Unit II Machine Leaning for Text Data

Recommendation System for Movies- Euclidean distances, Manhattan distance, Cosine distances, Cosine similarity, Unsupervised Machine Learning for Grouping Similar Text- K-means and Hierarchical Clustering, Supervised Machine Learning for Classification Problems - Naïve Bayes and Support Vector Machines

Euclidean distances

In K-means clustering, elbow method and silhouette analysis or score techniques are used to find the number of clusters in a dataset.

The elbow method is used to find the “elbow” point, where adding additional data samples does not change cluster membership much.

Silhouette score determines whether there are large gaps between each sample and all other samples within the same cluster or across different clusters.

```
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
#
# Load IRIS dataset
#
iris = datasets.load_iris()
X = iris.data
y = iris.target
#
# Instantiate the KMeans models
#
km = KMeans(n_clusters=3, init='k-means++', max_iter=300, n_init=10,
#
# Fit the KMeans model
#
km.fit_predict(X)
#
# Calculate Silhoutte Score
#
score = silhouette_score(X, km.labels_, metric='euclidean')
#
```

```
# Print the score
#
print('Silhouetter Score: %.3f' % score)
```

```
Silhouetter Score: 0.553
```

Executing the above code predicts the Silhouette score of 0.553. This indicates that the clustering is reasonably good, since the Silhouette Score is closer to 1 than to -1 or 0.

A Silhouette Score of 0.553 suggests that the clusters are well separated and that the objects within each cluster are relatively homogeneous

Elbow Method

Introduction To Elbow Method

A fundamental step for any unsupervised algorithm is to determine the optimal number of clusters into which the data may be clustered.

Since we do not have any predefined number of clusters in unsupervised learning. We tend to use some method that can help us decide the best number of clusters.

In the case of K-Means clustering, we use Elbow Method for defining the best number of clustering

What Is the Elbow Method in K-Means Clustering

As we know in the k-means clustering algorithm we randomly initialize k clusters and we iteratively adjust these k clusters till these k-centroids riches in an equilibrium state.

However, the main thing we do before initializing these clusters is that determine how many clusters we have to use.

For determining K(numbers of clusters) we use Elbow method.

Elbow Method is a technique that we use to determine the number of centroids(k) to use in a k-means clustering algorithm.

In this method to determine the k-value we continuously iterate for k=1 to k=n (Here n is the hyperparameter that we choose as per our requirement). For every value of k, we calculate the within-cluster sum of squares (WCSS) value.

Double-click (or enter) to edit

WCSS - It is defined as the sum of square distances between the centroids and each points.

Now For determining the best number of clusters(k) we plot a graph of k versus their WCSS value. Surprisingly the graph looks like an elbow (which we will see later). Also, When k=1 the WCSS has the highest value but with increasing k value WCSS value starts to decrease.

We choose that value of k from where the graph starts to look like a straight line.

Implementation of the Elbow Method Usking Sklearn in Python

We will see how to implement the elbow method in 4 steps. At first, we will create random dataset points, then we will apply k-means on this dataset and calculate wcss value for k between 1 to 4.

Step 1: Importing the required libraries

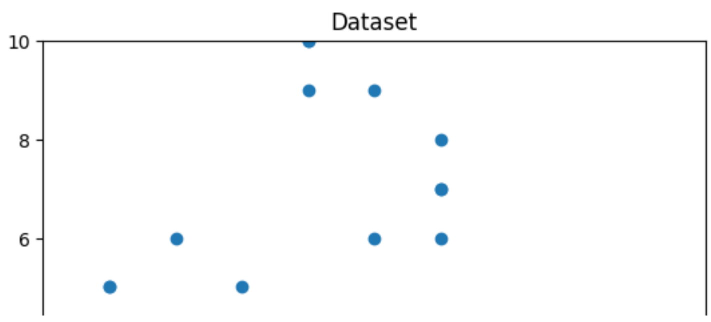
```
from sklearn.cluster import KMeans
from sklearn import metrics
from scipy.spatial.distance import cdist
import numpy as np
import matplotlib.pyplot as plt
```

Step 2: Creating and Visualizing the data

We will create a random array and visualize its distribution

```
# Creating the data
x1 = np.array([3, 1, 1, 2, 1, 6, 6, 6, 5, 6,\
               7, 8, 9, 8, 9, 9, 8, 4, 4, 5, 4])
x2 = np.array([5, 4, 5, 6, 5, 8, 6, 7, 6, 7, \
               1, 2, 1, 2, 3, 2, 3, 9, 10, 9, 10])
X = np.array(list(zip(x1, x2))).reshape(len(x1), 2)

# Visualizing the data
plt.plot()
plt.xlim([0, 10])
plt.ylim([0, 10])
plt.title('Dataset')
plt.scatter(x1, x2)
plt.show()
```



K Means Clustering Using the Elbow Method

Implementation of the Elbow Method

Sample Dataset

The dataset we are using here is the Mall Customers data (Download here).

