ML Tutorial

June 12, 2018

```
In [92]: import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         import sklearn
         import seaborn as sns

         %matplotlib inline
```

1 Primer on Stats and Numpy

1.1 1. Random Numbers and Vectors

Using arange to create a range:

```
In [19]: arr = np.arange(1,10)
         print(arr)
```

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

Then turn it into a matrix:

```
In [20]: mat = arr.reshape(3,3)
         print(mat)
```

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

Create an array of uniformly distributed random variables:

```
In [21]: randlist = np.random.random(100)
         print(randlist)
```

```
[0.3266449  0.5270581  0.8859421  0.35726976 0.90853515 0.62336012
 0.01582124 0.92943723 0.69089692 0.99732285 0.17234051 0.13713575
 0.93259546 0.69681816 0.06600017 0.75546305 0.75387619 0.92302454
 0.71152476 0.12427096 0.01988013 0.02621099 0.02830649 0.24621107
 0.86002795 0.53883106 0.55282198 0.84203089 0.12417332 0.27918368
 0.58575927 0.96959575 0.56103022 0.01864729 0.80063267 0.23297427
 0.8071052  0.38786064 0.86354185 0.74712164 0.55624023 0.13645523
 0.05991769 0.12134346 0.04455188 0.10749413 0.22570934 0.71298898
 0.55971698 0.01255598 0.07197428 0.96727633 0.56810046 0.20329323
 0.25232574 0.74382585 0.19542948 0.58135893 0.97001999 0.8468288
 0.23984776 0.49376971 0.61995572 0.8289809  0.15679139 0.0185762
 0.07002214 0.48634511 0.60632946 0.56885144 0.31736241 0.98861615
 0.57974522 0.38014117 0.55094822 0.74533443 0.66923289 0.26491956
 0.06633483 0.3700842  0.62971751 0.21017401 0.75275555 0.06653648
 0.2603151  0.80475456 0.19343428 0.63946088 0.52467031 0.92480797
 0.26329677 0.06596109 0.73506596 0.77217803 0.90781585 0.93197207
 0.01395157 0.23436209 0.61677836 0.94901632]
```

You can then multiply the list by a scale parameter and add a shift parameter to give the following random vector:

```
In [22]: randlist = randlist * 10 + 10
         print(randlist)
```

```
[13.26644902 15.27058102 18.85942099 13.5726976  19.08535151 16.23360116
 10.15821243 19.29437234 16.90896918 19.9732285  11.72340508 11.3713575
 19.32595463 16.96818161 10.66000173 17.55463053 17.53876188 19.23024536
 17.11524759 11.24270962 10.19880134 10.26210987 10.28306488 12.46211068
 18.60027949 15.38831064 15.52821979 18.42030892 11.24173315 12.79183679
 15.85759271 19.69595748 15.61030219 10.18647289 18.00632673 12.32974274
 18.07105196 13.87860644 18.63541855 17.47121643 15.56240234 11.36455226
 10.5991769  11.21343456 10.44551879 11.07494129 12.25709339 17.1298898
 15.59716982 10.1255598  10.7197428  19.6727633  15.68100462 12.03293235
 12.52325745 17.43825854 11.95429481 15.81358927 19.70019989 18.46828801
 12.39847759 14.93769714 16.19955718 18.289809   11.56791395 10.18576202
 10.70022144 14.86345111 16.06329462 15.68851437 13.17362409 19.88616154
 15.79745219 13.80141173 15.50948219 17.45334431 16.69232893 12.64919558
 10.66334834 13.70084198 16.29717507 12.1017401  17.52755554 10.66536481
 12.60315099 18.04754564 11.93434283 16.39460881 15.24670309 19.2480797
 12.6329677  10.65961091 17.35065963 17.7217803  19.07815853 19.31972069
 10.13951573 12.34362086 16.16778357 19.49016321]
```

Which is an array of random (`float`) numbers from 10 to 20 distributed uniformly in between.

