الهه محمدی ۹۶۱۳۱۱۳۵	پروژه نهایی	درس پردازش آماری زبان‌های طبیعی

خلاصه سازی متون و اسناد از جمله کاربردهای مهم در پردازش زبان‌های طبیعی است. روش های متعددی از گذشته تا به امروز برای آن پیشنهاد شده است اما با توجه به دقت‌های بدست آمده در این روش‌ها، همچنان رویکردهای دیگری برای بهبود دقت خلاصه‌سازی معرفی می‌شود. یکی از تکنیک‌هایی که در این حوزه مورد استفاده قرار گرفته است تکنیک *sparseCoding* است که مبتنی بر اعمالی از ضرب‌های برداری عمل می‌کند و در این پروژه نیز با به کارگیری یکی از الگوریتم‌های موجود در زیرمجموعه‌ی *sparseCoding* خلاصه‌سازی اسناد بر روی پیکره‌ای فارسی به نام پیکره "پاسخ" صورت می‌گیرد.

sparseCoding چیست؟

بردارهای ویژگی تصاویر، متون و سیگنال‌های صوتی را می‌توان با استفاده از ترکیب خطی تنکی از بردارهای اصلی آن‌ها نمایش داد. این بردارهای اصلی، بردارهای پایه محسوب می‌شوند و به مجموعه‌ی آن‌ها ماتریس دیکشنری گفته می‌شود. به صورت ریاضی ماتریس داده را برابر $X = [x_1,x_2,\dots,x_n] \in R^{m*n}$ در نظر بگیرید. پس از آن‌فرض کنید $B=[b_1,b_2,\dots,b_k] \in R^{m*k}$ ماتریس دیکشنری ما باشد که در آن هر b_i یک بردار پایه را نشان می‌دهد و $S=[s_1,s_2,\dots,s_n] \in R^{k*n}$ ماتریس ضرایب باشد که هر ستون آن نمایش *sparse* ای برای هر کدام از داده‌های ماست که در اینجا سند هستند. هر سند می‌تواند به صورت یک ترکیب خطی از بردارهای پایه در دیکشنری باشد:

$$X = BS$$

X و S هر دو یک سیگنال را نمابش می دهند که یکی در حوزه‌ی فضا و دیگری در حوزه‌ی دیکشنری است. برای تولید ضرایب اسپارس(S) مسئله‌ی نمایش اسپارس به صورت زیر است:

$$\hat{s_i} = \arg\min_{s_i} \| s_i \|_0 \quad \text{s.t.} \quad x_i = Bs_i$$

در رابطه‌ی بالا $||.||$ نرم صفر است که تعداد ورودی‌های غیر صفر را در یک بردار می‌شمارد. اما کمینه کردن نرم صفر سخت است و الگوریتمی که بتواند آن را حل کند موجود نیست. ثابت شده است که اگر s_i به اندازه‌ی کافی اسپارس باشد حل مسئله‌ی بهینه‌سازی نرم صفر معادل با حل کردن مسئله‌ی مینیمم‌سازی نرم یک است. از طرف دیگر این مسئله به عنوان یک مسئله‌ی بهینه‌سازی مطرح شده است و بنابراین تابعی که مقدار خطای مسئله را نشان می‌دهد به صورت زیر تعریف می‌شود:

$$\hat{s_i} = \arg\min_{s_i} \| x_i - Bs \|_2^2 + \lambda \| s_i \|_1$$

در این رابطه λ پارامتری است که *trade off* میان خطای بازسازی و اسپارسیتی را بالانس می‌کند. برای آموزش B روش‌های مختلفی وجود دارد. روش حل ما برای این موضوع برگرفته از الگوریتم موجود در مقاله‌ی ارجاع داده شده است. توضیحات داده شده در بالا توضیحاتی کلی از مسئله‌ی *sparseCoding* بود و در ادامه روش مقاله‌ی مورد استفاده[1] در این پروژه بررسی می‌شود.

