به نام خداوند جان و خرد

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

پردازش زبان طبیعی

تمرین شماره ۱

نام و نام خانوادگی: علی خرم فر

شماره دانشجویی: ۸۱۰۱۰۲۱۲۹

اسفندماه ۱۴۰۲

فهرست مطالب

۱_ پاسخ سوال اول ۱

۱-۱_ بخش اول ۱

عدم تشخیص علائم نگارشی: ۱

عدم تشخیص ارقام اعشاری ۲

عدم تشخیص پایان جمله ۲

عدم تشخیص آدرس‌هایی مثل ایمیل ۲

۱-۲_ بخش دوم ۲

عدم تشخیص درست M.Sc. ۳

عدم تشخیص فرمت تاریخ ۳

عدم تشخیص علائم نگارشی ۳

۱-۳_ بخش سوم ۳

۲_ پاسخ سوال دوم ۴

۲-۱_ بخش اول ۴

۲-۲_ بخش دوم ۵

Tokenizer استفاده شده در GPT ۵

Tokenizer استفاده شده در BERT ۵

مقایسه WordPiece و BPE ۶

۲-۳_ بخش سوم ۶

پیاده‌سازی اول – قبل از مشاهده پیام TA ۶

نتیجه پیاده‌سازی اول: ۷

پیاده سازی دوم (اصلی) ۷

خروجی پیاده‌سازی دوم: ۸

۳_ پاسخ سوال سوم ۹

۳-۱_ بخش اول ۹

پیش پردازش‌های انجام‌شده ۹

مرحله‌ی Tokenization ۱۰

۳-۲_ بخش دوم ۱۰

داده‌ی تنک Sparse Data ۱۰

پیاده‌سازی Bigram ۱۱
۳-۳_ بخش سوم ۱۳
خروجی رویکرد اول ۱۳
خروجی رویکرد دوم – بدون Smoothing ۱۳
خروجی رویکرد دوم – پس از Smoothing ۱۴
۴-۳_ بخش چهارم ۱۴
خروجی 3-gram ۱۵
خروجی 5-gram ۱۵
مقایسه مدل‌ها ۱۵
۵-۳_ بخش پنجم ۱۶
۴_ پاسخ سوال چهارم ۱۶
۱-۴_ بخش اول ۱۶
۲-۴_ بخش دوم ۱۷
5_ مراجع ۱۹

۱_ پاسخ سوال اول

۱-۱_ بخش اول

Tokenizer ارائه شده بر حسب کلمه است و بر اساس خروجی زیر جمله کلمات ورودی را در خروجی مشاهده می‌کنیم:

```
import re
def custom_tokenizer(text):
    pattern = r'\b\w+\b'
    tokens = re.findall(pattern, text)
    return tokens

result = custom_tokenizer('How is that even possible?')
print (result)
```

```
['How', 'is', 'that', 'even', 'possible']
```

پس این Tokenizer مبتنی بر کلمه است. از متد findall استفاده شده پس تمامی کلماتی که الگوی مربوط به ورودی pattern را دارد در خروجی چاپ می‌کند. این الگو به صورتی است که باتوجه به ملزومات کتابخانه ابتدای رشته r قرار می‌گیرد با رشته به صورت خام به متد داده شود. در این الگو مشخص شده که ابتدا و انتهای کلمه \b باشد. این به این معنی است که باید یک جداکننده یا مرزی در دو طرف کلمه بوده (مثل _ یا space) سپس در وسط نیز از \w+ استفاده شده که کلمات را تشخص می دهد. \w اول هر کلمه را انتخاب کرده و با توجه به + تا تمام کلمه قبل از جداکننده را انتخاب می‌کند. در اینجا از + استفاده شده تا جداکننده‌هایی مثل Space در خروجی نباشند.

برخی از ایرادات این Tokenizer به شرح زیر است :

عدم تشخیص علائم نگارشی:

```
import re
def custom_tokenizer(text):
    pattern = r'\b\w+\b'
    tokens = re.findall(pattern, text)
    return tokens

result = custom_tokenizer('How?? How is that even possible?')
print (result)
```

```
['How', 'How', 'is', 'that', 'even', 'possible']
```

برای نمونه در مثال بالا علامت تعجب و علامت سوال حذف شده است. این موضوع در تشخیص و تحلیل احساسات از متن بسیار مهم است که در مثال بالا این نمونه کاملا این موضوع را بیان می‌کند.

عدم تشخیص ارقام اعشاری

```
import re
def custom_tokenizer(text):
    pattern = r'\b\w+\b'
    tokens = re.findall(pattern, text)
    return tokens

result = custom_tokenizer('GPA is 2.05')
print (result)
```

```
['GPA', 'is', '2', '05']
```

باتوجه به مثال بالا قسمت صحیح و اعشاری معدل جداگانه دریافت شده است که این موضوع نیز یکی دیگر از مشکلات است.

```
result = custom_tokenizer('This is the end. end of the sentence.')
print (result)
```

```
['This', 'is', 'the', 'end', 'end', 'of', 'the', 'sentence']
```

عدم تشخیص پایان جمله

باتوجه به اینکه برای تشخیص پایان جمله علی‌رغم کاربرد مبهم علائم نگارشی مثل . و ؟ از این موارد برای Classification استفاده می‌شود، با حذف آن‌ها تشخیص پایان جمله مشکل است.

عدم تشخیص آدرس‌هایی مثل ایمیل

```
result = custom_tokenizer('My Email is ali@gmail.com')
print (result)
```

```
['My', 'Email', 'is', 'ali', 'gmail', 'com']
```

باتوجه به ایرادات ذکرشده ایمیل نیز قابل تشخیص نخواهدبود.

به طور کلی حذف این علائم اطلاعات مهمی که در برخی کاربردهای پردازش زبان طبیعی مهم هستند را حذف کرده و در خروجی نمایش نمی‌دهد.

۲-۱_ بخش دوم

خروجی به شرح زیر است :

```
result = custom_tokenizer('Just received my M.Sc. diploma today, on 2024/02/10! Excited to embark on this new journey of knowledg
print (result)
```

```
['Just', 'received', 'my', 'M', 'Sc', 'diploma', 'today', 'on', '2024', '02', '10', 'Excited', 'to', 'embark', 'on', 'this', 'n
ew', 'journey', 'of', 'knowledge', 'and', 'discovery', 'MScGraduate', 'EducationMatters']
```

ایراداتی که در بخش قبلی بررسی شد نیز در این بخش باعث ایجاد مشکل می‌شود که در ادامه چند مورد از آن‌ها بررسی می‌شود:

عدم تشخیص درست M.Sc.

