**使用Pytorch和BERT进行多标签文本分类**

*🔗 原文链接：* [*https://cloud.tencent.com/developer...*](https://cloud.tencent.com/developer/article/1814597)

**介绍**

[自然语言处理](https://cloud.tencent.com/product/nlp?from=10680) (NLP)是一种将非结构化文本处理成有意义的知识的人工智能技术。NLP解决了分类、主题建模、文本生成、问答、推荐等业务问题。虽然TF/IDF矢量化或其他高级词嵌入(如GLOVE和Word2Vec)在此类NLP业务问题上表现出了良好的性能，但这些模型存在局限性就是使用一个向量对词进行编码而不考虑上下文的不同含义。因此，当试图解决理解用户意图所需的问题时，这些模型可能不能很好地执行。一个例子是，当用户与自动聊天机器人交互时，它试图理解用户查询的意图并准确地提供响应。

对于这种情况，NLP中的另一个例子是从下面两个句子中解码上下文意义。

1. A thieve robbed a bank.
2. He went to river bank.

从以上两种表述中，人们很容易就能看出“bank”有两种不同的含义;然而，机器不能区分，因为上面提到的词嵌入使用相同的标记“bank”，而不管他们的上下文意义。为了克服这一挑战，谷歌从Transformers (BERT)模型开发了最先进的双向编码器表示。

**BERT是什么?**

BERT是在8亿单词的图书语料库和2500万单词的英语维基百科上训练的预训练模型。在BERT中，“bank”将有两个不同的含义，因为它们的上下文差异。在保持NLP任务的高性能的同时并不会降低模型构建的训练时间。并且可以从BERT中提取新的语言特征用于模型预测。与RNN、LSTM、CNN等深度学习模型相比，BERT的发展速度要快得多。作为高层次的理解，BERT有两种不同的架构变体:BERT base和BERT large。第一个变型有12个Transformers 块，12个注意头，1.1亿参数，后一个变型有24个Transformers ，16个注意头，3.4亿参数。它在使用过程中完成了两个NLP的任务:遮蔽语言建模和下一句预测。

**数据集**

从此处（https://datahack.analyticsvidhya.com/contest/janatahack-independence-day-2020-ml-hackathon/#ProblemStatement）获取数据集，该数据集可用于研究论文的主题建模的多标签分类对比。对比的目的是从大型的科学文章在线存档中尽可能地容易找到相关的文章。我选择此数据集的原因是，尽管有许多关于二进制分类的Twitter情绪讨论BERT和Pytorch的文章，但很少找到有关处理多类问题的。并且有很多共享代码可能无法正常工作。

查看如下的代码我建议具备python，NLP，深度学习和Pytorch框架的基础知识。必须使用Google帐户才能使用Google Colab帐户。

**处理数据的方法**

在传统的NLP机器学习问题中，我们倾向于清除不需要的文本，例如删除停用词，标点符号，删除符号和数字等。但是，在BERT中，不需要执行此类预处理任务，因为BERT使用了这些 单词的顺序和位置，以了解用户输入的意图。

ML / DL工程师应该从不同方面探索数据集，以真正了解他们手中的数据类型，这是一个好习惯。NLP的典型功能是单词计数，动词计数，形容词计数，数字计数，标点符号计数，双字母组计数，三字组计数等。为简便起见，我已展示了如何对单词计数列进行计数，其中单个标题中使用的总单词数将被计算在内。您可能还需要处理类似于TITLE的Abstract列，以及ABSTRACT和TITLE的组合。

下面的命令创建“ WORD\_COUNT”列。

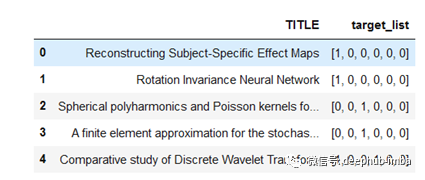
|  |
| --- |
| JavaScript  df\_raw['WORD\_COUNT'] = df\_raw['TITLE'].apply(lambda x: len(x.split()) |