[1]Summarizing Answers in Non-Factoid Community Question-Answerin,2017

الهه محمدی ۹۶۱۳۱۱۳۵	پروژه نهایی	درس پردازش آماری زبان‌های طبیعی

الگوریتم مورد استفاده در این مقاله برای تولید خلاصه از مجموعه جملات کاندید که از دسته الگوریتم‌های زیرمجموعه‌ی *sparseCoding* است، *Coordinate descent algorithm* می‌باشد و طی آن یک بردار به نام A ساخته می‌شود. هر خانه از بردار A یک امتیاز به جمله‌ی کاندید نظیر آن نسبت می‌دهد. تابع خطای نیز براساس رابطه‌ی زیر تعریف می‌شود:

$$\frac{1}{2S}\sum_{i=1}^{S} \|x_i - \sum_{j=1}^{|\mathcal{R}|} a_j \cdot x_j\|_2^2 + \lambda_s \cdot \|\mathcal{A}\|_1$$

x_i بردار ویژگی نظیر با یکی از جملات کاندید است و R تعداد جملاتی را نشان می‌دهد که در خلاصه‌سازی انتخاب می‌شوند. a_j نیز برابر امتیاز نظیر با جمله‌ی x_j است. الگوریتم اصلی با رویکرد مشتق‌گیری از مقدار خطا نسبت به هر جمله شروع می‌کند و سپس بیشترین مشتق ایجاد شده را تعیین کرده و مقدار a نظیر با آن المان در بردار A به روزرسانی می‌شود. نحوه‌ی به روزکردن این مقدار در شبه‌کد زیر مشخص می‌شود.

Initialize each $a \in \mathcal{A} \rightarrow 0$; $k \rightarrow 0$; $ite \rightarrow 0$;
Transfer sentences to basis vectors $\mathcal{X} = \{x_1, x_2, ..., x_S\}$;
while $ite < T$ **do**
  Reconstructing $\overline{x} = \sum_{i \in S} a_i^{ite} x_i$;
  Take partial derivatives:

$$\frac{\partial J}{\partial a_i} = \frac{1}{S}\sum_{j \in S} sim_j (x_j - \bar{x})^T \overrightarrow{x_i} - \frac{1}{S'}\sum_{j \in S} sim'_j (x'_j - \bar{x})^T \overrightarrow{x_i};$$

  Select the coordinate with maximum partial derivative:
  $i' = \arg\max_{i \in S} \left|\frac{\partial J}{\partial a_i}\right|$;
  Update the coordinate by soft-thresholding:
  $a_{i'}^{ite+1} = S_\lambda(a_{i'}^{ite} - \eta \frac{\partial J}{\partial a_{i'}})$;
  where $S_\lambda : a \rightarrow sign(a)\max(a - \lambda_s, 0)$;
  if $J_{\mathcal{A}^{ite+1}} - J_{\mathcal{A}^{ite}} < \gamma$ **then**
    break;
  end
  $ite \rightarrow ite + 1$;
end

پارامترهایی‌که در الگوریتم فوق مورد استفاده است به شرح زیر هستند:

- فاکتور η به عنوان فاکتور نرخ یادگیری ماست و در انجام پروژه برابر ۱ در نظرگرفته شده است.
- S_λ تابعی است که به صورت مطرح شده در الگوریتم یعنی $a \rightarrow sign(a)\max(a - \lambda_s, 0)$ عمل می‌کند.

- تعداد تکرار های مجاز برای تعیین بردار A و در نتیجه انتخاب نهایی جملات، با T نشان داده شده و مقدار آن در پروژه‌ی پیاده‌شده برابر ۵۰۰ درنظرگرفته شده است.
- همچنین مقداری که تابع خطا در هر تکرار می‌تواند بدتر شود اما اجرا همچنان ادامه پیدا کند با Υ نشان داده شده که مقدار آن را در اجرای این پروژه برابر ۰.۰۰۰۱ در نظرگرفته شده است.

همچنین قسمت‌هایی که مربوط به قسمت *question answering* بوده است در پیاده‌سازی آورده نشده است. در انتهای کد، تابع خطا براساس بردار A در تکرار جاری و تکرار بعدی بررسی می شود و برای این کار نیاز است تا براساس بردار A جملات خلاصه را در هر تکرار مشخص کنیم. برای این کار اگر فرض کنیم تعداد جملات منتخب را عدد ۷ تعریف کرده باشیم، ۷ مقدار بیشتر را در بردار A مشخص کرده و نمایه‌ی نظیر آن‌ها را یافته و پس از آن جملات نظیر با همان نمایه‌ها انتخاب می‌شوند تا پس از آن در محاسبه‌ی تابع خطا مورد استفاده قرار گیرند.

