客户ML case share 第4集

1. 客户项目的上下文：

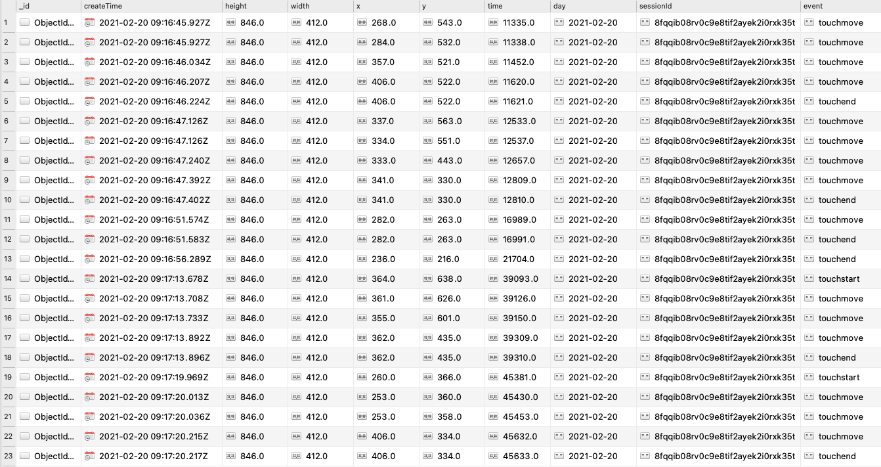
需求：利用ML模型来生成用户触屏行为序列（**业务侧不关心是否用ML来建模，但是算法侧还是倾向于先用ML建模来尝试**）。

使用平台和框架：SageMaker + TF

痛点：模型选型和效果调试。

1. 过程重放：

原始的打点数据如下图所示：

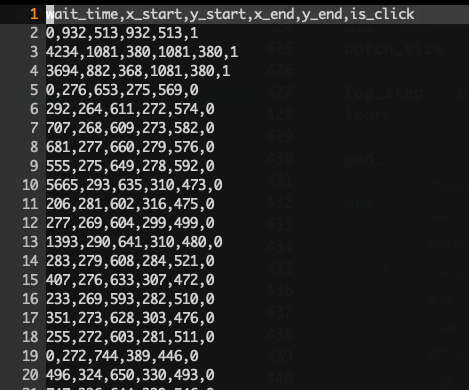


**字段简单解释**：

Touchstart，touchmove，touchend这三个事件按照这个先后顺序组合在一起称作用户的**一次滑屏操作**。如果touchstart的坐标等于touchend的坐标那么就是**一次点击操作**。

一个session反映的是单个用户从登录APP到离开APP的整个过程中的所有滑屏操作和点击操作的集合。**一个session的多个滑屏和点击操作即为用户触屏序列**。

在实际使用中，由于不同浏览器对事件touchmove的标记方法有所不同，为了减少这个带来的影响，最后客户决定不再使用touchmove事件，只使用touchstart和touchend两种事件。最终使用的数据类似如下：



上图中的x\_start/y\_start对应于touchstart事件，x\_end/y\_end对应于touchend事件，is\_click标记是说这个操作是点击操作------这个标记对于客户来说是一个非常有用的信号，对于原始数据中touchstart和touchend不相等但是is\_click标记为1的情况是由于前端收集有问题导致的。