这将生成“ WORD\_COUNT”的分布图，即标题的长度。

如您所见，文章标题的大部分以10个单词为中心，这是预期的结果，因为TITLE应该简短，简洁且有意义。

由于我将仅使用“ TITLE”和“ target\_list”，因此我创建了一个名为df2的新数据框。df2.head（）命令显示训练数据集中的前五个记录。如您所见，两个目标标签被标记到最后的记录，这就是为什么这种问题称为多标签分类问题的原因。

|  |
| --- |
| JavaScript  df2 = df\_raw[['TITLE', 'target\_list']].copy()    df2.head() |



同时，设置将用于模型训练的参数。由于我更喜欢使用2\*base数字，因此最大长度设置为16，这涵盖了大部分“ TITLE”长度。训练和有效批处理大小设置为32。epoch为4，因为它很容易在几个epoch上过度拟合。我从lr=0.00001开始学习。您可以随意尝试不同的值以提高准确性。

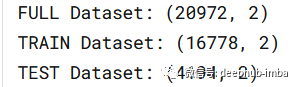
|  |
| --- |
| JavaScript  # Sections of config  # Defining some key variables that will be used later on in the training  MAX\_LEN = 16  TRAIN\_BATCH\_SIZE = 32  VALID\_BATCH\_SIZE = 32  EPOCHS = 4  LEARNING\_RATE = 1e-05  tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased') |

让我们创建一个称为“ CustomDataset”的通用类。Class从我们的原始输入特征生成张量，并且Pytorch张量可以接受class的输出。它期望具有上面定义的“ TITLE”，“ target\_list”，max\_len，并使用BERT toknizer.encode\_plus函数将输入设置为数字矢量格式，然后转换为张量格式返回。

|  |
| --- |
| JavaScript  class CustomDataset(Dataset):  def \_\_init\_\_(self, dataframe, tokenizer, max\_len):  self.tokenizer = tokenizer  self.data = dataframe  self.title = dataframe['TITLE']  self.targets = self.data.target\_list  self.max\_len = max\_len    def \_\_len\_\_(self):  return len(self.title)        def \_\_getitem\_\_(self, index):  title = str(self.title[index])  title = " ".join(title.split())    inputs = self.tokenizer.encode\_plus(  title,  None,  add\_special\_tokens=True,  max\_length=self.max\_len,  padding='max\_length',  return\_token\_type\_ids=True,  truncation=True    )    ids = inputs['input\_ids']  mask = inputs['attention\_mask']  token\_type\_ids = inputs["token\_type\_ids"]  return {  'ids': torch.tensor(ids, dtype=torch.long),  'mask': torch.tensor(mask, dtype=torch.long),  'token\_type\_ids': torch.tensor(token\_type\_ids, dtype=torch.long),  'targets': torch.tensor(self.targets[index], dtype=torch.float)    } |

数据集的80％用于模型训练，而20％用于验证。测试数据集完全用于测试目的。

|  |
| --- |
| JavaScript  train\_size = 0.8  train\_dataset = df2.sample(frac=train\_size,random\_state=200)  valid\_dataset = df2.drop(train\_dataset.index).reset\_index(drop=True)  train\_dataset = train\_dataset.reset\_index(drop=True)      print("FULL Dataset: {}".format(df2.shape))  print("TRAIN Dataset: {}".format(train\_dataset.shape))  print("TEST Dataset: {}".format(valid\_dataset.shape))      training\_set = CustomDataset(train\_dataset, tokenizer, MAX\_LEN)  validation\_set = CustomDataset(valid\_dataset, tokenizer, MAX\_LEN) |



