基于长短期记忆网络的重复问题检测

1. 背景

随着 Web2.0 技术和 HTML5 技术的兴起，当人们遇到问题时，越来越习惯于去在线问答网站（如百度知道，知乎，Quora 等）询问，一些专家也乐于在网站上解答问题。

在线问答网站上不可避免的会提交很多意义相同的问题，以 Quora 为例，如下两个问题实际上是一个问题：“What is the most populous state in the USA?” 和 “Which state in the United States has the most people?” 。重复问题会浪费回答问题的时间；对于网站来说，若重复问题过多，也会浪费大量的网络和存储资源。因此，重复问题检测是在线问答网站需要解决的关键问题之一。

有很多传统的自然语言处理方法可以被用来解决此问题：例如余弦相似度和 Jaccard 相似度[1]法可以求两段文本的相似度，若相似度超过阈值则认为是同一问题；另外使用特征工程的机器学习方法也可以从文本中挖掘大量的特征，然后利用挖掘到的特征使用分类方法求两个问题是否相同。相似度方法存在着阈值很难界定，导致查准和查全不能兼顾的问题；而机器学习方法依赖于复杂的特征工程，特征选择不合适会对分类结果有极大的影响。本文提出了一个基于长短期记忆网络（Long Short-Term Memory，LSTM）的模型用于重复问题分类，不需要进行复杂的特征抽取，且能得到较高的查准率和查全率。

2. 问题描述

在线问答网站 Quora 发布了相同问题检测的数据集[2]，该数据集的部分数据如图 1 所示。该数据集的每个条目由两个问题构成，id 和 qid1，qid2 分别是问题对，问题 1，问题 2 的 id，is_duplicate 是这两个问题是否是同一个问题的标签。

id	qid1	qid2	question1	question2	is_duplicate
447	895	896	What are natural numbers?	What is a least natural number?	0
1518	3037	3038	Which pizzas are the most popularly ordered pizzas on Domino's menu?	How many calories does a Dominos pizza have?	0
3272	6542	6543	How do you start a bakery?	How can one start a bakery business?	1
3362	6722	6723	Should I learn python or Java first?	If I had to choose between learning Java and Python, what should I choose to learn first?	1

图 1. Quora Duplicate Questions 数据集部分数据

需要设计一个模型，输入数据集中的一个条目，这个模型能给出条目中的两个问题是否是重复问题。

3. 模型

图 2 是模型的总体结构图，输入两个问题，给出这两个问题是否是重复问题。对于每个问题，其是一个由多个单词组成的序列，每个单词可以映射为一个向量（Each word is embedded into a d-dimensional space[3]），通过 embedding 层，这个单词序列被转换成一个二维矩阵；随后，利用 LSTM 对这个二维矩阵进行编码以提取出这个句子所表达的特征；然后将两个句子提取到的特征 r1 向量和 r2 向量求差、求点积，再连接成一个特征向量；将最终的向量输入多层感知机 MLP 中，求出分类结果。下面分各层详细阐述这个模型。

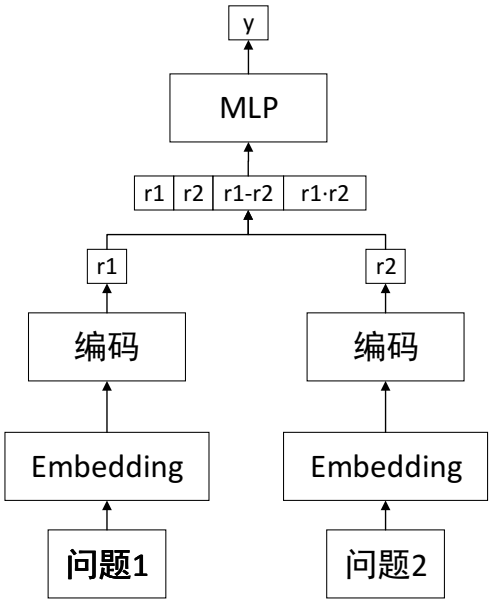

图 2. 模型的总体结构

3.1 Embedding 层

对于任意一个由 t 个单词组成的句子 $\mathrm{S}=(w_1,w_2,...,w_t)$，每个单词依据其在词表中的索引可表示成独热向量（One-hot Representation）的形式 $(0,0,...1,...,0)$，仅仅在其索引的位置为 1，其余位置为 0。使用查表函数 LT 将独热表示的单词 w_i 表示成 d_{emb} 维向量 $\boldsymbol{x}_i$（行向量），如下式所示，其中 $\boldsymbol{E}$ 为 Embedding 矩阵。

$$\boldsymbol{x}_i = w_i \boldsymbol{E} \tag{1}$$

此时句子变为$\mathrm{S}=\left(\boldsymbol{x}_1^{\mathrm{T}}, \boldsymbol{x}_2^{\mathrm{T}}, ..., \boldsymbol{x}_t^{\mathrm{T}}\right)^{\mathrm{T}}$，对于不足 t 个单词的句子，句子后面的部分全补 0 向量。将这个矩阵送入编码层。

