دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

گزارش فنی تمرین اول

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- علائم نقطه‌گذاری را چه هنگام باید به عنوان توکن مجزا در نظر بگیریم و در چه هنگام باید آنها را حذف کنیم؟

در پردازش زبان طبیعی(NLP) ، تصمیم‌گیری درباره نگه داشتن یا حذف علائم نقطه‌گذاری به کاربرد خاص و مسئله‌ای که در حال حل آن هستیم بستگی دارد.

- مواقعی که باید علائم نقطه‌گذاری را به‌عنوان توکن مستقل نگه داریم:

تحلیل احساسات(Sentiment Analysis)

- علائم نقطه‌گذاری، به‌ویژه علامت تعجب (!) و علامت سؤال (؟)، در تشخیص لحن و احساس متن مهم هستند. پس باید علائم را نگه داریم تا درک درستی از احساسات متن داشته باشیم.
- مثال :
  - **"I hate dogs."** جمله‌ای خبری که نشان‌دهنده‌ی نظر منفی است.
  - **"I hate dogs?"** می‌تواند نشان‌دهنده‌ی شک و تردید باشد.
  - **"I hate dogs!"** شدت نفرت را افزایش می‌دهد.

مدل‌های مولد زبان(Language Generation Models)

- در مدل‌هایی مثل **ChatGPT**، GPT-۳/۴ وBERT ، علائم نقطه‌گذاری در تولید متن طبیعی تأثیر زیادی دارند.
- حذف این علائم باعث تولید جملات نامفهوم یا غیرطبیعی می‌شود.

تحلیل نحوی(Syntactic Parsing)

- در وظایفی مانند **برچسب‌گذاری اجزای جمله (POS Tagging)** یا **تحلیل وابستگی (Dependency Parsing)**، علائم نقطه‌گذاری برای تعیین ساختار جمله مهم هستند.
- مثال :
  - **"Let's eat, grandma!"** بیایید غذا بخوریم، مادربزرگ!

- **"Let's eat grandma!"** بیایید مادربزرگ را بخوریم!
- بدون ویرگول، معنای جمله کاملاً تغییر می‌کند.

تشخیص موجودیت‌های نامدار(Named Entity Recognition – NER)

- در برخی موارد، نقطه‌گذاری می‌تواند به تشخیص نام‌ها و موجودیت‌های خاص کمک کند.
- مثال :
  - **"U.S.A."** نام کشور ایالات متحده آمریکا
  - حذف نقطه‌ها ممکن است منجر به شناسایی نادرست شود.

تحلیل متون حقوقی و رسمی

- در متون حقوقی، تغییر علامت نقطه‌گذاری می‌تواند باعث تغییر معنای قانونی شود.
- مثال :
  - قراردادی که بین دو جمله **"."**(نقطه) و **";"** تفاوت قائل شود.

- مواقعی که باید علائم نقطه‌گذاری را حذف کنیم:

تحلیل معنایی کلی(General Semantic Analysis)

- در مدل‌هایی که فقط بر تحلیل **کلمات کلیدی** تمرکز دارند، حذف علائم نقطه‌گذاری می‌تواند به سادگی داده‌ها کمک کند.
- مثال :
  - در **مدل‌های نمایه‌سازی متون** مثل TF-IDF یا Word۲Vec، علائم نقطه‌گذاری نقش مهمی ندارند.

مدل‌های طبقه‌بندی متون(Text Classification)

- اگر هدف فقط **تشخیص موضوع متن** باشد، علائم نقطه‌گذاری اهمیت کمتری دارند.
- مثال :
  - **"Machine learning is amazing!"** و **"Machine learning is amazing"** در تشخیص موضوع "هوش مصنوعی" یکسان هستند.

جستجوی اطلاعات(Information Retrieval)

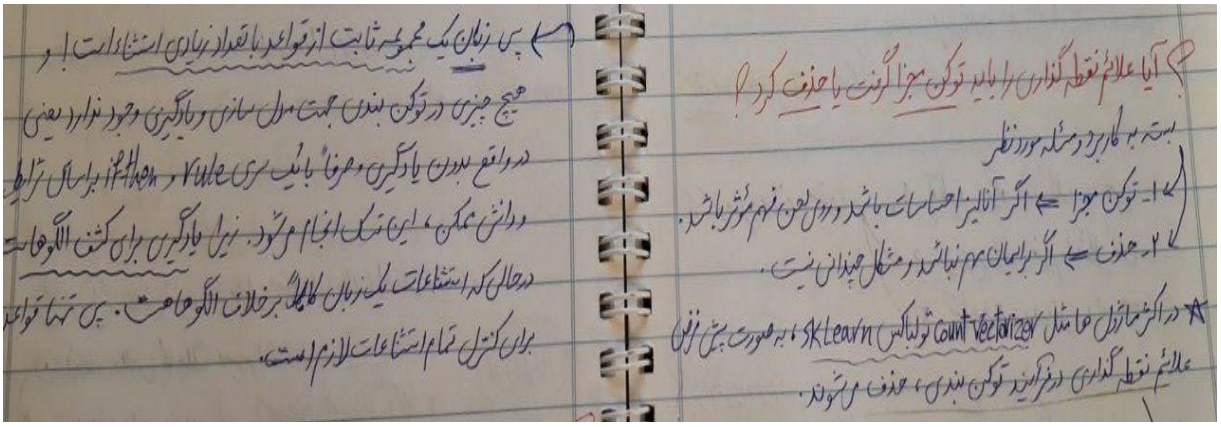
- در موتورهای جستجو، علائم نقطه‌گذاری معمولاً حذف می‌شوند زیرا کاربران در جستجوهایشان از آن‌ها استفاده نمی‌کنند.
- مثال :
  - جستجوی **"best laptops?"** و **"best laptops"** نتایج مشابهی دارند.

تبدیل متن به گفتار(Text-to-Speech - TTS)

- برخی از مدل‌های **TTS** نیاز دارند که نقطه‌گذاری حذف شود تا جریان گفتار یکنواخت‌تر باشد.

برخی روش‌های توکنایز کردن متون(Tokenization)

- در برخی توکنایزرها مثل **CountVectorizer** در **Scikit-learn** ، علائم نقطه‌گذاری به‌طور پیش‌فرض حذف می‌شوند زیرا در مدل‌های آماری نقش مهمی ندارند.



۲ - مزایا و معایب توکن بندی مبتنی بر کلمات و توکن بندی مبتنی بر کاراکتر را نام ببرید؟

- توکن‌بندی مبتنی بر کلمات(Word-level Tokenization) :

در این روش، واحد پایه برای تحلیل، **کلمات** هستند و متن به‌جای حروف، به کلمات تقسیم شده و هر کلمه به عنوان یک توکن در نظر گرفته می‌شود.

مزایا

۱. **درک معنایی بهتر**

- هر توکن دارای یک معنا و مفهوم مستقل است.
- برای مثال، کلمه **"کیبورد "**به تنهایی معنی دارد و بدون نیاز به ترکیب چندین توکن (مثل کاراکترها) قابل فهم است.

۲. **کاهش پیچیدگی مدل**

- با کاهش تعداد توکن‌ها نسبت به روش کاراکتری، مدل‌ها با ابعاد کوچکتری قابل پیاده‌سازی هستند.
- مثال: جمله‌ی **"من عاشق یادگیری NLP هستم "**در این روش ۵ توکن دارد، اما در روش کاراکتری بیش از ۲۰ توکن خواهد داشت.

۳. **پوشش بهتر زبان طبیعی**

- با داشتن وکتورهای مرتبط با کلمات، مدل می‌تواند معنای کلمات را بهتر یاد بگیرد مثل استفاده از Word۲Vec یا FastText .

٤. **سرعت پردازش بالاتر**

- چون تعداد توکن‌ها کمتر است، پردازش داده‌ها و اجرای مدل‌ها سریع‌تر انجام می‌شود.

معایب

۱. **نیاز به واژگان بسیار بزرگ(Vocabulary Size)**

- هر زبان دارای **میلیون‌ها کلمه‌ی یکتا** است که نیاز به ذخیره‌سازی در یک دیکشنری بزرگ دارند.
- مثال: یک مدل که بر روی **۱ میلیون کلمه** آموزش دیده باشد، باید بتواند تمامی آن‌ها را مدیریت کند.

۲. **مشکل با کلمات ناشناخته(Out of Vocabulary - OOV)**

- اگر مدلی یک کلمه‌ی جدید مانند **"متاورس "**را در داده‌های آموزشی ندیده باشد، نمی‌تواند آن را پردازش کند.
- راه‌حل‌هایی مانند **subword tokenization** یعنی BPE و WordPiece برای حل این مشکل به کار می‌روند.

۳. **مدیریت نام‌های خاص و شکل‌های گرامری سخت است**

- مثلاً، کلمات **"کتاب "**و **"کتاب‌ها "**به عنوان دو توکن جدا شناخته می‌شوند، درحالی‌که به یک مفهوم اشاره دارند.
- روش‌هایی مانند **lemmatization** و **stemming** می‌توانند این مشکل را کاهش دهند.

- توکن‌بندی مبتنی بر کاراکتر(Character-level Tokenization) :

در این روش، هر **حرف یا کاراکتر** یک توکن محسوب می‌شود. به‌عنوان مثال، کلمه **"کیبورد "**به شش توکن **"ک"، "ی"، "ب"، "و"، "ر"، "د "**تقسیم خواهد شد.

مزایا

۱. **واژگان کوچک‌تر و مدیریت آسان‌تر**

- برخلاف مدل‌های مبتنی بر کلمه که نیاز به **واژگان میلیونی** دارند، در اینجا تنها **چند صد کاراکتر** (شامل حروف، اعداد و علائم) وجود دارند.
- برای مثال، در زبان انگلیسی تنها **۲۶ حرف** + **اعداد و نشانه‌ها** وجود دارد که بسیار کمتر از تعداد کلمات ممکن است.

۲. **حل مشکل کلمات ناشناخته(OOV)**

- چون همه کلمات از ترکیب چندین کاراکتر ساخته شده‌اند، حتی اگر کلمه‌ای جدید باشد، مدل هنوز می‌تواند معنای آن را یاد بگیرد.

۳. **پوشش زبان‌های مختلف و ترکیبات نادر**

- در زبان‌هایی مانند **چینی، فارسی و عربی** که دارای ترکیبات پیچیده‌ای هستند، مدل‌های کاراکتری می‌توانند بهتر عمل کنند.

٤. **عدم وابستگی به فاصله‌ها و علائم نقطه‌گذاری**

- در متونی که دارای خطاهای تایپی یا تغییرات گرامری زیاد هستند، این روش انعطاف بیشتری دارد.

معایب

۱. **درک معنایی ضعیف‌تر**

- مدل نیاز دارد که از **ترکیب کاراکترها، معنا را استخراج کند**، که این کار بسیار سخت‌تر از کار با کلمات است.

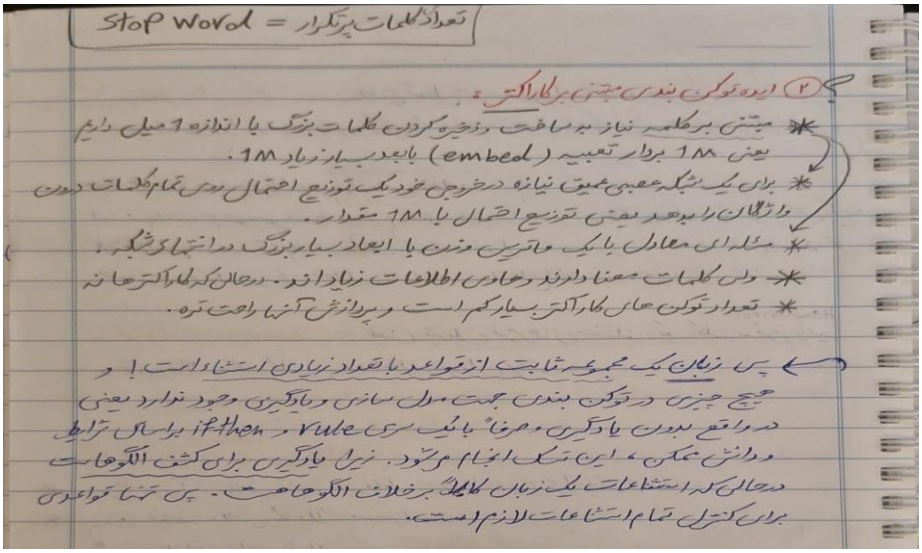
- برای مثال، کاراکتر **"d"** در کلمات **"dog", "duck", "door"** به‌کار رفته، اما هیچ مفهوم خاصی به‌تنهایی ندارد.

۲. **افزایش چشمگیر طول توکن‌ها و نیاز به پردازش سنگین‌تر**

- جمله **"من عاشق یادگیری NLP هستم "** در این روش بیش از **۲۰ توکن** خواهد داشت، که باعث افزایش زمان پردازش و سختی یادگیری مدل می‌شود.

۳. **افزایش حجم ماتریس‌های ورودی در شبکه‌های عصبی**

- مدل‌های مبتنی بر کاراکتر نیاز دارند که احتمال توزیع روی تمامی توکن‌های ممکن را یاد بگیرند.
- به دلیل اینکه تعداد توکن‌ها زیاد می‌شود، پردازش مدل کندتر شده و نیاز به محاسبات بیشتری دارد.



۳ - به طور کامل توضیح دهید در صورتی که فقط از توکن بندی مبتنی بر فاصله استفاده کنیم چه مشکلاتی ممکن است به وجود آید؟

توکن‌بندی مبتنی بر فاصله (Whitespace Tokenization) یکی از ساده‌ترین روش‌های توکن‌بندی است که در آن متن بر اساس **فضاهای خالی** (Whitespace) شکسته می‌شود و هر کلمه بین دو فاصله به عنوان یک

توکن در نظر گرفته می‌شود .با وجود سادگی و سرعت بالای این روش، در بسیاری از موارد **مشکلات جدی** ایجاد می‌کند که می‌تواند عملکرد مدل‌های NLP را به‌شدت تحت تأثیر قرار دهد:

- ترکیب نشدن علائم نقطه‌گذاری با کلمات اصلی

در توکن‌بندی مبتنی بر فاصله، علائم نقطه‌گذاری مانند **نقطه (.)، ویرگول (،)، علامت سؤال (؟)، علامت تعجب (!)** به عنوان بخشی از کلمه‌ی قبلی یا یک کلمه‌ی مستقل در نظر گرفته می‌شوند.

مثال:

- اگر جمله‌ی زیر را در نظر بگیریم:

"I hate dogs."

توکن‌های استخراج‌شده به این صورت خواهد بود:

["I", "hate", "dogs."]

این در حالی است که جمله‌ی:

"I hate dogs?"

به توکن‌های:

["I", "hate", "dogs?"] تبدیل می‌شود.

مشکل:

کلمات **"dogs."** و **"dogs?"**دو توکن متفاوت شناخته می‌شوند، در حالی که هر دو به همان واژه‌ی "dogs" اشاره دارند. این باعث افزایش اندازه واژگان و ایجاد ابهام در مدل‌های یادگیری ماشین می‌شود.

راه‌حل:

می‌توان از روش‌های پیش‌پردازش مانند حذف یا جداسازی علائم نقطه‌گذاری از کلمات استفاده کرد.

- افزایش اندازه واژگان(Vocabulary Size)

از آنجایی که توکن‌بندی مبتنی بر فاصله، علائم نگارشی را از کلمات جدا نمی‌کند، تعداد زیادی کلمه‌ی متفاوت اما مشابه در واژگان ایجاد می‌شود.

مثال:

- کلمه‌ی **"hello"** ممکن است در جملات زیر به شکل‌های مختلف ظاهر شود:

"Hello."

"Hello!"

"Hello?"

در این صورت، در مدل NLP این کلمات به صورت توکن‌های جداگانه شناخته می‌شوند.

