

实验报告

16337341 朱志儒

一、环境说明

系统: windows 10 64 位

处理器: Intel® Core™ i5-6300HQ CPU @ 2.30GHz

内存: 8.00GB

GPU: NVIDIA GeForce GTX 950M

语言: Python

调用的库: pandas-0.24.2, numpy-1.16.4, gensim-3.7.3, Keras-2.2.4, sklearn-0.21.2, nltk-3.4.3, matplotlib-3.0.2

二、流程

1、特征选择

Quora 训练集有 40 多万个可能相似的问题对, 每个问题对由问题对 id、问题 id (qid1、qid2)、问题内容 (question1、question2) 和是否重复 (is_duplicate) 组成。

可计算出训练集中正负样例的比例是不相同的, 其中正样例所占的比重为 0.3692, 将 0.3692 作为测试集的预测结果提交后 log loss 为 0.55, 根据对数损失函数:

$$L(y) = -\frac{1}{N} \sum_{i=1}^N (y_i \log p_i + (1 - y_i) \log(1 - p_i))$$

可以计算训练集上的 log loss 为 0.6585, 由此可见训练集与测试集的数据集分布是不相同的。对于这种情况, 我通过使用 Keras 的 class_weight 在模型的训练期间加权损失函数, 以解决训练集与测试集数据分布不相同的问题。

对于训练集和测试集的问题对, 除去停用词后, 统计两个问题的单词个数, 代码如下:

```

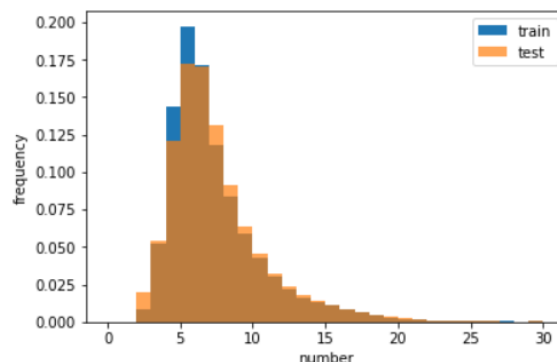
train_sentences = pd.Series(train_set['question1'].tolist() + train_set
                             ['question2'].tolist()).astype(str)
test_sentences = pd.Series(test_set['question1'].tolist() + test_set['qu
                             estion2'].tolist()).astype(str)
train_lens = train_sentences.apply(lambda x: len([i for i in x.split()
            if i not in stop]))
test_lens = test_sentences.apply(lambda x: len([i for i in x.split()
            if i not in stop]))

print("训练集平均值: ", np.mean(train_lens))
print("测试集平均值: ", np.mean(test_lens))

plot.hist(train_lens, bins=30, normed=1, range=[0, 30], label='train')
plot.hist(test_lens, bins=30, normed=1, range=[0, 30], alpha=0.7,
          label='test')
plot.xlabel("number")
plot.ylabel("frequency")
plot.legend()

```

可以得到:



其中，训练集平均值为 6.9613，测试集平均值为 7.0980。由此可见大部分问题对的单词个数为 5~10 个。

对于训练集中的问题对，除去停用词后，统计两个问题单词量之差，代码如下：

```

def length_difference(row):
    question1 = []
    question2 = []
    for word in str(row['question1']).lower().split(' '):
        if word not in stop:
            question1.append(word)
    for word in str(row['question2']).lower().split(' '):
        if word not in stop:

