به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

درس پردازش زبان طبیعی

پاسخ تمرین ۱

نام و نام خانودگی: زهرا ریحانیان

شماره دانشجویی: ۸۱۰۱۰۱۱۷۷

اسفند ماه ۱۴۰۲

فهرست

پاسخ سوال اول

کد مربوط به این بخش در مسیر codes/Q1.ipynb موجود است.

پاسخ بخش اول

مبتنی بر کلمه است. این tokenizer ایراداتی دارد که مهم ترین آن این است که با این روش مدل نمی تواند معنی کلماتی را که خارج از واژگان هستند، درک کند و مواجهه با آن ها می تواند مشکل ساز باشد.

علاوه بر آن چالش ها و محدودیت هایی نیز دارد که به شرح زیر است:

۱. برای زبان هایی که بین کلمات فاصله ندارند، این روش مشکل ساز می تواند باشد. مثل زبان چینی و ژاپنی
۲. کلماتی که به اختصار نوشته شده اند را به درستی tokenize نمی کند. مثل .M.Sc یا مثلا در I'm باید دو توکن باشد اما ممکن است تشخیص ندهد و یک توکن در نظر بگیرد.
۳. مشکل در tokenization تاریخ، لینک یا هشتگ ها. مثال:

2020/08/12
https://www.google.com
#Good_food یا #GoodFood

۴. مشکل با علائم نگارشی که ممکن است آن ها را دور بریزد یا در نظر نگیرد در حالی که در مواردی که معنی برای ما مهم است، داشتن این علائم می تواند مهم باشد.
۵. مشکل است کلمات مرکب را مدیریت کند. مثلا New York را ممکن است به دو توکن تقسیم کند.

پاسخ بخش دوم

جمله داده شده به صورت زیر tokenize شد:

```
['Just',
 'received',
 'my',
 'M',
 'Sc',
 'diploma',
 'today',
 'on',
 '2024',
 '02',
 '10',
 'Excited',
 'to',
 'embark',
```

```
'on',
'this',
'new',
'journey',
'of',
'knowledge',
'and',
'discovery',
'MScGraduate',
'EducationMatters']
```

دو مورد از مشکلاتی که در این tokenization رخ داد عبارت است از:

۱. علائم نگارشی را دور می ریزد.
۲. در tokenization تاریخ همه اعداد مربوط به روز و ماه و سال را جدا کرده است. همچنین در .M.Sc هم M و Sc را جدا کرده است.

پاسخ بخش سوم

کد داده شده را به صورت زیر تغییر دادم:

```
def my_custom_tokenizer(text):
  pattern = r'[^\w\s]|\b[\w\S]+\b'
  tokens = re.findall(pattern, text)
  return tokens
```

پترنی که تعریف کردم برای تاریخ و مختصر ها به خوبی عمل می کند و به درستی آن ها را tokenize می کند. همچنین علائم نگارشی هم دور نمی اندازد. نتیجه به صورت زیر حاصل شد:

```
['Just',
'received',
'my',
'M.Sc',
'.',
'diploma',
'today',
',',
'on',
'2024/02/10',
'!',
'Excited',
'to',
'embark',
'on',
'this',
'new',
'journey',
```

```
'of',
'knowledge',
'and',
'discovery',
'.',
'#',
'MScGraduate',
'#',
'EducationMatters',
'.']
```

پاسخ سوال دوم

کد مربوط به این بخش در مسیر codes/Q2.ipynb موجود است.

پاسخ بخش اول

Tokenizer هر دو مبتنی بر زیرکلمه است. الگوریتم های این نوع از tokenizer ها بر این اصل تکیه می کند که کلمات پرکاربرد نباید به زیرکلمه های کوچک تر تقسیم شوند، بلکه کلمات نادر باید به زیرکلمه های معنادار تجزیه شوند.

نشانه گذاری زیرکلمه برای مدیریت کلمات خارج از فرهنگ لغت و برای درک معنای کلمات در متن مفید است. با داشتن توکن های زیرکلمه (و اطمینان از اینکه هر کاراکتر بخشی از فرهنگ لغت زیرکلمه هستند)، امکان رمزگذاری کلماتی که حتی در داده‌های آموزشی وجود نداشتند را فراهم می‌کند.

