Resilient Distributed Dataset (RDD)

Core/fundamental of coding in spark. Spark makes sure that it is resilient. If one of your computers goes down/unstable network or cluster, it will retry. Distributed, and can be used for big data.

Start Spark script by getting a SparkContext object. This embodies the guts of spark - generates the RDD's to use in processing. Created by your driver program, is responsible for making RDD's resilient and distributed!

Creating RDD's:

- nums = parallelize([1, 2, 3, 4]) - the simplest case of a RDD. Defeats the purpose of big data though, i.e. if you have to load the data into memory before you can create an RDD from it, whats the point?
- sc.textFile("file:///c:/users/frank/gobs-o-text.txt") - Every line/row into an RDD with a row with one line of text. Then you can parse / break out delimiters. This is an example of load, transform on system itself, and then use. i.e. ETL and ELT. Can also use s3n:// , hdfs:// - can also use distributed s3 amazon bucket or a distributed hdfs cluster.
- hiveCtx = HiveContext(sc) rows = hiveCtx.sql("SELECT name, age FROM users") - Can talk to Hive. RDD from rows can be generated by executing a SQL query on your Hive database.

Can also create from:

- JDBC
- Cassandra
- HBase
- Elastisearch
- JSON, CSV, sequence files, object files, various compressed formats

10.2.1 Transformations

You can do 2 classes of things to RDD's - Transformations and Actions. Transforming RDD's means taking values and changing the values to a new value based on a function you provide. This includes

- map
- flatmap - both take any function and output a transformed row. Key is you can chain map, and have sequential operations. Flatmap vs Map - map only allows 1 output per row, by Flapmap allows multiple new rows for a given row. Can change size of RDD
- filter - creates Boolean - e.g. should this row be preserved
- distinct - returns only distinct values in RDD
- sample - take a sample
- union
- Intersection, subtract, cartesian

Map() example:

- rdd = sc.parallelize([1, 2, 3, 4])
- rdd.map(lambda x: x*x)
- This yields 1, 4, 9, 16

Reminder of lambda functions, they are just shorthad for defining a function in-line. Many RDD methods accept a function as a parameter. rdd.map(lambda x: x*x) Is the same thing as def squareIt(x): // return x*x // rdd.map(squareIt)

This is "functional programming".

10.2.2 Actions

Actions on RDD's include:

- collect - allows you to get the result of an RDD of a Python Object.
- count - Counts entries in RDD
- countByValue - how many times a certain unique value occurs
- take - samples. Random number of entries from your RDD
- top - Get a peek for debugging purposes
- reduce - powerful. Combine values together for a common key values.
- ... and a lot more

Remember though! Nothing actually happens in your driver program until an action is called! This may trip you up when writing Spark scripts.

10.3 Introducing MLLib

Built in component on top of spark core that makes it easy to perform complicated machine learning algorithms on large data sets and distribute that among a cluster of computers. What are some things MLLibe can do?

- Feature extraction - Term Frequency / Inverse Document Frequency stuff useful for search engines
- Basic statistics like Chi-squared test, Pearson or Spearman correlation, min, max, mean, variance/ Terribly exciting computing on a massive data set.
- Linear regression, logistic regression
- Support Vector Machines
- Naïve Bayes classifier
- Decision trees
- K-Means clustering
- Principal component analysis, singular value decomposition
- Recommendations using Alternating Least Squares

MLLib does introduce some new data types that you need to be aware of however. These include:

- Vector (dense or sparse) (if it had a value or not. Dense will present data for every possible combination. stores a missing value, which takes up space. A sparse vector is more compact form, but it introduces some complexity when processing. Good to ues if you know there are lots of empty data points)
- LabeledPoint is a point with a human meaning conveyed in the label.
- Rating