در حالی که همه‌ی آن‌ها به یک مفهوم اشاره دارند، اما مدل آن‌ها را کلمات متفاوتی در نظر می‌گیرد.

مشکل:

- تعداد توکن‌ها در دیکشنری مدل افزایش می‌یابد.
- نیاز به داده‌ی آموزشی بیشتری برای یادگیری معانی مشابه کلمات با علائم مختلف.
- پیچیدگی پردازش و ذخیره‌سازی مدل بیشتر می‌شود.

راه‌حل:

استفاده از **پیش‌پردازش متون** و **حذف علائم نگارشی از کلمات** یا استفاده از روش‌های پیشرفته‌تر توکن‌بندی مانند **WordPiece** و **BPE** .

- عدم شناسایی صحیح کلمات ترکیبی و چندبخشی

توکن‌بندی مبتنی بر فاصله نمی‌تواند **کلمات ترکیبی** (مانند اسامی خاص، عبارات چندکلمه‌ای، و اصطلاحات) را به درستی تشخیص دهد.

مثال:

- در نظر بگیرید عبارت **"New York City"** را با این روش پردازش کنیم:

["New", "York", "City"]

در این حالت، مدل ممکن است هر کلمه را **جداگانه** پردازش کند و تشخیص ندهد که این سه کلمه **یک نام خاص** هستند.

مشکل:

- کاهش دقت در **تشخیص موجودیت‌های نامدار Named Entity - NER) Recognition)**.
- اشتباه در تحلیل معنایی جملات.

راه‌حل:

استفاده از **توکن‌بندی مبتنی بر n-gram** یا **مدل‌های پیشرفته مثل WordPiece** که می‌توانند عبارات چندکلمه‌ای را به درستی شناسایی کنند.

- تأثیر منفی روی تحلیل احساسات(Sentiment Analysis)

در تحلیل احساسات، علائم نگارشی مانند **علامت تعجب** (!) **و علامت سؤال (?)** نقش مهمی دارند و می‌توانند شدت احساس را تغییر دهند. اما در توکن‌بندی مبتنی بر فاصله، این علائم معمولاً به درستی جدا نمی‌شوند.

مثال:

- جمله‌ی **"I love this!"** و **"I love this."** دارای احساسات متفاوتی هستند، اما در این روش هر دو به **["I", "love", "this!"]** و **["I", "love", "this."]** تبدیل می‌شوند که تفاوت معنایی میان این دو جمله را از بین می‌برد.

مشکل:

- از دست رفتن اطلاعات احساسی موجود در جمله.
- کاهش دقت در تحلیل احساسات.

راه‌حل:

استفاده از **مدل‌های ترکیبی که علائم نگارشی را جداگانه تحلیل می‌کنند.**

- مشکل در زبان‌هایی که کلمات با فاصله جدا نمی‌شوند

توکن‌بندی مبتنی بر فاصله برای زبان‌هایی که کلمات بدون فاصله نوشته می‌شوند، **کاملاً ناکارآمد** است.

مثال:

در زبان‌هایی مانند:

- **چینی** (汉字)
- **ژاپنی** (日本語)
- **فارسی و عربی (در برخی موارد)**

کلمات به‌طور طبیعی بدون فاصله از هم جدا نمی‌شوند، بنابراین مدل نمی‌تواند به درستی تشخیص دهد که **هر کلمه از کجا شروع و کجا تمام می‌شود.**

مشکل:

- در زبان‌هایی مانند چینی، **نیاز به روش‌های پیچیده‌تر مانند Word Segmentation** داریم.
- توکن‌بندی مبتنی بر فاصله در این زبان‌ها عملکرد ضعیفی دارد.

راه‌حل:

استفاده از **مدل‌های یادگیری عمیق برای شناسایی مرزهای کلمات** یا **روش‌های مبتنی برsubword مانندBPE** .

- تأثیر منفی روی یادگیری مدل‌هایNLP

توکن‌بندی مبتنی بر فاصله باعث می‌شود که مدل‌های یادگیری ماشین و شبکه‌های عصبی نیاز به **ابعاد بسیار بزرگی برای بردارهای ویژگی** داشته باشند.

مثال:

- فرض کنید مدل ما باید یک ماتریس وزن برای هر کلمه در نظر بگیرد.

- به دلیل اینکه **کلمات مشابه با علائم مختلف جدا شده‌اند، بردارهای زیادی باید آموزش داده شوند.**
- این امر باعث افزایش پیچیدگی محاسباتی مدل می‌شود.

مشکل:

- افزایش نیاز به **داده‌های آموزشی بیشتر.**
- افزایش **هزینه محاسباتی و حافظه.**
- افزایش **ابعاد ماتریس وزن شبکه عصبی.**

راه‌حل:

استفاده از روش‌های مدرن مانند **WordPiece یا SentencePiece** برای کاهش پیچیدگی مدل.

۴ - اگر کلمه‌ای مانند watched را به واحدهای زیر کلمه‌ای توکنایز نکنیم چه مشکلاتی پیش می‌آید؟

بهترین رویکرد در NLP مدرن، استفاده از توکن‌بندی زیرکلمه‌ای است که در مدل‌هایی مانندBERT ، GPT و T5پیاده‌سازی شده است.این روش باعث کاهشOOV ، بهبود تعمیم‌دهی، کاهش اندازه واژگان و بهبود دقت مدل‌های NLP می‌شود. اگر کلمه‌ای مانند **"watched"**را به واحدهای **زیرکلمه‌ای (subword units)** تقسیم نکنیم و آن را به‌عنوان یک توکن کامل در نظر بگیریم، ممکن است:

- افزایش تعداد کلمات ناشناخته(Out of Vocabulary - OOV)

در روش **توکن‌بندی مبتنی بر کلمات**، مدل فقط می‌تواند کلماتی را پردازش کند که در **واژگان (Vocabulary)**آن دیده شده باشند. اما اگر **"watched"**یک توکن مستقل باشد و در داده‌های آموزشی مدل وجود نداشته باشد، مدل نمی‌تواند آن را تشخیص دهد و در نتیجه با مشکل **OOV**مواجه می‌شود. پس اگر مدل روی یک مجموعه داده آموزشی **"watch"**را دیده باشد، اما **"watched"**را ندیده باشد و اگر **"watched"**را به عنوان یک توکن مستقل در نظر بگیریم، مدل نمی‌تواند معنای آن را از **"watch"** استنتاج کند.

مشکل:

- مدل قادر به پردازش کلمات جدید یا تصریف‌یافته نخواهد بود.
- این مشکل در زبان‌هایی که **تعداد زیادی تصریف دارند** (مانند فارسی و عربی) بسیار بزرگ‌تر است.

راه‌حل:

- استفاده از **توکن‌بندی زیرکلمه‌ای (Subword Tokenization)** مانند **Byte Pair Encoding (BPE)**یا **WordPiece**.
- این روش‌ها کلمات را به **اجزای کوچک‌تر** تقسیم می‌کنند تا مدل بتواند ساختار آن‌ها را بهتر درک کند.

- عدم درک ارتباط بین کلمات هم‌ریشه

اگر مدل **"watch"**و **"watched"**را به‌عنوان **توکن‌های مستقل** یاد بگیرد، ممکن است نتواند رابطه بین این دو کلمه را درک کند. در حالی که از نظر زبانی، **"watched"**شکل گذشته‌ی **"watch"**است و این دو مفهوم مشترکی دارند.

- اگر مدل **"watch"**و **"watched"**را به عنوان دو کلمه‌ی کاملاً متفاوت یاد بگیرد، در تحلیل زبان نمی‌تواند درک کند که هر دو به **عمل دیدن** مربوط هستند.

مشکل:

- کاهش دقت مدل در **تحلیل معنایی جملات.**
- مدل برای یادگیری کلمات مشابه **باید تعداد زیادی نمونه داشته باشد.**

راه‌حل:

- **توکن‌بندی زیرکلمه‌ای** می‌تواند این مشکل را حل کند :

watched → ["watch", "ed"]

در این روش، مدل متوجه می‌شود که **"watched"**از **"watch"**ساخته شده و می‌تواند ارتباط معنایی بین این دو را درک کند.

- افزایش اندازه واژگان و نیاز به داده‌های آموزشی بیشتر

اگر همه کلمات را بدون تجزیه به زیرکلمه در نظر بگیریم، مدل مجبور است **هر تصریف از یک کلمه را به‌طور جداگانه یاد بگیرد.** این کار منجر به **افزایش اندازه‌ی واژگان (Vocabulary)** شده و نیاز به **حجم بالای داده‌ی آموزشی** دارد.

- بدون توکن‌بندی زیرکلمه‌ای، واژگان مدل شامل این موارد خواهد بود :

watch, watches, watched, watching

- اما اگر از **Subword Tokenization**استفاده کنیم، فقط کافی است مدل این بخش‌ها را یاد بگیرد :

["watch", "es"], ["watch", "ed"], ["watch", "ing"]

در این حالت، مدل می‌تواند **از یک ریشه برای تولید سایر اشکال کلمه استفاده کند.**

مشکل:

- نیاز به **واژگان بسیار بزرگ** برای پوشش تمام تصریف‌ها.
- **افزایش مصرف حافظه و پردازش** برای مدل‌های.NLP

راه‌حل:

- استفاده از **BPE**، **WordPiece**یا **SentencePiece**برای کاهش اندازه واژگان.

- کاهش توانایی مدل در تعمیم دادن(Generalization)

یکی از اهداف اصلی مدل‌های NLP این است که **بتوانند تعمیم بدهند** و مفاهیم جدید را بر اساس داده‌های قبلی درک کنند. اما اگر **"watched"**را به عنوان یک توکن مستقل نگه داریم، مدل نمی‌تواند شباهت آن را به **"watch"**تشخیص دهد.

مثال:

- اگر جمله‌ی **"I watched a movie"**در داده‌های آموزشی مدل نباشد، اما جمله‌ی **"I watch movies"** باشد، مدلی که **"watched"**را به عنوان یک توکن مستقل می‌شناسد، قادر به درک **ارتباط بین این دو جمله** نخواهد بود.

مشکل:

- مدل نمی‌تواند یادگیری را تعمیم دهد و برای هر تصریف، نیاز به نمونه‌های جدید دارد.
- در زبان‌هایی که تصریف‌های زیادی دارند، مانند **عربی، ترکی، فارسی**، این مشکل بسیار جدی‌تر است.

راه‌حل:

- اگر از **Subword Tokenization** استفاده کنیم، مدل می‌تواند رابطه‌ی **"watch"**و **"watched"** را درک کند :

watched → ["watch", "ed"]

<u>- کاهش دقت در مدل‌های ترجمه ماشینی(Machine Translation)</u>

در **ترجمه ماشینی**، مدل باید ارتباط بین کلمات را حفظ کند. اما اگر کلمات را **بدون زیرکلمه‌سازی** پردازش کنیم، مدل نمی‌تواند به درستی ریشه‌ی کلمات را تشخیص دهد و ترجمه‌های نادرستی ایجاد خواهد کرد.

مثال:

- اگر **"watched"**به عنوان یک توکن مستقل یاد گرفته شود، مدل نمی‌فهمد که با **"watch"**مرتبط است.
- اما اگر مدل از **Subword Tokenization**استفاده کند :

watched → ["watch", "ed"]

مدل می‌تواند بفهمد که **"watch"**ریشه‌ی فعل است و ترجمه بهتری ارائه دهد.

مشکل:

- مدل‌های ترجمه ممکن است **کلمات با تصریف‌های مختلف را به اشتباه ترجمه کنند.**

راه‌حل:

- استفاده از توکن‌بندی زیرکلمه‌ای در **Transformer-based models** مانند **BERT**، **GPT**، **T5**.

- ناسازگاری در مدل‌های یادگیری عمیق

بسیاری از **مدل‌های مبتنی بر ترانسفورمرها (Transformers)** مانندBERT ، GPTو T5 از **توکن‌بندی زیرکلمه‌ای** استفاده می‌کنند. اگر ما **"watched"**را بدون شکستن به زیرکلمه نگه داریم، نمی‌توانیم از **مزایای این مدل‌ها** بهره‌مند شویم. **بهترین راه‌حل برای NLP مدرن:**

- استفاده از **WordPiece** مثل**BERT**
- استفاده از **Byte Pair Encoding (BPE)** مثل**GPT**
- استفاده از **SentencePiece** مانند **T5** و**mBERT**

۵ – درباره‌ی روش‌های تعبیه سازی کلمات(embedding word)Glove و Elmo تحقیق کنید و تفاوت‌های آن‌ها را شرح دهید؟

در پردازش زبان طبیعی(NLP) ، روش‌های **Word Embedding** برای تبدیل کلمات به **بردارهای عددی** استفاده می‌شوند تا مدل‌های یادگیری ماشین بتوانند معنای کلمات را در یک فضای ریاضیاتی درک کنند. دو روش مهم در این زمینه **GloVe** و **ELMo** هستند که هر کدام از آن‌ها دارای رویکرد و کاربردهای متفاوتی هستند. اگر نیاز به یک روش سریع و کم‌هزینه برای پردازش متن باشد، GloVe انتخاب بهتری است.اگر نیاز به درک دقیق‌تر متن و معنای کلمات در بافت جمله باشد ، ELMo انتخاب مناسبی است. با پیشرفت مدل‌های جدیدتر مانند BERT وGPT ، روش‌های مبتنی بر ترانسفورمر جایگزین ELMo شده‌اند. امروزه، بسیاری از پروژه‌های NLP به جای GloVe وELMo ، ازBERT ، GPTو word2vec استفاده می‌کنند. در کاربردهای مدرنNLP ، روش‌های مبتنی بر ترانسفورمر مانند BERT و T5 جایگزین روش‌های قدیمی‌تر مانندELMo و GloVeشده‌اند. حال مواردی در خصوص این دو روش به شرح زیر است:

GloVe (Global Vectors for Word Representation) -

GloVeیک 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 اصلیGloVe :

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)**

- درGloVe ، 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"**.

ELMo (Embeddings from Language Models) -

ELMo یک روش **پویا (Dynamic Embedding)** است که توسط **دانشگاه آلن برای هوش مصنوعی (AI2)** در سال ۲۰۱۸ ارائه شد. این روش برخلاف GloVe از **مدل‌های مبتنی بر یادگیری عمیق** استفاده می‌کند.ویژگی‌های اصلی ELMo :

۱. **بر پایه مدل‌های یادگیری عمیق و شبکه‌های عصبی بازگشتی(BiLSTM)**

- ELMo از **یک مدل دوبخشی LSTM (BiLSTM)** استفاده می‌کند تا بتواند معنای کلمات را بر اساس **متن و جایگاه آن‌ها در جمله** یاد بگیرد.

۲. **پویا بودن(Dynamic Embedding)**

- بر خلاف GloVe ، در ELMo بردار هر کلمه **متناسب با متن تغییر می‌کند.**
- مثال: کلمه‌ی **"bank"** در جمله‌ی **"He deposited money in the bank"** برداری متفاوت از جمله‌ی **"He sat on the bank of the river"** خواهد داشت.

۳. **استفاده از لایه‌های عمیق برای استخراج ویژگی‌های زبانی**

- بردارهای ELMo از **لایه‌های مختلف شبکه عصبی استخراج می‌شوند :**
  - لایه اول: اطلاعات سطحی (مثلاً ریشه‌ی کلمه و صرف آن)
  - لایه دوم: اطلاعات معنایی و نحوی (مثلاً ساختار گرامری جمله)
  - لایه سوم: اطلاعات معنایی سطح بالا (مثلاً بافت و مفهوم کلی جمله)

٤. **یادگیری وابسته به بافت(Contextualized Embedding)**

- چون مدل از **کل جمله** برای تعیین بردار هر کلمه استفاده می‌کند، می‌تواند تفاوت معنایی کلمات را متوجه شود.