این روش اندازه های فرهنگ لغت قابل کنترل را می دهد. شبکه های عصبی فعلی به یک فرهنگ لغت توکن گسسته بسته از پیش تعریف شده نیاز دارند. اندازه واژگانی که یک شبکه عصبی می‌تواند از عهده آن برآید، بسیار کمتر از تعداد کلمات مختلف (شکل‌های سطحی) در اکثر زبان‌های معمولی، به‌ویژه زبان‌های غنی از نظر صرف‌شناسی (و به‌ویژه زبان‌های چسبنده) است.

پراکندگی داده ها را کاهش می دهد. در یک واژگان مبتنی بر کلمه، کلمات کم بسامد ممکن است در داده های آموزشی بسیار کم ظاهر شوند. استفاده از توکنیزاسیون زیرکلمه امکان استفاده مجدد توکن را می دهد و فرکانس پیدایش آنها را افزایش می دهد.

هم چنین شبکه های عصبی با آنها عملکرد بسیار خوبی دارند.

همه ی این ها باعث می شود که با اتخاب توکن سازی زیر کلمه این مدل های زبانی بهتر و کارآمد تر عمل کنند.

پاسخ بخش دوم

مدل زبانی GPT از الگوریتم Byte-Pair Encoding (رمزگذاری بایت جفت) و مدل زبانی BERT از الگوریتم WordPiece استفاده می کند. در ادامه به معرفی و مقایسه این دو الگوریتم می پردازیم.

رمزگذاری بایت جفت (BPE) ابتدا به عنوان یک الگوریتم برای فشرده سازی متون توسعه داده شد و سپس توسط OpenAI برای توکن سازی هنگام پیش آموزش مدل GPT استفاده شد. این توسط بسیاری از مدل های ترانسفورماتور از جمله GPT، GPT-2، RoBERTa، BART و DeBERTa استفاده می شود. BPE به یک پیش نشانه‌ساز متکی است که داده‌های آموزشی را به کلمات تقسیم می‌کند. پیش توکن‌سازی می‌تواند به سادگی توکن‌سازی فضایی باشد، برای مثال GPT-2، RoBERTa. پیش توکن سازی پیشرفته تر شامل توکن سازی مبتنی بر قانون است، به عنوان مثال. XLM، FlauBERT که از Moses برای اکثر زبان ها استفاده می کند، یا GPT که از spaCy و ftfy برای شمارش فراوانی هر کلمه در مجموعه آموزشی استفاده می کند. پس

از پیش نشانه گذاری، مجموعه ای از کلمات منحصر به فرد ایجاد شده و فراوانی وقوع هر کلمه در داده های آموزشی مشخص شده است. در مرحله بعد، BPE یک واژگان پایه متشکل از تمام نمادهایی که در مجموعه کلمات منحصر به فرد رخ می دهد ایجاد می کند و قوانین ادغام را می آموزد تا یک نماد جدید از دو نماد از واژگان پایه تشکیل دهد. این کار را تا زمانی انجام می دهد که واژگان به اندازه واژگان مورد نظر برسد. توجه داشته باشید که اندازه واژگان مورد نظر یک فراپارامتر است که قبل از آموزش توکنایزر باید تعریف شود.

WordPiece الگوریتم توکن سازی زیرکلمه ای است که برای BERT، DistilBERT و Electra استفاده می شود. این الگوریتم در جستجوی صوتی ژاپنی و کره ای تشریح شد و بسیار شبیه BPE است. WordPiece ابتدا واژگان را برای گنجاندن هر کاراکتر موجود در داده های آموزشی مقداردهی اولیه می کند و به تدریج تعداد معینی از قوانین ادغام را یاد می گیرد.

برخلاف BPE، WordPiece متداول‌ترین جفت نماد را انتخاب نمی‌کند، بلکه یکی را انتخاب می‌کند که احتمال داده‌های آموزشی را پس از افزودن به واژگان به حداکثر می‌رساند. در واقع WordPiece به جای انتخاب متداول ترین جفت، با استفاده از فرمول زیر امتیازی را برای هر جفت محاسبه می کند و بهترین جفت که احتمال مشاهده را بیشینه می کند را انتخاب می کند:

$$score=(freq\_of\_pair)/(freq\_of\_first\_element\times freq\_of\_second\_element)$$

پاسخ بخش سوم

- پیاده سازی الگوریتم: برای این کار از کتابخانه hugging face tokenizer استفاده کردم و مدل BPE را برای پیاده سازی الگوریتم Byte-Pair Encoding و مدل WordPiece را برای پیاده سازی الگوریتم WordPiece، import کردم. همچنین trainer مربوط به هر یک را import کردم. برای این که بهینه باشد از ماژول pre_tokenizers متد Whitespace را import و استفاده کردم. در نهایت tokenizer مربوط به هر یک را روی فایل داده شده آموزش دادم.