10.4 Decision Trees in Spark

[ ]:
```
from pyspark.mllib.regression import LabeledPoint #need these classes
from pyspark.mllib.tree import DecisionTree
from pyspark import SparkConf, SparkContext #needed in most spark files
from numpy import array #make sure these libraries are all installed on every␣
 ↪machine you want to run on

# Boilerplate Spark stuff:
conf = SparkConf().setMaster("local").setAppName("SparkDecisionTree") #master node␣
 ↪is local - only n local deskt and a name
sc = SparkContext(conf = conf) #create the object

# Some functions that convert our CSV input data into numerical
# features for each job candidate
def binary(YN):
    if (YN == 'Y'):
        return 1
    else:
        return 0

def mapEducation(degree):
    if (degree == 'BS'):
        return 1
    elif (degree =='MS'):
        return 2
    elif (degree == 'PhD'):
        return 3
    else:
        return 0

# Convert a list of raw fields from our CSV file to a
# LabeledPoint that MLLib can use. All data must be numerical...
def createLabeledPoints(fields):
    yearsExperience = int(fields[0])
    employed = binary(fields[1])
    previousEmployers = int(fields[2])
    educationLevel = mapEducation(fields[3])
    topTier = binary(fields[4])
    interned = binary(fields[5])
    hired = binary(fields[6])

    return LabeledPoint(hired, array([yearsExperience, employed,
        previousEmployers, educationLevel, topTier, interned]))

#Load up our CSV file, and filter out the header line with the column names
rawData = sc.textFile("e:/shav/udemy/datascience/PastHires.csv") #RDD creation
header = rawData.first() #extract the first line for defining categories
rawData = rawData.filter(lambda x:x != header) #filter funciton - only allows lines␣
 ↪through that dont match up to the header row
```

```
# Split each line into a list based on the comma delimiters
csvData = rawData.map(lambda x: x.split(",")) #map function. Split into individual␣
 ↪fields into python list

# Convert these lists to LabeledPoints for Mllib to work
trainingData = cbsvData.map(createLabeledPoints) #transforms inputs to a bunch of␣
 ↪numbres

# Create a test candidate, with 10 years of experience, currently employed,
# 3 previous employers, a BS degree, but from a non-top-tier school where
# he or she did not do an internship. You could of course load up a whole
# huge RDD of test candidates from disk, too.
testCandidates = [ array([10, 1, 3, 1, 0, 0])]
testData = sc.parallelize(testCandidates)

# Train our DecisionTree classifier using our data set
model = DecisionTree.trainClassifier(trainingData, numClasses=2,
                                     categoricalFeaturesInfo={1:2, 3:4, 4:2, 5:2},
                                     impurity='gini', maxDepth=5, maxBins=32)

# Now get predictions for our unknown candidates. (Note, you could separate
# the source data into a training set and a test set while tuning
# parameters and measure accuracy as you go!)
predictions = model.predict(testData)
print('Hire prediction:')
results = predictions.collect()
for result in results:
    print(result)

# We can also print out the decision tree itself:
print('Learned classification tree model:')
print(model.toDebugString())
```

To run this, go to the command prompt with all the environment variables in place. Make the working directory the corect one with your .py file, and type spark-submit followed by the file name.

Can do the same for K-Means Clustering.

10.5 TF-IDF

Stands for “Term Frequency and Inverse Document Frequency”. Important data for search – figures out what terms are most relevant for a document. (relevancy of a word for a given document).

Term Frequency just measures how often a word occurs in a document. A word that occurs frequently is probably important to that document’s meaning. *Document Frequency* is how often a word occurs in an entire set of documents, i.e., all of Wikipedia or every web page. This tells us about common words that just appear everywhere no matter what the topic, like “a”, “the”, “and”, etc.

So a measure of the relevancy of a word to a document might be:

$$TermFrequency / DocumentFrequency$$

Or:

$$TermFrequency * InverseDocumentFrequency$$

That is, take how often the word appears in a document, over how often it just appears everywhere. That gives you a measure of how important and unique this word is for this document

TF-IDF In Practice: We actually use the log of the IDF, since word frequencies are distributed exponentially. That gives us a better weighting of a words overall popularity. Limittions however incude how TF-IDF assumes a document is just a "bag of words". Parsing documents into a bag of words can be most of the work because you have to deal with tenses and synonyms etc.

Words can be represented as a hash value (number) for efficiency. What about synonyms? Various tenses? Abbreviations? Capitalizations? Misspellings? Doing this at scale is the hard part, but that's where Spark comes in!

How do you actually turn this into a search problem? A very simple search algorithm could be: Compute TF-IDF for every word in the entire body of documents that we have (corpus). For a given search word, sort the documents by their TF-IDF score for that word and display the results.