It’s unlabeled data that contains the details of customers in a mall (features like genre, age, annual income(k$), and spending score). Our aim is to cluster the customers based on the relevant features of annual income and spending score.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sklearn
```

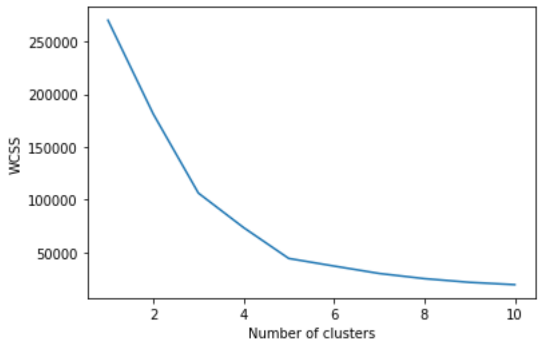
Now let’s import the given dataset and slice the important features.

```
dataset = pd.read_csv('Mall_Customers.csv')
X = dataset.iloc[:, [3, 4]].values
```

We have to find the optimal K value for clustering the data. Now we are using the Elbow Method to find the optimal K value.

```
from sklearn.cluster import KMeans
wcss = [] for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
```

The “init” argument is the method for initializing the centroid. We calculated the WCSS value for each K value. Now we have to plot the WCSS with the K value.



The point at which the elbow shape is created is 5; that is, our K value or an optimal number of clusters is 5. Now let’s train the model on the input data with a number of clusters 5.

```
kmeans = KMeans(n_clusters = 5, init = "k-means++", random_state = 4
y_kmeans = kmeans.fit_predict(X)
```

y_kmeans will be:

array([3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0,

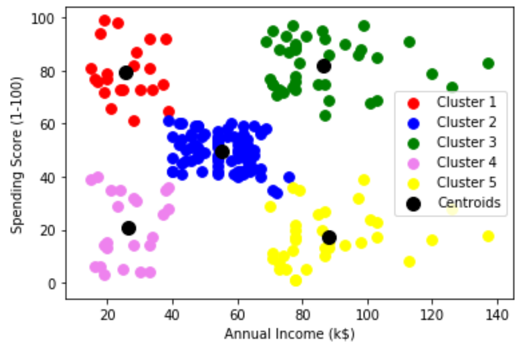
```
3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 1,
3, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 4, 2, 1, 2, 4, 2, 4, 2,
1, 2, 4, 2, 4, 2, 4, 2, 4, 2, 1, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2,
4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2,
4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2,
4, 2])
```

y_kmeans gives us different clusters corresponding to X. Now, let’s plot all the clusters using matplotlib.

```
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 60, c = 'r
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 60, c = 'b
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 60, c = 'g
plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 60, c = 'v
plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 60, c = 'y
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:
plt.xlabel('Annual Income (k$)') plt.ylabel('Spending Score (1-100)'

plt.show()
```

Graph:



Recommender System

We will be developing a movie recommender system based upon Collaborative-filtering to predict the name of the movie based upon the reviews of the other critics having similar taste.

The systesm uses two different methods for finding similairties between the critics known as Euclidean-Distance-Score and Pearson-Correlation-Score. The final reault for both the methods were almost similar.

After finding the similarity between critics, it uses the weighted average method to assign higher weight to the peer interest critics.

Finally, It normalizes the score by deviding it by the similarities of the critics who reviewed that movie.

Finding Similar DataPoint

Two ways for calculating similarity scores :

Euclidean Distance Score

Pearson Correlation Score

```
# Import Packages

from math import sqrt
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

# Getting more than one output Line
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# Getting the Dataset

movies= pd.read_csv("/content/drive/My Drive/Colab Notebooks/Dataset/
movies.head()

ratings=pd.read_csv("/content/drive/My Drive/Colab Notebooks/Dataset/
ratings.head()

tags= pd.read_csv('/content/drive/My Drive/Colab Notebooks/Dataset/ta
tags.head()
```

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

	userId	movieId	rating
0	1	1	4.0
1	1	3	4.0
2	1	6	4.0
3	1	47	5.0
4	1	50	5.0

	userId	movieId	tag	timestamp
0	2	60756	funny	1445714994
1	2	60756	Highly quotable	1445714996
2	2	60756	will ferrell	1445714992
3	2	89774	Boxing story	1445715207
4	2	89774	MMA	1445715200

Deleting unnecessary columns [Not using the timestamp column for our analysis in the meanwhile]