- اندازه واژگان برای هر یک از الگوریتم ها به صورت زیر بدست آمد:

WordPiece vocabulary size: 17557

Byte-Pair Encoding vocabulary size: 16552

دو عدد بدست آمده متفاوت هستند. WordPiece تعداد واژگان بیشتری دارد علت این است که برای ادغام توکن ها سخت گیری بیشتری دارد و بررسی می کند آیا ارزشش را دارد که دو توکن را با هم ادغام کند یا خیر. برای همین تعداد واژگان آن هم بیشتر شد.

- دو جمله داده شده به هر یک از مدل ها داده شد. نتیجه به صورت زیر بدست آمد:

WordPiece:

Sentence 1:

```
['This', 'darkness', 'is', 'absolutely', 'killing', '!', 'If', 'we', 'ever',
'take', 'this', 'trip', 'again', ',', 'it', 'must', 'be', 'about', 'the',
'time', 'of', 'the', 's', '##N', '##ew', 'Moon', '!']
```

Sentence 2:

```
['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', '.']
```

Byte-Pair Encoding:

Sentence 1:

```
['This', 'darkness', 'is', 'absolutely', 'killing', '!', 'If', 'we', 'ever',
'take', 'this', 'trip', 'again', ',', 'it', 'must', 'be', 'about', 'the',
'time', 'of', 'the', 's', 'New', 'Moon', '!']
```

Sentence 2:

```
['This', 'is', 'a', 'to', 'ken', 'ization', 'task', '.', 'T', 'ok', 'en',
'ization', 'is', 'the', 'first', 'step', 'in', 'a', 'N', 'L', 'P', 'pi',
'pe', 'line', '.', 'We', 'will', 'be', 'comparing', 'the', 'to', 'k', 'ens',
'generated', 'by', 'each', 'to', 'ken', 'ization', 'model', '.']
```

برای جمله اول تفاوت در کلمه sNew بود که در واژگان آن ها نبود و به طور متفاوت tokenize شد. در WordPiece به صورت `'s', '##N', '##ew'` و در BPE به صورت `'s', 'New'`.

در جمله دوم Tokenization و pipeline و tokens متفاوت tokenize شده اند.

در WordPiece به ترتیب به صورت `'To', '##ken', '##ization'`، `'pip', '##el', '##ine'` و `'to', '##ken', '##s'` و در BPE به ترتیب به صورت `'T', 'ok', 'en', 'ization'`، `'pi', 'pe', 'line'` و `'to', 'k', 'ens'` توکن سازی انجام شده است.

این تفاوت به این علت است که معیار ادغام در دو الگوریتم متفاوت است. BPE بیشترین فرکانس را در نظر می گیرد اما WordPiece دو جفتی که احتمال مشاهده را بیشینه می کنند را انتخاب می کند. همچنین در الگوریتم WordPiece پسوند ها به صورت "##" در ابتدای زیر کلمه مشخص شده در صورتی که در BPE چنین چیزی را نداریم.

پاسخ سوال سوم

کد مربوط به این بخش در مسیر codes/Q3.ipynb موجود است.

پاسخ بخش اول

فایل داده شده در برنامه بارگیری و ذخیره شد. برای پیش پردازش آن، از همان tokenization ای که در سوال ۱ کد آن را نوشتم و بر پایه کلمه بود، استفاده کردم. همچنین برای این که کلماتی مثل the و The دو بار شمارش نشوند، همه کلمات را lower case کردم. برای تشکیل واژگان هم خروجی corpus را به تابع set دادم که همه ی توکن ها را به صورت یکتا برگرداند.