باتوجه به اینکه در خروجی این کلمه به صورت جداگانه M و Sc توکینایز شده و علت این مورد نیز به عدم تشخیص نقطه به صورت درست است، این مورد باعث ابهام می‌شود. نقطه در این کلمه برای اختصار دو کلمه Master و Science است و تمام M.Sc. به معنی Master of Science است. برای اینکه این کلمه به درستی تشخیص داده‌شود نباید به صورت جداگانه قطعه‌بندی شود.

عدم تشخیص فرمت تاریخ

در مثال بالا تاریخ به صورت درست تشخیص داده نشده و باتوجه به عدم تشخیص علامت / تاریخ 2024/02/10 به صورت جداگانه هر کدام از بخش‌های روز و ماه و سال توکنایزه شده که این مورد باعث عدم تشخیص درست تاریخ می‌شود.

عدم تشخیص علائم نگارشی

در این مثال علامت ! تعجب تشخیص داده نشده که این علامت در تحلیل احساسات نقش مهمی دارد. همچنین هشتگ‌ها مشخص نیست که هشتگ بوده و به نوعی کلمات کلیدی محسوب می‌شود. همچنین باتوجه به اینکه نقطه تشخیص داده نشده و پس از آن هشتگ آمده پایان جمله مشخص نیست.

۳-۱_ بخش سوم

یکی از مهم‌ترین مشکلاتی که اشاره شد مربوط به علائم نگارشی بود که مثلا در تاریخ مشکل ایجاد میکرد که با توجه به تغییرات انجام شده در کد مربوطه این مورد نیز حل شد.

```
def fixed_custom_tokenizer(text):
    pattern = r'\b[\w./!]+'
    tokens = re.findall(pattern, text)
    return tokens

result = fixed_custom_tokenizer('Just received my M.Sc. diploma today, on 2024/02/10! Excited to embark on this new journey of kr
print (result)
```

```
['Just', 'received', 'my', 'M.Sc.', 'diploma', 'today', 'on', '2024/02/10!', 'Excited', 'to', 'embark', 'on', 'this', 'new', 'j
ourney', 'of', 'knowledge', 'and', 'discovery.', 'MScGraduate', 'EducationMatters.']
```

این کار با یک تغییر بسیار ساده انجام شد. کافی است که به دنبال ترکیباتی باشیم که که حداقل ۱ واحد از موارد بالا را داشته باشند. برای اینکار در خارج براکت از + استفاده کرده و علائم ! / و . را پس از \w قرار می‌دهیم. در آخر الگو نیز \b را حذف می‌کنیم. به این صورت همانطور که در خروجی بالا نیز مشخص می‌شود، مشکل مربوط به تاریخ و M.Sc. حل شده است. همچنین نقطه آخر جمله نیز در توکن مربوطه لحاظ

شده است. برای جزئیات بیشتر میتوان علامت , نیز برای توکن Today اضافه کرد. که این مورد به همراه کد الگوی اضافه کردن هشتگ نیز در کد پیوست شده انجام شد.

۲_پاسخ سوال دوم

در این قسمت از تمرین به بررسی توکنایزرهای استفاده شده دو مدل زبان بزرگ BERT و GPT می‌پردازیم.

BERT [۱] در سال ۲۰۱۸ توسط گوگل معرفی شد. این مدل از اولین مدل‌هایی است که از معماری Transformers استفاده کرده و به درک بهتر متن کمک کرده و در بسیاری از دیگر کاربردها همچون ترجمه و پیش‌بینی کلمه بعدی و همچنین خلاصه‌سازی متن استفاده می‌شود.(Devlin et al., 2018)

GPT [۲] در سال ۲۰۱۸ توسط OpenAI توسعه یافته و معماری آن مانند BERT بر پایه Transformerهاست و از رویکرد pre-training followed by fine-tuning استفاده می‌کند. در این رویکرد ابتدا مدل توسط داده‌های بزرگ و عمومی آموزش داده شده و سپس با داده‌های خاص‌تر و با هدف عملکرد خاصی تنظیم یا Fine-Tune می شود.

۲-۱_ بخش اول

باتوجه به بررسی‌های انجام شده هردو توکنایزری که معرفی شد، مبتنی بر زیرکلمه هستند.

به طور کلی ۳ نوع توکنایزر در طول درس معرفی شد که به طور خلاصه آن‌ها را بررسی می‌کنیم.

توکنایزر مبتنی بر حرف متن را به کاراکترهای تشکیل‌دهنده آن تقسیم می‌کند.

توکنایزر مبتنی بر کلمه که هر کلمه را به عنوان یک توکن در نظر می‌گیرد. این مورد باعث می‌شود که برخی کلمات بزرگ و طولانی که نمونه آن در زبان‌های مختلف در کلاس درس بررسی شد به عنوان یک توکن در نظر گرفته شده و این امر موجب می‌شود تا یادگیری مدل‌های زبانی کاهش یافته و عملکرد مناسبی نسبت به این نوع از کلمه‌ها نداشته باشند.

نوع سوم نیز مبتنی بر زیرکلمه است که یک تعادل بین روش اول و دوم برقرار می‌کند. این نوع از توکنایزرها کلمات موجود در متن را به زیرکلمات معنی‌دار تبدیل کرده و کمک می‌کنند که مدل‌های زبانی

---

[۱] Bidirectional Encoder Representations from Transformers

[۲] Generative Pre-trained Transformer

کارایی بیشتری داشته باشد. این رویکرد در زبان‌هایی که ریخت‌شناسی یا Morphology در ساخت کلمات آن نقش پررنگی دارد به فهم بهتر مدل کمک می‌کند.