```
del tags['timestamp']
tags.head()
```

	userId	movieId	tag
0	2	60756	funny
1	2	60756	Highly quotable
2	2	60756	will ferrell
3	2	89774	Boxing story
4	2	89774	MMA

having a look at the columns of movies df

```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   movieId  9742 non-null   int64 
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
dtypes: int64(1), object(2)
memory usage: 228.5+ KB
```

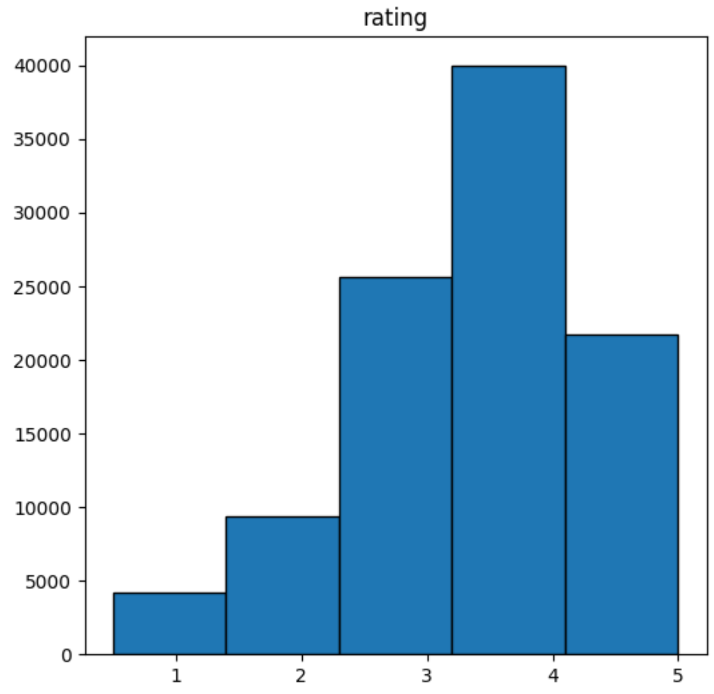
having a look at the distribution of ratings in ratings df

```
ratings['rating'].describe(include='all')
```

```
count    100836.000000
mean          3.501557
std           1.042529
min           0.500000
25%           3.000000
50%           3.500000
75%           4.000000
max           5.000000
Name: rating, dtype: float64
```

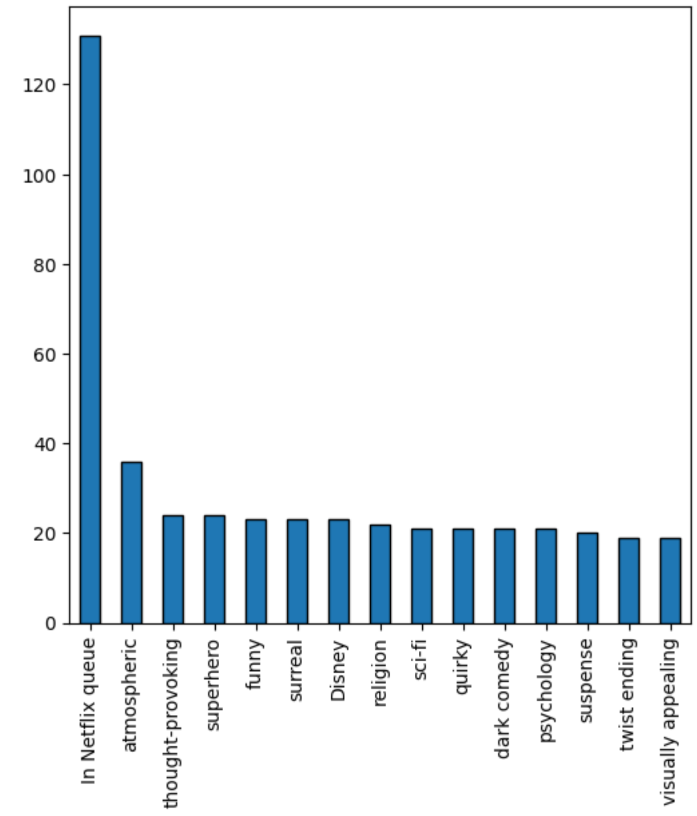
```
ratings.groupby('rating')['movieId'].nunique()
%matplotlib inline
ratings.hist(column='rating', figsize=(6,6), bins=5, grid=False, edg
```

