因子分解机（FM）简介及实践

因子分解机（Factorization Machines, FM）是一个在2010年被提出的算法，是预估CTR的经典模型之一。在这篇文章里，前半部分会介绍FM的原理，后半部分会通过tensorflow来实现FM算法。由于个人水平十分有限，文章中对算法理解不当的地方，烦请大家指出，不胜感激！

1 FM简介

1.1 提出的动机

我们知道逻辑回归方法是对所有特征的一个线性加权组合，其预测值可以写为如下形式：

$$\hat{y}(x) := w_0 + \sum_{i=1}^{n} w_i x_i. \tag{1}$$

但这样的形式只是单独考虑了每个特征对目标值y的影响，而没有考虑特征之间的关系，比如二阶组合特征$x_i \cdot x_j$或者三阶组合特征$x_i \cdot x_j \cdot x_k$对目标值的影响。现在我们只考虑二阶特征组合的情况，那么逻辑回归表达式可以改写为：

$$\hat{y}(x) := w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} w_{i,j} x_i x_j. \tag{2}$$

写成公式2的形式确实是考虑了二阶特征组合对模型预测的影响，但同时会带来一个新的问题。因为在CTR预估问题里面，或者很多分类、回归问题里面，我们常常需要对类别型特征进行one-hot编码（比如男性、女性分别被编码为01,10），当类别型特征变量较多的情况下，编码后其实是会产生了高维特征，这导致我们很难找到一组x_i和x_j使得$x_i \cdot x_j$不为0。而对于大多数$x_i x_j$乘积为0的项，对应的w_{ij}在梯度更新时$\frac{\partial \hat{y}(x)}{\partial w_{i,j}} = x_i x_j = 0$，也就无法进行梯度更新来求解$w_{ij}$。

而因子分解机的工作就是将$w_{i,j}$替换为两个向量的乘积$v_i v_j$，进而缓解了稀疏特征对模型求解的困难。

1.2 具体内容

仅考虑二阶特征组合的话，FM模型表达式可以写为：

$$\hat{y}(x) := w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} < \mathbf{v}_i, \mathbf{v}_j > x_i x_j. \tag{3}$$

其中$< \mathbf{v}_i, \mathbf{v}_j >$是$k$维向量$\mathbf{v}_i$和$\mathbf{v}_j$的点积，也是对应一个实数值：

$$< \mathbf{v}_i, \mathbf{v}_j > := \sum_{f=1}^{k} v_{i,f} \cdot v_{j,f}. \tag{4}$$

由矩阵分解可知，对任意一个正定矩阵$\mathbf{W}$，都可以找到一个矩阵$\mathbf{V}$，且在矩阵$\mathbf{V}$维度k足够大的情况下使得$\mathbf{W} = \mathbf{V} \cdot \mathbf{V}^t$成立。FM这样写法的精妙之处在于，首先通过矩阵分解用两个向量$\mathbf{v}_i$和$\mathbf{v}_j$的乘积近似原先矩阵$\mathbf{W}$，这保持了变化前后的统一。其次，在拆解为$\mathbf{v}_i$和$\mathbf{v}_j$之后，参数更新时是对这两个向量分别更新的，那么在更新时，对于向量$\mathbf{v}_i$，我们不需要寻找到一组x_i和x_j同时不为0，我们只需要在$x_i \neq 0$的情况下，找到任意一个样本$x_k \neq 0$即可通过$x_i x_k$来更新$\mathbf{v}_i$，也就是说，原先更新参数$w_{i,j}$的条件对于稀疏数据比较苛刻，FM这样的写法缓解了稀疏数据造成无法更新参数的困难。

其实我觉得FM算法的核心思想写到这里就介绍完了，因子分解机中的“因子分解”部分就是替换原先的$w_{i,j}$。而在FM论文里，作者更进一步，将公式3改写，降低FM算法的时间复杂度。对于公式3，我们知道二阶项有两次循环，而$<\mathbf{v}_i, \mathbf{v}_j>$又是经过$k$次计算，因此其时间复杂度为$O(kn^2)$，但通过如下的转换过程，可以将时间复杂度降低为$O(kn)$：

