本文介绍了三种用于表征句子的无监督深度学习方法：自编码器、语言模型和 Skip-Thought 向量模型，并与基线模型 Average Word2Vec 进行了对比。

近年来，由于用连续向量表示词语（而不是用稀疏的 one-hot 编码向量（Word2Vec））技术的发展，自然语言处理领域的性能获得了重大提升。

Word2Vec 示例

尽管 Word2Vec 性能不错，并且创建了很不错的语义，例如 King – Man + Woman = Queen，但是我们有时候并不在意单词的表征，而是句子的表征。

本文将介绍几个用于句子表征的无监督深度学习方法，并分享相关代码。我们将展示这些方法在特定文本分类任务中作为预处理步骤的效果。

分类任务

用来展示不同句子表征方法的数据基于从万维网抓取的 10000 篇新闻类文章。分类任务是将每篇文章归类为 10 个可能的主题之一（数据具备主题标签，所以这是一个有监督的任务）。为了便于演示，我会使用一个 logistic 回归模型，每次使用不同的预处理表征方法处理文章标题。

基线模型——Average Word2Vec

我们从一个简单的基线模型开始。我们会通过对标题单词的 Word2Vec 表征求平均来表征文章标题。正如之前提及的，Word2Vec 是一种将单词表征为向量的机器学习方法。Word2Vec 模型是通过使用浅层神经网络来预测与目标词接近的单词来训练的。你可以阅读更多内容来了解这个算法是如何运行的。

我们可以使用 Gensim 训练我们自己的 Word2Vec 模型，但是在这个例子中我们会使用一个 Google 预训练 Word2Vec 模型，它基于 Google 的新闻数据而建立。在将每一个单词表征为向量后，我们会将一个句子（文章标题）表征为其单词（向量）的均值，然后运行 logistic 回归对文章进行分类。

```
#load data and Word2vec model

df = pd.read_csv("news_dataset.csv")
```

```
data = df[['body','headline','category']]

w2v = gensim.models.KeyedVectors.load_word2vec_format('/GoogleNews-vectors-negative300.bin', binary=True)

#Build X and Y

x = np.random.rand(len(data),300)

for i in range(len(data)):

 k = 0

 non = 0

 values = np.zeros(300)

 for j in data['headline'].iloc[i].split(' '):

 if j in w2v:

 values+= w2v[j]

 k+=1

 if k >

 0:

 x[i,:]=values/k

 else: non+=1

y = LabelEncoder().fit_transform(data['category'].values)

msk = np.random.rand(len(data))

X_train,y_train,X_test,y_test = x[msk],y[msk],x[~msk],y[~msk]
```

#Train the model

lr = LogisticRegression().fit(X_train,y_train)

lr.score(X_test,y_test)

我们的基线 average Word2Vec 模型达到了 68%
的准确率。这很不错了，那么让我们来看一看能不能做得更好。

average Word2Vec 方法有两个弱点：它是词袋模型（bag-of-words model），与单词顺序无关，所有单词都具备相同的权重。为了进行句子表征，我们将在下面的方法中使用 RNN 架构解决这些问题。

自编码器

自编码器是一种无监督深度学习模型，它试图将自己的输入复制到输出。自编码器的技巧在于中间隐藏层的维度要低于输入数据的维度。所以这种神经网络必须以一种聪明、紧凑的方式来表征输入，以完成成功的重建。在很多情况下，使用自编码器进行特征提取被证明是非常有效的。

我们的自编码器是一个简单的序列到序列结构，由一个输入层、一个嵌入层、一个 LSTM 层，以及一个 softmax
层组成。整个结构的输入和输出都是标题，我们将使用 LSTM
的输出来表征标题。在得到自编码器的表征之后，我们将使用 logistics 回归来预测类别。为了得到更多的数据，我们会使用文章中所有句子来训练自编码器，而不是仅仅使用文章标题。

#parse all sentences

sentenses = []

for i in data['body'].values:

 for j in nltk.sent_tokenize(i):

 sentenses.append(j)

