Taking Raw Data Towards Analysis

Vince Croft

February 24, 2015

Abstract

Code fragments designed to explain the examples given in the inverted CERN School of Computing in february 2015. Examples given Python and Hadoop.

1 Introduction

1.1 Introduction to Python

Python is a dynamic object-oriented programming language. Its legibility and ease of use has lead it to become the number one programming language used and required by scientists engineers and employers the world over `http://blog.codeeval.com/codeevalblog/2015#py=`
This lecture course was not designed as a python tutorial but there are many fantastic resources out there for learning python for data analysis. I recommend the following resources:

- O'Reilly : Think Stats by Allen B Bowney.
  Free from `http://greenteapress.com/thinkstats/`
- kagle data science wiki
  `https://www.kaggle.com/wiki/GettingStartedWithPythonForDataScience`
- code accademy for basics
  `http://www.codecademy.com/en/tracks/python`
- Python Data Visualization Cookbook by Igor Milovanovic
  `http://shop.oreilly.com/product/9781782163367.do`

1.2 Manipulating Data

Exploratory Data Analysis is the way we display data to learn more about what it can tell us.

1.3 Extracting Linkedin Data

Linked in is a social network that focuses on career and employment. The data people choose to share here is designed to reflect their skills, experiences and personality in the most professional and proficient manner possible. As such there exist many ways to access and view the data that this repository of information contains.

The first and easiest way to access the contact information from your own linkedin profile is simply by clicking the connections tab on the linkedin webpage and then under advanced settings there is the option to export connections as a CSV file (designed to be read by address books in osx or outlook etc)

The method implemented here gives a little more data and after authentification downloads the data in a nice easy to use dictionaly. Following instructions found at `https://github.com/ozgur/python-linkedin` the python-linkedin library allows interface to the LinkedIn application program interface (API)

A request to access the API can be submitted at `https://developer.linkedin.com`

```
from linkedin import linkedin # pip install python-linkedin
# Define CONSUMER_KEY, CONSUMER_SECRET,
# USER_TOKEN, and USER_SECRET from the credentials
# provided in your LinkedIn application
CONSUMER_KEY = 'somekeyhere'
CONSUMER_SECRET = 'somesecrethere'
USER_TOKEN = 'sometokenhere'
USER_SECRET = 'anothersecrethere'

# Instantiate the developer authentication class
auth = linkedin.LinkedInDeveloperAuthentication(CONSUMER_KEY,
    CONSUMER_SECRET, USER_TOKEN, USER_SECRET, RETURN_URL,
    permissions=linkedin.PERMISSIONS.enums.values())
app = linkedin.LinkedInApplication(auth)

#the profile used to get the app can be accessed as such
vince=app.get_profile()
print vince

connections = app.get_connections()
```

This code should be sufficient to retrieve the contacts of the account supplied with some simple information such as the names, location and industry that each person uses.

```
from prettytable import PrettyTable # pip install prettytable
pt = PrettyTable(field_names=['Name', 'Location'])
pt.align = 'l'
[ pt.add_row((c['firstName'] + ' ' + c['lastName'],
```

```
    c['location']['name'])) for c in connections['values'] if c.has_k
ey('location')]
print pt
```

Should access and print to screen all of the connections names plus their region. As in the first lecture our goal should be to move these from events to vectors and from these look at the frequency.

First lets look at the industry that people work in.

```
pt = PrettyTable(field_names=['Name', 'Industry'])
[ pt.add_row((c['firstName'] + ' ' + c['lastName'], c['industry'])) for
    c in connections['values'] if c.has_key('industry')]
print pt #shows all names and industries.

#to sort industies by frequency let's add them to a list
fields = [c['industry'] for c in connections['values'] if
    c.has_key('industry')]

#use some libraries for calculating frequency

from collections import Counter
from operator import itemgetter

pt = PrettyTable(field_names=['Field', 'Freq'])
c = Counter(fields)
[pt.add_row([field, freq]) for (field, freq) in sorted(c.items(),
    key=itemgetter(1), reverse=True) if freq>1]
print pt #shows field sorted in reverse if there's more than one entry
```

I'm not sure what your linkedin data will show you but for me almost all of my contacts work in either *Research* or *Higher Education*

