数据挖掘第三次作业报告

小组成员：

李婧如 1700012993

余钟扬 1600018514

郭子瑜 1700012746

姚惠涵 1700012977

分工情况：

李婧如：QANet、报告、ppt

余钟扬：BERT*2、报告、ppt

郭子瑜：XLNet、报告、ppt

姚惠涵：QANet、数据预处理、结果测评、报告

一、实验目的

本次作业内容是基于文档的中文自动问答评测。输入一篇文章及一个相关问题，输出问题答案蕴含在哪句话之中（可能出现在多句话之中，也可能没有答案）。

二、实验过程

本小组主要实现了四个模型来解决问答评测问题，分别是基于问答的 BERT 模型、基于分类的 BERT 模型、XLNet 模型和 QANet 模型。四个模型均基于 Transformer 和 Attention 机制。其中，三个模型针对问答任务，对于每篇文章和问题，寻找蕴含在文章中的答案；一个模型针对分类任务，给定问题，依次输入文章中的每一个句子，判断问题的答案是否蕴含在该句子中。

1. 数据预处理

在使用神经网络模型解决问题之前，首先将数据处理成所需要的格式。

针对基于问答的 BERT 模型和 QANet 模型，将所给的训练集 `train-set.data`、验证集 `validation-set.data` 和测试集 `test-set.data` 整理成 SQuAD 数据集（Stanford Question Answering Dataset）的格式，即（原文，答案，问题）三元组。值得注意的是，虽然 SQuAD 数据集允许一个问题有多个回答，但这多个回答必须是相同短语（如同一个人名、地名）。因此，针对本任务，不能将两个不同的句子作为同一个问题的答案，而应将其作为两个同样问题的答案。在预处理时，还同时生成一个辅助测评的 demo 文件，其中保存每个句子的 idx，用以将生成的答案还原到句子之中。

针对基于分类的 XLNet 模型，将文本预处理成“answer这句话中是否包含对问题question的解答？”。处理过程如下：

```
def generate_sequence(input_path, output_path):
    f1 = open(input_path,'r')
    f2 = open(output_path,'w')
    tokens, classes = [],[]
    for line in f1:
        text, cls = ""“" + line[line.find('？')+1:-2].strip() + ""”这句话中是否包含对问题“" + line[0:line.find('？')] +"？”的解答？", line[-2]
        f2.write(text + " " + cls + '\n')
```

例如：对于下面的 case：

```
腾讯在线教育由哪几个部分组成？    门户网站腾讯网为国内四大门户网站。    0
```

经过文本预处理后的格式如下：

```
“门户网站腾讯网为国内四大门户网站。”这句话中是否包含对问题“腾讯在线教育由哪几个部分组成？”的解答？  0
```

另外，考虑到样本分布不平衡的问题，在预处理后增加了训练集中label为1的case。然后将中文文本转化成模型可读取的格式。

2. 基于问答的 BERT 模型

思路

本次作业中，数据集的格式为：问题 + 句子 + 0/1标签，一个问题对应一篇文章，而这篇文章被拆分为许多分句，句子中是否含有答案用01标签标注。尽管是问答问题，但数据集的形式具有分类任务的形式。经过考虑，我们优先考虑用基于问答的方式来处理数据，也就是在用于SQuAD的BERT上fine-tune模型。原因如下：

1. 作业数据集是将同一篇文章拆分成许多句子，这些句子之间的顺序可能会给答案作出贡献；
2. 问题文本提供了对答案范围的限制，如人名、地名、时间这些答案类型；但在缺乏全文信息的情况下，可能有“张冠李戴”的情况，选出并不符合问题信息的同类答案。

而基于问答处理的处理方法也有一些缺陷：在数据集中，我们发现一些问题的答案**在文章中出现多次。**SQuAD模型原本从文章中输出一段的输出方式不适合答案多次出现的情况。因此，在预训练模型输出答案片段后，我们与**所有句子**进行比对，这样就能得到多个答案的情况。