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n} <\mathbf{v}_i, \mathbf{v}_j> x_i x_j &= \frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} <\mathbf{v}_i, \mathbf{v}_j> x_i x_j - \sum_{i=1}^{n} <\mathbf{v}_i, \mathbf{v}_i> x_i x_i\right) \\
&= \frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{f=1}^{k} v_{i,f} v_{j,f} x_i x_j - \sum_{i=1}^{n}\sum_{f=1}^{k} v_{i,f} v_{i,f} x_i x_i\right) \\
&= \frac{1}{2}\sum_{f=1}^{k}\left(\left(\sum_{i=1}^{n} v_{i,f} x_i\right)\left(\sum_{j=1}^{n} v_{j,f} x_j\right) - \sum_{i=1}^{n} v_{i,f}^2 x_i^2\right) \\
&= \frac{1}{2}\sum_{f=1}^{k}\left(\left(\sum_{i=1}^{n} v_{i,f} x_i\right)^2 - \sum_{i=1}^{n} v_{i,f}^2 x_i^2\right).
\end{aligned} \tag{5}$$

需要解释一下的是公式5的第1步，原始求解的就是矩阵上三角之和，那么第1步就是整个矩阵之和减去矩阵的迹，然后求一半。对于稀疏数据中x_i大多数都为0，假设$\overline{m}_D$代表的是对于所有数据$\mathbf{x}$特征不为0个数的平均值，那么FM算法复杂度实际上为$O(k\overline{m}_D)$。

那么二阶FM表达式可以写为：

$$\hat{y}(x) := w_0 + \sum_{i=1}^{n} w_i x_i + \frac{1}{2}\sum_{f=1}^{k}\left(\left(\sum_{i=1}^{n} v_{i,f} x_i\right)^2 - \sum_{i=1}^{n} v_{i,f}^2 x_i^2\right). \tag{6}$$

接下来我们根据公式6计算一下FM算法更新时的梯度。

第一种情况，当参数为w_0时，很简单：

$$\frac{\partial \hat{y}(x)}{\partial w_0} = 1. \tag{7}$$

第二种情况，当参数为w_i时，显然更新w_i只跟和它相关的x_i有关：

$$\frac{\partial \hat{y}(x)}{\partial w_i} = x_i. \tag{8}$$

第三种情况，当参数为$v_{i,f}$时，因为我们所求的是模型对具体的某一个i和具体的某一个f对应的向量$v_{i,f}$产生的梯度，那么对于公式6中，第二项大括号外面的$\sum_{f=1}^{k}$其实是失效的，第二项大括号里面的第二小项$\sum_{i=1}^{n}$也是失效的，即：

$$\begin{aligned}
\frac{\partial \hat{y}(x)}{\partial v_{i,f}} &= \frac{\partial \frac{1}{2}\left(\left(\sum_{i=1}^{n} v_{i,f} x_i\right)^2 - v_{i,f}^2 x_i^2\right)}{\partial v_{i,f}} \\
&= \frac{1}{2}\left(2x_i \sum_{i=1}^{n} v_{i,f} x_i - 2 v_{i,f} x_i^2\right) \\
&= x_i \sum_{j=1}^{n} v_{j,f} x_j - v_{i,f} x_i^2.
\end{aligned} \tag{9}$$

总结来说，FM模型对参数的梯度为：

$$\frac{\partial \hat{y}(x)}{\partial \theta} = \begin{cases} 1, & \text{if } \theta \text{ is } w_0 \\ x_i, & \text{if } \theta \text{ is } w_i \\ x_i \sum_{j=1}^{n} v_{j,f} x_j - v_{i,f} x_i^2. & \text{if } \theta \text{ is } v_{i,f} \end{cases} \tag{10}$$

2 FM算法tensorflow实践

因为刚好要学习一下tensorflow，在这一小节中，我使用tensorflow来实现一下FM算法。其实相比于使用纯python实现，使用tensorflow不需要自己计算对每个参数的导数，框架本身在更新的时候会自动计算每个参数的梯度，这也是使用tensorflow方便的地方。

