Open Lab 2 - Mathematical Foundations

CSCI 4850-5850 - Neural Networks

Due: Feb. 10 @ 11:00pm

Foundations for Machine Learning

We've briefly touched on the three aspects of machine learning (ML) defined by Tom Mitchell in his book *Machine Learning* (1997) in past lectures:

1. Task (T)
2. Performance Measure (P)
3. Experience (E)

Our goal in machine learning is to construct a formal (i.e. mathematical) framework for these three ideas, and this will constitute a machine learning approach. Of course, the definition above is very broad, and says nothing about whether we find the machine learning algorithm figuratively *useful* or *informative* in some way. For example, just because we write a formal description of a performance measure (P) and calculate its value under a task (T) by using some experience (E), doesn't mean the algorithm has to obtain the *best* value of P. Instead, just *improving* according to P over the course of computing the algorithm could technically be called learning. To start delving into these ideas more, we need a simple example that at least complex *enough* in the sense that it can help us gain some understanding about how one generally constructs T, P, and E, but also that also performs something we might consider *useful* or *informative*. Alternatively, some algorithms are mathematically complex, and require a bit of study before we begin to understand how they work. Instead of diving directly into a machine learning approach, for this assignment we will be getting familiar with some mathematical concepts and the python libraries that we will be using to implement those mathematical concepts. Once we have these tools in hand, we will be prepared to begin exploring how these three elements T, P, and E are constructed, and how one might use them to solve engineering problems.

Classification Tasks and Vectors

Generally, a classification task involves sorting experiences amongst a discrete set of options. One might, for example, think about the last week and separate each day into the classes of "good days" and "bad days". Or maybe one might listen to a song to determine which genre it belongs (pop, rock, r-n-b, reggae, etc.). These are some potential tasks, T. Typically, you will use already-classified examples from the past (experience, E) to motivate your selection process and help you decide how to classify a new example. Being familiar with many rock songs might help one identify key features that are common to rock songs, and new songs which have these same features may be classified as rock then as well. However, different features might be used to identify pop songs, and if a song shares some features of both rock and pop then you will be forced to make a difficult decision. Songs which clearly have many features of the rock genre and few features of the pop genre might be desribed as more similar or *less distant* to other songs in the rock genre, but also less similar or *more distant* to songs in the pop genre. Notice that we are using the terms *similarity* and *distance* where one is generally taken to be roughly the inverse of the other. We often use the term *similarity* to describe relationships between songs, and hence put similar songs into the same genre. However, due the inverse relationship distance has to similarity, it is also reasonable to say that we instead put songs that are less distant into the same genre. Using distance, it's easier to say if something does *not* belong to a category. For example, if we say that a pop song doesn't sound like it belongs in the rock genre, we are saying that the song shares few features with other rock songs. That is, the pop song is distant from other rock songs. This concept is important because you have probably already studied and used some so-called *distance metrics* in other math and CS courses, and we will be using these mathematical tools to describe relationships between experiences in machine learning tasks. For example, a common *distance metric* is the Euclidean distance, defined between two vectors (or points) $\boldsymbol{x}$ and $\boldsymbol{y}$ to be: $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$ where $\boldsymbol{x}$ and $\boldsymbol{y}$ are both vectors of length n.

Vectors play an important role in machine learning. We need a formal way to deliver experiences to a machine learning algorithm, and vectors provide a general mathematical way to do this, and they also help us understand how this mode of delivery shapes the learning and performance of the algorithms that we develop.

As a quick exercise, let's construct a couple of vectors and calculate the Euclidean distance between them:

In [1]:
```
# Prep the environment
import numpy as np

import sympy as sp
sp.init_printing(use_latex=True)

# New stuff!!
import scipy.spatial.distance as ssd

import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
x = np.array([1,1])
print(x)
y = np.array([0.5,1.5])
print(y)
```

```
[1 1]
[0.5 1.5]
```

Notice that we have chosen to use two vectors ($\boldsymbol{x}$ and $\boldsymbol{y}$) with a length of $n = 2$ for this example. We create each by constructing a python list of the vector elements and the passing the list to numpy's `array()` function. The reason why we don't use standard python lists should become apparent very soon.

Now, we can calculate their distance...