٥. **بهبود عملکرد درNLP**

- ELMoتوانست **دقت مدل‌های NLP را در وظایفی مثلNER ، تحلیل احساسات، و ترجمه ماشینی افزایش دهد.**

مشکلاتELMo

- **هزینه‌ی محاسباتی بالا**

- به دلیل استفاده از **BiLSTM**و نیاز به پردازش کل متن، اجرای ELMo **نسبت بهGloVe بسیار کندتر و سنگین‌تر است.**

- **قدیمی شدن در برابر مدل‌های جدیدتر) مانند BERT و (GPT**

- با معرفی مدل‌های **ترانسفورمر (Transformers)** مانند **BERT وGPT** ، روش ELMo کمتر استفاده می‌شود.

۶ – دلیل حذف ایست واژه‌هاstop words در پیش‌پردازش متون چیست؟

ایست‌واژه‌ها (Stop Words) کلماتی هستند که در یک زبان به طور مکرر استفاده می‌شوند اما **ارزش معنایی خاصی ندارند** و بیشتر برای ساختار دستوری جملات کاربرد دارند. در زبان انگلیسی کلماتی مانند **"the"، "is"، "and"، "it"**و در زبان فارسی کلماتی مانند **"و"، "به"، "در"، "آن "**ایست‌واژه محسوب می‌شوند. در **پردازش زبان طبیعی(NLP)** ، حذف ایست‌واژه‌ها یکی از مراحل **پیش‌پردازش متون** است که به بهبود دقت و سرعت پردازش کمک می‌کند. دلایل حذف این کلمات :

۱ .کاهش حجم پردازش و افزایش کارایی مدل‌ها

یکی از مهم‌ترین دلایل حذف ایست‌واژه‌ها، **کاهش تعداد توکن‌ها** و در نتیجه کاهش حجم داده‌های پردازشی است.

- در پردازش متون، هر **کلمه یک توکن محسوب می‌شود** و نگه داشتن ایست‌واژه‌ها باعث **افزایش غیرضروری حجم داده‌ها** می‌شود.
- مدل‌های یادگیری ماشین برای پردازش داده‌های متنی از **ماتریس‌های برداری بزرگ** استفاده می‌کنند. حذف ایست‌واژه‌ها موجب **کاهش ابعاد این ماتریس‌ها** و در نتیجه **افزایش سرعت پردازش و کاهش مصرف حافظه** می‌شود.

مثال:

متن اصلی:

"The movie is not good at all."

پس از حذف ایست‌واژه‌ها:

"movie good"

- جمله از ۶ کلمه به ۲ کلمه کاهش پیدا کرده است.
- مدل NLP به جای پردازش کلمات بی‌اهمیت، روی کلمات کلیدی تمرکز می‌کند.

۲ .افزایش دقت مدل‌های یادگیری ماشین

نگه داشتن ایست‌واژه‌ها ممکن است **نویز (Noise) در داده‌ها ایجاد کند** و موجب کاهش دقت مدل‌های یادگیری ماشین شود.

ایست‌واژه‌ها در **تقریباً تمام متون موجودند** و باعث می‌شوند که **تفاوت بین متون کاهش یابد.**

- **نگه داشتن این کلمات ممکن است مدل را به اشتباه بیندازد،** زیرا این کلمات اطلاعات مهمی در مورد **محتوای واقعی متن** ارائه نمی‌کنند.

مثال:

متن ۱:

"He is a great actor."

متن ۲:

"He is an amazing director."

اگر ایستواژه‌ها را نگه داریم، مدل ممکن است **"He is a"**را ویژگی مهمی بداند، در حالی که این قسمت از جمله **بی‌ارزش** است. اما با حذف ایستواژه‌ها، تفاوت بین **"actor"**و **"director"**بهتر مشخص می‌شود.

۳ .بهبود دقت در مدل‌های دسته‌بندی متن(Text Classification)

- در دسته‌بندی متن (مثلاً تشخیص **احساسات کاربران در شبکه‌های اجتماعی** (حذف ایستواژه‌ها باعث می‌شود مدل **روی کلمات کلیدی تمرکز کند.**

مثال: تحلیل احساسات(Sentiment Analysis)

متن اولیه:

"The movie was not good at all."

پس از حذف ایستواژه‌ها:

"movie good"

- کلمات **"not"**و **"at all"** حذف شده‌اند، که می‌تواند **مشکل ایجاد کند.**
- این روش در تحلیل احساسات **ممکن است منجر به اشتباه در پیش‌بینی شود** در این مثال، مدل ممکن است جمله را مثبت در نظر بگیرد.
- پس در چنین مواردی، **حذف ایستواژه‌ها همیشه توصیه نمی‌شود.**

۴ .بهبود دقت در جستجوی اطلاعات(Information Retrieval)

- موتورهای جستجو (مانند گوگل) معمولاً ایستواژه‌ها را **نادیده می‌گیرند** تا روی کلمات **کلیدی جستجو** تمرکز کنند.

مثال:

فرض کنید کاربر در گوگل جستجو کند:

"What is the best way to learn Python?"

اگر ایست‌واژه‌ها حذف شوند:

"best way learn Python"

- موتور جستجو می‌تواند نتایج مرتبط‌تر را نمایش دهد.
- کاهش حجم پردازش در مقیاس وسیع.(Big Data)

۵ .تأثیر روی خوشه‌بندی و مدل‌های موضوع‌یابی(Topic Modeling)

- روش‌های **خوشه‌بندی متن** مانند **Latent Dirichlet Allocation (LDA)**و **k-Means** بر اساس **فراوانی کلمات کلیدی** کار می‌کنند.
- نگه داشتن ایست‌واژه‌ها باعث می‌شود مدل به جای کلمات کلیدی، روی کلمات بی‌ارزش تمرکز کند.

۶ .چه زمانی نباید ایست‌واژه‌ها را حذف کنیم؟

در بعضی از کاربردهای NLP حذف ایست‌واژه‌ها ممکن است مضر باشد.

۱ .ترجمه ماشینی(Machine Translation)

- در ترجمه، ایست‌واژه‌ها اهمیت دارند، زیرا حذف آن‌ها باعث تغییر معنی جمله می‌شود.

۲ .پاسخ به سوالات(Question Answering)

- در این کاربرد، کلمات ربط مانند "چرا"، "چگونه"، "کجا "اهمیت زیادی دارند. حذف آن‌ها ممکن است باعث کاهش دقت در درک سوال شود.

۳ .خلاصه‌سازی متن(Text Summarization)

- حذف بیش از حد ایست‌واژه‌ها ممکن است باعث تغییر ساختار متن و کاهش خوانایی خلاصه شود. بهترین روش این است که ایست‌واژه‌ها را **هوشمندانه انتخاب** و فقط کلماتی که واقعاً بی‌ارزش هستند را حذف کنیم.

۷ .ابزارهای پرکاربرد برای حذف ایست‌واژه‌ها

چندین کتابخانه محبوب در NLP دارای لیست‌های پیش‌فرض ایست‌واژه‌ها هستند:

- **NLTK** (Natural Language Toolkit)
- **spaCy**
- **Gensim**
- **Scikit-Learn**

این کتابخانه‌ها امکان حذف ایست‌واژه‌ها را به راحتی فراهم می‌کنند، اما **لیست ایست‌واژه‌ها همیشه قابل تنظیم است** و **باید متناسب با کاربرد تغییر داده شود.**

بخش دوم: پیش‌پردازش

```
!pip install hazm nltk wordcloud matplotlib

import nltk
# nltk.download('all')

print("All packages are successfully installed!")
```

متن فارسی:

```
# Import necessary libraries
import re
import os
import urllib.request
import hazm  # برای پردازش فارسی
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Load Persian text
file_path_fa = "/content/hp_fa.txt"
```

```
with open(file_path_fa, "r", encoding="utf-8") as file:
    text_fa = file.read()

# Persian font path
persian_font_path = "/content/Vazir.ttf"

# Try to download font if it doesn't exist
if not os.path.exists(persian_font_path):
    try:
        print("Downloading Persian font...")
        url = "https://github.com/rastikerdar/vazir-
font/releases/download/v33.003/Vazir-Regular.ttf"
        urllib.request.urlretrieve(url, persian_font_path)
        print("Persian font downloaded successfully!")
    except Exception as e:
        print("Font download failed:", e)
        print("Please upload a Persian font file (e.g., Vazir.ttf)
manually to /content/")

# Persian stopwords list
persian_stopwords = set(hazm.stopwords_list())

# Initialize Hazm tools
normalizer = hazm.Normalizer()
lemmatizer = hazm.Lemmatizer()

# Preprocessing function for Persian text
def preprocess_farsi(text):
    # 1. حذف فضاهای خالی اضافه
    text = re.sub(r'\s+', ' ', text)

    # 2. تجزیه متن به جملات
    sentences = hazm.sent_tokenize(text)

    processed_sentences = []

    for sentence in sentences:
        # 3. نرمال‌سازی متن
        sentence = normalizer.normalize(sentence)

        # 4. تجزیه جملات به کلمات
        words = hazm.word_tokenize(sentence)

        # 5. حذف علائم نگارشی
        words = [re.sub(r'[^\w\s]', '', word) for word in words]
```

```
        # 6. حذف ایست واژه ها
        words = [word for word in words if word not in persian_stopwords]

        # 7. حذف ایموجی‌ها
        words = [re.sub(r'[^\w\s]', '', word) for word in words]  # حذف
ایموجی‌ها با فیلترگذاری روی کلمات غیرمتنی

        # 8. اعمال فرآیند لم‌سازی
        words = [lemmatizer.lemmatize(word) for word in words]

        # ذخیره جملات پردازش‌شده
        processed_sentences.append(" ".join(words))

    return " ".join(processed_sentences)

# Apply preprocessing
processed_text_fa = preprocess_farsi(text_fa)

# Save processed text to file
processed_fa_path = "/content/NewProcessed_hp_fa.txt"

with open(processed_fa_path, "w", encoding="utf-8") as file:
    file.write(processed_text_fa)

# Generate WordCloud for Persian text only if font exists
if os.path.exists(persian_font_path):
    wordcloud_fa = WordCloud(width=800, height=400,
font_path=persian_font_path).generate(processed_text_fa)
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud_fa, interpolation="bilinear")
    plt.axis("off")
    plt.title("Persian WordCloud")
    plt.show()
else:
    print("Persian font not available. Skipping Persian WordCloud...")

# Provide file location
print(f"Processed Persian text saved at: {processed_fa_path}")
```

متن انگلیسی:

```
# Import necessary libraries
import re
import os
import urllib.request
```

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

# Download necessary NLTK components
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')  # for Lemm

# Load English text
file_path_en = "/content/hp_en.txt"
with open(file_path_en, "r", encoding="utf-8") as file:
    text_en = file.read()

# English stopwords list
english_stopwords = set(stopwords.words("english"))

# Preprocessing function for English text
def preprocess_english(text):
    # 1. حذف فضاهای خالی اضافه
    text = re.sub(r'\s+', ' ', text)

    # 2. تجزیه متن به جملات
    sentences = sent_tokenize(text)

    processed_sentences = []

    for sentence in sentences:
        # 3. تبدیل حروف بزرگ به کوچک
        sentence = sentence.lower()

        # 4. تجزیه جملات به کلمات
        words = word_tokenize(sentence)

        # 5. حذف اعداد و URL ها
        words = [word for word in words if not word.isdigit() and not
word.startswith("http")]

        # 6. حذف علائم نگارشی و ایست واژه ها
        words = [re.sub(r'[^\w\s]', '', word) for word in words if word
not in english_stopwords]

        # ذخیره جملات پردازششده
```

```
        processed_sentences.append(" ".join(words))

    return " ".join(processed_sentences)

# Apply preprocessing
processed_text_en = preprocess_english(text_en)

# Save processed text to file
processed_en_path = "/content/NewProcessed_hp_en.txt"

with open(processed_en_path, "w", encoding="utf-8") as file:
    file.write(processed_text_en)

# Generate WordCloud for English text
wordcloud_en = WordCloud(width=800,
height=400).generate(processed_text_en)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud_en, interpolation="bilinear") # کیفیت تصویر را بهتر
plt.axis("off")
plt.title("English WordCloud")
plt.show()

