مصطفی کرمانی نیا

شماره دانشجویی : **810101575**

گزارش کار مراحل انجام پروژه

بخش صفرم : شروع کار

مرحله اول :

در این مرحله شرایط لازم برای نوشتن پروژه را مهیا کردم، مثلا **extension** های لازم را در **vscode** ریخته و در نصب هضم روی ویندوز به مشکل خوردم که با اپدیت کردن **pip** حل شد.

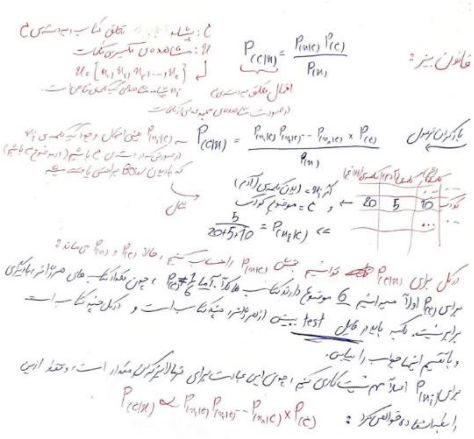
به فایل آموزش پایتون رجوع کردم و **numpy** و **pandas** و **matplotlid** را با **pip** ریختم و برای شروع در با **pandas** فایل **train** را باز کردم و چک کردم که درست کار بکند.(البته بعدا متوجه شدم با **jupyter** به مراتب راحتتر میتوان جدول های **pandas** را مشاهده کرد)

```
import pandas as pd
data=  pd.read_csv("books_test.csv")

for i in data.index:
    print(f" its title {data['title'][i]}")
    print(f" its describe {data['description'][i]}")
    print(f" its cat {data['categories'][i]}")
```

مرحله دوم:

بخش غیرامتیازی ویدیوی مربوط به پروژه را دیدم و پی دی اف پروژه را با دقت خواندم و اندکی روی روابط ریاضیاتی آن محاسبات انجام دادم که کلیات مسئله دستم بیاید



جواب های تقریبی ای برای پرسش های مطرح شده در فایل دارم که در ادامه پاسخ نهایی را خواهم داد.

بخش اول : پیش پردازش

مرحله سوم:

برای پیش پردازش ابتدا با هضم، تمام رشته ها را به کلمه تبدیل کردم و سپس با **pandas** آنها را به یک **dandas dataframe** تبدیل کردم سپس یک لیست از علائم نگارشی ساختم تا بوسیله ی آنها علائم نگارشی اضافه را فیلتر کنم

```
punctuationMarks = [':', '،', '?', '؟', '.', ';',
                    '(', ')', '!', '+', '-', '*', '/', '=', '\\', '»', '«',
'"', '_', '...']
```

در برخی موارد هضم درست عملیات جداسازی را انجام نمیداد و مثلا اگر عبارت (زکریای رازی(**1970-2021**) را با یکی دو تا فاصله ی اضافی بهش میدادیم، عبارت **1970-2021** را یک کلمه در نظر میگرفت که خب نه عدد بود و نه علامت نگارشی، پس نمیتوانستم این موارد را فیلتر کنم. پس از **regex** استفاده کردم و تلاش کردم کلا هر کلمه ای که هر حرفی بجز حروف فارسی دارد را حذف کنم.

این برای حروف انگلیسی بود ولی برای فارسی دردسر بیشتری داشت و باید با یونیکد حروف فارسی **regex** را مینوشتم:

```
result = re.sub(r'[^A-Za-z]', '', s)
```

این برای فارسی نتیجه شد:

```
res = "".join(re.findall(r"[\u0600-\u06FF]+", s))
```

سپس حدس زدم برخی کلمات انگلیسی هم مهم اند پس این را نوشتم

```
res = "".join(re.findall(r"[\u0600-\u06FF]|[A-Z]|[a-z]+", s))
```

اما نهایتا زمانی که صرف میشد برای این تحلیل دقیق، بسیار زیاد بود پس خطای زیادتر را بیخیال شده و با فرض اینکه تمامی متن های توضیحات و عنوان کتاب ها کاملا از لحاظ ادبی اصلاح شده و درست اند و دارای فاصله های بیجا نیستند، به همان حذف علائم نگارشی و اعداد اکتفا کردم:

```
punctuationMarks = [':', '،', '?', '؟', '.', ';',
                    '(', ')', '!', '+', '-', '*', '/', '=', '\\',
'»', '«', '"', '_', '...']


...

lambda w: w not in punctuationMarks and w.isdigit() == False
```

همچنین یکسری کلمات هستند که در بسیاری از متون وجود دارند و آمدن آنها در یک متن اطلاعات خاصی در باب موضوع آن متن به ما نمیدهد مانند برخی افعال پرکاربرد(است،بود،رفت) و برخی حرف اضافه ها(به، و ، اما) ، اینها را **StopWords** نامیده و مجموعه ای از آنها هم داخل خود کتابخانه ی هضم موجود بود و هم در یک فایل برایمان در تلگرام قرار داده شد که من از فایل تلگرامی استفاده کردم و تمام **StopWord** ها را از بین کلماتم حذف کردم.(در ادامه هر تحلیلی که میکنم، یکبار بدون حذف **stop words** و یکبار با حذف آن، نتیجه را نوشته ام)