3.2 编码层

使用 LSTM 来提取每个句子的信息，最后将其编码成一个向量，这个过程在编码层中实现。LSTM 按照句子顺序依次扫描每一个单词，记住一些信息，将所记住的信息表示成一个向量编码，所以 LSTM 与人类对于文字的阅读行为很相似，较好的保留了文本的特征，这个过程如图 3 所示。

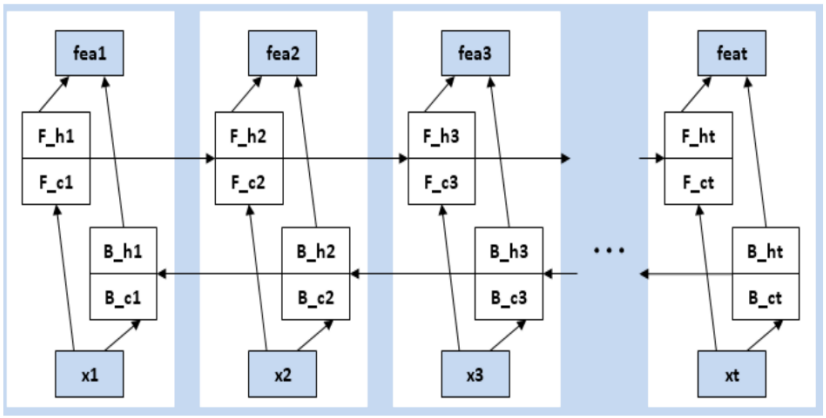

图 3. 双向 LSTM 的结构

每次读取一个单词 $\boldsymbol{x}_i$，称为一个时间步，每个时间步通过一个 LSTM 单元（LSTM Cell）保存信息。在第 i 个时间步，LSTM 单元以 $\boldsymbol{x}_i$，上一个 LSTM 单元的细胞状态 c_{i-1}，上一个 LSTM 单元的输出 h_{i-1} 为输入；输出 h_i 和该 LSTM 单元的细胞状态 c_i（h_i 和 c_i 均为 d_{state} 维的向量）。图中的 F_{c_i}，F_{h_i} 分别是前向 LSTM 的细胞状态和输出；B_{c_i}，B_{h_i} 分别是后向 LSTM 的细胞状态和输出。注意后向 LSTM 是将句子 $\mathrm{S}=\left(\boldsymbol{x}_1^{\mathrm{T}}, \boldsymbol{x}_2^{\mathrm{T}}, ..., \boldsymbol{x}_t^{\mathrm{T}}\right)^{\mathrm{T}}$ 反向输入到 LSTM 中，即先输入 $\boldsymbol{x}_t$，$\boldsymbol{x}_{t-1}$，最后输入 $\boldsymbol{x}_1$。双向 LSTM 既能捕捉到句子以往的信息，也能捕捉到句子往后要表达的信息，然后结合起来作为句子的特征。因此编码层抽取出的特征 r 是：

$$\mathrm{r}=\left[\mathrm{F}_{h_1},\mathrm{B}_{h_1},\mathrm{F}_{h_2},\mathrm{B}_{h_2},...,\mathrm{F}_{h_t},\mathrm{B}_{h_t}\right] \tag{2}$$