我们已经讨论了将张量准备为输入特征的大部分基础工作。现在，构建BERT模型很容易。由于来自模型的冗长输出，我已简化为仅显示模型。我已使用dropout 0.3来随机减少特征，以最大程度地减少第2层的过拟合。第3层采用了768维特征，这些特征是从使用BERT的第2层输出的。它返回6个特征，这是对目标列表的最终预测。

|  |
| --- |
| JavaScript  class BERTClass(torch.nn.Module):  def \_\_init\_\_(self):  super(BERTClass, self).\_\_init\_\_()  self.l1 = transformers.BertModel.from\_pretrained('bert-base-uncased')  self.l2 = torch.nn.Dropout(0.3)  self.l3 = torch.nn.Linear(768, 6)    def forward(self, ids, mask, token\_type\_ids):  \_, output\_1= self.l1(ids, attention\_mask = mask, token\_type\_ids = token\_type\_ids)  output\_2 = self.l2(output\_1)  output = self.l3(output\_2)  return output    model = BERTClass()  model.to(device) |

BCE损失函数用于找出模型预测值与实际目标值之间的误差。使用Adam优化器。损失功能请参见下文。

|  |
| --- |
| JavaScript  def loss\_fn(outputs, targets):  return torch.nn.BCEWithLogitsLoss()(outputs, targets)    optimizer = torch.optim.Adam(params = model.parameters(), lr=LEARNING\_RATE) |

同时我还创建了检查点，可以在训练期间保存最佳模型。当需要从停下来的地方继续训练时，这将有助于减少训练时间。创建检查点可以节省时间，以便从头开始进行重新训练。如果您对从最佳模型生成的输出感到满意，则不需要进一步的微调，则可以使用模型进行推断。

|  |
| --- |
| JavaScript  def load\_ckp(checkpoint\_fpath, model, optimizer):    """  checkpoint\_path: path to save checkpoint  model: model that we want to load checkpoint parameters into   optimizer: optimizer we defined in previous training    """  # load check point  checkpoint = torch.load(checkpoint\_fpath)  # initialize state\_dict from checkpoint to model  model.load\_state\_dict(checkpoint['state\_dict'])  # initialize optimizer from checkpoint to optimizer  optimizer.load\_state\_dict(checkpoint['optimizer'])  # initialize valid\_loss\_min from checkpoint to valid\_loss\_min  valid\_loss\_min = checkpoint['valid\_loss\_min']  # return model, optimizer, epoch value, min validation loss   return model, optimizer, checkpoint['epoch'], valid\_loss\_min.item()  import shutil, sys     def save\_ckp(state, is\_best, checkpoint\_path, best\_model\_path):    """  state: checkpoint we want to save  is\_best: is this the best checkpoint; min validation loss  checkpoint\_path: path to save checkpoint  best\_model\_path: path to save best model  """  f\_path = checkpoint\_path  # save checkpoint data to the path given, checkpoint\_path  torch.save(state, f\_path)  # if it is a best model, min validation loss  if is\_best:  best\_fpath = best\_model\_path  # copy that checkpoint file to best path given, best\_model\_path  shutil.copyfile(f\_path, best\_fpath)  def train\_model(start\_epochs, n\_epochs, valid\_loss\_min\_input,   training\_loader, validation\_loader, model,   optimizer, checkpoint\_path, best\_model\_path):    # initialize tracker for minimum validation loss  valid\_loss\_min = valid\_loss\_min\_input   for epoch in range(start\_epochs, n\_epochs+1):  train\_loss = 0  valid\_loss = 0  model.train()  print('############# Epoch {}: Training Start #############'.format(epoch))  for batch\_idx, data in enumerate(training\_loader):  #print('yyy epoch', batch\_idx)  ids = data['ids'].to(device, dtype = torch.long)  mask = data['mask'].to(device, dtype = torch.long)  token\_type\_ids = data['token\_type\_ids'].to(device, dtype = torch.long)  targets = data['targets'].to(device, dtype = torch.float)  outputs = model(ids, mask, token\_type\_ids)  optimizer.zero\_grad()  loss = loss\_fn(outputs, targets)  #if batch\_idx%5000==0:  # print(f'Epoch: {epoch}, Training Loss: {loss.item()}')  optimizer.zero\_grad()  loss.backward()  optimizer.step()  #print('before loss data in training', loss.item(), train\_loss)  train\_loss = train\_loss + ((1 / (batch\_idx + 1)) \* (loss.item() - train\_loss))  #print('after loss data in training', loss.item(), train\_loss)  print('############# Epoch {}: Training End #############'.format(epoch))  print('############# Epoch {}: Validation Start #############'.format(epoch))  ######################   # validate the model #  ######################  model.eval()  with torch.no\_grad():  for batch\_idx, data in enumerate(validation\_loader, 0):  ids = data['ids'].to(device, dtype = torch.long)  mask = data['mask'].to(device, dtype = torch.long)  token\_type\_ids = data['token\_type\_ids'].to(device, dtype = torch.long)  targets = data['targets'].to(device, dtype = torch.float)  outputs = model(ids, mask, token\_type\_ids)  loss = loss\_fn(outputs, targets)  valid\_loss = valid\_loss + ((1 / (batch\_idx + 1)) \* (loss.item() - valid\_loss))  val\_targets.extend(targets.cpu().detach().numpy().tolist())  val\_outputs.extend(torch.sigmoid(outputs).cpu().detach().numpy().tolist())    print('############# Epoch {}: Validation End #############'.format(epoch))  # calculate average losses  #print('before cal avg train loss', train\_loss)  train\_loss = train\_loss/len(training\_loader)  valid\_loss = valid\_loss/len(validation\_loader)  # print training/validation statistics   print('Epoch: {} \tAvgerage Training Loss: {:.6f} \tAverage Validation Loss: {:.6f}'.format(  epoch,   train\_loss,  valid\_loss  ))  # create checkpoint variable and add important data  checkpoint = {  'epoch': epoch + 1,  'valid\_loss\_min': valid\_loss,  'state\_dict': model.state\_dict(),  'optimizer': optimizer.state\_dict()  }    # save checkpoint  save\_ckp(checkpoint, False, checkpoint\_path, best\_model\_path)  ## TODO: save the model if validation loss has decreased  if valid\_loss <= valid\_loss\_min:  print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model ...'.format(valid\_loss\_min,valid\_loss))  # save checkpoint as best model  save\_ckp(checkpoint, True, checkpoint\_path, best\_model\_path)  valid\_loss\_min = valid\_loss  print('############# Epoch {} Done #############\n'.format(epoch))  return model |

