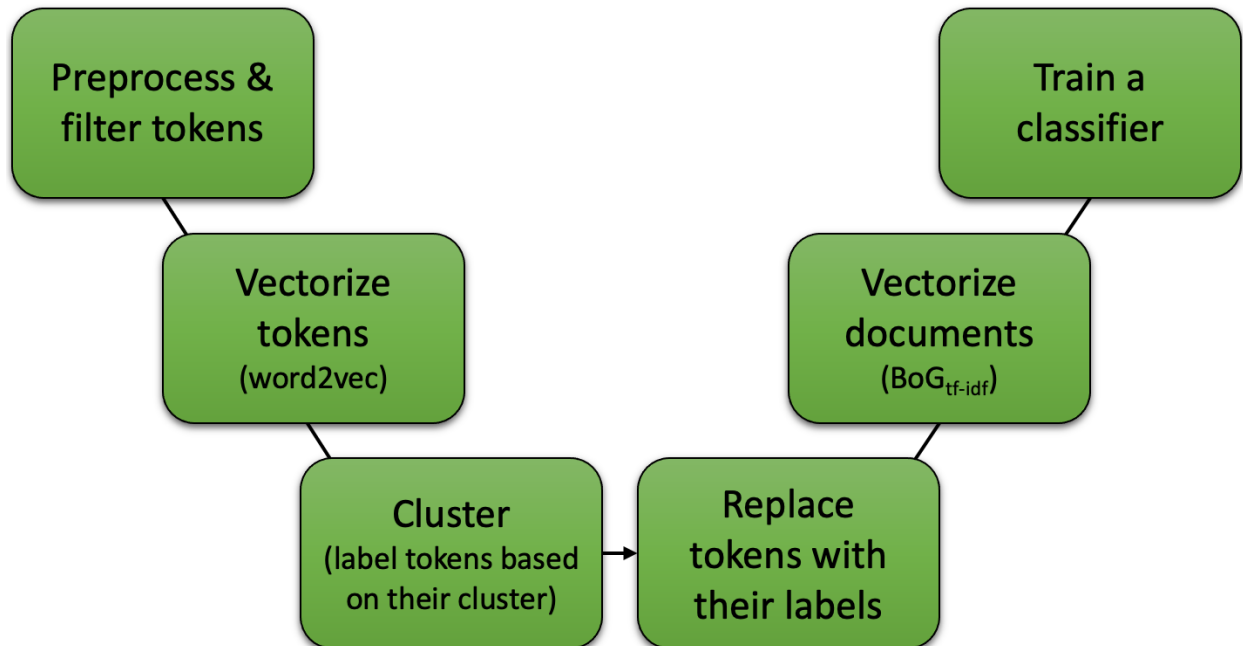


در این پروژه از ما خواسته شده بود تا با کمک گرفتن از متود word2vec، مشکل روش Bag of Words (Bog) که کلمات هم معنا و مفهوم را جداگانه در نظر میگرفت را حل کنیم. تصویری که در ادامه آمده است، چکیده روش ارائه شده را نشان میدهد:



:Preprocess & filter tokens

در این بخش به پیش پردازش و تمیز کردن دیتا مورد نظر پرداختیم. در این راستا ابتدا تمامی '\n' ها در متن را با کاراکتر فاصله ' ' جایگذاری کردیم. به دنبال آن، از آنجایی که اعداد در تعیین topic متن مورد نظر نقش بسزایی ندارند، همه ی اعداد در متن را پاک کردیم. پس از اصلاحات ذکر شده، متن بر اساس حروف و ارقام، tokenize کردیم. با اینکار کاراکتر هایی مثل ایموجی های موجود در برخی از متن ها به صورت خودکار ignore میشوند.