```

```

        question2.append(word)
    return abs(len(question1) - len(question2))

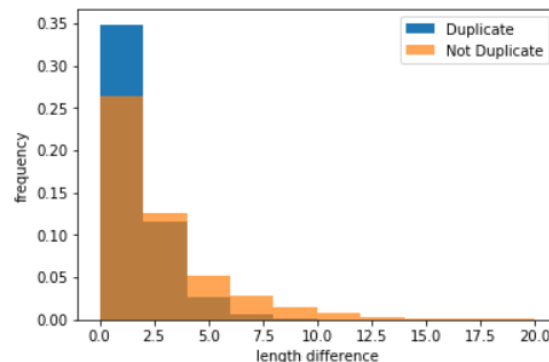
train_length_difference = train_set.apply(length_difference, axis=1,
raw=True)

print("Duplicate 平均值: ", np.mean(train_length_difference[train_set
['is_duplicate'] == 1]))
print("Not Duplicate 平均值: ", np.mean(train_length_difference[train_set
['is_duplicate'] == 0]))

plot.hist(train_length_difference[train_set['is_duplicate'] == 1],
range=[0, 20], normed=True, label='Duplicate')
plot.hist(train_length_difference[train_set['is_duplicate'] == 0],
range=[0, 20], normed=True, alpha=0.7, label='Not Duplicate')
plot.xlabel("length difference")
plot.ylabel("frequency")
plot.legend()

```

可以得到：



其中，标签为“Duplicate”的问题对的平均值为 1.2556，标签为“Not Duplicate”的问题对的平均值为 2.3762。由图可知标签为“Not Duplicate”的问题对中单词量差别更大一些，而标签为“Duplicate”的问题对中单词量差别较小。

对于训练集的问题对，除去停用词后，统计两个问题中相同单词的数目，代码如下：

```

def find_same(row):
    question1 = []
    question2 = []
    for word in str(row['question1']).lower().split(' '):
        if word not in stop:
            question1.append(word)

```

```

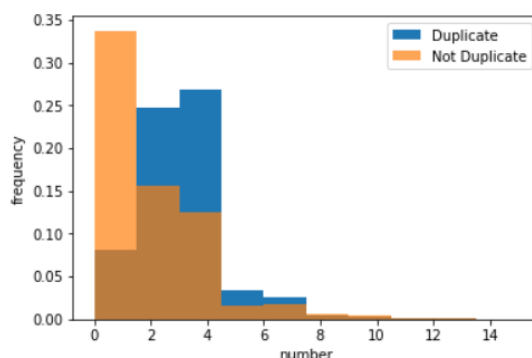
for word in str(row['question2']).lower().split(' '):
    if word not in stop:
        question2.append(word)
same_number1 = len([i for i in question1 if i in question2])
same_number2 = len([i for i in question2 if i in question1])
return max(same_number1, same_number2)

train_same = train_set.apply(find_same, axis=1, raw=True)

print("Duplicate 平均值: ", np.mean(train_same[train_set
    ['is_duplicate'] == 1]))
print("Not Duplicate 平均值: ", np.mean(train_same[train_set
    ['is_duplicate'] == 0]))

plot.hist(train_same[train_set['is_duplicate'] == 1], range=[0, 15],
    normed=True, label='Duplicate')
plot.hist(train_same[train_set['is_duplicate'] == 0], range=[0, 15],
    normed=True, alpha=0.7, label='Not Duplicate')
plot.xlabel("number")
plot.ylabel("frequency")
plot.legend()

```



其中，标签为“Duplicate”的问题对的平均值为 2.8485，标签为“Not Duplicate”的问题对的平均值为 1.8180。从图中可以看出，标签为“Duplicate”的问题对中相同单词的数目为 2~4 个，而标签为“Not Duplicate”的问题对中相同单词的数目为 0~2 个，由此可见标签为“Duplicate”的问题对中相同单词的数目更多一些。

对于训练集的问题对，除去停用词后，统计两个问题中 Tfidf 权重相同的单词数目，代码如下：

```

from sklearn.feature_extraction.text import TfidfVectorizer,