```
rating
0.5    1066
1.0    1726
1.5    1386
2.0    3339
2.5    2925
3.0    4986
3.5    4216
4.0    5109
4.5    2710
5.0    2954
Name: movieId, dtype: int64array([[<Axes: title={'center': 'rating'}>]], dtype=object)
```



```
tag_counts = tags['tag'].value_counts()
tag_counts[:15]
tag_counts[:15].plot(kind='bar' , figsize=(6,6), edgecolor='black')
```

```
In Netflix queue      131
atmospheric            36
thought-provoking      24
superhero              24
funny                  23
surreal                23
Disney                 23
religion               22
sci-fi                 21
quirky                 21
dark comedy            21
psychology             21
suspense               20
twist ending           19
visually appealing     19
Name: tag, dtype: int64<Axes: >
```



```
movies['movieId'].count()
```

```
9742
```

```
## Removing movies with no genre
genre_filter= (movies['genres'] == '(no genres listed)')

movies=movies[~genre_filter]

## Because removing filtered rows does not reindex the dataframe, so
movies=movies.reset_index(drop=True)

# Checking total genres present in DataSet

genres_count= {}
```

```
for row in range(movies['movieId'].count()):
    for genre in movies['genres'][row].split("|"):
        if(genre != ''):
            genres_count[genre]= genres_count.get(genre,0)+1
```

```
genres_count
```

```
{'Adventure': 1263,
 'Animation': 611,
 'Children': 664,
 'Comedy': 3756,
 'Fantasy': 779,
 'Romance': 1596,
 'Drama': 4361,
 'Action': 1828,
 'Crime': 1199,
 'Thriller': 1894,
 'Horror': 978,
 'Mystery': 573,
 'Sci-Fi': 980,
 'War': 382,
 'Musical': 334,
 'Documentary': 440,
 'IMAX': 158,
 'Western': 167,
 'Film-Noir': 87}
```

```
fig, ax = plt.subplots(figsize=(15,10))
plt.barh(range(len(genres_count)), list(genres_count.values()))
plt.yticks(range(len(genres_count)),list(genres_count.keys()))
plt.xlabel('Movie Count')
plt.title("Genre Popularty")
for i, v in enumerate(genres_count.values()):
    ax.text(v + 20, i + .10, v)
```

```
<BarContainer object of 19 artists>([<matplotlib.axis.YTick at 0x7963d74d2920>,
  <matplotlib.axis.YTick at 0x7963d74d22c0>,
  <matplotlib.axis.YTick at 0x7963d74d1300>,
  <matplotlib.axis.YTick at 0x7963d7344370>,
  <matplotlib.axis.YTick at 0x7963d7344d60>,
  <matplotlib.axis.YTick at 0x7963d7345810>,
  <matplotlib.axis.YTick at 0x7963d73462c0>,
  <matplotlib.axis.YTick at 0x7963d7481e10>,
  <matplotlib.axis.YTick at 0x7963d74826e0>,
  <matplotlib.axis.YTick at 0x7963d74824a0>,
  <matplotlib.axis.YTick at 0x7963d7481ab0>,
  <matplotlib.axis.YTick at 0x7963d7482e90>,
  <matplotlib.axis.YTick at 0x7963d7481780>,
  <matplotlib.axis.YTick at 0x7963d8137eb0>,
  <matplotlib.axis.YTick at 0x7963db07ac50>,
```

Observations:

There are high number of movies from genre Drama & Comedy. So, they might create abias toward the movies which are from these genres.

Film-noir & IMAX are the least popular category for films

Euclidean Distance Score

Euclidean Distance is the square root of the sum of squared differences between corresponding elements of the two vectors.

Euclidean distance is only appropriate for data measured on the same scale.

Distance = 1/(1+sqrt of sum of squares between two points)

Value varies between 0 to 1, where closeness to 1 implies higher similarity.

```
# Defining a function to calculate the Euclidean Distance between tw

def euclidean_distance(person1,person2):
    #Getting details of person1 and person2
    df_first= ratings.loc[ratings['userId']==person1]
    df_second= ratings.loc[ratings.userId==person2]

    #Finding Similar Movies for person1 & person2
    df= pd.merge(df_first,df_second,how='inner',on='movieId')

    #If no similar movie found, return 0 (No Similarity)
    if(len(df)==0): return 0

    #sum of squared difference between ratings
    sum_of_squares=sum(pow((df['rating_x']-df['rating_y']),2))
    return 1/(1+sum_of_squares)

# Check whether this function works by passing similar ID, the Corer
euclidean_distance(3,3)
```

```
1.0
```

Pearson Correlation Score

Correlation between sets of data is a measure of how well they are related. It shows the linear relationship between two sets of data. In simple terms, it answers the question, Can I draw a line graph to represent the data?

Value varies between -1 to 1.[0-> Not related ; -1 -> perfect negatively corelated ; 1-> perfect positively corelated]

Slightly better than Euclidean because it addresses the the situation where the data isn't normalised. Like a User is giving high movie ratings in comparison to AVERAGE user.

