Euclidean vs. Cosine Distance

March 25, 2017 | 10 minute read | Chris Emmery

This post was written as a reply to a question asked in the Data Mining course.

> When to use the cosine similarity?

Let's compare two different measures of distance in a vector space, and why either has its function under different circumstances. Starting off with quite a straight-forward example, we have our vector space `X`, that contains instances with animals. They are measured by their `length`, and `weight`. They have also been labelled by their stage of aging (`young = 0`, `mid = 1`, `adult = 2`). Here's some random data:

```
import numpy as np

X = np.array([[6.6, 6.2, 1],
              [9.7, 9.9, 2],
              [8.0, 8.3, 2],
              [6.3, 5.4, 1],
              [1.3, 2.7, 0],
              [2.3, 3.1, 0],
              [6.6, 6.0, 1],
              [6.5, 6.4, 1],
              [6.3, 5.8, 1],
              [9.5, 9.9, 2],
              [8.9, 8.9, 2],
              [8.7, 9.5, 2],
              [2.5, 3.8, 0],
              [2.0, 3.1, 0],
              [1.3, 1.3, 0]])
```

# Preparing the Data

We'll first put our data in a `DataFrame` table format, and assign the correct labels per column:

```
import pandas as pd

df = pd.DataFrame(X, columns=['weight', 'length', 'label'])
```

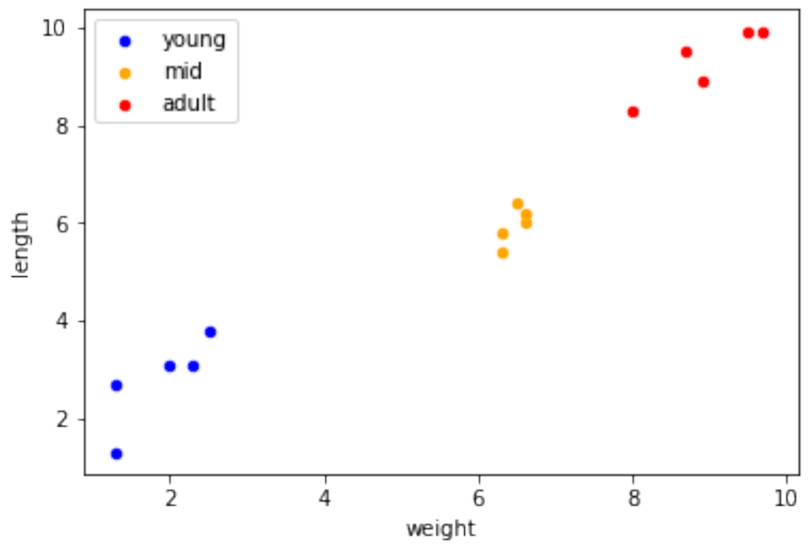
```
df
```

	weight	length	label
0	6.6	6.2	1.0
1	9.7	9.9	2.0
2	8.0	8.3	2.0
3	6.3	5.4	1.0
4	1.3	2.7	0.0
5	2.3	3.1	0.0
6	6.6	6.0	1.0
7	6.5	6.4	1.0
8	6.3	5.8	1.0
9	9.5	9.9	2.0
10	8.9	8.9	2.0
11	8.7	9.5	2.0
12	2.5	3.8	0.0
13	2.0	3.1	0.0
14	1.3	1.3	0.0

Now the data can be plotted to visualize the three different groups. They are subsetted by their label, assigned a different colour and label, and by repeating this they form different layers in the scatter plot.

```
%matplotlib inline

ax = df[df['label'] == 0].plot.scatter(x='weight', y='length', c='blue', label='young')
ax = df[df['label'] == 1].plot.scatter(x='weight', y='length', c='orange', label='mid', ax=
ax = df[df['label'] == 2].plot.scatter(x='weight', y='length', c='red', label='adult', ax=a
ax
```