1. 模型调研：

**通过网上搜索并没有发现有直接做用户触屏行为序列生成这样任务的模型**。但是如果以session来看的话，每个session都可以归结为时间序列生成，因此我们尝试把这个任务转化为时间序列生成的任务。

对于常见的生成任务的模型有三大类：基于LSTM的Seq2Seq，基于VAE变分自编码器的，基于GAN的。（三种方法的复杂度越来越高，超参数调试也越来越难）

**考虑到这个任务的特殊性，这里我们并没有考虑优先尝试seq2seq或者VAE的方案**。通过网上搜索时间序列生成任务的模型，发现了两个模型可以去尝试：

TimeGAN （<https://github.com/jsyoon0823/TimeGAN> ）

DoppelGANger（<https://github.com/fjxmlzn/DoppelGANger> ）

查看两个模型相关的paper，整体感觉TimeGAN相对比DoppelGANger要简单一些，因为我们先选择用TimeGAN来跑这个任务。

1. TimeGAN模型调试：
2. **训练样本的构造**：

我们首先参考的TimeGAN模型的第一个版本的实现（<https://github.com/ydataai/ydata-synthetic/blob/master/src/ydata_synthetic/synthesizers/timeseries/timegan/model.py>） ，它的训练集是基于stock股票数据，并且做了rolling window（即滑动窗口）的时间序列样本的构造。Github原作者有如下的代码：

# Preprocess the dataset

temp\_data = []

# Cut data by sequence length

for i in range(0, len(ori\_data) - seq\_len):

\_x = ori\_data[i:i + seq\_len]

temp\_data.append(\_x)

**也就是说它最后构造的每个样本都是一条时间序列（注意这里是如何从raw data构造训练样本的），序列长度是定长的seq\_len。**

**注意：这里有一个很大的区别，就是客户当前的数据集有session的边界而且session的长度不定长，而上面的股票数据并没有session边界**（**数据集的相似性是应该着重考虑的一个点，尤其是我们使用“拿来主义”的时候**。**相似性的角度最好从多个方面来综合看：比如语义层面，质量层面，形状层面，特征覆盖度层面等等**）。

所以在客户的这个任务中，我理解的样本的构造可能的思路如下：

1. 每个自然session就是一条单独的时间序列。
2. 通过统计session的长度分布，找到一个合适的长度作为max seqence len
3. 对于超过max sequence len的session：

或者按照max sequence len长度为间隔来把单个session切分为多条序列；

或者简单丢弃超过max sequence len的部分；

甚至可以在session内用roll window方式构造多条时间序列。

上面的三种处理方式那种更好，也是case by case的。

1. 对于得到的每条时间序列，补齐序列到max sequence len长度。

从上面看起来样本构造还是挺麻烦的，所以起初的时候，客户先按照原作者rolling window方法构造样本的方式来看看效果如何；但是最后的方案使用了上面提议的构造样本的方式来建模。

1. 训练过程

**训练速度很慢？**

TimeGAN模型的实现是基于LSTM或者GRU的，训练速度慢以及GPU使用率低看起来也符合预期。经过测试，发现用GPU训练的速度和用CPU训练的速度差不多（这里使用的是p3.2xlarge和C5.18xlarge）

**训练的时候发生OOM？**

原版代码的作者有如下的代码：

for \_ in tqdm(range(train\_steps), desc='Joint networks training'):

#Train the generator (k times as often as the discriminator)

# Here k=2

for \_ in range(2):

X\_ = next(self.get\_batch\_data(data, n\_windows=len(data)))

Z\_ = next(self.get\_batch\_noise())

# --------------------------

# Train the generator

# --------------------------

step\_g\_loss\_u, step\_g\_loss\_s, step\_g\_loss\_v = self.train\_generator(X\_, Z\_)

# --------------------------

# Train the embedder

# --------------------------

step\_e\_loss\_t0 = self.train\_embedder(X\_)

X\_ = next(self.get\_batch\_data(data, n\_windows=len(data)))

Z\_ = next(self.get\_batch\_noise())

step\_d\_loss = self.discriminator\_loss(X\_, Z\_)

if step\_d\_loss > 0.15:

step\_d\_loss = self.train\_discriminator(X\_, Z\_)

**上面的代码会在循环中调用get\_batch\_data，get\_batch\_noise；这样的用法其实不太好，因为在循环中不停的产生很多临时对象，而且Python的GC不会很及时的回收那些已经没有被引用的对象，从而导致内存的不断增加最后OOM。**

**解决办法**：

在train 函数的开头（for循环的外面）设置两个迭代器iter1 = self.get\_batch\_data(data, n\_windows=len(data))， iter2=self.get\_batch\_noise()，然后在循环中直接 X\_=iter1.get\_next()，Z\_=iter2.get\_next()

