**دانشگاه اصفهان**

**دانشکده مهندسی کامپیوتر**

گزارش فنی تمرین دوم

# NLP

پدیدآورنده:

**محمد امین کیانی**

**۴۰۰۳۶۱۳۰۵۲**

دانشجوی کارشناسی، دانشکده‌ی کامپیوتر، دانشگاه اصفهان، اصفهان،
Aminkianiworkeng@gmail.com

استاد درس: جناب اقای دکتر برادران

نیمسال دوم تحصیلی ۰۴-۱۴۰۳

# فهرست مطالب

# مستندات

## بخش اول: پرسش‌ها

### 1- تفاوت بین one-hot encoding و word embedding را بیان کنید.

در روش **One-Hot Encoding**، هر کلمه موجود در واژگان (Vocabulary) به یک بردار باینری تبدیل می‌شود که طول آن برابر با تعداد کل کلمات در واژگان است. در این بردار، تنها یک عنصر مقدار ۱ دارد (که موقعیت آن نشان‌دهنده کلمه موردنظر است) و بقیه عناصر مقدار ۰ دارند. این نوع نمایش بسیار ساده است و تنها حضور یا عدم حضور کلمه را در نظر می‌گیرد. اما یکی از معایب اصلی One-Hot Encoding این است که بردارهای حاصل بسیار پراکنده (Sparse) هستند و هیچ اطلاعاتی درباره ارتباط معنایی بین کلمات مختلف منتقل نمی‌کنند. برای مثال، در این نمایش، کلمات "king" و "queen" هیچ شباهتی به هم ندارند و فاصله‌ی اقلیدسی بین همه‌ی کلمات تقریباً برابر است.

در مقابل، **Word Embedding** روشی پیشرفته‌تر است که هر کلمه را به یک بردار چگال (Dense Vector) در فضای ابعاد پایین‌تر (مثلاً ۵۰، ۱۰۰ یا ۳۰۰ بعدی) نگاشت می‌کند. این بردارها طوری آموزش داده می‌شوند که کلمات دارای معانی مشابه در فضا نزدیک به یکدیگر قرار بگیرند. به عبارت دیگر، Word Embedding روابط معنایی و نحوی بین کلمات را در خود ذخیره می‌کند. برای مثال، بردار کلمه‌های "king" و "queen" در این فضا به یکدیگر نزدیک خواهند بود و می‌توان عملیات‌های معنایی مانند king - man + woman ≈ queenرا روی آن‌ها انجام داد.

از دیگر تفاوت‌های مهم می‌توان به این موارد اشاره کرد:

- One-Hot Encoding نیاز به حافظه زیاد و محاسبات سنگین دارد، در حالی که Word Embedding حافظه و منابع محاسباتی را بهینه‌تر استفاده می‌کند.
- One-Hot Encoding مناسب برای مدل‌های ساده و مسائل کوچک است، ولی Word Embedding برای مدل‌های یادگیری عمیق و پردازش زبان طبیعی در مقیاس بزرگ بسیار ضروری است.

در مجموع، Word Embedding نسبت به One-Hot Encoding مزایای قابل توجهی در درک بهتر معنا و روابط بین کلمات فراهم می‌کند و به همین دلیل در اغلب کاربردهای مدرن NLP به کار می‌رود.

## ۲ - به طور خلاصه توضیح دهید که الگوریتم GloVe چگونه Word Embedding ها را تولید می‌کند.

الگوریتم **GloVe** مخفف Global Vectors for Word Representation یک روش مبتنی بر آمار جهانی واژگان برای تولید Word Embedding است. برخلاف روش‌هایی مانند Word۲Vec که بیشتر به اطلاعات محلی (local context) تکیه دارند، GloVe از **آمار کلی** (**global statistics**) متن استفاده می‌کند تا بردارهای کلمات را یاد بگیرد.

مراحل کلی کار الگوریتم GloVe به این صورت است که:

۱. ابتدا یک **ماتریس هم‌وقوعی** (**co-occurrence matrix**) ساخته می‌شود. در این ماتریس، هر سطر و ستون مربوط به یک کلمه از واژگان است و مقدار هر خانه نشان‌دهنده‌ی تعداد دفعاتی است که دو کلمه با هم در یک پنجره مشخص (مثلاً پنج کلمه قبل و بعد) ظاهر شده‌اند.

۲. سپس الگوریتم سعی می‌کند برای هر کلمه یک بردار عددی یاد بگیرد، به طوری که **ضرب داخلی** (**dot product**)این بردارها بتواند نسبت‌های هم‌وقوعی بین کلمات را مدل کند. یعنی اگر دو کلمه در دنیای واقعی زیاد با هم ظاهر می‌شوند، حاصل ضرب بردارهای آن‌ها باید بزرگ‌تر باشد و بالعکس.

۳. برای این منظور، GloVe یک **تابع هزینه** (**cost function**) تعریف می‌کند که اختلاف بین ضرب داخلی بردارها و لگاریتم تعداد هم‌وقوعی واقعی را کمینه می‌کند. استفاده از لگاریتم باعث می‌شود که داده‌های بسیار بزرگ مقیاس‌بندی شوند و آموزش مدل پایدارتر شود و این تابع یک **تابع وزنی** است که برای کنترل اهمیت داده‌های مختلف استفاده می‌شود. این تابع معمولاً طوری طراحی می‌شود که برای مقادیر کوچک وزن کم و برای مقادیر بزرگ وزن بیشتر بدهد، ولی از رشد بی‌رویه برای مقادیر خیلی زیاد جلوگیری کند.

٤. الگوریتم با استفاده از بهینه‌سازی عددی مانند SGD یا روش‌های مشابه، این بردارها را طوری تنظیم می‌کند که تابع هزینه کمینه شود.

**ویژگی‌های کلیدی GloVe** :

- ترکیبی از مزایای مدل‌های مبتنی بر ماتریس و مدل‌های مبتنی بر پیش‌بینی است.
- اطلاعات آماری کامل کل پیکره (corpus) را لحاظ می‌کند، نه فقط پیش‌بینی یک کلمه از روی همسایگانش.

- به خوبی روابط خطی معنایی بین کلمات را مدل می‌کند. به عنوان مثال، عملیات برداری مانند king − woman + man ≈ queen در فضای تعبیه‌شده به خوبی جواب می‌دهد.

در نتیجه، GloVe یک الگوریتم بسیار کارآمد برای تولید بردارهای معنایی از کلمات است و می‌خواهد بردارهای کلمات را طوری یاد بگیرد که فاصله‌ی بین ضرب داخلی آن‌ها و لگاریتم هم‌وقوعی‌های واقعی به حداقل برسد، به شکلی که ساختار آماری زبان در فضای برداری کلمات بازنمایی شود. پس اگر نیاز به یک روش سریع و کم‌هزینه برای پردازش متن باشد، GloVe انتخاب خوبی است و البته امروزه، بسیاری از پروژه‌های NLP به جای GloVe و ELMo ، از BERT ، GPT و word2vec استفاده می‌کنند. در کاربردهای مدرن NLP ، روش‌های مبتنی بر ترانسفورمر مانند BERT و T5 جایگزین روش‌های قدیمی‌تر مانند ELMo و GloVe شده‌اند.

## GloVe (Global Vectors for Word Representation) -

GloVe یک روش **ایستا (Static Embedding)** برای تبدیل کلمات به بردارهای عددی است که توسط **استنفورد** در سال ۲۰۱۴ ارائه شد. این روش ترکیبی از مدل‌های **Word2Vec** و **الگوریتم‌های ماتریسی** مانند **SVD** است. ویژگی‌های اصلی GloVe :

۱. بر پایه **آمار هم‌رخدادی کلمات در کل مجموعه داده (Corpus-wide Co-occurrence Statistics)**

- GloVe از **ماتریس هم‌رخدادی (Co-occurrence Matrix)** استفاده می‌کند که نشان می‌دهد هر کلمه چند بار در کنار کلمات دیگر ظاهر شده است.
- این روش توزیع احتمال کلمات را در یک ماتریس بزرگ ذخیره کرده و سپس از تجزیه **ماتریس بردارهای ویژه (Singular Value Decomposition - SVD)** برای فشرده‌سازی آن استفاده می‌کند.

۲. **ایستا بودن(Static Embedding)**

- در GloVe ، هر کلمه یک بردار **ثابت و از پیش آموزش‌دیده‌شده** دارد که در طول پردازش تغییر نمی‌کند.
- پس اگر کلمه **"bank"** در دو جمله مختلف بیاید (یکی به معنای بانک مالی و دیگری به معنای کنار رودخانه)، در هر دو حالت **بردار یکسانی خواهد داشت.**

۳. **فراگیری معنایی از کل متن(Context-Independent)**

- این روش کل متن را تحلیل می‌کند تا روابط معنایی میان کلمات را استخراج کند.

٤. **کم هزینه و سریع در اجرا**

- به دلیل اینکه GloVe از قبل روی مجموعه داده‌های بزرگ مانند **Wikipedia** و **Common Crawl** آموزش داده شده، برای استفاده در پروژه‌های NLP بسیار سریع است.

**مشکلاتGloVe**

- **عدم درک تنوع معنایی کلمات (Polysemy Problem)**

- کلماتی که **چند معنی دارند** در همه جملات با یک بردار ثابت نمایش داده می‌شوند.
- مثال: کلمه‌ی **"apple"** همیشه یک بردار دارد، چه در جمله‌ی **"Apple Inc. released a new iPhone"** باشد و چه در جمله‌ی **"I ate an apple"**.

## ۳ - به طور خلاصه توضیح دهید که الگوریتم Word2Vec چگونه Word Embedding ها را تولید می‌کند.

الگوریتم **Word2Vec** یکی از محبوب‌ترین روش‌ها برای تولید Word Embedding است که توسط تیم Google در سال ۲۰۱۳ معرفی شد. برخلاف روش‌های مبتنی بر ماتریس مانندGloVe ، Word2Vec بر اساس **مدل‌سازی پیش‌بینی (Predictive Modeling)** کار می‌کند و هدفش این است که با یادگیری از داده‌های متنی، بتواند روابط معنایی بین کلمات را کشف کند.

**دو معماری اصلی در Word2Vec وجود دارد:**

۱. **Continuous Bag of Words (CBOW)** :در این روش، مدل سعی می‌کند با استفاده از کلمات زمینه(Context Words) ، کلمه‌ی مرکزی (Target Word) را پیش‌بینی کند. به عبارتی دیگر، مدل بردارهای کلمات زمینه را ترکیب کرده و خروجی مدل باید کلمه‌ی هدف باشد.

۲. **SGNS :**در این روش برعکس CBOW عمل می‌شود؛ یعنی مدل از یک کلمه مرکزی استفاده می‌کند تا کلمات زمینه اطرافش را پیش‌بینی کند. این مدل به ویژه برای داده‌های کم‌یاب (rare words) عملکرد بهتری دارد.

