**Handmade k-modes library**

이 석 재

**Contents**

1. Abstract

2. Description

3. Usage examples and results

4. Conclusion

5. Source code

6. References

**1. Abstract**

k-modes is used for clustering categorical variables. It defines clusters based on the number of matching categories between data points. (This is in contrast to the more well-known k-means algorithm, which clusters numerical data based on Euclidean distance.)

Basically, I tried to implement the library by keeping the contents of the lecture materials and the algorithm flow. I also tried to minimize the heterogeneity by making an interface similar to ‘scikit-learn’, which is a widely used machine learning library.

**2. Description**

K-Modes clustering

class k\_modes.custom\_kmodes.**kmodes**(n\_clusters=3, n\_init=3, max\_iter=300, random\_state=2019)

**Parameters**

**n\_clusters**: int, optional, default:3

The number of clusters to form as well as the number of centroids to generate.

**n\_init**: int, default: 3

Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n\_init consecutive runs in terms of inertia.

**Max\_iter**: int, default: 300

Maximum number of iterations of the k-means algorithm for a single run.

**Random\_state**: int, default: 2019

Determines random number generation for centroid initialization. Use an int to make the randomness deterministic.

**Attributes**

**Cluster\_centers**: list, [n\_clusters, n\_features]

Coordinates of cluster centers.

**labels\_**: list

Labels of each point.

**n\_iter**: int

Number of iterations run.

**Methods**

**1. \_\_init\_\_**(self, n\_clusters=3, n\_init=3, max\_iter=300, random\_state=2019, labels\_=list(), predict=list(), purity=0, cluster\_counter=Counter(), n\_iter=0)

**2. fit**(self, X)

Compute k-modes clustering

Parameters

**X**: ***array-like or sparse matrix, shape=(n\_samples, n\_features)***

Training instances to cluster. It must be noted that the data will be converted to categorical value-dataset.

**3. predict**(self, X, sample\_weight=None)

Predict the closest cluster each sample in X belongs to.

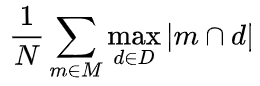
Parameters

**X : *{array-like, sparse matrix}, shape = [n\_samples, n\_features]***

New data to predict.

**4. cal\_purity**(self, X, y)

Returns the purity of the clustering result. Formally, given some set of clusters M and some set of classes D, both partitioning N data points, purity can be defined as:



Parameters

**X**: **list, shape=[n\_samples, target\_value]**

The list of clustered labels of each data in dataset

**y: DataFrame, shape=(n\_samples, 1)**

The DataFrame of target value of dataset which already known.

**3. Usage examples and results**

Usage examples

import pandas as pd  
# import custom kmodes library  
from k\_modes.custom\_kmodes import k\_modes  
  
# read dataset  
dataset=pd.read\_csv("mushrooms.csv")  
# drop target variable for clustering  
X = dataset.drop(columns=["class"])  
y = dataset["class"]  
# initialize model  
model=k\_modes(n\_clusters=8, n\_init=3, max\_iter=300, random\_state=2019)  
# training  
model.fit(X)  
# return clustering result  
cluster\_status=model.cluster\_counter  
centroid\_list=model.cluster\_centers  
purity=model.cal\_purity(model.labels\_, y)  
# print result  
print("Result")  
print("Cluster status: ",cluster\_status)  
print("Purity: ",purity)

Results

n\_clusters=3

\*\*\*\*\*\*\*\*\*\* iter 11 start \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 11 end \*\*\*\*\*\*\*\*\*\*\*\*  
Counter({0: 3875, 2: 2769, 1: 1480})  
\*\*\*\*\*\*\*\*\*\* Computing the new center \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 12 start \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 12 end \*\*\*\*\*\*\*\*\*\*\*\*  
Counter({0: 3875, 2: 2769, 1: 1480})  
\*\*\*\*\*\*\*\*\* Finish \*\*\*\*\*\*\*\*\*\*  
Result  
Cluster status: Counter({0: 3875, 2: 2769, 1: 1480})  
Purity: 0.7581240768094535  
  
Process finished with exit code 0

n\_clusters=4

\*\*\*\*\*\*\*\*\*\* iter 8 start \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 8 end \*\*\*\*\*\*\*\*\*\*\*\*  
Counter({1: 3596, 3: 1970, 2: 1371, 0: 1187})  
\*\*\*\*\*\*\*\*\*\* Computing the new center \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 9 start \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 9 end \*\*\*\*\*\*\*\*\*\*\*\*  
Counter({1: 3596, 3: 1970, 2: 1371, 0: 1187})  
\*\*\*\*\*\*\*\*\* Finish \*\*\*\*\*\*\*\*\*\*  
Result  
Cluster status: Counter({1: 3596, 3: 1970, 2: 1371, 0: 1187})  
Purity: 0.870384047267356  
  
