

بنام خداوند جان و خرد



دانشگاه تهران

دانشکده فنی

دانشکده مهندسی برق و کامپیوتر

# سامانه‌های پادگیری ماشین توزیع شده

## تمرین کامپیوتري ۳

نام و نام خانوادگی: علی خرم فر

شماره دانشجویی: ۸۱۰۱۰۲۱۲۹

دیماه ۱۴۰۳

## فهرست مطالب

۱	۱	۱ _ پاسخ سوال شماره ۱
۱	۱-۱	۱-۱ _ تعداد ابیات و کلمات
۲	۱-۲	۱-۲ _ پر تکرار ترین قافیه ها
۳	۱-۳	۱-۳ _ تعداد gram های شاهنامه
۴	۲	۲ _ پاسخ سوال دوم
۴	۲-۱	۲-۱ _ پاسخ قسمت الف
۷		حلقه آموزش مدل
۸	۲-۲	۲-۲ _ استخراج بردارها و محاسبه شباهت کلمات
۸	۲-۳	۲-۳ _ نمایش شباهت دو به دو کلمات با استفاده از Heatmap
۹	۲-۴	۲-۴ _ کاهش ابعاد بردارها به دو بعد با PCA و نمایش نمودار
۱۱	۳	۳ _ پاسخ سوال سوم
۱۱	۳-۱	۳-۱ _ آپلود مجموعه داده در HDFS
۱۱	۳-۲	۳-۲ _ محاسبه اطلاعات آماری درآمد سالانه
۱۲	۳-۳	۳-۳ _ طبقه بندی با روش K-Means
۱۲	۳-۴	۳-۴ _ نمایش نتایج خوش بندی
۱۳		زمان بندی اجرا
۱۳	۴	۴ _ استفاده از ابزارها
۱۳	۴-۱	۴-۱ _ لیست پرامپت ها

## فهرست جداول

۳	جدول ۱ - ۱۰ قافیه پر تکرار در شاهنامه
---	---------------------------------------

## فهرست اشکال

۹	شکل ۱-۲ نمودار Heatmap شباهت بین جفت کلمه ها
۱۰	شکل ۲-۲ نمودار Scatter بردارهای امبدینگ کاهش یافته به ۲ بعد با PCA
۱۲	شکل ۳-۱ نتایج خوش بندی مشتریان

# ۱\_پاسخ سوال شماره ۱

## ۱-۱\_تعداد ابیات و کلمات

برای پاسخ به این سوال و سوال دوم از محیط Kaggle استفاده شد. ابتدا کتابخانه PySpark نصب

شد:

```
!pip install pyspark
```

ابتدا نشست Spark را با استفاده از کلاس `SparkSession` ایجاد کردیم. این اولین قدم برای استفاده از امکانات PySpark است. در اینجا نام app را به عنوان "Shahname" تعیین کردیم:

```
spark = SparkSession\  
. builder\  
. appName("Shahname") \  
. getOrCreate()
```

متن شاهنامه که در قالب یک فایل متنی از مسیر مشخص شده (`input_path`) که به دیتاست آپلود شده کگل اشاره دارد، خوانده شد. این متن به شکل یک RDD بارگذاری شد.

```
input_path = "/kaggle/input/dmels-ca3/shahname.txt"  
lines = spark.read.text(input_path).rdd.map(lambda r: r[0])
```

هر خط از متن به عنوان یک رکورد در این RDD در نظر گرفته می‌شود.

از آنجایی که هر بیت در شاهنامه معمولاً در یک خط نوشته شده، تعداد خطوط متن برابر با تعداد ابیات در نظر می‌گیریم. برای محاسبه این مقدار، از متدهای `count()` استفاده کردیم.

```
total_verses = lines.count()
```

برای محاسبه تعداد کل کلمات و کلمات یکتا ابتدا تمام کلمات از هر خط جدا شدند (با استفاده از `reduceByKey` و جداسازی با فاصله) سپس، به هر کلمه مقدار ۱ نسبت داده شد و با استفاده از `flatMap` تعداد تکرار هر کلمه محاسبه گردید.

```
counts = lines.flatMap(lambda x: x.split()) \  
. map(lambda x: (x, 1)) \  
. reduceByKey(add)  
#Count words  
total_words = counts.map(lambda x: x[1]).sum()  
unique_words = counts.count()
```

تعداد کل کلمات با جمع کردن تعداد تکرارهای تمام کلمات محاسبه شد. برای کلمات یکتا هم، مجموعه‌ای از تمام کلمات بدون تکرار تشکیل دادیم و تعداد کل عناصر این مجموعه محاسبه شد

برای بررسی کارایی اجرای کد، مدت زمان کل محاسبه از شروع تا پایان برنامه ثبت شد که این کد را در تمام قسمت‌ها استفاده کردیم.

