بخش اول

پیاده‌سازی الگوریتم خوشه بندی:

الگوریتم به صورت یک تابع به نام Key_Identification پیاده‌سازی شده.

```python
import numpy as np
import math


def Key_Identification(vectors , k_nighboors , Data_Dimension, C_Target , g):
    vecrotsCount = len(vectors)

    data = np.empty((vecrotsCount  , Data_Dimension))
    # sakht label ha
    L_set = np.arange(vecrotsCount)
    C_Current = math.floor(vecrotsCount / g)

    for i in range(0 , vecrotsCount) :
            data[i] = vectors[i]

    if (C_Target >= C_Current):
        return L_set

```

ورودی های تابع شامل vectors که وکتورهای ورودی است، k_nighboors که تعداد همسایه های هر داده در محاسبات است، Data_Dimension که تعداد ویژگی‌های هر وکتور است، C_target و g است.

سپس با توجه به مقادیر ورودی، مقدار دهی های اولیه انجام میشود.

محاسبه آرایه دوبعدی فاصله نقطه به نقطه طبق الگوریتم مقاله به شکل زیر انجام میشود.

```python
    # mohasebe D_Original
    D_Original = np.zeros((vecrotsCount, vecrotsCount) )

    for i in range(0 , vecrotsCount ):
            for j in range(0 , vecrotsCount) :
                D_Original[i , j] = dist(i , j , data, Data_Dimension)


```

تابع dist:

```python
import numba
@numba.jit(nopython=True)
def dist(i , j, data, dimension):

        distance = 0
        distance = np.linalg.norm(float(data[i]),float(data[j]))
        # for k in range(0 , dimension):
        #     distance += math.pow(data[i, k] - data[j, k], 2)

        # return math.sqrt(distance)
        return distance

```

محاسبه مجموعه همسایگی های R به شکل زیر انجام میشود.

```
    # mohasebe R har element
    R_set = np.empty((vecrotsCount, k_nighboors + 1))
    for i in range( 0 , vecrotsCount):
        sorted = np.argsort(D_Original[i])
        for j in range(0 , k_nighboors + 1):
            R_set[i][j] = sorted[j]

    R_set = R_set.astype(int)

```

محاسبه ماتریس فاصله خوشه در گام اول به شکل زیر انجام میشود. در این گام هر داده یک خوشه است و برای محاسبه فواصل بین خوشه ها طبق الگوریتم، داده و k همسایه اش را در نظر میگیریم.

```
    # mohasebe D-Current
    D_Current = np.zeros((vecrotsCount ,vecrotsCount))

    for i in range(0 , vecrotsCount):
        for j in range(0 , i):
            # if (i == j):
            #     D_Current[i][j] = 0
            # else:
                avg_dist = 0
                for k in range(0 , k_nighboors+1):
                    for l in range(0 , k_nighboors+1):
                        # R_set[i][k] = int(R_set[i][k])

                        avg_dist += D_Original[R_set[i][k]][R_set[j][l]]

                avg_dist = avg_dist / ((k_nighboors+1)*(k_nighboors+1))
                D_Current[i][j] = avg_dist
                D_Current[j][i] = avg_dist

```

شروع حلقه تکرار الگوریتم با این شرط که مقدار C_target همواره از C_current کوچکتر باشد.
درگام اول این حلقه، عناصر کلیدی را محاسبه می کنیم.

```
    while(C_Current > C_Target):
        keyPoints = findKeyPoints(n_keys=C_Current ,D_Current_set=D_Current)

```

تابع پیدا کردن عناصر کلیدی، در گام اول به شکل زیر، کلید اولیه را طبق الگوریتم بدست می اورد.

```python
def findKeyPoints(n_keys, D_Current_set):
        m = len(D_Current_set)
        min = math.inf
        s_keys = []
        key = 0

        for i in range(0, m):
            distance = 0
            for j in range(0, m):
                distance += D_Current_set[i][j]
            avg_distance = distance/m

            if (avg_distance < min):
                min = avg_distance
                key = i
        s_keys.append(key)
```

پس از یافتن کلید اول، کلید های دیگر به شکل زیر پیدا می شوند. در انتهای تابع، آرایه s_keys که شامل کلید های پیدا شده است، بازگردانده می شود.

```
        for i in range(0 , n_keys-1):
            max = -math.inf
            key = 0

            for j in range(0 , m):
                min = math.inf

                for k in range(0 , len(s_keys)):
                    if (D_Current_set[j][s_keys[k]] < min):
                        min = D_Current_set[j][s_keys[k]]

                if min > max :
                 max = min
                 key = j

            s_keys.append(key)

        return s_keys

