بسم الله الرحمن الرحیم

پروژه پردازش متن

برای محصولات دیجیکالا

نسخه شماره ۳

تیم ارائه دهنده: داده کاوانِ فونیکس

ناظر و سرپرست تیم: دکتر علیرضا وفایی صدر

درخواست دهنده: ستاد علوم شناختی IPM

زمستان ۱۳۹۸

.

فهرست مطالب

تعریف پروژه

در ادامه فاز دوم و بحث های صورت گرفته، به ادامه کار تولید متن با لایک های بالا پرداختیم.

- تولید متن با همان مدل قبلی classifier ولی با نگهداری وزن ها و استفاده از مدل pre-trained

- تولید متن با GAN و تبدیل آن به کامنت هایی با لایک بالا

تولید متن با معماری pre-trained قبلی

برای اینکه کامنت های تولید کنیم که classifier آن ها را در کلاس با لایک بالا قرار دهد، از همان مدل قبلی که دقت ۷۵٪ برای چهار کلاسه به ما داد استفاده کردیم. ولی این بار به جای چهار کلاس، داده های را به دو کلاس طبقه بندی کردیم. در لایه آخر به جای Dense(4) یک لایه Dense(2) گذاشتیم و همینطور ستون تارگت را به دو کلاس ۰ و ۱ تبدیل کردیم. کلاس ۰ یعنی like ، و کلاس ۱ به معنای verylike است. و داده های را روی این مدل دو کلاسه از اول train کردیم. شکل زیر معماری مدل classifier دو کلاسه را نشان می دهد:

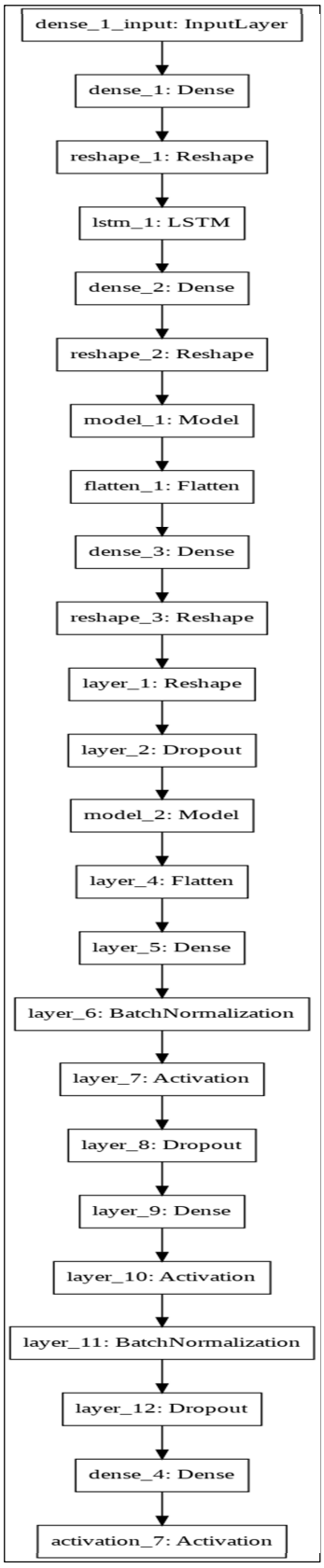
```
filter_sizes = [3, 4, 8]
def convolution():
    inn = Input(shape = (20, 20, 1))
    convolutions = []

    conv_1 = Conv2D(filters = 150, kernel_size = (3, 20), strides = 1, padding =
"valid",name='layer_31',trainable=True)(inn)
    nonlinearity_1 = Activation('relu',name='layer_32',trainable=True)(conv_1)
    maxpool_1 = MaxPooling2D(pool_size = (20- 3+ 1, 1), padding =
"valid",name='layer_33',trainable=True)(nonlinearity_1)
    convolutions.append(maxpool_1)

    conv_2 = Conv2D(filters = 150, kernel_size = (4, 20), strides = 1, padding =
"valid",name='layer_34',trainable=True)(inn)
    nonlinearity_2 = Activation('relu',name='layer_35',trainable=True)(conv_2)
    maxpool_2 = MaxPooling2D(pool_size = (20- 4+ 1, 1), padding =
"valid",name='layer_36',trainable=True)(nonlinearity_2)
    convolutions.append(maxpool_2)


    conv_3 = Conv2D(filters = 150, kernel_size = (8, 20), strides = 1, padding =
"valid",name='layer_37',trainable=True)(inn)
    nonlinearity_3 = Activation('relu',name='layer_38',trainable=True)(conv_3)
    maxpool_3 = MaxPooling2D(pool_size = (20- 8+ 1, 1), padding =
"valid",name='layer_39',trainable=True)(nonlinearity_3)
    convolutions.append(maxpool_3)

    outt = concatenate(convolutions)
    model = Model(inputs = inn, outputs = outt)
    return model


model = Sequential()
model.add(Embedding(input_dim =53019, output_dim = 20, input_length = 20,name='embed',trainable=True))
model.add(Reshape((20, 20, 1), input_shape = (20,20 ),name='layer_1',trainable=True))
model.add(Dropout(0.5 ,name='layer_2',trainable=True))
# call convolution method defined above
model.add(convolution())
model.add(Flatten(name='layer_4',trainable=True))
model.add(Dense(500, activation = 'relu', kernel_initializer='he_normal'
,kernel_regularizer=regularizers.l1(0.01),bias_regularizer=regularizers.l2(0.01),name='layer_5',trainab
le=True))
model.add(BatchNormalization(name='layer_6',trainable=True))
model.add(Activation('relu',name='layer_7',trainable=True))
model.add(Dropout(0.5,name='layer_8',trainable=True))
model.add(Dense(50, activation = 'relu', kernel_initializer='he_normal'
,kernel_regularizer=regularizers.l1(0.01),bias_regularizer=regularizers.l2(0.01),name='layer_9',trainab
le=True))
model.add(Activation('relu',name='layer_10',trainable=True))
model.add(BatchNormalization(name='layer_11',trainable=True))
# model.add(Dropout(0.5 ,name='layer_12',trainable=True))
model.add(Dense(2))
model.add(Activation('softmax'))

adam =Adam(lr = 0.000001)

model.compile(loss='binary_crossentropy', optimizer=adam , metrics=['accuracy'])
```