下面阐述在每个 LSTM 单元的内部，状态 c_i 和输出 h_i 是如何计算的。如图 4 所示是一个 LSTM 单元的内部结构示意。

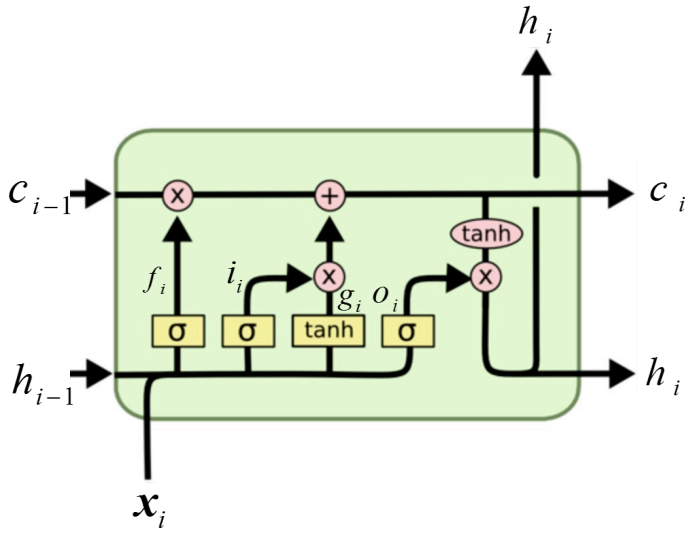

图 4. LSTM 单元的内部结构

首先需要定义 LSTM 单元需要记住多少信息，也称作状态个数 d_{state}，即 h_i 和 c_i 向量的维度。在时间步 i，需要忘记的信息为 f_i，

$$f_i=\sigma\left(\left[h_{i-1},\boldsymbol{x}_i\right]W_f+b_f\right) \tag{3}$$

其中 σ 表示 sigmoid 函数，使用如下一组等式将细胞状态由 c_{i-1} 变为 c_i：

$$i_i=\sigma\left(\left[h_{i-1},\boldsymbol{x}_i\right]W_i+b_i\right) \tag{4}$$

$$g_i=\tanh\left(\left[h_{i-1},\boldsymbol{x}_i\right]W_g+b_g\right) \tag{5}$$

$$c_i=c_{i-1}\otimes f_i+i_i\otimes g_i \tag{6}$$

其中 $\otimes$ 符号表示两个向量的相同位置的元素相乘得到的向量。此后，LSTM 单元的输出为 h_i，

$$o_i=\sigma\left(\left[h_{i-1},\boldsymbol{x}_i\right]W_o+b_o\right) \tag{7}$$

$$h_i=\tanh\left(c_i\right)\otimes o_i \tag{8}$$

总结起来，编码层将 Embedding 层的输出$\left(\boldsymbol{x}_1^{\mathrm{T}},\boldsymbol{x}_2^{\mathrm{T}},...,\boldsymbol{x}_t^{\mathrm{T}}\right)^{\mathrm{T}}$作为输入，通过 t 个前向 LSTM 单元和 t 个后向 LSTM 单元提取出句子的特征 r（参看公式 2）。对于要判断的两个句子S_1和S_2，抽取出两个特征向量r_1和r_2，将这两个特征向量求差，求点积之后输入到多层感知机（Multiple Layer Perceptron，MLP）中进行分类。如图 2 的总结构图所示。

3.3 多层感知机

多层感知机的输入为 p（行向量）：

$$p=\left[r_1,r_2,r_1-r_2,r_1\ r_2\right] \tag{9}$$

感知机由两个隐藏层和输出层组成，MLP 表示成如下的一组等式得到最后的类别得分 s，

$$H_3=\mathrm{relu}\left(pW_3+b_3\right) \tag{10}$$

$$H_4=\mathrm{relu}\left(H_3W_4+b_4\right) \tag{11}$$

$$s=\mathrm{softmax}\left(H_4W_5+b_5\right) \tag{12}$$

由于类别只有两类（0 或 1，分别表示两个问题是否是重复问题），故 s 是两个元素组成的向量，分类时比较那个类别的得分高就判定为哪一类。即类别 y 为：

$$y=\arg\max_i s_i \tag{13}$$

3.4 模型训练

使用交叉熵损失函数，对于两个句子组成的第 i 个样本，其损失函数为：

$$_i=-\sum_{c\in C}y_{ic}\ln\left(s_{ic}\right) \tag{14}$$

y_{ic}表示样本 i 是否属于类别 c，当且仅当属于时 y_{ic}为 1，其余情况为 0。s_{ic}表示样本 i 为类别 c 的概率（即公式 11 中的得分 s）。对于 N 个样本的批处理中，损失函数为：

$$=\frac{1}{N}\sum_{i=1}^{N}\ _i=-\frac{1}{N}\sum_{i=1}^{N}\sum_{c\in C}y_{ic}\ln\left(s_{ic}\right) \tag{15}$$

这个模型的参数是：

$$\theta=\left\{\boldsymbol{E},W_f,b_f,W_i,b_i,W_g,b_g,W_o,b_o,W_3,b_3,W_4,b_4,W_5,b_5\right\} \tag{16}$$

使用 Adam 优化方法[4]优化损失函数 (θ)即可训练出这个模型的参数θ。

4. 实现

使用 Python 和 Google TensorFlow[5]实现本文提出的模型。Python 和 TensorFlow 的版本分别为 2.7 和 r0.12。使用 PC 机 DELL Inspiron 3847，操作系统及版本 Ubuntu Kylin 14.04。下面介绍模型的各个组件的实现。