In [3]:
```
# Direct calculation with numpy
np.sqrt(np.sum(pow(x-y,2.0)))
```

Out[3]: 0.707106781186548

Here we have utilized a nice property of numpy where arrays of the *same size/dimensions* can have an operator or function apply to each of the *corresponding* elements in the arrays. For comparison, the $x - y$ operation would not be computable between two python lists (try it and you will see that it's *not supported* under python). However, numpy recognizes that you would like to perform a subtraction operation for the two vectors *and* that they have the same dimensions. Therefore, it knows to go through the corresponding elements in the vectors and subtract them one at a time, similar to if you had written a for-loop to iterate over the elements. It's a wonderful thing...

The end result of the subtraction operation is then another *temporary* vector (let's call it r) of the same size ($n = 2$) where $r_1 = x_1 - y_1$ and $r_2 = x_2 - y_2$. In a more python-like description, we would say `r[0] = x[0] - y[0]` and `r[1] = x[1] - y[1]` since numpy arrays are 0-indexed just like other python data structures.

Keep in mind that most mathematical descriptions that you see in these examples, as well as in textbooks, will usually be 1-indexed. You will sometimes need to take care to remember that the data structures are 0-indexed instead.

Additionally, the `pow()` function is also applied over the r array, and then the `sum()` function understands that it should sum over the values in the array provided to produce a single *scalar* value which is finally passed to the `sqrt()` function for evaluation of the square-root.

We'll get back to vector data manipulation soon, but before we move on, let's look at a utility function from the scipy.spatial.distance package which can make this even easier:

Distance and Similarity Calculations

In [4]:
```
# Using the scipy library spatial.distance
ssd.euclidean(x,y)
```

Out[4]: 0.707106781186548

There are many functions in numpy and/or scipy which provide useful routines for common mathematical calculations like distances. You can see the list of distance functions here.

Let's explore vectors and distance calculations a little more...

First note that the Euclidean distance is a special case of the so-called Minkowski distance which has an additional parameter, p:

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}} = \|\boldsymbol{x} - \boldsymbol{y}\|_p$$

If we set $p = 2$, we obtain the Euclidean distance:

$$\left(\sum_{i=1}^{n} |x_i - y_i|^2\right)^{\frac{1}{2}} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} = \|\boldsymbol{x} - \boldsymbol{y}\|_2$$

The notation used here to denote the Minkowski distance is worth remembering.

Another common distance metric used in machine learning is the Manhattan distance, which is also a special case of the Minkowski distance when $p = 1$:

$$\left(\sum_{i=1}^{n} |x_i - y_i|^1\right)^{\frac{1}{1}} = \sum_{i=1}^{n} |x_i - y_i| = \|\boldsymbol{x} - \boldsymbol{y}\|_1$$

You can calculate these distances using the `minkowski()` function:

In [5]:
```
print('Euclidean: %f'%ssd.minkowski(x,y,2))
print('Manhattan: %f'%ssd.minkowski(x,y,1))
```

```
Euclidean: 0.707107
Manhattan: 1.000000
```

Another commonly used metric is the *cosine similarity*. Note that this is not a distance function, but a **similarity** function. For all of the metrics above, similar vectors have *low* distance from one another and dissimilar vectors have *high* distance from one another. However, under the cosine similarity metric, similar items will have *high* similarity and dissimilar items will have *low* similarity.

Cosine similarity is defined as follows: $\cos\theta = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2}$

where

$\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}}$ (so-called: p-norm)

and

$\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{n} x_i y_i$ (so-called: dot-product)

and when $p = 2$, we refer to this as the L2-norm of $\boldsymbol{x}$ (or somethimes simply the *length* of the vector described by $\boldsymbol{x}$).

Let's calculate the cosine similarity between $\boldsymbol{x}$ and $\boldsymbol{y}$:

In [6]:
```
np.sum(x*y)/(np.sqrt(np.dot(x,x))*np.sqrt(np.dot(y,y)))
```

Out[6]: 0.894427190999916

An alternative way to formulate the L2-norm is to take the square-root of the dot product between a vector and itself:

$\|\boldsymbol{x}\|_2 = \sqrt{\boldsymbol{x} \cdot \boldsymbol{x}}$

The value, θ, represents the angle between the two vectors measured in *radians*. However, instead of solving for θ, we usually are only interested in $\cos\theta$ since this value will be in the range $[-1, 1]$ where vectors with high similarity will have a $\cos\theta$ close to 1, *orthogonal* vectors will have a $\cos\theta$ close to 0, and vectors which nearly lie on the same line but point in *opposite* directions will have a $\cos\theta$ of near -1.

