16 级计科 7 班: Kaggle 实验报告

Due on Friday, April 26, 2018

潘嵘 周五 7-8 节

颜彬

16337269

Content

1 实验环境

1.1 环境简介

本实验所使用的编程语言是 Python 3.7.1。实验环境是 macOS Mojave(10.14.4)。由于 Python 语言的跨平台特性，本实验的代码理应能运行在任何的平台之上。

1.2 环境安装

假设系统已预装 conda。使用代码 1所示的方式安装环境。

1.3 运行方法

首先进入 src 目录。使用 python main.py 来训练模型，并使用 python predict.py 来对测试集作预测。

trainingdata.py 文件调用了 processor.py 和 util.py，该文件的最终作用是返回一个已经经过预处理和特征工程的输入。

processor.py 提供了一系列预处理和进行特征工程的函数。它引用了 util.py。util.py 定义了一系列辅助函数。

代码 1: 安装环境的方式

```
cd AngelsAndDemons
conda create -f environment.yaml
```

2 代码介绍

2.1 流程简介

在本次项目中，首先需要读取数据，然后按小节 2.2所介绍的步骤进行数据预处理和特征工程。最后按照小节 2.3所介绍的步骤进行模型训练。

数据预处理指的是对不符合要求的数据的预先处理，使其符合模型的输入要求。例如大多数模型都要求输入是一个数值矩阵。预处理需要将训练集中所有的数据类型都转换成数值类型，同时为空值和 NaN 值填补上缺省值。

特征工程指的是从已有的特征中挖掘出新的特征。例如假设特征满足 *yyyy-mm-dd-ss.000000* 的模式。显然如果将该特征按 “-’’ 和 “:” 将特征分成 5 个部分时，前四个部分分别表示年、月、日和秒，而第五个部分永远为 000000。这样，新的 4 个特征就从这个旧的特征中挖掘出来了。

模型训练中，采用了 sci-kit learn 和 lightgbm 两个库进行训练。

2.2 数据预处理和特征工程

文件 src/processor.py 中定义了一系列函数，用于进行数据的预处理。整个预处理的过程就是这些函数的一个排列和组合。

2.2.1 填补空值

对于输入中的数值类型，可以直接采用所有特征的均值来填补空值。但是注意到输入特征中有大部分特征的类型是字符串。我们无法为字符串定义均值，故采用字符串出现次数最多的一类来填补空值。

如代码 2所示。当 mean 无法计算成功时，会抛出异常，改为使用出现次数最多的字符串。

代码 2: 填补空值的过程

```
def fillEmpty(data):
    try:
        mean = data[col].fillna(0).mean()
        print('column {} fill to mean {}'.format(col, mean))
        data[col].fillna(mean, inplace=True)
    except:
        mean = data[col].value_counts().index[0]
        data[col].fillna(mean, inplace=True)
```

2.2.2 分解特征

第 209 列和 247 列是十分特殊的两个列。特殊点在于，他们的类型都是字符串，且测试集中出现了训练集里没出现过的值。

这暗示了，这两列的值是具有含义的，所以他们很可能是难以枚举的。

分解特征的处理方式是，单独取出这两列，使用 split 函数将字符串分割，然后为数据集加上分割后得到的内容。同时把这两列从数据集中删掉。

以第 209 列为例 (特征的含义是日期)，如代码 3所示。

2.2.3 字符串转整数

字符串是没有办法被后续模型所处理的。一个简单的方法是将字符串 s 使用 f 映射成整数 i，且满足

- 当 $s = s'$ 时，$f(s) = f(s')$
- 当 $s \neq s'$ 时，$f(s) \neq f(s')$

这样的优点是把所有的字符串转换成了一个整数，且该整数能一定程度上代表字符串的特征。

这样的缺点也很明显，就是数值之间的大小关系并不反映两个例子之间的关系。例如字符串 s_1, s_2, s_3 被映射到了 1, 2, 10。并不代表 s_1 和 s_2 更接近，也不代表 s_1 和 s_3 更远。

代码如代码 4所示。当一个字符串首次出现时，递增地为其分配一个整数。当这个字符串第二次出现时，返回相同的整数。

代码 3: 对 209 列的特征工程处理

```
def split209Datatime(dataset):
    for df in dataset:
        dt_col = df[209] # the 209th column is to be split

        def getYear(s):
            return int(s.split('-')[0])
        def getMonth(s):
            # print('s is {}, split is {}'.format(s, s.split('-')[1]))
            return int(s.split('-')[1])
        def getDay(s):
            return int(s.split('-')[2])
        def getHour(s):
            return int(s.split('-')[3].split('.')[0])
        def getMin(s):
            return int(s.split('-')[3].split('.')[1])
        def getSec(s):
            return int(s.split('-')[3].split('.')[2])

        df['year'] = dt_col.apply(getYear)
        df['month'] = dt_col.apply(getMonth)
        df['day'] = dt_col.apply(getDay)
        df['hour'] = dt_col.apply(getHour)
        df['minute'] = dt_col.apply(getMin)
        df['second'] = dt_col.apply(getSec)

        df.drop([209], 'columns', inplace=True)
```

2.2.4 删去 id 和删去等值特征

id 这个列应该被删去，否则很容易产生过拟合现象。这是因为，使用 id 来直接分类，必定能达到训练集分类准确度 100% 的效果，但这没有任何意义。

等值特征是指训练集和测试集中这个特征的每个样例的取值都相等的特征。例如有某些特征，所有样例的取值都是 0。这个特征在训练中是没有意义的。再例如，节 2.2.2中提取了新的特征 “年份”。但实际上提取后才发现，所有样例的年份的取值都是 2009。这个特征也是毫无意义的。