4.1 Embedding 层

初始化的 Word Embedding 使用 GloVe 模型[6]预训练好的公开的词向量 Glove.6B[7]，选择词向量的维度是$d_{emb}=100$。训练过程中词向量将不断调整。使用 TensorFlow 的词向量查找函数完成将单词转化为词向量的过程。

```
embeddings = tf.nn.embedding_lookup(init_embeddings, x)
```

4.2 编码层

使用 TensorFlow 的 API 实现双向 LSTM 的编码层。

```
cell = tf.nn.rnn_cell.LSTMCell(num_units=state_size,
                               state_is_tuple=True)
cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layers,
                                   state_is_tuple=True)
outputs, states  = tf.nn.bidirectional_dynamic_rnn(
                  cell_fw=cell,
                  cell_bw=cell,
                  dtype=tf.float32,
                  sequence_length=x_lengths,
                  inputs=x)
```

4.3 多层感知机

使用 TensorFlow 的 API 实现多层感知机。首先用编码层的结果生成特征，之后输入多层感知机中进行重复问题检测，最后输出检测结果，这一过程的代码

如下。

```
features = tf.concat(1,
                     [r1, r2, r1 - r2, tf.multiply(r1, r2)])
H3 = tf.nn.relu(tf.nn.xw_plus_b(features, W3, b3,
                name="hidden"))
H4 = tf.nn.relu(tf.nn.xw_plus_b(H3, W4, b4, name="hidden"))
self.scores = tf.nn.xw_plus_b(H4, W5, b5, name="scores")
self.predictions = tf.argmax(self.scores, 1,
                             name="predictions")
```

4.4 模型训练

使用交叉熵计算一批样本的平均损失：

```
losses=tf.nn.softmax_cross_entropy_with_logits(self.scores,
                                               self.input_y)
self.loss = tf.reduce_mean(losses)
```

对求出的 loss 函数使用 Adam 优化方法进行优化。

```
optimizer = tf.train.AdamOptimizer(1e-3)
grads_and_vars = optimizer.compute_gradients(cnn.loss)
train_op = optimizer.apply_gradients(grads_and_vars,
                                     global_step=global_step)
```

然后在每一次训练 step 中使用 train_op 操作即可调整模型的参数。最终得到模型训练之后的模型。

5. 实验

5.1 数据预处理

数据预处理的工作是准备训练集和测试集。首先以一定的比例随机将数据集划分为两部分，删除一些过长的句子，然后统计出最大的句子长度，以便于后面讲句子处理成一致的单词个数。将每个样本的问题 1，问题 2，和是否是重复问题分别记录到训练集和测试集的文件中。这一过程通过运行 python 脚本 create_dataset.py 完成，如图 5 所示（请忽略掉白色的警告信息部分）。输出了训练集和测试集的大小，以及最大的句子长度，其中重复问题的比例是 0.37。忽略了 2413 个句子（主要原因是句子长度过长）。