Due to this property, another common distance metric is the *cosine distance* which is defined to be $1 - \cos\theta$, or:

$1 - \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2}$

This formulation allows for vectors which point in similar directions to have a distance of *zero*, while *orthgonal* vectors have a distance of *one*, and vectors pointing in opposite directions will have a distance of *two* (this is however, the maximum distance between two vectors using cosine distance).

Also, the dot product, $\boldsymbol{x} \cdot \boldsymbol{y}$, alone has similar properties to cosine similarity so it is sometimes preferred due to the ease of computation using numpy's `dot()` function. For example, if you can know *a priori* (beforehand, ahead of time) that your vectors will all be of length $n = 1$ then the dot product is an *exact* replacement for cosine similarity. We may investigate different ways to transform distance metrics into similarity metrics and vice-versa in future assignments.

Let's calculate cosine similarity and distance now...

In [7]:
```
print('Cosine similarity: %f'%(np.dot(x,y) /
                               (np.sqrt(np.dot(x,x)) * np.sqrt(np.dot(y,y)))))
print('Cosine distance: %f'%ssd.cosine(x,y))
```

```
Cosine similarity: 0.894427
Cosine distance: 0.105573
```

As mentioned above, we can rewrite cosine similarity as follows:

$\boldsymbol{x} \cdot \boldsymbol{y} = \|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2 \cos\theta$

This means that the dot product ($\boldsymbol{x} \cdot \boldsymbol{y}$) is an *unscaled* or *unnormalized* version of the cosine similarity metric. You can compute it in two different ways:

In [8]:
```
# The dot product
print(np.sum(x*y))
print(np.dot(x,y))
```

```
2.0
2.0
```

The `dot()` function actually is very useful for working with matrices and tensors as well, and will come in handy later. For now, if you wanted to learn a little more about its capabilities, you could just ask for help in the notebook. This will look up the basic documentation provided for the function:

In [9]:
```
help(np.dot)
```

```
Help on function dot in module numpy:

dot(...)
    dot(a, b, out=None)

    Dot product of two arrays. Specifically,

    - If both `a` and `b` are 1-D arrays, it is inner product of vectors
      (without complex conjugation).

    - If both `a` and `b` are 2-D arrays, it is matrix multiplication,
      but using :func:`matmul` or ``a @ b`` is preferred.

    - If either `a` or `b` is 0-D (scalar), it is equivalent to :func:`multiply`
      and using ``numpy.multiply(a, b)`` or ``a * b`` is preferred.

    - If `a` is an N-D array and `b` is a 1-D array, it is a sum product over
      the last axis of `a` and `b`.
```

```
    - If `a` is an N-D array and `b` is an M-D array (where ``M>=2``), it is a
      sum product over the last axis of `a` and the second-to-last axis of `b`::

        dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

    Parameters
    ----------
    a : array_like
        First argument.
    b : array_like
        Second argument.
    out : ndarray, optional
        Output argument. This must have the exact kind that would be returned
        if it was not used. In particular, it must have the right type, must be
        C-contiguous, and its dtype must be the dtype that would be returned
        for `dot(a,b)`. This is a performance feature. Therefore, if these
        conditions are not met, an exception is raised, instead of attempting
        to be flexible.

    Returns
    -------
    output : ndarray
        Returns the dot product of `a` and `b`.  If `a` and `b` are both
        scalars or both 1-D arrays then a scalar is returned; otherwise
        an array is returned.
        If `out` is given, then it is returned.

    Raises
    ------
    ValueError
        If the last dimension of `a` is not the same size as
        the second-to-last dimension of `b`.

    See Also
    --------
    vdot : Complex-conjugating dot product.
    tensordot : Sum products over arbitrary axes.
    einsum : Einstein summation convention.
    matmul : '@' operator as method with out parameter.

    Examples
    --------
    >>> np.dot(3, 4)
    12

    Neither argument is complex-conjugated:

    >>> np.dot([2j, 3j], [2j, 3j])
    (-13+0j)

    For 2-D arrays it is the matrix product:

    >>> a = [[1, 0], [0, 1]]
    >>> b = [[4, 1], [2, 2]]
    >>> np.dot(a, b)
    array([[4, 1],
           [2, 2]])

    >>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
    >>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
    >>> np.dot(a, b)[2,3,2,1,2,2]
    499128
    >>> sum(a[2,3,2,:] * b[1,2,:,2])
    499128
```