```

با یافتن کلید ها، الگوریتم شروع به یافتن ادغام خوشه ها و بروز رسانی آرایه L_set که مربوط به لیبل های داده هاست، میکند. این بخش فاصله هر خوشه از خوشه های کلیدی را بررسی و خوشه با فاصله کمینه را انتخاب میکند.

```
    # egham 2 cluster, ba peyda kardan cluster ba fasele minimum az cluster i
        m = len(D_Current)
        for i in range(0, m):
            min = math.inf
            min_cluster_index =0

            for j in range(0 , len(keyPoints)):
                if (D_Current[i][keyPoints[j]] < min):
                    min = D_Current[i][j]
                    min_cluster_index = j

            for k in range(0, len(L_set)):
                if (L_set[k] == i):
                    L_set[k] = min_cluster_index

```

در گام نهایی، باید ماتریس فاصله خوشه ها اپدیت شود. این کار با ساختن مجموعه های P مورد نیاز معرفی شده در الگوریتم، به شکل زیر انجام میشود.

```
#  update faseleha
    p_set = [[]*vecrotsCount]

    for i in range(0, len(keyPoints)):
        for j in range(0,vecrotsCount):
            if (L_set[j] == keyPoints[i]):
                p_set[keyPoints[i]].append(j)
                p_set[keyPoints[i]] = list(set().union(p_set[keyPoints[i]] , R_set[j]))
```

در ادامه، با داشتن مجموعه p های مورد نیاز برای هر خوشه، طبق شکل زیر، ماتریس فاصله های جدید متناسب با تعداد خوشه های جدید که برابر تعداد عناصر کلیدی جدید است ساخته می شود.

```
    m = len(keyPoints)
    D_Current = np.zeros((m , m))

    for i in range(0, m):
        for j in range(0, i):
        #     if (i == j):
        #         D_Current[i][j] = 0
        #     else:
                value = matrix_new_dist(p_set[i] , p_set[j] , D_Original)
                D_Current[i][j] = value
                D_Current[j][i] = value

    C_Current =  math.floor(C_Current/g)
```

برای محاسبه ماتریس فاصله جدید، از این تابع استفاده میشود.

```
def matrix_new_dist(p_set1 ,p_set2, Dist_Original):
    sum = 0
    for a in range( 0 , len(p_set1)) :
        for b in range( 0 , len(p_set2)):
            sum += Dist_Original[p_set1[a]][p_set2[b]]

    return  (sum / (len(p_set1) * len(p_set2)))
```

در پایان تابع، با خروج از حلقه، آرایه L_set شامل لیبل ها بازگردانده می شود.

خواندن داده و تبدیل به داده مناسب استفاده:

در این بخش به ابتدا به خواندن داده‌ها می پردازیم. در شروع کار لایبرری های مورد نیاز را ایمپورت میکنیم و داده‌ها را از فایل می خوانیم.

```
import os
import random
import re
import string
import  math
import nltk
import numpy as np
import pandas as pd

from gensim.models import Word2Vec

from nltk import word_tokenize
from nltk.corpus import stopwords

from sklearn.cluster import MiniBatchKMeans
from sklearn.metrics import silhouette_samples, silhouette_score

nltk.download("stopwords")
import nltk
nltk.download('punkt')

SEED = 42
random.seed(SEED)
os.environ["PYTHONHASHSEED"] = str(SEED)
np.random.seed(SEED)
```

read data

```
from google.colab import drive

drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_remount=True).
```

```
[ ] import pandas as pd
    train_path = '/content/gdrive/MyDrive/nlp/train.csv'
    test_path = '/content/gdrive/MyDrive/nlp/test.csv'

    df = pd.read_csv(train_path)


    df_test = pd.read_csv(test_path)

    comments = df.values[:10,1]
    comments_test = df_test.values[:10,1]
```