اندازه corpus و واژگان به شرح زیر بدست آمد:

Size of corpus: 85163

Size of vocabulary: 6970

پاسخ بخش دوم

در این بخش ابتدا تعداد هر کلمه را بدست آوردم و در یک دیکشنری ذخیره کردم. در مرحله بعد bigram های موجود را پیدا و تعداد آن ها را محاسبه کردم و در یک دیکشنری دیگری ذخیره کردم. بخشی آن به صورت زیر است:

```
{('\ufeff', 'the'): 1,
 ('the', 'project'): 33,
 ('project', 'gutenberg'): 87,
 ('gutenberg', 'ebook'): 3,
 ('ebook', 'of'): 1,
 ('of', 'tarzan'): 26,
 ('tarzan', ','): 47,
 (',', 'lord'): 10,
 ('lord', 'of'): 11,
 ('of', 'the'): 965,
 ('the', 'jungle'): 94,
 ('jungle', 'this'): 1,
 ('this', 'ebook'): 8,
 ('ebook', 'is'): 2,
 ('is', 'for'): 3,
 ('for', 'the'): 96,
 ('the', 'use'): 6,
 ('use', 'of'): 10,
...}
```

یکی از چالش های اصلی n-gram ها پراکندگی داده ها یا data sparsity است، به این معنی که برخی از n-gram ها ممکن است به طور مکرر یا اصلاً در داده های آموزشی رخ ندهند و در نتیجه احتمال آن کم یا صفر شود. این می‌تواند به مدل‌های زبانی

نادرست یا ناقص منجر شود که نتوانند گوناگونی و تنوع زبان طبیعی را درک کنند. پراکندگی داده ها همچنین می تواند بر مقیاس پذیری و کارایی n-gram تأثیر بگذارد، زیرا ذخیره و پردازش مدل های n-gram بزرگ می تواند پرهزینه و وقت گیر باشد. یکی از راه‌های غلبه بر پراکندگی داده‌ها، استفاده از تکنیک‌های هموارسازی یا smoothing است که با توزیع مجدد احتمالات n-gram های دیده شده، احتمالات غیرصفری را به n-gram های نادیده یا نادر اختصاص می‌دهند. یکی از روش های smoothing، تخمین Add-one است و فرمول آن به صورت زیر است:

$$P_{Add-1}(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + V}$$

برای محاسبه احتمال مربوط به هر bigram از فرمول بالا استفاده کردم و احتمال ها را بدست آورده و در یک دیکشنری ذخیره کردم.

پاسخ بخش سوم

برای این بخش دو تابع تعریف کردم. complete_sentence_by_number و complete_sentence_by_word.

تابع complete_sentence_by_number، متن مورد نظر و تعداد کلماتی (n) را که باید بعد از آن تولید کند، دریافت می کند. ابتدا متن را پیش پردازش می کند. یعنی lower case و tokenize می کند سپس n بار تابع complete_sentence_by_word را فراخوانی می کند که آخرین کلمه جمله را می گیرد و کلمه جدید تولید می کند و آن را به انتهای جمله اضافه و آن را چاپ می کند.

تابع complete_sentence_by_word به ازای هر توکن در واژگان، یک bigram شامل آخرین کلمه ی جمله و آن توکن می سازد و احتمال آن را محاسبه می کند و در یک دیکشنری ذخیره می کند. اگر این bigram قبل دیده شده بود، احتمال آن ذخیره شده بود و همان را در دیکشنری ذخیره می کند وگرنه احتمال آن را برابر 1/(count(word)+V) که V اندازه واژگان و word همان آخرین کلمه ی جمله داده شده است، قرار می دهد و در دیکشنری ذخیره می کند.

در نهایت ۳ تایی که بیشترین احتمال را دارند را بدست آورده و بین آن ها به طور تصادفی انتخاب می کند و کلمه را بر می گرداند. این کار به این علت انجام شده است که اگر کلمه با بیشترین احتمال را انتخاب می کردم آن گاه اگر دو کلمه شبیه به هم در یک bigram بیشترین احتمال را داشته باشند، تا آخر جمله را با همان توکن یکسان تکمیل می کرد. برای مثال داده شده، جمله اول، از یک جا به بعد این اتفاق برای توکن " افتاد ولی با اصلاح تابع این مشکل برطرف شد.