Process finished with exit code 0

n\_clusters=8

\*\*\*\*\*\*\*\*\*\* iter 16 start \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 16 end \*\*\*\*\*\*\*\*\*\*\*\*  
Counter({2: 2233, 4: 1336, 3: 1205, 6: 961, 5: 897, 1: 786, 0: 514, 7: 192})  
\*\*\*\*\*\*\*\*\*\* Computing the new center \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 17 start \*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\* iter 17 end \*\*\*\*\*\*\*\*\*\*\*\*  
Counter({2: 2233, 4: 1336, 3: 1205, 6: 961, 5: 897, 1: 786, 0: 514, 7: 192})  
\*\*\*\*\*\*\*\*\* Finish \*\*\*\*\*\*\*\*\*\*  
Result  
Cluster status: Counter({2: 2233, 4: 1336, 3: 1205, 6: 961, 5: 897, 1: 786, 0: 514, 7: 192})  
Purity: 0.8879862136878385  
  
Process finished with exit code 0

In the case of the scikit-learn library, the progress of the learning is not displayed on the console, but this time, the log is printed to confirm that the operation is working properly.

I adjusted the value of n\_clusters to 3, 4, and 8 to run it in the same test code. **As the number of clusters increased, the purity increased(0.76 🡪 0.87 🡪 0.89).** And the same thing happened with clustering using DBSCAN for the last programming homework 3.

**4. Conclusion**

This time I implemented k-modes, one of the clustering algorithms. I used a hamming distance policy to handle categorical values and purity as a clustering performance measure. Finally, I adjusted the variable and method structure so that it had an interface similar to scikit-learn, which is a popular machine learning library.

When coding, you usually concentrate on the implementation. But you need to think about whether you're doing dirty coding because you're focusing on the implementation. There is a reason that the code in popular and large open source libraries is clean and easy to understand. I think the library can evolve over time because of coding that takes care of other developers, such as setting variables that are easy to understand, creating methods, and even documenting them. This was a good experience to see my lack as a developer and the power of open source.

**5. Source Code**

custom\_kmodes.py

import pandas as pd  
import random  
from collections import Counter  
  
# initialize custom kmodes class  
class k\_modes:  
 # method  
 # initialize model  
 def \_\_init\_\_(self, n\_clusters=3, n\_init=10, max\_iter=300, random\_state=2019):  
 self.n\_clusters = n\_clusters  
 self.n\_init=n\_init  
 self.max\_iter=max\_iter  
 self.random\_state=random\_state  
 self.labels\_=list()  
 self.predict=list()  
 self.purity=0  
 self.cluster\_counter=Counter()  
 self.cluster\_centers=list()  
 self.n\_iter=0  
 # method  
 # train the model using dataset  
 def fit(self, dataset):  
  
 len1 = dataset.shape[0]  
 feature\_len = dataset.shape[1]  
  
 centroid\_index = []  
 centroid\_index2=[]  
 centroid\_width=[]  
 # 1. choose best initial centroid value using n\_init, random\_state  
 for j in range(0,self.n\_init):  
 for i in range(0, self.n\_clusters):  
 if j==0 and i==0:  
 random.seed(self.random\_state)  
 randnum = random.randint(0, len1)  
 centroid\_index.append(randnum)  
 pass  
 centroid\_index.sort()  
 # 1.1 calculate the maximum index length of picked centroids sets  
 centroid\_width.append(max(centroid\_index) - min(centroid\_index))  
 centroid\_index2.append(centroid\_index)  
 centroid\_index=[]  
 pass  
 # 1.2 choose initial centroids sets which has the maximum index length  
 final\_index=centroid\_width.index(max(centroid\_width))  
 centroid\_index=centroid\_index2[final\_index]  
 centroid\_len = len(centroid\_index)  
  
 print("Initialized Centroid Index: ", centroid\_index)  
 print("n\_clusters: ", centroid\_len)  
 predict = []  
 count = [0] \* centroid\_len  
  
 print("\*\*\*\*\*\*\*\*\*\* iter 1 start \*\*\*\*\*\*\*\*\*\*\*\*")  
 # calculate distance with mode vectors using hamming distance policy  
 for i in range(0, len1):  
 count = [0] \* centroid\_len  
 for j in range(0, centroid\_len):  
 for k in range(0, feature\_len):  
 # if comparing 2 values are different, distance=distance+1  
 if dataset.iloc[centroid\_index[j], k] != dataset.iloc[i, k]:  
 count[j] = count[j] + 1  
 pass  
 pass  
 pass  
 # check which centroid has the minimum distance value  
 centroid\_result = count.index(min(count))  
 # clustering  
 predict.append(centroid\_result)  
 pass  
 print("\*\*\*\*\*\*\*\*\*\* iter 1 end \*\*\*\*\*\*\*\*\*\*\*\*")  
 # print clustering result  
 print(Counter(predict))  
 dataset['cluster'] = predict  
  