**مراحل اصلی کار Word2Vec :**

- ابتدا متن ورودی به صورت توکنایز شده (کلمه به کلمه) آماده می‌شود.
- برای هر کلمه، بر اساس یک پنجره‌ی زمینه (مثلاً ۵ کلمه قبل و ۵ کلمه بعد)، جفت‌های input و output ساخته می‌شود.
- مدل یک شبکه عصبی ساده (اغلب یک لایه مخفی) با وزن‌های مشترک می‌سازد. بردارهای کلمات در وزن‌های این شبکه ذخیره می‌شوند.
- هدف مدل این است که احتمال درست بودن یک جفت (کلمه ورودی، کلمه هدف) را بیشینه کند.
- از تکنیک‌هایی مانند **Negative Sampling** یا **Hierarchical Softmax** برای بهینه‌سازی و کاهش هزینه محاسباتی استفاده می‌شود.
- **تابع هزینه درWord2Vec :**تابع هدف به گونه‌ای طراحی شده است که احتمال پیش‌بینی درست کلمات هدف را حداکثر کند.

**مزایای Word2Vec :**

- سرعت بالا در آموزش.
- قابلیت کشف روابط معنایی و نحوی پیچیده بین کلمات.
- امکان انجام عملیات‌های معنایی روی بردارها مانند(king - man + woman ≈ queen)

الگوریتم Word2Vec با یادگیری از توزیع آماری کلمات در زمینه‌هایشان، بردارهای چگال و کم‌بعدی برای کلمات تولید می‌کند که معانی و روابط بین کلمات را به شکلی بسیار کارآمد مدل می‌کنند.

## ۴ - واژه‌های دارای چندمعنی (Polysemy) چگونه در Word Embedding ها کنترل می‌شوند و چه چالش‌هایی تولید می‌کند؟

در زبان طبیعی، بسیاری از کلمات دارای **چندین معنی** هستند که به این ویژگی **چندمعنایی (Polysemy)** گفته می‌شود. مثلا کلمه‌ی "bank" می‌تواند به معنای بانک مالی یا کرانه‌ی رودخانه باشد. در مدل‌های کلاسیک Word Embedding مانند Word2Vec یا GloVe، **هر کلمه فقط به یک بردار واحد** نگاشت می‌شود، بدون توجه به تفاوت معانی مختلف آن در متن. این رویکرد باعث بروز چند چالش اساسی می‌شود:

۱. **از دست رفتن اطلاعات معنایی دقیق:** چون تنها یک بردار برای تمام معانی یک کلمه وجود دارد، مدل نمی‌تواند تشخیص دهد که در کدام موقعیت کلمه‌ی "bank" به موسسه مالی و در کدام موقعیت به کرانه‌ی رودخانه اشاره دارد.

۲. **کاهش دقت مدل در وظایف معنایی:** در کاربردهایی مانند ترجمه ماشینی، پاسخ به سوالات یا تحلیل احساسات، درک دقیق معنی کلمه بسیار حیاتی است. چندمعنایی می‌تواند باعث شود مدل پیش‌بینی‌های اشتباهی انجام دهد و به علت ترکیب چند معنی در یک بردار، گاهی بردار کلمات غیرمرتبط به هم نزدیک می‌شود یا کلمات مرتبط در فضای برداری از هم دور می‌شوند.

۳. **تداخل در فضای برداری:** بردار کلماتی که چندمعنی دارند، معمولاً جایی بین معانی مختلف قرار می‌گیرند و در نتیجه نمی‌توانند به طور دقیق هیچ معنایی را به‌خوبی بازنمایی کنند.

**روش‌های کنترل چندمعنایی درWord Embedding :**

برای رفع این مشکلات، در تحقیقات اخیر چندین روش معرفی شده است:

۱. **Multi-Sense Embedding :** به جای اختصاص یک بردار واحد به هر کلمه، چندین بردار برای هر کلمه ایجاد می‌شود که هر کدام نماینده‌ی یک معنای متفاوت هستند. برای مثال، مدل‌هایی مانند **Multi-Sense Skip-Gram** هر زمان که یک کلمه در متنی ظاهر شود، با توجه به زمینه (Context) مناسب‌ترین معنای آن را انتخاب می‌کنند.

۲. **Contextualized Embedding :** مدل‌های پیشرفته‌تر مانند **ELMo** و **BERT** این مشکل را به صورت بنیادی حل کرده‌اند. در این مدل‌ها، بردار هر کلمه وابسته به جمله و موقعیت آن در متن ساخته

می‌شود. بنابراین، کلمه‌ی "bank" در جمله‌ی "I deposited money in the bank" بردار متفاوتی نسبت به جمله‌ی "He sat by the bank of the river" خواهد داشت.

۳. **Clustering Based Approaches :** بعضی روش‌ها با خوشه‌بندی کاربردهای مختلف کلمات در پیکره‌های متنی بزرگ، معانی مختلف کلمات را شناسایی و برای هر خوشه یک بردار مجزا تولید می‌کنند.

٤. **Embedding مبتنی بر Word Sense Disambiguation (WSD) :** بعضی از روش‌ها ابتدا با استفاده از تکنیک‌های WSD (تعیین معنای دقیق کلمه بر اساس متن) معنای مناسب هر کلمه را تشخیص می‌دهند، سپس Embedding مرتبط با آن معنا را استفاده یا تولید می‌کنند.

## ۵ – Word Embeddingها نیاز دارند که تمام کلمات در مجموعه آموزش حضور داشته باشند. چگونه با کلمات خارج از واژگان (OOV) برخورد می‌کنید؟ یک روش برای تولیدWord Embedding برای کلماتی که در داده‌ی آموزش حضور نداشته‌اند پیشنهاد دهید.

در مدل‌های سنتی Word Embedding مانند Word2Vec وGloVe ، تنها کلماتی که در مجموعه‌ی آموزش دیده شده‌اند دارای بردار (Embedding) هستند. اما در بسیاری از کاربردهای واقعی، با کلماتی مواجه می‌شویم که در داده‌های آموزشی موجود نبوده‌اند. به این کلمات، کلمات خارج از واژگان یا **OOV** گفته می‌شود. مدیریت صحیح کلمات OOV یک مسأله‌ی بسیار مهم در پردازش زبان طبیعی است، زیرا مدل بدون بردار معنایی مناسب برای این کلمات نمی‌تواند پیش‌بینی‌های درستی انجام دهد.

مدیریت مناسب کلمات OOV نقش بسیار مهمی در کارایی مدل‌های پردازش زبان طبیعی دارد. استفاده از روش‌هایی مانند FastText وEmbedding های مبتنی بر ساختار داخلی کلمات می‌تواند این مشکل را تا حد زیادی حل کند و دقت مدل را در مواجهه با داده‌های جدید حفظ نماید.

**چالش‌های برخورد با OOV :**

۱. **نبود بردار عددی برای کلمه‌ی جدید:** بدون Embedding مناسب، کلمه OOV نمی‌تواند در مدل پردازش شود.

۲. **کاهش دقت مدل در فهم جملات جدید:** اگر کلمات مهم جمله OOV باشند، مدل نمی‌تواند معنای جمله را به درستی درک کند.

۳. **ناپایداری مدل در مواجهه با داده‌های جدید:** در محیط‌های واقعی مانند گفتگو، اخبار، شبکه‌های اجتماعی و غیره دائماً کلمات جدید یا کمیاب ظاهر می‌شوند.

**روش‌های برخورد با OOV :**

۱ .**استفاده از بردار تصادفی(Random Initialization) :** یکی از ساده‌ترین روش‌ها این است که برای کلمات OOV یک بردار تصادفی (Random Vector) ایجاد کنیم. این بردار ممکن است در طول آموزش بعدی یا Fine-tuning بهبود پیدا کند، ولی در ابتدا اطلاعات معنایی خاصی ندارد.

۲ .**استفاده از یک بردار مشترک برای تمامOOV ها:** در این روش، یک بردار خاص مثلاً Embedding مربوط به توکن [UNK] برای همه‌ی کلمات ناشناخته استفاده می‌شود. این روش ساده است ولی معانی متفاوت کلمات OOV را از بین می‌برد.

۳ .**تولید بردار بر اساس ساختار کلمه(Character-based Embedding) :** یک روش پیشرفته‌تر، تولید Embeddingکلمه‌ی OOV بر اساس اجزای داخلی آن (کاراکترها، زیرکلمات) است.

- مدل‌هایی مانند **FastText** این روش را پیاده‌سازی می‌کنند. درFastText ، کلمه به مجموعه‌ای از-n gramهای کاراکتری شکسته می‌شود و بردار کلمه از ترکیب بردارهای این n-gram ها به دست می‌آید.
- این رویکرد باعث می‌شود حتی کلمات ندیده شده، اگر ساختار مشابهی با کلمات موجود داشته باشند، بردار مناسبی دریافت کنند.

۴ .**تخمین بردار با استفاده از نزدیک‌ترین کلمات(Nearest Neighbors) :** اگر یک کلمه OOV شبیه به کلمات موجود در واژگان باشد (مثلاً از نظر ریشه یا پسوندها)، می‌توان بردار آن را بر اساس بردارهای کلمات مشابه موجود تخمین زد. این روش نیاز به الگوریتم‌هایی مانند fuzzy matching دارد.

۵. **Contextual Embedding Models . :** در مدل‌های پیشرفته مانند BERT یاGPT ، حتی برای کلمات جدید، می‌توان بردار معنایی تولید کرد، زیرا این مدل‌ها بر اساس توکنایز کردن به زیرواحدهای کلمه کار می‌کنند و کلمه‌ی جدید را به اجزای آشنای خود می‌شکنند. بنابراین OOV به عنوان ترکیبی ازsubword ها پردازش می‌شود.

**پس پیشنهاد نهایی در تولید Word Embedding برای کلماتOOV :**

یک روش بسیار مؤثر استفاده از **مدل‌های مبتنی بر زیرواژه (Subword Models)** مانند **FastText** است. در این روش:

- کلمات به n-gram های کاراکتری شکسته می‌شوند مثلاً "bank" به"ban" ، "ank"و غیره.
- برای هر n-gram یک بردار یادگرفته شده وجود دارد. پس به جای یادگیری بردار فقط برای خود کلمه، برای هر زیرواحد یک بردار مجزا آموزش داده می‌شود. این بردارها در طول آموزش روی کل پیکره‌ی متنی ساخته می‌شوند.
- بردار نهایی کلمه از مجموع یا میانگین بردارهای n-gram های آن ساخته می‌شود.