مدل bi-classification را که ترین کردیم روی داده های ترین دقت ۵۰٪ و روی داده ولیدیشن دقت بالای ۹۰٪ داد. سپس برای تولید نویز، لایه embedding همین معماری را برداشتیم و قبل از آن یکسری لایه trainable قرار دادیم که کار generator برای تولید داده های نویز را انجام دهد و تمامی لایه های معماری classifier قبلی را Freeze کرده، و وزن های لایه های classifier را save کردیم و لایه های آن را از حالت trainable خارج کردیم تا بتوان داده های نویز را با همان دقت قبل به کلاس verylike بیاندازد.

داده های نویز از لایه های trainable عبور کرده و تولید می شود و سپس وارد classifier فریز شده با وزن های قبلی، می شود که حکم discriminator را دارد که تشخیص دهد این داده های نویز وارد کلاس verylike شده اند یا نه. معماری کلی تولید داده و تبدیل آن به داده هایی با کامنت بالا در شکل زیر دیده می شود:

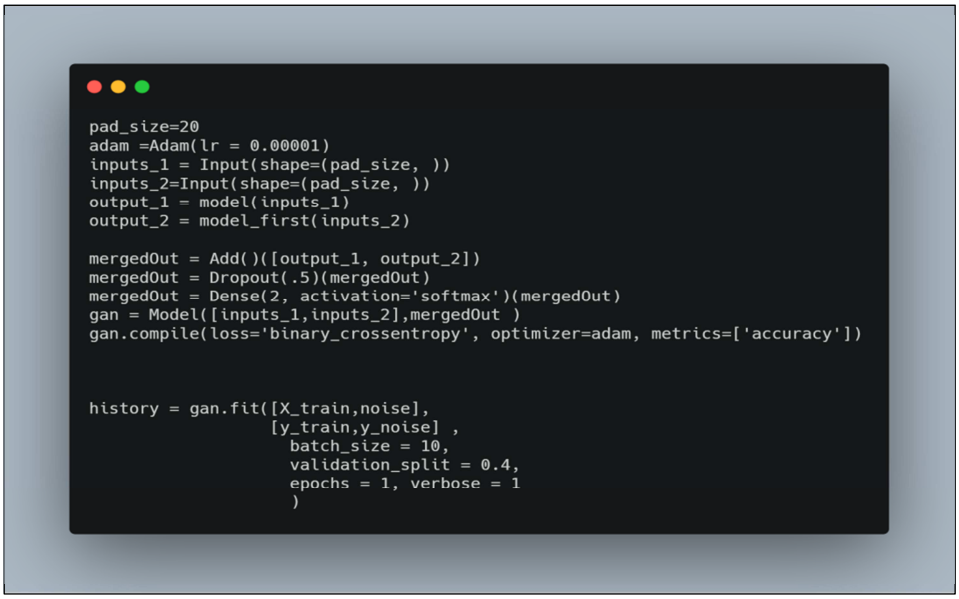
```
filter_sizes = [3, 4, 8]
def convolutionn():
    inn = Input(shape = (20, 20, 1))
    convolutions = []
    # we conduct three convolutions & poolings then concatenate them.
    for  fs in filter_sizes:
        conv = Conv2D(filters = 150, kernel_size = (fs, 20), strides = 1, padding = "valid")(inn)
        nonlinearity = Activation('relu')(conv)
        maxpool = MaxPooling2D(pool_size = (20- fs + 1, 1), padding = "valid")(nonlinearity)
        convolutions.append(maxpool)

    outt = concatenate(convolutions)
    model = Model(inputs = inn, outputs = outt)

    return model


model_first = Sequential()
model_first.add(Dense(36,input_shape=(20,)))
model_first.add(Reshape((6, 6),input_shape=(36,1)))
model_first.add(LSTM(256, input_shape=(6,6)))
model_first.add(Dense(400, activation='softmax'))
model_first.add(Reshape((20,20,1),input_shape=(400,1)))
model_first.add(convolutionn())

model_first.add(Flatten())
model_first.add(Dense(20*20, activation='softmax'))
model_first.add(Reshape((20, 20),input_shape=(400,1)))

model_first.add(Reshape((20, 20, 1), input_shape = (20,20 ),name='layer_1',trainable=False))
model_first.add(Dropout(0.5 ,name='layer_2',trainable=False))

# call convolution method defined above
 #start of freez layer

model_first.add(convolution())

model_first.add(Flatten(name='layer_4',trainable=False))
model_first.add(Dense(500, activation = 'relu', kernel_initializer='he_normal'
