Distance Measures for Machine Learning

Hamming Distance Hamming distance calculates the distance between two binary vectors, also referred to as binary strings or bitstrings for short.

You are most likely going to encounter bitstrings when you one-hot encode categorical columns of data.

For example, if a column had the categories 'red,' 'green,' and 'blue,' you might one hot encode each example as a bitstring with one bit for each column.

red = [1, 0, 0] green = [0, 1, 0] blue = [0, 0, 1] The distance between red and green could be calculated as the sum or the average number of bit differences between the two bitstrings. This is the Hamming distance.

For a one-hot encoded string, it might make more sense to summarize to the sum of the bit differences between the strings, which will always be a 0 or 1.

HammingDistance = sum for i to N abs(v1[i] – v2[i]) For bitstrings that may have many 1 bits, it is more common to calculate the average number of bit differences to give a hamming distance score between 0 (identical) and 1 (all different).

HammingDistance = (sum for i to N abs(v1[i] – v2[i])) / N We can demonstrate this with an example of calculating the Hamming distance between two bitstrings, listed below.

```
#calculating hamming distance between bit strings

# calculate hamming distance
def hamming_distance(a, b):
  return sum(abs(e1 - e2) for e1, e2 in zip(a, b)) / len(a)

# define data
row1 = [0, 0, 0, 0, 0, 1]
row2 = [0, 0, 0, 0, 1, 0]
# calculate distance
dist = hamming_distance(row1, row2)
print(dist)
```

```
0.3333333333333333
```

We can also perform the same calculation using the hamming() function from SciPy. The complete example is listed below.

```
# calculating hamming distance between bit strings
from scipy.spatial.distance import hamming
# define data
row1 = [0, 0, 0, 0, 0, 1]
row2 = [0, 0, 0, 0, 1, 0]
# calculate distance
```

```
dist = hamming(row1, row2)
print(dist)
```

```
0.3333333333333333
```

Euclidean distance is calculated as the square root of the sum of the squared differences between the two vectors.

EuclideanDistance = sqrt(sum for i to N (v1[i] – v2[i])^2) If the distance calculation is to be performed thousands or millions of times, it is common to remove the square root operation in an effort to speed up the calculation. The resulting scores will have the same relative proportions after this modification and can still be used effectively within a machine learning algorithm for finding the most similar examples.

EuclideanDistance = sum for i to N (v1[i] – v2[i])^2 This calculation is related to the L2 vector norm and is equivalent to the sum squared error and the root sum squared error if the square root is added.

We can demonstrate this with an example of calculating the Euclidean distance between two real-valued vectors, listed below.

```

# calculating euclidean distance between vectors
from math import sqrt

# calculate euclidean distance
def euclidean_distance(a, b):
  return sqrt(sum((e1-e2)**2 for e1, e2 in zip(a,b)))

# define data
row1 = [10, 20, 15, 10, 5]
row2 = [12, 24, 18, 8, 7]
# calculate distance
dist = euclidean_distance(row1, row2)
print(dist)


```

```
6.082762530298219
```

We can also perform the same calculation using the euclidean() function from SciPy. The complete example is listed below.

```

# calculating euclidean distance between vectors
from scipy.spatial.distance import euclidean
# define data
row1 = [10, 20, 15, 10, 5]
row2 = [12, 24, 18, 8, 7]
# calculate distance
```

```
dist = euclidean(row1, row2)
print(dist)
```

```
6.082762530298219
```

Manhattan distance is calculated as the sum of the absolute differences between the two vectors.

ManhattanDistance = sum for i to N sum |v1[i] – v2[i]| The Manhattan distance is related to the L1 vector norm and the sum absolute error and mean absolute error metric.

We can demonstrate this with an example of calculating the Manhattan distance between two integer vectors, listed below.

```
# calculating manhattan distance between vectors
from math import sqrt

# calculate manhattan distance
def manhattan_distance(a, b):
  return sum(abs(e1-e2) for e1, e2 in zip(a,b))

# define data
row1 = [10, 20, 15, 10, 5]
row2 = [12, 24, 18, 8, 7]
# calculate distance
dist = manhattan_distance(row1, row2)
print(dist)
```

```
13
```

We can also perform the same calculation using the cityblock() function from SciPy.

```
# calculating manhattan distance between vectors
from scipy.spatial.distance import cityblock
# define data
row1 = [10, 20, 15, 10, 5]
row2 = [12, 24, 18, 8, 7]
# calculate distance
dist = cityblock(row1, row2)
print(dist)
```

Minkowski Distance Minkowski distance calculates the distance between two real-valued vectors.

It is a generalization of the Euclidean and Manhattan distance measures and adds a parameter, called the “order” or “p“, that allows different distance measures to be calculated.

The Minkowski distance measure is calculated as follows:

EuclideanDistance = (sum for i to N (abs(v1[i] – v2[i]))^p)^(1/p) Where “p” is the order parameter.