این روش به جای این که به هر کلمه به طور مستقل یک بردار اختصاص دهیم مثل Word2Vec یا GloVe، هر کلمه به **ترکیبی از n-gram های کاراکتری** شکسته می‌شود و بردار کلمه از بردارهای این زیرواحدهای کوچک ساخته می‌شود که باعث می‌شود حتی اگر کلمه‌ی جدید در واژگان آموزشی وجود نداشته باشد، مدل بتواند یک بردار معنایی منطقی برای آن تولید کند. چون ممکن است n-gram هایش قبلاً دیده شده باشند.

**چرا این روش پیشنهادی مناسب است؟**

- ساختار داخلی کلمات را لحاظ می‌کند.
- قادر به درک و پردازش کلمات ترکیبی، جمع‌های جدید، یا تغییرات صرفی و نحوی کلمات است.
- بدون نیاز به ذخیره‌سازی جداگانه‌ی هر کلمه‌ی ممکن، می‌تواند Embedding معنادار برای کلمات جدید تولید کند.
- سرعت بالایی دارد و برای کاربردهای مقیاس بزرگ مناسب است.

## ۶ – ماتریس Co-occurrence متن زیر را بنویسید. اندازه پنجره را ۲ در نظر بگیرید.

**I love computer science and I love NLP even more.**

**در ماتریس هم‌وقوعی (co-occurrence matrix)**، هر سطر و ستون مربوط به یک کلمه از واژگان است و مقدار هر خانه نشان‌دهنده‌ی تعداد دفعاتی است که دو کلمه با هم در یک پنجره مشخص (مثلاً پنج کلمه قبل و بعد) ظاهر شده‌اند.

**اندازه پنجره** = **۲** یعنی برای هر کلمه، دو کلمه قبل و دو کلمه بعد به عنوان همسایه در نظر گرفته می‌شود و ماتریس Co-occurrence نشان می‌دهد که هر کلمه چند بار در کنار کلمات دیگر ظاهر شده است.

**مرحله ۱_ فهرست کلمات (واژگان متن):**

بعد از توکنایز کردن متن و بدون توجه به نقطه‌گذاری:

[I, love, computer, science, and, I, love, NLP, even, more]

واژگان(Vocabulary) :

I, love, computer, science, and, NLP, even, more

کلمه "I" و "love" دوبار آمده‌اند ولی در واژگان فقط یک بار لیست می‌شوند.

**مرحله ۲_ بررسی همسایه‌ها برای هر کلمه:**

حالا برای هر کلمه، همسایگان تا شعاع ۲ کلمه‌ی قبل و بعد را در نظر می‌گیریم:

| کلمه جاری | کلمات همسایه (حداکثر دو کلمه قبل و دو بعد) |
|---|---|
| **I (اول)** | love, computer |
| **love (اول)** | I, computer, science |
| **computer** | I, love, science, and |
| **science** | love, computer, and, I |
| **and** | computer, science, I, love |
| **I (دوم)** | science, and, love, NLP |
| **love (دوم)** | and, I, NLP, even |
| **NLP** | I, love, even, more |
| **even** | love, NLP, more |
| **more** | NLP, even |

**مرحله ۳: ساخت ماتریس Co-occurrence :**

اکنون ماتریس مربعی را بر اساس همسایگی بالا پر می‌کنیم. ردیف و ستون‌های ماتریس به ترتیب واژگان مرتب شده‌اند. ماتریس متقارن نیست چون ترتیب ظاهر شدن کلمات اهمیت دارد و همسایگی ممکنه فقط از یک طرف اتفاق افتاده باشه(جهت دار است و نه صرفا کنار هم بودن!)، ولی در بسیاری از کاربردها آن را متقارن هم در نظر می‌گیرند. مقادیر داخل ماتریس تعداد دفعاتی است که دو کلمه در شعاع ۲ کلمه‌ی همدیگر ظاهر شده‌اند.

| | I | love | computer | science | and | NLP | even | more |
|---|---|---|---|---|---|---|---|---|
| I | ۰ | ۲ | ۱ | ۱ | ۱ | ۱ | ۰ | ۰ |
| love | ۲ | ۰ | ۱ | ۱ | ۱ | ۱ | ۱ | ۰ |
| computer | ۱ | ۱ | ۰ | ۲ | ۱ | ۰ | ۰ | ۰ |
| science | ۱ | ۱ | ۲ | ۰ | ۲ | ۰ | ۰ | ۰ |
| and | ۱ | ۱ | ۱ | ۲ | ۰ | ۱ | ۰ | ۰ |
| NLP | ۱ | ۱ | ۰ | ۰ | ۱ | ۰ | ۱ | ۱ |
| even | ۰ | ۱ | ۰ | ۰ | ۰ | ۱ | ۰ | ۱ |
| more | ۰ | ۰ | ۰ | ۰ | ۰ | ۱ | ۱ | ۰ |

درایه ها: یعنی چند بار دو کلمه در یک پنجره‌ی مشخص (اینجا پنجره ۲) کنار هم ظاهر شده‌اند. یعنی مقدار هر خانه **(i,j)** برابر است با تعداد دفعاتی که کلمه‌ی همسایه **j** در پنجره‌ی اطراف کلمه‌ی مرکز **i** دیده شده است.

## ۷ – مزیت‌های نسبی Word2Vec و GloVe در چه کاربردهایی بیشتر نمایان می‌شود؟

هر دو الگوریتم Word2Vec و GloVe از روش‌های پرکاربرد برای تولید Word Embedding هستند، اما رویکردهای متفاوتی دارند و بسته به نوع کاربرد، یکی نسبت به دیگری بهتر عمل می‌کند.

**۱- ویژگی اصلی Word2Vec :**

- مبتنی بر مدل پیش‌بینی (Predictive Model) است.
- از اطلاعات **محلی (Local Context)** برای یادگیری بردارهای کلمات استفاده می‌کند.
- در دو نسخه CBOW و Skip-gram پیاده‌سازی شده است.

**کاربردهایی که Word2Vec بهتر عمل می‌کند:**

۱. **پردازش جملات کوتاه یا داده‌های کم:**

- چون Word2Vec روی پیش‌بینی کلمه بر اساس همسایگانش تمرکز دارد، برای جملات کوتاه یا متونی که آمار هم‌وقوعی کامل نیستند بهتر جواب می‌دهد.

۲. **یادگیری روابط نحوی و معنایی پیچیده:**

- در Word2Vec عملیات برداری به خوبی قابل اجراست و این در وظایفی مثل **تحلیل معنا**، **تشخیص مترادف‌ها** یا **واژه‌یابی معنایی** ارزشمند است.

۳. **کاربردهای بلادرنگ(Real-Time Applications) :**

- سرعت بالای آموزش Word2Vec مخصوصاً Skip-gram با Negative Sampling مناسب پروژه‌هایی است که نیاز به تولید سریع Embedding دارند.

**۲- ویژگی اصلی GloVe :**

- مبتنی بر آمار جهانی هم‌وقوعی (Global Co-occurrence) کلمات است.
- سعی می‌کند توزیع کلی واژگان را در نظر بگیرد، نه فقط همسایگی‌های محلی.

**کاربردهایی که GloVe بهتر عمل می‌کند:**

۱. **پردازش مجموعه‌های بزرگ داده(Big Data Processing) :**

- وقتی داده‌های بسیار حجیم و کامل داریم مثلاً کل ویکی‌پدیا یا پیکره‌ی Common Crawl ، با استفاده از کل آمار هم‌وقوعی دقیق‌تر عمل می‌کند.

۲. **تحلیل مفاهیم کلی و عمومی:**

- در کاربردهایی که نیاز به درک مفاهیم عمومی‌تر مثل **مفهوم کلی اسناد**، **مدیریت واژگان گسترده** داریم، GloVe بهتر از Word2Vec عمل می‌کند.

۳. **ثبات در بردارهای معنایی:**

- چون بر آمار جهانی متکی است، بردارهای کلمات در GloVe پایدارتر هستند و برای کارهایی مثل **دسته‌بندی اسناد** یا **تحلیل محتوا** کاربرد خوبی دارند.

**پس:**

- اگر نیاز به **درک دقیق روابط بین کلمات در متون کوتاه یا زمان آموزش سریع** داریم، **Word2Vec** گزینه بهتری است.
- اگر نیاز به **مدل‌سازی معنایی دقیق‌تر روی داده‌های بسیار بزرگ و جامع** داریم، **GloVe** عملکرد بهتری ارائه می‌دهد.

## بخش دوم: ساخت مدل زایشی با استفاده از N-Gram

```
import nltk
nltk.download('reuters')       # دانلود پیکره Reuters
nltk.download('punkt')         #   توکن‌ساز اصلی punkt_tab
nltk.download('stopwords')     # واژه های بی‌معنی

#  پاکسازی کامل کش‌های خراب
nltk.download('popular')
nltk.download('all')

# حالا ادامه پیش‌پردازش:

from nltk.corpus import reuters, stopwords
import string
import re
from nltk.tokenize import sent_tokenize, word_tokenize
from wordcloud import WordCloud
import matplotlib.pyplot as plt
```

پیش‌پردازش:

```
# حذف فاصله‌های اضافی و تقسیم به جملات
raw_text = ' '.join(reuters.words())
sentences = sent_tokenize(raw_text)
sentences = [s.strip() for s in sentences if s.strip()]

# تبدیل همه حروف به حروف کوچک
sentences = [s.lower() for s in sentences]

# توکن‌سازی هر جمله به کلمات
tokenized_sentences = [word_tokenize(sent) for sent in sentences]

#  حذف اعداد و لینک ها
```

```
def remove_numbers_urls(tokens):
    return [w for w in tokens if not re.match(r'http\S+|\d+', w)]

tokenized_sentences = [remove_numbers_urls(sent) for sent in
tokenized_sentences]

#  حذف علائم نگارشی و کلمات ایست
stop_words = set(stopwords.words('english'))
punct = set(string.punctuation)

def clean_tokens(tokens):
    return [w for w in tokens if w not in punct and w not in stop_words
and len(w) > 1]

cleaned_sentences = [clean_tokens(sent) for sent in tokenized_sentences]

print(cleaned_sentences[:50])

#  رسم ابر کلمات
all_words = ' '.join([' '.join(sent) for sent in cleaned_sentences])
wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(all_words)

plt.figure(figsize=(12, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("WordCloud of Cleaned Reuters Corpus")
plt.show()

# جمع‌آوری متن کامل
text = ' '.join(reuters.words())

# توکن‌سازی + کوچک‌سازی
tokens = word_tokenize(text.lower())

# حذف علائم نگارشی
tokens = [t for t in tokens if t not in string.punctuation]

# حذف stopwords انگلیسی
stop_words = set(stopwords.words('english'))
tokens = [t for t in tokens if t not in stop_words]

# حذف نویزهای عددی و تک‌حرفی
tokens = [t for t in tokens if t.isalpha() and len(t) > 1]
```

```
# نمایش نمونه
print(tokens[:50])
```

## ساخت مدل زایشی با استفاده از N-Gram :

ساختن یک مدل زبانی ساده که با استفاده از توالی‌های N کلمه، احتمال کلمه‌ی بعدی را یاد بگیرد و بتواند متن تولید کند که از **احتمالات شرطی آماری** استفاده می‌کند.