همانطور که در بالا نیز اشاره شد به طور کلی دلایل مختلفی برای استفاده از این نوع توکنایزر در مدل‌های زبانی بزرگ وجود دارد. از جمله اینکه اندازه کلمه بر کارایی مدل تاثیر داشته و مدل‌هایی که تمام کلمه را به عنوان توکن در نظر میگیرند علاوه بر مشکل بزرگ‌بودن اندازه توکن منجر می‌شود که کلماتی از نظر ظاهری و Morphology ریشه یکسانی دارند متفاوت شناخته شده و کارایی مدل را کاهش دهند. علاوه بر این برخی کلمات ناشناخته برای مدل نیز با شناسایی زیرکلمه آن‌ها قابل شناسایی می‌شوند. زیرا که برخی کلمات بزرگ از ترکیب چند زیرکلمه بامعنی و شناخته‌شده برای مدل ساخته می‌شوند. که همین امر به مدل کمک می‌کند تا ساختار کلمه‌ها و ریخت‌شناسی آن‌ها را بهتر یادگرفته و در رویارویی با کلمات ناشناخته و جدید بهتر عمل کند. این مورد خصوصا برای تطبیق مدل با زبان‌های جدید کمک‌کننده بوده، انعطاف‌پذیری مدل را افزایش داده و حتی با حجم داده کم می‌توان مدل را آموزش داد و نتیجه بهتری نسبت به حالتی گرفت که هر کلمه یک توکن باشد.

۲-۲_ بخش دوم

Tokenizer استفاده شده در GPT

در این مدل زبانی از توکنایزر BPE استفاده می‌شود که این الگوریتم در مقاله (Sennrich, Haddow, & Birch, 2015) معرفی شد. باتوجه به مطالب ارائه‌شده در اسلاید کلاس درس، این الگوریتم بر اساس یک متد فشرده‌سازی متن عمل می‌کند. این توکنایزر مبتنی بر زیرکلمه است و باتوجه به فراوانی هر توکن که ممکن است حرف یا یک زیرکلمه باشد، عمل ادغام را انجام می‌دهد.

Tokenizer استفاده شده در BERT

در این مدل زبانی از توکنایزر WordPiece استفاده می‌شود. این نوع از توکنایزر برای اولین بار در (Schuster & Nakajima, 2012) معرفی شد. عملکرد این الگوریتم شبیه به BPE است و برای ساخت توکن‌های جدید عمل ادغام را انجام می‌دهد با این تفاوت که در این الگوریتم جفتی که بیشترین فراوانی را دارند انتخاب نمی‌کند بلکه جفتی برای ادغام انتخاب می‌شوند که بیشترین احتمال را نسبت به توکن‌های قبل از ادغام داشته باشند. یعنی a و b در صورتی برای ادغام انتخاب می‌شوند که احتمال ab نسبت به a و b بیشتر از هر جفت نماد دیگری باشد. به بیان دیگر likelihood داده‌های آموزشی را در واژگان تولیدی بیشینه می‌کند.

مقایسه WordPiece و BPE

همانطور که گفته‌شد هردوی این روش‌ها مبتنی بر زیرکلمه هستند و هردوی آن‌ها از حروف و کاراکترهای پایه برای شروع استفاده کرده و در یک فرآیند تکراری یا Iterative عمل ادغام را انجام می‌دهند. با هربار عمل ادغام توکن ها و یا کاراکترهای کوچکتر یک توکن جدید ساخته شده و این عمل تا تعداد مشخصی انجام می‌شود. تفاوت اصلی این دو روش در شرط ادغام است. همانطور که در بخش قبل بررسی شد عمل ادغام در BPE بر اساس بیشترین تعداد تکرار و یا فراوانی جفت توکن مورد نظر است. در صورتی که در WordPiece این عمل بر پایه داده‌های ارائه شده آموزشی است و بر اساس اینکه جفت مورد نظر بیشترین احتمال را طبق داده‌های مشاهده شده از آن زبان داشته باشند، انتخاب می‌شوند. به طور کلی روش BPE سریعی است و نه تنها در مدل‌های زبانی در دیگر کاربردهای NLP مثل ترجمه ماشینی کاربرد دارد. روش WordPiece باتوجه به اینکه بر اساس داده‌های آموزشی است احتمال اینکه توکن‌های مفیدتر و بهینه‌تری برای مدل بسازد بیشتر است.

۳-۲_ بخش سوم

در پیاده‌سازی این الگوریتم‌ها دو رویکرد مورد بررسی قرار گرفت. یکی ادامه رویکرد آموزش تا زمانی که تعداد واژگان به حد خاصی برسد. که بدیهی است در این حالت تعداد واژگان هردو الگوریتم برابر است. و دیگری بر اساس پارامتر تعداد ادغام‌ها که باتوجه به لزوم گزارش تعداد واژگان از رویکرد دوم استفاده شد.

نتایج تا حدود زیادی مشابه هم هستند و سعی شده تمامی پاسخ‌های تمرین با کمک هر دو پیاده‌سازی پاسخ داده شوند. از جنبه آموزشی پیاده‌سازی اول بسیار به فهم مفاهیم و الگوریتم‌های استفاده شده کمک می‌کند.

پیاده‌سازی اول – قبل از مشاهده پیام TA

به طور کلی در شرایط یکسان و با تکرار یکسان، تعداد واژگان روش BPE به نسبت Wordpiece بیشتر بود که به دلیل حروف اولیه‌ای بود که در الگوریتم پیاده‌سازی آن اضافه شده بود. ولی به طور کلی باتوجه به اینکه تعداد حلقه تکرار یکسان در نظر گرفته شد، تعداد واژگان هردو الگوریتم برابر است. باتوجه به اینکه این پیاده‌سازی قسمت اول این سوال را پوشش نمیدهد، پیاده‌سازی دوم انجام شد ولی کد برنامه اول نیز پیوست شده است. تعداد کلمات روی ۱۰۰۰ تنظیم شد. و پاسخ‌ها نیز مناسب بود. در این پیاده‌سازی از نسخه‌های پیش‌آموزش دیده استفاده نشده است.

در این پیاده‌سازی متدهای مختلفی استفاده شده است. در متد train تعداد فراوانی هر کلمه محاسبه شده سپس کاراکترهای اولیه به واژگان اضافه می‌شوند. سپس هر کلمه به کاراکترهای تشکیل‌دهنده‌اش تقسیم می‌شود. سپس در یک حلقه به تعداد پارامتر تکرار، هربار جفت با بیشترین فراوانی ادغام می‌شوند. این مرحله

برای روش Wordpiece بر اساس امتیازی است که به هر جفت داده می‌شود. متد *compute_pair_freqs* برای محاسبه فراوانی جفت استفاده می‌شود که کاربردش در متد *train* بررسی شد. متد *merge_pair* هم برای ادغام دو جفت بکار می‌رود. متد *tokenize* هم برای توکنایز کردن متن‌های ارائه‌شده در تمرین استفاده می‌شود.