Matrices and Vectors

Now that we have worked with vectors a little, we will find ourselves needing to work with many of them at the same time. For that, we usually stack our vectors into a matrix so that we can utilize them for matrix-vector and matrix-matrix multiplication operations.

Let's make a new vector, z, and then combine each of the 2-element vectors x, y, z into a 3-row matrix.

The result (we will call it: data) will be a 3x2 matrix.

In [10]:
```
# Make another vector
z = np.array([1.5,0.25])

# Combine all vectors -by row- to form a matrix
data = np.array([x,y,z])
data
```

Out[10]:
```
array([[1.  , 1.  ],
       [0.5 , 1.5 ],
       [1.5 , 0.25]])
```

If you prefer, you can use sympy to print matrices and vectors for $\LaTeX$ style rendering. Note that you will get column-wise vectors by default. You can always see the transpose instead using the `T` member of the Matrix class.

In [11]:
```
# Note that Matrix() was imported from sympy
sp.Matrix(data)
```

Out[11]:
$$\begin{bmatrix} 1.0 & 1.0 \\ 0.5 & 1.5 \\ 1.5 & 0.25 \end{bmatrix}$$

In [12]:
```
# Same for vectors - but you will get a column-matrix
sp.Matrix(x)
```

Out[12]:
$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

In [13]:
```
# Maybe you want the transpose - use T?
sp.Matrix(x).T
```

Out[13]:
$$\begin{bmatrix} 1 & 1 \end{bmatrix}$$

Pairwise Distance Matrices

A common kind of matrix used in machine learning is the *pairwise distance* matrix. This matrix consists of distance calculations for m different vectors obtained from an m x n data matrix.

Our matrix was 3x2, which means we have $m = 3$ vectors, each of dimension $n = 2$. What we want to do is compare each of the 3 vectors to itself and the other two. After doing this for all 3 vectors, we will have a 3x3 matrix which contains the distance from vector i to j where $i = 1...n$ and $j = 1...n$.

The `pdist()` function from sypy.spatial is useful for making this calculation, but to get the full m x m to display, we then convert the result into a square matrix using the `squareform()` function. (Since the matrix is symmetric, the internal storage of the matrix prefers to just keep the lower triangle instead of the entire thing, but *printing* the square form is better supported.)

Here are some examples using the data we created above for a few distance metrics:

In [14]:
```
ssd.squareform(ssd.pdist(data,metric='euclidean'))
```

Out[14]:
```
array([[0.        , 0.70710678, 0.90138782],
       [0.70710678, 0.        , 1.60078106],
       [0.90138782, 1.60078106, 0.        ]])
```

In [15]:
```
ssd.squareform(ssd.pdist(data,metric='minkowski',p=2))
```

Out[15]:
```
array([[0.        , 0.70710678, 0.90138782],
       [0.70710678, 0.        , 1.60078106],
       [0.90138782, 1.60078106, 0.        ]])
```

In [16]:
```
ssd.squareform(ssd.pdist(data,metric='minkowski',p=1))
```

Out[16]:
```
array([[0.  , 1.  , 1.25],
       [1.  , 0.  , 2.25],
       [1.25, 2.25, 0.  ]])
```

In [17]:
```
ssd.squareform(ssd.pdist(data,metric='cosine'))
```

Out[17]:
```
array([[0.        , 0.10557281, 0.18626653],
       [0.10557281, 0.        , 0.53211228],
       [0.18626653, 0.53211228, 0.        ]])
```

In [18]:
```
# Just to make it pretty...
sp.Matrix(ssd.squareform(ssd.pdist(data,metric='cosine')))
```

Out[18]:
$$\begin{bmatrix} 0.0 & 0.105572809000084 & 0.186266528793265 \\ 0.105572809000084 & 0.0 & 0.532112279580967 \\ 0.186266528793265 & 0.532112279580967 & 0.0 \end{bmatrix}$$

Matrix-Vector Operations

We will be using common linear algebra operations often for neural network computations.