模型结构

输入为文章和问题，输出为文章中答案句的起始、结束位置。首先将文章和问题通过特殊字符 [SEG] 进行拼接，得到字符串作为 BERT 预训练模型的输入，并获得语义表示。随后将语义表示输入到全连接层中，输出为两个下标，分别表示答案句的起始、结束位置。关键代码如下：

```
class BertForQuestionAnswering(BertPreTrainedModel):
    def __init__(self, config):
        super(BertForQuestionAnswering, self).__init__(config)
        self.num_labels = config.num_labels
        self.bert = BertModel(config)
        #用全连接层输出起始和终止位置
        self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
        self.init_weights()

    def forward(self, input_ids, attention_mask=None, token_type_ids=None, position_ids=None, head_mask=None, start_positions=None, end_positions=None):

        outputs = self.bert(input_ids,
                            attention_mask=attention_mask,
                            token_type_ids=token_type_ids,
```

```
                            position_ids=position_ids,
                            head_mask=head_mask)

        sequence_output = outputs[0]
        #获得BERT的语义表示后送入全连接层，预测答案的起始、终止位置
        logits = self.qa_outputs(sequence_output)
        start_logits, end_logits = logits.split(1, dim=-1)
        start_logits = start_logits.squeeze(-1)
        end_logits = end_logits.squeeze(-1)

        outputs = (start_logits, end_logits,) + outputs[2:]
        if start_positions is not None and end_positions is not None:
            if len(start_positions.size()) > 1:
                start_positions = start_positions.squeeze(-1)
            if len(end_positions.size()) > 1:
                end_positions = end_positions.squeeze(-1)

            ignored_index = start_logits.size(1)
            start_positions.clamp_(0, ignored_index)
            end_positions.clamp_(0, ignored_index)

            loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
            start_loss = loss_fct(start_logits, start_positions)
            end_loss = loss_fct(end_logits, end_positions)
            total_loss = (start_loss + end_loss) / 2
            outputs = (total_loss,) + outputs

        return outputs  # (loss), start_logits, end_logits, (hidden_states),
(attentions)
```

3. 基于分类的 BERT 模型

思路

为了对比考虑全文的预训练模型与拆分成单句的预训练模型效果，我们也用分类任务的BERT模型进行了fine-tuning。

模型结构

输入为问题和答案，输出为0/1分类。首先将问题和答案通过特殊字符 [SEG] 进行拼接，得到字符串作为 BERT 预训练模型的输入，并获得语义表示。随后将语义表示输入到全连接层中，输出为类别。关键代码如下：

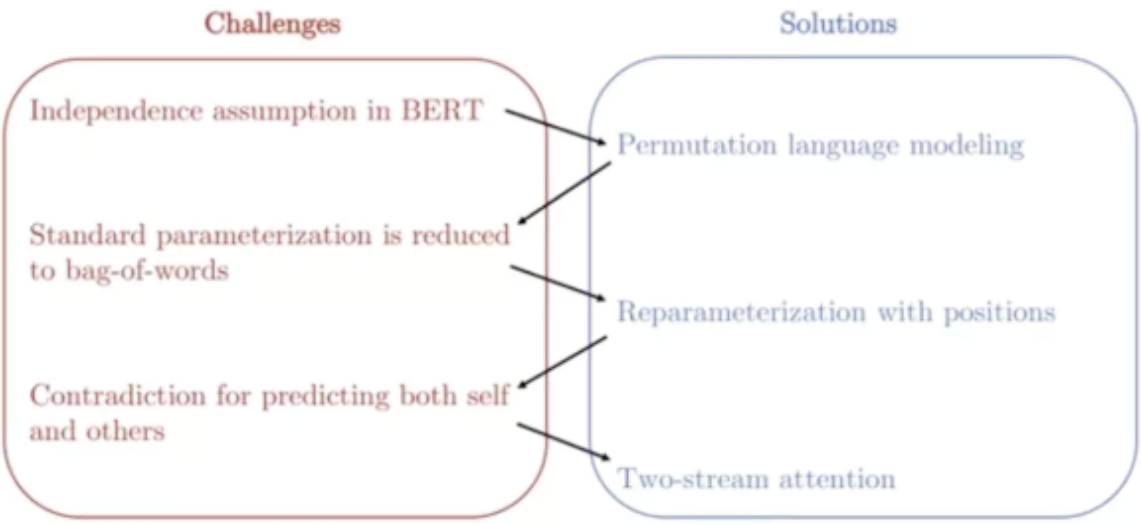
```
class BertForSequenceClassification(BertPreTrainedModel):
    def __init__(self, config):
        super(BertForSequenceClassification, self).__init__(config)
        self.num_labels = config.num_labels
        self.bert = BertModel(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        #用于分类的全连接层
        self.classifier = nn.Linear(config.hidden_size, self.config.num_labels)
        self.init_weights()

    def forward(self, input_ids, attention_mask=None, token_type_ids=None,
position_ids=None, head_mask=None, labels=None):
        outputs = self.bert(input_ids,
                            attention_mask=attention_mask,
```

```
                            token_type_ids=token_type_ids,
                            position_ids=position_ids,
                            head_mask=head_mask)
        pooled_output = outputs[1]
        #用dropout处理bert的输出后用全连接层进行分类
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        outputs = (logits,) + outputs[2:]
        return outputs  # (loss), logits, (hidden_states), (attentions)
```