نهایتا به یک تابع رسیدم که عملیات پیش پردازش را روی یک **cell** انجام میدهد و کافیست انرا روی **dataframe** اصلی **apply** کنم(البته یک تابع خوب دیگر برای اینکار تابع **isalpha** بود که کد آنرا هم در فایل کد کامنت کردم)

```
punctuationMarks = [':', '،', '؟', '؟', '.', '؛',
                    '(', ')', '!', '+', '-', '*', '/', '=', '\\', '»',
'«', '"', '_', '...']
stopWords = pd.read_csv("sw.csv")

def preprocess_a_cell(txt):
    wordsDf = pd.DataFrame(word_tokenize(txt), columns=["word"])
    wordsDf["word"] = wordsDf[wordsDf["word"].apply(
        lambda w: w not in punctuationMarks and w not in
stopWords["word"].values and w.isdigit() == False )]
    wordsDf = wordsDf.dropna()

    return wordsDf
```

روند **apply** کردن بوسیله ی پانداس هم اینگونه نوشتم:

```
df = pd.read_csv("bokbok.csv")
df["description"] = df["description"].apply(preprocess_a_cell)
df["title"] = df["title"].apply(preprocess_a_cell)
```

نتیجه گیری : در کل در این بخش تلاشم بر این بود که چیزهایی که بیش از حد عام هستند(مانند **stopWords** و علائم نگارشی) و یا چیزهایی که بیش از حد اختصاصی هستند(مانند کلمات خاص انگلیسی یا اعداد) را از بین کلماتم حذف کنم تا احتمال هایی که در نهایت محاسبه می شوند، دقیق تر باشند و احتمال خطا کمتر شود.(در ادامه نیز برای دقیقتر شدن از ریشه یابی نیز استفاده خواهیم کرد چون کلماتی که از یک ریشه هستند، امرنشان در جمله اطلاعات یکسانی از موضوع آن جمله به ما میدهد)

بخش دوم : حل مسئله

تا اینجای کار ما یک دیتافریم پانداس داریم که **3** ستون دارد و در ستون **title** و **description** در هر کدام از سطر ها یک دیتافریم دیگر موجود است(البته **pandas** برای دیتافریم هایی که یک ستون دارند چیزی بنام **series** هم دارد که چون استفاده از آن تاثیر خاصی در ذخیره ی زمان نداشت، همان دیتافریم را ترجیح دادم) که یک ستون دارد و شامل کلمات بدرد بخوری است که نگه داشته ایم و در ستون **categories** هم یک رشته(**string**) بعنوان موضوع کتاب آمده است.

مرحله چهارم :

برای شروع ساخت **BOW** باید تعداد سطر های آنرا محاسبه کنیم، پس آمدم و تعداد کتاب های موجود در هر موضوع را شمردم تا احتمال وجود هر کتاب در هر موضوع هم بدست بیاید چون بعدا در محاسبات به آن نیاز داریم:

```
countOfCategories = {}
for i in df.index:
    if (df['categories'][i] in countOfCategories):
        countOfCategories[df['categories'][i]] += 1
    else:
        countOfCategories[df['categories'][i]] = 1
print(countOfCategories)
out:
{'جامعه شناسی': 425, 'کلیات اسلام': 425, 'داستان کودک و نوجوانان': 425, 'داستان کوتاه': 425, 'مدیریت و کسب و کار': 425, 'رمان': 425}
```

در نتیجه جمعا **6** موضوع داریم که یعنی **BOW** دارای **6** سطر است و همچنین احتمال اینکه یک کتاب رندوم دارای موضوع **x** باشد، دقیقا یک ششم است.

مرحله پنجم:

یک **BOW** خالی دارای **6** ستون میسازیم با نام های موضوعاتمان و سپس به وسیله ی کد زیر پیمایش کرده و پس از حدود **9** دقیقه و **30** ثانیه جدول **BOW** تکمیل میشود(وجود **BOW.copy** این است که اگر نباشد در طول کد در اصل به تعداد بالایی ما ستون اضافه میکنیم به دیتافریم اصلی و این اضافه کردن ها **performance** را بشدت کاهش میدهند و هم زمان اجرای برنامه زیاد میشود و هم به **performanceWarning** میفوریم. پس برای رفع این مشکلات یک کپی ساخته و تغییرات را رویش اعمال می کنیم.

```
categories = ['جامعه‌شناسی', 'کلیات اسلام',
                    'داستان کودک و نوجوانان', 'داستان کوتاه', 'مدیریت و کسب و کار', 'رمان']
BOW = pd.DataFrame(index=categories)
for i in df.index:
    for word in df['description'][i]['word']:
        if word not in BOW.columns:
            BOW = BOW.copy()
            BOW[word] = 0
        BOW[word][df['categories'][i]] += 1
BOW
```

Out :