When you are dealing with a random number generator (rng), it is often helpful to set the seed so that you can reproduce your result.

```
In [84]: np.random.seed(1)
         randlist1 =  np.random.random(100) * 10 + 10
         np.random.seed(1)
         randlist2 =  np.random.random(100) * 10 + 10
         print(randlist1-randlist2)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]
```

1.2 2. Matrices

To create a matrix, use `np.matrix`:

```
In [29]: mat1 = np.matrix([[1,2,3],[4,5,6],[6,7,8]])
         print(mat1)
```

```
[[1 2 3]
 [4 5 6]
 [6 7 8]]
```

All matrix objects have `.I` **attribute**, which calculates the inverse (if it exists):

```
In [33]: print(mat1.I)
```

```
[[ 1.7156570e+15 -4.2891425e+15  2.5734855e+15]
 [-3.4313140e+15  8.5782850e+15 -5.1469710e+15]
 [ 1.7156570e+15 -4.2891425e+15  2.5734855e+15]]
```

You can also view the `.T` **attribute**:

```
In [34]: print(mat1.T)
```

```
[[1 4 6]
 [2 5 7]
 [3 6 8]]
```

The `*` operator carries out matrix multiplication instead of multiplying it element-wise:

```
In [37]: mat2 = np.matrix([[9,10,11],[12,13,14],[15,16,17]])
         mat1*mat2
```

```
Out[37]: matrix([[ 78,  84,  90],
                 [186, 201, 216],
                 [258, 279, 300]])
```

NumPy has built in functions to create commonly used matrices:

```
In [106]: #Identity Matrix

          np.eye(10)
```

```
Out[106]: array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

```
In [107]: #Matrix of Ones

          np.ones((3,3))
```

```
Out[107]: array([[1., 1., 1.],
                 [1., 1., 1.],
                 [1., 1., 1.]])
```

```
In [108]: np.ones((2,3))
```

```
Out[108]: array([[1., 1., 1.],
                 [1., 1., 1.]])
```

```
In [109]: np.ones((3,2))
```

```
Out[109]: array([[1., 1.],
                 [1., 1.],
                 [1., 1.]])
```

1.3 3. Summary Statistics

First, let's start off with a seed for our rng:

```
In [62]: rng = np.random.RandomState(7)
```

Then let's create a vector of 10000 observations from a normal distribution with mean = 10 and standard deviation = 5.

```
In [63]: P = rng.normal(10,5,10000)
```

Let's view the first 10 elements of P:

```
In [64]: P[:10]
```

```
Out[64]: array([18.45262852,  7.67031315, 10.16410082, 12.03758141,  6.05538486,
                10.01032786,  9.99554807,  1.22637847, 15.08829003, 13.00249258])
```

Let's get the mean of this population. Recall the mean is simply the average:

$$\frac{\sum_{i=0}^{N-1} X_i}{N}$$

```
In [110]: mean = np.sum(P)/len(P)
          ## Note the different ways of summing a vector in numpy:
          ## - np.add.reduce(sample)
          ## - np.sum(sample)

          ## Can also determine the mean from the .mean() method


          print(mean)
```

```
9.95463191307012
```

Now, let's get the median and compare the results:

```
In [112]: np.median(P)


          ## To do this manually, just sort the vector and pick the middle element.
          ## If the array length is even, take the average of the middle two elements,
          ## otherwise just take the middle element:

          # sorted_sample = np.sort(sample)
          # if len(sorted_sample) % 2==0:
          #     print((sorted_sample[len(sorted_sample)//2-1]
          #    + sorted_sample[len(sorted_sample)//2])/2)
          # else:
          #     print((sorted_sample[len//2])
```

```
Out[112]: 9.964381471455374
```

We now look at the variance and standard deviation, which represents the spread of the data from the mean:

The calculation of the variance, is given by:

$$Var(Population) = \frac{1}{N}\sum_{i=0}^{N-1}(X_i - \mu)^2$$

Where N is the length of the population vector and μ is the mean of the population.