```

CountVectorizer

```
df_data = pd.concat([train_set[['question1', 'question2']],
                     test_set[['question1', 'question2']]], axis=0)
tfidf = TfidfVectorizer(stop_words='english', ngram_range=(1, 1))
questions = pd.Series(df_data['question1'].tolist() +
                     df_data['question2'].tolist()).astype(str)
tfidf.fit_transform(questions)

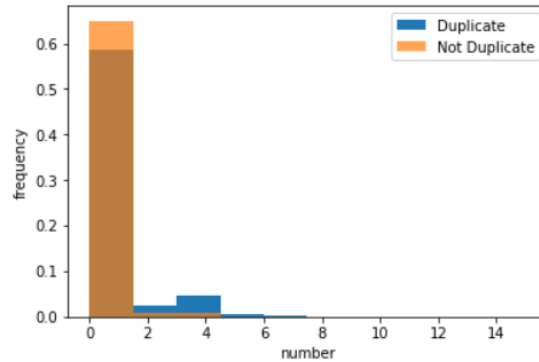
def tfidf_same(row):
    tfidf_q1 = tfidf.transform([str(row['question1'])]).data
    tfidf_q2 = tfidf.transform([str(row['question2'])]).data
    count1 = 0
    count2 = 0
    for w in tfidf_q1:
        if w in tfidf_q2:
            count1 += 1
    for w in tfidf_q2:
        if w in tfidf_q1:
            count2 += 1
    return max(count1, count2)

train_tfidf_same = train_set.apply(tfidf_same, axis=1, raw=True)

print("Duplicate 平均值: ", np.mean(train_tfidf_same[train_set
    ['is_duplicate'] == 1]))
print("Not Duplicate 平均值: ", np.mean(train_tfidf_same[train_set
    ['is_duplicate'] == 0]))

plot.hist(train_tfidf_same[train_set['is_duplicate'] == 1],
          range=[0, 15], normed=True, label='Duplicate')
plot.hist(train_tfidf_same[train_set['is_duplicate'] == 0],
          range=[0, 15], normed=True, alpha=0.7, label='Not Duplicate')
plot.xlabel("number")
plot.ylabel("frequency")
plot.legend()
```

可以得到:



其中，标签为“Duplicate”的问题对的平均值为 0.4218，标签为“Not Duplicate”的问题对的平均值为 0.08346。从图中可看到，标签为“Duplicate”的问题对 Tfidf 权重相同的单词数目较多，而标签为“Not Duplicate”的问题对 Tfidf 权重相同的单词数目大多数为 0。

2、数据处理

使用 Keras 的 Tokenizer 对训练集和测试集的问题对中的词进行统计计数，生成文档词典，以支持基于词典位序生成文本的向量表示。调用 `fit_on_text` 函数根据整个语料库生成 token 词典，再调用 `texts_to_sequences` 函数将每个问题转换为以词为下标的序列形式，接着调用 `pad_sequences` 函数将序列截断或补齐为相同的长度，代码如下：

```
tokenizer = Tokenizer(num_words=MAX_WORDS, lower=False)
tokenizer.fit_on_texts(questions)

question1 = tokenizer.texts_to_sequences(
    train_set['question1'].tolist())
question2 = tokenizer.texts_to_sequences(
    train_set['question2'].tolist())
test_question1 = tokenizer.texts_to_sequences(
    test_set['question1'].tolist())
test_question2 = tokenizer.texts_to_sequences(
    test_set['question2'].tolist())

data1 = pad_sequences(question1, maxlen=MAX_SEQUENCE_LENGTH)
data2 = pad_sequences(question2, maxlen=MAX_SEQUENCE_LENGTH)
labels = np.array(train_set['is_duplicate'].tolist())
```