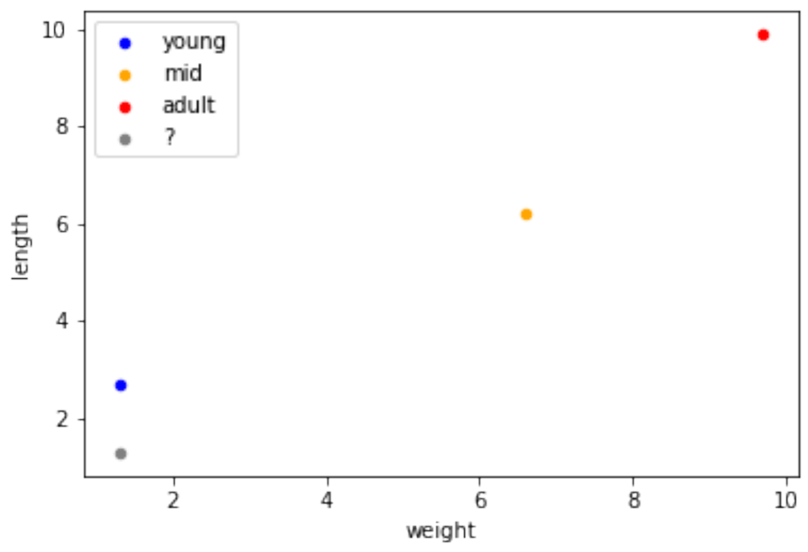
Looking at the plot above, we can see that the three classes are pretty well distinguishable by these two features that we have. Say that we apply k-NN to our data that will learn to classify new instances based on their distance to our known instances (and their labels). The algorithm needs a distance metric to determine which of the known instances are closest to the new one. Let’s try to choose between either euclidean or cosine for this example.

# Picking our Metric

Considering instance #0, #1, and #4 to be our known instances, we assume that we don’t know the label of #14. Plotting this will look as follows:

```
df2 = pd.DataFrame([df.iloc[0], df.iloc[1], df.iloc[4]], columns=['weight', 'length', 'labe
df3 = pd.DataFrame([df.iloc[14]], columns=['weight', 'length', 'label'])

ax = df2[df2['label'] == 0].plot.scatter(x='weight', y='length', c='blue', label='young')
ax = df2[df2['label'] == 1].plot.scatter(x='weight', y='length', c='orange', label='mid', a
ax = df2[df2['label'] == 2].plot.scatter(x='weight', y='length', c='red', label='adult', ax
ax = df3.plot.scatter(x='weight', y='length', c='gray', label='?', ax=ax)
ax
```



## Euclidean

Our euclidean distance function can be defined as follows:

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Where x and y are two vectors. Or:

```
def euclidean_distance(x, y):
    return np.sqrt(np.sum((x - y) ** 2))
```

Let’s see this for all our vectors:

```
x0 = X[0][:-1]
x1 = X[1][:-1]
x4 = X[4][:-1]
x14 = X[14][:-1]
print(" x0:", x0, "\n x1:", x1, "\n x4:", x4, "\nx14:", x14)
```

```
x0:  [ 6.6  6.2]
x1:  [ 9.7  9.9]
x4:  [ 1.3  2.7]
x14: [ 1.3  1.3]
```

Doing the calculations:

```
print(" x14 and x0:", euclidean_distance(x14, x0), "\n",
      "x14 and x1:", euclidean_distance(x14, x1), "\n",
      "x14 and x4:", euclidean_distance(x14, x4))
```

```
x14 and x0: 7.21803297305
x14 and x1: 12.0216471417
x14 and x4: 1.4
```

According to euclidean distance, instance #14 is closest to #4. Our 4th instance had the label:

```
X[4]
```

```
array([ 1.3,  2.7,  0. ])
```

`0 = young`, which is what we would visually also deem the correct label for this instance.

Now let’s see what happens when we use Cosine similarity.

## Cosine

Our cosine similarity function can be defined as follows:

$$\frac{x \bullet y}{\sqrt{x \bullet x}\sqrt{y \bullet y}}$$

Where x and y are two vectors. Or:

```
def cosine_similarity(x, y):
    return np.dot(x, y) / (np.sqrt(np.dot(x, x)) * np.sqrt(np.dot(y, y)))
```

Let’s see these calculations for all our vectors:

```
print(" x14 and x0:", cosine_similarity(x14, x0), "\n",
      "x14 and x1:", cosine_similarity(x14, x1), "\n",
      "x14 and x4:", cosine_similarity(x14, x4))
```

```
x14 and x0: 0.999512076087
x14 and x1: 0.999947942424
x14 and x4: 0.943858356366
```

According to cosine similarity, instance #14 is closest to #1. However, our 1st instance had the label:

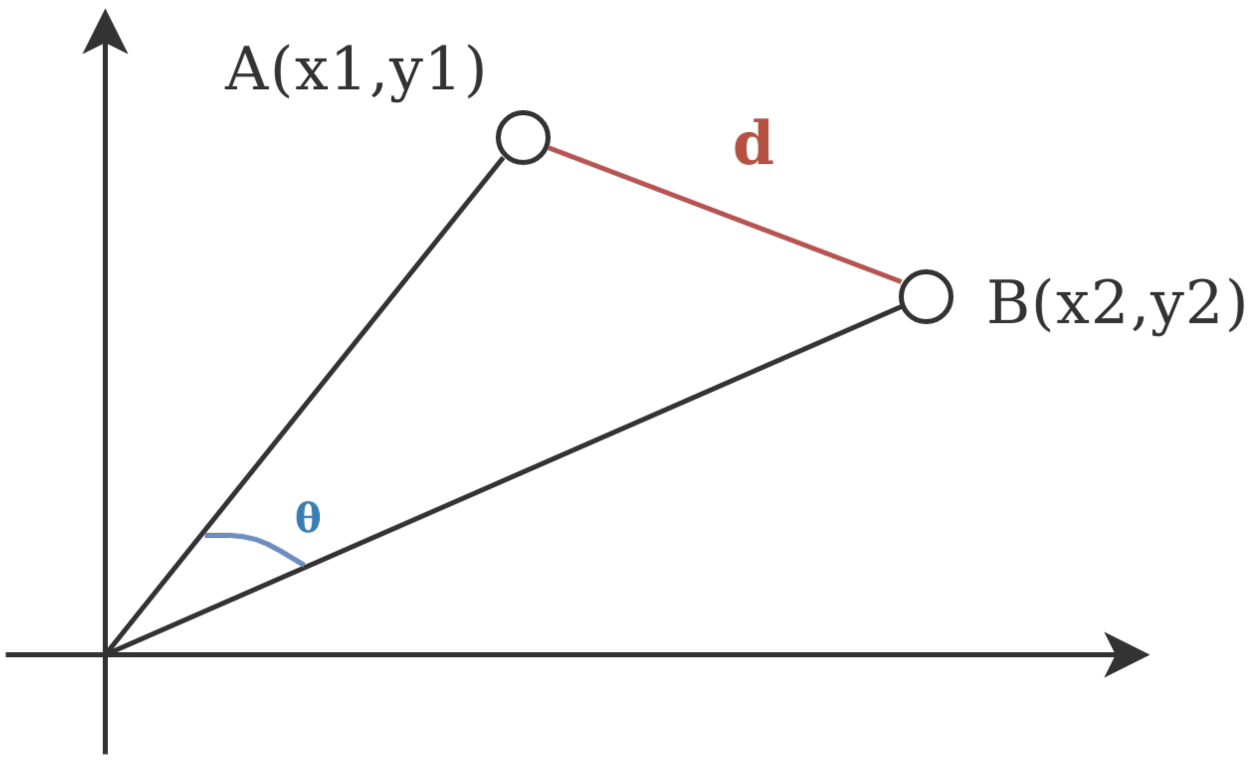
```
X[1]
```

```
array([ 9.7,  9.9,  2. ])
```

`2 = adult`, which is definitely NOT what we would deem the correct label!

# So What Happened?

Consider the following picture:



This is a visual representation of euclidean distance (d) and cosine similarity (θ). While cosine looks at the **angle** between vectors (thus not taking into regard their weight or magnitude), euclidean distance is similar to using a ruler to actually measure the distance. In our example the angle between `x14` and `x4` was larger than those of the other vectors, even though they were further away.

# When to Use Cosine?

Cosine similarity is generally used as a metric for measuring distance when the magnitude of the vectors does not matter. This happens for example when working with text data represented by word counts. We could assume that when a word (e.g. `science`) occurs more frequent in document 1 than it does in document 2, that document 1 is more related to the topic of `science`. However, it could also be the case that we are working with documents of uneven lengths (Wikipedia articles for example). Then, `science` probably occurred more in document 1 just because it was way longer than document 2. Cosine similarity corrects for this.

Text data is the most typical example for when to use this metric. However, you might also want to apply cosine similarity for other cases where some properties of the instances make so that the weights might be larger without meaning anything different. Sensor values that were captured in various lengths (in time) between instances could be such an example.