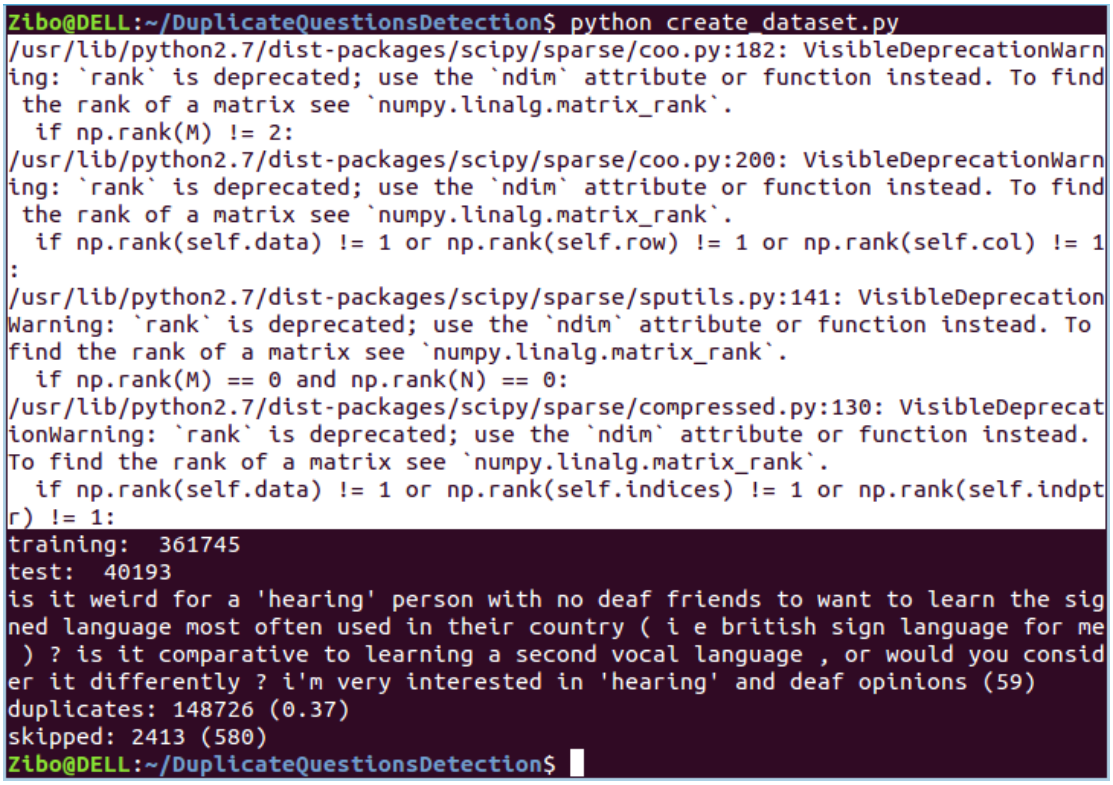

图 5. 数据预处理结果

5.2 模型训练

图 6 和图 7 分别是训练开始时和训练过程中的输出，训练开始时，输出训练模型的超参数，如批处理大小为 64，每隔 100 步保存检查点，dropout 的比例为 0.5，词向量的维度是 100，epoch 的个数为 200 等。