در ادامه هر token را به صورت جداگانه بررسی میکنیم که اگر جزو stop word های مشهور بود، کلاً آن token را ignore میکنیم در غیر این صورت آن token را ریشه یابی (lemmatizing) میکنیم و سپس به لیست token ها اضافه میکنیم. در آخر، به ازای هر متن (هر سطر از dataframe) مجموعه ای token های فیلتر شده داریم:

```
def preprocess_sentence(sentence, tokenizer, lemmatizer, stop_words):
    sentence = sentence.replace(r'\n', ' ')
    sentence = re.sub(r'\d+', '', sentence) # remove numbers
    tokens = tokenizer.tokenize(sentence) # split a sentence into tokens
    filtered_tokens = []
    for token in tokens:
        if token.lower() not in stop_words:
            lemmatized_token = lemmatizer.lemmatize(token)
            filtered_tokens.append(lemmatized_token.lower())

    return '#'.join(np.unique(filtered_tokens)) # form a string from obtained tokens

stop_words = stopwords.words('english')
lemmatizer = WordNetLemmatizer()
tokenizer = RegexpTokenizer(r'([a-zA-Z]+|[0-9]+)')

train_df['Unique_tokens'] = train_df['Comment'].apply(
    lambda row: preprocess_sentence(
        row,
        tokenizer,
        lemmatizer,
        stop_words
    )
)
```

پس از اعمال موارد ذکر شده، تعداد token های منحصر به فرد کل دیتاست از 34,079 به 15,107 تا کاهش پیدا میکند.

train_df.head()			
Id	Comment	Topic	Unique_tokens
0x840	A few things. You might have negative- frequen...	Biology	advantage#allele#alter#alternating#animal#anot...
0xbf0	Is it so hard to believe that there exist part...	Physics	anything#believe#detect#exist#far#find#hard#hu...
0x1dfc	There are bees	Biology	bee
0xc7e	I'm a medication technician. And that's alot o...	Biology	alot#body#care#definitely#die#drug#fine#good#i...
0xbba	Cesium is such a pretty metal.	Chemistry	cesium#metal#pretty

:Vectorize tokens

در این بخش ما از مدل pre-train شده word2vec-google-news-300 استفاده کردیم. این مدل شامل بردار برای 3,000,000 کلمه است که هرکدام از این بردار ها، 300 بعدی هستند.

با داشتن token های فیلتر شده، کافیت تا آنها را به این مدل بدهیم و بردار های آنها را برای مرحله بعدی ذخیره کنیم. اما نکته ای که وجود داشت این بود که این دیتا، از سطح اینترنت جمع آوری شده بود لذا شامل مواردی مثل غلط املائی، حاوی URL، و یا حتی space نخوردن بین دو کلمه بود، پس دور از انتظار نبود که token هایی وجود داشته باشند مدل word2vec استفاده شده، آنها را شامل نشود. در این راستا برای حل مشکل غلط املائی از کتابخانه ای تحت عنوان SpellChecker استفاده کردیم. این کتابخانه بدین صورت کار میکرد که اگر token پاس داده شده به آن واقعا شامل غلط املائی بود، شکل اصلاح شده آن token را باز میگرداند و در غیر اینصورت خود token را بدون تغییر بازمیگرداند. همچنین بقیه مشکل های ذکر شده را در مراحل بعدی، به آنها رسیدگی کردیم که در جلوتر به آنها اشاره میکنیم.

```
cnt = 0
words = []
words_vectors = []
spell_checker = SpellChecker()

for indx, word in enumerate(all_unique_words_list):
    if indx % 2000 == 0:
        print(f'Stored {indx}/{len(all_unique_words_list)} words already!')

    try:
        if word not in word2vec_model.vocab:
            word = spell_checker.correction(word)

        words_vectors.append(word2vec_model[word])
        words.append(word)
    except Exception as e:
        print(word)
        cnt += 1
```