# How do Euclidean Distance and Cosine Similarity Relate?

Let’s consider two of our vectors, their euclidean distance, as well as their cosine similarity.

```
print("vectors \t", x0, x1, "\n"
      "euclidean \t", euclidean_distance(x0, x1), "\n"
      "cosine \t\t", cosine_similarity(x0, x1))
```

```
vectors      [ 6.6  6.2] [ 9.7  9.9]
euclidean    4.82700735446
cosine       0.99914133854
```

Cosine similarity takes a unit length vector to calculate dot products. However, what happens if we do the same for the vectors we’re calculating the euclidian distance for (i.e. normalize them)? For this, we can for example use the L_1 norm:

$$\sum_i x_i$$

Or the L_2 norm:

$$\sqrt{\sum_i x_i^2}$$

In functions:

```
def l1_normalize(v):
    norm = np.sum(v)
    return v / norm

def l2_normalize(v):
    norm = np.sqrt(np.sum(np.square(v)))
    return v / norm
```

We divide the values of our vector by these norms to get a normalized vector.

Applying the L_1 norm to our vectors will make them sum up to 1 respectively, as such:

```
x0_n = l1_normalize(x0)
x1_n = l1_normalize(x1)
print(x0_n, x1_n)
```

```
[ 0.515625  0.484375] [ 0.49489796  0.50510204]
```

Let’s compare the result we had before against these normalized vectors:

```
print("vectors \t", x0_n, x1_n, "\n"
      "euclidean \t", euclidean_distance(x0_n, x1_n), "\n"
      "cosine \t\t", cosine_similarity(x0_n, x1_n))
```

```
vectors      [ 0.515625  0.484375] [ 0.49489796  0.50510204]
euclidean    0.0293124622303
cosine       0.99914133854
```

As we can see, before, the distance was pretty big, but the cosine similarity very high. Now that we normalized our vectors, it turns out that the distance is *now* very small. The same pattern occurs when we compare it against vector 4. Unnormalized:

```
print("vectors \t", x0, x4, "\n"
      "euclidean \t", euclidean_distance(x0, x4), "\n"
      "cosine \t\t", cosine_similarity(x0, x4))
```

```
vectors      [ 6.6  6.2] [ 1.3  2.7]
euclidean    6.35137780328
cosine       0.933079411589
```

Normalized:

```
x4_n = l1_norm(x4)

print("vectors \t", x0_n, x4_n, "\n"
      "euclidean \t", euclidean_distance(x0_n, x4_n), "\n"
      "cosine \t\t", cosine_similarity(x0_n, x4_n))
```

```
vectors      [ 0.515625  0.484375] [ 0.325  0.675]
euclidean    0.269584460327
cosine       0.933079411589
```

Notice that because the cosine similarity is a bit lower between `x0` and `x4` than it was for `x0` and `x1`, the euclidean distance is now also a bit larger. To take this point home, let’s construct a vector that is almost evenly distant in our euclidean space, but where the cosine similarity is much lower (because the angle is larger):

```
x00 = np.array([0.1, 6])

print("vectors \t", x0, x00, "\n"
      "euclidean \t", euclidean_distance(x0, x00), "\n"
      "cosine \t\t", cosine_similarity(x0, x00))
```

```
vectors      [ 6.6  6.2] [ 0.1  6. ]
euclidean    6.50307619516
cosine       0.696726168728
```

If we normalize this, we should see the same behaviour from our euclidean distance (i.e. it should be larger than for `x0` and `x4`). As follows:

```
x00_n = l1_normalize(x00)
```

```
print("vectors \t", x0_n, x00_n, "\n"
      "euclidean \t", euclidean_distance(x0_n, x00_n), "\n"
      "cosine \t\t", cosine_similarity(x0_n, x00_n))
```

```
vectors      [ 0.515625  0.484375] [ 0.01639344  0.98360656]
euclidean    0.706020039207
cosine       0.696726168728
```

# Cosine in Action

So when is cosine handy? Let's consider four articles from Wikipedia. We use the Wikipedia API to extract them, after which we can access their text with the `.content` method.

```
import wikipedia

q1 = wikipedia.page('Machine Learning')
q2 = wikipedia.page('Artifical Intelligence')
q3 = wikipedia.page('Soccer')
q4 = wikipedia.page('Tennis')
```

We represent these by their frequency vectors. Each instance is a document, and each word will be a feature. The feature values will then represent how many times a word occurs in a certain document. So the feature `ball`, will probably be 0 for both machine learning and AI, but definitely not 0 for soccer and tennis. For this example I'll use `sklearn`:

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer()
X = np.array(cv.fit_transform([q1.content, q2.content, q3.content, q4.content]).todense())
```

The `CountVectorizer` by default splits up the text into words using white spaces. We can do the same to see how many words are in each article. Like this:

```
print("ML \t", len(q1.content.split()), "\n"
      "AI \t", len(q2.content.split()), "\n"
      "soccer \t", len(q3.content.split()), "\n"
      "tennis \t", len(q4.content.split()))
```