 # iteration >= 2  
 iter\_num = 2  
 while True:  
 # flag for stop iteration which there's no movement of clusters change  
 stop\_flag = 0  
 new\_centroid\_list = []  
 new\_centroid\_value = []  
 print("\*\*\*\*\*\*\*\*\*\* Computing the new center \*\*\*\*\*\*\*\*\*\*\*\*")  
 # find the new centroid(mode vector) for cluster  
 for k in range(0, centroid\_len):  
 df2 = dataset[dataset['cluster'] == k]  
 df2\_len = df2.shape[0]  
 # if cluster size is 0, continue  
 if df2\_len == 0:  
 continue  
 # new centroid is the vector which has the most frequent values in each features  
 # use mode() to find the most frequent values  
 new\_centroid = df2.mode().loc[0]  
 # add new centroid  
 new\_centroid\_list.append(new\_centroid)  
 # pre-calculate a distance if a centroid & new data has same vector value  
 list2 = []  
 for i in range(0, feature\_len):  
 num = df2.iloc[:, i].value\_counts().values[0]  
 num = 1 - (num / df2\_len)  
 list2.append(num)  
 pass  
 new\_centroid\_value.append(list2)  
 pass  
 # print("n\_clusters: ", len(new\_centroid\_list))  
 # new iter start  
 predict = []  
 print("\*\*\*\*\*\*\*\*\*\* iter ", iter\_num, " start \*\*\*\*\*\*\*\*\*\*\*\*")  
 for i in range(0, len1):  
 count = [0] \* len(new\_centroid\_list)  
 for j in range(0, len(new\_centroid\_list)):  
 for k in range(0, feature\_len):  
 if new\_centroid\_list[j][k] != dataset.iloc[i, k]:  
 count[j] = count[j] + 1  
 pass  
 # if centroid's feature's value == dataset's feature's value  
 elif new\_centroid\_list[j][k] == dataset.iloc[i, k]:  
 # add the pre-calculated value  
 count[j] = count[j] + new\_centroid\_value[j][k]  
 pass  
 pass  
 pass  
 centroid\_result = count.index(min(count))  
 # print(centroid\_result, " // ", dataset.iloc[i, feature\_len])  
 if centroid\_result != dataset.iloc[i, feature\_len]:  
 # if there's no cluster movement, set stop flag  
 stop\_flag = 1  
 pass  
 predict.append(centroid\_result)  
 pass  
 dataset['cluster'] = predict  
 print("\*\*\*\*\*\*\*\*\*\* iter ", iter\_num, " end \*\*\*\*\*\*\*\*\*\*\*\*")  
 print(Counter(predict))  
 # save attributes  
 self.predict=predict  
 self.labels\_=predict  
 self.cluster\_counter=Counter(predict)  
 iter\_num = iter\_num + 1  
 # if stop flag is 1, or over max\_iter, stop the training  
 if stop\_flag == 0 or iter\_num+1==self.max\_iter+1:  
 self.n\_iter=iter\_num  
 self.cluster\_centers=new\_centroid\_list  
 print("\*\*\*\*\*\*\*\*\* Finish \*\*\*\*\*\*\*\*\*\*")  
 break  
 pass  
 pass  
 # methood  
 # return predict cluster list  
 def predict(self):  
 return self.predict  
 # method  
 # return purity value  
 def cal\_purity(self, predict, y):  
 predict = pd.DataFrame(predict, columns=['cluster'])  
 y = pd.DataFrame(y)  
 r = pd.concat([predict, y], axis=1)  
  
 list\_key = list(Counter(predict['cluster'].values).keys())  
 list\_value = list(Counter(predict['cluster'].values).values())  
 r = r.sort\_values(['cluster'], ascending=True)  
 dic = dict(zip(list\_key, list\_value))  
  
 list\_key2 = list(dic.keys())  
 list\_value2 = list(dic.values())  
  
 poison\_sum = r.shape[0]  
 purity = 0  
  
 for n in range(0, len(list\_key2)):  
 poison1 = r.loc[(r['cluster'] == n) & (r['class'] == 'p')].shape[0]  
 poison2 = r.loc[(r['cluster'] == n) & (r['class'] == 'e')].shape[0]  
 poison\_check = max(poison1, poison2)  
 poison\_check = poison\_check / poison\_sum  
 purity = purity + poison\_check  
 pass  
 return purity  
 pass  
 pass

**6. References**

Python package index – kmodes

<https://pypi.org/project/kmodes/>

Scikit-learn official documents – sklearn.cluster.KMeans

<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>