# Provide file location
print(f"Processed English text saved at: {processed_en_path}")
```

۱ .بارگیری کتابخانه‌ها

- **Re** : برای اعمال regex (حذف فاصله‌های اضافی، حذف علائم نگارشی و اعداد و ایموجی‌ها)
- **Os** : برای بررسی وجود فونت فارسی
- **urllib.request** : برای دانلود فونت فارسی در صورت نیاز
- **hazm** : برای پردازش متن فارسی (نرمال‌سازی، توکن‌بندی، لم‌سازی)
- **Nltk** : برای پردازش متن انگلیسی (توکن‌بندی، حذف علائم نگارشی، حذف ایست‌واژه‌ها)
- **Wordcloud و matplotlib.pyplot** : برای رسم ابر کلمات(WordCloud)

۲ .دانلود داده‌های موردنیازNLTK

- **nltk.download('punkt')** : برای توکن‌بندی متن به جملات و کلمات
- **nltk.download('stopwords')** : برای حذف ایست‌واژه‌های انگلیسی

۳ .بارگیری داده‌ها

- بازکردن فایل‌های متنی **hp_fa.txt** و **hp_en.txt** .

۴ .پردازش متن فارسی

- **حذف فاصله‌های اضافی**
- **تقسیم متن به جملات** (sent_tokenize)
- **نرمال‌سازی متن** (hazm.Normalizer)
- **تقسیم جملات به کلمات** (word_tokenize)
- **حذف علائم نگارشی و ایست‌واژه‌ها**
- **حذف ایموجی‌ها**
- **اعمال فرآیند لم‌سازی** (hazm.Lemmatizer)
- **مراحل به شرح زیر است**:
  - **۱ . تعریف ایست واژه‌ها(Stopwords)**
  - persian_stopwords = set(hazm.stopwords_list())
  - در این خط، ابتدا ایست واژه‌ها را از کتابخانه **Hazm** بارگذاری می‌کنیم.
  - ایست واژه‌ها کلماتی هستند که معمولاً در تحلیل‌های زبانی اهمیتی ندارند و باید حذف شوند (مثل "و"، "در"، "با" و غیره).
  - این ایست واژه‌ها در متنی که پردازش می‌شود، نقشی در تحلیل معنای کلی ندارند، بنابراین باید از متن حذف شوند.
  - **۲ . نرمال‌سازی(Normalization)**
  - normalizer = hazm.Normalizer()

- در این خط، یک شیء از کلاس **Normalizer** ساخته می‌شود که برای نرمال‌سازی متن استفاده می‌شود.
- نرمال‌سازی به معنای تبدیل نویزهای زبانی (مثل حروف مشابه و نادرست) به یک شکل استاندارد است.
- برای مثال، در زبان فارسی حروف "ی" و "ک" باید به ترتیب به "ی" و "ک" تبدیل شوند.
- **۳ . لِماتیزه کردن(Lemmatization)**
- lemmatizer = hazm.Lemmatizer()
- در این خط، یک شیء از کلاس **Lemmatizer** ساخته می‌شود که برای لِماتیزه کردن کلمات استفاده می‌شود.
- لماتیزه کردن فرآیندی است که کلمات به شکل اصلی و ریشه‌ای خود تبدیل می‌شوند.
- مثلاً "می‌روم" به "رفت" و "کتاب‌ها" به "کتاب" تبدیل می‌شود. این فرآیند باعث کاهش پیچیدگی‌های زبان می‌شود و تحلیل کلمات را دقیق‌تر می‌کند.
- **۴ . تابع پردازش متن(Preprocessing)**
- def preprocess_farsi(text):
- این بخش یک تابع به نام preprocess_farsiتعریف می‌کند که ورودی آن متن فارسی است و خروجی آن متن پردازش‌شده می‌باشد.
- **۵ . حذف فضاهای اضافی(Whitespace Removal)**
- text = re.sub(r'\s+', ' ', text)
- در این خط، از تابع re.sub برای حذف فضاهای اضافی و تبدیل آن‌ها به یک فضای واحد استفاده می‌شود.
- در متن‌های واقعی، ممکن است فاصله‌های زیادی بین کلمات وجود داشته باشد که باید به یک فاصله تبدیل شوند تا پردازش دقیق‌تری انجام شود.
- **۶ . تجزیه متن به جملات(Sentence Tokenization)**

- sentences = hazm.sent_tokenize(text)
- این خط از کتابخانه **Hazm** برای تقسیم متن به جملات استفاده می‌کند.
- تابع sent_tokenize متن را به لیستی از جملات تقسیم می‌کند.
- این مرحله برای انجام پردازش‌های بعدی به صورت جداگانه روی هر جمله ضروری است.
- **۷. پردازش هر جمله**
- processed_sentences = []
- for sentence in sentences:
- sentence = normalizer.normalize(sentence)
- words = hazm.word_tokenize(sentence)
- words = [re.sub(r'[^\w\s]', '', word) for word in words]
- words = [word for word in words if word not in persian_stopwords]
- words = [re.sub(r'[^\w\s]', '', word) for word in words]
- words = [lemmatizer.lemmatize(word) for word in words]
- processed_sentences.append(" ".join(words))
- در این بخش، هر جمله از جملات تقسیم‌شده را به ترتیب پردازش می‌کنیم.
- **توکن‌بندی به کلمات**:
- words = hazm.word_tokenize(sentence)
- در این خط، جمله به کلمات جداگانه تقسیم می‌شود. این فرآیند به نام توکن‌بندی شناخته می‌شود.
- نتیجه این کار یک لیست از کلمات موجود در جمله است.

- **حذف علائم نگارشی**:
- `words = [re.sub(r'[^\w\s]', '', word) for word in words]`
- در این خط، با استفاده از re.sub، هر کلمه‌ای که شامل علائم نگارشی باشد (مثل نقطه، کاما، علامت سوال و ...) حذف می‌شود.
- این مرحله برای اطمینان از این است که تنها کلمات از جمله‌ها استخراج شوند.
- **حذف ایست واژه‌ها**:
- `words = [word for word in words if word not in persian_stopwords]`
- در این خط، هر کلمه‌ای که در لیست ایست واژه‌ها باشد (مثلاً "و"، "با" و غیره) حذف می‌شود.
- **حذف ایموجی‌ها**:
- `words = [re.sub(r'[^\w\s]', '', word) for word in words]`
- این خط به صورت اضافی کلمات غیر متنی (مانند ایموجی‌ها) را حذف می‌کند. این می‌تواند برای پاک‌سازی دقیق‌تر متن استفاده شود.
- **ذخیره جملات پردازش‌شده**:
- `processed_sentences.append(" ".join(words))`
- پس از پردازش هر جمله، کلمات دوباره به صورت یک جمله با فاصله‌های مناسب به هم پیوسته می‌شوند و به لیست processed_sentences اضافه می‌شوند.

- r : این حرف در ابتدا نشان می‌دهد که از raw string استفاده می‌کنیم. این به این معناست که پایتون هیچ کاراکتر خاصی مانند \n (newline) یا \t (tab) را به صورت escape sequence تفسیر نخواهد کرد. به عبارت دیگر، escape characterها در string فعال نمی‌شوند .در Regex هم به همین صورت رفتار می‌کند.
- [ ] : این بخش نشان‌دهنده یک مجموعه است. داخل براکت‌ها می‌توانید یک یا چند کاراکتر یا دامنه از کاراکترها را مشخص کنید.
- ^ : اگر ^در ابتدای یک مجموعه (در داخل [ ]) بیاید، به معنای "منفی "است. یعنی به جای جستجو برای کاراکترهای موجود در مجموعه، به دنبال کاراکترهایی می‌گردد که در مجموعه قرار ندارند.

- \w : این یک علامت واژه (word) است که به معنای هر حرف، عدد یا آندرلاین (underscore) می‌باشد. معادل کاراکترهای الفبایی، عددی و آندرلاین است. یعنی این شامل حروف انگلیسی، اعداد و _ می‌شود. در حالت پیش‌فرض، این شامل حروف کوچک و بزرگ انگلیسی (A-Z, a-z) و اعداد (۰-۹) می‌شود.
- \s : این علامت به معنای فضاهای سفید (whitespace) است که شامل فضاها، تب‌ها و خط‌های جدید (newlines) می‌شود.

۵ .پردازش متن انگلیسی

- **حذف فاصله‌های اضافی**
- **تقسیم متن به جملات** (sent_tokenize)
- **تبدیل به حروف کوچک**
- **تقسیم جملات به کلمات** (word_tokenize)
- **حذف اعداد وURL ها**
- **حذف علائم نگارشی و ایست‌واژه‌ها**
- **مراحل به شرح زیر است:**
  - **۱ . بارگذاری ایست واژه‌ها(Stopwords)**
  - english_stopwords = set(stopwords.words("english"))
  - در این خط، ایست واژه‌ها (کلمات غیرضروری که معمولاً در پردازش‌های زبان طبیعی حذف می‌شوند) برای زبان انگلیسی بارگذاری می‌شود.
  - ایست واژه‌ها شامل کلماتی مانند"the" ، "is"، "at"و غیره هستند که به تحلیل متن کمک نمی‌کنند.
  - **۲ . تابع پردازش متن انگلیسی(Preprocessing)**
  - def preprocess_english(text):

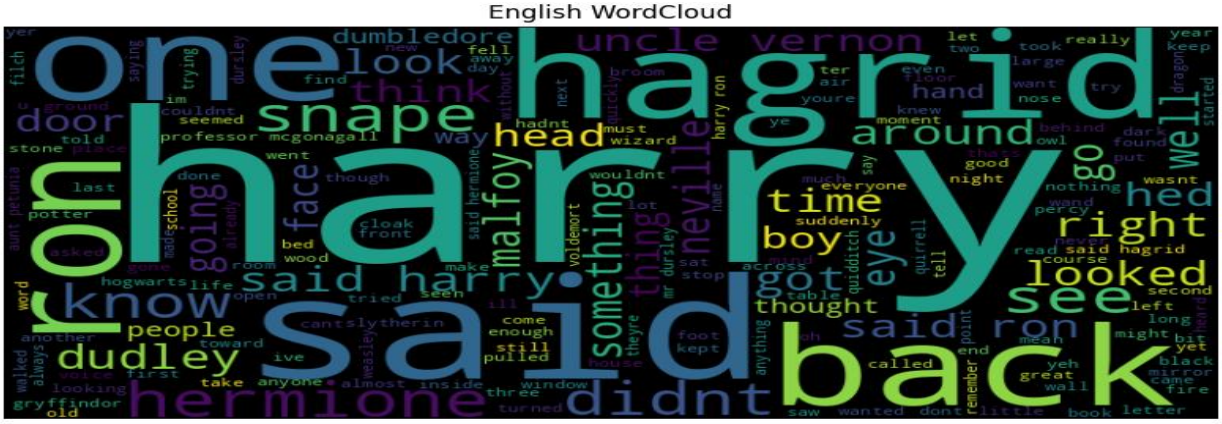
- این خط یک تابع به نام preprocess_englishتعریف می‌کند که متن انگلیسی ورودی را پردازش می‌کند و خروجی آن متن پردازش‌شده است.
- **۳ . حذف فضاهای اضافی(Whitespace Removal)**
- text = re.sub(r'\s+', ' ', text)
- در این خط از تابع re.sub برای حذف فضاهای اضافی و تبدیل آن‌ها به یک فضای واحد استفاده می‌شود.
- در متن‌های واقعی، ممکن است فاصله‌های اضافی وجود داشته باشد که باید به یک فاصله تبدیل شوند تا پردازش دقیق‌تری انجام شود.
- **۴ . تجزیه متن به جملات(Sentence Tokenization)**
- sentences = sent_tokenize(text)
- در این خط، از تابع sent_tokenizeاز کتابخانه **NLTK** استفاده می‌شود تا متن به جملات تقسیم شود.
- **۵ . پردازش هر جمله**
- processed_sentences = []
- for sentence in sentences:
- این بخش به ازای هر جمله در متن، آن را پردازش می‌کند.
- **۶ . تبدیل حروف بزرگ به کوچک(Lowercasing)**
- sentence = sentence.lower()
- در این خط، همه حروف بزرگ به حروف کوچک تبدیل می‌شوند.
- این کار باعث می‌شود که تفاوت بین "Apple" و "apple" از بین برود و پردازش متن دقیق‌تر باشد.
- **۷ . تجزیه جمله به کلمات(Word Tokenization)**

- words = word_tokenize(sentence)
- در این خط، جمله به کلمات جداگانه تقسیم می‌شود.
- این فرآیند به نام **توکن‌بندی** شناخته می‌شود و از کتابخانه **NLTK** برای انجام آن استفاده می‌شود.
- **۸. حذف اعداد وURL ها**
- words = [word for word in words if not word.isdigit() and not word.startswith("http")]
- در این خط، کلمات غیرمتنی مانند اعداد وURL ها) که با "http" شروع می‌شوند (حذف می‌شوند.
- isdigit() چک می‌کند که کلمه عددی است یا خیر.
- startswith("http")چک می‌کند که کلمه URL است یا خیر.
- **۹. حذف علائم نگارشی و ایست واژه‌ها**
- words = [re.sub(r'[^\w\s]', '', word) for word in words if word not in english_stopwords]
- در این خط، علائم نگارشی (مثل نقطه، کاما، و غیره) از کلمات حذف می‌شوند.
- همچنین، ایست واژه‌ها از جمله حذف می‌شوند. به عبارت دیگر، اگر کلمه‌ای در لیست ایست واژه‌ها باشد، حذف می‌شود.
- از تابع re.sub(r'[^\w\s]', '', word) برای حذف علائم نگارشی استفاده می‌شود. این الگو (regex)تنها کلمات و فضاها را نگه می‌دارد.
- **۱۰. ذخیره جملات پردازش‌شده**
- processed_sentences.append(" ".join(words))
- پس از پردازش کلمات هر جمله، کلمات دوباره به یک جمله با فاصله‌های مناسب پیوسته می‌شوند.

- این جمله پردازش‌شده به لیست processed_sentences اضافه می‌شود.

۶.رسم ابر کلمات برای هر زبان

English WordCloud



Persian WordCloud

بخش سوم: سیستم تبدیل متن به اعداد

```
!pip install hazm
from hazm import Normalizer
import re

words_to_numbers = {
    "صفر": 0,
    "یک": 1, "دو": 2, "سه": 3, "چهار": 4, "پنج": 5,
    "شش": 6, "هفت": 7, "هشت": 8, "نه": 9,
    "ده": 10, "یازده": 11, "دوازده": 12,
    "سیزده": 13, "چهارده": 14, "پانزده": 15,
    "شانزده": 16, "هفده": 17, "هجده": 18, "نوزده": 19,
    "بیست": 20, "سی": 30, "چهل": 40, "پنجاه": 50,
```

```
    "شصت": 60, "هفتاد": 70, "هشتاد": 80, "نود": 90,
    "صد": 100, "دویست": 200, "سیصد": 300, "چهارصد": 400,
    "پانصد": 500, "ششصد": 600, "هفتصد": 700,
    "هشتصد": 800, "نهصد": 900
}

magnitudes = {
    "میلیارد": 1000000000,
    "میلیون": 1000000,
    "هزار": 1000
}

def convert_to_persian_number(number):
    english_to_persian_digits = str.maketrans("0123456789", "۰۱۲۳۴۵۶۷۸۹")
    return str(number).translate(english_to_persian_digits)

def parse_simple_number(words):
    total = 0
    for word in words:
        if word in words_to_numbers:
            total += words_to_numbers[word]
    return total

def text_to_number_fa(text):
    normalizer = Normalizer()
    text = normalizer.normalize(text)
    text = text.replace(" و ", " ")
    words = text.strip().split()

    total = 0
    current_chunk = []
    current_magnitude = None

    for word in words:
        if word in magnitudes:
            num = parse_simple_number(current_chunk)
            total += num * magnitudes[word]
            current_chunk = []
        else:
            current_chunk.append(word)

    if current_chunk:
        total += parse_simple_number(current_chunk)

    return total
```

```
# --- اجرا ---
user_input = input("عدد را به صورت حروف فارسی وارد کنید: ")
number_result = text_to_number_fa(user_input)
persian_number = convert_to_persian_number(number_result)

print("ورودی :", user_input)
print("خروجی عددی (فارسی):", persian_number)
```

یک ابزار تبدیل دوطرفه عدد و متن است که به‌صورت هوشمند، زبان فارسی یا انگلیسی را تشخیص می‌دهد و متن عددی را به عدد و بالعکس تبدیل می‌کند. این سیستم از اعداد صحیح، اعشاری و منفی پشتیبانی می‌کند و بدون نیاز به رابط گرافیکی یا صوتی، به‌صورت خط فرمان (CLI) اجرا می‌شود.(در این اینجا فقط کد بخش مورد سوال شده در تمرین را قرار دادیم و نسخه‌ی کامل آن که حتی به صورت ویس هم اعداد انگلیسی را خروجی می‌دهد، در داخل گیت هابم قرار داده‌ام و یک ابزار تمام عیار تبدیلات ساختم.)

شرح فقط نسخه‌ی فارسی :

مراحل الگوریتم:

1 . نرمال‌سازی متن:

- در ابتدا از Normalizer استفاده می‌کنیم تا متن واردشده را **نرمال کنیم** .این کار شامل حذف نیم‌فاصله‌ها و تغییرات غیرضروری است.

normalizer = Normalizer()

text = normalizer.normalize(text)

- این به این دلیل است که در فارسی ممکن است کاربر از نیم‌فاصله‌ها (که در برخی متون تایپی و فونت‌ها وجود دارند) استفاده کند و آن‌ها را باید حذف کنیم تا مشکلی در پردازش متن ایجاد نشود.

2 . تقسیم متن به کلمات:

text = text.replace("\u200c", " ")

words = text.strip().split()

- پس از نرمال‌سازی، متن ورودی به کلمات مختلف تقسیم می‌شود .**کلمات عددی** (مثل "یک"، "پنجاه") و **مراتب عددی** (مثل "هزار"، "میلیون") باید شناسایی شوند.

3 . پردازش کلمات:

الگوریتم با استفاده از حلقه برای پردازش کلمات و تبدیل آن‌ها به عدد به این صورت عمل می‌کند:

```
total = 0
current_chunk = []
for word in words:
    # if word in magnitudes:  اگر کلمه در دیکشنری مراتب باشد
        num = parse_simple_number(current_chunk)
        # total += num * magnitudes[word]  ضرب در مرتبه
        # current_chunk = []  صفر کردن chunk برای کلمه‌های بعدی
    else:
        # current_chunk.append(word)  ذخیره کلمه‌ها تا رسیدن به مرتبه
```

شرح منطق:

۱. **کلمات عددی** (مثل "یک"، "پانزده"، "سیصد") در متغیر current_chunk جمع می‌شوند. این متغیر یک مجموعه از کلمات است که به یک عدد تبدیل خواهد شد.

۲. **مراتب عددی** (مثل "هزار"، "میلیون") شناسایی می‌شوند. هنگامی که یکی از این کلمات شناسایی شد، مقدار عددی current_chunk محاسبه می‌شود و سپس با **مرتبه** (مثل هزار یا میلیون) ضرب می‌شود.

۳. پس از ضرب عدد با مرتبه، نتیجه به **مجموع نهایی total** اضافه می‌شود.

٤. در نهایت، وقتی هیچ مرتبه‌ای باقی نماند و کلمات به پایان رسید، باقی‌مانده‌ی current_chunk به عدد تبدیل می‌شود و به total اضافه می‌شود.

4 . عملیات در parse_simple_number:

```
def parse_simple_number(words):
    total = 0
```

```
for word in words:
if word in words_to_numbers:
total += words_to_numbers[word]
return total
```

- این بخش کلمات عددی را به **عدد** تبدیل می‌کند. به‌طور مثال، "پانزده" به ۱۵، "سیصد" به ۳۰۰ و... . این تبدیل با استفاده از دیکشنری words_to_numbers انجام می‌شود که کلمات عددی را به مقادیر عددی تبدیل می‌کند.

5 . اضافه کردن کلمات با مرتبه:

در صورتی که کلمه‌ای از **مراتب عددی** (مثل "هزار" یا "میلیون") پیدا شود، تمام اعداد جمع‌شده در current_chunk محاسبه شده و با مقدار مربوط به مرتبه (برای مثال ۱۰۰۰ برای "هزار") ضرب می‌شود و در نهایت به totalاضافه می‌شود.

6 . نتیجه:

کد کل **عدد** را در total جمع می‌کند و اگر عدد منفی باشد (که در کد فعلی مورد استفاده نیست)، از علامت منفی به مقدار اضافه می‌کند.