**这样修改完代码以后，发现不仅解决了OOM的问题，而且还使得训练速度大幅提升**（修改代码后，p3.2xlarge速度是C5.18xlarge的很多倍。主要原因是不用每个step/batch都创建新的TF data对象，也就是data input这块的处理上去了，从而使得CPU准备一个step/batch的数据的速度更快了，也就是GPU处在idle状态等待CPU准备好的一个step/batch的数据的时间更少了，因此在单位时间内GPU使用率提升很多，从而训练速度提升很多）。

**模型超参数的各种尝试**？

根据GAN-based模型的套路，一般会设置generator的学习率比discriminator的学习率稍微大一些；

而且让generator跑的step是discriminator跑的step的倍数。

上面这两个trick其实是WGAN出来之前，GAN相关模型经常使用的；而这里我们使用了这些trick来训练TimeGAN，发现效果没有什么提升。

**这里其实我们还尝试了很多方法（比如把模型变小以及其他的超参数调整）来尝试提升生成序列的效果，很不幸的是基本没有什么本质的提升**！

**训练过程中发现的数据质量问题？**

在训练过程中，经常发现生成的序列的一些奇怪的现象：比如坐标有负坐标的情况，序列中有连续点击事件发生等等。

通过反复检查数据集，发现原始数据集确实有负坐标以及连续点击事件。和业务侧沟通发现连续点击事件是合理的用户行为。

最后做的处理就是把负坐标的数据清理掉。连续点击事件在原始数据中会出现，**但是用TimeGAN模型来生成的序列中的连续点击事件太长并且频率太高，这个是有问题的**。

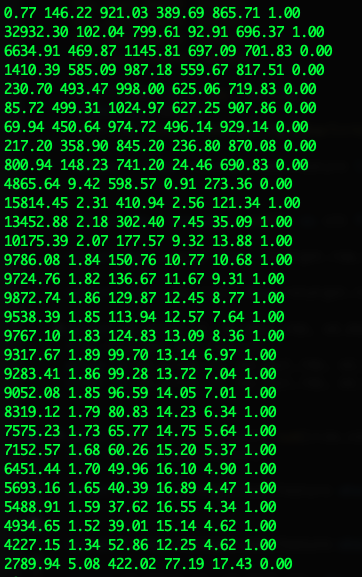
**改造代码对序列变长处理？**

刚开始的想法是利用tf.keras的padding and mask相关的API来实现变长序列的支持。后来仔细研究了一下这个github的代码，发现它并不适合做变长RNN，因为有的组件（指的是TimeGAN模型的组成模块）的输入输出是有依赖的，比如embedding模块的输出会作为recovery模块的输入，这里embedding模块的输入是变长的序列train data，但是对于recovery组件的话它并不知道对应mask的位置（如果要告诉它需要复杂的代码改动），从而对模型的训练带入很多噪声。

正因为上面的这些，我们开始尝试参考<https://github.com/jsyoon0823/TimeGAN/blob/899d02890db46f6be83d139176cc0518120020b1/timegan.py#L84> 这个版本的代码，这个版本的代码使用TF的low level API（即TF session-based API）来实现变长RNN，而且代码中会告诉每个组件使用的RNN的实际sequence长度。

