Big Data Mining

1st assignment, 24/5/2020
Vangelis Christou, p2821805

Introduction to the Groceries.csv dataset

Looking into the given dataset, we were happy to figure out that the data were in pretty good shape, overall. With a first look anyone would see that the data were divided in 4 different categories.

- **Numeric**
  Age
  Income
  Persons_in_Household
- **Nominal**
  Sex
  Marital_Status
  Occupation
- **Ordinal**
  Customer_Rating
  Education
- **Set**
  Groceries

and of course a primary key attribute: Customer_ID

Missing Values

First we checked for any missing values. Indeed there were a bunch of them in the form of space characters (' ') and luckily only on numeric attributes. What we did is we replaces ' ' values with NaN and them, as indicated, filled thos nas with the mean of the given attribute.

Code snippet excerpt:

```
df = df.replace(r'^\s*$', np.nan, regex=True)
df.fillna(df.mean().astype(int), inplace=True)
```

Transformations

We then moved on to transform the attribute to their actual types.

Especially for the ordinal attributes we created an extra column that maps the rank of the ordinal value to an integer. This would help us later to calculate the similarity matrices.

Code snippet excerpt:

```
# set proper data types
# numeric
df["Age"] = pd.to_numeric(df["Age"],downcast='integer')

# nominal
df["Sex"] = df["Sex"].astype('category')

# ordinal
# for the ordinal columns we will replace with an enumeration to calculate similarity
# as numeric

cRatings = df["Customer_Rating"] = pd.Categorical(df["Customer_Rating"],
            categories=["poor", "fair", "good", "very_good", "excellent"], ordered=True)

cRatings = cRatings.replace("poor",1).replace("fair",2).replace("very_good",4)
            .replace("good",3).replace("excellent",5)
df["Customer_Rating_enum"] = cRatings.astype(np.int)
```

Measuring Similarity

Now that our data are clean and well defined we were ready to move on with the customer similarity calculation. The steps we would do would in the first iteration where brute force style.

- Create methods to calculate similarity between a pair of values for each different attribute type
- Create dictionaries, utilizing the methods above , with all the similarities for a given pair of customers for all attributes

These would serve the purpose of the similarity matrix.

- Combine the different dictionaries and come up with a final one that would have the avg similarity for all attributes

Code snippet excerpt:

```
# similarity methods
def jaccard_sim(list1, list2):
    intersection = len(list(set(list1).intersection(list2)))
    union = (len(list1) + len(list2)) - intersection
    return round(float(intersection) / union, 2)

def numeric_sim(num1, num2, maxvalue,minvalue):
    return 1- abs(num1-num2) / (maxvalue - minvalue)

def nominal_sim(nom1,nom2):
    return int(nom1 == nom2)

#calculate the similarities for a given numeric attribute
def pairwiseNumeric(cust, maxvalue,minvalue):
    sim={}
    print("Computing numeric similarities")
    start_time = time.time()
    for pair in itertools.combinations(cust, r=2):
        s=numeric_sim(cust[pair[0]],cust[pair[1]],maxvalue,minvalue)
        sim[tuple([pair[0],pair[1]])]=s

    end_time = time.time()
    print("Numeric Success after: " ,end_time-start_time)
    return sim
```

Here is an example of the construction of the [Age] similarity matrix:

```
#calculate similarity matrix using the numeric method
simAge = pairwiseNumeric(age_d, df.Age.max(),df.Age.min())

#show a preview of the matrix
dict(list(simAge.items())[0:10])

{(1, 2): 0.45,
 (1, 3): 0.57,
 (1, 4): 0.84,
 (1, 5): 0.59,
 (1, 6): 0.51,
 (1, 7): 0.21,
 (1, 8): 0.45,
 (1, 9): 0.53,
 (1, 10): 0.46,
 (1, 11): 0.28}
```

Code snippet for calculating the avg of all similarities:

```
def GetDictionaryWithAvgSimilarity(listSim):
    d0=listSim[0]
    i=1;
    while i < len(listSim) :
        for k, value in d0.items():
            d0[k] = round(float((d0[k] + listSim[i][k]) / 2),2)

        i+=1
    return d0

SimilarityList = [simAge,simHouseholdPersons,simIncome,
                  simMaritalStatus, simGroceries, simSex,simOccupation,
                  simCustomerRating, simEducation]

avgSim =  GetDictionaryWithAvgSimilarity(SimilarityList)
```

Bringing it all together to get the 10 - Nearest Neigbors

Now that we had a dictionary with all the customer pairs as a key and the avg Similarity as a Value, it was easy to get the k nearest customers to a given one. Again we used a straight forward approach:

- Filter our dictionary to get all the pairs for a given customer
- Order by the avg Similarity descending
- Get the Top K (= 10) of the list to get the K nearest neigbors

In order to provide the customer Ids as input the only thing you have to do is update the customerIDs list in the attached Jupyter notebook