#preprocess for keras

```
num_words=2000

maxlen=20

tokenizer = Tokenizer(num_words = num_words, split=' ')

tokenizer.fit_on_texts(sentenses)

seqs = tokenizer.texts_to_sequences(sentenses)

pad_seqs = []

for i in seqs:

 if len(i)>

4:

 pad_seqs.append(i)

pad_seqs = pad_sequences(pad_seqs,maxlen)

#The model

embed_dim = 150

latent_dim = 128

batch_size = 64

#### Encoder Model ####

encoder_inputs = Input(shape=(maxlen,), name='Encoder-Input')

emb_layer = Embedding(num_words, embed_dim,input_length = maxlen,
name='Body-Word-Embedding', mask_zero=False)

# Word embeding for encoder (ex: Issue Body)
```

```
x = emb_layer(encoder_inputs)

state_h = GRU(latent_dim, name='Encoder-Last-GRU')(x)

encoder_model = Model(inputs=encoder_inputs, outputs=state_h,
name='Encoder-Model')

seq2seq_encoder_out = encoder_model(encoder_inputs)

#### Decoder Model ####

decoded = RepeatVector(maxlen)(seq2seq_encoder_out)

decoder_gru = GRU(latent_dim, return_sequences=True, name='Decoder-
GRU-before')

decoder_gru_output = decoder_gru(decoded)

decoder_dense = Dense(num_words, activation='softmax', name='Final-
Output-Dense-before')

decoder_outputs = decoder_dense(decoder_gru_output)

#### Seq2Seq Model ####

#seq2seq_decoder_out = decoder_model([decoder_inputs,
seq2seq_encoder_out])

seq2seq_Model = Model(encoder_inputs,decoder_outputs )

seq2seq_Model.compile(optimizer=optimizers.Nadam(lr=0.001),
loss='sparse_categorical_crossentropy')

history = seq2seq_Model.fit(pad_seqs, np.expand_dims(pad_seqs, -1),

 batch_size=batch_size,

 epochs=5,
```

validation_split=0.12)

#Feature extraction

headlines = tokenizer.texts_to_sequences(data['headline'].values)

headlines = pad_sequences(headlines,maxlen=maxlen)x = encoder_model.predict(headlines)

#classifier

X_train,y_train,X_test,y_test = x[msk],y[msk],x[~msk],y[~msk]

lr = LogisticRegression().fit(X_train,y_train)

lr.score(X_test,y_test)

我们实现了 60%
的准确率，比基线模型要差一些。我们可能通过优化超参数、增加训练 epoch 数量或者在更多的数据上训练模型，来改进该分数。

语言模型

我们的第二个方法是训练语言模型来表征句子。语言模型描述的是某种语言中一段文本存在的概率。例如，「我喜欢吃香蕉」（I like eating bananas）这个句子会比「我喜欢吃卷积」（I like eating convolutions）这个句子具备更高的存在概率。我们通过分割 n 个单词组成的窗口以及预测文本中的下一个单词来训练语言模型。你可以在这里了解到更多基于 RNN
的语言模型的内容。通过构建语言模型，我们理解了「新闻英语」（journalistic English）是如何建立的，并且模型应该聚焦于重要的单词及其表征。

我们的架构和自编码器的架构是类似的，但是我们只预测一个单词，而不是一个单词序列。输入将包含由新闻文章中的 20 个单词组成的窗口，标签是第 21 个单词。在训练完语言模型之后，我们将从 LSTM
的输出隐藏状态中得到标题表征，然后运行 logistics 回归模型来预测类别。

#Building X and Y

```
num_words=2000

maxlen=20

tokenizer = Tokenizer(num_words = num_words, split=' ')

tokenizer.fit_on_texts(df['body'].values)

seqs = tokenizer.texts_to_sequences(df['body'].values)

seq = []

for i in seqs:

 seq+=i

X = []

Y = []

for i in tqdm(range(len(seq)-maxlen-1)):

 X.append(seq[i:i+maxlen])

 Y.append(seq[i+maxlen+1])

X = pd.DataFrame(X)

Y = pd.DataFrame(Y)

Y[0]=Y[0].astype('category')

Y =pd.get_dummies(Y)

#Buidling the network

embed_dim = 150

lstm_out = 128
```

```
batch_size= 128

model = Sequential()

model.add(Embedding(num_words, embed_dim,input_length = maxlen))

model.add(Bidirectional(LSTM(lstm_out)))

model.add(Dense(Y.shape[1],activation='softmax'))

adam = Adam(lr=0.001, beta_1=0.7, beta_2=0.99, epsilon=None,
decay=0.0, amsgrad=False)

model.compile(loss = 'categorical_crossentropy', optimizer=adam)

model.summary()

print('fit')

model.fit(X, Y, batch_size =batch_size,validation_split=0.1, epochs = 5,
verbose = 1)

#Feature extraction

headlines = tokenizer.texts_to_sequences(data['headline'].values)

headlines = pad_sequences(headlines,maxlen=maxlen)

inp = model.input

outputs = [model.layers[1].output]

functor = K.function([inp]+ [K.learning_phase()], outputs )

x = functor([headlines, 1.])[0]

#classifier

X_train,y_train,X_test,y_test = x[msk],y[msk],x[~msk],y[~msk]
```

lr = LogisticRegression().fit(X_train,y_train)

lr.score(X_test,y_test)

这一次我们得到了 72%
的准确率，要比基线模型好一些，那我们能否让它变得更好呢？

Skip-Thought 向量模型

在 2015 年关于 skip-thought 的论文《Skip-Thought
Vectors》中，作者从语言模型中获得了同样的直觉知识。然而，在 skip-thought 中，我们并没有预测下一个单词，而是预测之前和之后的句子。这给模型关于句子的更多语境，所以，我们可以构建更好的句子表征。您可以阅读这篇博客），了解关于这个模型的更多信息。

skip-thought 论文中的例子）

我们将构造一个类似于自编码器的序列到序列结构，但是它与自编码器有两个主要的区别。第一，我们有两个 LSTM
输出层：一个用于之前的句子，一个用于下一个句子；第二，我们会在输出 LSTM 中使用教师强迫（teacher forcing）。这意味着我们不仅仅给输出 LSTM 提供了之前的隐藏状态，还提供了实际的前一个单词（可在上图和输出最后一行中查看输入）。