One common operations are the matrix-vector product. I will show how to print the expression we are wanting to compute using sympy, but most of the time, I will *not* ask you to do this for your assignments. Sympy, by default, will simplify expressions automatically, so it takes some work to turn off the simplication engine in order to use it for printing.

Nevertheless, let's try to compute a matrix-vector product using the numpy `dot()` function...

In [19]:
```
# Let's use sympy to print a pretty version of the operation..
sp.Eq(sp.MatMul(sp.Matrix(data),sp.Matrix(x),evaluate=False),
      sp.Matrix(np.dot(data,x)),evaluate=False)
```

Out[19]:

$$\begin{bmatrix} 1.0 & 1.0 \\ 0.5 & 1.5 \\ 1.5 & 0.25 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 2.0 \\ 1.75 \end{bmatrix}$$

In [20]:
```
# Most of the time, you will simply use:
np.dot(data,x)
```

Out[20]:
```
array([2.  , 2.  , 1.75])
```

In [21]:
```
# And a simple pretty-print of the results...
sp.Matrix(np.dot(data,x))
```

Out[21]:
$$\begin{bmatrix} 2.0 \\ 2.0 \\ 1.75 \end{bmatrix}$$

Note how numpy knows that `data` is a 3x2 array and `x` is a 2-dimensional vector. This means that it treats the vector as a 2x1 matrix for this operation. The result is therefore a 3x1, which is expressed more clearly in the pretty-printed version.

This property of numpy which allows it match vector, matrix, and other arrays of arbitrary dimension (tensors - we will see them later) by correspondingly sized dimensions will be useful.

Note what happens if we reverse the order of `data` and `x`:

In [22]:
```
np.dot(x,data)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-3ed90fd0f3fa> in <module>
----> 1 np.dot(x,data)

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (2,) and (3,2) not aligned: 2 (dim 0) != 3 (dim 0)
```

While numpy is smart enough to match corresponding dimensions when requested, you still have to formulate your computation in the order that makes sense for the array and matrix you are using.

In the case above, since numpy sees the `x` vector as a column vector, it has size 2x1, and trying to multply a 2x1 by a 3x2 doesn't work (the "inner" numbers would need to match in the order of calculations, but 1 and 3 are not the same).

So, just remember, numpy is *great*, but *order matters*...

Matrix-Matrix operations

Numpy is also good for performing matrix-matrix operations (as well as vector-tensor, matrix-tensor, and tensor-tensor... we will do those later).

For now, we will calculate `data` times its transpose. That's a 3x2 times a 2x3, so the result is a 3x3. Again, I will use sympy to show the math, but most of the time the `dot()` function is what you will be using...

In [23]:
```
sp.Eq(sp.MatMul(sp.Matrix(data),sp.Matrix(data).T,evaluate=False),
      sp.Matrix(np.dot(data,data.T)),evaluate=False)
```

Out[23]:
$$\begin{bmatrix} 1.0 & 1.0 \\ 0.5 & 1.5 \\ 1.5 & 0.25 \end{bmatrix} \begin{bmatrix} 1.0 & 0.5 & 1.5 \\ 1.0 & 1.5 & 0.25 \end{bmatrix} = \begin{bmatrix} 2.0 & 2.0 & 1.75 \\ 2.0 & 2.5 & 1.125 \\ 1.75 & 1.125 & 2.3125 \end{bmatrix}$$

In [24]:
```
# Let's use numpy!
sp.Matrix(np.dot(data,data.T))
```

Out[24]:
$$\begin{bmatrix} 2.0 & 2.0 & 1.75 \\ 2.0 & 2.5 & 1.125 \\ 1.75 & 1.125 & 2.3125 \end{bmatrix}$$

Notice, this is equivalent to calculating the **pairwise dot-product** between all three vectors!

With a little more work, we can calculate the pairwise cosine distances like we did using the `pdist()` function and the `metric=cosine` argument, but using only pure numpy...