9m 23.0s Python

	ساختار	نظریه‌های	جامعه‌شناسی	ایران	نوشته	ابوالفضل	رمضانی	دربردارنده	نظریه	جامعه	...	واهمه‌ای	ستمگر	خیرخواهی	عاد	بیاور
جامعه‌شناسی	32	26	127	182	132	3	1	9	52	206	...	0	0	0	0	0
کلیات اسلام	2	1	1	34	97	2	0	6	13	61	...	0	0	0	0	0
داستان کودک و نوجوانان	0	0	0	21	122	2	6	5	0	4	...	1	1	1	1	1
داستان کوتاه	3	0	0	46	156	1	0	22	1	24	...	0	0	0	0	0
مدیریت و کسب و کار	7	6	5	23	159	0	0	5	8	26	...	0	0	0	0	0
رمان	5	0	0	39	147	0	0	1	1	39	...	0	0	0	0	0

6 rows × 41110 columns

مرحله ششم:

زمانی که برای ساختن **BOW** مصرف میشد بیش از حد انتظار بود در نتیجه تصمیم گرفتم ساختار را به دو دیکشنری تو در تو تغییر دهم تا با **dynamic** شدن تعداد اعضای هر سطر، از بوجود آمدن **cell** های اضافه با مقدار **0** جلوگیری کرده و در زمان صرفه جویی کنم.

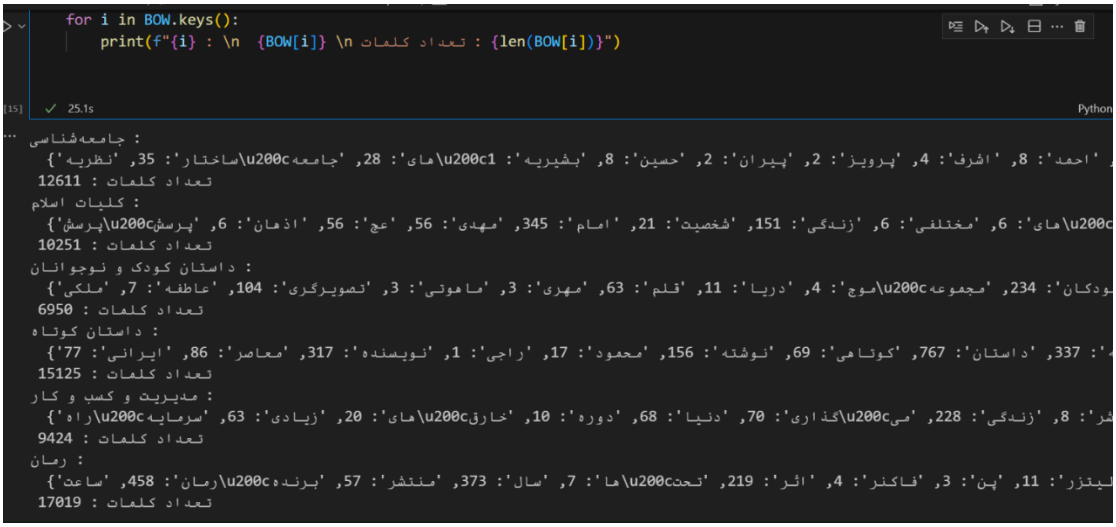
در نهایت یک دیکشنری تو در تو به شکل زیر داریم :

```
BOW = {'جامعه‌شناسی': {},  'کلیات اسلام': {},  'داستان کودک و نوجوانان': {},
       'داستان کوتاه': {},  'مدیریت و کسب و کار': {},  'رمان': {}}
```

و کد نهایی اینگونه شد:

```
for bookIndex in df.index:
    curCategory = df['categories'][bookIndex]
    for word in df['description'][bookIndex]['word']:
        if word not in BOW[curCategory].keys():
            BOW[curCategory][word] = 0
        BOW[curCategory][word] += 1
    for word in df['title'][bookIndex]['word']:
        if word not in BOW[curCategory].keys():
            BOW[curCategory][word] = 0
        BOW[curCategory][word] += 1
```

همچنین از لحاظ زمانی بخاطر کم کردن تعداد ستون ها و تغییر فرمت از **pandas DF** به **python dict** بسیار صرفه جویی کردیم و زمان اجرای کد به کمتر از **30** ثانیه کاهش یافت:

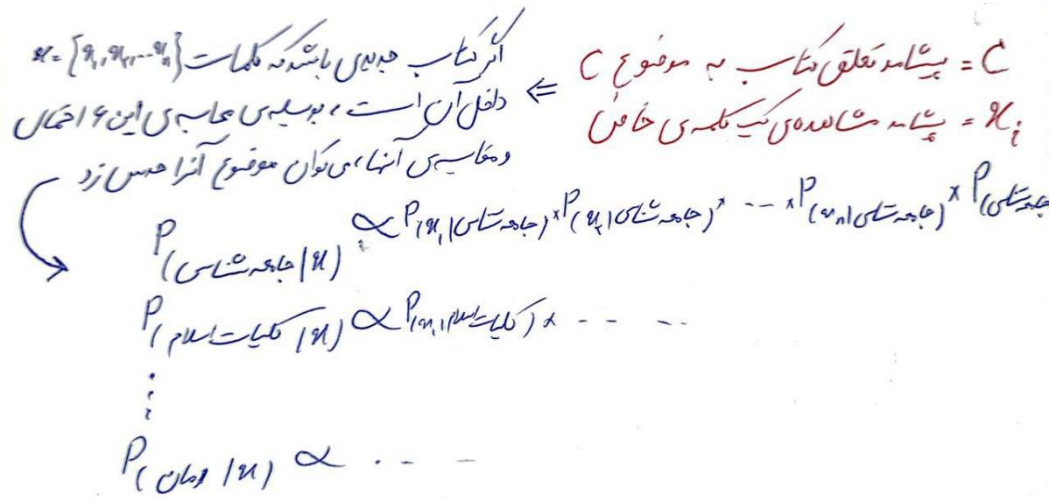


بخش سوم : تست کردن الگوریتم

مرحله هفتم:

حالا که مدل ما آماده شده است باید آنرا تست کنیم و بکمک فایل **books_test** درصد درستی آنرا چک کنیم

اگر فرض کنیم که :



تا اینجا ما در مرحله ی چهارم احتمال **prior** یا همان p_c را که مساوی یک ششم بود یافتیم و بوسیله ی **BOW** رسما $p_{x_i|c}$ ها را داریم.

پس در این مرحله ابتدا **BOW** را به **PROBS BOW** تبدیل میکنیم یعنی همون دیکشنری قبل را داریم اما بعنوان مقدار جلوی هر کلمه، $p_{x_i|c}$ آنرا نوشته ایم.

```
def convert_BOW_to_probs_BOW(BOW):
    for category in BOW.keys():
        countOfAllWords = 0

        for word in BOW[category].keys():
            countOfAllWords += BOW[category][word]

        for word in BOW[category].keys():
            # change value of words to conditional prob of them
            BOW[category][word] = BOW[category][word]/countOfAllWords

    return BOW
```

نتیجه:

مرحله ی هشتم:

به سراغ ساختن تابع اصلی تست میرویم

```
def test(BOW,testFileAddress):
    testDf = pd.read_csv(testFileAddress)
    testDf["description"] =
testDf["description"].apply(preprocess_a_test_cell)
    testDf["title"] = testDf["title"].apply(preprocess_a_test_cell)
    trueTests = 0
    for bookIndex in testDf.index:
        categoryProbs = {'جامعه‌شناسی' : 0, 'کلیات اسلام' : 0, ' داستان کودک و
نوجوانان':0,
            'داستان کوتاه':0, 'مدیریت و کسب و کار':0, 'رمان':0}
        bookTitle = testDf['title'][bookIndex]['word']
        bookDescription = testDf['description'][bookIndex]['word']
        for category in categoryProbs.keys():
            categoryProbs[category] =
calculateCategorylikelihood(bookTitle,bookDescription,BOW[category])
        if returnModelGuess(categoryProbs) ==
testDf['categories'][bookIndex] :
            trueTests += 1
    return trueTests/len(testDf)
```

اگر در توضیحات موجود درباره ی یک کتاب، با کلمه ای مواجه شویم که در BoW وجود نداشته باشد چه باید کرد؟ احتمال صفر باید در نظر گرفت یا باید آن کلمه را در نظر نگرفت؟ *برای تابع* **calculateCategorylikelihood** *این حالات امتحان کرده و نتیجه را گزارش و تفسیر می کنیم:*

حالت اول :

احتمال **1** *میگیریم برای کلمه ای که در* **BOW** *نیست؛(یعنی آن کلمه را در نظر نمیگیریم)*

```
def calculateCategorylikelihood (bookTitle,bookDescription,categoryDict):
    likelyhood = 1/6
    #  ایگنور کردن کلماتی که نیستن و دادن مقدار 1 بجاشون
    for word in bookDescription :
        if word in categoryDict.keys():
            likelyhood *= categoryDict[word]
        else:
            likelihood *= 1
    for word in bookTitle :
        if word in categoryDict.keys():
            likelyhood *= categoryDict[word]
        else:
            likelihood *= 1
    return likelyhood
```

و در نهایت **6** *عدد بعنوان احتمال حضور در هر موضوع در دیکشنری* **categoryProbs** *خواهیم داشت اما نکته اینجاست که چون برای کلماتی که در* **BOW** *نیستند مقدار* **1** *را ذکر کرده ایم که بیشترین احتمال ممکن است، باید آن موضوعی را بعنوان نتیجه ی تحلیل ذکر کنیم که احتمالش از همه کمتر شده است(یعنی تعداد* **1** *های کمتری در هم ضرب شده اند و تعداد اعداد بین* **0** *و* **1** *زیاد بوده پس عدد نهایی یک عدد کوچک است)؛*

```
def returnModelGuess(categoryProbs):
    probs = list(categoryProbs.values())
    categories = list(categoryProbs.keys())
    return categories[probs.index(min(probs))]
```

نتیجه ی حالت اول با حذف stop words**:**

```
تعداد کل تست ها :  450
تعداد تست های درست پاسخ داده شده :  158
درصد بازدهی این مدل :  35.11111111111111
```

نتیجه ی حالت اول بدون حذف stop words**:**

```
تعداد کل تست ها :  450
تعداد تست های درست پاسخ داده شده :  132
درصد بازدهی این مدل :  29.333333333333332
```

حالت دوم :