在这一小节里，为方便起见，我们利用sklearn中自带的boston房价数据集，来实现针对回归任务的FM算法，后续有机会再实现分类任务，我感觉分类任务其实就是加上一个softmax函数。其中FM算法代码如下，读者可以根据实际问题进行功能扩展：

```
# -*- coding: utf-8 -*-
"""
Created on Tue May 26 19:18:32 2020
A simple implementation of Factorization Machines for regression problem
@author: an
"""
import tensorflow as tf
import os

class FM:
    def __init__(self, x, y, learning_rate = 1e-6, batch_size = 16, epoch =
100):
        self.x = x
        self.y = y
        self.sam_num = x.shape[0]
        self.fea_num = x.shape[1]
        self.learning_rate = learning_rate
        self.epoch = epoch
        # the size of the vector v
        self.inner_size = self.fea_num // 2

        if batch_size <= self.sam_num:
            self.batch_size = batch_size
        else:
            # modify the batch size according to the times between batch_size
and sam_num
            self.batch_size = max(2, batch_size // -(-batch_size//self.sam_num))

    def fit(self):
        x_input = tf.placeholder(tf.float32, shape=[None, self.fea_num], name =
"x_input")
        y_input = tf.placeholder(tf.float32, shape=[None], name="y_input")
        biases = tf.Variable(tf.zeros([1]), name="biases")
        linear_weights = tf.Variable(tf.random_uniform(shape=[self.fea_num, 1],
minval = -1, maxval = 1),name="linear_weights")
        second_order_weights = tf.Variable(tf.random_uniform(shape=
[self.fea_num, self.inner_size], minval = -1, maxval = 1),
name="second_order_weights")

        part1 = biases
        part2 = tf.reduce_mean(tf.matmul(x_input, linear_weights),0)
        part3 = 0.5 * tf.reduce_sum(tf.square(tf.matmul(x_input,
second_order_weights)) -  tf.matmul(tf.square(x_input),
tf.square(second_order_weights)), 1)
        y_predicted = tf.add(tf.add(part1, part2), part3)
        loss = tf.reduce_mean(tf.square(y_predicted - y_input), 0)
```

```
        optimizer = tf.train.AdamOptimizer(learning_rate = self.learning_rate)
        train_op = optimizer.minimize(loss)

        saver = tf.train.Saver(max_to_keep = 1)
        tf.add_to_collection('y_p', y_predicted)
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            for i in range(self.epoch):
                for batch_x, batch_y in self._batch():
                    sess.run(train_op, feed_dict = {x_input:batch_x,
y_input:batch_y})
                if i % 100 == 0:
                    train_loss = sess.run(loss, feed_dict = {x_input: self.x,
y_input: self.y})
                    print("epoch" + str(i) + "_loss: ", int(train_loss))
                    saver.save(sess, "models/")
            sess.close()

    def predict(self, x_test):
        if not os.path.exists("models/"):
            print("Please train the model before test!")
            return

        with tf.Session() as sess:
            saver = tf.train.import_meta_graph('models/.meta')
            saver.restore(sess, tf.train.latest_checkpoint("models/"))
            y_ = tf.get_collection('y_p')
            print(sess.run(y_, feed_dict = {"x_input:0": x_test}))

    def _batch(self):
        for i in range(0, self.sam_num, self.batch_size):
            upper_bound = min(i + self.batch_size, self.sam_num)
            batch_x = self.x[i:upper_bound]
            batch_y = self.y[i:upper_bound]
            yield batch_x, batch_y
```

在这里我对每个函数做下解释：

1. **__init__**：初始化FM模型的参数，并且保证batch_size不会超过样本数量。
2. **fit**：按照设定的epoch进行模型训练，并在最后保存模型。
3. **predict**：加载之前的模型，并实现预测功能。
4. **_batch**：将训练数据按照batch_size大小，划分成不同的批次。

经过1000个epoch的训练，模型在全体训练集上的MSE损失变化如下：

```
epoch0_loss:  629
epoch100_loss:  177
epoch200_loss:  158
epoch300_loss:  158
epoch400_loss:  159
epoch500_loss:  156
epoch600_loss:  148
epoch700_loss:  138
epoch800_loss:  128
epoch900_loss:  119
```

以上代码基于tensorflow 1.10.0、python 3.6。在写tensorflow的时候要注意的地方还是挺多的，模型的保存和加载就是一个需要关注的点。

完整代码以及本文的pdf版本放在了我的github里，欢迎来看！

参考

1. Factorization Machines
2. 推荐系统系列（一）：FM理论与实践
3. 推荐系统召回四模型之：全能的FM模型