Final code excerpt

```
import operator

def GetSimilarCustomers(customerID,head,sim):
    f_dict = {k:v for (k,v) in sim.items() if customerID in k}
    sorted_d = dict(sorted(f_dict.items(), key=operator.itemgetter(1),reverse=True))
    return dict(list(sorted_d.items())[0:head])
```

```
#input CustomerIDs
customerIDs = [73, 563, 1603, 2200,3703, 4263, 5300, 6129, 7800, 8555]
kn=10

for i in customerIDs:
    print("Top ",kn, " nearest customers to CustomerID:",i, " ")
    simC = GetSimilarCustomers(i,kn,avgSim)
    for k,v in simC.items():
        if k[0] != i:
            print(k[0]," with similarity score:",v)
        else:
            print(k[1]," with similarity score:",v)

    print("\n")


#snippet of results
Top  10  nearest customers to CustomerID: 73
1203 with similarity score: 0.97
1291 with similarity score: 0.97
1846 with similarity score: 0.97
3623 with similarity score: 0.97
66   with similarity score: 0.96
143  with similarity score: 0.96
347  with similarity score: 0.96
468  with similarity score: 0.96
797  with similarity score: 0.96
872  with similarity score: 0.96


Top  10  nearest customers to CustomerID: 563
3634 with similarity score: 0.99
4290 with similarity score: 0.98
7    with similarity score: 0.97
44   with similarity score: 0.97
351  with similarity score: 0.97
419  with similarity score: 0.97
421  with similarity score: 0.97
426  with similarity score: 0.97
559  with similarity score: 0.97
866  with similarity score: 0.97


Top  10  nearest customers to CustomerID: 1603
109  with similarity score: 0.97
168  with similarity score: 0.97
568  with similarity score: 0.97
4628 with similarity score: 0.97
412  with similarity score: 0.96
1708 with similarity score: 0.96
3286 with similarity score: 0.96
4759 with similarity score: 0.96
651  with similarity score: 0.95
2175 with similarity score: 0.95


Top  10  nearest customers to CustomerID: 2200
203  with similarity score: 0.96
176  with similarity score: 0.95
838  with similarity score: 0.95
```

```
2231 with similarity score: 0.95
2465 with similarity score: 0.95
2562 with similarity score: 0.95
4521 with similarity score: 0.95
4701 with similarity score: 0.95
4903 with similarity score: 0.95
2375 with similarity score: 0.92


Top  10  nearest customers to CustomerID: 3703
1505  with similarity score: 0.97
1604  with similarity score: 0.97
1837  with similarity score: 0.97
3352  with similarity score: 0.97
3410  with similarity score: 0.97
3990  with similarity score: 0.97
4046  with similarity score: 0.97
4373  with similarity score: 0.97
4838  with similarity score: 0.97
448   with similarity score: 0.96


Top  10  nearest customers to CustomerID: 4263
3434  with similarity score: 0.97
1693  with similarity score: 0.96
2733  with similarity score: 0.96
1415  with similarity score: 0.95
231   with similarity score: 0.91
1169  with similarity score: 0.91
1896  with similarity score: 0.91
2195  with similarity score: 0.91
3822  with similarity score: 0.91
4763  with similarity score: 0.91
```

Big Data Optimizations

This approach works good in my powerful Desktop PC with an i7 and 32GB RAM. But of course we can do waaaaaaaay better and we could, if we had a little more time.

Actually the scaling problem we have is not the cpu usage since operations in dictionaries are really fast. Here are the timings for the construction of the similarity matrices, in **seconds**:

```
Computing numeric similarities
Numeric Success after:  9.34

Computing numeric similarities
Numeric Success after:  8.11

Computing numeric similarities
Numeric Success after:  8.70
```

```
Computing Nominal similarities
Nominal Success after:  9.41

Computing Nominal similarities
Nominal Success after:  7.95

Computing Nominal similarities
Nominal Success after:  9.44

Computing numeric similarities
Numeric Success after:  8.61

Computing numeric similarities
Numeric Success after:  8.77

Computing Jaccard similarities
Jaccard Success after:  24.39
```

So it's pretty fast for a naive brute force approach.

The actual price we pay is in the **memory** usage since we end up with 9 different dictionaries each one consisting of 49995000 rows :S .

What we can do, however is keep only one dictionary and add the similarities as a list of values. So here is a small transformation of our code:

Optimized Code excerpt:

```
def pairwiseNominal(cust, appendTo):
    sim={}
    print("Computing Nominal similarities")
    start_time = time.time()
    for pair in itertools.combinations(cust, r=2):
        s=nominal_sim(cust[pair[0]],cust[pair[1]])
        dict(appendTo)[tuple([pair[0],pair[1]])].append(s)
    end_time = time.time()
    print("Nominal Success after: " ,end_time-start_time)
    return appendTo

def pairwiseJaccard(cust, appendTo):
    sim={}
    print("Computing Jaccard similarities")
    start_time = time.time()
    for pair in itertools.combinations(cust, r=2):
        s=jaccard_sim(cust[pair[0]],cust[pair[1]])
        appendTo[tuple([pair[0],pair[1]])] = [s]
    end_time = time.time()
    print("Jaccard Success after: " ,end_time-start_time)
    return appendTo
```

The appendTo parameter is an existing dictionary where we can add to its values

Example usage of the method and its results preview:

```
from collections import defaultdict
dictSim = defaultdict(list)
simGroceries = pairwiseJaccard(groceries_d, dictSim)
simMaritalStatus = pairwiseNominal(marital_d, dictSim)

dictSim

#our one and only dictionary
 (4, 95): [0.0, 1],
 (4, 96): [0.0, 1],
 (4, 97): [0.0, 0],
 (4, 98): [0.0, 1],
 (4, 99): [0.17, 1],
 (4, 100): [0.0, 0],
 (5, 6): [0.12, 0],
 (5, 7): [0.0, 1],
 (5, 8): [0.12, 0],
 (5, 9): [0.0, 1],
 (5, 10): [0.2, 1],
 (5, 11): [0.12, 1],
 (5, 12): [0.08, 1],
 (5, 13): [0.0, 0],
 (5, 14): [0.0, 1],
 (5, 15): [0.0, 0],
 (5, 16): [0.0, 1],
 (5, 17): [0.0, 1].....
```

Of course our final dictionary would have 9 elements in the value lists but the assignment deadline is dangerously close and we should wrap this up and hit the submit button :D.

Have a great day,
Vangelis Christou