در زیر نمونه ای از token هایی که vector ای برای آنها در مدل استفاده شده وجود ندارد، آورده شده است:

```

duranium
duckduckgo
dumas
dumbshits
dupont
dysparxia
dystanttyger
eafajtulslzh
easycalculation
eddddedecebeacbcca
edna
eenzbbmm
efeedbcdfc
elsa
efwxrsdug
ehat
ehy
eigensolvers
eigenstates
einsteinian
einsteinian
einsteins
einstenium
    
```

:Cluster

برای خوشه بندی کردن کلمات از این روش خوشه بندی استفاده میکنیم. در این مرحله هدف کلاستر کردن کلمات نزدیک به یکدیگر از نظر معنایی است. پیاده سازی:

پیاده سازی این بخش مرحله به مرحله طبق مقاله داده شده صورت گرفته. در ابتدا پس از دریافت داده و تعریف متغیرها، ماتریس فاصله را به کمک فاصله اقلیدسی محاسبه میکنیم.

$$D_{original}[i,j]=dist(i,j)$$

```

for i in range (0,n):
    for j in range (0,n):
        d[i][j] = math.dist(data[i],data[j])
    
```

پس از آن کد به دو بخش اصلی تقسیم شده است:
تابع key_clustering 2- تابع detect_key که عناصر کلیدی را پیدا میکند.

: key_clustering

ورودی این تابع ماتریس فاصله، تعداد کلاسترهای هدف (و یا هر شرط پایان دیگر)، k که مشخص کننده تعداد همسایه برای هر عنصر است و g .
در ابتدا پس از تعریف متغیرها، آرایه R را محاسبه میکنیم که یک آرایه $N*k$ است و حاوی k تا نزدیک ترین همسایه به عنصر موجود در آن اندیس است.

```
for i in range (0,n):
    list_k = sorted(dd[i])[0:k]
    for j in range(0,n): # finde index of that values
        for x in range(0,len(list_k)):
            if dd[i][j] == list_k[x] :
                r[i][x] = j
```

C_{per} , c_{curr} را مقدار دهی اولیه میکنیم طبق فرمول مقابل :

$$C_{previous}=N, C_{current}= \lfloor N/g \rfloor$$

پس از آن لیبل ها را مقدار دهی اولیه میکنیم .
بعد از این مرحله طبق فرمول مقابل $d_{current}$ را محاسبه میکنیم.

$$D_{current}[i,j] = \frac{1}{(k+1)^2} \times \sum_{a \in \{i\} \cup R_k(i), b \in \{j\} \cup R_k(j)} D_{original}[a,b]$$

```
for i in range (0,n):
    for j in range(0,n):
        temp = 0
        for a in range (0,k):
            for b in range (0,k):
                temp = temp + dd[int(r[i][a] ),int(r[j][b])]
        d_cur[i][j] = temp / ((k)*(k))
```

در این مرحله وارد حلقه میشویم:

شرط حلقه: $c_cur > c_target$

در ابتدا تابع `detect_key` را با مقادیر ورودی که برابر با d_cur , c_cur هستند را فراخوانی میکنیم و به عنوان خروجی $s_current$ را دریافت میکنیم.

```
s_cur = detect_key(d_cur ,c_cur)
```

با توجه به خروجی تابع `detect_key` لازم است تا لیبل ها آپدیت شوند.

$$L[i] = \arg \min_{j \in S_{current}} D_{current}[L[i],j]$$

```
for i in range(len(l)):
    min = float('inf')
    if(l[i] not in s_cur):
        for j in s_cur:
            if d_cur[l[i],j] < min:
                min = d_cur[ l[i] , j ]
                index = j
        l[i] = index
```