4. XLNet 模型

原理

XLNet是对BERT进行改进后的模型，如下图所示，针对左边的三类问题，XLNet通过三种方法对模型进行了优化。



以下主要通过以开源的中文XLNet预训练模型（Chinese Pre-Trained XLNet）为基础，期望通过实现一个特殊的文本分类模型，来完成给定任务。

模型的具体实现

加载模型

```
# 加载模型 #
# Load pretrained model
model = load_trained_model_from_checkpoint(
    config_path=paths.config,
    checkpoint_path=paths.model,
    batch_size=BATCH_SIZE,
    memory_len=0,
    target_len=SEQ_LEN,
    in_train_phase=False,
    attention_type=ATTENTION_TYPE_BI,
)

# 加载预训练权重 #
# Build classification model
last = model.output
extract = Extract(index=-1, name='Extract')(last)
dense = keras.layers.Dense(units=768, name='Dense')(extract)
```

```
norm = keras.layers.BatchNormalization(name='Normal')(dense)
output = keras.layers.Dense(units=2, activation='softmax', name='Softmax')(norm)
model = keras.models.Model(inputs=model.inputs, outputs=output)
model.summary()
```

针对下游任务fine-tuning

定义优化器，学习率，损失函数，评估函数，以及回调函数 ,并针对问答判断任务进行模型微调。

```
# 针对下游的finetuning #
# 定义优化器，loss和metrics
model.compile(
    optimizer=RAdam(learning_rate=1e-5),
    loss='sparse_categorical_crossentropy',
    metrics=['sparse_categorical_accuracy'],
)
# 定义callback函数，只保留val_sparse_categorical_accuracy 得分最高的模型 #
from keras.callbacks import ModelCheckpoint
checkpoint = ModelCheckpoint("./model/best_xlnet2.h5",
monitor='val_sparse_categorical_accuracy', verbose=1, save_best_only=True,
                             mode='max')
# 模型训练 #
model.fit_generator(
    generator=train_g,
    validation_data=test_g,
    epochs=EPOCH,
    callbacks=[checkpoint],
)
```

模型训练过程

```
Epoch 1/5
220/220 [==============================] - 7184s 33s/step - loss: 0.9750 - sparse_categorical_accuracy: 0.5419
- val_loss: 1.0137 - val_sparse_categorical_accuracy: 0.5507

Epoch 00001: val_sparse_categorical_accuracy improved from -inf to 0.55067, saving model to ./model/
best_xlnet.h5
_________________________________________________________________
Epoch 2/5
220/220 [==============================] - 6821s 31s/step - loss: 0.9745 - sparse_categorical_accuracy: 0.5396
- val_loss: 1.1513 - val_sparse_categorical_accuracy: 0.5373

Epoch 00002: val_sparse_categorical_accuracy did not improve from 0.55067
_________________________________________________________________
Epoch 3/5
220/220 [==============================] - 6550s 30s/step - loss: 0.9742 - sparse_categorical_accuracy: 0.5413
- val_loss: 1.0562 - val_sparse_categorical_accuracy: 0.5597

Epoch 00003: val_sparse_categorical_accuracy improved from 0.55067 to 0.55967, saving model to ./model/
best_xlnet.h5
_________________________________________________________________
Epoch 4/5
220/220 [==============================] - 6548s 30s/step - loss: 0.9744 - sparse_categorical_accuracy: 0.5413
- val_loss: 1.0713 - val_sparse_categorical_accuracy: 0.5463

Epoch 00004: val_sparse_categorical_accuracy did not improve from 0.55967
_________________________________________________________________
Epoch 5/5
220/220 [==============================] - 6541s 30s/step - loss: 0.9748 - sparse_categorical_accuracy: 0.5419
- val_loss: 1.2497 - val_sparse_categorical_accuracy: 0.5163

Epoch 00005: val_sparse_categorical_accuracy did not improve from 0.55967
_________________________________________________________________
```