#Build x and y

num_words=2000

maxlen=20

tokenizer = Tokenizer(num_words = num_words, split=' ')

tokenizer.fit_on_texts(sentenses)

seqs = tokenizer.texts_to_sequences(sentenses)

pad_seqs = pad_sequences(seqs,maxlen)

```
x_skip = []

y_before = []

y_after = []

for i in tqdm(range(1,len(seqs)-1)):

 if len(seqs[i])>

4:

 x_skip.append(pad_seqs[i].tolist())

 y_before.append(pad_seqs[i-1].tolist())

 y_after.append(pad_seqs[i+1].tolist())

x_before = np.matrix([[0]+i[:-1] for i in y_before])

x_after =np.matrix([[0]+i[:-1] for i in y_after])

x_skip = np.matrix(x_skip)

y_before = np.matrix(y_before)

y_after = np.matrix(y_after)

#Building the model

embed_dim = 150

latent_dim = 128

batch_size = 64

#### Encoder Model ####

encoder_inputs = Input(shape=(maxlen,), name='Encoder-Input')
```

```
emb_layer = Embedding(num_words, embed_dim,input_length = maxlen, name='Body-Word-Embedding', mask_zero=False)

x = emb_layer(encoder_inputs)

_, state_h = GRU(latent_dim, return_state=True, name='Encoder-Last-GRU')(x)

encoder_model = Model(inputs=encoder_inputs, outputs=state_h, name='Encoder-Model')

seq2seq_encoder_out = encoder_model(encoder_inputs)

#### Decoder Model ####

decoder_inputs_before = Input(shape=(None,), name='Decoder-Input-before') # for teacher forcing

dec_emb_before = emb_layer(decoder_inputs_before)

decoder_gru_before = GRU(latent_dim, return_state=True, return_sequences=True, name='Decoder-GRU-before')

decoder_gru_output_before, _ = decoder_gru_before(dec_emb_before, initial_state=seq2seq_encoder_out)

decoder_dense_before = Dense(num_words, activation='softmax', name='Final-Output-Dense-before')

decoder_outputs_before = decoder_dense_before(decoder_gru_output_before)

decoder_inputs_after = Input(shape=(None,), name='Decoder-Input-after') # for teacher forcing

dec_emb_after = emb_layer(decoder_inputs_after)

decoder_gru_after = GRU(latent_dim, return_state=True, return_sequences=True, name='Decoder-GRU-after')
```

```
decoder_gru_output_after, _ = decoder_gru_after(dec_emb_after,
initial_state=seq2seq_encoder_out)

decoder_dense_after = Dense(num_words, activation='softmax',
name='Final-Output-Dense-after')

decoder_outputs_after = decoder_dense_after(decoder_gru_output_after)

#### Seq2Seq Model ####

seq2seq_Model = Model([encoder_inputs,
decoder_inputs_before,decoder_inputs_after],
[decoder_outputs_before,decoder_outputs_after])

seq2seq_Model.compile(optimizer=optimizers.Nadam(lr=0.001),
loss='sparse_categorical_crossentropy')

seq2seq_Model.summary()

history = seq2seq_Model.fit([x_skip,x_before, x_after],
[np.expand_dims(y_before, -1),np.expand_dims(y_after, -1)],

 batch_size=batch_size,

 epochs=10,

 validation_split=0.12)

#Feature extraction

headlines = tokenizer.texts_to_sequences(data['headline'].values)

headlines = pad_sequences(headlines,maxlen=maxlen)x =
encoder_model.predict(headlines)

#classifier

X_train,y_train,X_test,y_test = x[msk],y[msk],x[~msk],y[~msk]
```

lr = LogisticRegression().fit(X_train,y_train)

lr.score(X_test,y_test)

这一次我们达到了 74% 的准确率。这是目前得到的最佳准确率。

总结

本文中，我们介绍了三个使用 RNN 创建句子向量表征的无监督方法，并且在解决一个监督任务的过程中展现了它们的效率。自编码器的结果比我们的基线模型要差一些（这可能是因为所用的数据集相对较小的缘故）。skip-thought 向量模型语言模型都利用语境来预测句子表征，并得到了最佳结果。

能够提升我们所展示的方法性能的可用方法有：调节超参数、训练更多 epoch 次数、使用预训练嵌入矩阵、改变神经网络架构等等。理论上，这些高级的调节工作或许能够在一定程度上改变结果。但是，我认为每一个预处理方法的基本直觉知识都能使用上述分享示例实现。