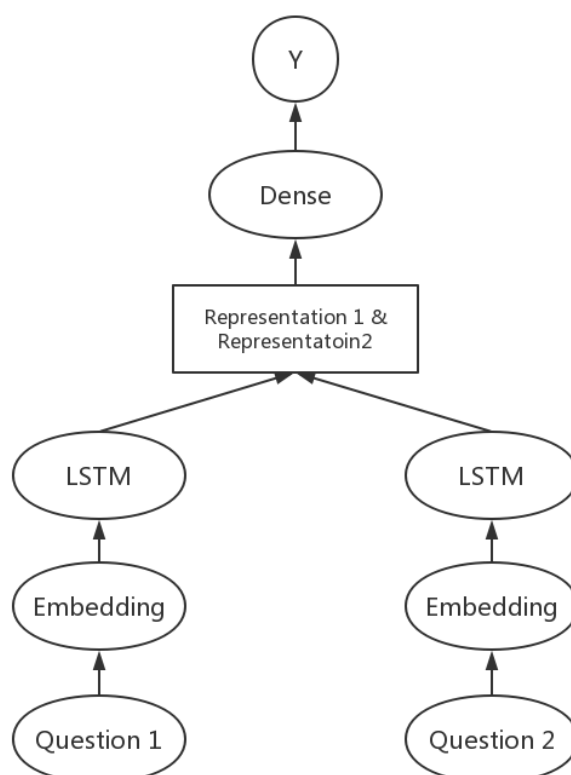
```
test_data1 = pad_sequences(test_question1, maxlen=MAX_SEQUENCE_LENGTH)
test_data2 = pad_sequences(test_question2, maxlen=MAX_SEQUENCE_LENGTH)
test_ids = np.array(test_set['test_id'].tolist())
```

刚开始,我想使用训练集和测试集中的问题作为语料库训练词向量,但采用这种方法最后的效果并不是特别理想。于是,我便使用 Glove 的基于 Wikipedia 2014+Gigaword5 已训练好的词向量,再根据词向量和上一步得到的 word_index 字典便可构造 embedding 矩阵,该 embedding 矩阵可作为模型 embedding 层的权重,代码如下:

```
word_index = tokenizer.word_index
words_number = min(MAX_WORDS, len(word_index)) + 1
word2vec = Word2Vec.load("../input/mymodel/w2v.mod")
embedding_matrix = np.zeros((words_number, EMBEDDING_DIM))
for word, index in word_index.items():
    if word in word2vec.wv.vocab:
        embedding_matrix[index] = word2vec.wv.word_vec(word)
```

3、模型介绍

在这次比赛中,我使用的模型结构如下:



向模型输入问题对可根据词向量生成对应的 embedding, 再将问题对的 embedding 输

入至 LSTM 层得到问题对的 representation，将两个问题的 representation 拼接后，直接输入 Dense 层最后得到预测标签 y。代码如下：

```
def build_model():
    embedding_layer = Embedding(words_number, EMBEDDING_DIM,
                                weights=[embedding_matrix], input_length=MAX_SEQUENCE_LENGTH,
                                trainable=False)
    lstm_layer = LSTM(lstm_num, dropout=drop_lstm,
                      recurrent_dropout=drop_lstm)

    sequence_input1 = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype="int32")
    embedding_sequences1 = embedding_layer(sequence_input1)
    x1 = lstm_layer(embedding_sequences1)

    sequence_input2 = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype="int32")
    embedding_sequences2 = embedding_layer(sequence_input2)
    y1 = lstm_layer(embedding_sequences2)

    merged = concatenate([x1, y1])
    merged = Dropout(drop_dense)(merged)
    merged = BatchNormalization()(merged)

    merged = Dense(dense_num, activation='relu')(merged)
    merged = Dropout(drop_dense)(merged)
    merged = BatchNormalization()(merged)

    preds = Dense(1, activation='sigmoid')(merged)
    model = Model(input=[sequence_input1, sequence_input2],
                  outputs=preds)
    model.compile(loss='binary_crossentropy', optimizer='nadam',
                  metrics=['acc'])
    return model
```

4、模型原理

(1) word2vec 原理

word2vec 的最基本的功能就是通过训练将每个词映射成 k 维实数向量，它采用一个“输入层-隐藏层-输出层”三层的神经网络，根据词语出现的频率用哈夫曼编码，使得所有词频

相似的词在隐藏层激活的激活的内容基本一致，其中出现频率越高的词语，激活的隐藏层数目越少。

word2vec 有 CBOW 和 Skip-Gram 两种模型。

CBOW 模型的训练输入是某个特征词的上下文相关的词，输出则是该特征词。例如：

an efficient method for learning high quality distributed vector

将上下文大小设置为 3，则上下文对应的词有 6 个，前后各 3 个，这 6 个词是 CBOW 模型的输入，而特征词 “learning” 就是需要的输出词。CBOW 模型使用的词袋模型，所以输入的 6 个词是平等的，不考虑他们与需要的输出词 “learning” 的距离，只要在上下文中即可。在 CBOW 模型中，输出是词库中所有词的 softmax 概率，训练目标是期望训练样本中特征词对应的 softmax 概率达到最大。