```
return total
```

۱. **کلمات عددی** به **مقدار عددی** تبدیل می‌شوند.

۲. در صورتی که کلمه‌ای مربوط به **مرتبه‌های عددی** باشد (مثل هزار، میلیون، میلیارد)، آن مقدار عددی به مرتبه ضرب می‌شود.

۳. **عدد نهایی** پس از پردازش تمامی کلمات و مرتبه‌ها محاسبه می‌شود.

شرح نسخه‌ی کامل :

```
عدد را به صورت حروف فارسی وارد کنید: سه میلیون و سیصد و پنجاه و سه
ورودی : سه میلیون و سیصد و پنجاه و سه
خروجی عددی (فارسی): ۳۰۰۰۳۵۳

:عدد یا متن عددی وارد کنید (فارسی یا انگلیسی)
منفی چهارصد و هفتاد و پنج هزار و ده <
خروجی: -475,010.0
```

```
عدد را به صورت حروف فارسی وارد کنید (با یا بدون فاصله): سهمیلیونوسیصدوپنجاهوسه
ورودی: سهمیلیونوسیصدوپنجاهوسه
خروجی عددی (فارسی): ۳۰۰۰۳۵۳

لطفاً عدد مورد نظر خود را وارد کنید: 3000353
عدد به حروف فارسی: سه میلیون و سیصد و پنجاه و سه
```

1 . نصب کتابخانه‌ها:

```
!pip install hazm
```

- این خط برای نصب **کتابخانه‌ی hazm** است که مخصوص پردازش زبان طبیعی فارسی است و برای تحلیل و نرمال‌سازی متن فارسی استفاده می‌شود.

```
!pip install num2fawords
```

- این خط برای نصب **کتابخانه‌ی num2fawords** است که از آن برای تبدیل عدد به متن فارسی استفاده می‌کنیم.

2 . دیکشنری اعداد فارسی:

```
words_to_numbers = {
    "صفر": 0, "یک": 1, "دو": 2, "سه": 3, "چهار": 4, "پنج": 5,
    "شش": 6, "هفت": 7, "هشت": 8, "نه": 9, "ده": 10, "یازده": 11,
    "دوازده": 12, "سیزده": 13, "چهارده": 14, "پانزده": 15,
    "شانزده": 16, "هفده": 17, "هجده": 18, "نوزده": 19,
    "بیست": 20, "سی": 30, "چهل": 40, "پنجاه": 50,
    "شصت": 60, "هفتاد": 70, "هشتاد": 80, "نود": 90,
    "صد": 100, "دویست": 200, "سیصد": 300, "چهارصد": 400,
    "پانصد": 500, "ششصد": 600, "هفتصد": 700, "هشتصد": 800, "نهصد": 900
}
```

- این بخش یک **دیکشنری** تعریف می‌کند که کلمات فارسی مربوط به اعداد را به مقادیر عددی متناظر آن‌ها نگاشت می‌کند. به‌طور مثال، "یک" به ۱ و "پانصد" به ۵۰۰ تبدیل می‌شود.

3 . دیکشنری مراتب:

```
magnitudes = {
```

" میلیارد,9**10 :"

" میلیون,6**10 :"

" هزار3**10 :"

}

- این بخش دیکشنری‌ای برای **مراتب اعداد** مانند "هزار", "میلیون", و "میلیارد" ایجاد می‌کند که مقادیر آن‌ها به صورت عددی (بر حسب توان‌های ۱۰) در نظر گرفته می‌شود.

4 . تبدیل متن فارسی به عدد:

```
def fa_text_to_number(text):
normalizer = Normalizer()
text = normalizer.normalize(text).replace("", "").replace(" ", "")
pattern = "(" + "|".join(sorted(fa_all_keywords, key=len, reverse=True)) + ")"
tokens = re.findall(pattern, text)
total = 0
chunk = []
negative = False
decimal_part = 0.0
is_decimal = False
decimal_chunk = []
```

- این بخش از کد به **نرمال‌سازی متن** می‌پردازد (حذف فاصله‌ها و نیم‌فاصله‌ها) و سپس با استفاده از یک **الگوی regular expression (regex)** کلمات عددی فارسی را استخراج می‌کند.
- همچنین به متغیرهایی برای شناسایی عدد منفی و اعشاری اشاره می‌شود.

```
for word in tokens:
    if word == "  منفی:"
```

negative = True

elif word == " ممیز:"

is_decimal = True

elif word in fa_magnitudes:

chunk_val = sum(fa_words_to_numbers.get(w, 0) for w in chunk)

total += chunk_val * fa_magnitudes[word]

chunk = []

elif is_decimal:

decimal_chunk.append(str(fa_words_to_numbers.get(word, 0)))

else:

chunk.append(word)

- این قسمت به بررسی کلمات استخراج‌شده پرداخته و آن‌ها را با توجه به نقش‌شان (عدد منفی، عدد اعشاری یا مقدار مربوط به مرتبه‌های عددی) پردازش می‌کند.
- **اعداد** جمع می‌شوند و سپس در **مراتب** ضرب می‌شوند.

total += sum(fa_words_to_numbers.get(w, 0) for w in chunk)

if decimal_chunk:

decimal_part = float("0." + "".join(decimal_chunk))

final = total + decimal_part

return -final if negative else final

- در این قسمت، اگر عدد اعشاری وجود داشته باشد، بخش اعشاری نیز محاسبه می‌شود و در نهایت نتیجه عددی به دست می‌آید.

5 . تبدیل عدد به متن فارسی:

def fa_number_to_text(n):

try:

```
n = float(n)
if n < 0:
    return "منفی " + fa_words(abs(int(n))) # بدون اعشار برای متن
else:
    return fa_words(int(n))
except:
    return "خطا در تبدیل عدد فارسی"
```

- این تابع برای تبدیل یک **عدد صحیح یا اعشاری** به متن فارسی است. در صورتی که عدد منفی باشد، ابتدا کلمه "منفی" به متن اضافه می‌شود.

6 . تبدیل متن انگلیسی به عدد:

```
def en_text_to_number(text):
    try:
        text = text.lower().replace("minus", "negative")
        text = text.replace("point", ".")
        if "negative" in text:
            text = text.replace("negative", "").strip()
            return -float(w2n.word_to_num(text))
        return float(w2n.word_to_num(text))
    except:
        return "Invalid English input"
```

- این قسمت برای تبدیل **متن عددی انگلیسی** به عدد استفاده می‌شود.
- کلمات مانند "minus" به "negative" تبدیل می‌شوند تا قابلیت پردازش داشته باشند. همچنین برای تبدیل اعداد اعشاری، کلمه "point" به "." تبدیل می‌شود.

7 . تبدیل عدد به متن انگلیسی:

```
def en_number_to_text(n):
try:
n = float(n)
if n < 0:
return "minus " + num2words(abs(n), lang='en')
return num2words(n, lang='en')
except:
return " Invalid number"
```

- این تابع برای تبدیل **عدد به متن انگلیسی** استفاده می‌شود.
- مشابه با تبدیل فارسی، اگر عدد منفی باشد، کلمه "minus" به متن اضافه می‌شود.

8 . تشخیص زبان و نوع ورودی:

```
def detect_lang(text):
farsi_chars = " آابپتثجچحخدذرزژسشصضطظعغفقکگلمنوهی"
return any(ch in text for ch in farsi_chars)
```

- این تابع با بررسی حضور **حروف فارسی** در ورودی، زبان ورودی را شناسایی می‌کند.

9 . تبدیل هوشمند:

```
def smart_converter(text):
text = text.strip()
is_farsi = detect_lang(text)
...
```

- این بخش، نوع ورودی را شناسایی کرده و سپس متن را به عدد یا عدد را به متن تبدیل می‌کند.

10 . اجرای برنامه:

user_input = input("عدد یا متن عددی وارد کنید (فارسی یا انگلیسی)(" <n\>:

result = smart_converter(user_input)

print("خروجی", result):

- در اینجا، ورودی از کاربر گرفته می‌شود و سپس با استفاده از **تابع smart_converter** تبدیل انجام می‌شود.

بخش چهارم: تصحیح خطاهای املایی

این کد برای تصحیح غلط‌های املایی در یک جمله است که شامل الگوریتم های متنوع برای یافتن بهترین پاسخ می‌باشد و درنهایت به علت نتایج دقیق‌تر با استفاده از دو الگوریتم مختلف انجام می‌شود:

الگوریتم SymSpell

الگوریتم **SymSpell** یکی از سریع‌ترین و دقیق‌ترین روش‌های تصحیح غلط‌های املایی است. این الگوریتم با استفاده از یک دیکشنری بزرگ لغات(سفارشی)، مشابه‌ترین کلمه به کلمه غلط املایی را پیدا کرده و آن را اصلاح می‌کند. به این ترتیب که:

- ابتدا جمله به کلمات جداگانه تقسیم می‌شود.
- سپس برای هر کلمه، کلمه‌های نزدیک به آن از دیکشنری انتخاب می‌شود (بر اساس فاصله ویرایشی).
- نزدیک‌ترین کلمه به کلمه غلط انتخاب شده و به جای آن قرار می‌گیرد.
- **بارگذاری دیکشنری** :دیکشنری لغات باید از یک فایل متنی بارگذاری شود که در اینجا نام آن **Vocabulary.txt** است.
- **فاصله ویرایشی** :الگوریتم از یک معیار به نام **فاصله ویرایشی** استفاده می‌کند که تعیین می‌کند چقدر دو کلمه از یکدیگر متفاوت هستند.
- **سرعت بالا** SymSpell :به دلیل استفاده از ساختار داده‌ای خاص و پیش‌پردازش کلمات، سرعت بسیار بالایی دارد.

کد SymSpell

در کد نمونه، ابتدا یک دیکشنری بارگذاری می‌شود، سپس هر کلمه در جمله بررسی شده و نزدیک‌ترین کلمه از دیکشنری به آن پیدا می‌شود.

```
SymSpell 40% الگوریتم -

!pip install symspellpy

Collecting symspellpy
  Downloading symspellpy-6.9.0-py3-none-any.whl.metadata (3.9 kB)
Collecting editdistpy>=0.1.3 (from symspellpy)
  Downloading editdistpy-0.1.5-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014
_x86_64.whl.metadata (7.9 kB)
Downloading symspellpy-6.9.0-py3-none-any.whl (2.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
2.6/2.6 MB 33.7 MB/s eta 0:00:00
Downloading editdistpy-0.1.5-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014
_x86_64.whl (144 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
144.1/144.1 kB 8.9 MB/s eta 0:00:00
Installing collected packages: editdistpy, symspellpy
Successfully installed editdistpy-0.1.5 symspellpy-6.9.0

from symspellpy.symspellpy import SymSpell, Verbosity
import re

# Initialize SymSpell
max_edit_distance_dictionary = 2
prefix_length = 7
sym_spell = SymSpell(max_edit_distance_dictionary, prefix_length)

# Load dictionary
sym_spell.load_dictionary("/content/Vocabulary.txt", term_index=0,
count_index=1)

# تصحیح جمله
def symspell_correct_sentence(sentence):
    tokens = re.findall(r"\b\w+\b|[^\w\s]", sentence)
    corrected = []
    for token in tokens:
        if token.isalpha():
```

```
            suggestions = sym_spell.lookup(token.lower(),
Verbosity.CLOSEST, max_edit_distance=2)
            corrected_word = suggestions[0].term if suggestions else token
            corrected.append(corrected_word)
        else:
            corrected.append(token)
    return ' '.join(corrected).replace(" .", ".")

# تست
test_sentence = "Th quik braown foxs jmups ovem te lagzy qog."
print(symspell_correct_sentence(test_sentence))


the quirk brown foxes jumps ovum te lazy log.
```

مثال خروجی:

برای جمله"Th quik braown foxs jmups ovem te lagzy qog." ، خروجی اصلاح‌شده به شکل زیر خواهد بود:

the quirk brown foxes jumps ovum te lazy log.

الگوریتم Levenshtein/SequenceMatcher

به آن **Distance Edit** نیز گفته می‌شود. فاصله ویرایشی بین دو رشته را محاسبه می‌کند و آن را بر اساس تعداد تغییراتی که برای تبدیل یک رشته به دیگری لازم است اندازه‌گیری می‌کند. این تغییرات شامل:

- **حذف** حروف
- **اضافه کردن** حروف
- **تعویض** حروف

در این الگوریتم:

۱. ابتدا جمله دارای غلط املایی به کلمات جداگانه تقسیم می‌شود.

۲. سپس برای هر کلمه، نزدیک‌ترین کلمه موجود در دیکشنری با استفاده از تابع **get_close_matches** پیدا می‌شود.

۳. کلمه اصلاح‌شده به جمله اضافه می‌شود.

کد Levenshtein

```
الگوریتم مقایسه شباهت(Levenshtein/SequenceMatcher) 20%

import re
from difflib import get_close_matches

# مرحله ۱: لود کردن دیکشنری لغات از فایل
with open("/content/Vocabulary.txt", "r") as f:
    vocabulary = set(word.strip().lower() for word in f if word.strip())

# مرحله ۲: جمله‌ی دارای غلط املایی
# The quick brown fox jumps over the lazy dog.
misspelled_sentence = "Th quik braown foxs jmups ovem te lagzy qog."

# مرحله ۳: توکنایز جمله به کلمات
words = re.findall(r"\b\w+\b", misspelled_sentence.lower())

# تابع اصلاح غلط‌های املایی
def correct_word(word, vocab):
    matches = get_close_matches(word, vocab, n=1, cutoff=0.8)
    return matches[0] if matches else word

# اصلاح تمام کلمات جمله
corrected_words = [correct_word(word, vocabulary) for word in words]

# بازسازی جمله اصلاح‌شده
corrected_sentence = " ".join(corrected_words).capitalize() + "."

# نمایش نتیجه‌ها
print("\nOriginal sentence:")
print("The quick brown fox jumps over the lazy dog.")
print("\nMisspelled sentence:")
print(misspelled_sentence)
print("\nCorrected sentence:")
print(corrected_sentence)


Original sentence:
The quick brown fox jumps over the lazy dog.