“train\_model”被创建来训练模型，“checkpoint\_path”是训练模型的参数将被保存为每个epoch，“best\_model”是最好的模型将被保存的地方。

|  |
| --- |
| JavaScript  checkpoint\_path = '/content/drive/My Drive/NLP/ResearchArticlesClassification/checkpoint/current\_checkpoint.pt'    best\_model = '/content/drive/My Drive/NLP/ResearchArticlesClassification/best\_model/best\_model.pt'    trained\_model = train\_model(1, 4, np.Inf, training\_loader, validation\_loader, model,   optimizer,checkpoint\_path,best\_model) |

训练结果如下：

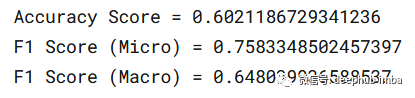
|  |
| --- |
| JavaScript  ############# Epoch 1: Training Start #############  ############# Epoch 1: Training End #############  ############# Epoch 1: Validation Start #############  ############# Epoch 1: Validation End #############  Epoch: 1 Avgerage Training Loss: 0.000347 Average Validation Loss: 0.001765  Validation loss decreased (inf --> 0.001765). Saving model ...  ############# Epoch 1 Done #############    ############# Epoch 2: Training Start #############  ############# Epoch 2: Training End #############  ############# Epoch 2: Validation Start #############  ############# Epoch 2: Validation End #############  Epoch: 2 Avgerage Training Loss: 0.000301 Average Validation Loss: 0.001831  ############# Epoch 2 Done #############    ############# Epoch 3: Training Start #############  ############# Epoch 3: Training End #############  ############# Epoch 3: Validation Start #############  ############# Epoch 3: Validation End #############  Epoch: 3 Avgerage Training Loss: 0.000263 Average Validation Loss: 0.001896  ############# Epoch 3 Done #############    ############# Epoch 4: Training Start #############  ############# Epoch 4: Training End #############  ############# Epoch 4: Validation Start #############  ############# Epoch 4: Validation End #############  Epoch: 4 Avgerage Training Loss: 0.000228 Average Validation Loss: 0.002048  ############# Epoch 4 Done ############# |