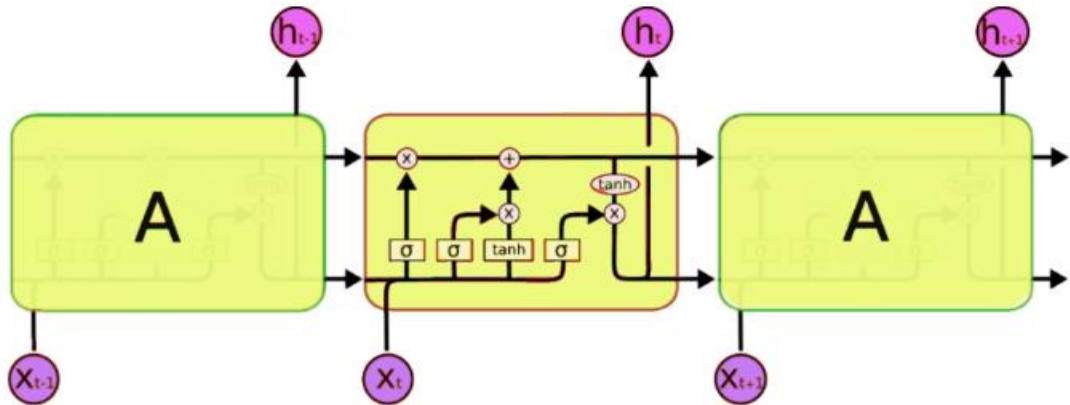
Skip-Gram 模型的思路正好与 CBOW 模型相反，Skip-Gram 模型的输入是一个特定的词，而输出是该特定词所对应的上下文。在上一个例子中，特定词 “learning” 是 Skip-Gram 模型的输入，而其前后各 3 个上下文词是期望的输出。在 Skip-Gram 模型中，输出是 softmax 概率排前 6 的 6 个词，他们作为特定词的上下文。

(2) LSTM 原理

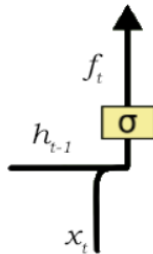
LSTM 的全称是 Long Short-Term Memory，它为提升 RNN 的性能而出现。RNN 只有短期记忆，也就是说，在一小段时间后需要这些信息是可行的，但是一旦输入大量的单词，信息就会在某处丢失，因为在计算距离较远的节点之间的联系时会涉及雅可比矩阵的多次相乘，这会出现梯度消失或梯度膨胀的问题，从而导致信息的丢失。

LSTM 通过增加输入门、遗忘门和输出门使自循环的权重是变化的，这样在模型参数固定的情况下，不同时刻的积分尺度可以动态改变，从而避免梯度消失或梯度膨胀的问题。

LSTM 网络的架构如图所示：

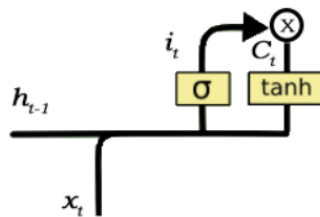


①遗忘门



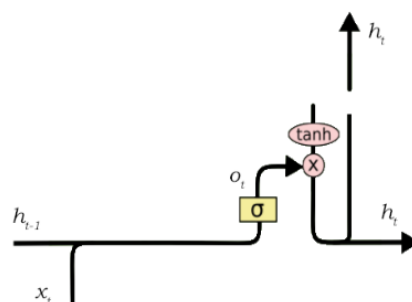
遗忘门负责从单元状态中移除信息，LSTM 不需要这些信息来理解事物，这些不太重要的信息将通过滤波器运算而被移除。上图中遗忘门有 h_{t-1} 和 x_t 两个输入， h_{t-1} 是前一个单元的隐藏状态或输出状态， x_t 是特定时间步的输入，也就是输入序列 x 的第 t 的元素。给定的输入向量与权重矩阵的乘积，再加上偏置项输入至 Sigmoid 函数，Sigmoid 函数将会输出一个取值的范围为 0 到 1 的向量，其对应于单元状态中的每个数值。Sigmoid 函数决定保留哪些值和忘记哪些值。若单元状态取零，那么遗忘门就要求单元状态完全忘记该信息。输出的 Sigmoid 函数向量最后会乘以单元状态。