Misspelled sentence:
Th quik braown foxs jmups ovem te lagzy qog.
```

```
Corrected sentence:
The quirk brown foxes mumps ovem te lazy qog.
```

مثال خروجی:

برای جمله "Th quik braown foxs jmups ovem te lagzy qog."، خروجی اصلاح‌شده به شکل زیر خواهد بود:

The quirk brown foxes mumps ovem te lazy qog.

مقایسه الگوریتم‌ها

۱. **SymSpell**:

- سریع‌تر است زیرا از پیش‌پردازش و ذخیره‌سازی نتایج استفاده می‌کند.
- دقت بالاتری دارد، به ویژه زمانی که کلمات اصلاح‌شده باید از دیکشنری لغات بزرگ انتخاب شوند.
- مناسب برای جملات بلند و پیچیده که نیاز به سرعت و دقت بالا دارند.

۲. **Levenshtein/SequenceMatcher**:

- دقت کمتری نسبت به SymSpell دارد، زیرا به‌طور کلی نیاز به محاسبات پیچیده‌تری برای یافتن نزدیک‌ترین کلمه دارد.
- مناسب برای اصلاح غلط‌های املایی ساده‌تر است که نیاز به بررسی‌های پیچیده ندارند.

نتیجه‌گیری

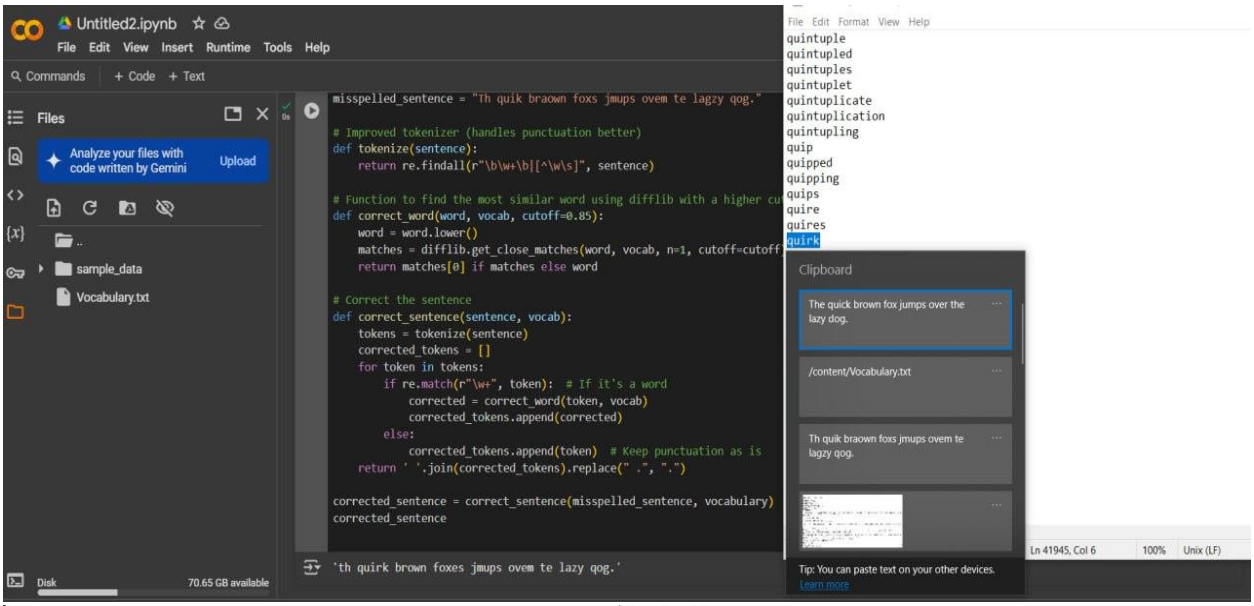
الگوریتم **SymSpell** به دلیل سرعت و دقت بالای آن در تصحیح غلط‌های املایی و استفاده از دیکشنری پیش‌پردازش‌شده، گزینه بهتری برای کاربردهای بزرگتر و پیچیده‌تر است. در حالی که **Levenshtein/SequenceMatcher**ممکن است برای کارهای ساده‌تر و جملات کوتاه‌تر مناسب‌تر باشد.

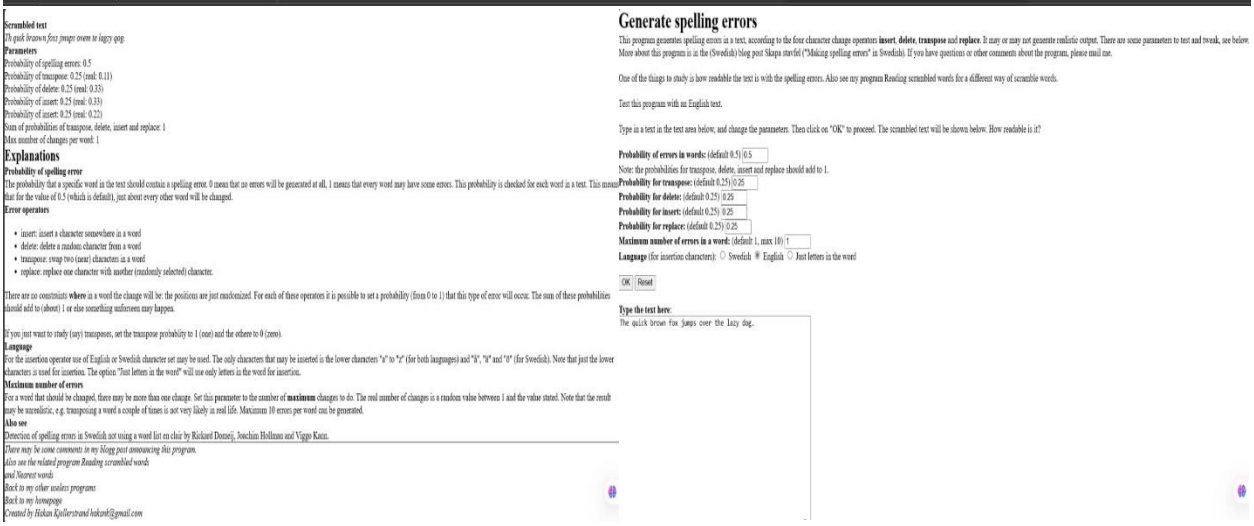
سایت‌هایی که به‌طور خودکار متن‌ها را خراب می‌کنند، معمولاً از **تکنولوژی‌های تصحیح خودکار آنلاین** استفاده می‌کنند که بیشتر به منظور تسریع و اصلاح فوری متن‌ها طراحی شده‌اند. این سایت‌ها معمولاً از روش‌هایی مانند **Autocorrect** یا **Spelling Checkers** استفاده می‌کنند. برخی از مشکلات رایج که باعث خراب شدن متن توسط این سایت‌ها می‌شوند، عبارتند از:

۱. **نحوه شناسایی غلط‌های املایی** :این سایت‌ها ممکن است تنها به کلمات ناآشنا یا اشتباهات واضح در متن توجه کنند و تصحیح‌های خود را به صورت خودکار اعمال کنند. این تصحیح‌ها گاهی اوقات باعث تغییر معنای کلمات یا جملات می‌شوند.

۲. **عدم درک زمینه** :این سایت‌ها معمولاً نمی‌توانند زمینه‌ی معنایی جمله را به درستی تحلیل کنند و ممکن است کلمات صحیح را به اشتباه تغییر دهند یا آن‌ها را به کلمات مشابهی تبدیل کنند که کاملاً بی‌معنی یا اشتباه در زمینه‌ی جمله هستند.

۳. **تصحیح‌های نادرست و غیرضروری** :گاهی اوقات این ابزارها اصلاحات غیرضروری را اعمال می‌کنند که باعث به هم ریختن جمله و تغییر ساختار آن می‌شود، به‌ویژه زمانی که به اصلاحات در دیکته، نشانه‌گذاری یا ترکیب‌های خاص توجه نمی‌کنند.





بخش پنجم: تشخیص اخبار جعلی

TF-IDF-1

TF-IDF (Term Frequency-Inverse Document Frequency) : این روش یکی از روش‌های رایج برای **بردارسازی متن** است که با محاسبه اهمیت یک کلمه در متن نسبت به کل مجموعه داده‌ها، یک نمایه عددی از متن‌ها می‌سازد.

ویژگی‌ها:

- **بردارسازی**: این روش به هر کلمه یک وزن اختصاص می‌دهد که نشان می‌دهد آن کلمه چقدر مهم است. هر کلمه در هر مستند (متن) می‌تواند به‌عنوان یک ویژگی جداگانه در نظر گرفته شود.
- **عدم نیاز به Padding** : در **TF-IDF**، داده‌های ورودی به‌طور مستقیم به صورت یک ماتریس ویژگی‌های **اسپارس** (sparse) تبدیل می‌شوند. هر ویژگی نمایانگر تعداد دفعات حضور یک کلمه در یک سند (با وزن‌دهی) است. در اینجا هیچ نیازی به یکسان کردن طول دنباله‌ها نیست چون هر کلمه یک ویژگی منحصر به فرد است.
- **ویژگی‌ها**: هر کلمه‌ای که در **TF-IDF**وارد می‌شود یک ویژگی جداگانه است و هیچکدام از ویژگی‌ها به ترتیب یا طول وابسته نیستند. به همین دلیل **پدینگ** در اینجا استفاده نمی‌شود.

مراحل کار با TF-IDF:

۱. ایجاد **مجموعه کلمات** از مجموعه داده‌ها.

۲. محاسبه **فرکانس کلمات** (TF) در هر سند.

۳. محاسبه **معکوس فرکانس سند** (IDF) برای کلمات.

۴. ضرب TF و IDF برای هر کلمه.

```
# 1. خواندن داده ها
df_true = pd.read_csv(true_path)
df_fake = pd.read_csv(fake_path)

# 2. برچسبگذاری
df_true["label"] = 1
df_fake["label"] = 0
```

```
df = pd.concat([df_true, df_fake], axis=0)
df = df.sample(frac=1).reset_index(drop=True)  # مخلوط کردن داده ها

# 3. انتخاب فقط ستون 'text' (در صورت وجود)
# text_column = "text" if "text" in df.columns else df.columns[0]

df['text'] = df['title'] + " " + df['text'] + " " + df['subject'] + " " +
df['date'].astype(str)  # ترکیب عنوان، متن، موضوع و تاریخ

#
================================================================================
==============================

# # اعمال پیشپردازش روی داده ها
# df["clean_text"] = df[text_column].astype(str).apply(clean_text)

def clean_text(text):
    # 1. تبدیل حروف بزرگ به حروف کوچک
    text = text.lower()

    # 2. حذف فضاهای خالی اضافی
    text = ' '.join(text.split())

    # 3. تجزیه متن به جملات
    sentences = text.split(".")  # سادهترین روش تقسیم جملات با نقطه

    words = []
    stop_words = set(nltk.corpus.stopwords.words('english'))

    for sentence in sentences:
        # 4. توکنبندی جملات به کلمات (با استفاده از str.split)
        tokens = sentence.split()

        # 5. حذف اعداد و URLها
        tokens = [word for word in tokens if not word.isdigit() and not
word.startswith('http')]

        # 6. حذف علائم نگارشی و ایستواژه ها
        tokens = [word for word in tokens if word.isalpha() and word not
in stop_words]

        words.extend(tokens)

    return " ".join(words)
```

```
# اعمال پیش‌پردازش روی داده‌ها
# df["clean_text"] = df[text_column].astype(str).apply(clean_text)

df["clean_text"] = df['text'].astype(str).apply(clean_text)

#
================================================================================
==============================

# 5. تقسیم داده‌ها
X_train_val, X_test, y_train_val, y_test =
train_test_split(df["clean_text"], df["label"], test_size=0.15,
random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val,
y_train_val, test_size=0.176, random_state=42)  # 0.176×0.85 ≈ 0.15

# 6. بردارسازی متن
vectorizer = TfidfVectorizer(max_features=5000, min_df=5, max_df=0.95)
# vectorizer = TfidfVectorizer(max_features=5000)
X_train_vec = vectorizer.fit_transform(X_train)
X_val_vec = vectorizer.transform(X_val)
X_test_vec = vectorizer.transform(X_test)

# 7. آموزش مدل KNN
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_vec, y_train)

# 8. ارزیابی مدل KNN
y_val_pred_knn = knn.predict(X_val_vec)
print("KNN on Validation:")
print("Accuracy:", accuracy_score(y_val, y_val_pred_knn))
print("Precision:", precision_score(y_val, y_val_pred_knn))
print("Recall:", recall_score(y_val, y_val_pred_knn))
print("F1 Score:", f1_score(y_val, y_val_pred_knn))

# 9. آموزش مدل SVM
svm = SVC(kernel='rbf', C=1, gamma='scale')
# svm = SVC()
svm.fit(X_train_vec, y_train)

# 10. ارزیابی مدل SVM
y_val_pred_svm = svm.predict(X_val_vec)
print("\nSVM on Validation:")
print("Accuracy:", accuracy_score(y_val, y_val_pred_svm))
print("Precision:", precision_score(y_val, y_val_pred_svm))
```

```
print("Recall:", recall_score(y_val, y_val_pred_svm))
print("F1 Score:", f1_score(y_val, y_val_pred_svm))

# 11. ارزیابی نهایی روی Test
y_test_pred_knn = knn.predict(X_test_vec)
y_test_pred_svm = svm.predict(X_test_vec)

print("\nFinal Comparison on Test Data:")
print("Model\tAccuracy\tPrecision\tRecall\t\tF1 Score")
print(f"KNN\t{accuracy_score(y_test,
y_test_pred_knn):.4f}\t\t{precision_score(y_test,
y_test_pred_knn):.4f}\t\t{recall_score(y_test,
y_test_pred_knn):.4f}\t\t{f1_score(y_test, y_test_pred_knn):.4f}")
print(f"SVM\t{accuracy_score(y_test,
y_test_pred_svm):.4f}\t\t{precision_score(y_test,
y_test_pred_svm):.4f}\t\t{recall_score(y_test,
y_test_pred_svm):.4f}\t\t{f1_score(y_test, y_test_pred_svm):.4f}")
```

```
KNN on Validation:
Accuracy: 0.8841744826559476
Precision: 0.9376114081996435
Recall: 0.8134859263841633
F1 Score: 0.8711493872143093

SVM on Validation:
Accuracy: 0.9955337204108977
Precision: 0.995666976168369
Recall: 0.9950510361892979
F1 Score: 0.9953589108910891

Final Comparison on Test Data:
Model   Accuracy        Precision       Recall          F1 Score
KNN     0.8937          0.9562          0.8219          0.8840
SVM     0.9952          0.9949          0.9955          0.9952
```

1 . خواندن داده‌ها:

df_true = pd.read_csv(true_path)

df_fake = pd.read_csv(fake_path)

- در اینجا فایل‌های CSV که اخبار واقعی و جعلی هستند از مسیرهای داده‌شده در متغیرهای true_path و fake_path بارگذاری می‌شوند.

- داده‌های این فایل‌ها درDataFrame های df_true و df_fake ذخیره می‌شوند.

2 . برچسب‌گذاری(Labeling) :

df_true["label"] = 1

df_fake["label"] = 0

- **ربط متن در فایل‌های CSV به لیبل‌ها**

  - در واقع، متن در فایل‌های CSV (که شامل اخبار واقعی یا جعلی هستند) ربط مستقیم به **صحّت یا جعل** اخبار ندارد، بلکه متن همانند "ویژگی" یا "فیچر" برای شناسایی اخبار جعلی است. ما از متن برای شناسایی اخبار جعلی استفاده می‌کنیم. اما اگر بخواهیم کار با متن را دقیق‌تر و مؤثرتر انجام دهیم، می‌توانیم از ترکیب اطلاعات اضافی مثل تاریخ، عنوان یا موضوع نیز استفاده کنیم. همگی می‌توانند به ما کمک کنند تا بفهمیم کدام اخبار واقعی و کدام جعلی هستند. برای مثال، اخبار جعلی معمولاً در دسته‌بندی‌های خاصی قرار دارند و ممکن است تاریخ‌های خاصی داشته باشند. این اطلاعات می‌توانند در کنار متن برای افزایش دقت مدل مفید باشند.

- به هر خبر واقعی برچسب 1 داده می‌شود و به هر خبر جعلی برچسب 0 داده می‌شود. این برچسب‌ها برای استفاده در مدل‌های یادگیری ماشین بعداً به عنوان هدف (Target) استفاده می‌شوند.

3 . ترکیب داده‌ها:(Concatenation)

df = pd.concat([df_true, df_fake], axis=0)

df = df.sample(frac=1).reset_index(drop=True) # مخلوط کردن داده‌ها

- داده‌های واقعی و جعلی به هم متصل می‌شوند در یک DataFrame به نام df
- سپس داده‌ها با استفاده از sample(frac=1) به صورت تصادفی مخلوط می‌شوند تا ترتیب داده‌ها تصادفی باشد و مدل بر اساس ترتیب خاصی آموزش نبیند.
- reset_index(drop=True) هم باعث می‌شود که ایندکس‌های جدید برای داده‌ها در نظر گرفته شوند.

4 . انتخاب ستون متنی:

text_column = "text" if "text" in df.columns else df.columns[0]

- در اینجا بررسی می‌شود که آیا ستون text در DataFrame موجود است یا خیر.
- اگر ستون textوجود داشته باشد، همان ستون به عنوان متن برای مدل انتخاب می‌شود. در غیر این صورت، اولین ستون (df.columns[0]) به عنوان متن در نظر گرفته می‌شود. که در ادامه و کد نهایی که داخل این گزارش اوردیم، تنها به ویژگی متن کفایت نکرده و ترکیبی از دیگر ویژگی‌ها را نیز بر آن افزودیم و دقت هر دو مدل knn و svm به ترتیب از ۶۵درصد به ۸۸ درصد و از ۹۳ درصد به ۹۹ درصد افزایش یافت. کد آن به فرم زیر است:

```
df['text'] = df['title'] + " " + df['text'] + " " + df['subject'] + " " + df['date'].astype(str)  # ترکیب عنوان، متن، موضوع و تاریخ
```

5 . پیش‌پردازش متن:(Text Preprocessing)

- این بخش مطابق بخش دوم تمرین، متن‌ها را پیش‌پردازش می‌کند تا برای مدل یادگیری ماشین آماده شوند:

۱. **حروف کوچک می‌شوند** برای جلوگیری از حساسیت به حروف بزرگ و کوچک.

۲. **URLها** حذف می‌شوند.

۳. **اعداد** حذف می‌شوند.

۴. **علامات نگارشی** (مثل نقطه، کاما و غیره) حذف می‌شوند.

۵. **توکن‌بندی** انجام می‌شود: با استفاده از split()که متن را به کلمات مجزا تبدیل می‌کند.

۶. **حذف stopwords** : کلمات رایجی مانند"the" ، "is" که مفهومی ندارند، حذف می‌شوند.

- پس از این پیش‌پردازش‌ها، نتایج در ستون جدید clean_text ذخیره می‌شود.

6 . تقسیم داده‌ها به Train, Validation, Test :

```
X_train_val, X_test, y_train_val, y_test = train_test_split(df["clean_text"], df["label"], test_size=0.15, random_state=42)

X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.176, random_state=42)  # 0.176×0.85 ≈ 0.15
```

- **Train**: 70%

- **Validation**: 15%
- **Test**: 15%
- ابتدا داده‌ها به دو قسمت تقسیم می‌شوند:

۱. **Train و Test** : 85% داده‌ها برای آموزش و ۱۵٪ برای تست.

- سپس، داده‌های **Train** به دو قسمت تقسیم می‌شوند: **Train و Validation**
- 85% از داده‌های آموزش به عنوان داده‌های Trainو ۱۵٪ باقی‌مانده به عنوان داده‌های Validation
- بنابراین، برای اینکه تست ۱۵ درصد از کل داده‌ها را شامل شود، ابتدا داده‌ها به دو بخش تقسیم می‌شوند با نسبت ۸۵٪ به ۱۵٪، سپس بخش ۸۵٪ دوباره به دو بخش تقسیم می‌شود. این کار باعث می‌شود که در مجموع، داده‌های Train، Validation و Test به ترتیب ۷۰٪، ۱۵٪ و ۱۵٪ باشند.
- در واقع، عدد ۰.۱۷۶ برای test_size به این دلیل است که:

۱. 15% از کل داده‌ها برای Test در نظر گرفته شده‌اند.

۲. داده‌های باقی‌مانده (۸۵٪) برای Train و Validation هستند.

۳. سپس ۱۵٪ از این ۸۵٪ برای Validation در نظر گرفته می‌شود که به این ترتیب ۰.۱۷۶ از ۰.۸۵ می‌شود که حدود ۱۵٪ است.

7 . بردارسازی متن:(Text Vectorization)

```
vectorizer = TfidfVectorizer(max_features=5000)
X_train_vec = vectorizer.fit_transform(X_train)
X_val_vec = vectorizer.transform(X_val)
X_test_vec = vectorizer.transform(X_test)
```

- **چرا باید متن رو به دنباله عددی تبدیل کنیم؟**
  - تبدیل متن به دنباله‌های عددی یکی از مراحل اصلی در پردازش زبان طبیعی (NLP) است. در واقع، مدل‌های یادگیری ماشین مانند KNN و SVM قادر به پردازش مستقیم داده‌های متنی

نیستند؛ به همین دلیل باید این داده‌ها را به شکلی تبدیل کنیم که قابل پردازش باشند. این تبدیل معمولاً به دو صورت انجام می‌شود:

- **توکن‌بندی** (Tokenization) :که متن به کلمات جداگانه تقسیم می‌شود.
- **نمایش عددی** (Numeric Representation) :که هر کلمه به یک عدد تبدیل می‌شود. برای این کار از تکنیک‌های مختلفی مانند Bag of Words، TF-IDF و Word Embeddings و TTSq استفاده می‌شود.
- در کد از Tokenizer استفاده می‌شود که متن را به دنباله‌های عددی تبدیل می‌کند. هر کلمه یک عدد منحصر به فرد دریافت می‌کند و این تبدیل برای مدل‌های یادگیری ماشین ضروری است.

- این بخش داده‌های متنی را به بردارهای عددی تبدیل می‌کند:

- از TfidfVectorizer استفاده می‌شود تا ویژگی‌های مهم از متن استخراج شود. این مدل، اهمیت هر کلمه در متن را براساس میزان استفاده آن کلمه در مستندات مختلف ارزیابی می‌کند.
- max_features=5000 یعنی فقط ۵۰۰۰ ویژگی برتر (کلمات با بیشترین تاثیر) برای مدل انتخاب می‌شود. پس این پارامتر تعداد ویژگی‌ها (کلمات) را به ۵۰۰۰ محدود می‌کند. پس دارای موارد زیر است:
  - **کاهش پیچیدگی محاسباتی :**وقتی تعداد ویژگی‌ها بیشتر می‌شود، حجم محاسبات و زمان آموزش مدل به طور قابل توجهی افزایش می‌یابد. بنابراین، محدود کردن تعداد ویژگی‌ها به ۵۰۰۰، زمان پردازش را کاهش می‌دهد.
  - **بهبود کارایی :**معمولاً تعداد زیادی از ویژگی‌ها اطلاعات مهمی ندارند و ممکن است فقط نویز ایجاد کنند. با محدود کردن تعداد ویژگی‌ها به مقدار معقولی (در اینجا ۵۰۰۰)، می‌توانیم مدل را برای شناسایی ویژگی‌های مهم‌تر و معنادارتر آموزش دهیم.

چگونه تغییر دادن max_features بر عملکرد تأثیر می‌گذارد؟

- اگر مقدار آن را **کمتر** کنیم (مثلاً ۱۰۰۰ ویژگی)، ممکن است مدل ما از اطلاعات کافی برای شناسایی الگوها برخوردار نباشد و دقت کاهش یابد.

- اگر آن را **بیشتر** کنیم (مثلاً ۱۰۰۰۰ ویژگی)، ممکن است مدل ما بیش از حد پیچیده شود و زمان آموزش به شدت افزایش یابد. همچنین ممکن است دچار **overfitting** (پیش‌بینی بهتر روی داده‌های آموزشی و ضعیف‌تر شدن روی داده‌های جدید) شود.

- min_df=5: این پارامتر مشخص می‌کند که کلمات باید در حداقل ۵ سند ظاهر شوند تا به عنوان ویژگی در نظر گرفته شوند.
- max_df=0.95: این پارامتر تعیین می‌کند که کلمات نباید در بیشتر از ۹۵٪ اسناد ظاهر شوند.
- fit_transform برای داده‌های آموزشی استفاده می‌شود و transform برای داده‌های Validation و Test برای اطمینان از استفاده از همان ویژگی‌ها.

8 . آموزش مدل:KNN

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_vec, y_train)
```

- در اینجا مدل KNN با ۵ همسایه آموزش داده می‌شود:

- KNeighborsClassifier مدل طبقه‌بندی KNN است که از شباهت‌ها برای پیش‌بینی استفاده می‌کند.
- Fit مدل را بر اساس داده‌های آموزش X_train_vec و برچسب‌ها y_train آموزش می‌دهد.

9 . ارزیابی مدل:KNN

```
y_val_pred_knn = knn.predict(X_val_vec)
print("KNN on Validation:")
print("Accuracy:", accuracy_score(y_val, y_val_pred_knn))
print("Precision:", precision_score(y_val, y_val_pred_knn))
print("Recall:", recall_score(y_val, y_val_pred_knn))
print("F1 Score:", f1_score(y_val, y_val_pred_knn))
```

- پارامتر n_neighbors=5 یعنی تعداد نزدیک‌ترین همسایگان در نظر گرفته شده برای هر پیش‌بینی برابر با ۵ است.

- این بخش مدل KNN را بر روی داده‌های Validation ارزیابی می‌کند و نتایج زیر را چاپ می‌کند:
  - **Accuracy** :درصد درستی پیش‌بینی‌ها.
  - **Precision** :دقت مدل در شناسایی کلاس مثبت.
  - **Recall** :توانایی مدل در شناسایی تمام نمونه‌های مثبت.
  - **F1 Score** :میانگین هارمونیک Precision و Recall.

10 . آموزش مدلSVM :

```
svm = SVC(kernel='rbf', C=1, gamma='scale')
svm.fit(X_train_vec, y_train)
```

- اینجا مدل **SVM** آموزش داده می‌شود:
  - مدل SVM با هسته rbf(هسته گوسی) استفاده می‌شود. این هسته برای داده‌های غیرخطی مناسب است. پارامتر C=1 برای تنظیم قدرت جریمه و gamma='scale' برای مقیاس کردن مقدار gamma استفاده می‌شود.
  - SVC()مدل ماشین بردار پشتیبان (Support Vector Machine) است.
  - مدل بر اساس داده‌های آموزش X_train_vec و برچسب‌ها y_train آموزش داده می‌شود.

11 . ارزیابی مدل:SVM

```
y_val_pred_svm = svm.predict(X_val_vec)
print("\n SVM on Validation:")
print("Accuracy:", accuracy_score(y_val, y_val_pred_svm))
print("Precision:", precision_score(y_val, y_val_pred_svm))
print("Recall:", recall_score(y_val, y_val_pred_svm))
print("F1 Score:", f1_score(y_val, y_val_pred_svm))
```

- این بخش همانند مدل KNN است، ولی اینجا عملکرد مدل **SVM** را ارزیابی می‌کند.

کرنل(Kernel) SVM

- در (Support Vector Machine) **SVM**، کرنل به تابعی گفته می‌شود که داده‌ها را به یک فضای جدید (معمولاً با ابعاد بالاتر) منتقل می‌کند. این انتقال باعث می‌شود که مرزهای تصمیم‌گیری (hyperplane) که بین کلاس‌ها قرار دارند، پیچیده‌تر و غیرخطی شوند. در حقیقت، کرنل‌ها به **SVM** این امکان را می‌دهند که حتی در مواردی که داده‌ها به طور خطی قابل جداسازی نیستند، به خوبی پیش‌بینی کنند.
    - **انواع کرنل‌ها:**
    - **Linear Kernel** :این کرنل، خطی بودن داده‌ها را فرض می‌کند و بهترین مرز تصمیم‌گیری را به صورت یک خط (یا هایپرپلین) در فضای داده‌ها ایجاد می‌کند.
    - **Polynomial Kernel** :این کرنل برای داده‌هایی که رابطه غیرخطی دارند، مفید است و بین ویژگی‌ها ارتباطات درجه بالاتر را مدل‌سازی می‌کند.
    - **Radial Basis Function (RBF) Kernel** :این کرنل برای داده‌هایی که به وضوح قابل جداسازی نیستند و ویژگی‌های پیچیده‌تری دارند، بهترین است و به طور خودکار فضای داده‌ها را به یک فضای بالاتر انتقال می‌دهد تا نقاط داده‌ها را بهتر از هم جدا کند.
    - در داده‌های متنی، اغلب کرنل rbf به دلیل قدرت آن در شبیه‌سازی مرزهای پیچیده و غیرخطی بهتر عمل می‌کند.
    - **پارامترهای C و gamma**
        - **C (Regularization Parameter) :**
        - پارامتر C مسئول تعیین میزان جریمه به خطاها است. اگر مقدار C بزرگ باشد، مدل تلاش می‌کند تا تمامی داده‌ها را با دقت بیشتری طبقه‌بندی کند و به **overfitting** (فیت شدن بیش از حد روی داده‌های آموزشی) منجر می‌شود.
        - اگر C کوچک باشد، مدل بیشتر به دنبال یک مرز تصمیم‌گیری نرم‌تر است و تحمل بیشتری برای خطا دارد که ممکن است به **underfitting**(عدم یادگیری صحیح مدل از داده‌ها) منجر شود.

- **gamma (Kernel Coefficient) :**
  - پارامتر gamma به نحوه تأثیر هر داده بر تصمیم‌گیری مدل اشاره دارد. اگر gamma بسیار بزرگ باشد، تنها نزدیک‌ترین داده‌ها تأثیر زیادی بر مرز تصمیم‌گیری خواهند داشت که ممکن است باعث **overfitting** شود.
  - اگر gamma بسیار کوچک باشد، مدل می‌تواند بیش از حد نرم باشد و نتواند روابط پیچیده بین داده‌ها را شبیه‌سازی کند.

- **kernel='rbf'** :در اینجا، از کرنل RBF استفاده شده است که برای داده‌های غیرخطی بسیار مناسب است و قدرت بالایی در دسته‌بندی داده‌های پیچیده دارد.
- **C=1** :مقدار C=1برای تنظیم جریمه بهینه است. این مقدار باعث می‌شود که مدل نه تنها به دقت نیاز دارد بلکه باید مرز تصمیم‌گیری را به نحوی تنظیم کند که خطاها را به طور معقولانه‌ای جریمه کند.
- **gamma='scale'** :این تنظیم، مقدار gamma را به صورت خودکار بر اساس تعداد ویژگی‌ها محاسبه می‌کند. این گزینه معمولاً باعث تنظیم بهتر پارامترها و بهبود عملکرد مدل می‌شود.
- مدل SVM با کرنل RBF برای داده‌های متنی که پیچیدگی‌های غیرخطی دارند، مناسب‌تر است. این کرنل به مدل کمک می‌کند تا مرزهای تصمیم‌گیری پیچیده‌تری را پیدا کند، در حالی که کرنل خطی تنها قادر به مدل‌سازی مرزهای خطی است. تنظیمات مناسب برای C و gamma باعث می‌شود که مدل عملکرد بهتری داشته باشد.

12 . ارزیابی نهایی روی داده‌های Test :

```
y_test_pred_knn = knn.predict(X_test_vec)
y_test_pred_svm = svm.predict(X_test_vec)
print("\n Final Comparison on Test Data:")
print("Model\tAccuracy\tPrecision\tRecall\t\tF1 Score")
print(f"KNN\t{accuracy_score(y_test, y_test_pred_knn):.4f}\t\t{precision_score(y_test, y_test_pred_knn):.4f}\t\t{recall_score(y_test, y_test_pred_knn):.4f}\t\t{f1_score(y_test, y_test_pred_knn):.4f}")
```

```
print(f"SVM\t{accuracy_score(y_test,
y_test_pred_svm):.4f}\t\t{precision_score(y_test,
y_test_pred_svm):.4f}\t\t{recall_score(y_test,
y_test_pred_svm):.4f}\t\t{f1_score(y_test, y_test_pred_svm):.4f}")
```

- مدل‌های KNN و SVM روی داده‌های Test ارزیابی می‌شوند.
- نتایج عملکرد هر مدل بر اساس **Accuracy**، **Precision**، **Recall**و **F1 Score**چاپ می‌شود تا بتوان مقایسه‌ای بین آن‌ها انجام داد.