5. QANet 模型

QANet 模型与以上介绍的模型不同，没有预训练做辅助。我们下面介绍该模型的结构。

问题的形式化定义为：给定一个包含 n 个单词的上下文片段 $C = \{c_1, c_2, \ldots, c_n\}$，包含 m 个单词的查询语句 $Q = \{q_1, q_2, \ldots, q_m\}$。模型输出为包含 j 个单词的片段 C 中的答案区间 $S = \{c_i, c_{i+1} \ldots, c_{i+j}\}$。

QANet 包含五个部分：嵌入层（embedding layer），嵌入编码层（embedding encoder layer），文章-查询注意层（context-query attention layer），模型编码层（model encoder），输出层（output layer）。

embedding layer

对于一个单词 w，通过连接词向量和字向量表示得到单词的向量。我们采用了网站上的开源中文词向量：https://github.com/Embedding/Chinese-Word-Vectors。形式化表示为：$[x_w; x_c] \in \mathbf{R}^{p_1+p_2}$, $p_1 = 300$, $p_2 = 200$。**固定长度字向量**计算的方式为：每个字得到一个 200 维的向量，每个单词长度为 16 * 200，选取这个矩阵每一列的最大值作为该单词的字向量表示。

embedding encoder layer

基础块的堆叠，每个块形如 $[convolution - layer \times \# + self - attention - layer + feed - forward - layer]$

context-query attention layer

C，Q 表示被编码的文章和问题。context-to-query attention 定义如下：首先计算每个 C，Q 中单词对的相似度矩阵 $S \in \mathbf{R}^{n \times m}$，然后通过 softmax 函数对每行做正规化，得到 $\overline{S}$。注意力机制计算方法为，$A = \overline{S} \cdot Q^T \in \mathbf{R}^{n \times d}$。相似度计算的方程为：$f(q, c) = W_0[q, c, q \odot c]$。$\odot$ 表示矩阵对应位相乘。query-to-context attention 定义如下：设列正规化后得到的矩阵为$\overline{\overline{S}}$，query-to-context attention 为：$B = \overline{S} \cdot \overline{\overline{S}}^T \cdot C^T$。最终输出$[c, a, c \odot a, c \odot b]$。关键代码如下：

```
class CQAttention(nn.Module):
    def __init__(self):
        super().__init__()
        w = torch.empty(d_model * 3)
        lim = 1 / d_model
        nn.init.uniform_(w, -math.sqrt(lim), math.sqrt(lim))
        self.w = nn.Parameter(w)     #weight for similarity matrix

    def forward(self, C, Q, cmask, qmask):
        ss = []
        C = C.transpose(1, 2)
        Q = Q.transpose(1, 2)
        cmask = cmask.unsqueeze(2)
        qmask = qmask.unsqueeze(1)

        shape = (C.size(0), C.size(1), Q.size(1), C.size(2))
        Ct = C.unsqueeze(2).expand(shape)
        Qt = Q.unsqueeze(1).expand(shape)
        CQ = torch.mul(Ct, Qt)              #element-wise multiply
        S = torch.cat([Ct, Qt, CQ], dim=3)  #similar matrix S
        S = torch.matmul(S, self.w)
        S1 = F.softmax(mask_logits(S, qmask), dim=2)
        S2 = F.softmax(mask_logits(S, cmask), dim=1)
        A = torch.bmm(S1, Q)                #context-to-query attention
        B = torch.bmm(torch.bmm(S1, S2.transpose(1, 2)), C)
        #query-to-context attention
        #construct output for model encoder layer
        out = torch.cat([C, A, torch.mul(C, A), torch.mul(C, B)], dim=2)
```

```
        out = F.dropout(out, p=dropout, training=self.training)
        return out.transpose(1, 2)
```

model encoder layer

三个模型编码器堆叠，且参数共享。除了卷积参数有所不同之外，其结构类似于 embedding encoder layer。

output layer

预测每一个文章中的位置是答案起始点和终止点的概率。该概率的计算方法如下：
$p^1 = softmax(W_1[M_0; M_1])$, $p^2 = softmax(W_2[M_0; M_2])$，$M_0, M_1, M_2$ 自底向上分别是三层 model encoder 的输出，$W_1, W2$ 为可训练参数。

loss

$L(\theta) = -\frac{1}{N}\sum_i^N [log(p^1_{y_i^1}), log(p^2_{y_i^2})]$，其中 y_i^1, y_i^2 表示正确的答案开始和结束位置，θ 表示所有可训练参数。