- **Bigram (N=2)** → سریع ولی ساده
- **Trigram (N=3)** → متعادل و متداول
- **4-gram یا بیشتر** → دقیق‌تر ولی حساس به داده کم

در اینجا از **Trigram (N=3)** استفاده می‌کنیم. با دو روش آماده و دستی :

```
from collections import defaultdict, Counter
from nltk.util import ngrams

# تبدیل لیست جمله‌ها به یک لیست بلند از کلمات:
flattened_tokens = [word for sentence in cleaned_sentences for word in
sentence]
flattened_tokens

# تولید همه‌ی trigrams (n=3)
trigrams = list(ngrams(flattened_tokens, 3))
trigrams

# ساخت مدل: احتمال وقوع کلمه سوم با توجه به دو کلمه قبلی
model = defaultdict(Counter)
for w1, w2, w3 in trigrams:
    model[(w1, w2)][w3] += 1

model
```

$$\frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2, w) \sum_w} = P(w_3 | w_1, w_2)$$

```
print(model[("new", "york")].most_common(5))

[('investor', 24), ('stock', 22), ('said', 21), ('times', 18),
('investment', 15)]
```

```
# مدل ۱-گرم (Unigram)
unigram_model = defaultdict(int)
for word in train_tokens:
    unigram_model[word] += 1

# مدل ۲-گرم (Bigram)
bigram_model = defaultdict(lambda: defaultdict(int))
for i in range(len(train_tokens) - 1):
    w1 = train_tokens[i]
    w2 = train_tokens[i + 1]
    bigram_model[w1][w2] += 1

# مدل ۳-گرم (Trigram)
trigram_model = defaultdict(lambda: defaultdict(int))
for i in range(len(train_tokens) - 2):
    w1 = train_tokens[i]
    w2 = train_tokens[i + 1]
    w3 = train_tokens[i + 2]
    trigram_model[(w1, w2)][w3] += 1
```

در کل پیکره‌ی Reuters، بعد از توالی **"new york"** ، چه کلماتی بیشتر ظاهر شده‌اند؟ و چند بار؟

## تولید متن:

در مدل‌های زایشی مبتنی برTri-gram :

- از دو کلمه‌ی اولیه شروع می‌کنیم مثلاً " interest "," rates "
- مدل بررسی می‌کنه که با توجه به این دو کلمه، **چه کلمات سومی محتمل‌تر هستند**.
- یکی از آن‌ها (معمولاً به صورت تصادفی با توجه به وزن احتمال) انتخاب می‌شود.
- سپس دو کلمه‌ی جدید می‌شوند: (کلمه‌ی دوم + کلمه‌ی تازه تولید شده) و این روند ادامه پیدا می‌کند.

اگه برای (w1, w2)داده‌ای تو مدل نباشد یا فقط یک کلمه‌ی بعدی بیاد که خودش ادامه ندارد، تولید متن **متوقف** می‌شود. پس:

۱. بررسی اینکه دوتایی ورودی اصلاً در مدل وجود دارد یا نه

۲. اطمینان از اینکه دوتایی حداقل دو مسیر ادامه دارد (تنوع دارد)

۳. در صورت نبود یا محدود بودن، انتخاب تصادفی یک دوتایی پرتکرار برای شروع

البته برای بهبود آن می‌توان Smoothing را اعمال کرد :

| توضیح | قابلیت |
|---|---|
| حتی اگر توالی خاصی در مدل نباشه، تولید ادامه می‌یابد | **عدم توقف** |
| اگر trigram پیدا نشد ← از bigram اگر نشد ← از unigram استفاده می‌کند | **Backoff** |
| همیشه محتمل‌ترین کلمه رو انتخاب می‌کند (نه تصادفی) | **Greedy Selection** |
| به همه کلمات حتی اونی که قبلاً ندیده احتمال غیرصفر می‌دهد | **Laplace Smoothing (Add-1)** |
| مناسب تحلیل، ارزیابی و تولید متن قابل تکرار | **پایدار، غیرتصادفی و دقیق** |

```python
import random

def generate_text(model, start=("the", "market"), length=50):
    w1, w2 = start
    output = [w1, w2]

    for _ in range(length):
        next_words = model.get((w1, w2), None)
        if not next_words:
            break  # اگر دنباله‌ای نباشد، متوقف شود

        # انتخاب تصادفی کلمه بعدی بر اساس وزن تکرارها
        w3 = random.choices(
            population=list(next_words.keys()),
            weights=list(next_words.values())
        )[0]

        output.append(w3)
        w1, w2 = w2, w3  # به‌روزرسانی دوتایی برای مرحله بعد

    # print(output)
```

```
    return ' '.join(output)

# شروع متن با یک دوتایی پرتکرار مثل
print(generate_text(model, start=("interest", "rates"), length=50))

interest rates tuesday bundesbank wanted set marker stock market sources
say concern anticipated...
```

## ارزیابی مدل با معیار Perplexity و بهینه‌سازی:

Perplexity(سردرگمی) یک معیار عددی برای ارزیابی کیفیت مدل‌های زبانی است. این معیار اندازه‌گیری می‌کند که مدل چقدر در پیش‌بینی کلمه بعدی خوب عمل می‌کند. هرچه Perplexity کمتر باشد، مدل بهتر است. به زبان ساده یعنی مدل به طور میانگین بین چند انتخاب "سردرگم" هست برای هر کلمه. اگر perplexity = 100، یعنی مدل برای پیش‌بینی هر کلمه انگار بین ۱۰۰ گزینه می‌چرخد و اگر log رو بر پایه ۲ بگیریم (مثل اطلاعات نظری)، عددها قابل تفسیرتر می‌شوند.

$$\log_2 P(w_i|w_{i-(n-1)},\ldots,w_{i-1}) \sum_{i=1}^{N} \frac{1}{N} 2 = \text{Perplexity}$$

**گام‌های ارزیابی:**

۱. تقسیم داده‌ها به دو بخش **train / test** :

۲. ساخت مدل بر اساس داده‌ی آموزش

۳. محاسبه احتمال شرطی کلمات تست با استفاده از مدل

٤. محاسبهPerplexity

پیکره Reuters ( 1.3میلیون کلمه و بیش از ۱۰,۰۰۰ سند) نسبت به نیاز مدل‌های n-gram با n بالا، پیکره‌ی کوچکی است. پس:

-برای unigram یا bigram کافیست .

-برای trigram مدل بسیار sparse می‌شود.

-به همین دلیل که با backoff و حذف کلمات نادر می‌توان بهبود داد ولی نه تا سطح عالی! تکرار واژگان پرتکرار (billion, dlrs) نشون می‌دهد مدل stuck شده است.

| مدل | Train Perp | Test Perp | CV Perp | تحلیل |
|---|---|---|---|---|
| **Unigram** | 1713 | 1620 | 1750 | ساده ولی پایدار |
| **Bigram** | 1432 | 2155 | 4938 | **به شدت Overfitting** |
| **Trigram** | 3086 | 5693 | 9860 | **حافظه‌کوتاه و sparsity** |

پس در نهایت با تست فراوان، بهترین حالت برای کار با مدل های کلاسیک این چنینی دریافتیم ترکیب آنها بهترین پاسخ را می‌دهد یعنی:

یک نسخه‌ی **پیشرفته‌تر از محاسبه Perplexity** برای مدل‌های N-Gram هست که به جای تکیه به یک منبع، از **ترکیب هم‌زمان سه مدلUnigram** ، **Bigramو Trigram** استفاده می‌کند. یعنی از **Smoothing از نوع Interpolation استفاده می‌کند.**

- اگر trigram نبود، برویم سراغ bigram مثل Backoff
- و اگر bigram نبود، برویم سراغ unigram
- $P(w3 \mid w1, w2) = \lambda_3 * P(w3 \mid w1, w2) + \lambda_2 * P(w3 \mid w2) + \lambda_1 * P(w3)$
- این روش در سطح کلاسیک، **نزدیک‌ترین عملکرد به مدل‌های neural** رو دارد.

```
def calculate_perplexity_interpolated(tokens, vocab, V, lambdas=(0.6, 0.3,
0.1)):
    λ3, λ2, λ1 = lambdas  # trigram, bigram, unigram

    tokens = [w for w in tokens if w in vocab]
    test_ngrams = list(ngrams(tokens, 3))
    log_prob = 0
    N = 0

    for w1, w2, w3 in test_ngrams:
        # Trigram
        trigram_count = trigram_model[(w1, w2)][w3] if (w1, w2) in
trigram_model else 0
        trigram_total = sum(trigram_model[(w1, w2)].values()) if (w1, w2)
in trigram_model else 0
        p_tri = (trigram_count + 1) / (trigram_total + V)

        # Bigram
        bigram_count = bigram_model[w2][w3] if w2 in bigram_model else 0
        bigram_total = sum(bigram_model[w2].values()) if w2 in
bigram_model else 0
        p_bi = (bigram_count + 1) / (bigram_total + V)
```

```
        # Unigram
        p_uni = (unigrams[w3] + 1) / (sum(unigrams.values()) + V)

        # Interpolated probability
        prob = λ3 * p_tri + λ2 * p_bi + λ1 * p_uni

        log_prob += math.log2(prob)
        N += 1

    return 2 ** (-log_prob / N)

print("Interpolated Trigram Perplexity on TEST:")
print(calculate_perplexity_interpolated(test_tokens, vocab, V,
lambdas=(0.6, 0.3, 0.1)))

print("Interpolated Trigram Perplexity on TRAIN:")
print(calculate_perplexity_interpolated(train_tokens, vocab, V,
lambdas=(0.6, 0.3, 0.1)))

Interpolated Trigram Perplexity on TEST:
2230.2769626038757
Interpolated Trigram Perplexity on TRAIN:
1594.400042138285
```

نتایج:

Text Generation :

**Unigram** پراکنده، بی‌ساختار، بدون مفهوم ← چون مدل فقط احتمال هر کلمه را جداگانه در نظر می‌گیرد .

**Bigram** روان‌تر و ساختاریافته‌تر، اما گاهی در حلقه تکرار یا قطع ناگهانی گیر می‌کند.

**Trigram** معنادار، ساختارمند و نزدیک به جمله‌های خبری واقعی — بهترین تعادل بین دقت و روانی.

Perplexity:

**Unigram** ساده‌ترین و پایدارترین مدل است، اما وابستگی زبانی را در نظر نمی‌گیرد.

**Bigram** دقت بیشتری دارد اما در تست و cross-val دچار overfitting می‌شود.

**Trigram** **خام آن** عملکرد ضعیفی دارد به دلیل sparsity و عدم توانایی تعمیم.

**Interpolated Trigram** ترکیبی هوشمند از همه مدل‌هاست و تعادلی بین دقت و پایداری ایجاد می‌کند.

- با توجه به داده‌ی واقعی، مدل‌های **کلاسیک N-Gram به‌تنهایی برای کاربردهای واقعی کافی نیستند**، ولی برای تمرین و تحلیل زبانی عالی‌اند.
- استفاده از Interpolation بهترین راه برای افزایش عملکرد بدون استفاده از مدل‌های عمیق است.
- Perplexity در مرتبه‌ی هزارگان برای Reuters طبیعی و قابل‌قبول است.

