DWM CODE

Cleaning/Transformation/Normalization/Discretization/Visualization:

```
import pandas as pd;
data=pd.read_csv('sales.csv').head(10)
print(data)
des=data.describe()
print(des)
```

###cleaning###

```
#missing and duplicate value
data=data.dropna();
data=data.drop_duplicates();

# Removing Outliers
z_scores = (data[['Units Sold', 'Total Profit']] - data[['Units Sold','Total Profit']].mean()) /
data[['Units Sold', 'Total Profit']].std()
data = data[(z_scores.abs() < 3).all(axis=1)]

# Reset index
data = data.reset_index(drop=True)
data.to_csv('cleaned_sales.csv',index=False);
```

 # Transformation #

```
df=data.copy()

#strip
for col in df:
    if df[col].dtype==str:
        df[col]=df[col].str.strip()

#to_datatime
df['Order Date']=pd.to_datetime(df['Order Date'],errors='coerce')
df['Ship Date']=pd.to_datetime(df['Ship Date'],errors='coerce')

#Fixing Total values
df['Total Revenue']=df['Units Sold']*df['Unit Price']
df['Total Cost']=df['Unit Price']*df['Unit Cost']
df['Total Profit'] = df['Total Revenue'] - df['Total Cost']
```

```python
# Extract date components
df['Order Year'] = df['Order Date'].dt.year
df['Order Month'] = df['Order Date'].dt.month
df['Order Day'] = df['Order Date'].dt.day
df['Ship Year'] = df['Ship Date'].dt.year
df['Ship Month'] = df['Ship Date'].dt.month
df['Ship Day'] = df['Ship Date'].dt.day

# Calculate Profit Margin
df['Profit Margin'] = (df['Total Profit'] / df['Total Revenue']) * 100

df.to_csv('transform.csv',index=False)
print(df)
```

#Normalization

```python
def min_max_norm(col):
    return (col-col.min())/(col.max()-col.min())

def z_scores_norm(col):
    return (col-col.mean())/col.std()

def decimal_norm(col):
    return col/(col.max()*10)

numeric_cols = ['Total Revenue', 'Total Profit', 'Total Cost','Unit Cost','Unit Price','Units Sold']

min_max_norm_data=data.copy()
z_scores_norm_data=data.copy()
decimal_norm_data=data.copy()

for cols in numeric_cols:
    min_max_norm_data[cols]=min_max_norm(min_max_norm_data[cols])
    z_scores_norm_data[cols]=z_scores_norm(z_scores_norm_data[cols])
    decimal_norm_data[cols]=decimal_norm(decimal_norm_data[cols])

min_max_norm_data.to_csv('min_max_norm.csv',index=False)
z_scores_norm_data.to_csv('z_scores_norm.csv',index=False)
decimal_norm_data.to_csv('decimal_norm.csv',index=False)
```

#discretization

```
def discretization(cols,bins=4):
    bin_labels = [f'Bin {i+1}' for i in range(bins)]
    return pd.cut(cols,bins=bins,labels=bin_labels)

numeric_cols = ['Total Revenue', 'Total Profit', 'Total Cost','Unit Cost','Unit Price','Units Sold']

dis=data.copy()

for col in numeric_cols:
    dis[col]=discretization(dis[col])

dis.to_csv('descrtization.csv')

import matplotlib.pyplot as plt
import seaborn as sns

numeric_cols = ['Total Revenue', 'Total Profit','Total Cost', 'Unit Cost', 'Unit Price', 'Units Sold']
# plt.figure(figsize=(7,6))
plt.title('Before Normalization')
sns.boxplot(data[numeric_cols])
plt.show()
plt.title('min_max_norm boxplot')
sns.boxplot(min_max_norm_data[numeric_cols])
plt.show()
plt.title('z_scores_norm boxplot')
sns.boxplot(z_scores_norm_data[numeric_cols])
plt.show()
plt.title('decimal_norm boxplot')
sns.boxplot(decimal_norm_data[numeric_cols])
plt.show()

# #without seaborn
data[numeric_cols].boxplot()
plt.show()
min_max_norm_data[numeric_cols].boxplot()
plt.show()
z_scores_norm_data[numeric_cols].boxplot()
plt.show()
decimal_norm_data[numeric_cols].boxplot()
plt.show()

data[numeric_cols].hist(figsize=(6,6));
```