10.6 Create a Working, Scaleable Search Algorithm using MLLib in Spark

[ ]:
```
from pyspark import SparkConf, SparkContext
from pyspark.mllib.feature import HashingTF #compute term frequency and IDF
from pyspark.mllib.feature import IDF

# Boilerplate Spark stuff: #local spark configuration, and spark context
conf = SparkConf().setMaster("local").setAppName("SparkTFIDF")
sc = SparkContext(conf = conf)

# Load documents (one per line).
rawData = sc.textFile("C:/Shav/Documents/DataScience/subset-small.tsv") #import as␣
 ↪RDD
fields = rawData.map(lambda x: x.split("\t")) #This tsv file contains a whole␣
 ↪Wikipedia article on every line
documents = fields.map(lambda x: x[3].split(" ")) #field 3 (body) is split by␣
 ↪spaces

# Store the document names for later:
documentNames = fields.map(lambda x: x[1]) #extract document name for lookup later

# Now hash the words in each document to their term frequencies:
hashingTF = HashingTF(100000)  #100K hash buckets just to save some memory instead␣
 ↪of keeping them internally as strings
tf = hashingTF.transform(documents) #i.e map words to numbers

# At this point we have an RDD of sparse vectors representing each document, i.e.␣
 ↪no missing data
# where each value maps to the term frequency of each unique hash value.

# Let's compute the TF*IDF of each term in each document:
tf.cache() #cache as we will use it more than once
idf = IDF(minDocFreq=2).fit(tf) #ignore every word that doest appear twice
```

```
tfidf = idf.transform(tf)

# Now we have an RDD of sparse vectors, where each value is the TFxIDF
# of each unique hash value for each document.

# I happen to know that the article for "Abraham Lincoln" is in our data
# set, so let's search for "Gettysburg" (Lincoln gave a famous speech there):

# First, let's figure out what hash value "Gettysburg" maps to by finding the
# index a sparse vector from HashingTF gives us back:
gettysburgTF = hashingTF.transform(["Gettysburg"])
gettysburgHashValue = int(gettysburgTF.indices[0])

# Now we will extract the TF*IDF score for Gettsyburg's hash value into
# a new RDD for each document:
gettysburgRelevance = tfidf.map(lambda x: x[gettysburgHashValue])

# We'll zip in the document names so we can see which is which:
zippedResults = gettysburgRelevance.zip(documentNames)

# And, print the document with the maximum TF*IDF value:
print("Best document for Gettysburg is:")
print(zippedResults.max())
```

10.7 Data Sets Data Frames

Like RDD's but has structured data - has a defined schema to it I knows ahead of time what columns exist in each row of a data set, and what types those are. Because it knows the structure of the data set ahead of time, it can optimise more efficiently. We can also issue SQL's on it. And it creates a higher level API wit which we can

11 Experimental Design

11.1 A/B Tests

A/B tests are a controlled experiment, usually in the context of a website, to measure the impact of a given change. You test the performance of some change to your website (the variant) and measure conversion relative to your unchanged site (the control.). E.g. What colour should a button be. You can also test:

- Design changes
- UI flow / Purchase Pipeline
- Algorithmic changes
- Pricing changes
- You name it

It is important to understand what you are trying to drive with this change. Ideally choose what you are trying to influence:

- Order amounts
- Profit
- Ad Clicks

- Order Quantity

But attributing actions downstream from your change can be hard, especially if you're running more than one experiment. Very easy to produce misleading results.

Here, *Variance* is your Enemy. A common mistake to make is to run a test for some small period of time that results in a few purchases to analyze. You take the mean order amount from A and B, and declare victory or defeat. **But**, there's so much random variation in order amounts to begin with, that your result was just based on chance. You then fool yourself into thinking some change to your website, which could actually be harmful, has made tons of money.

Sometimes you need to also look at conversion metrics with less variance. Order quantities vs. order dollar amounts, for example. At the end of the day thats your judgement.

11.2 T-Tests and P-Values

How do we know if a result is likely to be "real" as opposed to just random variation? T-tests and P-values.

The T-Statistic is a measure of the difference between the two sets expressed in units of standard error i.e. The size of the difference relative to the variance in the data. A high t value means there's probably a real difference between the two sets.