## بخش سوم: تحلیل احساسات با Naive Bayes

```
import nltk
nltk.download('movie_reviews')
from nltk.corpus import movie_reviews
import random

import string
from nltk.tokenize import word_tokenize
from collections import defaultdict, Counter
import re
import math
from nltk.corpus import stopwords

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

# پاکسازی کامل کش‌های خراب
nltk.download('popular')
nltk.download('all')
# لیست مستندات و برچسب‌ها
documents = [(list(movie_reviews.words(fileid)), category)
             for category in movie_reviews.categories()
             for fileid in movie_reviews.fileids(category)]

# درهم‌ریزی برای train/test
random.shuffle(documents)

# تعداد کل و توزیع برچسب‌ها
print(f"تعداد کل مستندات: {len(documents)}")
from collections import Counter
print("توزیع برچسب‌ها:", Counter([label for (text, label) in documents]))
# استخراج واژگان مهم
```

```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = list(all_words)[:2000]  # 2000 کلمه برتر

# تابع استخراج ویژگی از متن
def document_features(document):
    words = set(document)
    features = {}
    for w in word_features:
        features[f'contains({w})'] = (w in words)
    return features

# ایجاد مجموعه ویژگی‌ها
featuresets = [(document_features(d), c) for (d, c) in documents]

stop_words = set(stopwords.words('english'))
punct = set(string.punctuation)
custom_garbage = {'just', 'really', 'get', 'got', 'even', 'thing',
'things', 'also', 'though'}

# پیش‌پردازش کامل
def clean_tokens(tokens):
    return [
        w.lower() for w in tokens
        if w.lower() not in stop_words
        and w.lower() not in custom_garbage
        and w not in punct
        and len(w) > 2
        and w.isalpha()
        and not re.match(r'https?://\S+|\d+', w)
    ]

# بارگذاری داده‌ها
documents = []
for category in movie_reviews.categories():
    for fileid in movie_reviews.fileids(category):
        words = clean_tokens(movie_reviews.words(fileid))
        documents.append((words, category))

random.shuffle(documents)

# تقسیم به آموزش/تست
split_idx = int(0.7 * len(documents))
train_docs = documents[:split_idx]
test_docs = documents[split_idx:]
```

```
# ساخت واژگان و شمارش اولیه
class_word_counts = {'pos': Counter(), 'neg': Counter()}
class_doc_counts = {'pos': 0, 'neg': 0}
total_vocab = Counter()

for words, label in train_docs:
    class_doc_counts[label] += 1
    class_word_counts[label].update(words)
    total_vocab.update(words)

# فیلتر واژه‌های کم‌تکرار
vocab = {w for w, c in total_vocab.items() if c >= 3}
# محدود به ۴۰۰۰ واژه مهم
top_words = set([w for w, c in total_vocab.most_common(4000)])
vocab = vocab.intersection(top_words)
V = len(vocab)

# به‌روزرسانی مدل بر اساس واژگان بهبود یافته
for cls in ['pos', 'neg']:
    class_word_counts[cls] = Counter({w: c for w, c in
class_word_counts[cls].items() if w in vocab})

def predict_naive_bayes(text_tokens):
    tokens = clean_tokens(text_tokens)
    scores = {}
    total_docs = sum(class_doc_counts.values())

    for cls in ['pos', 'neg']:
        log_prob = math.log(class_doc_counts[cls] / total_docs)
        if cls == 'pos':
            log_prob += 0.2  # تقویت prior کلاس مثبت

        total_words = sum(class_word_counts[cls].values())

        for word in tokens:
            if word in vocab:
                count = class_word_counts[cls][word]
                numerator = math.log(count + 1)
                denominator = math.log(total_words + V)
                log_prob += numerator - denominator

        scores[cls] = log_prob

    return max(scores, key=scores.get)
```

```
y_true, y_pred = [], []

for words, label in test_docs:
    y_true.append(label)
    y_pred.append(predict_naive_bayes(words))

print(classification_report(y_true, y_pred, digits=4))
```

تعداد کل مستندات: ۲۰۰۰

توزیع برچسب‌ها: {'pos': 1000, 'neg': 1000}

دیتاست movie_reviews به‌صورت **balanced**(متوازن) بین نظرات مثبت و منفی هست و باعث می‌شود مدل درگیر bias به یک کلاس خاص نشده و آموزش متعادل‌تری داشته باشد.

0.85: Accuracy

macro avg / weighted avg: حدود ۰.۸۵ در تمام معیارها

یعنی از بین ۴۰۰ نقد تست، مدل **340 مورد رو درست پیش‌بینی کرده** است و این برای یک مدل ساده Bag of Words با Naive Bayes کاملاً خوب است که به ازای اموزش ۸۰ درصد بود و در ادامه به ۷۰ درصد می‌رسانیم:

| معیار | مقدار فعلی | وضعیت |
|---|---|---|
| **Accuracy** | 82.00% | خوب |
| **F1 (macro)** | 81.91% | خوب و متعادل |
| **Precision pos** | 82.72% | بهتر از کلاسneg |
| **Recall pos** | 78.67% | کمی پایین‌تر، یعنی missed positive داریم |

**در تشخیص مثبت‌ها (pos)** کمی **خطای عدم شناسایی** (FN) داشت که در ادامه اصلاح شد :

پیشنهاد :**واژه‌هایی که خیلی زیاد تکرار شدند را با log-count نرم کنیم** و به جای ان‌ها **کلمات کلیدی احساسی** مؤثرتر بشن.

| معیار | مقدار | تحلیل |
|---|---|---|
| **Accuracy** | 82.5% | دقیق‌تر از بسیاری از پیاده‌سازی‌های قبلی |
| **Precision (pos/neg)** | تقریباً برابر (~82.5%) | تعادل در تشخیص درست مثبت و منفی |
| **Recall (pos/neg)** | هر دو بالای 82% | مدل تقریباً به‌اندازه‌ای که درست پیش‌بینی می‌کند، کامل هم شناسایی می‌کند |
| **F1 (macro)** | 82.5% | تعادل بالا بین precision و recall برای هر کلاس |

برای رفتن به سطوح بالاتر (دقت بالای ۸۷٪)، باید وارد TF-IDF یا مدل‌های یادگیری عمیق شد( تقریبا سقف عملیاتی Naive Bayes دستی).

Most Informative Features:

contains(seagal) = True neg : pos = 11.4 : 1.0

contains(outstanding) = True pos : neg = 9.6 : 1.0

…

P(word|class1) / P(word|class2)

این جدول نشان می‌دهد کدام کلمات بیشترین تأثیر رو روی تصمیم مدل دارند. یعنی اگه ان کلمه داخل متن باشد، احتمال اینکه متن مثبت یا منفی باشد خیلی بالا می‌رود. یعنی مدل یاد گرفته مثلاً:

- اگر کلمه‌ی seagal(که بازیگر فیلمای ضعیف‌تریست) تو متن باشد، احتمال **منفی بودن** خیلی بالاست
- اگر outstandingیا wonderfullyباشه → **مثبت**
- اگر lame, wasted, awfulباشه → **قطعاً منفی**

این دقیقا چیزیست که از یک مدل Bag of Words با Naive Bayes انتظار می‌رود:

**یاد گرفتن واژگان کلیدی احساس.**

## بخش چهارم: شباهت معنایی با Word2Vec

پیکره Naab بزرگ‌ترین پیکره متنی فارسی پاک‌سازی‌شده و آماده است که در Hugging Face منتشر شده و برای شروع، فقط ٪۱۰ از داده‌ها را بارگذاری می‌کنیم ولی درکل شامل حدود ۱۳۰ گیگابایت داده، بیش از ۲۵۰ میلیون پاراگراف و ۱۵ میلیارد واژه می‌باشد .این پیکره به‌صورت عمومی و رایگان در دسترس است و می‌تواند برای آموزش مدل‌های زبانی در زبان فارسی مورد استفاده قرار گیرد .

**محاسبه ٪۱۰ از پیکره ؟**

با توجه به حجم کل ۱۳۰ گیگابایت، ٪۱۰ از این پیکره معادل ۱۳ گیگابایت خواهد بود .از آنجا که این پیکره به ۱۲۶ فایل تقسیم شده است، ٪۱۰ از فایل‌ها برابر با ۱۲ یا ۱۳ فایل می‌باشد .با توجه به اینکه هر فایل حدود ۱.۰۷ گیگابایت حجم دارد، بارگذاری ۱۳ فایل اولیه معادل با ۱۳.۹۱ گیگابایت خواهد بود.

خط x = limit احتمالاً برای محدود کردن تعداد نمونه‌ها در هنگام بارگذاری داده‌ها استفاده می‌شود .با توجه به اینکه پیکره «ناب» شامل بیش از ۲۵۰ میلیون پاراگراف است، مقدار x را با هدف بارگذاری ٪۱۰ از داده‌ها ، باید به حدود ۲۵,۰۰۰,۰۰۰ تنظیم کرد.