因为我只执行了4个epoch，所以完成得很快，我将threshold设置为0.5。你可以试试这个阈值，看看是否能提高结果。

|  |
| --- |
| JavaScript  val\_preds = (np.array(val\_outputs) > 0.5).astype(int)  val\_preds  array([[0, 0, 1, 0, 0, 0],  [0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0],  ...,  [0, 0, 1, 0, 0, 0],  [0, 1, 0, 0, 0, 0],  [1, 0, 0, 0, 0, 0]]) |

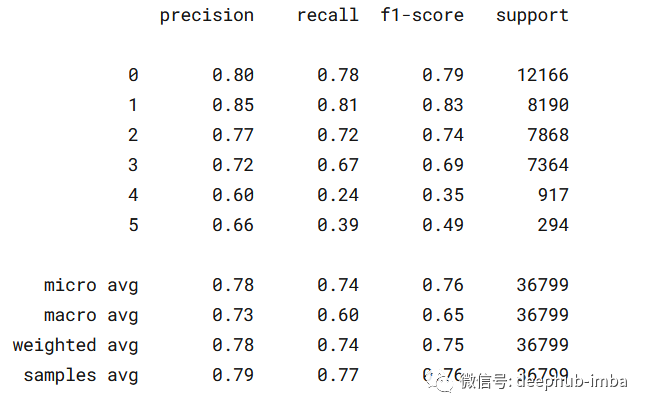
让我们将精确度和F1得分定义为模型性能的指标。F1将被用于评估。

|  |
| --- |
| JavaScript  accuracy = metrics.accuracy\_score(val\_targets, val\_preds)  f1\_score\_micro = metrics.f1\_score(val\_targets, val\_preds, average='micro')  f1\_score\_macro = metrics.f1\_score(val\_targets, val\_preds, average='macro')  print(f"Accuracy Score = {accuracy}")  print(f"F1 Score (Micro) = {f1\_score\_micro}")  print(f"F1 Score (Macro) = {f1\_score\_macro}") |



使用混淆矩阵和分类报告，以可视化我们的模型如何正确/不正确地预测每个单独的目标。

|  |
| --- |
| JavaScript  from sklearn.metrics import multilabel\_confusion\_matrix as mcm, classification\_report  cm\_labels = ['Computer Science', 'Physics', 'Mathematics',  'Statistics', 'Quantitative Biology', 'Quantitative Finance']  cm = mcm(val\_targets, val\_preds)  print(classification\_report(val\_targets, val\_preds)) |



模型预测的准确率为76%。F1得分低的原因是有六个类的预测，通过结合“TITLE”和“ABSTRACT”或者只使用“ABSTRACT”来训练可以提高它。我对这两个案例都进行了训练，发现“ABSTRACT”特征本身的F1分数比标题和标题与抽象相结合要好得多。在没有进行超参数优化的情况下，我使用测试数据进行推理，并在private score中获得0.82分。

有一些事情可以做，以提高F1成绩。一个是微调模型的超参数，你可能想要实验改变学习速率，退出率和时代的数量。在对模型微调的结果满意之后，我们可以使用整个训练数据集，而不是分成训练和验证集，因为训练模型已经看到了所有可能的场景，使模型更好地执行。

你可以在谷歌Colab查看这个项目源代码

https://colab.research.google.com/drive/1SPxxEW9okgnbMdk1ORlfSQI4rjV2tVW\_#scrollTo=EJQRHd7VVMap

作者:Kyawkhaung

原文地址:https://kyawkhaung.medium.com/multi-label-text-classification-with-bert-using-pytorch-47011a7313b9

deephub翻译组

文章分享自微信公众号：

本文参与 [腾讯云自媒体分享计划](https://cloud.tencent.com/developer/support-plan) ，欢迎热爱写作的你一起参与！

如有侵权，请联系

cloudcommunity@tencent.com

删除。