Texts-to-Sequences-2

در این روش، هر کلمه از متن به یک عدد تبدیل می‌شود. این تبدیل معمولاً در مدل‌های **Recurrent Neural Networks** (**RNN**) و **LSTM** استفاده می‌شود که برای پردازش دنباله‌ها مناسب هستند.

ویژگی‌ها:

- **بردارسازی دنباله‌ای** :در این روش، هر کلمه به یک عدد منحصر به فرد تبدیل می‌شود و ترتیب کلمات در دنباله حفظ می‌شود.
- **پدینگ**: از آنجا که دنباله‌ها ممکن است طول متفاوتی داشته باشند (یعنی تعداد کلمات در هر متن می‌تواند متفاوت باشد)، برای اینکه همه دنباله‌ها طول یکسانی داشته باشند، **پدینگ** استفاده می‌شود.
- **پدینگ چیست؟** در این روش برای دنباله‌هایی که کمتر از طول مشخصی هستند، به انتهای دنباله‌ها کلمه‌ای (معمولاً ۰) اضافه می‌شود تا طول همه دنباله‌ها یکسان شود.

مراحل کار با Text to Sequence :

۱. تبدیل کلمات به عدد مثلاً از Tokenizer در Keras .

۲. استفاده از **پدینگ** برای یکسان کردن طول دنباله‌ها.

۳. استفاده از این دنباله‌ها به عنوان ورودی مدل‌های یادگیری عمیق مانند LSTM یا GRU . پس استفاده از این روش در اینجا که با مدل‌های یادگیری ماشین یعنی knn و svm سرکار داریم، بدرد نمی‌خورد.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
import nltk

# 1. خواندن داده‌ها
df_true = pd.read_csv(true_path)
df_fake = pd.read_csv(fake_path)

# 2. برچسب‌گذاری
df_true["label"] = 1
df_fake["label"] = 0
df = pd.concat([df_true, df_fake], axis=0)
df = df.sample(frac=1).reset_index(drop=True)  # مخلوط کردن داده‌ها

# 3. انتخاب ستون‌های اضافی برای بهبود دقت مدل
df['text'] = df['title'] + " " + df['text'] + " " + df['subject'] + " " +
df['date'].astype(str)  # ترکیب عنوان، متن، موضوع و تاریخ

#
================================================================================
==============================

# 4. پیش‌پردازش متن با استفاده از Keras Tokenizer
def clean_text(text):
    # 1. تبدیل حروف بزرگ به حروف کوچک
    text = text.lower()

    # 2. حذف فضاهای خالی اضافی
    text = ' '.join(text.split())

    # 3. تجزیه متن به جملات
    sentences = text.split(".")  # ساده‌ترین روش تقسیم جملات با نقطه

    words = []
    stop_words = set(nltk.corpus.stopwords.words('english'))

    for sentence in sentences:
        # 4. توکن‌بندی جملات به کلمات (با استفاده از str.split)
        tokens = sentence.split()

        # 5. حذف اعداد و URLها
```

```
        tokens = [word for word in tokens if not word.isdigit() and not
word.startswith('http')]

        # 6. حذف علائم نگارشی و ایستواژه‌ها
        tokens = [word for word in tokens if word.isalpha() and word not
in stop_words]

        words.extend(tokens)

    return " ".join(words)

# اعمال پیش‌پردازش روی داده‌ها
df["clean_text"] = df['text'].astype(str).apply(clean_text)

#
================================================================================
================================

# 5. تقسیم داده‌ها
X_train_val, X_test, y_train_val, y_test =
train_test_split(df["clean_text"], df["label"], test_size=0.15,
random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val,
y_train_val, test_size=0.176, random_state=42)  # 0.176×0.85 ≈ 0.15

#
================================================================================
================================

# 6. استفاده از Keras Tokenizer برای تبدیل متن به دنباله‌های عددی
tokenizer = Tokenizer(num_words=5000)  # تعداد کلمات را محدود به ۵۰۰۰
کلمه می‌کنیم
tokenizer.fit_on_texts(X_train)  # آموزش Tokenizer روی داده‌های آموزشی

X_train_seq = tokenizer.texts_to_sequences(X_train)  # تبدیل متن به
دنباله‌های عددی
X_val_seq = tokenizer.texts_to_sequences(X_val)
X_test_seq = tokenizer.texts_to_sequences(X_test)

#
================================================================================
================================

# 7. Padding دنباله‌ها برای یکسان کردن طول دنباله‌ها
```

```
max_length = max([len(seq) for seq in X_train_seq])  # طول دنباله‌ها را
پیدا می‌کنیم (از طول بزرگترین دنباله استفاده می‌شود)

X_train_pad = pad_sequences(X_train_seq, maxlen=max_length,
padding='post')
X_val_pad = pad_sequences(X_val_seq, maxlen=max_length, padding='post')
X_test_pad = pad_sequences(X_test_seq, maxlen=max_length, padding='post')

#
==========================================================================
==============================

# 8. آموزش مدل KNN
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_pad, y_train)

# 9. ارزیابی مدل KNN
y_val_pred_knn = knn.predict(X_val_pad)
print(" _ KNN on Validation:")
print("Accuracy:", accuracy_score(y_val, y_val_pred_knn))
print("Precision:", precision_score(y_val, y_val_pred_knn))
print("Recall:", recall_score(y_val, y_val_pred_knn))
print("F1 Score:", f1_score(y_val, y_val_pred_knn))

#
==========================================================================
==============================

# 10. آموزش مدل SVM
svm = SVC()
svm.fit(X_train_pad, y_train)

# 11. ارزیابی مدل SVM
y_val_pred_svm = svm.predict(X_val_pad)
print("\n _ SVM on Validation:")
print("Accuracy:", accuracy_score(y_val, y_val_pred_svm))
print("Precision:", precision_score(y_val, y_val_pred_svm))
print("Recall:", recall_score(y_val, y_val_pred_svm))
print("F1 Score:", f1_score(y_val, y_val_pred_svm))

#
==========================================================================
==============================

# 12. ارزیابی نهایی روی Test
```

```
y_test_pred_knn = knn.predict(X_test_pad)
y_test_pred_svm = svm.predict(X_test_pad)

print("\n ____Final Comparison on Test Data:")
print("Model\tAccuracy\tPrecision\tRecall\t\tF1 Score")
print(f"KNN\t{accuracy_score(y_test,
y_test_pred_knn):.4f}\t\t{precision_score(y_test,
y_test_pred_knn):.4f}\t\t{recall_score(y_test,
y_test_pred_knn):.4f}\t\t{f1_score(y_test, y_test_pred_knn):.4f}")
print(f"SVM\t{accuracy_score(y_test,
y_test_pred_svm):.4f}\t\t{precision_score(y_test,
y_test_pred_svm):.4f}\t\t{recall_score(y_test,
y_test_pred_svm):.4f}\t\t{f1_score(y_test, y_test_pred_svm):.4f}")
```

```
 _ KNN on Validation:
Accuracy: 0.5813607265148132
Precision: 0.5592137592137593
Recall: 0.6909532483302975
F1 Score: 0.6181423139598045

 _ SVM on Validation:
Accuracy: 0.6601161232693167
Precision: 0.6966161026837806
Recall: 0.5437158469945356
F1 Score: 0.6107416879795396

 ____Final Comparison on Test Data:
Model   Accuracy        Precision       Recall          F1 Score
KNN     0.5791          0.5418          0.6925          0.6079
SVM     0.6775          0.6998          0.5529          0.6177
```

1 . استفاده از Keras Tokenizer برای تبدیل متن به دنباله‌های عددی

tokenizer = Tokenizer(num_words=5000)تعداد کلمات را محدود به ۵۰۰۰ کلمه می‌کنیم

tokenizer.fit_on_texts(X_train)آموزش Tokenizer روی داده‌های آموزشی

توضیح:

- **Tokenizer**یک ابزار از Keras است که متن‌ها را به دنباله‌های عددی تبدیل می‌کند.

- **num_words=5000** :با این پارامتر تعداد کلمات مورد نظر محدود به ۵۰۰۰ کلمه می‌شود. به عبارت دیگر، فقط ۵۰۰۰ کلمه با بیشترین فراوانی در داده‌ها انتخاب می‌شوند و باقی کلمات نادیده گرفته می‌شوند. هدف این کار کاهش تعداد ویژگی‌ها و حذف کلمات کم‌فراوان است که ممکن است اطلاعات کمی برای مدل فراهم کنند.
- **fit_on_texts(X_train)** :این متد روی داده‌های آموزشی اجرا می‌شود تا یک واژه‌نامه (vocabulary)از تمام کلمات موجود در داده‌ها ساخته شود. این واژه‌نامه هر کلمه را به یک شناسه عددی اختصاص می‌دهد.

2 . تبدیل متن به دنباله‌های عددی

```
# X_train_seq = tokenizer.texts_to_sequences(X_train)تبدیل متن به دنباله‌های عددی
X_val_seq = tokenizer.texts_to_sequences(X_val)
X_test_seq = tokenizer.texts_to_sequences(X_test)
```

- **texts_to_sequences** :این متد برای تبدیل هر متن به دنباله‌ای از شناسه‌های عددی استفاده می‌شود. به عبارت دیگر، هر کلمه در متن به شناسه عددی مربوط به آن کلمه در واژه‌نامه (که در مرحله قبل ساخته شد) تبدیل می‌شود.
- **X_train, X_val, X_test** :این متغیرها به ترتیب داده‌های آموزشی، اعتبارسنجی و تست هستند که به دنباله‌های عددی تبدیل می‌شوند.
- مدل‌های یادگیری ماشین مانند SVM و KNN نمی‌توانند مستقیماً با متن‌ها کار کنند. بنابراین، متن‌ها باید به یک نمای عددی تبدیل شوند که برای مدل قابل درک باشد.

3 . Padding دنباله‌ها برای یکسان کردن طول دنباله‌ها

```
# max_length = max([len(seq) for seq in X_train_seq])طول دنباله‌ها را پیدا می‌کنیم (از طول بزرگترین دنباله استفاده می‌شود)
X_train_pad = pad_sequences(X_train_seq, maxlen=max_length, padding='post')
X_val_pad = pad_sequences(X_val_seq, maxlen=max_length, padding='post')
X_test_pad = pad_sequences(X_test_seq, maxlen=max_length, padding='post')
```

- **max_length** :در اینجا، برای تعیین طول دنباله‌ها از بزرگترین دنباله در داده‌های آموزشی استفاده می‌شود. این کار باعث می‌شود که تمام دنباله‌ها (چه کوتاه و چه بلند) به یک اندازه یکسان شوند.
  - max([len(seq) for seq in X_train_seq]): این عبارت طول بزرگترین دنباله در داده‌های آموزشی را پیدا می‌کند.
- **pad_sequences** :این متد برای یکسان کردن طول دنباله‌ها استفاده می‌شود. به عبارت دیگر، دنباله‌هایی که طول کوتاه‌تری دارند به اندازه max_length پر می‌شوند. این پر کردن می‌تواند از سمت شروع یا انتهای دنباله باشد.
  - **maxlen=max_length** :این پارامتر تعیین می‌کند که طول نهایی تمام دنباله‌ها باید برابر با max_length باشد. در صورتی که دنباله‌ای کوتاه‌تر از این مقدار باشد، با صفر پر می‌شود.
  - **padding='post'** :این پارامتر تعیین می‌کند که صفرها به انتهای دنباله اضافه شوند. اگر padding='pre' بود، صفرها به ابتدای دنباله اضافه می‌شدند.

چرا Padding ضروری است؟

- مدل‌های یادگیری ماشین برای پردازش داده‌های ورودی به یک اندازه یکسان نیاز دارند. از آنجایی که هر متن طول متفاوتی دارد، برای جلوگیری از بروز خطا در هنگام ورودی به مدل، باید طول تمام دنباله‌ها یکسان شود.
- **پر کردن دنباله‌ها** باعث می‌شود که تمام داده‌ها برای پردازش توسط مدل‌های یادگیری ماشین، یکپارچه و هم‌طول شوند.

مقایسه‌ی **روش TF-IDF** و **text-to-sequence** که باعث تفاوت قابل توجه در زمان اجرا می‌شود، به دلایل مختلفی از جمله نحوه‌ی پردازش و محاسبات انجام‌شده در هر روش مربوط می‌شود.

۱. **روش TF-IDF** معمولاً برای مدل‌های **غیرعمیق** مانند **SVM** و **KNN** استفاده می‌شود، جایی که ویژگی‌ها (کلمات) از نظر اهمیتی که دارند وزن‌دهی می‌شوند. این مدل‌ها نیازی به ترتیب دنباله‌ها یا حفظ اطلاعات زمانی ندارند.

۲. **Text to Sequence** بیشتر برای مدل‌های **یادگیری عمیق** مثل **LSTM** یا **GRU** استفاده می‌شود که برای پردازش دنباله‌ها مناسب هستند و ترتیب کلمات در جمله مهم است. در این روش، برای تطابق طول دنباله‌ها نیاز به **پدینگ** داریم.

دلایل تفاوت در زمان اجرا:

۱. **پردازش سریع‌تر با TF-IDF :**

- **TF-IDF** تنها به بردارهای ویژگی برای **کلمات کلیدی** و **نمایش‌های عددی** در متن نیاز دارد و به همین دلیل سریع‌تر از **text-to-sequence** است که شامل توکن‌بندی دقیق‌تر و پیچیده‌تری است.
- این فرآیند مستقیماً یک ماتریس **sparse** تولید می‌کند، که منجر به محاسبات سریع‌تر می‌شود. برخلاف مدل‌های **LSTM** و **RNN** نیازی به **Padding** برای داده‌های **Sparse** ندارد، چرا که به طور خودکار طول ویژگی‌ها را هم‌راستا می‌کند.

2 . Optimizing TF-IDF Parameters :

- پارامترهای **TF-IDF** مانند max_features، min_df و max_df می‌توانند به بهینه‌سازی سرعت و دقت کمک کنند. با استفاده از این تنظیمات می‌توانیم تعداد ویژگی‌های مورد استفاده را محدود کرده و از ویژگی‌های غیرضروری جلوگیری کنیم.
- استفاده از پارامترهای **min_df** و **max_df** در **TF-IDF** : این تنظیمات باعث می‌شود که **کلمات با فراوانی کم** یا **بیش از حد** از ویژگی‌ها حذف شوند و ویژگی‌های مؤثرتر باقی بمانند.
- **بهینه‌سازی مدل‌ها: SVM** ابتدا با هسته **linear** تنظیم شد تا پردازش سریع‌تر و دقت بالاتری داشته باشد اما برخلاف فرضیات سبب تضعیف بیشتر مدل تا ۵۷ درصد رفت پس دریافتیم که **SVM** با **linear kernel** برای داده‌های متنی ممکن است بهینه نباشد با اینکه سرعت اجرا را بهبود بخشید. سپس از **kernel های دیگر** مانند **rbf** و تنظیمات مناسب برای **C** و **gamma** استفاده کردیم و به شدت مدل قوی‌تر شد.

۲. **text-to-sequence زمان‌بر است:**

- فرآیند تبدیل **text-to-sequence** شامل چندین مرحله پیچیده مانند توکن‌بندی، حذف علائم نگارشی، و اعمال **padding** است که زمان بیشتری می‌برد. درحالی که روش قبلی نیازی به اعمال پدینگ نداشت حتی سبب بدتر شدن فرایند آموزش می‌شد.
- با استفاده از این روش، ما دنباله‌های عددی می‌سازیم که باید به مدل وارد شوند، که ممکن است در پردازش زمان‌بر باشد.
- پدینگ برای این است که تمام دنباله‌ها (مجموعه‌ای از کلمات که به دنباله عددی تبدیل شده‌اند) به طول یکسان برسند، زیرا بیشتر مدل‌های یادگیری ماشین نیاز دارند که ورودی‌ها طول یکسانی داشته باشند.