In [25]:
```
def mynorm(a_vector):
    return np.sqrt(np.dot(a_vector,a_vector))

A = np.round(
    1.0 - (np.dot(data,data.T) / np.dot(
    np.reshape(np.apply_along_axis(mynorm,1,data),[3,1]),
    np.reshape(np.apply_along_axis(mynorm,1,data),[1,3]))),12)
# Notice how the object returned by final line of code
# in a code cell gets printed?
sp.Matrix(A)
```

Out[25]:
$$\begin{bmatrix} 0.0 & 0.105572809 & 0.186266528793 \\ 0.105572809 & 0.0 & 0.532112279581 \\ 0.186266528793 & 0.532112279581 & 0.0 \end{bmatrix}$$

There are a lot of new functions being used here. Some are fairly straight-forward to understand, such as `round()` which simply rounds numbers to the specified number of digits following the decimal point (in this case, 12). Also, the `dot()` and `T` operations you have already seen.

However, another useful tool is the `apply_along_axis()` function. This function allows us to call a function and apply it to individual vectors (or even sub-matrices if we are using tensors) in a matrix. In the case of the cosine distance, we need to calculate the vector norms in the denominator of the equation. First, we create a function, `mynorm` which takes a vector as input, and calculates the norm (a scalar) for that vector. We use this as the first argument to apply_along_axis since this is the function we need to apply to the vectors in `data`. Next, we know we need to apply the function to the vectors in each *row* and the first dimension of a matrix specifies the number of *rows* in that matrix. Thus, if we want to apply the function to each row, we select the first dimension (0) as the second argument. Finally, we supply the matrix containing the vectors we are performing the operation on: `data`. The end result is a 3-element vector which contains the *lengths* (vector norms) of each of the 3 vectors in the matrix.

The `reshape()` function then comes in handy for forming the 3 elements into first a 3x1 matrix, and then a 1x3 matrix (one matrix for each call). By multiplying the two matrices together with `dot()` we have calculated all pairs of vector norms (a 3x3 matrix - the denominators in the cosine distance equation).

We then use `dot()` to calculate the dot products for all vectors and divide it by the pairwise norm matrix before we finally subtract the result from 1, and pass it to `round()` to eliminate some numerical round-off error accumulation. (Feel free to remove the `round()` function to observe the left-over values along the diagonal, the values of which are very near, but not quite, zero.

Matrix Decompositions

Finally, let's perform some matrix factorizations (or often matrix decomposition). This is similar to finding the factors of a number, like how you can get $12 = 4 * 3$ or $12 = 2 * 6$. Each pair is a factorization of the number 12.

For matrix factorization, we want a combination of matrices that can be multiplied together to obtain a given matrix, $\boldsymbol{A}$.

We will explore how to use these operations in neural networks and machine learning in the future. For now, just learn how to compute them.

Let's start with *Eigen decomposition*, where we want to factor a square and (in most cases) symmetric matrix $\boldsymbol{A}$ (which we computed above) into:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\intercal}$$

In [26]:
```
L,Q = np.linalg.eig(A)

sp.MatMul(sp.Matrix(Q),sp.Matrix(np.diag(L)),sp.Matrix(Q).T,evaluate=False)
```

Out[26]:
$$\begin{bmatrix} 0.325098539559343 & 0.938831137274348 & -0.113609133702079 \\ 0.657042235063221 & -0.310640328350735 & -0.686875598449439 \\ 0.680151777855141 & -0.148656254783652 & 0.717840425856742 \end{bmatrix} \begin{bmatrix} 0.603064244356408 & 0.0 & 0.0 \\ 0.0 & -0.064425703636322 & 0.0 \\ 0.0 & 0.0 & -0.538638540720086 \end{bmatrix} \begin{bmatrix} 0.325098539559343 \\ 0.938831137274348 \\ -0.113609133702079 \end{bmatrix}$$

The result is a 3x3 matrix of eigen vectors, $\boldsymbol{Q}$, and a *diagonal* matrix of eigen values, $\boldsymbol{\Lambda}$ (which I have assigned to L for simplicity).