With this you assume a normal distribution of behavior. This is a good assumption if you're measuring revenue as conversion. See also: Fisher's exact test (for clickthrough rates), E-test (for transactions per user) and chi-squared test (for product quantities purchased).

The P-Value, can be thought of as the probability of A and B satisfying the "null hypothesis". So, a low P-Value implies significance. It is the probability of an observation lying at an extreme t-value assuming the null hypothesis.

Using P-values has to be done by choosing some threshold for "significance" before your experiment. 1%? 5%? When your experiment is over, measure your P-value. If it's less than your significance threshold, then you can reject the null hypothesis. if it's a positive change, roll it out. If it's a negative change, discard it before you lose more money.

11.3 T-Tests and P-Values in Python

Let's say we're running an A/B test. We'll fabricate some data that randomly assigns order amounts from customers in sets A and B, with B being a little bit higher. A is our treatment group and B is our control.

[9]:
```
import numpy as np
from scipy import stats

A = np.random.normal(25.0, 5.0, 10000)
B = np.random.normal(26.0, 5.0, 10000)

stats.ttest_ind(A, B)
```

[9]:
```
Ttest_indResult(statistic=-13.871706679725438, pvalue=1.496269828122858e-43)
```

The t-statistic is a measure of the difference between the two sets expressed in units of standard error. Put differently, it's the size of the difference relative to the variance in the data. A high t value means there's probably a real difference between the two sets; you have "significance". The P-value is a measure of the probability of an observation lying at extreme t-values; so a low p-value also

implies "significance." If you're looking for a "statistically significant" result, you want to see a very low p-value and a high t-statistic (well, a high absolute value of the t-statistic more precisely). In the real world, statisticians seem to put more weight on the p-value result.

Let's change things up so both A and B are just random, generated under the same parameters. So there's no "real" difference between the two:

[10]:
```
B = np.random.normal(25.0, 5.0, 10000)

stats.ttest_ind(A, B)
```

[10]:
```
Ttest_indResult(statistic=1.0538624380972859, pvalue=0.2919586346703036)
```

Now, our t-statistic is much lower and our p-value is really high. This supports the null hypothesis - that there is no real difference in behavior between these two sets.

Does the sample size make a difference? Let's do the same thing - where the null hypothesis is accurate - but with 10X as many samples:

[52]:
```
A = np.random.normal(25.0, 5.0, 100000)
B = np.random.normal(25.0, 5.0, 100000)

stats.ttest_ind(A, B)
```

[52]:
```
Ttest_indResult(statistic=1.5331544648848152, pvalue=0.12523938242504115)
```

Our p-value actually got a little lower, and the t-test a little larger, but still not enough to declare a real difference. So, you could have reached the right decision with just 10,000 samples instead of 100,000. Even a million samples doesn't help, so if we were to keep running this A/B test for years, you'd never acheive the result you're hoping for:

[58]:
```
A = np.random.normal(25.0, 5.0, 1000000)
B = np.random.normal(25.0, 5.0, 1000000)

stats.ttest_ind(A, B)
```

[58]:
```
Ttest_indResult(statistic=-0.7320744173634869, pvalue=0.4641232332765902)
```

If we compare the same set to itself, by definition we get a t-statistic of 0 and p-value of 1:

[5]:
```
stats.ttest_ind(A, A)
```

[5]:
```
Ttest_indResult(statistic=0.0, pvalue=1.0)
```

The threshold of significance on p-value is really just a judgment call. As everything is a matter of probabilities, you can never definitively say that an experiment's results are "significant". But you can use the t-test and p-value as a measure of signficance, and look at trends in these metrics as the experiment runs to see if there might be something real happening between the two.

11.4 How Long Do I Run an Experiment?

When you run an experiment, it is easy to have a bias/vested interest in seeing the success of the new thing you've implemented. How do I know when I'm done with an A/B test? Dont fall into the trap of running it indefinitely in hopes of seeing a positive result. Draw the line when:

- You have achieved significance (positive or negative)
- You no longer observe meaningful trends in your p-value i.e. That is, you don't see any indication that your experiment will "converge" on a result over time. If you don't see a trend, you wont see a measurable result no matter how long you run it. Plot your value over time. If it is coming down, its working etc.
- You reach some pre-established upper bound on time

11.5 Drawbacks of A/B Testing

11.5.1 Correlation does not imply causation!

Even your low p-value score on a well-designed experiment does not imply causation! It could still be random chance, and you could have other factors at play. It's your duty to ensure business owners understand this as well if you're working for someone.