خروجی برای جمله اول:

Knowing well the windings of the trail he had not the great tourney , and his own people

خروجی برای جمله دوم:

For half a day he lolled on the huge back and the ape-man , but a few moments the great,

پاسخ بخش چهارم

مشابه کاری که برای bigram انجام شد برای trigram و 5-gram نیز انجام شد. بیشترین تفاوت در محاسبه ی احتمال بود. در تابع trigram_complete_sentence_by_word برای محاسبه احتمال ابتدا بررسی شد که احتمال آن از قبل محاسبه شده یا خیر. اگر نبود باید بررسی شود جفت کلمه ی اول trigram در دیکشنری شمارش bigram ها موجود است یا خیر. اگر موجود بود، احتمال به صورت 1/(bigram_counts[bigram]+V) که V اندازه واژگان و bigram جفت کلمه ی اول trigram است. اگر هم موجود نبود، احتمال را 1/V ذخیره می کند. در trigram، مدل بر اساس دو کلمه آخر جمله، کلمه ی بعدی را تولید می کند.

خروجی برای جمله اول:

Knowing well the windings of the trail he took himself factions panting panting factions cast belongings panting panting

خروجی برای جمله دوم:

For half a day he lolled on the huge back and forth panting panting cast cast belongings belongings panting belongings belongings

برای 5-gram هم عملیاتی مشابه انجام شد. در 5-gram، مدل بر اساس چهار کلمه آخر جمله، کلمه ی بعدی را تولید می کند.

خروجی برای جمله اول:

Knowing well the windings of the trail he took short glanced history glanced history history belongings glanced history

خروجی برای جمله دوم:

For half a day he lolled on the huge back and panting panting cast glanced factions factions factions factions glanced cast

پاسخ بخش پنجم

خیر، چون افزایش پارامتر n در مدل های زبان n-gram چالش هایی را به همراه خواهد داشت:

۱. افزایش پیچیدگی محاسباتی: با افزایش n، تعداد n-gram های ممکن به صورت تصاعدی افزایش می یابد که منجر به هزینه های محاسباتی بالاتر برای آموزش و استنتاج می شود.
۲. پراکندگی داده ها: با n بزرگتر، تخمین های احتمال برای n-gram های دیده نشده به طور فزاینده ای پراکنده می شوند که منجر به عملکرد تعمیم ضعیف تر می شود، به ویژه در مواردی که داده های آموزشی محدود است.

۳. نیازهای حافظه: ذخیره و پردازش تعداد یا احتمالات n-gram به حافظه بیشتری با افزایش n نیاز دارد که احتمالاً از منابع موجود بیشتر می شود.
۴. overfitting: n بالاتر می تواند منجر به overfit شود که در آن مدل به جای یادگیری الگوهای کلی، داده های آموزشی را به خاطر می سپارد و در نتیجه باعث کاهش عملکرد در داده های دیده نشده می شود.

پاسخ سوال چهارم

کد مربوط به این بخش در مسیر codes/Q4.ipynb موجود است.

پاسخ بخش اول

برای تکمیل تابع test_ngram، یک for روی داده تست زدم که به ازای هر سطر، n-gram های ستون review را استخراج کند. سپس به ازای هر یک از n-gram های استخراج شده، فرکانس آن را در positive_freq و negative_freq به طور جداگانه می یابیم و این فرکانس ها را جداگانه تحت عنوان امتیاز مثبت و امتیاز منفی با هم جمع می کنیم. در واقع به n-gram ای که بیشتر در دیکشنری تکرار شده، بیشتر بها می دهیم. چرا که به نوعی نشان می دهد این کلمه چقدر مثبت و یا چقدر منفی است و در کلاس بندی به ما کمک می کند.

در نهایت مقدار هر امتیازی بیشتر بود، یعنی داده ی آن سطر به آن کلاس مربوط است و این کلاس به آن تعلق می گیرد.

در نهایت این تابع برچسب های تخمینی را برمی گرداند.

پاسخ بخش دوم

با استفاده از برچسب های واقعی و برچسب های پیش بینی شده که از بخش قبل بدست آوردیم، دقت مدل بدست آمد که حدودا 76% شد که دقت نسبتا خوبی است. یعنی در ۷۶ درصد مواقع، مثبت یا منفی بودن کامنت ها را به درستی تشخیص می دهد.