You can obtain the original matrix, A, (within a little numerical rounding error) by multiplying the parts together as prescribed above:

In [27]:
```
sp.Matrix(np.round(np.dot(np.dot(Q,np.diag(L)),Q.T),12))
```

Out[27]:
$$\begin{bmatrix} 0.0 & 0.105572809 & 0.186266528793 \\ 0.105572809 & 0.0 & 0.532112279581 \\ 0.186266528793 & 0.532112279581 & 0.0 \end{bmatrix}$$

Another common decomposition is the *singular value decomposition*, which can be applied to m x n (i.e. rectangular) matrices:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\intercal}$$

In [28]:
```
U,S,V = np.linalg.svd(A,full_matrices=True)

sp.MatMul(sp.Matrix(U),sp.Matrix(np.diag(S)),sp.Matrix(V),evaluate=False)
```

Out[28]:
$$\begin{bmatrix} 0.325098539559343 & -0.113609133702079 & 0.938831137274348 \\ 0.657042235063221 & -0.686875598449439 & -0.310640328350735 \\ 0.680151777855141 & 0.717840425856742 & -0.148656254783652 \end{bmatrix} \begin{bmatrix} 0.603064244356408 & 0.0 & 0.0 \\ 0.0 & 0.538638540720086 & 0.0 \\ 0.0 & 0.0 & 0.0644257036363219 \end{bmatrix} \begin{bmatrix} 0.325098539559343 \\ 0.113609133702079 \\ -0.938831137274348 \end{bmatrix}$$

The result is a 3x3 matrix of left-singular vectors, $\boldsymbol{U}$, and a *diagonal* matrix of singular values, $\boldsymbol{\Sigma}$ (which I have assigned to S for simplicity), and a 3x3 matrix of right-singular vectors, $\boldsymbol{V}$. (However, note that V is already in transposed form when returned by numpy.)

You can obtain the original matrix, $\boldsymbol{A}$, (within a little numerical rounding error) by multiplying the parts together as prescribed above:

In [29]:
```
sp.Matrix(np.round(np.dot(np.dot(U,np.diag(S)),V),12))
```

Out[29]:

$$\begin{bmatrix} 0.0 & 0.105572809 & 0.186266528793 \\ 0.105572809 & 0.0 & 0.532112279581 \\ 0.186266528793 & 0.532112279581 & 0.0 \end{bmatrix}$$

Assignment

Now that you have had some practice working with vectors and matrices, create a notebook using your new skills that addresses the problems below:

1. Create the vector, $\boldsymbol{x} = [5, 2, 8]$, and calculate its L2-norm and L1-norm (replace $p = 2$ with $p = 1$ in the p-norm equation above).
2. Create another vector, $\boldsymbol{y} = [1, 3, 2]$, and calculate **both** the Euclidean and Cosine distances between $\boldsymbol{y}$ and the vector $\boldsymbol{x}$.
3. Create and pretty-print the following matrix.
   $$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix}$$
4. Multiply $\boldsymbol{A}$ by $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. That is, calculate $\boldsymbol{Ax}$ and then $\boldsymbol{Ay}$. Be sure to pretty-print each result.
5. Create the following matrix, $\boldsymbol{B}$, and *then* compute the result of $\boldsymbol{AB}$.
   $$\boldsymbol{B} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$
6. Calculate the *pairwise Euclidean* distance matrix for the row-vectors in $\boldsymbol{A}$ (pretty-print the result(s)).
7. Calculate the square 3x3 matrix $\boldsymbol{C} = \boldsymbol{B}\boldsymbol{B}^\intercal$ (pretty-print the result(s)).
8. Calculate the Eigen Decomposition for $\boldsymbol{C}$ (pretty-print the result(s)).
9. Use the $\boldsymbol{Q}$ and $\boldsymbol{\Lambda}$ matrices from Problem 8 to numerically recalculate $\boldsymbol{C}$ (don't forget to pretty-print the result).
10. Calculate the Singular Value Decomposition for $\boldsymbol{C}$ (pretty-print the result(s)).
11. Use the $\boldsymbol{U}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{V}^\intercal$ matrices from Problem 10 to numerically recalculate $\boldsymbol{C}$ (don't forget to pretty-print the result).

Submission

Compile **well-labeled** solutions to all problems in a single iPython Notebook file, zip and upload your notebook to the assignment system by the deadline at the top of this assignment.