Introduction to Statistics and Numpy

AcadView

May 16, 2018

---

1 Overview

In this section we shall study about some basic statistics that will be helpful in understanding machine learning with great mathematical details.Also we will discuss the mathematical and scientific computing library of python numpy.

Statistics is a branch of mathematics dealing with the collection, analysis, interpretation, presentation, and organization of data.Knowing some statistics can be very helpful to understand the language used in machine learning. Knowing some statistics will eventually be required when you want to start making strong claims about your results.

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

2 Statistics for Machine Learning

Machine Learning is a buzzword right now and there are many who would be trying their hands at it. Therefore, it would help you a great deal if you are able to understand a little bit about the intricacies that make machine learning tools and libraries so powerful. In this post, we will look at the basic statistical concepts that make up the bedrock of analysis in machine learning.

2.1 Types of data

Understanding different types of data will help you in choosing the different types of techniques that you may use to get insights into the data. The major types of Data are:

- **Numerical:**

This represents some quantifiable thing that you can measure. This can be discrete, which is normally the counts of some event. As for example, how many times a person visits the same doctor when he is sick, or the number of clothes bought by a customer on an eCommerce platform.

- **Categorical:**

These are data that has no inherent numerical meaning, such as man, woman. Or the state of birth of a group of people and so on.

2.2 Mean Median Mode

These are the measures of central tendency of a data set.

2.2.1 Mean

Mean is given by the total of the values of the samples divided by the number of samples. In mathematical terms this would mean:

$$\overline{x} = \frac{1}{n} \sum x_i$$

where *x* is the mean and there are n values of *x*. ^{P} means that you need to take the sum of all the data points.

Below is an example where the mean of a set of data points is calculated.

$$x = \begin{bmatrix} 10 & 20 & 30 & 40 & 50 \end{bmatrix}$$
$$\overline{x} = (10 + 20 + 30 + 40 + 50)/5$$

Thus $\overline{x} = 30$

This is a simple example taken to explain discrete data. The mean can be calculated in case of continuous data as well.

2.2.2 Median

To calculate the median, sort the values and take the middle value. To illustrate let x denote the following numbers:

$$x = \begin{bmatrix} 23 & 40 & 6 & 74 & 38 & 1 & 70 \end{bmatrix}$$

In this case, you would need to sort the list first and then take the midpoint to find the median.

sorted x = $\begin{bmatrix} 1 & 6 & 23 & 38 & 40 & 70 & 74 \end{bmatrix}$

In this case, the midpoint is 38 since the value 38 divides sorted x into 3 values which are smaller than 38 and three values which are higher than 38. Now, in case there are even number of values then the average of the two middle values are taken as the median. The advantage of the median over the mean is that median is less susceptible to outliers. So, in situations where there is a high chance that there may be outliers present in the data set, it is wiser to take the median instead of the mean.

For example, to understand what is the per capita income of a country the median is taken because the rich may be extremely rich which would skew the average and show a different picture than what the average people might experience. In that case, probably, taking the median would give a better insight.

2.2.3 Mode

Mode represents the most common value in a data set. Mode is most useful when you need to understand clustering or number of 'hits'. For example, a retailer may want to understand the mode of sizes purchased so that he can set stocking labels optimally. Say, store A has a mode of 'small' while store B has a mode of 'XXL'.

For example:

X = $\begin{bmatrix} 1 & 40 & 6 & 1 & 23 & 1 & 38 & 74 & 40 & 70 \end{bmatrix}$

The mode is 1 which occurs the most in the dataset.The most common value in the data set.

2.3 Variance and Standard Deviation

Variance and Standard Deviation are essentially a measure of the spread of the data in the data set.