و به دنبال آن ماتریس فاصله $d_current$ براساس فرمول مقابل برورسانی شود:

$$D_{current}[i,j] = \frac{1}{\|P_i\| \|P_j\|} \times \sum_{a \in p_i, b \in p_j} D_{original}[a,b]$$

```

unique_values = list(Counter(l).keys()) # equals to list(set(words))
unique_count = list(Counter(l).values()) # counts the elements' frequency

for i in unique_count:
    e = np.zeros((i*(3)))
    samp.append(e)

for x in range(0, len(unique_values)):
    nn = 0
    for y in range(0, len(l)):
        if unique_values[x] == l[y]:
            samp[x][nn] = y
            nn = nn+1

            for i in range(1, k):
                samp[x][nn] = r[y][i]
                nn = nn+1

for e in samp:
    lo.append(list(map(int, e)))

label_dict = dict(zip(unique_values, lo))

for x, y in label_dict.items():
    for p, q in label_dict.items():
        sum = 0
        for a in y:
            for b in q:
                sum += dd[a, b]

        d_cur[x, p] = sum / (len(q) * len(y))

c_per = c_cur
c_cur = math.floor(c_cur / g)
    
```

در این مرحله لازم است که c_{Cur} , c_{per} آپدیت شوند.

$$C_{previous} = C_{current}, C_{current} = \lfloor C_{current} / g \rfloor$$

اگر شرط حلقه برقرار باشد این روند تکرار میشود.
اگر از حلقه خارج شود دوباره `detect_key` با ورودی های `d_curr` , `c_target` فراخوانی شده و `s_final` را دریافت میکنیم.

در نهایت لیبل ها براساس s_final آپدیت شده و به عنوان خروجی بازگردانده میشود.

تابع detect_key

پس از تغییر متغیرها لازم است که I1 را پیدا کنیم و سپس K و S را آپدیت کنیم.

$$I_1 = \arg \min_{1 \leq i \leq m} \frac{1}{m} \sum_{j=1}^m D[i, j]$$

```
for i in range (0,m):# i1
    sum = 0
    for j in range (0,m):
        sum += d_cur[i][j]

    temp.append(sum / m)

i1 = temp.index(sorted(temp)[0])

for t in range (0,m):# update s and k for i1
    if t != i1:
        k.append(t)
s.append(i1)
```

در این مرحله وارد حلقه با شرط $n < c_cur - 1$ میشویم:
در هر مرحله طبق فرمول مقدار I را بدست آورده و S, k را آپدیت میکنیم تا زمانی که از حلقه خارج شویم. پس از خارج شدن از حلقه S را به عنوان مجموعه ای از I ها خروجی میدهیم.

$$I_n = \arg \max_{I_k \in K_n} \min_{I_j \in S_n} D[I_k, I_j]$$


```

for i in k: #finde min
    min = float('inf')
    for j in s:
        if d_cur[i][j] < min:
            min = d_cur[i][j]
    min_i.append(min)# list of min values in each row

for i in k:# find max of mins
    for j in s:
        if d_cur[i][j] == max( min_i):
            i_n = i
            k.remove(i_n)
            p = 1
            break
    if p == 1:
        break

s.append(i_n)
n+=1

```

تعداد خوشه ها را با این فرض که به طور میانگین در هر خوشه 2 تا 3 تا token اگر قرار بگیرد حالت مطلوبی است در نظر گرفتیم که با کمی tune کردن پارامتر ها به $c_{target}=5147$ رسیدیم که به طور میانگین تعداد 2.7 تا token را به ازای هر خوشه میدهد. در زیر نمونه ای از تعداد token ها به ازای هر label خوشه آمده است:

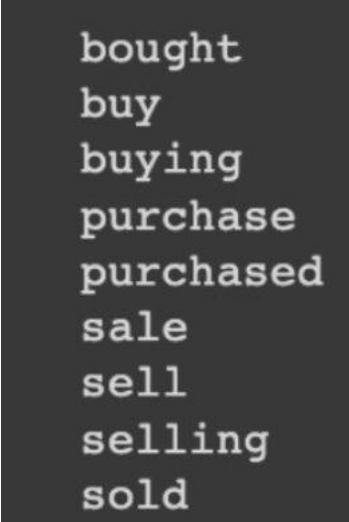
```

Counter({474: 13,
          979: 3,
          626: 16,
          641: 6,
          4312: 4,
          4988: 1,
          1613: 2,
          1007: 4,
          1692: 2,
          638: 3,