删去等值特征的方法是采用 describe 函数。由于节 2.2.3 已经将所有的字符串转换成整数了，如果一个特征是等值特征，那么必定每个样例在该特征下最小值会等于最大值。使用 pandas.describe 函数得到每个特征的最小值和最大值。删除最小值等于最大值的特征。

2.2.5 数据标准化

通过 describe 函数也发现了，数据的均值和方差差别很大。例如有一列的数量级在 $10^10 - 10^11$，但是它的 range 却并不大。这提示了我们可以通过对数据标准化来整理数据。

使用代码 6的方法来标准化。值得注意的是，由于有些特征的值比较大，在直接运行 scale 函数时，会报错。所以首先先进行 range 标准化，把数据拉伸到 0-1 的范围内，再进行 scale。

代码 4: 字符串转整数的代码

```
def string2int(dataset):
    '''
    string2int convert string fields in df to int.
    '''
    print('convert all string fields to int ...')
    s = strColumns(dataset[0])
    for colIdx in s:
        names = {}
        last = 0
        partial = dataset[0][colIdx]
        for name in partial:
            if name not in names:
                names[name] = last
                last += 1

        def convertor(name):
            try:
                return int(names[name])
            except:
                print('[WARNING] {} not in converting dict'.format(name))
                return -1
        for df in dataset:
            df[colIdx] = df[colIdx].apply(convertor)
```

代码 5: 删去 id 和删去等值特征

```
def dropId(dataset):
    for df in dataset:
        df.drop([0], 'columns', inplace=True)


def filterAllSameCols(dataset):
    azc = allSameCol(dataset[0])
    for data in dataset:
        azc_cur = allSameCol(data)
        azc = np.logical_and(azc, azc_cur)
    azci = azc.index[azc] # a list of index that azc is true
    for data in dataset:
        data.drop(azci, 'columns', inplace=True)
```

2.3 模型训练

本实验主要采用 lightGBM 模型来进行训练，并取得了比较好的拟合效果。在 public 榜单上超过 2 天获得 auc 为 0.98652 的成绩 (rank 5)。

代码 6: 数据标准化

```
def scaleToStandard(dataset):
    print('stardardize to 0-mean and 1-var ...')
    ret = []
    for idx, df in enumerate(dataset):
        val = df.values
        # scale to [0, 1]
        min_max_scaler = preprocessing.MinMaxScaler()
        val_range = min_max_scaler.fit_transform(val)

        # scale to 0-means and 1-std
        val_scaled = preprocessing.scale(val_range)
        dataset[idx] = pd.DataFrame(val_scaled, columns=df.columns)
```

3 相关解释

lightGBM 是 GBDT 的一个更优良的实现。

GBDT(Gradient Boosting Decision Tree) 是机器学习中的一个著名的模型。它的主要思想为训练出若干个弱分类器，给予弱分类器一定的权重，通过某种方式综合弱分类器的决策，产生最终的结果。这样，这些弱分类器就成为了强分类器。

lightGBM(light Gradient Boosting Machin) 是一个实现了 GBDT 算法的框架。它具有更快的训练速度，更低的内存消耗和更好的准确率等优势。

3.1 过拟合的防止

过拟合的防止方式有很多种。

3.1.1 验证集

可以采用验证集的方式，让 lightGBM 在发现无法在验证集上获得更好的效果时，及时停止训练 (early stop)。

如代码 7所示。按照一定的比例划分出 train_index 和 test_index，然后根据 test_index 得到验证集。把验证集给予 lightGBM，让其决定训练程度。

3.1.2 调整验证集大小

验证集如果太小，很可能模型发生了过拟合却无法及时发现。这在我的实验中发生地比较明显。我的本地训练集达到了 0.988 的 auc，但是在 kaggle 的公榜上却仅获得了 0.984 的 auc 值。

验证集如果很大，会导致训练集过小，训练结果也不佳。当我把验证集取到 30%-40% 时，会发现本地的 auc 和 kaggle 公榜的 auc 比较接近，但均不能超过 0.984。

最终我确定了验证集为 10%，它能在保证训练集较多的情况下，留出一定的验证集。这个设定让我获得了榜单的最好成绩。

代码 7: 验证集防止过拟合

```
def lightgbm_clissify(trainX, trainY, testX, sss):
    times = 0
    auc = 0
    for train_index, test_index in sss.split(trainX, trainY):
        sub_X_train, sub_X_test = trainX[train_index], trainX[test_index]
        sub_Y_train, sub_Y_test = trainY[train_index], trainY[test_index]

        lgb_train = lgb.Dataset(sub_X_train, sub_Y_train)
        lgb_eval = lgb.Dataset(sub_X_test, sub_Y_test, reference=lgb_train)

        params = {
            # ...
        }

        gbm = lgb.train(
            params,
            lgb_train,
            learning_rates=lambda iter: 0.1 * (0.995 ** (iter / 10)),
            num_boost_round=3000,
            valid_sets=lgb_eval,
            early_stopping_rounds=100)
        # ...
```

3.1.3 叶子结点数

叶子结点数太小时，会导致欠拟合。但如果叶子节点数太大，会在发生严重的过拟合现象。经过试验发现，叶子结点数基本决定了最终结果的好坏。

模型最后为每个子树采用了 160 个叶子结点。获得了榜单的最好成绩。