در نهایت، Spark بسته شد تا منابع آزاد شوند.

```
spark.stop()
```

تعداد کل ابیات: ۵۱۵۸۰

تعداد کل کلمات: ۵۷۰۸۴۹

تعداد کلمات یکتا: ۱۸۱۰۳

مدت زمان اجرا: ۲.۸۷ ثانیه

## ۲-۱\_ پر تکرارترین قافیه‌ها

قافیه‌ها آخرین کلمه یا عبارت هر بیت هستند. در ادامه فقط موارد مربوط به همین بخش را گزارش می‌کنیم و از توضیح موارد تکراری صرف نظر می‌کنیم.

ابتدا، آخرین کلمه هر خط به عنوان قافیه استخراج شد. این کار با استفاده از تابع extract\_rhyme انجام شد که برای هر خط آخرین کلمه را خروجی می‌دهد.

```
def extract_rhyme(line):
    if line.strip():
        words = line.split()
        return words[-1] # last word
    return None

rhymes = lines.map(extract_rhyme)
```

برای حذف داده‌های خالی، قافیه‌هایی که مقدار None داشتند در نظر نمی‌گیریم. با استفاده از reduceByKey، تعداد تکرار هر قافیه محاسبه شد.

```
counts = rhymes.filter(lambda rhyme: rhyme is not None) \
    .map(lambda rhyme: (rhyme, 1)) \
    .reduceByKey(add)
```

برای یافتن ۱۰ قافیه پر تکرار با استفاده از متدهای takeOrdered و reduceByKey، به همراه تعداد تکرارشان آنها را استخراج کردیم.

```
top_rhymes = counts.takeOrdered(10, key=lambda x: -x[1])
```

کل کد در ۲.۷۲ ثانیه انجام شد.

تعداد تکرار	قافیه
۸۸۱	بود
۶۳۲	سپاه
۵۱۸	راه
۴۶۳	شاه
۴۲۳	اوی
۴۱۲	کرد
۳۸۵	را
۳۸۱	روی
۳۶۳	شد
۳۳۸	زمین

جدول ۱ - قافیه پر تکرار در شاهنامه

### ۱-۳ \_تعداد ۳-gram های شاهنامه

ابتدا هر خط از متن شاهنامه به دنباله های سه کلمه ای تقسیم شد. برای این کار، تابع map\_to\_trigrams پیاده سازی شد که از هر خط مجموعه ای از ۳-gramها استخراج می کند.

```
def map_to_trigrams(line):
    words = line.split()
    trigrams []=
    if len(words) < 3 :
        return trigrams
    for i in range(len(words) - 2) :
        trigram = words[i] + " " + words[i+1] + " " + words[i+2]
        trigrams.append((trigram, 1))
    return trigrams
```

تغییر در اینجا باعث تغییر در جواب نهایی می شد. اگر که فقط مصوع ها را در نظر می گرفتیم تعداد نهایی کمتر می شد. یا شرایط دیگری برقرار می کردیم تعداد بیشتر می شد. تعداد فعلی یک حالت متوسط داشت و کد نهایی را به همین حالت گذاشتیم.

با استفاده از flatMap تمامی ۳-gramها از متن استخراج و سپس با reduceByKey تعداد تکرار هر ۳-gram محاسبه شد. همچنین تعداد کل ۳-gram های موجود در متن شاهنامه نیز محاسبه گردید:

```
trigram_counts = lines.flatMap(map_to_trigrams) \
    .reduceByKey(add)
total_trigrams = trigram_counts.count()
```

در نهایت با استفاده از متدها `takeOrdered` ۱۰ عدد از پر تکرارترین ۳-gram‌ها به همراه تعداد تکرار آن‌ها خروجی گرفتیم:

```
top_trigrams = trigram_counts.takeOrdered(10, key=lambda x: -x[1])
```

کل کد در ۴.۶۸ ثانیه انجام شد.