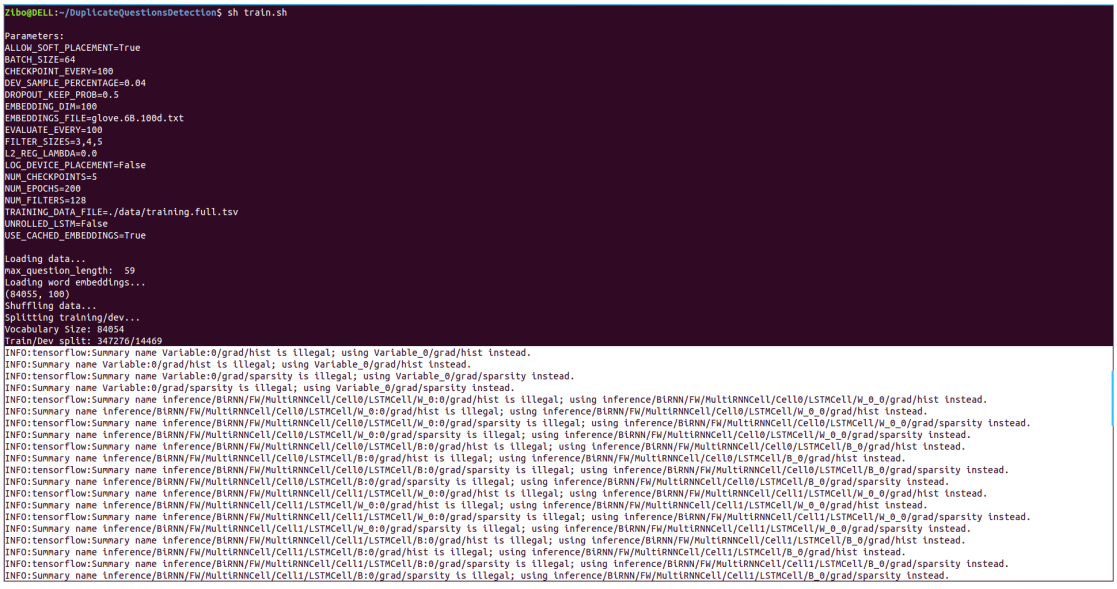

图 6. 训练开始时输出训练参数信息

图 7 是训练过程中的输出，由于每隔 100 步保存一个检查点，所以在 500 步的时候保存了模型，并拿验证集做了验证。从图中可以看到模型在训练过程中的 loss 函数变化，准确率，精确率，召回率和 F1 值。之后进行了模型评价与保存，输出了模型的类别得分，及其预测结果和各个指标的值。

```
2017-04-13T15:18:39.127507:     epoch 0,        step 489,       loss 0.554231,  acc 0.765625,   prec 0.708333,  rec 0.68,       f1 0.693878
2017-04-13T15:18:39.385806:     epoch 0,        step 490,       loss 0.501194,  acc 0.71875,    prec 0.714286,  rec 0.416667,   f1 0.526316
2017-04-13T15:18:39.645399:     epoch 0,        step 491,       loss 0.516224,  acc 0.796875,   prec 0.6,       rec 0.4,        f1 0.48
2017-04-13T15:18:39.899432:     epoch 0,        step 492,       loss 0.440033,  acc 0.78125,    prec 0.642857,  rec 0.5,        f1 0.5625
2017-04-13T15:18:40.154785:     epoch 0,        step 493,       loss 0.613619,  acc 0.6875,     prec 0.615385,  rec 0.347826,   f1 0.444444
2017-04-13T15:18:40.412802:     epoch 0,        step 494,       loss 0.596403,  acc 0.65625,    prec 0.642857,  rec 0.346154,   f1 0.45
2017-04-13T15:18:40.669537:     epoch 0,        step 495,       loss 0.616019,  acc 0.625,      prec 0.4,       rec 0.285714,   f1 0.333333
2017-04-13T15:18:40.926843:     epoch 0,        step 496,       loss 0.473154,  acc 0.78125,    prec 0.882353,  rec 0.555556,   f1 0.681818
2017-04-13T15:18:41.191519:     epoch 0,        step 497,       loss 0.603495,  acc 0.703125,   prec 0.823529,  rec 0.466667,   f1 0.595745
2017-04-13T15:18:41.445714:     epoch 0,        step 498,       loss 0.505302,  acc 0.71875,    prec 0.692308,  rec 0.391304,   f1 0.5
2017-04-13T15:18:41.704002:     epoch 0,        step 499,       loss 0.572887,  acc 0.6875,     prec 0.611111,  rec 0.458333,   f1 0.52381
2017-04-13T15:18:41.960012:     epoch 0,        step 500,       loss 0.506062,  acc 0.65625,    prec 0.518519,  rec 0.608696,   f1 0.56

Evaluation:
scores:  [[-0.54085702  0.79839039]
 [ 0.2259399   0.03918071]
 [ 0.51173711 -0.29744005]
 ...,
 [ 0.9303872  -0.67256153]
 [ 0.92695439 -0.72903132]
 [ 0.20704198  0.05345402]]
predictions:  [1 0 0 ..., 0 0 0]
y_truth:  [1 0 0 ..., 0 0 0]
2017-04-13T15:18:47.615977:     epoch 0,        step 500,       loss 0.544901,  acc 0.722372,   prec 0.632601,  rec 0.636181,   f1 0.634386

Saved model checkpoint to /home/sqt/DuplicateQuestionsDetection/runs/1492067743/checkpoints/model-500

2017-04-13T15:18:50.624832:     epoch 0,        step 501,       loss 0.544468,  acc 0.734375,   prec 0.642857,  rec 0.72,       f1 0.679245
2017-04-13T15:18:50.877965:     epoch 0,        step 502,       loss 0.538743,  acc 0.703125,   prec 0.565217,  rec 0.590909,   f1 0.577778
2017-04-13T15:18:51.141573:     epoch 0,        step 503,       loss 0.513899,  acc 0.765625,   prec 0.619048,  rec 0.65,       f1 0.634146
2017-04-13T15:18:51.398253:     epoch 0,        step 504,       loss 0.587048,  acc 0.71875,    prec 0.785714,  rec 0.647059,   f1 0.709677
2017-04-13T15:18:51.653324:     epoch 0,        step 505,       loss 0.569996,  acc 0.734375,   prec 0.65,       rec 0.565217,   f1 0.604651
2017-04-13T15:18:51.910571:     epoch 0,        step 506,       loss 0.509371,  acc 0.71875,    prec 0.666667,  rec 0.56,       f1 0.608696
2017-04-13T15:18:52.165844:     epoch 0,        step 507,       loss 0.553356,  acc 0.6875,     prec 0.611111,  rec 0.458333,   f1 0.52381
2017-04-13T15:18:52.436470:     epoch 0,        step 508,       loss 0.613405,  acc 0.625,      prec 0.5,       rec 0.25,       f1 0.333333
2017-04-13T15:18:52.695307:     epoch 0,        step 509,       loss 0.569681,  acc 0.6875,     prec 0.705882,  rec 0.444444,   f1 0.545455
2017-04-13T15:18:52.984873:     epoch 0,        step 510,       loss 0.496656,  acc 0.78125,    prec 0.73913,   rec 0.68,       f1 0.708333
2017-04-13T15:18:53.251686:     epoch 0,        step 511,       loss 0.554811,  acc 0.6875,     prec 0.583333,  rec 0.318182,   f1 0.411765
2017-04-13T15:18:53.508981:     epoch 0,        step 512,       loss 0.58896,   acc 0.75,       prec 0.73913,   rec 0.62963,    f1 0.68
2017-04-13T15:18:53.765470:     epoch 0,        step 513,       loss 0.486345,  acc 0.71875,    prec 0.578947,  rec 0.52381,    f1 0.55
2017-04-13T15:18:54.034033:     epoch 0,        step 514,       loss 0.54978,   acc 0.75,       prec 0.8125,    rec 0.5,        f1 0.619048
2017-04-13T15:18:54.310152:     epoch 0,        step 515,       loss 0.485036,  acc 0.796875,   prec 0.761905,  rec 0.666667,   f1 0.711111
2017-04-13T15:18:54.577883:     epoch 0,        step 516,       loss 0.550085,  acc 0.796875,   prec 0.818182,  rec 0.666667,   f1 0.734694
2017-04-13T15:18:54.835686:     epoch 0,        step 517,       loss 0.629762,  acc 0.609375,   prec 0.619048,  rec 0.433333,   f1 0.509804
2017-04-13T15:18:55.090884:     epoch 0,        step 518,       loss 0.603578,  acc 0.671875,   prec 0.619048,  rec 0.5,        f1 0.553191
2017-04-13T15:18:55.349771:     epoch 0,        step 519,       loss 0.562157,  acc 0.6875,     prec 0.565217,  rec 0.565217,   f1 0.565217
2017-04-13T15:18:55.609226:     epoch 0,        step 520,       loss 0.568845,  acc 0.609375,   prec 0.535714,  rec 0.555556,   f1 0.545455
2017-04-13T15:18:55.882037:     epoch 0,        step 521,       loss 0.464827,  acc 0.78125,    prec 0.583333,  rec 0.777778,   f1 0.666667
```