Naive Bayes:

```
import pandas as pd

data = pd.read_csv('weather.csv')

# Precomputing the all frequency
classification = input('Enter the class Name:')
info = {}
prior = data[classification].value_counts()
row = 0
for val in data[classification]:
    if val not in info:
        info[val] = {}
    for e in data.iloc[row]:
        if e == val:
            continue
        if e in info[val]:
            info[val][e] += 1
        else:
            info[val][e] = 1
    row += 1

# Naive Bayes Formula
def Naive_Bayes(l):
    prob, sum, class_name = 0, 0, ''
    for p in prior.keys():
        curr = prior[p]/data.shape[0]
        for v in l:
            curr = curr*((info[p].get(v, 0))/prior[p])
        sum += curr
        if prob < curr:
            prob, class_name = curr, p
    return [class_name, (prob*100)/sum, prob]


# For all rows prediction
prediction, probability = [], []
accuracy = 0
for i in range(len(data)):
    row = []
    for k in data.iloc[i]:
        row.append(k)
    row.pop(len(row)-1)
    res = Naive_Bayes(row)
```

```
    prediction.append(res[0])
    probability.append(res[1])
    if res[0] == data.iloc[i][len(row)]:
        accuracy += 1

data['Prediction'] = prediction
data['Probability'] = probability

print("\n", data)
print(f'\nAccuracy: {((accuracy*100)/len(data)):.3f}%')
```

K Means code:

```
import math
import matplotlib.pyplot as plt


def oneDim():
    data = [int(i) for i in input('Enter your data:').split()]
    n = int(input('Enter the number of clusters you want:'))
    data.sort()
    center = data[0:n]

    while True:
        cluster = [[] for _ in range(n)]

        for e in data:
            dis = float('inf')
            cls = 0
            for i in range(n):
                if abs(e - center[i]) < dis:
                    dis = abs(e - center[i])
                    cls = i
            cluster[cls].append(e)

        is_change = True
        for i in range(n):
            m = sum(cluster[i]) / len(cluster[i])
            if m != center[i]:
                is_change = False
            center[i] = m

        if is_change:
```

```
        print('Clusters after applying K-means algorithm:')
        for i in range(n):
            print(f'\nCluster {i+1}:', cluster[i])
            print(f'Centroid {i+1}:', center[i])

        # Plot the clusters
        colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k']
        for i in range(n):
            plt.scatter(cluster[i], [i] * len(cluster[i]),c=colors[i], label=f'Cluster {i+1}')
            plt.scatter(center[i], i, c=colors[i],marker='X', s=100, label=f'Centroid {i+1}')

        plt.xlabel('Data Values')
        plt.ylabel('Cluster')
        plt.legend()
        plt.show()
        break


def twoDim():
    cod = int(input('Enter the number of coordinates:'))
    data = []
    for _ in range(cod):
        data.append([float(i) for i in input(Enter the {_+1} coordinate:').split(' ')])
    n = int(input('Enter the number of clusters:'))
    center = data[0:n]

    while True:
        cluster = [[] for _ in range(n)]

        for e in data:
            dis = float('inf')
            cls = 0
            for i in range(n):
                distance = math.sqrt((e[0]-center[i][0])**2 + (e[1]-center[i][1])**2)
                if distance < dis:
                    dis = distance
                    cls = i
            cluster[cls].append(e)

        is_change = True
        for i in range(n):
            s1, s2 = 0.0, 0.0
            for j in cluster[i]:
                s1 += j[0]
```

```
            s2 += j[1]
         m1 = s1 / len(cluster[i])
         m2 = s2 / len(cluster[i])
         if m1 != center[i][0] or m2 != center[i][1]:
            is_change = False
         center[i] = [m1, m2]

      if is_change:
         print('Clusters after applying K-means algorithm:')
         for i in range(n):
            print(f'\nCluster {i+1}:', cluster[i])
            print(f'Centroid {i+1}:', center[i])

         # Plot the clusters
         colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k']
         for i in range(n):
            x, y = zip(*cluster[i])
            plt.scatter(x, y, c=colors[i], label=f'Cluster {i+1}')
            plt.scatter(center[i][0], center[i][1], c=colors[i], marker='X', s=100, label=f'Centroid
{i+1}')

         plt.xlabel('X-Axis')
         plt.ylabel('Y-Axis')
         plt.legend()
         plt.show()
         Break


if input('What type of data you have:\n1.One dimension\n2.Two dimension\nYour choice:') == '1':
   oneDim()
else:
   twoDim()
```

Hierarchical Clustering :

```
import pandas as pd
import math
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

data = pd.read_csv('./heirch.csv')
cod = data[['X', 'Y']].values

distance = [[math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2) for x1, y1 in cod] for x2, y2 in cod]
df = pd.DataFrame(distance)

clusters = {i: [i] for i in range(len(cod))}

def agglomerate(dist):
  min_distance = float('inf')
  min_indices = [0, 0]

  for i in range(len(dist)):
     for j in range(i):
        if min_distance > dist.iloc[i, j]:
           min_distance = dist.iloc[i, j]
           min_indices = [dist.columns[j], dist.index[i]]

  min_indices.sort()
  print(f'[P{clusters[min_indices[0]]}, P{clusters[min_indices[1]]}]')
  print("Minimum distance:", min_distance, "\n\n")
  dist = dist.drop(index=min_indices[1], columns=min_indices[1])

  for i in range(len(dist)):
     for j in range(len(dist)):
        curr = min([distance[m][n] for m in clusters[dist.columns[j]] for n in clusters[dist.index[i]]])
        dist.iloc[i, j] = curr

  clusters[min_indices[0]].extend(clusters[min_indices[1]])
  del clusters[min_indices[1]]

  return dist

while len(df) > 1:
  df = agglomerate(df)

Z = linkage(data[['X', 'Y']], method='single')
```

```
plt.figure(figsize=(10, 5))
dendrogram(Z, labels=[f'P{i + 1}' for i in range(len(data))])
plt.xlabel("Sample Index")
plt.ylabel("Distance")
plt.title("Dendrogram")
plt.show()
```