احتمال **0** *میگیریم برای کلمه ای که در* **BOW** *نیست:*

```
def calculateCategorylikelihood(bookTitle,bookDescription,categoryDict):
    likelyhood = 1/6
    #  ایگنور کردن کلماتی که نیستن و دادن مقدار 1 بجاشون
    for word in bookDescription :
        if word in categoryDict.keys():
            likelyhood *= categoryDict[word]
        else:
            likelihood *= 0
    for word in bookTitle :
        if word in categoryDict.keys():
            likelyhood *= categoryDict[word]
        else:
            likelihood *= 0
    return likelyhood
```

در این حالت برخلاف حالت **1** *،برای کلمه ای که در* **BOW** *نبوده کمترین احتمال ممکن(***0***) را در نظر گرفته ایم پس نهایتا باید بین احتمال های حضور در موضوعات مختلف،* **MAX** *را محاسبه کنیم و بعنوان نتیجه ذکر کنیم:*

```
def returnModelGuess(categoryProbs):
    probs = list(categoryProbs.values())
    categories = list(categoryProbs.keys())
    return categories[probs.index(max(probs))
```

نتیجه ی حالت دوم بدون حذف stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 75
درصد بازدهی این مدل : 16.666666666666664
```

نتیجه ی حالت دوم با حذف stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 75
درصد بازدهی این مدل : 16.666666666666664
```

تفسیر: بدیهتا بالاخره در توضیحات و نام هر کتاب یک کلمه یافت میشود که در **BOW** نبوده باشد، در نتیجه همان یدونه **0** باعث میشود $p_{x|c}$ برای تمامی کتاب ها **0** شده و پایتون بطور رندوم یک کتاب را انتخاب کند که در نهایت هم به احتمال یک ششم یا همان **16**٪ انتخاب های درست را انجام داده ایم.

حالت سوم :

استفاده از لوگاریتم و در نظر نگرفتن کلماتی که در **BOW** نیستند

```
def calculateCategorylikelihood(bookTitle,bookDescription,categoryDict):
    likelyhood = 0
    #  ایگنور کردن کلماتی که نیستن و دادن مقدار0 بجاشون
    for word in bookDescription :
        if word in categoryDict.keys():
            likelyhood+= np.log(categoryDict[word])
        else:
            likelyhood += 0
    for word in bookTitle :
        if word in categoryDict.keys():
            likelyhood += np.log(categoryDict[word])
        else:
            likelyhood += 0
    return likelyhood
```

نتیجه ی حالت سوم بدون حذف stop words**:**

```
تعداد کل تست ها :  450
تعداد تست های درست پاسخ داده شده :  190
درصد بازدهی این مدل :  42.22222222222222
```

نتیجه ی حالت سوم با حذف stop words**:**

```
تعداد کل تست ها :  450
تعداد تست های درست پاسخ داده شده :  194
درصد بازدهی این مدل :  43.111111111111114
```

پاسخ پرسش1 : اگر کلمه ای باشد که در سطر مد نظر ما در **BOW** نباشد، کار بهتر این است که آنرا در نظر نگیریم، چون در این صورت اولا که به احتمال بسیار زیاد در هر داده ی **TEST** بالاخره کلمه ای است که **train** نشده باشد، پس تقریبا در تمام تست ها به حالتی می رسیم که احتمال تعلق داشتن به هر موضوع همان یک ششم است که در **prior** بود، و یا مثلا

یک کلمه در هیچ موضوعی **train** نشده ولی بطور شانسی در یک موضوع بی ربط مثلا یک بار آمده است، و همین ضرب کردن باعث میشود که مدل ما فکر کند به احتمال **100** درصد آن کتاب مال آن موضوع است که کاملا هم بی ربط بوده است با آن.

پاسخ پرسش2 : در کل میدانیم که $p_{x_i|c}$ ها اعداد کوچکی هستند(همانطور که در عکس **PROBS BOW** دیدیم از اردر **0.0001** هستند) و در روش قبلی برای محاسبه، تعداد زیادی ازین اعداد کوچک در هم ضرب میشدند که باعث میشد در برخی مواقع از میزان دقت پایتون فراتر رفته و مقدار آنها را تقریبا **0** محسوب کند که باعث خطا در محاسبات میشد، اما با لوگاریتم گرفتن، در اصل بجای ضرب کردن اعداد بسیار نزدیک به صفر، لوگاریتم آنها را که یکسری اعداد منفی هستند را با هم جمع میکنیم که خب عدد حاصل یک عدد منفی بزرگ ممکن است بشود اما انقدر بزرگ نمیشود که پایتون نتواند آنرا حساب کند پس ازین لحاظ همانند روش قبلی به مشکل نخواهیم خورد.

حالت چهارم :

استفاده از لوگاریتم و **0** گذاشتن احتمال کلماتی که در **BOW** نیستند:

نتیجه ی حالت چهارم بدون حذف stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 75
درصد بازدهی این مدل : 16.666666666666664
```

نتیجه ی حالت چهارم با حذف stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 75
درصد بازدهی این مدل : 16.666666666666664
```

اینکار بدیهتا بدلیل استفاده از **log(0)** در برخی مواقع، به **warning** هایی میخورد اما نهایتا نتیجه دقیقا مثل حالت **2** میشود چون بالاخره در توضیحات و نام هر کتاب یک کلمه یافت میشود که در **BOW** نبوده باشد، در نتیجه همان برونه **log(0)** باعث میشود $p_{x|c}$ برای تمامی کتاب ها منفی بینهایت شده و پایتون بطور رندوم یک کتاب را انتخاب کند که در نهایت هم به احتمال یک ششم یا همان **16**٪ انتخاب های درست را انجام داده ایم

در باب additive smooth :

در روش **2** و **4** دیدیم که اگر حتی یکی از $p_{x_i|c}$ ها مساوی صفر شود، باعث میشود که کل $p_{x|c}$ هم صفر شود، پس در این روش می آییم و بجای صفر، یک مقدار کوچک بعنوان $p_{x_i|c}$ برای کلماتی که در **BOW** نبودند میگذاریم که به آن مشکل نخوریم.