Now we know how to access the location and field that our contacts work in, what about the information that they give us about what it is they do? The API also provides us with a headline. This is expressed as a string of words which we can then extract to find keywords that link contacts.

```
headline = [c['headline'] for c in connections['values'] if
    c.has_key('headline') and c\
.has_key('industry') and c['industry'] in ('Research','Higher
    Education')]

keywords=[]
for line in headline:
    for word in line.split():
        keywords.append(word)

#this produces a high likelihood of link words which we can exclude
```

```
banned=['at','University','of','in','and','&','en']

pt = PrettyTable(field_names=['Keyword', 'Freq'])
c = Counter(keywords)
[pt.add_row([keyword, freq]) for (keyword, freq) in sorted(c.items(),
    key=itemgetter(1)\
, reverse=True) if freq>7 and keyword not in banned]
print pt
```

Interestingly if we only look at candidates from certain countries the frequency of keywords changes. In my data this is due to many of my contacts from Denmark going into fields such as Geoengineering. My contacts from my undergraduate being either professors or still being students and contacts from the netherlands where I'm studying for my PhD having a high frequency of the word PhD in the title.

This method of looking at frequency tables and highlighting certain regions may give hints of where my contacts might be clustered in terms of location and research interests. But it is far from conclusive and misses out many points.

For those interested one last tool for plotting the information obtained in this way is the google Visualization API Reference. This lives in the package called gvis `https://code.google.com/p/google-visualization-python/`

```
#build json map
import gviz_api
from gvis import get_page_template # this is a simple html file to
    display the json data table
countries = [c['location']['country']['code'] for c in
    connections['values'] if c.has_key('location')]
c = Counter(countries)

data=[]
for country,freq in c.items():
    data.append({"Country":country,"freq":(float(freq),str(freq))})

description={'Country':("string","Country"),"freq":("number","freq"),}
data_table=gviz_api.DataTable(description)
data_table.LoadData(data)
json=data_table.ToJSon(columns_order=("Country","freq"),order_by="Country",)
with open('geoLinked.html','w') as out:
    out.write(get_page_template() % (json,))
```

2 Visualising MVA

2.1 Heat Maps

Humans have always had trouble expressing themselves in 3 dimensions. If we express ourselves with pictures and pictures are just 2 dimensional how then to express something that has 3 dimensions. From a one dimensional array we made a histogram. This histogram was 2 dimensional since it had the demension of the variable and the probability. When moving to 2 dimensions we run in to the problem of how to express this 3rd Dimension.
When looking at scatter plots we can often intuatively understand the inherent density of areas of the plot. A useful way to make this more obvious is to use the same technique as before. If we bin our axis we are left with a 2 dimensional plane where the values expressed are no longer points but probability densities. These values however must be expressed somehow.
A common method for displaying such dimensional histograms are heat maps. Where each point on the map is assigned a color depending on how many experimental points fall within its boarders.
In python this is very easy to do using numpy for our random numbers and matplotlib for plotting

```
import numpy as np
from matplotlib import pylab as plt

x=np.random.randn(200)
y=np.random.randn(200)

plt.hist2d(x,y,bins=6)
plt.show()
```

2.2 Mean Vector

From the basics of exploring data we stumbled through some pretty big concepts like the mean and the variance along with some common distributions. Knowledge of distributions such as these can be very useful for optimising the preparation of our data. E.g. we might know that one variable we are interested in follows a gaussian distribution. In this case we can accurately and quickly optimise an algorithm that reduces all of the possible measurements of this variable to 3 numbers. The mean, the variance and the number of measurements. In this way the data can be summarised.
The mean vector for a multivariate data set does this also. Reduces the information from all of the variables in the distribution to a simple vector of the mean/expectation value of each variable.

Comparing the means of different collections of information can be very beneficial as we see later on.

I'll be using numpy to represent the vectors here and hope to extend this tutorial to include also the mathematical description as well as the computational.

2.3 Covariance Matrix

In the same way that a 3D object can be sumarised as 3 numbers, the average x value the average y value and the average z value, the variance of the joint distribution can also be represented this way. As seen before the variance in x, y and z won't contain the information contained in the relationship between the vectors.

```
mean_1 = np.array([-1,-1]) #first mean vector
cov_1 = np.array([[1,0],[0,1]]) #covariance matrix
signal_1 = np.random.multivariate_normal(mean_1, cov_1, 20).T

mean_2 = np.array([1,1]) #second mean vector
cov_2 = np.array([[1,0],[0,1]]) #covariance matrix
signal_2 = np.random.multivariate_normal(mean_2, cov_2, 20).T

from matplotlib import pyplot as plt

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
ax.plot(signal_1[0,:], signal_1[1,:],'o', markersize=8, color='blue',
    label='signal 1')
ax.plot(signal_2[0,:], signal_2[1,:],'^', markersize=8, color='red',
    label='signal 2')

combined=np.concatenate((signal_1, signal_2, signal_3),axis=1)
fig = plt.figure(figsize=(8,8))
ax2=fig.add_subplot(111)
ax2.hist(combined[0,:],combined[1,:],bins=6)

plt.show()

cov_matrix=np.cov([combined=np.concatenate(combined[0,:],combined[1,:]))
print cov_matrix
```