2.3.1 Variance

Variance is the average of the squared differences from the mean. In mathematical terms this would mean:

$$\sigma_2 = {}_{N1}\mathrm{P}(X - \mu)_2$$

where σ^2 is the variance, N is the number of observations (whole population), X is the individual set of observations and μ is the mean. Taking the same example as we took before, if x is given by the following numbers,

$$x = \begin{bmatrix} 23 & 40 & 6 & 74 & 38 & 1 & 70 \end{bmatrix}$$

then the variance can be calculated as shown below.

$$\mathbf{x} = \begin{bmatrix} 23 & 40 & 6 & 74 & 38 & 1 & 70 \end{bmatrix} \textbf{mean=36 difference from the mean}$$
$$= \begin{bmatrix} 13 & 4 & 30 & 38 & 2 & 35 & 34 \end{bmatrix} \textbf{square of the differences}$$
$$= \begin{bmatrix} 169 & 16 & 900 & 1444 & 4 & 1225 & 1156 \end{bmatrix}$$

variance= (169 + 16 + 900 + 1444 + 4 + 1225 + 1156)/7 = 4914/7 = 702

Now one question may come to mind as why the variance is given as σ^2. This is because standard deviation is denoted as and standard deviation is the square root of the variance.

2.3.2 Standard deviation

Standard deviation is an excellent way to identify outliers. Data points that lie more than one standard deviation from the mean can be considered unusual. In many cases, data points that are more than two standard deviations away from the mean are not considered in analysis. We can talk about how extreme a data point is by asking the question "how many sigmas away from the mean is this?"

So in the above case, the standard deviation can be calculated as below.

$\sigma^2 = 702$ $\sigma = 26.49$

2.4 Covariance and Correlation

Lets say we have two different attributes of something. Covariance and Correlation are the tools that we have to measure if the two attributes are related to each other or not.

2.4.1 Covariance

Covariance measures how two variables vary in tandem to their means. The formula to calculate covariance is shown below.

$$cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - E(X))(y_i - E(Y))$$

where x and y are the individual values of X and Y ranging from i = 1,2,..., n where the probability that each value may occur is equal and is equal to (1/n). $E(x)$ and $E(y)$ are the means of X and Y.

2.4.2 Correlation

Correlation also measures how two variables move with respect to each other. A perfect positive correlation means that the correlation coefficient is 1. A perfect negative correlation means that the correlation coefficient is -1. A correlation coefficient of 0 means that the two variables are independent of each other. The formula for finding the correlation coefficient can be found using the following formula.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})\sum_{i=1}^{n}(y_i - \bar{y})}}$$

Both correlation and covariance only measure the linear relationship between data. They will fail to discover any nth order relationship between the two. Correlation is a special case of covariance when the data is standardized. If we are interested in only knowing if there is a relationship then correlation is a better measure as they also measure the extent of the relationship.

2.5 Skewness and Kurtosis

A fundamental task in many statistical analyses is to characterize the location and variability of a data set. A further characterization of the data includes skewness and kurtosis. Skewness is a measure of symmetry, or more precisely, the lack of symmetry. A distribution, or data set, is symmetric if it looks the same to the left and right of the center point.

Kurtosis is a measure of whether the data are heavy-tailed or light-tailed relative to a normal distribution. That is, data sets with high kurtosis tend to have heavy tails, or outliers. Data sets with low kurtosis tend to have light tails, or lack of outliers. A uniform distribution would be the extreme case.

2.5.1 Skewness

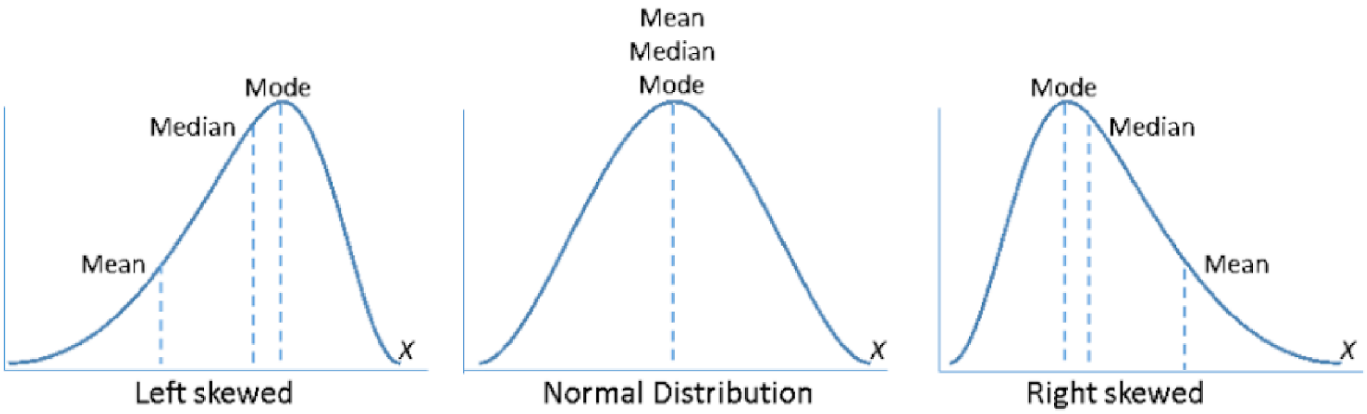
Skew, or skewness, can be mathematically defined as the averaged cubed deviation from the mean divided by the standard deviation cubed. If the result of the computation is greater than zero, the distribution is positively skewed. If it's less than zero, it's negatively skewed and equal to zero means it's symmetric. For interpretation and analysis, focus on downside risk. Negatively skewed distributions have what statisticians call a long left tail (refer to graphs on previous page), which for investors can mean a greater chance of