تعداد کل ۳-gram‌ها: ۲۶۳۰۶۰

تعداد تکرار	3-gram
۳۹۰	چنین داد پاسخ
۲۸۳	داد پاسخ که
۱۸۸	چنین گفت با
۱۸۵	چنین گفت کای
۱۶۵	مرا او را
۱۵۵	بدو گفت کای
۱۴۵	تاج و تخت
۱۲۶	ز هر سو
۱۲۴	نشست از بر
۱۱۴	فروند آمد از

## ۲\_پاسخ سوال دوم

### ۲-۱\_پاسخ قسمت الف

این سوال چالش برانگیزترین سوال این تمرین بود. حدود ۲ روز کامل در تلاش بودم که به پاسخ ببهنه برسم که به طور کامل از قابلیتهای Spark هم استفاده کنم. پاسخ فعلی بهترین پاسخی است که توانستم دریافت کنم. هرچند پاسخ‌های بهتری دریافت شد ولی آن‌ها به طور دقیق توزیع شده نبودند و Collect در حلقه آموزش استفاده می‌شد. پس در نهایت پیاده‌سازی فعلی به عنوان پاسخ قرار دادم که اگرچه پاسخ ضعیفتر بود ولی اصولی تر است. همچنین در پیاده‌سازی کتابخانه از Negative Sampling استفاده نشده بود. در پیاده‌سازی نهایی وقتی که Negative Sampling را اضافه می‌کردم نه تنها زمان محاسبه بیشتر می‌شد پاسخ نیز ضعیفتر بود.

مانند سوال قبل ابتدا کتابخانه‌های مورد نیاز نصب شد. در اینجا یک کتابخانه arabic-reshaper هم نصب شد که بتوانیم کلمات فارسی را بدون مشکل در نمودارها نمایش دهیم.

مانند قسمت قبل ابتدا نشست SparkSession با استفاده از Spark ایجاد شد و شاهنامه را بارگذاری

کردیم:

```
spark = SparkSession\  
builder\  
.appName("word2vec")\  
.getOrCreate()  
  
input_path = "/kaggle/input/dmles-ca3/shahname.txt"  
lines = spark.read.text(input_path).rdd.map(lambda row: row[0])
```

هر خط از شاهنامه که به صورت بیت شعر است، به یک لیست از کلمات شکسته شد. برای این کار از split() استفاده شد. همچنین، خطوطی که کمتر از ۲ کلمه داشتند، حذف شدند.

```
sentences = lines.map(lambda line: line.split()).filter(lambda words: len(words) > 1)
```

در Skip-gram، نیاز به ایجاد جفت‌های کلمه هدف و کلمه context داریم. خطوط کوتاه با کمتر از ۲ کلمه نمی‌توانند جفت‌های خوبی ایجاد کنند، بنابراین حذف می‌شوند. در این روش، یک کلمه مرکزی انتخاب شده و کلمات اطراف آن یا کلمات زمینه پیش‌بینی می‌شوند. هدف این است که مدل یاد بگیرد کلمات در چه زمینه‌ای Context‌ی بیشتر به کار می‌روند.

```
def generate_pairs(sentence, window_size=2):  
    pairs[] =  
    for i, word in enumerate(sentence):  
        start = max(0, i - window_size)  
        end = min(len(sentence), i + window_size + 1)  
        for j in range(start, end):  
            if i != j:  
                pairs.append((word, sentence[j]))  
    return pairs  
  
skip_grams = sentences.flatMap(lambda sentence: generate_pairs(sentence))
```

ابتدا تابع generate\_pairs را تعریف کردیم که از یک جمله که در اصل لیستی از کلمات است، جفت‌های کلمه Context تولید می‌کند. در این تابع، به هر کلمه، کلمات اطرافش در یک پنجره مشخص اختصاص داده می‌شوند.

تعداد کلماتی که به عنوان زمینه برای هر کلمه مرکزی در نظر گرفته می‌شود ک این مقدار را ۲ در نظر گرفتیم. برای هر کلمه، از ایجاد جفت (word, word) خودداری کردیم. سپس رای FLAT تولید جفت‌ها از متده استفاده کردیم. این متده لیست جفت‌های تولید شده برای هر جمله را کرده و به یک لیست کلی از تمام جفت‌ها تبدیل می‌کند.

پس در این مرحله جفت‌های کلمه-زمینه با استفاده از تابع generate\_pairs ایجاد شدند. در مرحله بعدی هدف ایجاد واژگان متنی شاهنامه و نگاشت هر کلمه به یک عدد منحصر به فرد است. ابتدا، تمام کلمات

موجود در جفتهای استخراج و با استفاده از skip\_grams یکتا شدند. این کار تضمین می‌کند که هر کلمه فقط یک بار در واژگان وجود داشته باشد:

```
vocab = skip_grams.flatMap(lambda pair: pair).distinct().zipWithIndex()
```

به هر کلمه یک شاخص عددی اختصاص می‌دهد که از صفر شروع می‌شود. در نهایت از متD برای تبدیل RDD به یک دیکشنری استفاده شد که در آن، هر کلمه به شاخص اختصاص داده شده MAP می‌شود:

```
vocab_dict = vocab.collectAsMap()
```

تعداد کل کلمات موجود در واژگان و چند نمونه از کلمات:

```
Number of words: 18103
Samples[(١٤, 'برتر'), (١٠, 'اندیشه'), (١١, 'جای'), (١٢, 'رهنمای'), (٣, 'گردان')]:
```

پس هر کلمه به یک index عددی منحصر به فرد نگاشت شد که برای آموزش مدل Word2Vec آماده است. گام بعدی، آماده سازی داده ها برای ورودی مدل و شروع فرآیند آموزش است.