2.4 Distance Matrix

The distance matrix is a relatively simple concept to understand however its usage takes a little more meditation before it can be fully appreciated. A distance matrix is a two dimensional structure expressing the distance between a series of points. e.g. if we have 5 measurements a,b,c,d,e then the matrix is a 5x5 grid where the diagonals are 0 and the first row is the distance between point a and

all of the other measurements.
This easily allows us to convert a multi dimensional problem into something we can easily visualise with a heat map. The distances can be calculated with vectors of measurements or a mean vector or a vetor of cluster means.

```
import math
#print distance matrix
Dist_Matrix=np.abs(((signal_1[0,:])**2+(signal_1[0,:])**2)**0.5-(((signal_1[0,:,np.newaxis])**2+(signa
print ''DistMatrix''
print Dist_Matrix
```

3 Dimensional Reduction

Despite my excitement at being able to extract additional information from a distribution, once we ascend to the plain of mutlivariate analysis several applications to reducing the number of information vectors become apparent.

As demonstrated by the examples in 2 dimensions. A joint distribution contains information in the covariance matrix that is not contained in the marginal distributions. The maximum information cannot be obtained from a set of 2 dependent vectors so often simple analysis require only that the dependent variables

One huge reason behind dimensionality reduction appears when the scale of the data worked with becomes to get large. Most analysis use some degree of measuring one thing in comparison with another but when we want to assess the similarity of a set of things with all other things we end up for a set of n members a calculation on the order of n^2 similarity computations. This quickly scales and becomes un reasonable.

3.1 Feature Selection

Dimensional Reduction can be optimised for one of two goals, in feature selection a subset of the orriginal variables is selected that either gives a gain of information (for the data given) or to get the most accurate subset (this requires knowing the desired result)

The example of dividing one dimension by another reduces 2 dimensions to 1, but it may be more optimal to divide it by 10 and then divide it by the second. There are many different combinatorial optimisations that aim to give a dimensional reuduction but they won't be discussed further here.

3.2 Feature Extraction

Feature Extraction is a method for transforming dimensions instead of combining them. This method is more complicated then simply combining the values. Usually this is done by transforming the co-ordinate system from a space of higher dimensionality to a lower one. This transformation is usually linear but this is not always the case. In this way the features being extracted are the variables with the highest covariance.

3.2.1 Principle Component Analysis

The following code relies heavily on the instructions given here. `http://sebastianraschka.com/Articles/2014_pca_step_by_step.html#sample_data` This page also includes instructions for using the built in locations.

```
from matplotlib import pyplot as plt
import numpy as np

np.random.seed(004176636) # random seed for consistency

mean_1 = np.array([0,-1]) #first mean vector
cov_1 = np.array([[0.25,0],[0,0.25]]) #covariance matrix
signal_1 = np.random.multivariate_normal(mean_1, cov_1, 20).T

mean_2 = np.array([1,1]) #second mean vector
cov_2 = np.array([[0.25,0],[0,0.25]]) #covariance matrix
signal_2 = np.random.multivariate_normal(mean_2, cov_2, 20).T

mean_3 = np.array([-1,0.5]) #second mean vector
cov_3 = np.array([[0.25,0],[0,0.25]]) #covariance matrix
signal_3 = np.random.multivariate_normal(mean_3, cov_3, 20).T

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
ax.plot(signal_1[0,:], signal_1[1,:],'o', markersize=8, color='blue',
    label='signal 1')
ax.plot(signal_2[0,:], signal_2[1,:],'^', markersize=8, color='red',
    label='signal 2')
ax.plot(signal_3[0,:], signal_3[1,:],'v', markersize=8, color='yellow',
    label='signal 3')
```

first we set up our clusters, I just took some random gaussians and gave them some nice labels. We however need to combine these into one complete data set. For this we use the numpy concatenate and collect the summary data.

```
combined=np.concatenate((signal_1, signal_2, signal_3),axis=1)
assert combined.shape==(2,60), " the matrix isn't 2x60"
#make mean vector
mean_x = np.mean(combined[0,:])
```

```
mean_y = np.mean(combined[1,:])
mean_vec=np.array([[mean_x],[mean_y]])

print 'Mean Vector '
print mean_vec

#next comes the covariance matrix
cov_mat = np.cov([combined[0,:],combined[1,:]])
print "Covariance Matrix "
print cov_mat
```