حالا برای یافت آن مقدار کوچک من ازین فرمول استفاده کردم:

$$\hat{P}(x_i \mid \omega_j) = \frac{N_{x_i,\omega_j} + \alpha}{N_{\omega_j} + \alpha d} \quad (i = (1, \ldots, d))$$

where

- N_{x_i,ω_j}: Number of times feature x_i appears in samples from class ω_j.
- N_{ω_j}: Total count of all features in class ω_j.
- α: Parameter for additive smoothing.
- d: Dimensionality of the feature vector $\mathbf{x} = [x_1, \ldots, x_d]$.

برای ثابت دلخواه الفا هم با اندکی تحلیل و همچنین آزمودن مقادیر مختلف، فهمیدم عددی از اردر **1/100000** بهترین بازدهی را میدهد چون در کل از اردر **100000** کلمه داریم در **train** و اگر بطور کلی به احتمالات بنگریم برای کلمه ای که در **BOW** نیست هم بالاخره **1/100000** احتمال حضور وجود دارد. و با بررسی مقادیر مختلف این اردر دیدم **1/500000** بهترین نتیجه را میدهد.

حالت پنجم :

استفاده از **additive smooth** (بدون گرفتن لوگاریتم)

در این حالت باید تغییراتی در مرحله ای که **BOW** را اپدیت میکردیم بوجود بیاوریم و احتمال هاییکه مینویسیم را با فرمول جدید بنویسیم:

```
def convert_BOW_to_additive_smooth_probs_BOW(BOW):
    for category in BOW.keys():
        countOfDifferentWords = len(BOW[category])
        countOfAllWords = countAllWords(BOW[category])
        for word in BOW[category].keys():
            # change value of words to conditional prob of them
            BOW[category][word] = (BOW[category][word] +ALPHA)/
(countOfAllWords + (ALPHA * countOfDifferentWords))

    return BOW
```

و در محاسبه ی بخش **likelihood** نیز به گونه ای دیگر برای کلماتی که در **BOW** نیستند عمل میکنیم:

```
def calculateCategorylikelihood(bookTitle, bookDescription, categoryDict):
    likelyhood = 1/6
    wordsCount = countAllWords(categoryDict)
    for word in bookDescription:
        if word in categoryDict.keys():
            likelyhood *= categoryDict[word]
        else:
            likelyhood *= ALPHA/((wordsCount +
                                  (ALPHA * len(categoryDict)))
    for word in bookTitle:
        if word in categoryDict.keys():
            likelyhood *= categoryDict[word]
        else:
            likelyhood *= ALPHA/((wordsCount +
                                  (ALPHA * len(categoryDict)))
    return likelyhood
```

نتیجه ی حالت پنجم بدون حذف stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 140
درصد بازدهی این مدل : 31.11111111111111
```

نتیجه ی حالت پنجم با حذف stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 160
درصد بازدهی این مدل : 35.55555555555556
```

حالت ششم :

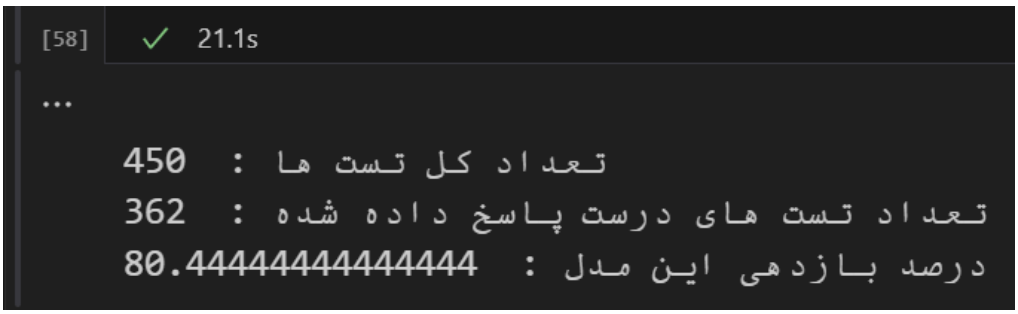
استفاده از **additive smooth** (با گرفتن لوگاریتم)

نهایتا این حالت از همه مدل ها بهتر عمل خواهد کرد زیرا لوگاریتم گرفتن خودش مشکلات مربوط به نزدیک صفر شدن احتمال ها را حل میکند و **smooth additive** هم مشکلات مربوط به اعدادی که در سطر مرد بررسی در **BOW** نیستند را حل می کند و بین روش های موجود، بهترین روش همین است.(که حالا با حذف **stop words** و یا ریشه یابی تا حدی هم میتوان آنرا بیشتر بهبود داد)