Apriori Algorithm :

```
class CombinationsGenerator:
    def __init__(self):
        self.result = []
        self.tmp = []

    def generate_combinations(self, max_number, k):
        self._generate_combinations_util(max_number, 0, k)

    def _generate_combinations_util(self, max_number, left, k):
        if k == 0:
            self.result.append(self.tmp[:])
            return

        for e in range(left, len(max_number)):
            self.tmp.append(max_number[e])
            self._generate_combinations_util(max_number, e + 1, k - 1)
            self.tmp.pop()


def print_table(table, freq):
    print('------------------')
    print("ItemSet\tFrequency")
    print('------------------')
    for index in range(len(freq)):
        print(f'{table[index]}\t==  {freq[index]}')


n = int(input('Enter the number of transactions:'))
dict = {}


for i in range(n):
    dict[i] = [int(_) for _ in input(f'Enter the items of transactions:{i+1}').split()]
```

```python
support = int(input('Enter the minimum support:'))


def print_association_rule(frequent_list, freq, confidence=0.75):
  for index in range(len(frequent_list)):
     all_groups = []
     for size in range(len(frequent_list[index]) - 1):
        c = CombinationsGenerator()
        c.generate_combinations(frequent_list[index], size+1)
        rules = c.result

        for group1 in rules:
           group2 = []
           for e in frequent_list[index]:
              if e not in group1:
                 group2.append(e)

           count = 0
           for k in range(n):
              if all(m in dict[k] for m in group2):
                 count += 1

           if freq[index]/count >= confidence:
              all_groups.append((group1, group2))

  print("\n\nFinal Association rule:")
  for it, (g1, g2) in enumerate(all_groups):
     print(f"{g1} ==> {g2}")


frequent_item_list = [k for k in set(j for i in dict.keys() for j in dict[i])]
item_size = 1

while True:
  print("\nIteration ", item_size, ":")
  combinations_generator = CombinationsGenerator()
  combinations_generator.generate_combinations(frequent_item_list, item_size)
  comb_item_set = combinations_generator.result

  current_frequent_item_list = []
  frequent = []
  for item_set in comb_item_set:
     cnt = 0
     for i in range(n):
```

```
        if all(item in dict[i] for item in item_set):
            cnt += 1

    if cnt >= support:
        current_frequent_item_list.append(item_set)
        frequent.append(cnt)

  print_table(current_frequent_item_list, frequent)
  frequent_item_list = [k for k in set(j for i in current_frequent_item_list for j in i)]
  if item_size+1 >= len(frequent_item_list):
    print_association_rule(current_frequent_item_list, frequent)
    break
  item_size += 1
```

Page Rank :

```
import string

n = int(input('Enter the total number of vertex:'))
print('Enter the transition matrix:')
matrix = [[] for _ in range(n)]

# input adjacency
for i in range(n):
  matrix[i] = [int(val) for val in input().split()]

# transition matrix
for i in range(n):
  if sum(matrix[i]) > 0:
    p = 1 / sum(matrix[i])
  else:
    for k in range(n):
      matrix[i][k] = 1 / n

  for j in range(n):
    if matrix[i][j] == 1:
      matrix[i][j] = p

# transpose
for i in range(n):
  for j in range(i):
    matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
```

```python
# teleport factor
dump_factor = eval(input('\nEnter the dumping factor:'))
r = [1/n] * n

# transition matrix with dumping factor
for i in range(n):
  for j in range(n):
     matrix[i][j] = (matrix[i][j] * dump_factor) + ((1 - dump_factor) / n)

# page rank algorithm
def page_rank():
  temp = r.copy()
  for row in range(n):
     curr = 0
     for col in range(n):
        curr += matrix[row][col] * temp[col]
     r[row] = curr

# max iteration
for _ in range(int(input('Enter the number of iterations:'))):
  page_rank()

ans = [[r[i], i] for i in range(n)]
ans.sort(key=lambda x: x[0], reverse=True)

print('Rank  WebPage == PageRankValue')
for i in range(n):
  print(f'{i+1}  :  {string.ascii_uppercase[ans[i][1]]} == {ans[i][0]}')
```