نتیجه پیاده‌سازی اول:

خروجی متن اول روش BPE:

```
['T', 'is', 'ar', 'nes', 'is', 'b', 'o', 'ute', 'y', 'illing', 'I', 'we
', 'ver', 'a', 'et', 'istri', 'again', 'it', 'ust', 'eab', 'ut', 'he',
'im', 'of', 'hes', 'e', 'Moon', '</w>']
```

خروجی متن اول روش Wordpiece:

```
['This', 'd##', '[UNK]', 'is', 'a##', '[UNK]', 'k##', '[UNK]', 'If', 'w
e', 'ever', 'take', 'this', 't##', '[UNK]', 'a##', '[UNK]', 'it', 'must
', 'be', 'about', 'the', 'time', 'of', 'the', 's##', '[UNK]', '[UNK]']
```

خروجی متن دوم روش BPE:

```
['T', 'is', 'ar', 'nes', 'is', 'b', 'o', 'ute', 'y', 'illing', 'I', 'we
', 'ver', 'a', 'et', 'istri', 'again', 'it', 'ust', 'eab', 'ut', 'he',
'im', 'of', 'hes', 'e', 'Moon', '</w>']
```

خروجی متن دوم روش Wordpiece:

```
['This', 'is', 'a', 'to##', '[UNK]', 't##', '[UNK]', '[UNK]', 'is', 'th
e', 'first', 'st##', '[UNK]', 'in', 'a', '[UNK]', 'p##', '[UNK]', 'We',
'will', 'be', 'c##', '[UNK]', 'the', 'to##', '[UNK]', 'ge##', '[UNK]',
'by', 'each', 'to##', '[UNK]', 'm##', '[UNK]']
```

نتیجه ای که از مقایسه دو رشته در پیاده‌سازی اول حاصل شد این بود که در متن اول نتیجه Wordpiece بهتر بود زیرا که نوع این متن بیشتر شبیه به متن کتاب است و باتوجه به اینکه Wordpiece در مرحله آموزش بر اساس بالا بودن احتمال وجود کلمه در داده آموزشی عمل می‌کند، این موضوع توجیه‌پذیر است. ولی در متن دوم عملکرد BPE خیلی کم بهتر بود زیرا که بیشتر کلمه‌ها خارج از موضوع کتاب بودند و احتمال اینکه با ترکیب دیگر توکن‌ها در مرحله آموزش BPE بدست آمده‌باشند بیشتر است.

پیاده سازی دوم (اصلی)

در این قسمت برای مقایسه دو توکنایزر از کتابخانه tokenizers از huggingface استفاده شده‌است. باتوجه به خواسته سوال در ساده‌ترین حالت ممکن متد مربوط به آموزش کتاب اجرا شد. هر دو متد استفاده شده در این تمرین از نوع Pretrained نیستند.

تعداد واژگان روش BPE پس از آموزش برابر است با ۱۱۳۲۲

تعداد واژگان روش Wordpieceپس از آموزش برابر است با ۸۷۵۵

دلیل این اختلاف در این روش را می‌توان این موضوع دانست که در روش Wordpiece هر جفتی را به واژگان اضافه نمی‌کند و فقط جفت‌هایی اضافه می‌کند که ارزش اضافه‌شدن را داشته‌باشند و این موضوع قبل از هربار عمل ادغام و اضافه شدن به واژگان بررسی می‌شود. که همین موضوع باعث می‌شود واژگان انتخاب شده در این روش برای داده ورودی بهینه باشند.

خروجی پیاده‌سازی دوم:

خروجی متن اول روش BPE:

```
['This', 'Ġdarkness', 'Ġis', 'Ġabsolutely', 'Ġkilling', '!', 'ĠIf', 'Ġw
e', 'Ġever', 'Ġtake', 'Ġthis', 'Ġtrip', 'Ġagain', ',', 'Ġit', 'Ġmust',
'Ġbe', 'Ġabout', 'Ġthe', 'Ġtime', 'Ġof', 'Ġthe', 'Ġs', 'New', 'ĠMoon',
'!']
```

خروجی متن اول روش Wordpiece:

```
['this', 'darkness', 'is', 'absolutely', 'killing', '!', 'if', 'we', 'e
ver', 'take', 'this', 'trip', 'again', ',', 'it', 'must', 'be', 'about'
, 'the', 'time', 'of', 'the', 'sne', '##w', 'moon', '!']
```

خروجی متن دوم روش BPE:

```
['This', 'Ġis', 'Ġa', 'Ġto', 'ken', 'ization', 'Ġtask', '.', 'ĠT', 'oke
n', 'ization', 'Ġis', 'Ġthe', 'Ġfirst', 'Ġstep', 'Ġin', 'Ġa', 'ĠN', 'L'
, 'P', 'Ġpip', 'el', 'ine', '.', 'ĠWe', 'Ġwill', 'Ġbe', 'Ġcomp', 'aring
', 'Ġthe', 'Ġto', 'k', 'ens', 'Ġgener', 'ated', 'Ġby', 'Ġeach', 'Ġto',
'ken', 'ization', 'Ġmod', 'el', '.']
```

خروجی متن دوم روش Wordpiece:

```
['this', 'is', 'a', 'to', '##ken', '##ization', 'task', '.', 'to', '##k
en', '##ization', 'is', 'the', 'first', 'step', 'in', 'a', 'n', '##l',
'##p', 'pip', '##el', '##ine', '.', 'we', 'will', 'be', 'compar', '##in
g', 'the', 'to', '##ken', '##s', 'generated', 'by', 'each', 'to', '##ke
n', '##ization', 'mode', '##l', '.']
```

علت وجود کاراکتر Ġ در ابتدای توکن‌ها در روش BPE از کتابخانه Tokenizer این است که space به کاراکتر ویژه Ġ تبدیل می‌شود.Space جز هر کلمه است و این کاراکتر نمایانگر space قبل از توکن است.

در این روش که از کتابخانه Tokenizer استفاده شده نیز مانند روش قبلی، متن اول باتوجه به نزدیکی به متن کتاب، عملکرد Wordpiece بهتر بوده و کلمات بیشتری را تشخیص داده است. در صورتی که متن

دوم این مورد برقرار نیست و بسیاری از کلمات متن دوم باتوجه به اینکه متن در حوزه فنی است قابل تشخیص نبوده است و کلماتی تشخیص داده‌شده که عمومی تر هستند و احتمالا در کتاب استفاده شده‌اند.