ok so if we take a geometric interpretation of the x and y axis as vectors then the covariance can be understood as related to the cosine of the angle between them. This allows us to define a series of eigen pairs from the covariance matrix such that we can take those that we want.

```
eig_val, eig_vec = np.linalg.eig(cov_mat)

print(40 * '-')
for i in range(len(eig_val)):
    eigvec = eig_vec[:,i].reshape(1,2).T

    print('Eigenvector {}: \n{}'.format(i+1, eigvec))
    print('Eigenvalue {}: {}'.format(i+1, eig_val[i]))
    print(40 * '-')

#test the vectors
for i in range(len(eig_val)):
    eigv = eig_vec[:,i].reshape(1,2).T
    np.testing.assert_array_almost_equal(cov_mat.dot(eigv),\
           eig_val[i] * eigv, decimal=6,\
           err_msg='', verbose=True)
```

The vectors represent the vectors of a smaller feature subspace since we want to reduce the number of vectors we want those with the biggest eigen values since (generally speaking these have the most information)

```
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_val[i]), eig_vec[:,i]) for i in
    range(len(eig_val))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
eig_pairs.sort()
eig_pairs.reverse()

# Visually confirm that the list is correctly sorted by decreasing
    eigenvalues
for i in eig_pairs:
   print(i[0])
```

In this example we are moving form 2D to 1D if there are more you want. e.g. if you would like to transform a 3D space into 2D then you want to take the top 2 eigen pairs in the list and then add them to the matrix.

```
matrix_w = eig_pairs[0][1].reshape(2,1)
print('Matrix W:\n', matrix_w)
```

This is the transformation that we want so now all that's left to do is transform the samples onto this new subspace

```
transformed = matrix_w.T.dot(combined)
assert transformed.shape == (1,60), "The matrix is not 1x60 dimensional."
zeros=np.zeros(60)
fig2 = plt.figure(figsize=(8,8))
ax2 = fig2.add_subplot(111)
ax2.hist(transformed[0,:],bins=10)
plt.title('Transformed samples with class labels')
plt.show()
```

And there we have it.

4 Clustering

Still in the frame of mind of data reduction what we're aiming to do is take all of the things that are similar and merge them into a smaller object that constains as much of the initial information as possible. In this context similar can mean similar positioning on a graph or that have similar covariances.
Clustering is the process by which we group similar objects into larger objects. This way we have one larger object which constitutes less data than two smaller ones.

4.1 Defining Distance

In order to define objects as being close to each other we need to define a standard for similarity. In terms of euclidean distance this is more simple than others. The smaller the distance between two points the more similar they are. There are different metrics for defining distance so a simple data preparation would be to define which clusters we wish to look at.
The simplest metric I believe is the Euclidean distance. In this metric the distance is the square root of the sum of the differences in x and y. Other metrics include the mahalanobis distance known also as the standardised statistical distance or the Manhattan distance which accounts for large steps in x and y.
Data however comes in all shapes and sizes. The variables that we wish to compare might not be continuous numbers. The Levenshtein distance measures the quantity of insertions deletions and replacements it would take to convert

one string into another.

4.2 Merging and Stopping

We must also define what happens when we determine that two points come from the same cluster. Should the cluster simply be the sum of the two points at the center of mass of the pair? or should there be some other heuristic? When should we stop merging clusters? at a defined quantity of clusters or when the clusters get to a certain size?

4.3 Stability

The number of clusters that an algorithm searches for is often very important and is often taken from the number that gives the most stable clusters. The stability is the expected distance between two clusters on different data sets such that if we choose 2 clusters to represent the data produced by 3 gaussians then the results should be further appart than the same thing with 4. The clustering stability is a very important point when making robust clustering algorithms.

4.4 K-means Clustering with Lloyds Algorithm

K-means clustering is the cornerstone of distance based clustering algorithms. This requires that you know a priori how many clusters you would like and also requires that you provide an initial guess for this value. These initial conditions can be interpreted from the nature of the problem e.g you might want to split your events into signal and background and you know from looking at marginal distributions that these come from only two processes.

Once again we will generate a series of 3 2dimensional joint distributions. I'm very aware that I switch between numpy arrays and python lists and this is very bad coding style. Any tips please let me know.