```
# Defining a function to calculate the Pearson Correlation Score betw

def pearson_score(person1,person2):

    #Get detail for Person1 and Person2
    df_first= ratings.loc[ratings.userId==person1]
    df_second= ratings.loc[ratings.userId==person2]

    # Getting mutually rated items
    df= pd.merge(df_first,df_second,how='inner',on='movieId')

    # If no rating in common
    n=len(df)
    if n==0: return 0

    #Adding up all the ratings
    sum1=sum(df['rating_x'])
    sum2=sum(df['rating_y'])

    ##Summing up squares of ratings
    sum1_square= sum(pow(df['rating_x'],2))
    sum2_square= sum(pow(df['rating_y'],2))

    # sum of products
    product_sum= sum(df['rating_x']*df['rating_y'])

    ## Calculating Pearson Score
    numerator= product_sum - (sum1*sum2/n)
    denominator=sqrt((sum1_square- pow(sum1,2)/n) * (sum2_square - po
    if denominator==0: return 0

    r=numerator/denominator
    return r

#Checking function by passing similar ID, Output should be 1
pearson_score(1,1)
```

```
1.0
```

Getting the results based on Pearson Score

```
# Returns the best matches for person from the prefs dictionary.
# Number of results and similarity function are optional params.
def topMatches(personId,n=5,similarity=pearson_score):
    scores=[(similarity(personId,other),other) for other in ratings.
    # Sort the list so the highest scores appear at the top
    scores.sort( )
    scores.reverse( )
    return scores[0:n]

topMatches(1,n=3) ## Getting 3 most similar Users for Example
```

```
[(1.000000000000016, 550), (1.000000000000016, 550), (1.000000000000016, 550)]
```

Getting Recommendations

```
# Gets recommendations for a person by using a weighted average of e
# Defining a function to get the recommendations

def getRecommendation(personId, similarity=pearson_score):
    '''
    totals: Dictionary containing sum of product of Movie Ratings by
    simSums: Dictionary containung sum of weights for all the users
    '''
    totals,simSums= {},{}

    df_person= ratings.loc[ratings.userId==personId]

    for otherId in ratings.loc[ratings['userId']!=personId]['userId'

        # Getting Similarity with OtherID
        sim=similarity(personId,otherId)

        # Ignores Score of Zero or Negatie correlation
        if sim<=0: continue

        df_other=ratings.loc[ratings.userId==otherId]

        #Movies not seen by the personID
        movie=df_other[~df_other.isin(df_person).all(1)]
        for movieid,rating in (np.array(movie[['movieId','rating']])
            #similarity* Score
            totals.setdefault(movieid,0)
            totals[movieid]+=rating*sim

            #Sum of Similarities
            simSums.setdefault(movieid,0)
            simSums[movieid]+=sim




        # Creating Normalized List
        ranking=[(t/simSums[item],item) for item,t in totals.items()

        # return the sorted List
        ranking.sort()
        ranking.reverse()
        recommendedId=np.array([x[1] for x in ranking])


        return list(np.array(movies[movies['movieId'].isin(recommende

### Example Recommendation
# Returns 20 recommended movie for the given UserID
# userId can be ranged from 1 to 671

# taking the User_Id as input and recommending movies for the user

user_id = int(input("Enter Your UserId: "))

recommended_movies = getRecommendation(user_id)
print("____________________________________________________")
print("\n Recommended Movies: User {}".format(user_id))
print("____________________________________________________")
print(*recommended_movies, sep='\n')
print("____________________________________________________")
```

```
Enter Your UserId: 143
____________________________________________________

 Recommended Movies: User 143
____________________________________________________
Shawshank Redemption, The (1994)
Tommy Boy (1995)
Good Will Hunting (1997)
Gladiator (2000)
Kill Bill: Vol. 1 (2003)
Collateral (2004)
Talladega Nights: The Ballad of Ricky Bobby (2006)
Departed, The (2006)
Dark Knight, The (2008)
Step Brothers (2008)
Inglourious Basterds (2009)
Zombieland (2009)
```

```
Shutter Island (2010)
Exit Through the Gift Shop (2010)
Inception (2010)
Town, The (2010)
Inside Job (2010)
Louis C.K.: Hilarious (2010)
Warrior (2011)
Dark Knight Rises, The (2012)
____________________________________________________
```

Euclidean-Distance

Euclidean Distance

Euclidean Distance represents the shortest distance between two vectors.

It is the square root of the sum of squares of differences between corresponding elements.