```
In [94]: pop_var = np.var(P)
         print(pop_var)


         # We can also get the variance manually by doing:
         # pop_var = np.sum((P-P.mean())**2)/N
         # This is just doing a sum of the element-wise squared
         #errors divided by the length of the vector.
```

```
24.614376450825333
```

The standard deviation is defined as the square-root of the variance:

$$\sigma(Population) = \sqrt{Var(X)}$$

```
In [95]: pop_sd = np.sqrt(pop_var)
         print(pop_sd)
```

```
4.961287781496386
```

Thus if we check the standard deviation of the population, the answer looks very close to 5! This is exactly the true standard deviation of the population.

We talked about the population mean and variance. Now, let's talk about the mean and variance of the sample.

Let's sample 100 observations from this population vector, P:

```
In [113]: sample = np.random.choice(P,100)
          print(sample[:10])
```

```
[18.25349846 11.97787858 10.99136953 18.53291841  7.28488333  3.02145545
 13.61480915 18.29212392  1.16888876 12.32285265]
```

In order to get the sample mean, we can just take the average of the sample. Let's use $\hat{\mu}$ to represent the ** sample mean**:

```
In [115]: sample_mean = sample.mean()
          print(sample_mean)

          ## You can also do:
          ## sample_mean = np.sum(sample)/len(sample)
          ## print(sample_mean)
```

```
10.319664970434495
```

Then for the sample variance, we use this formula:

$$Var(Sample) = \frac{1}{n-1}\sum_{i=0}^{n-1}(X_i - \hat{\mu})^2$$

Where n is the size of the sample and $\hat{\mu}$ represents the sample mean.

The intuition behind the difference in the sum being scaled by $\frac{1}{n-1}$ instead of $\frac{1}{n}$ is due to the fact that the act of calculating the mean from the sample data introduces a form of deviation from the true mean. I.e:

$$\mu = \hat{\mu} + \epsilon \qquad where\ \epsilon \geq 0$$

If we look closer at the calculation of the population variance, we then see:

$$\sum_{i=0}^{n-1}(X_i-\mu)^2=\sum_{i=0}^{n-1}(X_i-\hat{\mu}+\hat{\mu}-\mu)^2$$

This gives:

$$\underbrace{\sum_{i=0}^{n-1}(X_i-\hat{\mu}+\hat{\mu}-\mu)^2}_{\text{Population Variance}}=\underbrace{\sum_{i=0}^{n-1}(X_i-\hat{\mu})^2}_{\text{Sample Variance}}+\underbrace{\sum_{i=0}^{n-1}(\hat{\mu}-\mu)^2}_{\text{this is}\geq 0}+2\sum_{i=0}^{n-1}(X_i-\hat{\mu})(\hat{\mu}-\mu)$$

Now, we know that $\hat{\mu}=\frac{1}{n}\sum_{i=0}^{n-1}X_i$ by definition. So $\sum_{i=0}^{n-1}(X_i-\hat{\mu})(\hat{\mu}-\mu)$ is necessarily equal to 0. This means that we have:

$$\underbrace{\sum_{i=0}^{n-1}(X_i-\hat{\mu}+\hat{\mu}-\mu)^2}_{\text{Population Variance}}=\underbrace{\sum_{i=0}^{n-1}(X_i-\hat{\mu})^2}_{\text{Sample Variance}}+\underbrace{\sum_{i=0}^{n-1}(\hat{\mu}-\mu)^2}_{\text{this is}\geq 0}$$

Which implies:

$$Variance(Population)\geq Variance(Sample)$$

Thus, the sample variance that is calculated is less than would be expected, which is why we need to scale it by a smaller factor of $\frac{1}{n-1}$ instead of $\frac{1}{n}$.

Let's now talk about percentiles:

The n-th percentile of a vector determines the value at which the probability of getting an observations below it is n%.