در آخر نیز از یک متد دیگر که در این کتابخانه ارائه شده نیز استفاده شد Tokenizer(models.WordPiece()) و الگوریتم Wordpiece مجددا به این روش اجرا شد. اندازه واژگان در این اجرا برابر با ۲۳۸۷۶ بود. نکته قابل توجه این بود که این الگوریتم تعداد واژگان بیشتری نسبت به BPE روش قبلی دارد.

نکته قابل توجه این بود که در این روش عملکرد بسیار عالی در متن اول است:

خروجی متن اول:

```
['this', 'darkness', 'is', 'absolutely', 'killing!', 'if', 'we', 'ever'
, 'take', 'this', 'trip', 'again,', 'it', 'must', 'be', 'about', 'the',
'time', 'of', 'the', 'sne', '##w', 'moon!']
```

خروجی متن دوم:

```
['this', 'is', 'a', 'to', '##ken', '##ization', 'task.', 'to', '##ken',
'##ization', 'is', 'the', 'first', 'step', 'in', 'a', 'n', '##l', '##p'
, 'pip', '##el', '##ine.', 'we', 'will', 'be', 'comparing', 'the', 'to'
, '##ken', '##s', 'generated', 'by', 'each', 'to', '##ken', '##ization'
, 'model', '##.']
```

باتوجه به بررسی انجام شده تفاوت این ۲ پیاده‌سازی از Wordpiece از کتابخانه Tokenizer فقط در زبان پیاده‌سازی آن‌ها بوده و دلیل تفاوت احتمالا به دلیل یکسان نبودن تنظیمات اولیه قبل از آموزش است.

۳_ پاسخ سوال سوم

۳-۱_ بخش اول

پیش پردازش‌های انجام‌شده

پس از بارگذاری فایل کتاب تارزان، باتوجه به اینکه هدف از این تمرین تحلیل احساسات نیست، می‌توانیم تمامی کلمات را به حروف کوچک تبدیل کنیم.

```
corpus = corpus.lower()
```

باتوجه به مطالب مطالعه‌شده در وب‌سایت analyticsvidhya قبل از ساخت n-gram بهتر است stop words مثل and و or حذف شوند. ولی این مورد را در این مرحله از اجرا انجام **نمی‌دهیم**.

در مرحله بعد از نرمال‌سازی کاراکترهای خاص را حذف می‌کنیم.

normalized_text = re.sub(r'[^a-zA-Z0-9\s]', '', corpus)

در این روش از عبارات منظم کمک می‌گیریم.

مرحله‌ی Tokenization

در این مرحله برای توکنایزکردن متن ورودی از کتابخانه NLTK کمک می‌گیریم. این کتابخانه پکیجی از Tokenizer ها ارائه می‌دهد که می‌توانیم از آن‌ها استفاده کنیم. به این منظور با دستور nltk.download('punkt') موارد مورد نیاز دانلود شده و از متد word_tokenize برای توکنایزکردن ورودی استفاده می‌کنیم.

Tokenization

```
In [50]: import nltk
         from nltk.tokenize import word_tokenize
         tokenized_text = word_tokenize(normalized_text)
         tokenized_text

Out[50]: ['the',
          'project',
          'gutenberg',
          'ebook',
          'of',
          'tarzan',
          'lord',
          'of',
          'the',
          'jungle',
          'this',
          'ebook',
          'is',
          'for',
          'the',
          'use',
          'of',
          'anyone',
          'anywhere',
          'in',
          'the',
```

۲-۳_ بخش دوم

داده‌ی تنک Sparse Data

داده‌ی تنک یا Sparse Data داده‌ای است که تعداد زیادی مقادیر صفر دارد. داده‌ی تنک را نباید با داده‌ی از دست رفته یا missing data اشتباه گرفت زیرا داده‌ی تنک مقادیر خالی یا صفر را نشان می‌دهد در حالیکه داده‌ی از دست رفته به این معنی است که برخی از مقادیر از دست رفته یا گم شده‌اند و مقادیر آنها مشخص نیست و معمولاً این مقادیر از دست رفته را با null نشان می‌دهند. (*Spars Data*, 2023)

این مشکل در Bigram وجود دارد و این موضوع در اسلایدهای محتوای درسی به طور کامل بررسی شد. این مشکل مربوط به زمانی است که مقدار ۰ به احتمال Bigram تخصیص داده شود. این موضوع منجر می‌شود که محاسبه Perplexity به دلیل خطای تقسیم بر ۰ غیرممکن شود. برای حل این مشکل از روش‌های Smoothing می‌توان استفاده کرد که روش لاپلاس یا add_one در اسلایدهای درسی بررسی شد.

احتمال Bigram از فرمول زیر محاسبه می‌شود:

$$P(w_i|w_{n-1}) = \frac{Count(w_{i-1}|w_i)}{Count(w_{i-1})}$$

که در ادامه این مورد در برنامه پیاده‌سازی می‌شود.

برای حل مشکل Data Sparsity با کمک روش Laplace از رابطه زیر استفاده می‌شود:

$$P_{add-1} = \frac{C(w_{i-1}|w_i) + 1}{C(w_{i-1}) + V}$$

که این روش را با کمک توابع آماده اعمال می‌کنیم.

پیاده‌سازی Bigram

در اولین مرحله Bigram ها باید ساخته شوند. به این منظور از کد زیر استفاده می‌شود:

nltk.ngrams(tokenized_text, 2)

در این روش، تمام کلمات به دو لیست تقسیم شده و سپس هر عنصر از لیست اول با عنصر بعدی از لیست دوم جفت می‌شود. این عملیات تمام جفت کلمات ممکن را تولید می‌کند.