在使用这个版本的代码后，经过很长时间的各种调试，生成序列的效果还是不理想。

TimeGAN生成的序列如下所示（如图中看到的，**连续点击问题太严重**）：

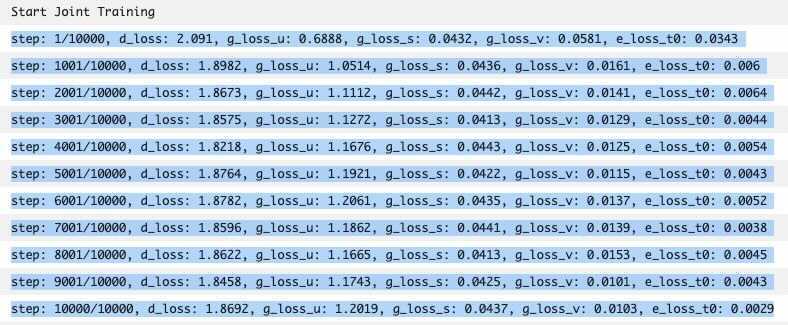


**如何继续？**

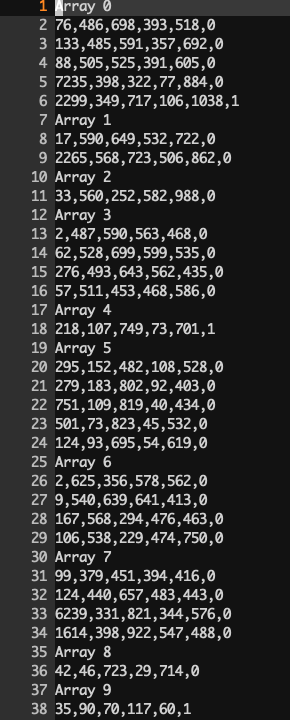
**在选择了TimeGAN模型后，我们花很多时间尝试了各种办法，仍然没有什么靠谱的生成序列，最后决定抱着碰一碰运气的方式来换模型DoppelGANger来做**。

**使用这个模型反而并没有花很多时间调试，生成的序列还可以接受；和TimeGAN生成序列中的严重的连续点击的问题，DoppelGANger基本没有离谱的连续点击问题**。

训练过程中的loss如下所示：



**DoppelGANger**生成的序列如下所示**：**



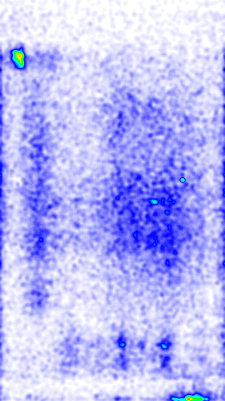
**另外一个痛点：没有什么直观的指标来指示生成的序列的效果如何？**

**不同的模型的开源代码都有一些评估的指标，但是都不是很直观。其实这里我们想要的是生成的序列的分布尽量与原始数据集的分布一致或者相似，也就是判断两个分布的相似性**。

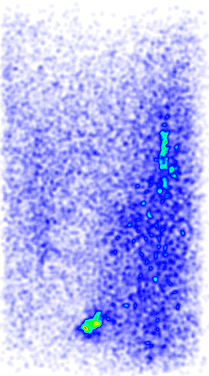
客户最终使用基于python的热力图来对比生成的序列与原始数据的相似性，这样的可视化效果会比较好。

下面分为点击操作和滑屏操作来分别对比（**从下图也可以看出，相对于滑屏操作，点击操作学习起来也要困难一些，因为点击操作本身在数据集中就相对是稀疏的**）：

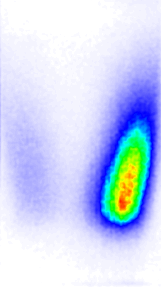
原始数据中的点击操作的热力图：



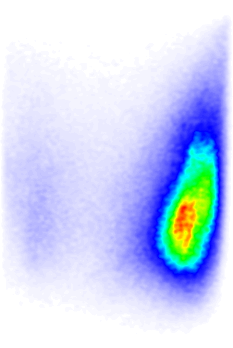
生成数据中的点击操作的热力图：



原始数据中的滑动操作的热力图：



生成数据的滑动操作的热力图：



从上面的图看起来，DoppelGANger生成的序列还算靠谱；和TimeGAN生成的序列的点击操作和滑动操作的热力图对比起来，也是DoppelGANger生成的序列更接近真实序列。

注意：由于时间关系，并没有把数据集中对应的点击操作的touchend和touchstart坐标做修正使得它们是相等的值。如果把它们修正后再进行训练，可能得到的效果会更好。

1. 吸取的教训：
2. 对于比较小众的ML任务，建议还是从最新的模型来尝试。

（DoppelGANger比TimeGAN出现在更新的paper中，应该优先尝试）

1. 开源的代码要尽量先仔细读懂，不太合理的地方可以适当进行改造。

（比如这里我们遇到的TimeGAN代码中引发OOM的代码片段）