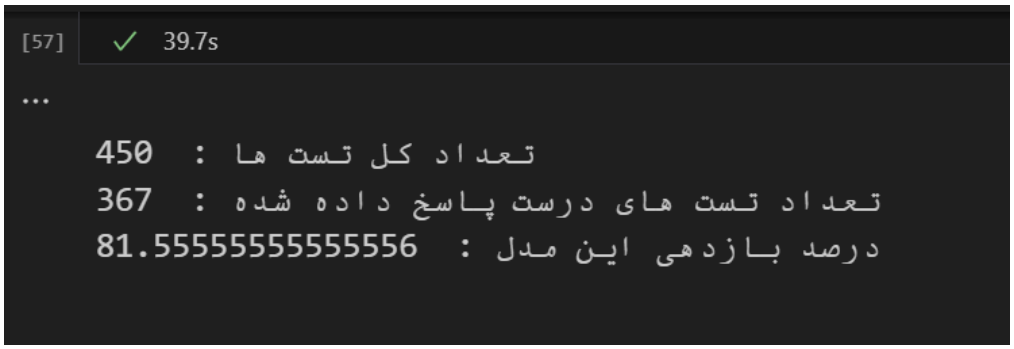
```
def calculateCategorylikelihood(bookTitle, bookDescription, categoryDict):
```

```
    likelyhood = np.log(1)
    wordsCount = countAllWords(categoryDict)
    for word in bookDescription:
        if word in categoryDict.keys():
            likelyhood += np.log(categoryDict[word])
        else:
            likelyhood += np.log(ALPHA/(wordsCount +
                                (ALPHA * len(categoryDict))))
     for word in bookTitle:
         if word in categoryDict.keys():
             likelyhood += np.log(categoryDict[word])
         else:
             likelyhood += np.log(ALPHA/(wordsCount +
                                    (ALPHA * len(categoryDict))))
     return likelyhood
```

نتیجه ی حالت ششم بدون حذف stop words**:**



نتیجه ی حالت ششم با حذف stop words**:**



با اینکه حالت ششم بدون حذف **stop words** از لحاظ تناسب کارایی و زمان اجرا در کل بهترین حالت است، اما با انجام ریشه یابی، از لحاظ صرفا بازدهی، به بهترین حالت ممکن میرسیم

در باب stemming و lemmatization

برای ریشه‌یابیِ کلمات معمولاً از دو روشِ **Stemming** و **Lemmatization** استفاده می‌شود که هر دو روش قادر هستند ریشه‌ی یک کلمه را به دست بیاورند. برای مثال کلمه‌ای مانندِ "رفتن" را تصور کنید که ممکن است در جمله‌های مختلف به شکل‌های گوناگون ظاهر شود. برای مثال: رفتم/ برویم/ می‌رفتم/ بروند. تمامیِ فعل‌های "رفتن" در این جملات را می‌توان به کلمه‌ی " رفت " نگاشت کرد. با این کار می‌توان ۴ویژگی (رفتم، برویم، می‌رفتم، بروند) را به یک ویژگی (رفت) نگاشت کرد و با این عمل کاهش ویژگی (کاهش ابعاد) نیز انجام می‌شود. عمل **Lemmatization** نیز می‌تواند توسط روش‌هایی انجام شود. در این عمل نیاز است که از یک فرهنگ لغت یا چیزی شبیه به آن برای به دست آوردنِ ریشه‌ی لغات استفاده شود، چون عموماً روش‌های **Lemmatization** به صورت با قاعده نیستند.

حالت هفتم :

با **stemming**: در **stemming** ما عمل ریشه یابی را انجام میدهیم و در هضم هم این عمل خیلی دقیق انجام نمیشود چون در این عمل فقط چند کاراکتر اخر کلمه را حذف میکند به خیال اینکه به ریشه برسد که اغلب منجر به املا و معنای نادرست میشود و حتی در افعالی از یک نوع هم این عمل به خوبی انجام نمیشود:

```
from hazm import *
print(Stemmer().stem("رفتم"),Stemmer().stem("رفتی"),Stemmer().stem("رفت"),
      Stemmer().stem("رفتیم"),Stemmer().stem("رفتید"),Stemmer().stem("رفتند"))
✓ 0.0s
رف رفت رف رفت رفتید رفتند
```

یا حتی کلماتی که ظاهری شبیه فعل دارند را خراب میکند:

```
from hazm import *
print(Stemmer().stem("تابستان"),Stemmer().stem("زمستان"),Stemmer().stem("نسیم"),)
✓ 0.0s
تابس زمس نس
```

به هر حال اگر بخواهیم از **stemming** استفاده کنیم براحتی با **apply**اینکار انجام میشود و توابع پیش پردازشمان اینگونه می شود

```
def preprocess_a_cell(txt):
    punctuationMarks = [':', '،', '?', '؟', '.', ';',
                        '(', ')', '!', '+', '-', '*', '/',
                        '=', '\\', '»', '«', '"', '_', '-', '...']
    stopWords = pd.read_csv("sw.csv")
```

```
    wordsDf = pd.DataFrame(word_tokenize(txt), columns=["word"])
    wordsDf["word"] = wordsDf[wordsDf["word"].apply(
        lambda w: w not in punctuationMarks and
        w not in stopWords["word"].values and
        w.isdigit() == False)]
    wordsDf = wordsDf.dropna()
    wordsDf["word"] = wordsDf["word"].apply(Stemmer().stem)
    return wordsDf
def preprocess_a_test_cell(txt):
    wordsDf = pd.DataFrame(word_tokenize(txt), columns=["word"])
    wordsDf["word"] = wordsDf["word"].apply(Stemmer().stem)
    return wordsDf
```