ابتدا چند پارامتر برای مدل Word2Vec تعریف شدنD:

```
vector_size = 100
learning_rate = 0.001
vocab_size = len(vocab_dict)
num_iterations = 5
window_size = 2
```

:vector\_size اندازه هر بردار که هر کلمه در چند بعد نمایش داده شود.

:learning\_rate نرخ یادگیری. حالت های مختلفی اجرا شد در نهایت حس می کنم ۰.۰۲۵ بهتر از همه بود ولی خب نتیجه گزارش بر اساس ۰.۰۰۱ بود.

:num\_iterations تعداد دفعاتی که مدل کل داده ها در حلقه آموزش می بیند.

:window\_size تعداد کلمات به عنوان زمینه در اطراف کلمه مرکزی

دو ماتریس مجزا برای ذخیره بردارهای کلمه و بردارهای زمینه تعریف شدند. این ماتریس ها به صورت تصادفی با مقادیر در بازه منفی تا مثبت ۰.۵ مقداردهی شدند:

```
word_vectors = np.random.uniform(-0.5, 0.5, (vocab_size, vector_size))
context_vectors = np.random.uniform(-0.5, 0.5, (vocab_size, vector_size))
```

این مقداردهی اولیه خیلی مهم بود. در بعضی اجراهای که از توزیع نرمال استفاده می‌شد نتیجه حتی بدون آموزش زیاد هم خوب بود ولی خب اصولی نبود.

:word\_vectors بردارهای کلمه که هر سطر نشان‌دهنده یک کلمه در فضای امبدینگ است.

:context\_vectors بردارهای زمینه که برای پیش‌بینی کلمات زمینه از کلمه مرکزی استفاده می‌شوند.

برای دسترسی به vocab\_dict در تمام گره‌های Spark، از قابلیت broadcast استفاده شد. این کار باعث می‌شود که واژگان به هر گره ارسال شود. این خط کد مربوط به کتابخانه الهام گرفته شد 😊

```
vocab_broadcast: Broadcast = spark.sparkContext.broadcast(vocab_dict)
```

## حلقه آموزش مدل

این حلقه برای هر جفت کلمه-زمینه، خطاب و به روزرسانی بردارهای کلمه و زمینه را محاسبه می‌کند:

```
def train_skip_gram(index_pair):
    global word_vectors, context_vectors
    word_idx, context_idx = index_pair

    word_vector = word_vectors[word_idx]
    context_vector = context_vectors[context_idx]

    dot_product = np.dot(word_vector, context_vector)
    prediction = sigmoid(dot_product)

    error = 1 - prediction
    gradient = learning_rate * error

    word_vector_update = gradient * context_vector
    context_vector_update = gradient * word_vector

    #Regularization
    word_vector_update -= 0.001 * word_vector
    context_vector_update -= 0.001 * context_vector

    return (word_idx, word_vector_update), (context_idx, context_vector_update)

for iteration in range(num_iterations):
    print(f"Starting iteration {iteration + 1}/{num_iterations}")

    index_pairs = skip_grams.map(
        lambda pair:
        vocab_broadcast.value[pair[0]],
        vocab_broadcast.value[pair[1]]
    )

    updates = index_pairs.map(train_skip_gram).flatMap(lambda x: x)

    aggregated_updates = updates.reduceByKey(lambda a, b: a + b).collect()

    for idx, update in aggregated_updates:
        if idx < vocab_size:
            word_vectors[idx] += update
            word_vectors[idx] = normalize(word_vectors[idx])
        else:
            context_vectors[idx - vocab_size] += update
```

```

context_vectors[idx - vocab_size] = normalize(context_vectors[idx - vocab_size])
learning_rate *= 0.95
print("Training complete!")

```

در حلقه آموزش، مدل طی چند iteration بر روی تمام داده‌ها آموزش می‌بیند در ابتدای هر تکرار، شمارش فعلی را پرینت می‌کنیم.