,kernel_regularizer=regularizers.l1(0.01),bias_regularizer=regularizers.l2(0.01)
,name='layer_5',trainable=False))
model_first.add(BatchNormalization(name='layer_6',trainable=False))
model_first.add(Activation('relu' ,  name='layer_7',trainable=False))
model_first.add(Dropout(0.5 , name='layer_8',trainable=False))
model_first.add(Dense(50, activation = 'relu', kernel_initializer='he_normal'
,kernel_regularizer=regularizers.l1(0.01),bias_regularizer=regularizers.l2(0.01),
name='layer_9',trainable=False))
model_first.add(Activation('relu' , name='layer_10',trainable=False))
model_first.add(BatchNormalization(name='layer_11',trainable=False))
model_first.add(Dropout(0.5 , name='layer_12',trainable=False))
model_first.add(Dense(2))
model_first.add(Activation('softmax'))

adam =Adam(lr = 0.00001)


model_first.compile(loss='binary_crossentropy', optimizer=adam , metrics=['accuracy'])
```

شکل معماری مدلی که داده های نویز را در کلاس verylike می اندازد را در زیر مشاهده می کنید:



معماری کلی برای تبدیل داده های نویز به کلاس لایک بالا دقتی بالای ۹۷٪ داده است و این به خاطر این است که چون از مدل قبلی که pre-trained شده استفاده کردیم و داده های واقعی قبلا توسط مدل دیده شده بود، به این خاطر دقت بالایی به دست آورده ایم.

در نهایت برای اینکه مدل بهینه شود و نزدیکی داده های نویز به داده های واقعی با سنجیده شود از یک مدل دیگر استفاده کدیم که ورودی داده واقعی و داده نویز تولید شده را می گیرد و با هم مقایسه می کند:



```
pad_size=20
adam =Adam(lr = 0.00001)
inputs_1 = Input(shape=(pad_size, ))
inputs_2=Input(shape=(pad_size, ))
output_1 = model(inputs_1)
output_2 = model_first(inputs_2)

mergedOut = Add()([output_1, output_2])
mergedOut = Dropout(.5)(mergedOut)
mergedOut = Dense(2, activation='softmax')(mergedOut)
gan = Model([inputs_1,inputs_2],mergedOut )
gan.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])



history = gan.fit([X_train,noise],
                  [y_train,y_noise] ,
                    batch_size = 10,
                    validation_split = 0.4,
                    epochs = 1, verbose = 1
                    )
```

مدل به اسم gan ، داده های نویز را از مدل model_first و داده های واقعی و فعلی را از مدل قبلی دو کلاسه pre-trained شده به نام model دریافت کرده و آن را train می کند تا تفاوت بین این دو را بفهمد همانطور که در شکل بالا مشاهده می کنید.