```
# importing the library
from scipy.spatial import distance

# defining the points
point_1 = (1, 2, 3)
point_2 = (4, 5, 6)
point_1, point_2

# computing the euclidean distance
euclidean_distance = distance.euclidean(point_1, point_2)
print('Euclidean Distance b/w', point_1, 'and', point_2, 'is: ', euc
```

```
Euclidean Distance b/w (1, 2, 3) and (4, 5, 6) is:  5.196152422706632
```

Manhattan Distance

Manhattan Distance is the sum of absolute differences between points across all the dimensions.

Formula for Manhattan Distance

Since the above representation is 2 dimensional, to calculate Manhattan Distance,

we will take the sum of absolute distances in both the x and y directions. So, the Manhattan distance in a 2-dimensional space is given as:

```
# computing the manhattan distance
manhattan_distance = distance.cityblock(point_1, point_2)
print('Manhattan Distance b/w', point_1, 'and', point_2, 'is: ', manh
```

```
Manhattan Distance b/w (1, 2, 3) and (4, 5, 6) is:  9
```

Minkowski Distance

Minkowski Distance is the generalized form of Euclidean and Manhattan Distance.

```
# computing the minkowski distance
minkowski_distance = distance.minkowski(point_1, point_2, p=3)
```

```
print('Minkowski Distance b/w', point_1, 'and', point_2, 'is: ', minl
```

```
Minkowski Distance b/w (1, 2, 3) and (4, 5, 6) is:  4.3267487109222245
```

The p parameter of the Minkowski Distance metric of SciPy represents the order of the norm. When the order(p) is 1, it will represent Manhattan Distance and when the order in the above formula is 2, it will represent Euclidean Distance.

```
# minkowski and manhattan distance
minkowski_distance_order_1 = distance.minkowski(point_1, point_2, p=
print('Minkowski Distance of order 1:',minkowski_distance_order_1, '
```

```
Minkowski Distance of order 1: 9.0
Manhattan Distance:  9
```

When the order is 2, we can see that Minkowski and Euclidean distances are the same.

So far, we have covered the distance metrics that are used when we are dealing with continuous or numerical variables.

But what if we have categorical variables? How can we decide the similarity between categorical variables? This is where we can make use of another distance metric called Hamming Distance.

Hamming Distance

Hamming Distance measures the similarity between two strings of the same length. The Hamming Distance between two strings of the same length is the number of positions at which the corresponding characters are different.

Let’s understand the concept using an example. Let’s say we have two strings:

“euclidean” and “manhattan”

Since the length of these strings is equal, we can calculate the Hamming Distance. We will go character by character and match the strings. The first character of both the strings (e and m, respectively) is different. Similarly, the second character of both the strings (u and a) is different. and so on.

Look carefully – seven characters are different, whereas two characters (the last two characters) are similar:

euclidean and manhattan distances

Hence, the Hamming Distance here will be 7.

Note that the larger the Hamming Distance between two strings, the more dissimilar those strings will be (and vice versa).

These are the two strings “euclidean” and “manhattan”, which we have seen in the example as well. Let’s now calculate the Hamming distance between these two strings:

```
# defining two strings
string_1 = 'euclidean'
string_2 = 'manhattan'

# computing the hamming distance
hamming_distance = distance.hamming(list(string_1), list(string_2))*
print('Hamming Distance b/w', string_1, 'and', string_2, 'is: ', ham
```

```
Hamming Distance b/w euclidean and manhattan is:  7.0
```

As we saw in the example above, the Hamming Distance between “euclidean” and “manhattan” is 7. We also saw that Hamming Distance only works when we have strings of the same length.

Let’s see what happens when we have strings of different lengths

```
# strings of different shapes
new_string_1 = 'data'
new_string_2 = 'science'
len(new_string_1), len(new_string_2)
```

```
(4, 7)
```

You can see that the lengths of both the strings are different. Let’s see what will happen when we try to calculate the Hamming Distance between these two strings:

```
# computing the hamming distance
hamming_distance = distance.hamming(list(new_string_1), list(new_str
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-51d01805d328> in <cell line: 2>()
      1 # computing the hamming distance
----> 2 hamming_distance = distance.hamming(list(new_string_1), list(new_string_2))

/usr/local/lib/python3.10/dist-packages/scipy/spatial/distance.py in hamming(u, v, w)
    718     v = _validate_vector(v)
    719     if u.shape != v.shape:
--> 720         raise ValueError('The 1d arrays must have equal lengths.')
    721     u_ne_v = u != v
    722     if w is not None:

ValueError: The 1d arrays must have equal lengths.
```

SEARCH STACK OVERFLOW

This throws an error saying that the lengths of the arrays must be the same. Hence, Hamming distance only works when we have strings or arrays of the same length.

These are some of the most commonly used similarity measures or distance matrices in Machine Learning.

Cosine similarity is a metric, helpful in determining, how similar the data objects are irrespective of their size. We can measure the similarity between two sentences in Python using Cosine Similarity. In cosine similarity, data objects in a dataset are treated as a vector. The formula to find the cosine similarity between two vectors is –

S_C(x, y) = x . y / ||x|| \times ||y||

where,

x . y = product (dot) of the vectors ‘x’ and ‘y’.

||x|| and ||y|| = length (magnitude) of the two vectors ‘x’ and ‘y’.

||x|| \times ||y|| = regular product of the two vectors ‘x’ and ‘y’.

Example : Consider an example to find the similarity between two vectors – ‘x’ and ‘y’, using Cosine Similarity.

The ‘x’ vector has values, x = { 3, 2, 0, 5 } The ‘y’ vector has values, y = { 1, 0, 0, 0 } The formula for calculating the cosine similarity is : S_C(x, y) = x . y / ||x|| \times ||y||

x . y = 3*1 + 2*0 + 0*0 + 5*0* = 3

||x|| = √ (3)^2 + (2)^2 + (0)^2 + (5)^2 = 6.16

||y|| = √ (1)^2 + (0)^2 + (0)^2 + (0)^2 = 1

∴ S_C(x, y) = 3 / (6.16 * 1) = 0.49

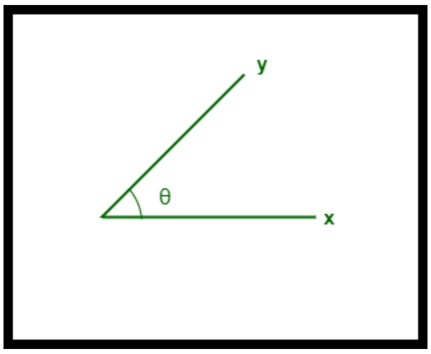
The dissimilarity between the two vectors ‘x’ and ‘y’ is given by –

∴ D_C(x, y) = 1 - S_C(x, y) = 1 - 0.49 = 0.51

The cosine similarity between two vectors is measured in ‘θ’.

If θ = 0°, the ‘x’ and ‘y’ vectors overlap, thus proving they are similar.

If θ = 90°, the ‘x’ and ‘y’ vectors are dissimilar.



Advantages :

The cosine similarity is beneficial because even if the two similar data objects are far apart by the Euclidean distance because of the size, they could still have a smaller angle between them. Smaller the angle, higher the similarity.

When plotted on a multi-dimensional space, the cosine similarity captures the orientation (the angle) of the data objects and not the magnitude.

```
doc_1 = "Data is the oil of the digital economy"
doc_2 = "Data is a new oil"

# Vector representation of the document
doc_1_vector = [1, 1, 1, 1, 0, 1, 1, 2]
doc_2_vector = [1, 0, 0, 1, 1, 0, 1, 0]
data = [doc_1, doc_2]
```

```
 from sklearn.feature_extraction.text import CountVectorizer
```

```
count_vectorizer = CountVectorizer()
vector_matrix = count_vectorizer.fit_transform(data)
vector_matrix
```

```
<2x8 sparse matrix of type '<class 'numpy.int64'>'
        with 11 stored elements in Compressed Sparse Row format>
```

```
tokens = count_vectorizer.get_feature_names_out()
tokens
```

```
array(['data', 'digital', 'economy', 'is', 'new', 'of', 'oil', 'the'],
      dtype=object)
```

```
vector_matrix.toarray()
```

```
array([[1, 1, 1, 1, 0, 1, 1, 2],
       [1, 0, 0, 1, 1, 0, 1, 0]])
```

```
import pandas as pd

def create_dataframe(matrix, tokens):

    doc_names = [f'doc_{i+1}' for i, _ in enumerate(matrix)]
    df = pd.DataFrame(data=matrix, index=doc_names, columns=tokens)
    return(df)
```

```
create_dataframe(vector_matrix.toarray(),tokens)
```

	data	digital	economy	is	new	of	oil	the
doc_1	1	1	1	1	0	1	1	2
doc_2	1	0	0	1	1	0	1	0

```
from sklearn.metrics.pairwise import cosine_similarity
```

```
cosine_similarity_matrix = cosine_similarity(vector_matrix)
create_dataframe(cosine_similarity_matrix,['doc_1','doc_2'])
```

	doc_1	doc_2
doc_1	1.000000	0.474342
doc_2	0.474342	1.000000

by observing the above table, we can say that the Cosine Similarity between doc_1 and doc_2 is 0.47

Let’s check the cosine similarity with TfidfVectorizer, and see how it change over CountVectorizer.