图 7. 训练过程中在检查点保存模型

如图 8 所示是训练过程中损失函数逐渐减少的过程，从图中可以看出模型训练效果不够好，主要是收敛速度不够快。

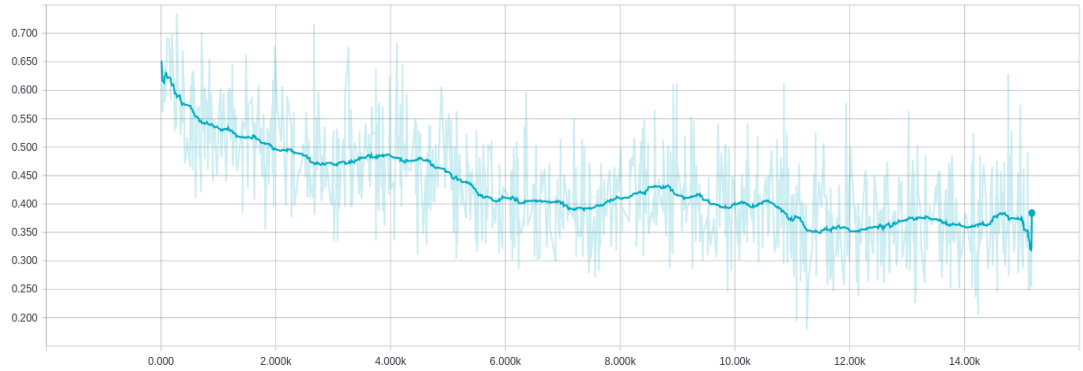

图 8. 训练过程中损失函数的变化

5.3 模型评估

模型评估部分主要评估模型在测试集上的 4 个指标：准确率，精确率，召回率和 F1 值。将 Jaccard 相似度[1]判别法模型判定的结果作为本实验的 baseline，如图 9 所示。图 10 是使用 CNN 作为编码器的模型的评估。图 11 评估本文提出的模型，即基于 LSTM 的模型。评测这些模型的各个指标，如表 1 所示。可以看出，本文提出的基于 LSTM 的模型达到了比传统方法和 CNN 模型更好的效果。