در ابتدا جفت‌های کلمه-زمینه به index عددی آن‌ها تبدیل می‌شوند سپس برای هر جفت،تابع train\_skip\_gram فراخوانی شده و به روزرسانی‌های لازم محاسبه می‌شود. به روزرسانی‌های بردارها با استفاده از reduceByKey برای هر Index جمع شده و هر بردار کلمه و زمینه با توجه به مقادیر تجمعی شده به روزرسانی می‌شود.

در پایان هر تکرار، نرخ یادگیری کاهش دادیم تا مدل بهتر آموزش ببیند. نرخ یادگیری روی نسخه آخر مدل خیلی تاثیر داشت و تغییر آن باعث می‌شد برخی اوقات مدل نتیجه خوبی نداشته باشد و واگرا شود.

## ۲-۲\_ استخراج بردارها و محاسبه شباهت کلمات

در این مرحله، هدف استخراج بردارهای معنایی یا embedding کلمات خاص از مدل Word2Vec و بررسی شباهت معنایی بین آن‌ها با استفاده از شباهت کسینوسی یا Cosine Similarity است. ابتدا کلمات مورد نظر برای استخراج بردارها تعریف شدند. سپس، برای هر کلمه، اگر در واژگان مدل وجود داشت، بردار معنایی متناظر آن استخراج شد:

```

target_words = ["رستم", "سهراب", "اسفنديار", "رخش", "زال"]
embeddings{} =
for word in target_words:
    if word in vocab_dict:
        embeddings[word] = word_vectors[vocab_dict[word]]

```

## ۲-۳\_ نمایش شباهت دو به دو کلمات با استفاده از Heatmap

در این مرحله هدف ما محاسبه میزان شباهت دو به دو بین کلمات و نمایش آن به صورت بصری در قالب یک نمودار Heatmap است. برای محاسبه شباهت دو به دو بین کلمات، یک ماتریس مربعی ایجاد کردیم. که کلمه i و زبهاتشان در آن ثبت شود.

```

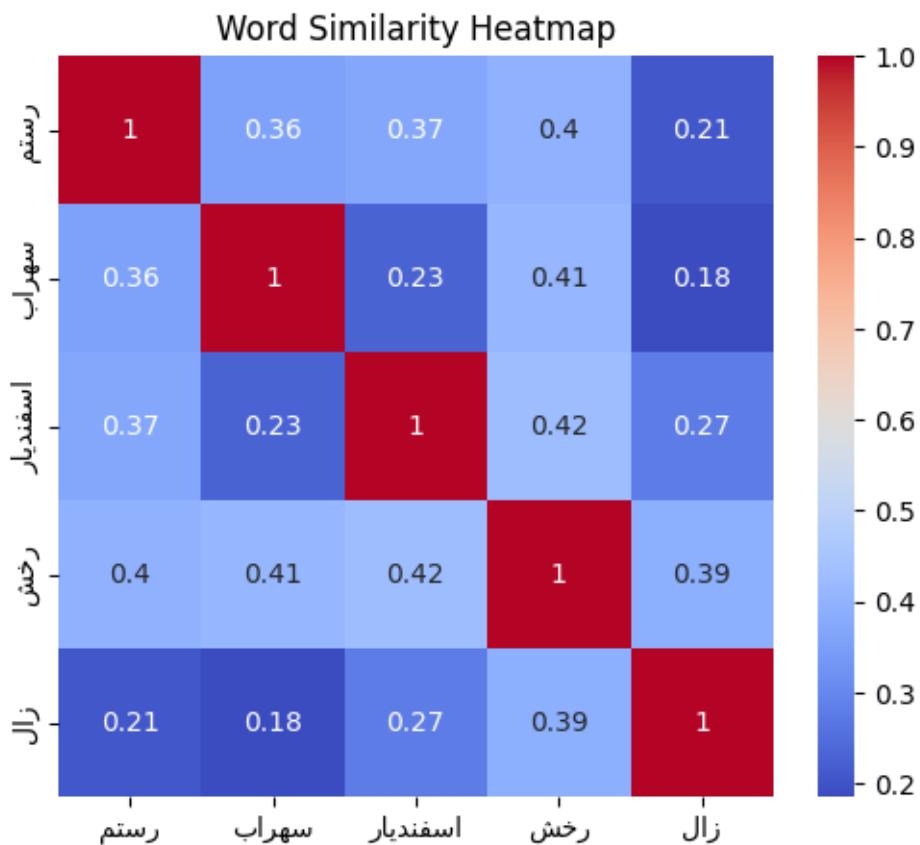
words = list(embeddings.keys())
similarity_matrix = np.zeros((len(words), len(words)))

for i, w1 in enumerate(words):
    for j, w2 in enumerate(words):
        similarity_matrix[i][j] = cosine_sim(embeddings[w1], embeddings[w2])

```

برای نمایش صحیح کلمات فارسی در نمودار، از کتابخانه‌های arabic\_reshaper و get\_display استفاده شد.