```
from sklearn.feature_extraction.text import TfidfVectorizer

Tfidf_vect = TfidfVectorizer()
vector_matrix = Tfidf_vect.fit_transform(data)

tokens = Tfidf_vect.get_feature_names_out()
create_dataframe(vector_matrix.toarray(),tokens)
```

	data	digital	economy	is	new	of	oil	the
doc_1	0.243777	0.34262	0.34262	0.243777	0.000000	0.34262	0.243777	0.68524
doc_2	0.448321	0.00000	0.00000	0.448321	0.630099	0.00000	0.448321	0.00000

```
cosine_similarity_matrix = cosine_similarity(vector_matrix)
create_dataframe(cosine_similarity_matrix,['doc_1','doc_2'])
```

	doc_1	doc_2
doc_1	1.000000	0.327871
doc_2	0.327871	1.000000

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

```
from sklearn.metrics.pairwise import cosine_similarity, cosine_dista
```

```
cosine_similarity([[3,1]],[[6,2]])
```

```
array([[1.]])
```

```
cosine_similarity([[3,0]],[[0,8]])
```

```
array([[0.]])
```

```
cosine_similarity([[3,1]],[[3,2]])
```

```
array([[0.96476382]])
```

```
doc1 = """
iphone sales contributed to 70% of revenue. iphone demand is increasi
the main competitor phone galaxy recorded 5% less growth compared to 
"""

doc2 = """
The upside pressure on volumes for the iPhone 12 series, historical 
in the July-September time period heading into launch event, and fur
to outperformance for iPhone 13 volumes relative to lowered investor 
very attractive set up for the shares.
"""

doc3 = """
samsung's flagship product galaxy is able to penetrate more into asia
iphone. galaxy is redesigned with new look that appeals young demogra
are coming from galaxy phone sales
"""

doc4 = """
Samsung Electronics unveils its Galaxy S21 flagship, with modest spe
and a significantly lower price point. Galaxy S21 price is lower by 
which highlights Samsung's focus on boosting shipments and regaining 
"""
```

```
import pandas as pd
```

```
df = pd.DataFrame([
        {'iPhone': 3,'galaxy': 1},
        {'iPhone': 2,'galaxy': 0},
        {'iPhone': 1,'galaxy': 3},
        {'iPhone': 1,'galaxy': 2},
    ],
    index=[
        "doc1",
        "doc2",
        "doc3",
        "doc4"
    ])
```

```
df
```

	iPhone	galaxy
doc1	3	1

```
df.loc["doc1":"doc1"]
```

	iPhone	galaxy
doc1	3	1

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-8a721012f8e7> in <cell line: 1>()
----> 1 cosine_similarity(df.loc["doc1"],df.loc["doc2"])

                                    2 frames
/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py in check_array(array, accept_sparse, accept_large_sparse,
dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
    900             # If input is 1D raise error
    901             if array.ndim == 1:
--> 902                 raise ValueError(
    903                     "Expected 2D array, got 1D array instead:\narray={}.\n"
    904                     "Reshape your data either using array.reshape(-1, 1) if "

ValueError: Expected 2D array, got 1D array instead:
array=[3. 1.].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a
single sample.
```

SEARCH STACK OVERFLOW

```
cosine_similarity(df.loc["doc1":"doc1"],df.loc["doc2":"doc2"])
```

```
array([[0.9486833]])
```

```
cosine_similarity(df.loc["doc1":"doc1"],df.loc["doc3":"doc3"])
```

```
array([[0.6]])
```

```
cosine_similarity(df.loc["doc3":"doc3"],df.loc["doc4":"doc4"])
```

```
array([[0.98994949]])
```

```
cosine_similarity(df.loc["doc1":"doc1"],df.loc["doc4":"doc4"])
```

```
array([[0.70710678]])
```

```
cosine_distances(df.loc["doc1":"doc1"],df.loc["doc4":"doc4"])
```

```
array([[0.29289322]])
```

```
1-0.70710678
```

```
0.29289321999999995
```

Solve The Below Questions

doc1 = "I am pursuing MSC from school of data science"

doc2 = "right now school of data science is running these course"

Find the Following Distance matrix

Cosine Similarity

Cosine Distance

Eculidean Distance

Manthan Distance

find Cosine Similarity

doc_1 = "Text Processing is must for preprocesing "

doc_2 = "Text Processing is must for preprocesing"

✓ 0s completed at 8:27 AM