The classes used were adapted (butchered) from the example found here `https://datasciencelab.wordpress.com/2013/12/12/clustering-with-k-means-in-python/` which I found as a fantastic resource but I found the nature of k-means covered in depth in numerical rescipies if people would rather an standard resource.

```
np.random.seed(004176636) # random seed for consistency
mean_1 = np.array([0,-1]) #first mean vector
cov_1 = np.array([[0.25,0],[0,0.25]]) #covariance matrix
signal_1 = np.random.multivariate_normal(mean_1, cov_1, 20).T

mean_2 = np.array([1,1]) #second mean vector
cov_2 = np.array([[0.25,0],[0,0.25]]) #covariance matrix
signal_2 = np.random.multivariate_normal(mean_2, cov_2, 20).T
```

```
mean_3 = np.array([-1,0.5]) #second mean vector
cov_3 = np.array([[0.25,0],[0,0.25]]) #covariance matrix
signal_3 = np.random.multivariate_normal(mean_3, cov_3, 20).T

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
ax.plot(signal_1[0,:], signal_1[1,:],'o', markersize=8, color='blue',
    label='signal 1')
ax.plot(signal_2[0,:], signal_2[1,:],'^', markersize=8, color='red',
    label='signal 2')
ax.plot(signal_3[0,:], signal_3[1,:],'v', markersize=8, color='yellow',
    label='signal 3')

combined=np.concatenate((signal_1, signal_2, signal_3),axis=1)
assert combined.shape==(2,60), " the matrix isn't 2x60"

points=[]
xpoints=[]
ypoints=[]
for x in range(len(combined[0])):
    xpoints.append(combined[0][x])
    ypoints.append(combined[1][x])
for point in range(len(xpoints)):
    points.append([xpoints[point],ypoints[point]])

fig2 = plt.figure(figsize=(8,8))
ax2 = fig2.add_subplot(111)
ax2.plot(xpoints,ypoints,'o', markersize=8, color='blue',
    label='combined')
```

Remembering now that we have points we must supply a number of clusters. We define a function find centers to retrieve the mean vector and clusters that make up our reduced dataset. This algorithm first takes a series of 3 random points from the distribution to use as the mean vector, and then clusters all the points according to which of these mu values they are closest to. The means of these clusters are then computed. This algorithm repeats until the mu value doesn't change.

Here we define 4 functions. Find centers controlls the process by initialising mu and containing the loop. The function cluster points calculates the distance for every point to every cluster center. One we have the events that are closest to each center we find the new center of this group. The has converged class checks if the new center and old center are different if not then the process ends.

```
def cluster_points(X, mu):
    clusters = {}
    for point in X:
        closest=len(mu)+1
        dist=40
```

```
        mus=np.array(mu)
        for k in enumerate(mu):
            if np.linalg.norm(point-mus[k[0]])<dist:
                dist=np.linalg.norm(point-mus[k[0]])
                closest=k[0]
        if closest!=len(mu)+1:
            try :
                clusters[closest].append(point)
            except KeyError:
                clusters[closest]=[point]
    return clusters

def reevaluate_centers(mu, clusters):
    newmu = []
    keys = sorted(clusters.keys())
    for k in keys:
        newmu.append(np.mean(clusters[k], axis = 0))
    return newmu

def has_converged(mu, oldmu):
    return (set([tuple(a) for a in mu]) == set([tuple(a) for a in
         oldmu]))

def find_centers(X, K):
    # Initialize to K random centers
    oldmu = random.sample(X, K)
    mu = random.sample(X, K)
    print "first mu is ", mu
    iteration=0
    while not has_converged(mu, oldmu):
        oldmu = mu
        iteration+=1
        # count iterations to convergence
        print "this is the ",iteration,"th iteration"
        # Assign all points in X to clusters
        clusters = cluster_points(X, mu)
        # Reevaluate centers
        mu = reevaluate_centers(oldmu, clusters)
    return(mu, clusters)
```

All that's left now is to call this function for our data and the option to plot the clusters that are returned.

```
mu, clusters=find_centers(points,3)

cluster1=np.array(clusters[0])
cluster2=np.array(clusters[1])
cluster3=np.array(clusters[2])
```

```
fig3 = plt.figure(figsize=(8,8))
ax3 = fig3.add_subplot(111)
ax3.plot(cluster1[:,0],cluster1[:,1],'o', markersize=8, color='blue',
    label='closest 1')
ax3.plot(cluster2[:,0],cluster2[:,1],'^', markersize=8, color='red',
    label='closest 2')
ax3.plot(cluster3[:,0],cluster3[:,1],'v', markersize=8, color='yellow',
    label='closest 3')
plt.show()
```

5 summary

And that's all folks. I wish you many hours of fun with PCA and SVD and K-means lovelyness. I hope to eventually extend this file. Perhaps to expand the PCA section to include some of the mathematics involved. Any tips or comments. let me know!

I'm easy to find at vincent.croft at cern.ch