表 1. 评价三个模型在测试集上的效果

模型	准确率	精确率	召回率	F1
Jaccard 相似度	0.6657	0.5151	0.7297	0.6039
基于 CNN 的模型	0.7911	0.6894	0.7316	0.7098
基于 LSTM 的模型	**0.8126**	**0.7085**	**0.7874**	**0.7459**

```
Zibo@DELL:~/DuplicateQuestionsDetection$ python baseline.py
/usr/lib/python2.7/dist-packages/scipy/sparse/coo.py:182: VisibleDeprecationWarn
ing: `rank` is deprecated; use the `ndim` attribute or function instead. To find
 the rank of a matrix see `numpy.linalg.matrix_rank`.
  if np.rank(M) != 2:
/usr/lib/python2.7/dist-packages/scipy/sparse/coo.py:200: VisibleDeprecationWarn
ing: `rank` is deprecated; use the `ndim` attribute or function instead. To find
 the rank of a matrix see `numpy.linalg.matrix_rank`.
  if np.rank(self.data) != 1 or np.rank(self.row) != 1 or np.rank(self.col) != 1
:
/usr/lib/python2.7/dist-packages/scipy/sparse/sputils.py:141: VisibleDeprecation
Warning: `rank` is deprecated; use the `ndim` attribute or function instead. To
find the rank of a matrix see `numpy.linalg.matrix_rank`.
  if np.rank(M) == 0 and np.rank(N) == 0:
/usr/lib/python2.7/dist-packages/scipy/sparse/compressed.py:130: VisibleDeprecat
ionWarning: `rank` is deprecated; use the `ndim` attribute or function instead.
To find the rank of a matrix see `numpy.linalg.matrix_rank`.
  if np.rank(self.data) != 1 or np.rank(self.indices) != 1 or np.rank(self.indpt
r) != 1:

Parameters:
TRAINING_DATA_FILE=./data/test.full.tsv

Loading data...
acc 0.6657      prec 0.5151     rec 0.7297      f1 0.6039
Zibo@DELL:~/DuplicateQuestionsDetection$
```

图 9. 评估基于 Jaccard 距离的模型

```
Zibo@DELL:~/DuplicateQuestionsDetection$ sh eval.sh

Parameters:
ALLOW_SOFT_PLACEMENT=True
BATCH_SIZE=64
CHECKPOINT_DIR=runs/1491655385/checkpoints
EMBEDDINGS_FILE=glove.6B.100d.txt
EVAL_TRAIN=False
LOG_DEVICE_PLACEMENT=False
TEST_DATA_FILE=./data/test.full.tsv
UNROLLED_LSTM=False

Loading data...

Evaluating...

Total number of test examples: 40193
Accuracy: 0.7911
Precision: 0.6894
Recall: 0.7316
F1: 0.7098
Saving evaluation to runs/1491655385/checkpoints/../prediction.csv
Zibo@DELL:~/DuplicateQuestionsDetection$
```

图 10. 评估基于 CNN 的模型

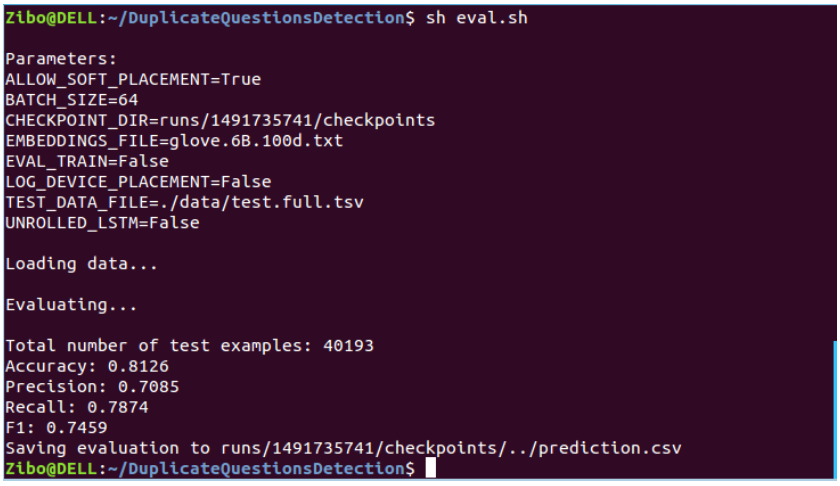

图 11. 评估基于 LSTM 的模型

6. 总结

本文提出了一种基于长短期记忆网络的模型来判定两个问题是否是重复问题。该模型将要判定的两个句子分别使用 LSTM 抽取特征，然后将抽取到的特征使用多层感知机 MLP 进行分类，分类的结果指出这两个句子是否是重复问题。

实际实验过程中测量了分类器在测试集上的准确率，精确率，召回率和 F1 值。相较于传统的使用 Jaccard 距离判断相似度的模型，或者使用卷积神经网络抽取特征的模型，基于 LSTM 的模型具有更好的效果。

但本模型也存在着如收敛速度慢等问题，需要在以后的工作中进行改进。

参考文献

[1] Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin de la Société Vaudoise des Sciences Naturelles 37, 547-579.

[2] https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs, Quora Duplicate Questions Dataset.

[3] Collobert R, Weston J. A unified architecture for natural language processing: Deep neural networks with multitask learning[C]//Proceedings of the 25th international conference on Machine learning. ACM, 2008: 160-167.

[4] Kingma D, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

[5] Google TensorFlow: https://www.tensorflow.org/

[6] Pennington J, Socher R, Manning C D. GloVe: Global Vectors for Word Representation[C]//EMNLP. 2014, 14: 1532-1543.

[7] GloVe: https://nlp.stanford.edu/projects/glove/