برای محاسبه احتمالات باید $Count(w_{i-1}|w_i)$ برای Bigram که تعداد تکرار یک کلمه پس از یک کلمه دیگر است بدست آورده بر مقدار $Count(w_{i-1})$ که تعداد تکرار یک کلمه است تقسیم شود. این مقادیر با کمک تابع Counter در متغیرهای مربوطه ذخیره می‌کنیم.

bigrams_count = Counter(bigrams)

words_count = Counter(tokenized_text)

سپس فرمول بالا را پیاده‌سازی می‌کنیم و با توجه به تمامی جفت‌های ممکن تولید شده با کمک itertools.product ، مقادیر احتمال را برای هرکدام جداگانه محاسبه می‌کنیم:

Bigram

```
from collections import Counter
import itertools

# Finding Bigrams
bigrams = nltk.ngrams(tokenized_text, 2)
bigrams_count = Counter(bigrams)
words_count = Counter(tokenized_text)

uniqueWords = []
for i in tokenized_text:
      if not i in uniqueWords:
        uniqueWords.append(i);

all_bigrams = list(itertools.product(uniqueWords, uniqueWords))
len(all_bigrams)
```

Calculate Probabilities

```
bigram_probabilities = {}

for bigram in all_bigrams:
    word = bigram[0]
    bigram_probabilities[bigram] = bigrams_count[bigram] / words_count[word]
```

برخی از خروجی:

```
In [132]: bigram_probabilities

Out[132]: {('the', 'the'): 0.0,
           ('the', 'project'): 0.005938456001439625,
           ('the', 'gutenberg'): 0.0,
           ('the', 'ebook'): 0.0,
           ('the', 'of'): 0.0,
           ('the', 'tarzan'): 0.0,
           ('the', 'lord'): 0.00035990642432967427,
           ('the', 'jungle'): 0.01709555515565953,
           ('the', 'this'): 0.0,
           ('the', 'is'): 0.0,
           ('the', 'for'): 0.0,
           ('the', 'use'): 0.001079719272989023,
           ('the', 'anyone'): 0.0,
           ('the', 'anywhere'): 0.0,
           ('the', 'in'): 0.0,
           ('the', 'united'): 0.002699298182472557,
           ('the', 'states'): 0.0,
           ('the', 'and'): 0.0,
           ('the', 'most'): 0.001439625697318697,
```

به طور کلی ۳۹۵۵۳ جفت با کمک تابع bigram از کتابخانه NLTK بدست آمده است. از طرفی تمامی ترکیب‌های ممکن که همان خانه‌های جدول می‌باشند، تعدادی برابر با ۴۶,۸۸۱,۴۰۹ دارند. یعنی بیش از ۹۹ درصد جفت‌ها احتمال صفر دارند.

```
In [128]: zeros=0
          for i in bigram_probabilities:
              if(bigram_probabilities[i]==0):
                  zeros+=1
          zeros

Out[128]: 46841856
```

ولی هدف در این پیاده‌سازی، حالتی است که مشکل تنک‌بودن داده‌ها وجود نداشته باشد. باتوجه به کد زیر ضریبی به اسم آلفا ضرب در احتمالات جمع شده و مقدار تقسیم شده نیز با تعداد بایگرام در آلفا جمع می‌شود.

Smoothed Probabilities

```
In [8]: add1_probabilities = {}
        V = len(uniqueWords)
        for bigram in all_bigrams:
                word = bigram[0]
                alpha = 1
                add1_probabilities[bigram] = ( bigrams_count[bigram] + alpha) / (words_count[word] + (alpha*V))

        add1_probabilities

Out[8]: {('the', 'the'): 8.061915511125443e-05,
         ('the', 'project'): 0.0027410512737826506,
         ('the', 'gutenberg'): 8.061915511125443e-05,
         ('the', 'ebook'): 8.061915511125443e-05,
         ('the', 'of'): 8.061915511125443e-05,
         ('the', 'tarzan'): 8.061915511125443e-05,
         ('the', 'lord'): 0.0002418574653337633,
         ('the', 'jungle'): 0.007739438890680426,
         ('the', 'this'): 8.061915511125443e-05,
         ('the', 'is'): 8.061915511125443e-05,
         ('the', 'for'): 8.061915511125443e-05,
         ('the', 'use'): 0.000564334085778781,
```

برای تایید نهایی بار دیگر تعداد صفرها را میشماریم.

```
zeros=0
for i in add1_probabilities:
    if(add1_probabilities[i]==0):
        zeros+=1
zeros
```

```
0
```

که مشاهده می‌شود پس از Smoothing صفر شد.

۳-۳_ بخش سوم

برای این بخش از دو رویکرد استفاده کرده‌ایم.

رویکرد اول کلمه بعدی را بر اساس بیشترین احتمال در جدول Bigram انتخاب می‌کند.

Completing Sentence

first Approach

```
2]: import numpy as np
    sentence = "For half a day he lolled on the huge back and"

    for _ in range(10):
        last_word = sentence.split()[-1]
        possible_words = [word for (prev_word, word) in add1_probabilities.keys() if prev_word == last_word]
        probabilities = [add1_probabilities[(last_word, word)] for word in possible_words]
        next_word_idx = np.argmax(probabilities)
        # next_word = max(possible_words, probabilities)[0]
        next_word = possible_words[next_word_idx]
        sentence += " " + next_word

    print(sentence)
```

```
For half a day he lolled on the huge back and the great tourney and the great tourney and the great
```

رویکرد دوم کلمه بعدی را بر اساس نمونه‌برداری و سپس به صورت تصادفی انتخاب می‌کند. هرچه که احتمال احتمال کلمه بعدی در جدول احتمالات بیشتر باشد، احتمال انتخاب آن بیشتر است.

خروجی رویکرد اول

```
Knowing well the windings of the trail he had been a great tourney and
the great tourney and
```

```
For half a day he lolled on the huge back and the great tourney and the
great tourney and the great
```

خروجی رویکرد دوم – بدون Smoothing

```
Knowing well the windings of the trail he had witnessed the freedmen an
d whats wrong with project Gutenberg
```

```
For half a day he lolled on the huge back and silver and the spokesman all
the sun was to their
```

مشاهده می‌شود که کیفیت این رویکرد بهتر است.

خروجی رویکرد دوم – پس از Smoothing

```
Knowing well the windings of the trail he already fastened defenders of
fended disappear waziri axiomatic clambered urge bonds


For half a day he lolled on the huge back and met quotasee laborers mused p
ig crown harmed treasure restlessness asserted
```

باتوجه به اینکه در این رویکرد احتمالات دخیل است و از طرفی احتمالات زیادی از طرف Smoothing بر داده‌هایی که در واقع کلمه بعدی نیامده‌اند اضافه شده، این رویکرد پس از Smoothing کیفیت مناسب ندارد.