دقت مدل gan روی ترین ۷۷٪ و روی داده های ولیدیشن به ۹۹٪ رسید.

برای تولید داده های نویز با کلاس، داده های واقعی را گرفته و در ستون کامنت هایی که کلاس verylike داشتند (ستون تارگت مقدار ۱ داشت) بعد از Vectorize کردن، به جای کامنت های واقعی، مقادیر رندوم نویز در همان بازه ی مقادیر vectorize شده برای متون کامنت، قرار دادیم.

معماری مربوط به تولید نویز و تبدیل این داده های به کامنت هایی با کلاس verylike در فایل model_final.ipybn قرار دارد که توسط خانم رقیه فرجی انجام شده است.

مسیر فایل:

Report2-> roghaye_faraji

تولید متن با معماری GAN

از آنجا که ما قبلا سه نوع معماری مختلف برای تولید کامنت با لایک بالا بوسیله معماری GAN داشتیم، تصمیم گرفتیم علاوه بر مدل تولید شده فعلی، از همان GAN قبلی نیز استفاده کنیم ولی این بار به جای دادن کامنت هایی با لایک معمولی در generator و دادن داده هایی با لایک بالا به discriminator ، به مدل generator داده های نویز را وارد کنیم.

از معاری خانم نفریه استفاده کردیم و مدل train شد و داده هایی با لایک بالا را تولید کرد.

در معماری خانم نفریه برای discriminator ، از مدل FC یا Fully connected ها استفاده کردند و

برای generator از دو معماری استفاده کرده اند، یکی FC با LSTM و یکی FC بدون LSTM

که هر دو جواب میده ولی در معماری generator ای که FC و LSTM هر دو وجود دارند لاس dis و gen به همگرایی رسیده است و جواب بهتری گرفته اند.

شکل معماری و فایل های GAN را می توانید در مسیری که گفته شده است ببینید.

Report2-> Zahra_Nafarieh-> Nafarieh_GAN.ipynb

همچنین آقای غفوریان در ابتدا یک کلاسیفایر برای discriminator تولید کرد و داده ها را دو کلاسه کرد ولی دقت کلاسیفایر روی ترین بیشتر از ۵۰٪ نرسید و برای پی بردن به این موضوع آقای غفوریان کلمات مشترک بین دو کلاس like و verylike را بررسی کرد و متوجه شد که این دو کلاس کلمات و کامنت های مشترک بسیاری دارند و به همین دلیل نزدیکی و شباهت کامنت ها کلاسیفایر نمی تواند دو کلاس را از هم به خوبی تشخیص داده و به نوعی حالت تصادفی عمل می کند. از دیگر دلایلی که آقای غفوریان به آن پی بردند این بود که چون ما دسته محصولات مختلفی داریم و کامنت ها مختص به یک سری محصولات خاص نیستند، کلمات بسیار زیاد و مشترکی در هر دسته بندی محصولات وجود دارد و این باعث شباهت و نزدیکی دو کلاس به هم می شود و کار کلاسیفایر دو کلاسه برای تمایز بین این دو بسیار کلاس بسیار سخت می شود.

فایل مربوط به این بررسی در مسیر report2-> VahidGhafourian قرار دارد.

برگرداندن داده های تولید شده به جملات

قرار شد بعد از تولید داده هایی با کامنت بالا، آن داده های را تبدیل به جملات کنیم و ببینیم که مدل پیشنهادی، چه جملات و کامنت هایی را تولید کرده است.

در مدل GAN خانم نفریه موفق به انجام اینکار شدیم و توانستیم خروجی GAN را برگردونیم به کلمات ولی با توجه به اینکه RAM کافی نداریم قسمتی از کار به تعویق افتاد، به همین علت جملات با مفهومی تولید نکردیم.

در معماری خانم فرجی، برای این کار، خروجی آخرین لایه قبل از لایه freeze شده در مدل model_first را گرفتیم و آن را در Transpose ماتریس embedding که وزن های آن را نگهداشته بودیم ضرب کردیم و سپس قرار شد از Inverse دیکشنری word2vec که کلمات را به اعداد تبدیل کرده بودیم استفاده کنیم و بتوانیم اعداد تولید شده را به جملات تبدیل کنیم. این کار توسط آرمیتا رضوی انجام شد که متاسفانه به هنگام عملیات ضرب ماتریسی با مشکل پر شدن RAM در Google Colab مواجه شدیم و علی رغم تلاش چندباره موفق نشدیم این کار را انجام دهیم و داده های تولید شده را توسط به این و معماری، به متن بازگردانیم.