最终选取开始、结束位置时采用如下标准：$max(p_s^1 p_e^2)$, $s \leq e$。

三、实验结果及分析

	macro-F1	micro-F1	macro-P	micro-P	macro-R	micro-R	MAP	MI
基于分类的 BERT	0.8971	0.9816	0.9303	0.9816	0.8691	0.9815	0.8581	0.8
基于问答的 BERT	0.7723	0.9436	0.9154	0.9436	0.7102	0.9436	0.8492	0.8
XLNet	/	/	/	/	/	/	0.374	0.2
QANet	0.1595	/	/	/	/	/	/	/

BERT 结果分析

从结果中可见，分类结果比问答的方法较好，但没有特别明显的优势。这和数据预处理时的猜测并不符合。可能有如下情况：

- 问答模型输出的答案片段不够准确，出现答案片段跨两个句子，无法正确标注等情况。但在检查结果时发现，答案片段基本都刚好是一个分句，无跨句情况。猜测是模型学习到了答案由标点分割的特点。
- 分类任务的优势可能是：每一个句子是否为答案是独立判断的，当出现多个句子为答案时，分类的网络能够将它们全都标成1。而问答任务由于输出一句话的限制，需要与文章中每一句话计算相似度。若答案句表述相差较远，则无法将所有符合条件的答案标为1。
- 另外，由于分类任务的label中绝大多数为0，两种标签的比例不均衡，则模型有可能对一个问题的每个分句都输出0。

XLNet 结果分析

从结果可以看到，在验证集上的accuracy在56%左右，可见分类效果一般；评测结果显示的MAP和MRR数值也不算理想，我们认为可能有以下原因：

- 文本内容过短。一些case的question或answer的句子过短，内容不充分，语意特征不明显，导致训练或预测出现偏差。
- 训练不充分。后期有增大epoch的数值进行训练，但是由于时间不足，没有及时完成模型的训练并评测，会在之后完成。
- 预处理文本训练集后没有打乱数据，可能导致一个batch里面只包含一种类别（0或1），会严重影响训练效果。另外，也可能是因为batch_size大小不合适，训练时影响了拟合过程。
- 其他超参设置不是最佳的，仍需要进行对比判断并调节。

QANet 结果分析

由于算力、时间原因，我们并没有跑出在验证集上完整的结果。但就我们完成的部分而言，QANet 的结果同样不算理想，F1 值较低。我们认为可能有以下原因：

- 将答案直接定义为完整的一句话，而不是确切的答案，可能会令句子以外的其他信息干扰到问答模型自身的训练，也会令模型对问答逻辑产生一定的迷惑性。
- train 过程中 loss 的下降有限，会陷入持平或震荡阶段，可能是受到局部最优影响，没有收敛到全局最优。
- 一些超参数的设置有待调整，如 batch size, learning rate, dropout 等等。
- 在 QANet 的官方论文中，提到训练需要至少 150K 个 steps。可见对于未经预训练的模型，其训练成本较大。以下是训练到 38K 后的结果，可见在有限的时间内，F1 值和 EM（exact match）均不理想。

```
Learning rate: [0.001]

100%|##########| 150/150 [00:11<00:00, 12.63it/s]

STEP    38000 loss 1.038079

VALID loss 0.494301 F1 15.953542 EM 12.195122

TEST loss 0.377586 F1 14.970814 EM 11.612903
```

四、参考文献

1. https://arxiv.org/abs/1906.08237
2. https://arxiv.org/abs/1810.04805
3. https://openreview.net/pdf?id=B14TlG-RW
4. https://github.com/CyberZHG/keras-xlnet](https://links.jianshu.com/go?to=https%3A%2F%2Fgithub.com%2FCyberZHG%2Fkeras-xlnet)
5. https://github.com/ymcui/Chinese-PreTrained-XLNet
6. https://www.jianshu.com/p/c08c2937bd48
7. https://github.com/priya-dwivedi/cs224n-Squad-Project
8. https://github.com/NLPLearn/QANet
9. https://github.com/hengruo/QANet-pytorch
10. https://github.com/Embedding/Chinese-Word-Vectors
11. https://github.com/allenai/bi-att-flow