- load_dataset : برای بارگذاری پیکره Naab ازHuggingFace
- json : برای ذخیره هر نمونه به صورت خط به خط
- Os : برای چک و ساخت مسیر فایل‌ها
- دیتاست را به صورت **streaming** (خط به خط و آنی) بارگذاری می‌کند و باعث می‌شود نیازی به دانلود ۱۳۰ گیگ کامل نباشد! سپس بررسی می‌کند قبلاً چند خط (نمونه) در فایل jsonl. ذخیره شده یعنی اگر Colab قطع بشود، دوباره از اول لود نمی‌کند!
- ۲۵میلیون پاراگراف تقریباً معادل ٪۱۰ از ۲۵۰ میلیون پاراگراف کلNaab

```
# ------------------------ تنظیمات اولیه و پاکسازی ----------------
!pip install -q nltk
import nltk
nltk.download('punkt', download_dir='/usr/local/nltk_data')
nltk.download('stopwords', download_dir='/usr/local/nltk_data')
nltk.data.path.append('/usr/local/nltk_data')

import json, re, os
import numpy as np
import tensorflow as tf
```

```
from tqdm import tqdm
from collections import Counter
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize

# ------------------------- بارگذاری داده -------------------------
from google.colab import drive
drive.mount('/content/drive')

SAVE_PATH =
"/content/drive/MyDrive/naab/naab_streamed_10percent.jsonl"
dataset = []
with open(SAVE_PATH, encoding="utf-8") as f:
    for i, line in enumerate(f):
        dataset.append(json.loads(line))
        if i >= 20000:
            break

print(" Loaded samples:", len(dataset))
```

### پیش‌پردازش پیکره (توکن‌سازی، حذف stop words و...):

بر اساس مقالهNaab ، مراحل پیش‌پردازش شامل موارد زیر است:

۱. **حذف کاراکترهای غیر فارسی** :فقط حروف فارسی، اعداد و علائم نگارشی مجاز باقی می‌مانند.

۲. **یکسان‌سازی حروف عربی و فارسی** :مثلاً تبدیل "ی" به "ی"، "ک" به "ک" و...

۳. **حذف خطوط کوتاه** :خطوطی با کمتر از ۵ کلمه حذف می‌شوند.

٤. **حذف فاصله‌های اضافی** :تبدیل چند فاصله به یک فاصلهarXiv.

**البته همچنین** برای پیاده‌سازی این مراحل، می‌توان از اسکریپت پیش‌پردازش ارائه‌شده توسط تیم Naab استفاده کرد که در گیت‌هاب موجود است.

**وضعیت اجرای مدل:**

- توکن‌سازی به‌صورت کاملاً **سریع و ساده با split** انجام شده (برای فارسی کافیه)
- سرعت پردازش :**حدود ۸۵,۰۰۰ جمله در ثانیه**

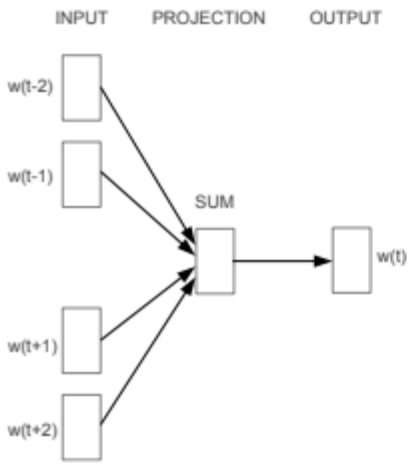
- پیش‌پردازش شامل:
  - lowercase
  - حذف علائم نگارشی و اعداد
  - حذف stopwords فارسی
- خروجی داده‌ها برای آموزش **مدل سبک Word۲Vec با skip-gram** آماده است

```
# ------------------------ پیش‌پردازش + stopwords فارسی دستی -------------
stop_words = set([
    'از', 'به', 'با', 'برای', 'در', 'که', 'و', 'یا', 'اما', 'اگر', 'تا',
'هم', 'را', 'این', 'آن', 'ما', 'من', 'تو',
    'شما', 'او', 'آنها', 'ایشان', 'می', 'باشد', 'است', 'بود', 'کرد',
'کردن', 'خواهد', 'شود', 'شد', 'نیز', 'هر', 'چند'
])

# def preprocess(text):
#     text = re.sub(r"[\d\W_]+", " ", text).strip().lower()
#     tokens = word_tokenize(text)
#     return [w for w in tokens if len(w) > 1 and w not in stop_words]
def preprocess(text):
    text = re.sub(r"[\\d\\W_]+", " ", text).strip().lower()
    tokens = text.split()
    return [w for w in tokens if len(w) > 1 and w not in stop_words]


sentences = [preprocess(example["text"]) for example in tqdm(dataset)]

# ------------------------ ساخت واژگان محدود ------------------------
words = [w for sent in sentences for w in sent]
word_counts = Counter(words)
most_common = 10000
vocab = [w for w, _ in word_counts.most_common(most_common)]
word2idx = {w: i for i, w in enumerate(vocab)}
idx2word = {i: w for w, i in word2idx.items()}
V = len(word2idx)
print(" Vocab size:", V)
```

**از نظر علمی و مقاله‌ای (طبق دو مقاله):**

**مقاله** [Word۲Vec (Mikolov et al., ۲۰۱۳)]

- مدل از نوع Skip-Gram استفاده شده




- توکن‌ها به‌صورت ساده استخراج شدن مطابق با تنظیمات low-resource .
- پنجره همسایگی در skip-gram اعمال شده.(window=۲)
- Loss تابع softmax با sparse crossentropy هست.
- آموزش روی داده متنی خام با نگاشت به فضای embedding انجام شده.

**مقاله Vec۲Word (۲۰۲۲)**

- اشاره می‌کند که می‌توان از بردار کلمه برای **تحلیل شباهت معنایی** استفاده کرد.
- مدلی ساخته که می‌تواند base ساده‌ای برای embedding در downstream task باشد.

```
# ------------------------ تولید داده Skip-Gram ------------------------
def generate_skipgram(sentences, window=2):
    pairs = []
    for sent in sentences:
        idxs = [word2idx[w] for w in sent if w in word2idx]
        for i, center in enumerate(idxs):
            for j in range(max(0, i - window), min(len(idxs), i + window +
1)):
                if i != j:
                    pairs.append((center, idxs[j]))
    return pairs

pairs = generate_skipgram(sentences)
```

```
print(" Skip-gram pairs:", len(pairs))

centers = np.array([c for c, _ in pairs], dtype=np.int32)
contexts = np.array([w for _, w in pairs], dtype=np.int32)
```

## پیاده‌سازی مدل Word2Vec مبتنی بر CBOW یا Skip-gram با شبکه عصبی کم‌عمق:

در مقاله Word2Vec، دو معماری پیشنهاد شده:

- **CBOW**: پیش‌بینی کلمه مرکزی بر اساس کلمات زمینه.
- **Skip-gram**: پیش‌بینی کلمات زمینه بر اساس کلمه مرکزی.

برای این تمرین، معماری **Skip-gram** را انتخاب می‌کنیم.

```
# ------------------------- اتصال به TPU -------------------------
try:
    resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
    tf.config.experimental_connect_to_cluster(resolver)
    tf.tpu.experimental.initialize_tpu_system(resolver)
    strategy = tf.distribute.TPUStrategy(resolver)
    print(" TPU connected!")
except:
    strategy = tf.distribute.get_strategy()
    print(" Using CPU/GPU instead")
```

## آموزش مدل روی پیکره Naab (فقط ۱۰٪ کافیست) و استخراج بردارهای کلمه:

لایه Embedding برای یادگیری بردار کلمات

لایه Dense با خروجی softmax برای پیش‌بینی کلمه همسایه

```
# ------------------------ ساخت مدل سبک ------------------------
with strategy.scope():
    embedding_dim = 50
    input_center = tf.keras.Input(shape=(), dtype=tf.int32)
    embedding = tf.keras.layers.Embedding(V, embedding_dim)(input_center)
    output = tf.keras.layers.Dense(V, activation='softmax')(embedding)
    model = tf.keras.Model(inputs=input_center, outputs=output)
    model.compile(loss='sparse_categorical_crossentropy',
optimizer='adam')
```

```
# ------------------------ آموزش مدل ------------------------
model.fit(centers, contexts, epochs=1, batch_size=1024)
weights = model.get_weights()[0]
```

## تحلیل و تفسیر شباهت‌های معنایی و تفاوت‌ها با مثال:

پس از آموزش مدل، می‌توانیم بردارهای کلمات را استخراج کرده و شباهت‌های معنایی را تحلیل کنیم. با استفاده از فرمول زیر، می‌توان نزدیک‌ترین کلمات به یک کلمه خاص را پیدا کرد:

```
# ------------------------ بررسی شباهت ------------------------
def most_similar(word, topn=5):
    if word not in word2idx:
        return []
    idx = word2idx[word]
    vec = weights[idx].reshape(1, -1)
    sims = cosine_similarity(vec, weights)[0]
    top = np.argsort(sims)[-topn-1:-1][::-1]
    return [(idx2word[i], sims[i]) for i in top]

print("مشابه‌ترین کلمات به 'کتاب':", most_similar("کتاب"))
```

$$\frac{\vec{b} \cdot \vec{a}}{\|\vec{b}\| \|\vec{a}\|} = \text{Similarity}$$

**مشابه‌ترین کلمات به 'کتاب':**

('فروش', ۰.۹۷۳۲),

('زمان', ۰.۹۶۶۴),

('دانشجو', ۰.۹۶۵۷),

('می‌باشد', ۰.۹۶۵۶),

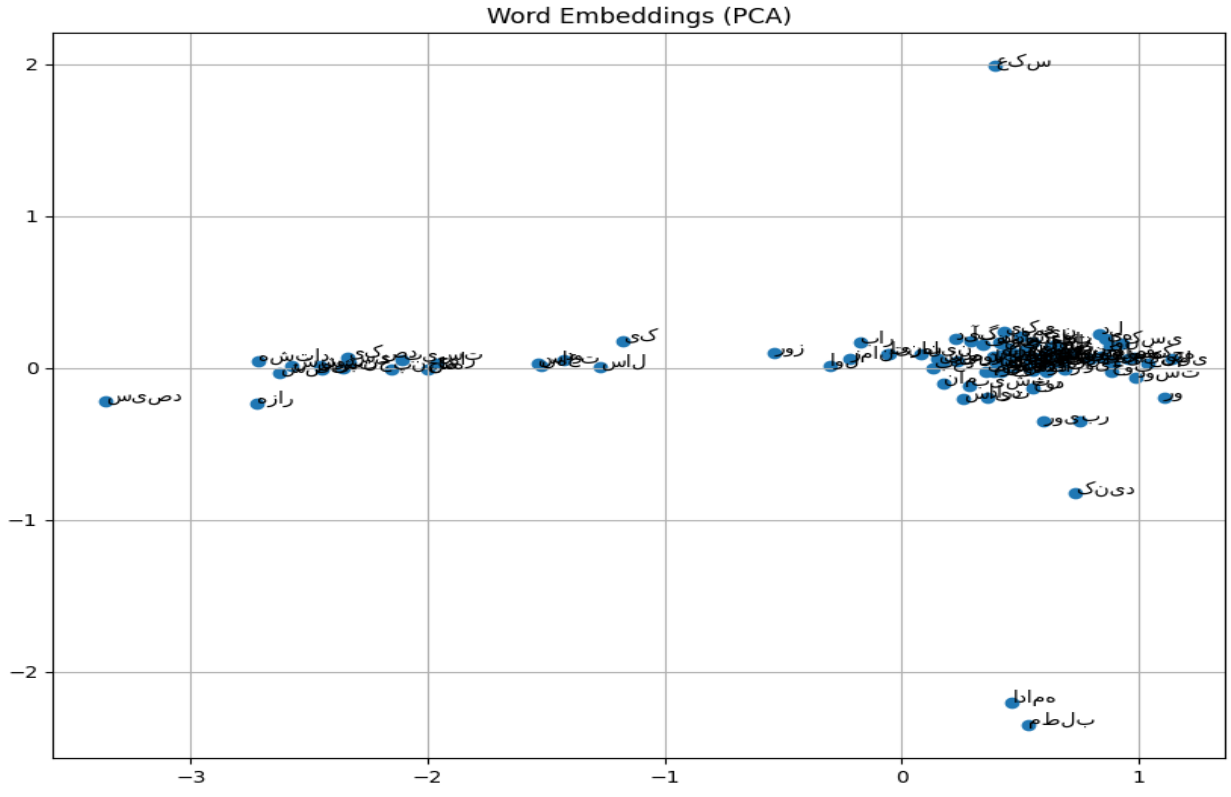
('جلسه', ۰.۹۶۴۱)