جمع بندی و تحقیقات آتی

در این فاز از پروژه (تولید کامنت هایی که لایک بالا داشته باشند) چندین مطلب و مسئله وجود دارد که مایل هستیم بیان کنیم:

برای تولید متن و حتی عکس توسط هر نوع معماری آن هم به صورت بهینه، نیاز به حجم زیادی از داده وجود دارد و از آنجا که تعداد کل داده های این پروژه ۱۰۰۰۰۰ کامنت بود، طبیعتا معماری نمی تواند خروجی مطلوبی را داشته باشد.

همچنین داده ای که در اختیار داشتیم دارای ستون های هدف یا تارگت های بسیار نامتوازن بودند و پیدا کردن یک threshold برای کلاسه بندی لایک ها و دیسلایک ها بسیار مهم است و به دلیل نامتوازن بودن و شباهت بسیار زیاد بین نمودارهای توزیع دو ستون لایک و دیسلایک و شباهت بسیار این دو ستون تارگت به همدیگر، کار پیدا کردن این آستانه برای کلاسه بندی سخت بود و اگر داده های متوازنی با توزیع نرمال تری وجود داشت قطعا کلاسیفایر، خیلی بهتر عمل کلاسه بندی را انجام می داد و شاید در معماری discriminator به دقت ۵۰٪ نمی رسیدیم و خروجی بهتری می گرفتیم. شاید به همین دلیل است که وقتی ستون تارگت چهار کلاسه را به دو کلاسه تبدیل کردیم، دقت از ۷۵٪ به ۵۰٪ رسید. زیرا با تلفیق هر دو مقدار like-dislike و like-verydislike به یک مقدار like و تلفیق دو مقدار verylike-dislike

و verylike-verydislike به Verylike در ستون هدف، جملات مشترک زیادی با هم یکی می شوند و کار کلاسیفایر برای جداسازی سخت خواهد شد مخصوصا به دلیل توزیع نا متقارن داده های ستون هدف.

مشکل دیگری که با آن مواجه بودیم نبود امکانات سخت افزاری لازم برای اجرای معماری بود. در این پروژه برای آموزش مدل نیاز به ران های طولانی با epoch های زیاد داشتیم که متاسفانه با مشکلات پر شدن RAM و کند بودن برخوردیم که زمان زیادی را برای این موضوع از دست دادیم.

همچنین کلا ۶ روز زمان برای کار preprocessing و data cleaning داشتیم و قطعا این بخش از کار مهمترین بخش از کار NLP می باشد که باید بیشترین زمان به این بخش اختصاص داده شود. در صورت زمان بیشتر و داده های بهتر برای پیش پردازش داده، به جواب های بهتر و بهینه تری خواهیم رسید.

امید است در آینده با توجه به حجم کاری که درخواست می شود، داده های مناسب با حجم زیاد و امکانات سروری بهتری در اختیار ما قرار گیرد تا بتوانیم تمامی مدل ها را بهینه کرده و خروجی کار را بهبود ببخشیم.

در این فاز از پروژه با توجه به مواردی که بالا ذکر شد و کمبود وقت، هدف اولِ تیم، بیشتر روی تولید معماری صحیح و اصولی و دریافت یک خروجی اولیه برای کار بود و نه خروجی نهایی و بهینه، و صد البته با داشتن زمان بیشتر و سرورهای قوی، خروجی کار همان خواهد شد که مورد انتظار ستاد و خود ما می باشد.

با داشتن سرورهای قوی تر، کار بازگرداندن داده های تولید شده با لایک بالا به جملات را ادامه می دهیم و معماری و مدل های خود را بهبود می بخشیم و مشکل مربوط به کلاسیفایر دو کلاسه را حل خواهیم کرد و روی روش های مختلف تولید متن از تغییر کامنت های آماده با لایک کم به کامنت هایی با لایک بالا کار می کنیم.

یعنی معماری بتواند کامنت های موجود را طوری تغییر دهد که از کلاس لایک کم به کلاس لایک بالا تبدیل شوند. نه اینکه داده های نویزی بسازیم که لایک بالا بخورند. در واقع تلاش می کنیم کار style transformation را برای متون انجام دهیم.

تیم فونیکس تمام توان و تلاش خود را به کار برد تا بهترین نتیجه را در حد امکان و با توجه به زمان تعیین شده در اختیار ستاد قرار دهد و امید است مورد رضایت واقع شده باشد.

کلیه فایل های مربوط به این گزارش نسخه ۳ در گیت هاب زیر و در فولدر Report2 موجود است.

https://github.com/phoenix-dataminers/Digikala2