اگر در قسمت پیش پردازش StopWords را حذف کنیم، نتیجه این مرحله به صورت زیر خواهدشد:

۴-۳_ بخش چهارم

باتوجه به اینکه در این مرحله نیاز است ترکیبات خیلی بیشتری از کلمات را ذخیره کرده تا محاسبات مربوط به ngram بر روی آن‌ها اعمال شود، یک رویکرد دیگری متفاوت با بخش قبلی برای پیاده‌سازی 3gram و 5gram پیاده‌سازی شد.

```
def create_n_gram_model(tokenized_text, n):
    bigrams = nltk.ngrams(tokenized_text, n)
    bigrams_count = Counter(bigrams)
    words_count = Counter(nltk.ngrams(tokenized_text, n-1))
    return bigrams_count, words_count

trigrams, bigrams = create_n_gram_model(tokenized_text, 3)
quadgrams, pentagrams = create_n_gram_model(tokenized_text, 5)
```

در این روش نیازی نیست که تمامی احتمالات در ابتدا محاسبه شده و سپس به تولید کلمات بعدی بپردازیم. کافی است هربار یک سری n-1 تایی را در نظر گرفته و احتمالا کلمه nم را از بین تمام کلمات ممکن محاسبه کرده و سپس بر اساس آن تصمیم بگیریم که کدام کلمه انتخاب شود که این کار توسط تابع Smoothing انجام می‌شود.

```
def smoothing(n_grams, n_minus_grams, n_minus_gram):
    smoothed_n_gram = []
    smoothed_n_gram_probs = []
    for word in uniqueWords:
        smoothed_n_gram.append(n_minus_gram + (word, ))
        if n_minus_gram + (word, ) in n_grams.keys():
            temp = n_grams[smoothed_n_gram[-1]]
        else:
            temp = 0
        smoothed_n_gram_probs.append((temp + 1)/(n_minus_grams[n_minus_gram] + len(n_minus_grams)))
    return smoothed_n_gram, smoothed_n_gram_probs
```

خروجی 3-gram

```
Knowing well the windings of the trail he did not know that he had been
the original company


For half a day he lolled on the huge back and forth slowly there were n
o others about the lists and
```

خروجی 5-gram

```
Knowing well the windings of the trail he took short cuts swinging thro
ugh the branches of the trees


For half a day he lolled on the huge back and the the the the the the t
he the the the
```

مقایسه مدل‌ها

با بررسی خروجی‌های جمله‌ها می‌توان متوجه شد که نتایج بعد از Smoothing در 3-gram از کیفیت مطلوبی برخوردار است. در Bigram کلمات فقط به صورت جفت ارتباط معنایی ضعیف دارند. ولی جملات در 3-gram ارتباط بهتری با یکدیگر دارند. در 5-gram جمله اول به خوبی تکمیل شده‌است ولی باید توجه دشت که ادامه این جمله دقیقا از متن کتاب است و بیش‌برازش رخ داده‌است و تشخیص کلمه بعدی عمومیت ندارد. این مورد در جمله دوم کاملا مشخص است و دلیل آن این است که ترکیب ۴ کلمه آخر این جمله در متن ورودی که مدل با آن آموزش دیده، یافت نشده است. احتمالات کلمات بعدی در زیر مشخص هستند که همان احتمالات اضافه شده پس از Smoothing هستند:

```
(smoothing(pentagrams, quadgrams, ('the', 'huge', 'back', 'and'))[1])
```

```
[1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
 1.4226167612706813e-05,
```

۵-۳_ بخش پنجم

خیر- شاید از نظر عملی امکان‌پذیر باشد ولی این عمل منجر به کاهش کارایی مدل خواهدشد. نمونه آن را در مثال بالا و در مورد 5-gram دیدیم و متوجه شدیم که عمومیت 3-gram بیشتر است. پس یکی از دلایل این است که این امر منجر می‌شود که مشکل بیش‌برازش یا Overfitting رخ دهد و مدل فقط بر روی داده‌های آموزشی خوب کار کند. دلیل دیگر این است که با افزایش دنباله کلمات احتمال وجود رشته با آن کلمه کاهش یافته و مشکل Sparse Data شدیدتر می‌شود. از طرفی مشکل حافظه نیز در این مورد وجود دارد. هرچند که در روش ما از رویکرد دیگری استفاده شد ولی اگر بخواهیم احتمالات تمام داده‌ها را محاسبه کنیم حجم داده‌ها خیلی زیاد خواهدشد. پس به طور کلی تا یک حد خاصی از n مدل به خوبی جواب می‌دهد و پس از آن کارایی کاهش می‌یابد.

۴_پاسخ سوال چهارم

۴-۱_ بخش اول

ابتدا n=2 را تنظیم می‌کنیم. یعنی مدلی که آموزش می‌بیند Bigram است. تابع زیر داده‌های تست را به همراه فراوانی مثبت و فراوانی منفی و همچنین n و alpha دریافت می‌کند. Alpha به عنوان پارامتر تنظیم Smoothing استفاده می‌شود. در این تابع تعداد کلمات مثبت و منفی محاسبه می‌شود. یعنی فراوانی کلمات قبلی که در فرمول در مخرج قرار می‌گیرد. ما در ابتدا به صورت زیر کلمات قبلی را طبق فرمول Bigram قرار میدادیم ولی متوجه شدم که در این صورت مقدار دقت و صحت مدل پایین می‌آید:

```
accuracy = 0.7262569832402235
precision = 0.5714285714285714
```

پس بجای این کار تعداد کل فراوانی مثبت و منفی در مخرج قرار می‌گیرد. که در این حالت دقت و صحت بالاتری دریافت شد:

```
accuracy = 0.7988826815642458
precision = 0.7073170731707317
```

```python
# Step 5: test the n-gram
def test_ngram(data, positive_freq, negative_freq, n, alpha):
    results = []

    total_positive_count = sum(positive_freq.values())
    total_negative_count = sum(negative_freq.values())
    V = len(set(positive_freq.keys()).union(set(negative_freq.keys())))

    for index, row in data.iterrows():
        ngrams = get_ngrams(row['review'], n)
        positive_probability = 0
        negative_probability = 0

        alpha = alpha
        for ngram in ngrams:
            positive_probability += (positive_freq.get(ngram, 0) + alpha) / (total_positive_count + alpha*V)
            negative_probability += (negative_freq.get(ngram, 0) + alpha) / (total_negative_count + alpha*V)
            #print(ngram)

        if positive_probability > negative_probability:
            label = 1
        else:
            label = 0

        results.append((row['review'],row['polarity'], label))

    return results
```

در این تابع تمامی داده‌های تست در یک حلقه بررسی شده و برای هرکدام n-gram را بدست آورده و در یک حلقه داخلی تر، احتمال اینکه کلمات آن نظر پس از کلمات دریافت شده از داده‌های منفی و یا مثبت باشد محاسبه می‌شود. در این حلقه احتمالات به مانند یک امتیاز برای هرکلمه محاسبه شده و جمع آن‌ها امتیاز کل یک نظر در داده‌های تست را به ما می‌دهد. پس هرکدام که بیشتر بود لیبل مربوطه را بر روی داده تست قرار می‌دهیم.