نکته: به نظر من، در انتهای این شبه کد نیاز است تا رابطه‌ی $loss1 - loss2 < -1 * \Upsilon$ به جای رابطه‌ی $loss1 - loss2 < \Upsilon$ تعریف شود چرا که طبق تعریف تابع خطا باید میزان خطا کمتر شود. البته هر دو مورد در حالت *single* تست شده و به نظر نمی‌آمد تفاوت قابل‌توجهی در نتایج آنها با تغییر در این شرط دیده نشود.

هم‌چنین ساخت بردارهای هر جمله در حالت *TF* به کمک *countVectorizer* صورت می‌گیرد که از کلاس‌های کتابخانه‌ی *scikit learn* است.

```
vectorizer = CountVectorizer (ngram_range= (1, 1), token_pattern=r'\b\w+\b',
min_df=1, max_df=0.5, lowercase=True)
```

در رابطه‌ی بالا *countVectorizer* را به گونه‌ای ست می‌کنیم که *ngram* های یکی را برای ما شمارش کند و اگر کلمه‌ای در حداقل یک جمله دیده شود آن را از جزو ویژگی‌ها می‌آوریم (*min_df=1*) و اگر کلمه‌ای در بیش از ۵۰ درصد جملات وجود دارد آن را از ویژگی‌های بردار ویژگی حذف می‌کنیم (*max_df=0.5*). برای ساخت بردار ویژگی کلمات از دستور *vectorizer.fit_transform(sentences)* استفاده می‌شود و برای ساخت بردار ویژگی داده‌های تست براساس بردار ویژگی‌های تشخیص داده شده توسط داده‌ی آموزش، از دستور *vectorizer.transform(sentences)* استفاده می‌کنیم. برای ساخت بردار جملات براساس *w2v* از مدل وردتووک همشهری استفاده شده که به علت حجم بالا در محتویات پوشه‌ی این پروژه قرار ندارد.

تعداد جملات خلاصه برای حالت *singleDoc* برابر ۷ و برای حالت *multiDoc* برابر ۳۰ است.

در ادامه نتایج پروژه اجرا شده آورده می‌شود:

مقادیر ستونی به ترتیب برابر *precision* و *recall* و *F1_score* است.

در حالت *tf_singleDoc* :

Rouge1 = [0.46012226787494653, 0.5869253903081032, 0.480097969173178]

Rouge2 = [0.34032208488791094, 0.4647129268407671, 0.35536324554634896]

در حالت *w2v_singleDoc* :

Rouge1 = [0.4640479341796874, 0.6359646180099159, 0.5032478754579688]

Rouge2 = [0.3533086425841196, 0.5187993003494382, 0.384511215136727 04]

در حالت *tf_multiDoc*:

Rouge1 = [0.2650202970253799, 0.3380405122897867, 0.2810291868560773]

Rouge2 = [0.11763127025028339, 0.15830313338648527, 0.12284667608471095]

در حالت *w2v_multiDoc* :

Rouge1 = [0.3192376042864618, 0.3164093483027287, 0.296398596783273]

Rouge2 = [0.14299051235321394, 0.1434192440301684, 0.130297244326151 67]]

Word2vec باعث بهبود در انتخاب جملات خلاصه می‌شود. البته زمان اجرای آن نیز در تولید خلاصه بیشتر می‌شود. همچنین از مقادیر به دست آمده مشخص است که در حالت چندسندی به نسبت تک سندی افت محسوسی در دقت و *recall* وجود دارد.

فایل نهایی که باید در ارزیابی حالت *w2v_multiDoc* مورداستفاده قرار می‌گرفت به دلیل بالابودن زمان اجرای آن، کامل نیست و صرفا به ازای ۷ مجموعه‌ی سندی خلاصه تولید شده و در ارزیابی مورد استفاده قرار گرفته است.