11.5.2 Novelty Effects

are when changes to a website catch the attention of previous users who are used to the way it used to be simply because it is *new*. They might click on something simply because it is new but this attention won't last forever. The Achilles heel of A/B is the short time frame it is run. It is a good idea to re-run experiments much later and validate their impact. Often the "old" website will outperform the "new" one after awhile, simply because it is a change. In this context at least, being different is not a virtue.

11.5.3 Seasonal Effects

An experiment run over a short period of time may only be valid for that period of time. An example: Consumer behavior near Christmas is very different than other times of year. An experiment run near Christmas may not represent behavior during the rest of the year. Can look at this quantitatively, by looking at the metric you re measuring, and look at it last year without the change to see fluctuations.

11.5.4 Selection Bias

Sometimes your random selection of customers for A or B isn't really random. For example: assignment is based somehow on customer ID but customers with low ID's are better, long time, more loyal customers than ones with high ID's. Run an A/A test periodically to check and audit your segment assignment algorithms. Also make sure they do not change groups between clicks. Could use google experiment etc.

11.5.5 Data Pollution

Are robots/crawlers (both self-identified and malicious) affecting your experiment? A good reason to measure conversion based on something that requires spending real money. More generally, are outliers skewing the result?

11.5.6 Attribution Errors

Often there are errors in how conversion is attributed to an experiment. Using a widely used A/B test platform can help mitigate that risk. If your framework is home-grown, it deserves auditing. Watch for "gray areas" i.e. Are you counting purchases toward an experiment within some given timeframe of exposure to it? Is that time too large? Could other changes downstream from the change

you're measuring affect your results? Are you running multiple experiments at once? Is downstream behaviour from a change a good reflection?

Using an off the shelf experiment framework like Google Experiments or Optimizely can help mitigate some of these issues.

12 Deep Learning and Neural Networks

12.1 Pre-Requisites

You will need to remember the content from PMSO, i.e. optimization techniques. Local and global optimisation. Called "gradient descent" here. Also, it is best to familiarise yourself with "autodiff".

Overall:

- Gradient descent is an algorithm for minimizing error over multiple steps. Technique to find local minima
- Autodiff is a calculus trick for finding the gradients in gradient descent. Tensorflow uses this. Accelerates gradient descent.
- Softmax is a function for choosing the most probable classification given several input values

Gradient descent requires knowledge of the gradient from your cost function (MSE). Mathematically we need the first partial derivatives of all the inputs, but this is hard and inefficient if you just throw calculus at the problem. Reverse-mode autodiff to the rescue!

Computes all the partial derivatives you need just by traversing the graph in the number of outputs + 1 that you have. (Computes all partial derivatives in of outputs + 1 graph traversals.) This is useful in neural networks because in neural networks, you tend to have artificial neurons that have very many inputs and a few outputs (normally 1). Still fundamentally though a calculus trick – it's complicated but it works. This is what Tensorflow library uses under the hood, and will be importnt to understand when doing deep learning.

Overall, gradient descent is the technique were using to find the local minima of what we are trying to optimise for.

softmax is imprtant to understand; what it is and what it is for? when you have the end result of a neural network you end up with weights. how we make use of that? How do we make practical use of the output of our neural networks? That's where softmax comes in.

It converts each of the final weights that come out of your neural network into a probability. If you are trying to calssify something in your neural network, for example to decide if a picture s a dog or cat, you may use softmax at the end to convert the final weight outputs of those neurons into a probability for each class. Then just pick the one with the highest probabilty. Essentially, it gives a score for each class, it produces a probability of each class and the class with the highest probability is the "answer" you get. (highest value of the softmax function is the best choice or classification).

$$h_\theta(x) = \frac{1}{1+exp(-\theta^T x)}$$

12.2 Introducing Artificial Neural Networks

Based on an understanding of our own brains - ANN's have a biological inspiration. Neurons in your cerebral cortex are connected via axons and dendrites. A neuron "fires" to the neurons it's connected to, when enough of its own input signals are activated ("threshold" is reached). This is very simple at the individual neuron level, but layers of neurons connected in this way can yield learning behavior. Billions of neurons, each with thousands of connections, yields a mind. Emergent behaviour; complex behaviour from a simple model.