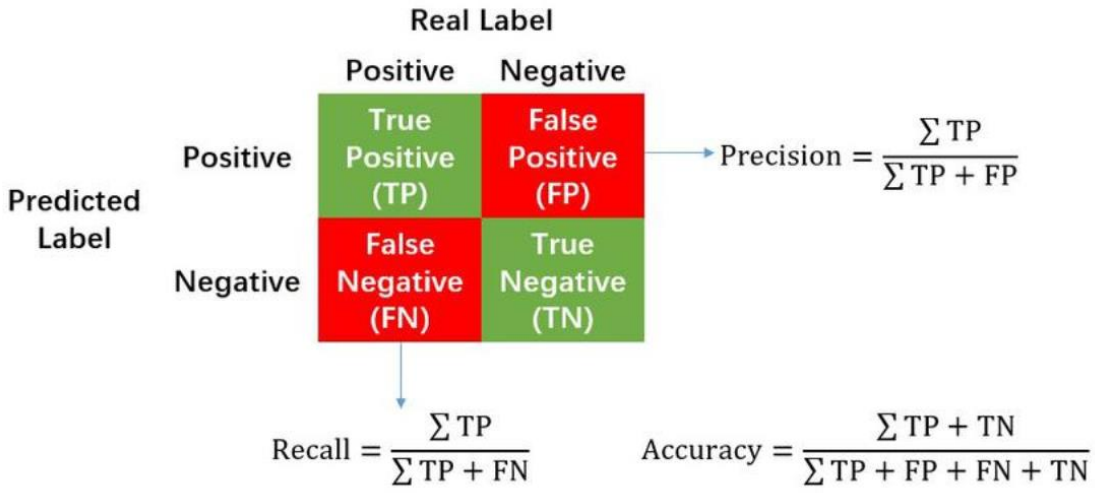
خروجی این تابع متن نظر به همراه لیبل واقعی آن و همچین لیبلی است که مدل تشخیص داده:

```
results = test_ngram(test_data, positive_freq, negative_freq, n , 1)
results
```

```
[(" love/hate has bug and security issues. i tried to report that facebook and google plus have security issues and it would
n't allow me to do so! well i just did didn't i! ......",
  0,
  0),
 (' whatsapp i use this app now that blackberry messenger has basically gone away. my friends & family live all over the worl
d. this really helps keep us in touch!',
  1,
  1),
 (' usefully verry  nice app', 1, 1),
 (' fonts why in the heck is this thing analysing my fonts??? not really quick browsing when i have to wait 5minutes for the
fonts to load. are you asking my opinion? avoid this. terrible',
  0,
  0),
 (" app doesn't work after latest upgrade the facebook app refuses to work on my mobile data (3g) after the latest upgrade! i
t says it cannot connect right now.",
  0,
  0),
```

۲-۴_ بخش دوم

از کتابخانه sklearn برای محاسبه accuracy، precision و recall استفاده می‌کنیم.



بهترین حالت زمانی است که ضریب Smoothing یا همان alpha برابر ۱ باشد:

```
accuracy = 0.7988826815642458
precision = 0.7073170731707317
recall = 0.5471698113207547
```

اگر ضریب را برابر صفر قرار دهیم، یعنی Smoothing انجام نشود:

```
accuracy = 0.7318435754189944
precision = 0.5384615384615384
recall = 0.660377358490566
```

صحت و دقت پایین می‌آید ولی recall یا فراخوانی افزایش می‌یابد.

اگر n=1 قرار داده شود یعنی unigram:

```
accuracy = 0.7653631284916201
precision = 0.5901639344262295
recall = 0.6792452830188679
```

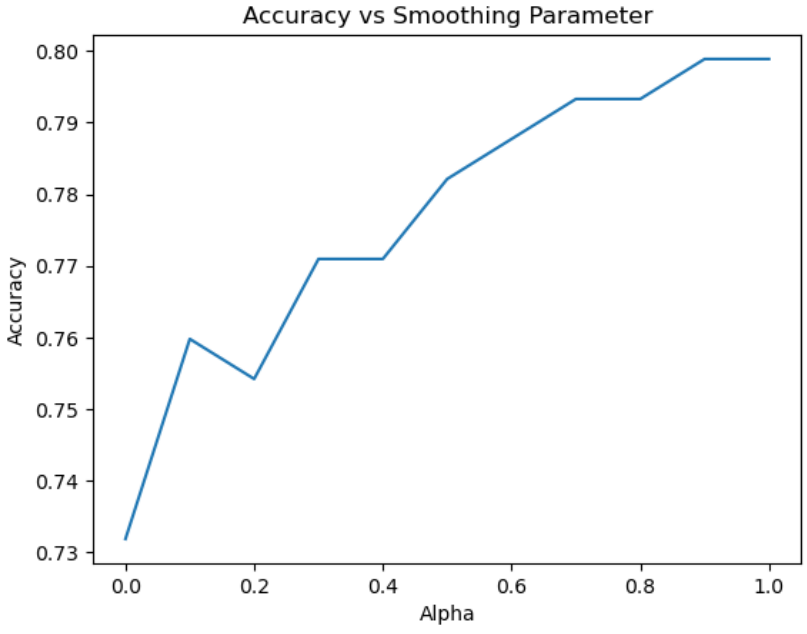
باز هم نسبت به تنظیم اولیه دقت و صحت کمتری دارد.

برای n=3

```
accuracy = 0.6536312849162011
precision = 0.41818181818181815
recall = 0.4339622641509434
```

می‌توان نتیجه گرفت در این مدل n=2 انتخاب خوبی است. n=1 کیفیت کافی نداشت پس یکی افزایش دادیم و نتیجه بهبود پیدا کرد. ولی افزایش مجدد باعث کاهش کارایی شد که در بخش ۵ سوال قبل این مورد بررسی شد.

به طور کلی می‌توانیم تاثیر ضرایب مختلف Smoothing بر روی Accuracy را در نمودار زیر مشاهده کنیم.



۵_مراجع

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Schuster, M., & Nakajima, K. (2012). Japanese and korean voice search. 2012 IEEE international conference on acoustics, speech and signal processing (ICASSP),
Sennrich, R., Haddow, B., & Birch, A. (2015).

Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Spars Data. (2023). https://onlinebme.com/sparse-data/