نمودار صفحه بعد نشان می‌دهد که شباهت کسینوسی بین هر دو کلمه محاسبه شده است. مقادیر روی قطر اصلی برابر با ۱ هستند، زیرا هر کلمه با خودش مشابه است. رستم و رخش شباهت بالایی دارند که به دلیل ارتباط نزدیک این دو شخصیت در داستان شاهنامه است. شباهت رستم و سهراب نیز به دلیل روابط پدر فرزندی قابل توجه است. زال و رخش نسبت به سایر کلمات شباهت کمتری دارند.



شکل ۲-۱ مودار Heatmap شباهت بین جفت کلمه‌ها

## ۴-۲\_ کاهش ابعاد بردارها به دو بعد با PCA و نمایش نمودار

این کاهش ابعاد به ما کمک می‌کند که توزیع و شباهت بردارهای معنایی را به صورت بصری در کنیم.

ابتدا، بردارهای کلمات در قالب ردیف‌هایی برای یک DataFrame ایجاد شدند:

```
embedding_rows = [Row(word=word, features=Vectors.dense(embeddings[word])) for word in target_words]
embedding_df = spark.createDataFrame(embedding_rows)
```

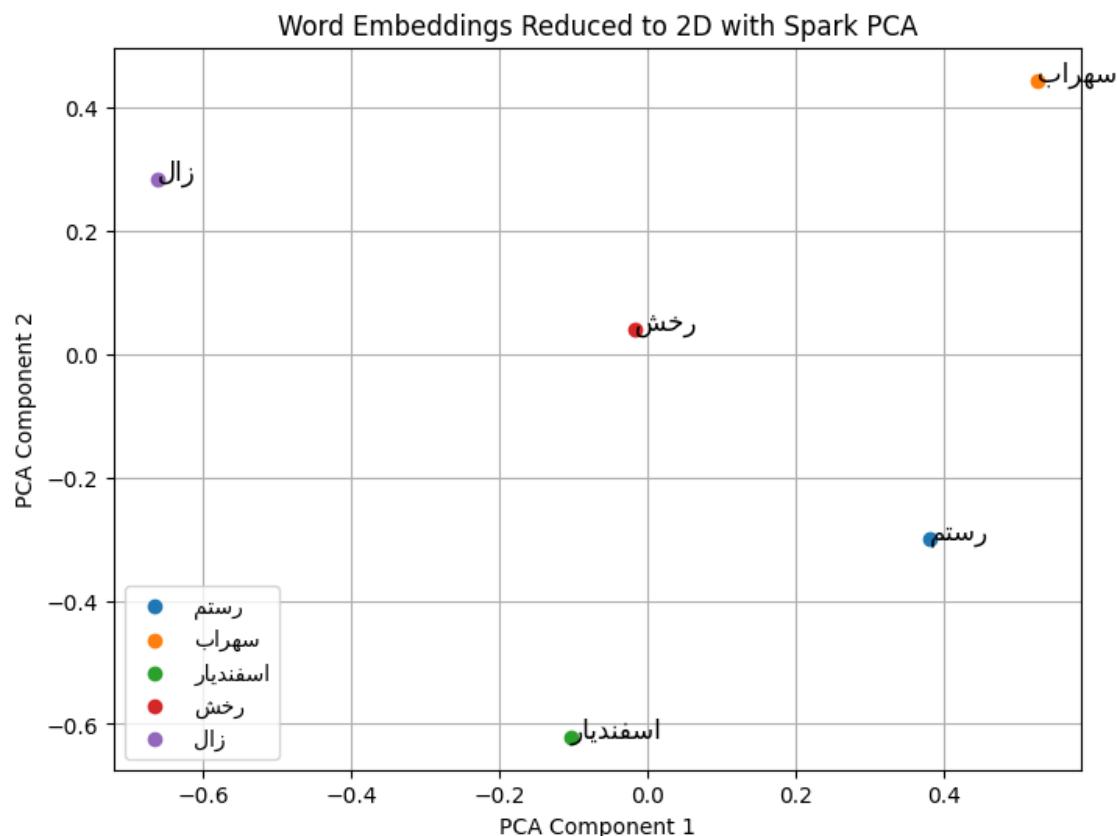
هر کلمه به همراه بردار معنایی اش به یک ردیف در DataFrame تبدیل می‌شود.

بردار معنایی کلمات است که به صورت بردار dense ذخیره می‌شود.