When p is set to 1, the calculation is the same as the Manhattan distance. When p is set to 2, it is the same as the Euclidean distance.

p=1: Manhattan distance. p=2: Euclidean distance.

```
# calculating minkowski distance between vectors
from math import sqrt

# calculate minkowski distance
def minkowski_distance(a, b, p):
  return sum(abs(e1-e2)**p for e1, e2 in zip(a,b))**(1/p)

# define data
row1 = [10, 20, 15, 10, 5]
row2 = [12, 24, 18, 8, 7]
# calculate distance (p=1)
dist = minkowski_distance(row1, row2, 1)
print(dist)
# calculate distance (p=2)
dist = minkowski_distance(row1, row2, 2)
print(dist)
```

We can also perform the same calculation using the minkowski_distance() function from SciPy.

```
# calculating minkowski distance between vectors
from scipy.spatial import minkowski_distance
# define data
row1 = [10, 20, 15, 10, 5]
row2 = [12, 24, 18, 8, 7]
# calculate distance (p=1)
dist = minkowski_distance(row1, row2, 1)
print(dist)
# calculate distance (p=2)
dist = minkowski_distance(row1, row2, 2)
print(dist)
```

Cosine similarity metric finds the normalized dot product of the two attributes. By determining the cosine similarity, we will effectively trying to find cosine of the angle between the two objects. The cosine of 0° is 1, and it is less than 1 for any other angle. It is thus a judgement of orientation and not magnitude: two vectors with the same orientation have a cosine similarity of 1, two vectors at 90° have a similarity of 0, and two vectors diametrically opposed have a similarity of -1, independent of their magnitude. Cosine similarity is particularly used in positive space, where the outcome is neatly bounded in [0,1]. One of the reasons for the popularity of cosine similarity is that it is very efficient to evaluate, especially for sparse vectors.

```
from math import*

def square_rooted(x):

   return round(sqrt(sum([a*a for a in x])),3)

def cosine_similarity(x,y):

 numerator = sum(a*b for a,b in zip(x,y))
 denominator = square_rooted(x)*square_rooted(y)
 return round(numerator/float(denominator),3)

print (cosine_similarity([3, 45, 7, 2], [2, 54, 13, 15]))
```

```
0.972
```

Jaccard Similarity

We use Jaccard Similarity to find similarities between sets. So first, let's learn the very basics of sets.

Sets:

A set is (unordered) collection of objects {a,b,c}. we use the notation as elements separated by commas inside curly brackets { }. They are unordered so {a,b} = {b,a}.

Cardinality:

The Cardinality of A (denoted by |A|) counts how many elements are in A.

Intersection:

Intersection between two sets A and B is denoted A ∩ B and reveals all items which are in both sets A,B.

Union:

Union between two sets A and B is denoted A ∪ B and reveals all items which are in either set. The **Jaccard similarity** measures **similarity between finite sample sets**, and is defined as the cardinality of the intersection of sets divided by the cardinality of the union of the sample sets. Suppose you want to find jaccard similarity between two sets A and B it is the ratio of cardinality of A ∩ B and A ∪ B.

```
from math import*

def jaccard_similarity(x,y):

 intersection_cardinality = len(set.intersection(*[set(x), set(y)]))
 union_cardinality = len(set.union(*[set(x), set(y)]))
 return intersection_cardinality/float(union_cardinality)

print (jaccard_similarity([0,1,2,5,6],[0,2,3,5,7,9]))
```

```
0.375
```

Contingency Table is one of the techniques for exploring two or even more variables. It is basically a tally of counts between two or more categorical variables.

```
#Loading libraries
import numpy as np
import pandas as pd
import matplotlib as plt

```

```
#Loading data
data = pd.read_csv("loan_status.csv")

print (data.head(10))

```

```
  grade sub_grade  loan_status             purpose
0     B        B2   Fully Paid         credit_card
1     C        C4  Charged Off                 car
2     C        C5   Fully Paid      small_business
3     C        C1   Fully Paid               other
4     B        B5   Fully Paid               other
5     A        A4   Fully Paid             wedding
6     C        C5   Fully Paid  debt_consolidation
7     E        E1   Fully Paid                 car
8     F        F2  Charged Off      small_business
9     B        B5  Charged Off               other
```

Describe Data

```
#Describe Data
data.describe()
```

	grade	sub_grade	loan_status	purpose
count	50	50	50	50
unique	6	19	2	10
top	B	B3	Fully Paid	debt_consolidation
freq	21	6	39	22

Data Info

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50 entries, 0 to 49
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   grade        50 non-null     object
 1   sub_grade    50 non-null     object
```

Data Types

```
memory usage: 1.7+ KB
```

```
# data types of feature/attributes
# in the data
data.dtypes

```

```
grade          object
sub_grade      object
loan_status    object
purpose        object
dtype: object
```

** Contingency Table showing correlation between Grades and loan status.**

```
data_crosstab = pd.crosstab(data['grade'],
              data['loan_status'],
              margins = False)
print(data_crosstab)

```

```
loan_status  Charged Off  Fully Paid
grade                               
A                      1          11
B                      5          16
C                      3           8
D                      1           3
E                      0           1
F                      1           0
```

Contingency Table showing correlation between Purpose and loan status.

```
data_crosstab = pd.crosstab(data['purpose'],
              data['loan_status'],
                margins = False)
print(data_crosstab)

```

```
loan_status         Charged Off  Fully Paid
purpose
car                           1           1
credit_card                   0           8
debt_consolidation            4          18
```

Contingency Table showing correlation between Grades+Purpose and loan status.

```
medical                       0           1
```

```
1
2 data_crosstab = pd.crosstab([data.grade, data.purpose],
3                             data.loan_status, margins = False)
4 print(data_crosstab)
```

```
loan_status               Charged Off  Fully Paid
grade purpose
A     credit_card                   0           1
      debt_consolidation            1           7
      major_purchase                0           1
      other                         0           1
      wedding                       0           1
B     credit_card                   0           6
      debt_consolidation            1           5
      major_purchase                1           0
      medical                       0           1
      moving                        0           1
      other                         3           2
      small_business                0           1
C     car                           1           0
      credit_card                   0           1
      debt_consolidation            2           4
      home_improvement              0           1
      other                         0           1
      small_business                0           1
D     debt_consolidation            0           2
      other                         1           1
E     car                           0           1
F     small_business                1           0
```