```

:Replace tokens with labels

در این مرحله با قرار گرفتن هر token رد یک خوشه، در متن اصلی، هر token را با label خوشه ای که در آن قرار گرفته است، جایگزین میکنیم. به عنوان نمونه در متن اصلی، در هر جا هر کدام از کلمات زیر را که مشاهده کردیم، آنها را به label آن کلاستر جایگذاری میکنیم. با اینکار در مراحل بعدی، هنگام محاسبه tf-idf هر کدام از آنها به صورت جدا در نظر گرفته نمیشوند. بلکه مانند این است که همگی یک کلمه هستند که در محاسبه frequency تاثیر خواهند گذاشت.



bought
buy
buying
purchase
purchased
sale
sell
selling
sold

همانطور که در مراحل قبل اشاره شد، بنابر یک سری مشکلات، وکتور برخی از token ها در مدل word2vec استفاده شده موجود نیست، در این حالت token مورد نظر را به صورت یک خوشه تک عضوی در نظر گرفته که label خوشه با خود token برابر است:

```
token = str(labels.get(token, token)) #key: token, default: token
```

```
def preprocess_sentence(sentence, spell_checker, tokenizer, lemmatizer, stop_words, labels):
    sentence = sentence.replace(r'\n', ' ')
    sentence = re.sub(r'\d+', '', sentence) # remove numbers
    tokens = tokenizer.tokenize(sentence) # split a sentence into tokens

    filtered_tokens = []
    for token in tokens:
        if token.lower() not in stop_words:
            token = lemmatizer.lemmatize(token).lower()
            if token not in list(labels.keys()):
                token = spell_checker.correction(token)

            token = str(labels.get(token, token)) #key: token, default: token

            filtered_tokens.append(token)

    return '#'.join(filtered_tokens) # form a string from obtained tokens

stop_words = stopwords.words('english')
lemmatizer = WordNetLemmatizer()
tokenizer = RegexpTokenizer(r'([a-zA-Z]+|[0-9]+)')
spell_checker = SpellChecker()

train_df['filtered_tokens'] = train_df['Comment'].apply(
    lambda row: preprocess_sentence(
        row,
        spell_checker,
        tokenizer,
        lemmatizer,
        stop_words,
        labels
    )
)
```

:Vectorize documents (BoG)

حال با یکسان در نظر گرفتن کلماتی که از لحاظ معنایی نزدیک بهم هستند، میتوان الگوریتم BoG را اجرا کرد. بدین منظور هر document را به وکتوری شامل مقدار tf-idf هر token منحصر بفرد، تبدیل میکنیم. لازم به ذکر است که در این مرحله فقط token هایی را نگه میداریم که حداقل در 2 تا document ظاهر شده باشند (در غیر این صورت آن token در ساخت وکتور ها برای هر document در نظر گرفته نمیشود)

```
tfidf = TfidfVectorizer(min_df=2, sublinear_tf=True)
```

با انجام کارهای ذکر شده، وکتور مربوط به هر document دارای 4730 بعد خواهد بود:

```

1 features = tfidf.fit_transform(train_df['filtered_tokens'])
1 features = features.todense()
1 features.shape
(8695, 4730)

```

:Train a classifier

در این مرحله با داشتن وکتور داکيومنت ها (features) و topic آنها (ground_truth)، میتوانیم به صورت supervised یک classifier آموزش بدهیم. بدین منظور ما از الگوریتم های SVM, Perceptron و MLP استفاده کردیم که نتایج آنها در ادامه آورده شده اند:

Algorithm	Accuracy
SVM	63.14%
Perceptron	60.66%
MLP	70.80%