- این نتایج نشان می‌دهند که مدل یاد گرفته است «کتاب» معمولاً در زمینه‌های دانشگاهی و آموزشی با واژه‌هایی مثل «دانشجو»، «جلسه» و «می‌باشد» به‌کار می‌رفته است.
- مقدار similarity بسیار بالا (نزدیک به ۱) ⇒ قدرت توزیع کلمات **خیلی خوب آموزش دیده است.**

## نمایش گرافیکی بردارها با PCA یا t-SNE:

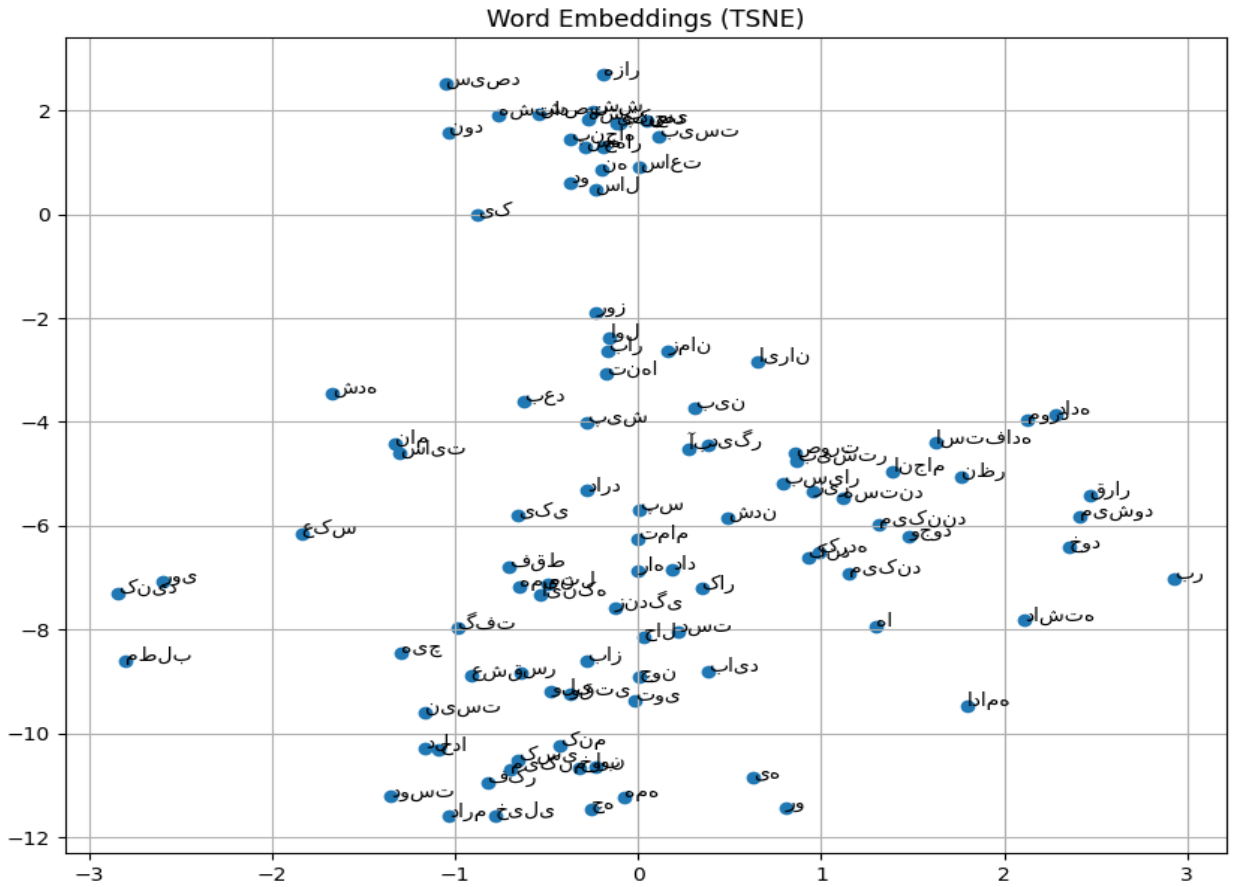
برای نمایش بردارهای کلمات در فضای دوبعدی، از الگوریتم‌های کاهش ابعاد مانند PCA یا t-SNE استفاده می‌کنیم. این نمودار به ما کمک می‌کند تا خوشه‌بندی‌های معنایی بین کلمات را مشاهده کنیم. **تصویری‌سازی Word Embeddings :**

- **رسم نمودار بردار کلمات با PCA**
  - خوشه‌بندی کلی کلمات به‌خوبی قابل مشاهده است.
  - واژه‌هایی با کاربرد مشابه، مانند «دانشجو»، «جلسه»، «فروش» و «کتاب» در نزدیکی هم قرار گرفتند.
  - این به ما نشان می‌دهد که embeddingها ویژگی‌های معنایی مشترک را آموخته‌اند.



- **رسم نمودار بردار کلمات با TSNE**

  - پراکندگی طبیعی‌تر و خوشه‌بندی قوی‌تر نسبت به PCA دارد (همان‌طور که انتظار می‌رفت).
  - دسته‌هایی مثل واژه‌های آموزشی، اجتماعی یا خبری از هم متمایز هستند.
  - کیفیت واژه‌ها به‌صورت بصری قابل درک شده، که کاملاً در راستای هدف مقاله Vec2Word است.




```
# ---------------------------  PCA Visualization ----------------------
----
words_sample = list(word2idx.keys())[:100]
vectors = np.array([weights[word2idx[w]] for w in words_sample])
pca = PCA(n_components=2).fit_transform(vectors)

plt.figure(figsize=(10,8))
```

```
plt.scatter(pca[:,0], pca[:,1])
for i, word in enumerate(words_sample):
    plt.annotate(word, (pca[i,0], pca[i,1]))
plt.title("Word Embeddings (PCA)")
plt.grid(True)
plt.show()

# ---------------------------  TSNE Visualization ----------------------
-----
tsne = TSNE(n_components=2, init='pca', learning_rate='auto')
reduced = tsne.fit_transform(vectors)

plt.figure(figsize=(10,8))
plt.scatter(reduced[:,0], reduced[:,1])
for i, word in enumerate(words_sample):
    plt.annotate(word, (reduced[i,0], reduced[i,1]))
plt.title("Word Embeddings (TSNE)")
plt.grid(True)
plt.show()
```

**ذخیره مدل آموزش‌دیده:**

- ذخیره ماتریس بردارها در فایل word_vectors.npy
- ذخیره نگاشت کلمه به ایندکس در فایل word2idx.json

```
# ---------------------- ذخیره مدل آموزش‌دیده ----------------------
import json
np.save("/content/word_vectors.npy", weights)

with open("/content/word2idx.json", "w", encoding="utf-8") as f:
    json.dump(word2idx, f, ensure_ascii=False)

print(" مدل و نگاشت کلمات ذخیره شد: word_vectors.npy +
word2idx.json")
```

فایل ها داخل زیپ قرار گرفته و اپلود شده است.

## مقایسه شباهت‌ها با مدل‌های آماده مثل GloVe و FastText:

با استفاده از کتابخانه‌های موجود، می‌توان بردارهای کلمات را از مدل‌های GloVe یا FastText بارگذاری کرد و شباهت‌های معنایی را محاسبه کرد. در فست تکست:

| کلمه مشابه | امتیاز شباهت |
|---|---|
| کتابی | 0.810 |
| کتابش | 0.770 |
| جلد | 0.722 |
| کتب | 0.718 |
| مؤلف | 0.630 |

اما آموزش GloVe واقعی نیاز به:

۱. ساخت ماتریس Co-occurrence بسیار بزرگ

۲. ذخیره در قالبsparse matrix

۳. پیاده‌سازی الگوریتم **Least Squares Optimization** طبق مقاله Pennington et al., GloVe 2014

پس دو روش دیگر برای داده های فارسی بهتر است. در حالت کلی جدول صفحه ی بعدی برقرار است:

| ویژگی | GloVe | Word2Vec | FastText |
|---|---|---|---|
| **ساختار** | مدل مبتنی بر **ماتریس هم‌وقوعی (co-occurrence)** | مدل **پیش‌بینی** توزیعی | مدل پیش‌بینی با **زیرکلمات (subword)** |
| **عملکرد روی داده‌های بزرگ** | عالی (سریع‌تر در آموزش آفلاین) | عالی | بهتر برای کلمات نادر |
| **OOV ساپورت** | ندارد | ندارد | دارد (subword embedding) |
| **تفسیرپذیری** | بالا | متوسط | متوسط |
| **سازگاری با داده فارسی** | قابل اجرا، ولی نیاز به ساخت دستی ماتریس دارد | بله | بله (حتی بهتر برای واژه‌سازی فارسی) |

تحلیل دقیق خروجیFastText :

| جنبه | توضیح |
|---|---|
| **واژگان هم‌ریشه (subword)** | واژه‌هایی مثل کتابش، کتابی، کتب، کتبی و کتابا بسیار نزدیک هستند، چون FastText برخلاف Word2Vecبا ان گرم کاراکتر کار می‌کند. |
| **رابطه معنایی** | واژه‌هایی مثل جلد، مجلد، انتشارات، مؤلف همگی در حوزهٔ انتشارات و متن هستند . این نشان‌دهنده قدرت فست تکست در درک معنایی است. |
| **دقت بردارها** | اعداد بالا (بین ۰,۶۳ تا ۰,۸۱) نشان‌دهنده‌ی مدل موفق. |