Furthermore, if we look deeper into the biology of the brain, we can see that neurons in your cortex seem to be arranged into many stacks, or "columns" that process information in parallel. "Mini-columns" of around 100 neurons are organized into larger "hyper-columns". There are 100 million mini-columns in your cortex. Coincidentally, this is similar to how GPU's work. . .

Key is that we see emergent behaviour; we have very simple building blocks, but very very complex systems.

12.2.1 The First Artificial Neurons

Goes all the way back to 1943. Can create logical Boolean expressions by using this. e.g. An artificial neuron "fires" if more than N input connections are active. Depending on the number of connections from each input neuron, and whether a connection activates or suppresses a neuron, you can construct AND, OR, and NOT logical constructs this way.

12.2.2 The Linear Threshold Unit (LTU)

Build upon the idea using the LTU in 1957 by assigning weights to those inputs. Not just ON or OFF switches; output is given by a step function. Since the inputs now have weights associated with them, we not have a step function to tell when we should fire the final signal. Threshold etc e.g. Sum up the products of the inputs and their weights Output 1 if sum is >= 0 (Can have negative weights coming in that make it less than 0)

12.2.3 The Perceptron

This is a layer of multiple LTU. A perceptron can learn by reinforcing weights that lead to correct behavior during training. This too has a biological basis ("cells that fire together, wire together" speaking to the learning mechanism in our perceptron here.) Reward the strength of connections between neurons (i.e. weight) that have the behaviour that we want.

A Bias Neuron may be needed, a fixed value, to make the maths work out.

12.2.4 Multi-Layer Perceptrons

One more step further. Have a hidden layer in the middle, between inputs and outputs. Addition of "hidden layers". This is a Deep Neural Network. Training them is trickier –but we'll talk about that. Should appreciate there are LOTS of connections, and a loot of opportunity for optimising the weights in between each connection.

12.2.5 A Modern Deep Neural Network

Here we replace the step activation function with something better (alternative activation functions like ReLU). Apply softmaxto the output and training using gradient descent (or some variant therof).

Key here is that the step function has a lot of nasty mathematic properties for slopes and derivatives.

12.3 playground.tensorflow.org

People behind tensorflow at google have created this site to play around with neural networks without code and helps you get an intuitive way to see how deep leaning works.

12.4 Deep Learning

Constructing, training, and tuning multi-layer perceptrons. How do you train it? you use "backpropogation".

12.4.1 Backpropagation

How do you train a MLP's weights? How does it learn? You use backpropagation ... or more specifically; gradient descent using reverse-mode autodiff. Here, for each training step:

- Compute the output error for the weights we have currently in place
- Compute how much each neuron in the previous hidden layer contributed - where backpropogation comes in
- Back-propagate that error in a reverse pass
- Tweak weights to reduce the error using gradient descent
- arrive at a (hopefully) better value on next pass/epoch

12.4.2 Activation Functions (aka Rectifier)

Step functions don't work with gradient descent –there is no gradient to calculate! Mathematically, they have no useful derivative. Alternatives are therefore used:

- Logistic function
- Hyperbolic tangent function
- Exponential linear unit (ELU)
- ReLU function (Rectified Linear Unit)

These determine the output of a neuron given the sum of its inputs. Activation function takes the sum and turns it into an output signal. ReLUis common. It is simple and fast to compute and works well. Converges quickly. Also: "Leaky ReLU", "Noisy ReLU" converge better but more slowly. ELU can sometimes lead to faster learning though.

12.4.3 Optimization Functions

There are faster (as in faster learning) optimizers than gradient descent. Examples include:

1. **Momentum Optimization**: Introduces a momentum term to the descent, so it slows down as things start to flatten and speeds up as the slope is steep
2. **Nesterov Accelerated Gradient**: A small tweak on momentum optimization – computes momentum based on the gradient slightly ahead of you, not where you are
3. **RMSProp**: Adaptive learning rate to help point toward the minimum
4. **Adam**: Standing for "Adaptive moment estimation" – uses momentum & RMSProp combined. It is a popular choice today, easy to use.