برای PCA گفته نشده بود که از کتابخانه استفاده کنیم یانه پس با استفاده از کلاس PCA در PySpark، ابعاد بردارهای کلمه به دو مؤلفه اصلی کاهش یافتند. این دو مؤلفه بیشترین اطلاعات ممکن را از بردارهای اصلی حفظ می‌کنند. میتوانستیم از کتابخانه Sklearn هم استفاده کنیم ولی خوب نتیجه بررسی شد و مشابه بود.

```
pca = PCA(k=2, inputCol="features", outputCol="pca_features")
pca_model = pca.fit(embedding_df)
pca_result = pca_model.transform(embedding_df)
```

در نهایت با استفاده از کتابخانه Matplotlib، نمودار دو بعدی برای نمایش کلمات بر اساس مختصات کاهش یافته ایجاد شد:



شکل ۲-۲ نمودار Scatter بردارهای امبدینگ کاهش یافته به ۲ بعد با PCA

کلماتی که به صورت نزدیک به هم نمایش داده مانند رستم و رخشن از لحاظ معنایی در فضای برداری مدل شباهت بیشتری دارند.

کلماتی که فاصله بیشتری دارند مانند زال و اسفندیار ارتباط معنایی کمتری دارند.

## ۳- پاسخ سوال سوم

### ۱- آپلود مجموعه داده در HDFS

هدف این بخش ایجاد یک پوشه در سیستم توزیع شده HDFS و بارگذاری مجموعه داده مربوط به مشتریان فروشگاه در این پوشه است. ابتدا یک پوشه با نام شماره دانشجویی خود در HDFS ایجاد کردیم. این کار با دستور زیر انجام شد:

```
hdfs dfs -mkdir /810102129
```

سپس فایل داده‌های مشتریان (customers.csv) به پوشه‌ای که در HDFS ایجاد کردیم منتقل شد:

```
hdfs dfs -put customers.csv /810102129
```

پس از انتقال فایل، برای اطمینان از موفقیت‌آمیز بودن فرآیند، لیست فایل‌های موجود در پوشه بررسی شد:

```
hdfs dfs -ls /810102129
```

خروجی:

```
khorramfar@raspberrypi-dml0:~/CA3 $ hdfs dfs -ls /810102129
Found 1 items
-rw-r--r-- 1 khorramfar supergroup 3981 2024-12-27 06:41 /810102129/customers.csv
```

### ۲- محاسبه اطلاعات آماری درآمد سالانه

مانند قسمت‌های قبل ابتدا Session ایجاد شد:

```
spark = SparkSession\
    .builder\
    .appName("KMeans-Khorramfar")\
    .getOrCreate()

file_path = "hdfs://raspberrypi-dml0:9000/810102129/customers.csv"
data = spark.read.csv(file_path, header=True, inferSchema=True)
```

با استفاده از Spark SQL، آمار توصیفی شامل میانگین، حداقل، حداکثر و واریانس درآمد سالانه

مشتریان محاسبه شد:

```
stats = data.select(
    mean("Annual Income (k$)").alias("Mean"),
    min("Annual Income (k$)").alias("Min"),
    max("Annual Income (k$)").alias("Max"),
    variance("Annual Income (k$)").alias("Variance")
)

() stats.show
```

خروجی با

واریانس	ماکزیمم	مینیمم	میانگین
689.8355778894471	137	15	60.56

### ۳-۳\_ طبقه‌بندی با روش K-Means

مقدار K در سوال مشخص نشده ولی با توجه به خروجی‌های مختلف عدد ۵ در نظر گرفته شد.

ابتدا ستون‌های مربوط به درآمد سالانه و spending score یا امتیاز خرچ کردن مشتری به بردار ویژگی تبدیل شدند:

```
assembler = VectorAssembler()
inputCols=["Annual Income (k$)", "Spending Score (1-100)"], outputCol="features"
(
```