در گام بعدی، پروسس داده‌های خوانده شده را انجام می دهیم. پروسس های این بخش شامل حذف سطر و ستون‌های اضافی، حذف داده‌های با مقدار null، حذف کلمات زائد و توکن سازی داده هاست. منظور از توکن سازی به صورت خلاصه یعنی تبدیل رشته‌های خام به ارایه هایی شامل کلمات مفید است.

```python
custom_stopwords = set(stopwords.words("english") + ["news", "new", "top"])
text_columns = ["Id", "Comment", "Topic"]

df = df[:10].copy()
df_test = df_test[:10].copy()

df["Topic"] = df["Topic"].fillna("")
df_test["Topic"] = df_test["Topic"].fillna("")

for col in text_columns:
    df[col] = df[col].astype(str)
    df_test[col] = df_test[col].astype(str)

# Create text column based on title, description, and Topic
df["text"] = df[text_columns].apply(lambda x: " | ".join(x), axis=1)
df["tokens"] = df["text"].map(lambda x: clean_text(x, word_tokenize, custom_stopwords))


#for test
df_test["text"] = df_test[text_columns].apply(lambda x: " | ".join(x), axis=1)
df_test["tokens"] = df_test["text"].map(lambda x: clean_text(x, word_tokenize, custom_stopwords))
```

تابع clean_text وظیفه توکن سازی کامنت های ورودی را دارد.

```python
def clean_text(comments, tokenizer, stopwords):
    """Pre-process comments and generate Kword

    Args:
        comments: comments to tokenize.

    Returns:
        Tokenized comments.
    """
    comments = str(comments).lower()  # Lowercase words
    comments = re.sub(r"\[(.*?)\]", "", comments)  # Remove [+XYZ chars] in content
    comments = re.sub(r"\s+", " ", comments)  # Remove multiple spaces in content
    comments = re.sub(r"\w+…|…", "", comments)  # Remove ellipsis (and last word)
    comments = re.sub(r"(?<=\w)-(?=\w)", " ", comments)  # Replace dash between words
    comments = re.sub(
        f"[{re.escape(string.punctuation)}]", "", comments
    )  # Remove punctuation

    Kword = tokenizer(comments)  # Get Kword from comments
    Kword = [t for t in Kword if not t in stopwords]  # Remove stopwords
    Kword = ["" if t.isdigit() else t for t in Kword]  # Remove digits
    Kword = [t for t in Kword if len(t) > 1]  # Remove short Kword
    return Kword
```

پردازش هایی پس از توکن سازی، مانند حذف کلمات تکراری و...

```
tok = [df["tokens"].values[i][j]for i in range(len(df["tokens"])) for j in range(1,len(df["tokens"].values[i]))]
tok_test = [df_test["tokens"].values[i][j]for i in range(len(df_test["tokens"])) for j in range(1,len(df_test["tokens"].values[i]))]

tok1 = list(set(tok))
tok1_test = list(set(tok_test))
#print(tok1)

tok2 = []
for i in range(len(df["tokens"])):
  a = list(df["tokens"].values[i][1:len(df["tokens"].values[i])-1])
  tok2.append(a)


tok2_test = []
for i in range(len(df_test["tokens"])):
  a = list(df_test["tokens"].values[i][1:len(df_test["tokens"].values[i])-1])
  tok2_test.append(a)
```

```
  tok2_test.append(a)


# Remove duplicated after preprocessing
_, idx = np.unique(df["tokens"], return_index=True)
df = df.iloc[idx, :]
#for test
_, idx = np.unique(df_test["tokens"], return_index=True)
df_test = df_test.iloc[idx, :]

# Remove empty values and keep relevant columns
df = df.loc[df.tokens.map(lambda x: len(x) > 0), ["text", "tokens"]]
df_test = df_test.loc[df.tokens.map(lambda x: len(x) > 0), ["text", "tokens"]]

docs = df["text"].values
tokenized_docs = df["tokens"].values

#for test
docs = df_test["text"].values
tokenized_docs = df_test["tokens"].values
```

محاسبات مربوط به bag of words:

در این تابع، با توجه به کامنت های ورودی، توکن های استخراج شده و کلمات هر کامنت، محاسبات مربوط به پیدا کردن tf و idf وtdf انجـــام میشود. ورودی تابع:

```
tok_ALL = list(set(tok1 + tok1_test))
X_train = tf(comments,tok_ALL,tok2)
X_test = tf(comments_test,tok_ALL,tok2_test)
```

مقدار دهی اولیه متغییر های مورد نیاز:

```
def tf(comment,tok,AW):

    corpus = comment
    words_set = tok



    n_docs = len(corpus)          #·Number of documents in the corpus
    n_words_set = len(words_set) #·Number of unique words in the
    df_tf = pd.DataFrame(np.zeros((n_docs, n_words_set)), columns=words_set)
```