نتیجه ی حالت هفتم بدون additive smooth **و با حذف**stop words **:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 139
درصد بازدهی این مدل : 30.88888888888889
```

نتیجه ی حالت هفتم با additive smooth **و با حذف** stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 313
درصد بازدهی این مدل : 69.5555555555556
```

نتیجه ی حالت هفتم بدون additive smooth **و با** stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 133
درصد بازدهی این مدل : 29.5555555555557
```

نتیجه ی حالت هفتم با additive smooth **و با** stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 361
درصد بازدهی این مدل : 80.222222222222
```

تفسیر: حالت بدون **additive smooth** *را اگر* **stemming** *هم نکنیم، همان حالت سوم میشود و حالت با* **additive smooth** *هم همان حالت ششم میشود که همانطور که گفتیم بدلیل ضعیف بودن عملیات* **stemming** *در ریشه یابی، نه تنها دقتمان زیاد نشده بلکه مقداری کمتر هم شده است. و در بین این حالات بهترین کار این بود که* **stop words** *را حذف نکرده و* **stemming** *کنیم که احتمالا بخاطر این است که اگر هم* **stop words** *و هم* **stemming** *کنیم، تعداد زیادی از کلمات حذف شده و یا نصفه میشوند که در نهایت چون تعداد داده های* **train** *هم خیلی نیستند، باعث نتایج اشتباه میشوند و استفاده از* **stemming** *خالی بهترین کار است*

حالت هشتم :

با **lemmatizing**، *در* **lemmatizing** *عمل بن واژه یابی را انجام میدهیم یعنی اینکار حرفه ای تر و پیچیده تر از* **stemming** *است و نیاز به پردازش هایی مثل* **pos tagging** *دارد و بر اساس نوع کلمه و جایی که بکار رفته است به یافتن ریشه می پردازیم.*

این عملیات در هضم هم بخوبی انجام میشود اما بدلیل پیچیده بودن، سرعت اجرا را بسیار کند می کند و البته که این روش هم مشکلاتی دارد و مثلا در هضم کلماتی که واقعا شبیه به هم اند و از یک خانواده هستند را هم از هم جدا می گیرد و تعداد بسیاری از کلمات را تحلیل نمیکند

```
from hazm import *
print(Lemmatizer().lemmatize("رفتم"),Lemmatizer().lemmatize("رفتن"),
      Lemmatizer().lemmatize("روان"),Lemmatizer().lemmatize("رونده"),)
```

✓ 1.9s

```
رفت#رو رفتن روان رونده
```

در کل برای **lemmatizing** *با هضم تابع های پیش پردازش را به این شکل تغییر میدهیم:*

```
def preprocess_a_cell(txt):
    punctuationMarks = [':', '،', '?', '؟', '.', ';',
                        '(', ')', '!', '+', '-', '*', '/',
                        '=', '\\', '»', '«', '"', '_', '-', '...']
    stopWords = pd.read_csv("sw.csv")

    wordsDf = pd.DataFrame(word_tokenize(txt), columns=["word"])
    wordsDf["word"] = wordsDf[wordsDf["word"].apply(
        lambda w: w not in punctuationMarks and
        w.isdigit() == False)]
    wordsDf = wordsDf.dropna()
    wordsDf["word"]=wordsDf["word"].apply(Lemmatizer().lemmatize)
    return wordsDf
def preprocess_a_test_cell(txt):
    wordsDf = pd.DataFrame(word_tokenize(txt), columns=["word"])
    wordsDf["word"]= wordsDf["word"].apply(Lemmatizer().lemmatize)
    return wordsDf
```

نتیجه ی حالت هشتم بدون additive smooth **و بدون حذف**stop words **:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 130
درصد بازدهی این مدل : 28.888888888888
```

نتیجه ی حالت هشتم با additive smooth **و با حذف** stop words**:**

```
تعداد کل تست ها : 450
تعداد تست های درست پاسخ داده شده : 368
درصد بازدهی این مدل : 81.77777777777778
```

تفسیر: نهایتا درست است **lemmatizing** *زمان ران شدن را بالا میبرد اما دقت را مقدار اندکی بیشتر میکند در حالت دارای* **additive smooth** *اما اگر* **additive smooth***نباشد انقدر خطا زیاد است و مقدار* **train** *کم است که نتایج دارای منطق خاصی نیستند و استفاده از ریشه یابی هم کمک زیادی به ماجرا نکرده و حتی بازدهی را کمتر میکند.*

نتیجه گیری

در کل از لحاظ مجموعه ی بازدهی و زمان اجرا، حالت ششم از همه بهتر بود که جهت تحلیل دقیقتر، احتمال هر کدام از دسته بندی ها را حساب کردم و در فایل txt **ای بصورت زیر گذاشتم**