```
In [127]: print(np.percentile(P,.25))
```

```
-4.528680954058658
```

This means that the probability of the viewing an observation of less than -4.53 is 0.25%

1.4 4. Multivariate Distributions

When we deal with more than one variables, sometimes we can see that they can be related in some way or another.

Let's generate two variables that we are going to use to represent height and weight of the population in Singapore. Let's assume that they are unrelated, and height has a mean of 155 cm with a standard deviation of 5 cm and weight has a mean of 65 kg with standard deviation of 5 kg.

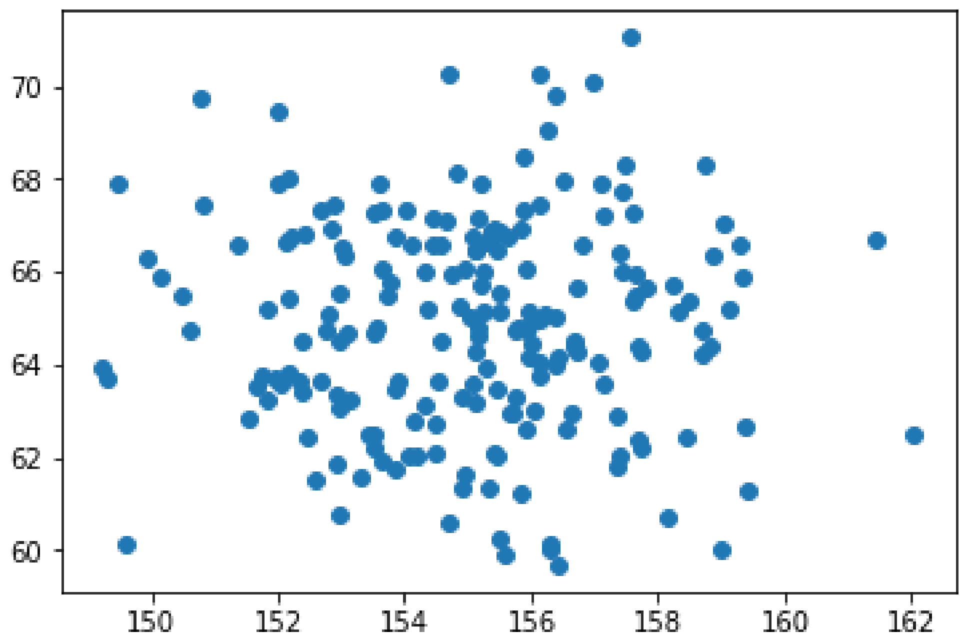
```
In [128]: data = np.random.multivariate_normal([155,65],[[5,0],[0,5]],200)
```

```
In [140]: height = data[:,0]
          weight = data[:,1]
```

Let's plot the height and the weight on a scatter plot:

```
In [141]: plt.scatter(height,weight)
```

```
Out[141]: <matplotlib.collections.PathCollection at 0x10fcf9898>
```



This provides some empirical evidence that height and weight are not correlated.

We can even place a value to measure the related-ness of the two variables by considering a quantity known as the covariance. This is given by:

$$Cov(X, Y) = \frac{1}{n} \sum_{i=0}^{n} (X - \mu_x)(Y - \mu_y)$$

Where μ_x and μ_y are the true means of the populations of the X and Y random variables respectively.

```
In [145]: np.cov(height,weight)
```

```
Out[145]: array([[ 5.41215244, -0.06906557],
                 [-0.06906557,  5.32692431]])
```

The above is a matrix that tells us the covariance between the height and weight variables as well as their corresponding variances. This is known as a covariance matrix, Σ and it has the following form:

$$\Sigma_{ij} = Cov(X_i, X_j)$$

Where X_i and X_j are the different variables in question.

The sign of the covariance determines how the variables are related. A positive covariance indicates that there is positive correlation and vice versa.

Sometimes, we can have a situation where the magnitude of a certain variable far outweighs the other. (Say, you are trying to find out whether a planet moves at a certain angle from the Earth's horizon based on the planet's mass!!!). This means that when we calculate covariance we need to scale the variables in some way such that they can be compared in a sensible way. This leads to the idea of correlation, which is the covariance scaled by the product of the standard deviations:

$$Corr(X,Y) = \frac{1}{n}\frac{\sum_{i=0}^{n}(X-\mu_x)(Y-\mu_y)}{\sqrt{Var_x Var_y}}$$

```
In [146]: np.corrcoef(height,weight)

Out[146]: array([[ 1.        , -0.01286288],
                 [-0.01286288,  1.        ]])
```

The values of the correlation coefficient will always be between 0 and 1. With 0 indicating no correlation and 1 indicating perfect correlation.

In the example we see above, we surmise that height and weight variables are not correlated from the plot; and the correlation coefficient supports this statement.