```
ML        3694
AI        10844
soccer    6134
tennis    9715
```

AI is a much larger article than Machine Learning (ML). This would mean that if we do not normalize our vectors, AI will be much further away from ML just because it has many more words. ML will probably be closer to an article with less words. Let's try it out:

```
print("ML - AI \t", euclidean_distance(X[0], X[1]), "\n"
      "ML - soccer \t", euclidean_distance(X[0], X[2]), "\n"
      "ML - tennis \t", euclidean_distance(X[0], X[3]))
```

```
ML - AI        661.102110116
ML - soccer    459.307086817
ML - tennis    805.405487938
```

Here we can see pretty clearly that our prior assumptions have been confirmed. ML seems to be closest to soccer, which doesn't make a lot of sense intuitively. So, what happens if we look at cosine similairty (thus normalising our vectors)?

```
print("ML - AI \t", cosine_similarity(X[0], X[1]), "\n"
      "ML - soccer \t", cosine_similarity(X[0], X[2]), "\n"
      "ML - tennis \t", cosine_similarity(X[0], X[3]))
```

```
ML - AI        0.886729067818
ML - soccer    0.785735757699
ML - tennis    0.797450973312
```

ML is closer to AI! Granted, it still seems pretty close to soccer an tennis judging from these scores, but please note that word frequency is not that great of a representation for texts with such rich content.

## Categorize a Tweet

Now, just for fun, let's see how this plays out for the following tweet by OpenAI:

```
ml_tweet = "New research release: overcoming many of Reinforcement Learning's limitations w
x = np.array(cv.transform([ml_tweet]).todense())[0]
```

Again we represent this tweet as a word vector, and we try to measure the distance between the tweet and our four wikipedia documents:

```
print("tweet - ML \t", euclidean_distance(x[0], X[0]), "\n"
      "tweet - AI \t", euclidean_distance(x[0], X[1]), "\n"
      "tweet - soccer \t", euclidean_distance(x[0], X[2]), "\n"
      "tweet - tennis \t", euclidean_distance(x[0], X[3]))
```

```
tweet - ML        342.575539115
tweet - AI        945.624661269
tweet - soccer    676.677914521
tweet - tennis    1051.61589946
```

Well, that worked out pretty well at first glance, it's closest to ML. However, see how it's also closer to soccer than AI? There's so many dimensions that come into play here that it's hard to say why this is the case. However, soccer being our second smallest document might have something to do with it. Now we'll do the same for cosine:

```
print("tweet - ML \t", cosine_similarity(x, X[0]), "\n"
      "tweet - AI \t", cosine_similarity(x, X[1]), "\n"
      "tweet - soccer \t", cosine_similarity(x, X[2]), "\n"
      "tweet - tennis \t", cosine_similarity(x, X[3]))
```

```
tweet - ML        0.299293065515
tweet - AI        0.215356854916
tweet - soccer    0.135323358719
tweet - tennis    0.129773245953
```

There we go! This seems definitely more in line with our intuitions. So, remember how euclidean distance in this example seemed to slightly relate to the length of the document? Let’s try the same for a soccer tweet, by Manchester United:

```
so_tweet = "#LegendsDownUnder The Reds are out for the warm up at the @nibStadium. Not long
x2 = np.array(cv.transform([so_tweet]).todense())[0]
```

Same trick, different tweet:

```
print("tweet - ML \t", euclidean_distance(x2, X[0]), "\n"
      "tweet - AI \t", euclidean_distance(x2, X[1]), "\n"
      "tweet - soccer \t", euclidean_distance(x2, X[2]), "\n"
      "tweet - tennis \t", euclidean_distance(x2, X[3]))
```

```
tweet - ML        340.549555865
tweet - AI        943.455351355
tweet - soccer    673.818224746
tweet - tennis    1048.82124311
```

See how awfully similar these distances are to that of our previous tweet, even though there’s very little overlap? Now let’s try the same with cosine similarity:

```
print("tweet - ML \t", cosine_similarity(x2, X[0]), "\n"
      "tweet - AI \t", cosine_similarity(x2, X[1]), "\n"
      "tweet - soccer \t", cosine_similarity(x2, X[2]), "\n"
      "tweet - tennis \t", cosine_similarity(x2, X[3]))
```

```
tweet - ML        0.437509648194
tweet - AI        0.464447992614
tweet - soccer    0.611865382744
tweet - tennis    0.597261139457
```

Nailed it!

Hopefully this, by example, proves why for text data normalizing your vectors can make all the difference!