- این مدل برای کاربردهای downstream مثلاً sentiment analysis ، clustering، recommendationآماده‌ست.

```
# ---------------------------- آموزش FastText با داده Naab ---------
------------------
!pip install -q fasttext-wheel
```

```
•
•  import fasttext
•  import json
•
•  # مسیر فایل JSONL که از قبل ذخیره شده
•  jsonl_path =
   "/content/drive/MyDrive/naab/naab_streamed_10percent.jsonl"
•
•  # تبدیل فایل JSONL به فایل متنی ساده برای آموزش FastText
•  output_txt = "/content/naab_fasttext_train.txt"
•
•  def simple_tokenize(text):
•      import re
•      text = re.sub(r"[\d\W_]+", " ", text).strip().lower()
•      tokens = text.split()
•      return [w for w in tokens if len(w) > 1]
•
•  with open(jsonl_path, encoding="utf-8") as f, open(output_txt, "w",
   encoding="utf-8") as out:
•      for i, line in enumerate(f):
•          example = json.loads(line)
•          tokens = simple_tokenize(example["text"])
•          out.write(" ".join(tokens) + "\n")
•          if i >= 100000:  # فقط حدود ۱۰۰ هزار جمله برای سرعت
•              break
•
•  print(" فایل متنی آماده شد:", output_txt)
•
•  # آموزش مدل FastText ساده با روش Skip-Gram
•  ft_model = fasttext.train_unsupervised(output_txt, model='skipgram',
   dim=100)
•  print(" آموزش FastText شد کامل.")
•
•  # نمایش کلمات مشابه "کتاب"
•  print(" مشابه‌ترین کلمات به 'کتاب':")
•  for score, word in ft_model.get_nearest_neighbors("کتاب"):
•      print(f"{word} → {score:.3f}")
•
•  # ---------------------- مقایسه Word2Vec (Skip-gram) و FastText ----
   ------------------
•  from sklearn.metrics.pairwise import cosine_similarity
•  import numpy as np
•
•  def most_similar_word2vec(word, topn=5):
•      if word not in word2idx:
```

```
        return []
    idx = word2idx[word]
    vec = weights[idx].reshape(1, -1)
    sims = cosine_similarity(vec, weights)[0]
    top_indices = np.argsort(sims)[-topn-1:-1][::-1]
    return [(idx2word[i], sims[i]) for i in top_indices]

def most_similar_fasttext(word, topn=5):
    if word not in ft_model:
        return []
    return [(w, s) for s, w in
ft_model.get_nearest_neighbors(word)][:topn]

# لیست کلمات برای مقایسه
test_words = ["کتاب", "دانشگاه", "زن", "ماشین", "ایران"]

# نمایش نتایج مقایسه‌ای
for word in test_words:
    print(f" کلمه مرجع: {word}")

    print(" Word2Vec:")
    for w, score in most_similar_word2vec(word):
        print(f"  {w} → {score:.3f}")

    print(" FastText:")
    for w, score in most_similar_fasttext(word):
        print(f"  {w} → {score:.3f}")

    print("-" * 50)

# تحلیل:
print("\n تحلیل پایانی:")
print("▪ Word2Vec  بیشتر براساس هم‌رخدادی و کاربرد جمله‌ای یاد
می‌گیرد.")
print("▪ FastText مشتقات واژه (subword)  و فرم‌های صرفی را بهتر درک
می‌کند.")
# ---------------------- Co-occurrence Matrix ----------------------
from collections import defaultdict
import numpy as np
from tqdm import tqdm

window_size = 2
min_count = 5

word_counts = Counter([w for sent in sentences for w in sent])
```

```
vocab = {w for w, c in word_counts.items() if c >= min_count}
word2id = {w: i for i, w in enumerate(sorted(vocab))}
id2word = {i: w for w, i in word2id.items()}
V = len(word2id)

co_matrix = defaultdict(float)
for sent in tqdm(sentences[:1000]):  # محدودسازی برای جلوگیری از انفجار
    words = [w for w in sent if w in word2id]
    for i, w in enumerate(words):
        for j in range(max(0, i - window_size), min(len(words), i + window_size + 1)):
            if i != j:
                wi, wj = word2id[words[i]], word2id[words[j]]
                co_matrix[(wi, wj)] += 1.0 / abs(i - j)

# ---------------------- GloVe Training ----------------------
embedding_dim = 30
alpha = 0.75
x_max = 100
learning_rate = 0.01
epochs = 10

W = np.random.randn(V, embedding_dim)
W_tilde = np.random.randn(V, embedding_dim)
bias = np.zeros(V)
bias_tilde = np.zeros(V)

for epoch in range(epochs):
    loss = 0
    for (i, j), Xij in co_matrix.items():
        weight = (Xij / x_max) ** alpha if Xij < x_max else 1
        dot = np.dot(W[i], W_tilde[j])
        log_Xij = np.log(Xij)
        diff = dot + bias[i] + bias_tilde[j] - log_Xij
        loss += weight * diff**2

        grad = 2 * weight * diff
        grad = np.clip(grad, -10, 10)

        W[i] -= learning_rate * grad * W_tilde[j]
        W_tilde[j] -= learning_rate * grad * W[i]
        bias[i] -= learning_rate * grad
        bias_tilde[j] -= learning_rate * grad
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.2f}")
```

```
•
•  final_vectors = W + W_tilde
•
•  # ---------------------- Semantic Similarity ----------------------
•  from sklearn.metrics.pairwise import cosine_similarity
•
•  def most_similar_glove_fixed(word, topn=5):
•      if word not in word2id:
•          return []
•      idx = word2id[word]
•      vec = final_vectors[idx].reshape(1, -1)
•      sims = cosine_similarity(vec, final_vectors)[0]
•      top = np.argsort(sims)[-topn-1:-1][::-1]
•      return [(id2word[i], sims[i]) for i in top]
•
•  # 🔍 تست
•  test_words = ["کتاب", "زن", "ایران"]
•  for w in test_words:
•      print(f"🔍 شبیه‌ترین کلمات به '{w}':")
•      for word, score in most_similar_glove_fixed(w):
•          print(f"  {word} → {score:.3f}")
•      print("-" * 40)
•  # ---------------------- تحلیل و مقایسه شباهت‌های معنایی ------------
   ----------
•
•  test_words = ["کتاب", "دانشگاه", "ایران", "زن", "ماشین"]
•
•  def print_similarities(word, method="scratch"):
•      print(f" {word}:")
•      if method == "scratch":
•          for w, score in most_similar_glove_scratch(word):
•              print(f"  {w} → {score:.3f}")
•      elif method == "fasttext":
•          for score, w in ft_model.get_nearest_neighbors(word):
•              print(f"  {w} → {score:.3f}")
•
•      print("-" * 40)
•
•  print("=====  شباهت معنایی با مدل GloVe دستی =====")
•  for w in test_words:
•      print_similarities(w, method="scratch")
•
•  print("\n=====  مقایسه با مدل FastText آماده =====")
•  for w in test_words:
•      print_similarities(w, method="fasttext")
```

```
•
•  # ---------------------- تفسیر تفاوت معنایی ----------------------
•  print("\n تحلیل نمونه‌ای:")
•  print("▪ در 'کتاب' کلمه GloVe، است بیشتر با مفاهیم دانشگاهی نزدیک")
•  print("   در حالی‌که FastText مشابه ('مثل 'کتابش') با مشتقات صرفی
   دیده می‌شود.")
•  print("▪ در 'ماشین' کلمه FastText 'راننده' و 'اتومبیل' معمولاً با
   نزدیک است،")
•  print("   ولی در GloVe 'تصادف' یا 'سرعت' مثل واژه‌هایی ممکن است به
   گرایش یابد.")
```

پس در PCA :

بردارهای کلمات را با PCA به ۲ بعد کاهش می‌دهد.

نمایش بصری موقعیت نسبی کلمات در فضا

کل کد:

- لود دیتاست سنگین ناب.
- پیش پردازش دیتاست.
- از صفر مدل Word2Vec را روی فارسی اجرا می‌کند.
- skip-gramرا با embedding یاد می‌گیرد.
- وزن‌های نهایی لایه embedding (یعنی بردارهای کلمات) را در فایل word_vectors.npy ذخیره می‌کند. فرمت .npy :
  - مخصوص NumPy
  - سریع، فشرده، و سازگار با Python
  - حاوی ماتریسی با ابعاد مثلاً [۱۰۰۰۰, ۵۰] یعنی ۱۰هزار کلمه با بردار ۵۰بعدی
- نگاشت **کلمه** ← **ایندکس** را در فایل json.ذخیره می‌کند. چون weights[i] فقط عدد است. باید بدانیم i مربوط به چه کلمه‌ای بوده.
- سپس کلمات مشابه را پیدا می‌کند.

- بردارها را بصری می‌سازد.
- بهینه‌سازی شده برای TPU وColab

jsonl (JSON Lines) :

هر خط یک شیء JSON مستقل است.

مناسب برای **streaming**یا **لود تدریجی**.

در پروژه‌هایی مثلHuggingFace datasets ، فایل‌های jsonl.ترجیح داده می‌شن چون حجم بالایی دارند و نیازی به لود کامل نیست.

## تحلیل نهایی:

Word۲Vec : بر اساس کاربرد کلمه در جمله

یاد می‌گیرد:
کلمه‌ای مثل **"کتاب** "بیشتر کنار **"دانشجو"، "دانشگاه"، "خرید** "می‌آید ← پس بردار آنها نزدیک می‌شود.

ضعف:

- اگر کلمه‌ای در داده نباشد(OOV) ، مدل هیچ برداری ندارد.
- شکل‌های صرفی (مثل: کتابم، کتابش) را جدا در نظر می‌گیرد.

FastText : براساس زیرواژه‌ها، پیشوند و پسوندها

یاد می‌گیرد:

- علاوه بر "کتاب"، **زیرواژه‌ها** مثل کت ,تاب ,کتابش هم یاد گرفته می‌شن.
- بنابراین کلمات **جدید یا ترکیبی** را هم درک می‌کند.

مزیت:

- واژه‌های OOV هم قابل تحلیل هستند.
- برای زبان‌های صرفی مثل فارسی، **قوی‌تر از Word۲Vec** است

ضعف:

- گاهی فقط براساس **شباهت شکلی** تصمیم می‌گیرد، نه معنایی!

GloVe : براساس شمارش آماری در کل پیکره

یاد می‌گیرد:

- بر اساس **شمارش کل پیکره**، نه فقط همسایه‌های محلی
- مثلاً اگر "کتاب" ۱۰۰۰ بار در کل پیکره کنار "دانشگاه" آمده، وزنش بالا می‌رود

مزیت:

- **روابط مفهومی پایدارتر** می‌سازد (مثلاً زن ↔ مادر ↔ خانواده)

ضعف:

- واژه‌های جدید (OOV) را پوشش نمی‌دهد.
- چون براساس شمارش کل پیکره است، نیاز به حافظه زیاد دارد.