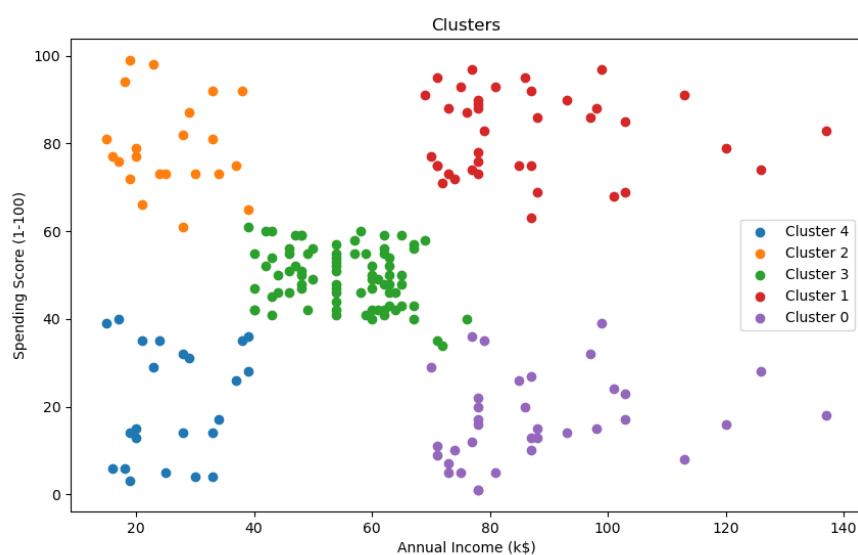
سپس مدل K-Means با تعداد k برابر ۵ و تصادفی برابر ۱ آموزش داده شد.

```
kmeans = KMeans().setK(5).setSeed(1)
model = kmeans.fit(dataset)
```

predictions = model.transform(dataset)  
predictions شامل داده‌های اصلی همراه با ستون جدیدی به نام prediction است که کلاستر تخصیص یافته به هر مشتری را نشان می‌دهد.

### ۳-۴\_ نمایش نتایج خوشه‌بندی

برای نمایش نتایج، از Matplotlib استفاده شد. داده‌های طبقه‌بندی شده به صورت نمودار پراکندگی با رنگ‌های مختلف برای هر خوشه رسم شد و فایل تصویر هم ذخیره شد:



شکل ۳-۱ نتایج خوشه‌بندی مشتریان

## زمان اجرای برنامه

مدت زمان اجرای کد توسط کتابخانه : ۱۰۲۰۳ ثانیه. یا ۱.۷۰۰۵ دقیقه

مدت زمان ثبت شده در Spark Master ۱.۶ دقیقه.

کد با دستور زیر اجرا شد:

```
khorramfar@raspberrypi-dml0:~/CA3 $ spark-submit k-means.py 4
```

## ۴\_استفاده از ابزارها

برای این تمرین هم از چت بات و هم از ابزار پاکنویس برای رعایت علائم نگارشی، تصحیح متن و رعایت نیمه فاصله استفاده شد. در ادامه لیست پرامپتها را مشاهده می کنیم.

همچنین از کدهای آماده نمونه از گیتهاب Spark استفاده شد. نمونه های شمارش کلمات و همچنین

و پیاده سازی کتابخانه آن Word2vec

### ۱-۴\_لیست پرامپتها

نحوه اجرای کد زیر در اسپارک را توضیح بده: کد نمونه پایتون و شمارش تعداد کلمه ها در گیتهاب  
چرا وکتورهای من با Word2Vec تغییر نمی کنند و مقادیر همگی ثابت می مانند؟ به همراه کد حلقه  
آموزش

چطور Word2Vec را در حالت Skip-gram بدون استفاده از کتابخانه آماده مثل ML از spark پیاده سازی کنم؟

چرا تمام Similarity برابر با ۱ هستند؟ کجای کد مشکل دارد؟

تابع Cosine Similarity زیر درست است یانه و چرا خروجی من همیشه ثابت است؟  
کد مربوط به ۳-gram را با استفاده از NLTK بنویس تا نتیجه کد Spark زیر را چک کنم.  
چرا تعداد ۳-gramها درست محاسبه نمی شود و نتیجه استفاده از کتابخانه با پیاده سازی من متفاوت  
است.

چطور می توانم داده های پیش بینی شده Kmeans را به صورت نمودار نمایش دهم کد آن را بدهید.

چرا دستور `hdfs dfs -put` زیر فایل را آپلود نمی‌کند.

میخواهم از PCA را در Spark برای کاهش ابعاد به دو بعد استفاده کنم. یک نمونه کد بدهید.

چطور می‌توانم حداقل، حداقل و میانگین را با کمک Spark از یک فایل csv حساب کنم.

چرا کلمات فارسی در نمودار Heatmap درست نمایش داده نمی‌شوند.

کد زیر (کد word2vec کتابخانه) را به ساده ترین حالت ممکن برایم توضیح بده.

How can I implement Skip-gram Word2Vec without using libraries like MLlib

How can I convert an RDD to a DataFrame

Whats the role of regularization in Word2Vec, and why do my word vectors explode in size during training

Whats the difference between my code and library code of word2vec in ML library of spark

چندین پرامپت مربوط به تلاش برای حل نتیجه نگرفتن حلقه آموزش WORD2VEC بود.