②输入门



输入门负责将信息添加到单元状态，通过 Sigmoid 函数来调节需要添加到单元状态的值，其作用就是作为一个滤波器过滤来自 h_{t-1} 和 x_t 的信息，接着通过使用 tanh 函数创建一个包含所有可能值的向量，然后将调节滤波器的值（Sigmoid 门控）乘以创建的向量（tanh 函数），然后将这些有用的信息添加到单元状态中。最后基本上确保了添加到单元状态的信息都是重要的，且不是冗余的。

③输出门



并非所有在单元状态运行的信息都适合在特定时间输出, 输出门的工作就是从当前状态中选择有用信息并将其显示为输出。它的功能是把 tanh 函数应用到单元状态之后创建一个向量，将值缩放到-1 与 1 之间，接着使用 h_{t-1} 和 x_t 的值通过 Sigmoid 函数生成一个过滤器，从而调节之前创建的向量中输出的值，最后将过滤器的值乘以之前创建的向量，将其作为输出发送到下个单元的隐藏层。

5、调参方法

在数据处理的时候，我原本打算使用训练集和测试集中的问题作为语料库训练词向量，模型对测试集的预测结果的 logloss 为 0.43 左右，并不是特别理想。于是，我便采用 Glove 的基于 Wikipedia 2014+Gigaword5 训练好的词向量，最后模型对测试集的预测结果的 log loss 为 0.35 左右，效果提升。

由于训练集与测试集的数据集分布是不相同的，所以在训练模型时设置了 class_weight 参数，用来映射类索引到权重值，用于加权损失函数，让模型更为关注来自代表性不足的类

的样本。

在训练模型时，为了增强模型的泛化能力，我借鉴了交叉验证的思想，将训练集数据平均分为 5 份，其中 4 份作为新训练集，1 份作为验证集。使用新训练集训练模型，利用验证集在每轮训练结束时评估损失，训练结束后再对测试集数据进行预测。如此就可获得 5 份不同的预测结果，只需将这 5 份预测结果求平均值就可以得到最终的预测结果。这样做的确可以提升模型的泛化能力，预测结果的 $\log \text{loss}$ 也有所下降。

然后，我使用 Grid search 寻找最佳的 `batch_size`，即模型训练过程中每次梯度更新的样本数，最终找到最佳的 `batch_size` 为 100。

经过上述的调参，最终的预测结果最好的成绩为 0.30646。

6、防止过拟合

(1) 在模型的 Dense 层中将 Dropout 应用于下一层传上来的输入。在训练模型的过程中每次更新时，按一定比率将输入单元随机设置为 0，这将有助于防止过拟合。








(2) 在训练模型时，调用回调函数集合中的 EarlyStopping 函数，将 monitor 设置为 loss，patience 设置为 3。这样，在训练模型的过程中检测 loss 的变化，当存在 3 轮训练 loss 没有发生变化时，将提前停止训练以防模型过拟合。

三、排名

最佳成绩如图所示：

Submission and Description	Private Score	Public Score
kernel526924ad0e (version 26/26) a day ago by JairZhu From "kernel526924ad0e" Script	0.30646	0.30423

在 Private Leaderboard 上的排名为第 995 名，如下图所示：

993	—	Limonium427	  	0.30640	6	2y
994	▲5	林-旭鸣		0.30643	4	2y
995	▼1	akhileshsk		0.30646	15	2y
996	▲7	Alphamon		0.30667	18	2y
997	▼8	pavel17415		0.30671	9	2y