12.4.4 Avoiding Overfitting

With thousands of weights to tune, overfitting is a problem. Ways to combat this include: **Early stopping** (when performance starts dropping) will stop reinforcing those spikes, where you're not fitting to the pattern but to the training data given. **Regularization terms** added to cost function during training. **Dropout** –ignore say 50% of all neurons randomly at each training step. This works surprisingly well! Forces your model to spread out its learning. Forces all neurons to be used effectively.

12.4.5 Tuning Your Topology

Trial & error is one way to improve the performance of your deep leaning network. Evaluate a network with a different number of neurons in the hidden layers, and also a network with a different number of hidden layers. Try reducing the size of each layer as you progress to form a funnel. More layers usually yield faster learning than more neurons and less layers, or just use more layers and neurons than you need, and don't care because you use early stopping. There also, with the modern computing environment, there wont be much expense with having more neurons than you need. Or use "model zoos" which are libraries of optimal neural network topologies for specific problems.

12.5 Tensorflow

It's not specifically for neural networks – it's more generally an architecture for executing a graph of numerical operations. Just happens to be useful for developing Deep Learning Neural Networks. Tensorflow can optimize the processing of a graph of numerical operations, and distribute its processing across a network (e.g. GPU cores, machines on a network). Sounds a lot like Apache Spark, eh? Key differences include:

- Can handle massive scale –it was made by Google
- Runs on about anything, including your phone. Not limited, therefore you can push the processing down to the end users device. E.G. self-driving car, don't want it to crash just because it lost connection to the network. Can train offline though, and then run the NN on the car.
- Highly efficient C++ code with easy to use Python API's - easy to use python interface, so is easy to use as a developer.
- Can work on GPU's as well, not just CPU's
- Free and made by Google

Install with pip install tensorflow or pip install tensorflow-gpu in python environment. A tensor is just a fancy name for an array or matrix of values. To use Tensorflow, you:

- Construct a graph to compute your tensors
- Initialize your variables
- Execute that graph –nothing actually happens until then, like Apache.

As an example, here is the worlds simplest tensor flow application:

[ ]:
```
import tensorflow as tf
a = tf.Variable(1, name="a") #variable object in tensorflow, name will only appear␣
 ↪in visualisation tools
b = tf.Variable(2, name="b")
f = a + b # Not actually added together, f is a graph. Connection beteen a and b␣
 ↪tensors (dependedncy on f graph we created).
init= tf.global_variables_initializer()
with tf.Session() as s:
init.run()
print( f.eval() )
```

12.5.1 Creating a Neural Network with Tensorflow

Mathematical insights include: All those interconnected arrows multiplying weights can be thought of as a big **matrix multiplication**. In addition, the bias term can just be added onto the result of that matrix multiplication.

In Tensorflow, we can define a layer of a neural network as:

output = tf.matmul(previous_layer, layer_weights) + layer_biases

By using Tensorflow directly we're kinda doing this the "hard way." There are libraries built on top which can do this for you so you don't need to use it for matrix multiplication, or even define deep learning networks.

There is more to creating a deep learning neural network than computing the weights of each connection between neurons.

1. **Load up** training and testing data.
2. **Construct a graph** describing neural network. Here, use **placeholders** for the input data and target labels. This way we can use the same graph for training and testing! Use **variables** for the learned weights for each connection and learned biases for each neuron. Variables are preserved across runs within a Tensorflow session. As we iterate through steps, we have memory in-between runs.
3. Associate an optimizer (i.e. gradient descent) to the network.
4. Run the optimizer with your training data.
5. Evaluate your trained network with your testing data.

One "Gotcha!" here is to **make sure your features are normalized**. Neural networks usually work best if your input data is normalized. That is, 0 mean and unit variance. The real goal is that every input feature is comparable in terms of magnitude. Scikit_learn's 'StandardScaler' can do this for you. Many data sets are normalized to begin with –such as the one we're about to use.

To activate this environment, use conda activate tf To deactivate an active environment, use conda deactivate

Also, to update TensorFlow .py scripts, visit this website:

https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/upgrade.ipynbscrollTo=UGO7xSyL89wX