extremely negative outcomes. Positive skew would mean frequent small negative outcomes, and extremely bad scenarios are not as likely.

2.5.2 Kurtosis

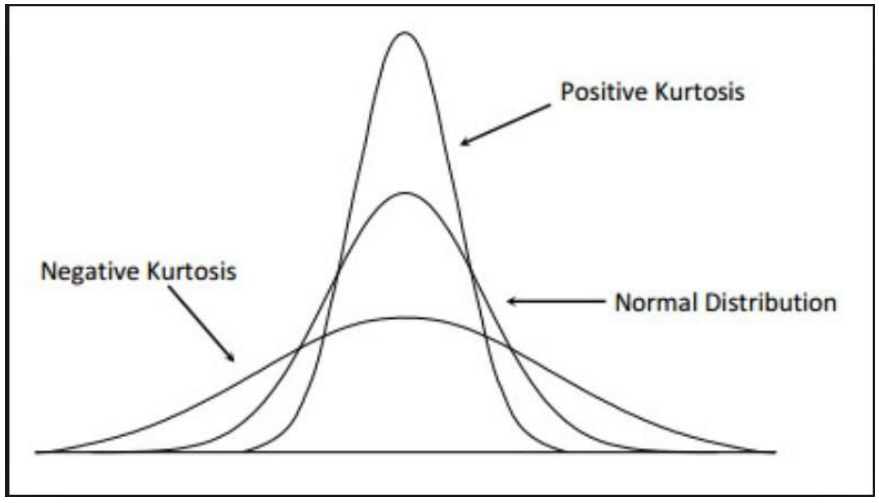
Kurtosis refers to the degree of peak in a distribution. More peak than normal (leptokurtic) means that a distribution also has fatter tails and that there are more chances of extreme outcomes compared to a normal distribution.

- A positive value tells you that you have heavy-tails (i.e. a lot of data in your tails) also called Leptokurtic data.



- A negative value means that you have light-tails (i.e. little data in your tails) also called Platykurtic data.

This heaviness or lightness in the tails usually means that your data looks flatter (or less flat) compared to the normal distribution. The standard normal distribution has a kurtosis of 3, so if your values are close to that then your graph is nearly normal. These nearly normal distributions are called mesokurtic.



3 Numpy module in Python

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

3.1 Numpy arrays(ndarrays)

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

Numpy also provides many functions to create arrays:

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"
```

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

The more important attributes of an ndarray object are:

- **ndarray.ndim** the number of axes (dimensions) of the array.
- **ndarray.shape** the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
- **ndarray.size** the total number of elements of the array. This is equal to the product of the elements of shape.
- **ndarray.dtype** an object describing the type of the elements in the array. One can create or specify dtypes using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- **ndarray.itemsize** the size in bytes of each element of the array.

3.1.1 Array indexing

Numpy offers several ways to index into array.

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
```

- **Slicing:** Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

- **Integer array indexing:** When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

- **Boolean array indexing:** Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

3.1.2 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

3.1.3 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is “sum”:

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:


```
import numpy as np

x = np.array([[1,2],[3,4]])
```

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```