محاسبات مربوط به tf:

```
# Compute Term Frequency (TF)
for i in range(n_docs):
    words = AW[i]
    #words = corpus[i].split(' ') # Words in the document
    for w in words:
        df_tf[w][i] = df_tf[w][i] + (1 / len(words))
```

محاسبات مربوط به idf:

```
# Compute Inverse Document Frequency (IDF)
idf = {}

for w in words_set:
    k = 1    # number of documents in the corpus that contain this word

    for i in range(n_docs):
        if w in AW[i]:
            k += 1

    idf[w] =  np.log10(n_docs / k)
```

محاسبه نهایی:

```
df_tf_idf = df_tf.copy()

for w in words_set:
    for i in range(n_docs):
        df_tf_idf[w][i] = df_tf[w][i] * idf[w]

return df_tf_idf
```

تبدیل کلمات توکن شده به وکتور جهت استفاده برای خوشه بندی با کمک word2vec:

در این بخش، جهت اینکه کلمات برای خوشه بندی و ادغام کلمات هم معنی مناسب باشند، باید کلمات را به وکتور تبدیل کنیم.

برای اینکار از یک مدل از پیش ترین شده از گوگل که با 3 میلیون کلمه ترین شده استفاده می کنیم.

create model and load vocabulary

```
[ ] model = Word2Vec(sentences=tokenized_docs, size=100, workers=1, seed=SEED)
```

```
import gensim
import gensim.downloader as api

wv = api.load('word2vec-google-news-300')
```

تابع انجام عمل وکتورسازی:

vectorize

```
def vectorize(list_of_docs, model):
    """Generate vectors for list of documents using a Word Embedding

    Args:
        list_of_docs: List of documents
        model: Gensim's Word Embedding

    Returns:
        List of document vectors
    """
    features = []
```

```
    for tokens in list_of_docs:
        zero_vector = np.zeros(model.vector_size)
        vectors = []
        for token in tokens:
            if token in model.wv:
                try:
                    vectors.append(model.wv[token])
                except KeyError:
                    continue
        if vectors:
            vectors = np.asarray(vectors)
            avg_vec = vectors.mean(axis=0)
            features.append(avg_vec)
        else:
            features.append(zero_vector)
    return features
```

انجام وکتور سازی با کال کردن این تابع و مدل گوگل:

```
vectorized_docs = vectorize(tokenized_docs, model=model)
```

خروجی این تابع به مدل خوشه بندی ارسال می شود.

پیاده‌سازی مدل کلاس بندی:

در این بخش، از یک درخت تصمیم، برای کلاس بندی داده‌ها استفاده کرده ایم. این مدل داده‌های ترین و تست را دریافت میکند، و لیبل های خروجی را باز میگرداند.

DecisionTree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Y = label haye comments

label = df.drop(['Id','Comment'], axis=1)
label_test = df_test.drop(['Id','Comment'], axis=1)

Y = [label.values[i][0] for i in range(len(label))]
Y_test = [label_test.values[i][0] for i in range(len(label_test))]

Y = Y[:10]
Y_test = Y_test[:10]

model = DecisionTreeClassifier()
clf = model.fit(X_train, Y)


labels = clf.predict(X_test)
labels1 = clf.score(X_test,Y_test)
#label1 = accuracy_score(Y_test,labels)
labels1
```

```
0.6
```

دقت بدست آمده در روش ما 0.6 است.

بخش دوم

توضیح منطق:

روش پیشنهادی ما به این شکل است که ابتدا داده‌ها خوانده و پردازش می شوند. برای پیدا کردن مقادیر ورودی، داده‌های به توکن هایی تبدیل میشوند. چالش مساله که استفاده از word2vec برای بهبود کلاس بندی رخ میدهد، به این شکل است که ما در نظر داریم کلمات مشابه را با کمک خوشه بندی در یک خوشه قرار دهیم، سپس با توجه به این نکته که داده‌های هم معنی و شمارش یکسان این داده ها، اشکال مطرح شده در کلاس برای bag of words را برطرف میکند، کلمات خوشه بندی شده را به bag of words میدهیم. برای ورودی خوشه بندی نیز، نیاز به تبدیل کلمات به وکتور داریم.