



# 研究生《人工智能基础与应用》 实验报告

实验名称:	<u>Kddcup 99 数据集分类</u>
姓 名:	<u></u>
学 号:	<u></u>
日 期:	<u>2020.12.30</u>

## 一、实验内容

**实验要求：**完成 kddcup 99 数据集的分类，方法可选用课程方法之一，如决策树 DT、支持向量机 SVM 等。

**实验目的：**学会使用 python 语言及其相应的库对数据集进行读取、处理和写入等操作，然后学会使用 DT、SVM 等算法对数据集分类。

**算法：**决策树(Decision Tree)、支持向量机 (Support Vector Machine, SVM)

**原理：**决策树：以信息熵为度量构造一棵熵值下降最快的树，到叶子节点处熵值为 0，每个内部结点表示在一个属性上的测试，每个分支代表一个测试输出，每个叶结点代表一种类别。支持向量机：先求解能够正确划分训练数据集并且几何间隔最大的分离超平面，分离超平面间隔最大化可转化为一个凸二次规划问题求解，最后利用求解得到的分离超平面对数据进行划分。

## 二、实验设计

**kddcup99 数据集的分类过程主要分三步完成：**

**第一步：数据数值化**

目的是将 kddcup99 数据集中的字符型特征或标签转换为数值型表示。

方法是将字符型特征排序，采用字符型特征的下标表示该字符型特征。

**第二步：数据标准化**

目的是应对特征向量中数据很分散的情况，防止小数吃大数的情况，同时也可以加速训练。

方法是采用 Z-score 标准化。假设该数据集的分布近似服从高斯分布，基于数据的均值和方差进行标准化，标准化公式如下：

$$x' = \frac{x - \bar{x}}{\sigma}$$

**第三步：模型训练、预测并输出分类报告**

采用数值化和标准化处理后的数据集，进行 SVM 算法分类并输出混淆矩阵和分类报告，从精确率：precision、召回率：recall、调和平均 f1 值:f1-score 和支持度:support 四个维度评价分类预测效果。

### 三、实验环境及实验数据集

#### 1. 实验环境配置：python 和 package 的版本

python: 3.6.12

csv: 1.0

numpy: 1.19.2

pandas: 1.1.5

scikit-learn: 0.23.2

IPython: 7.16.1

Pytorch: 1.4.0

#### 2. 数据集：kdd cup99 数据集

### 四、实验过程

#### 第一步：

##### 1) 导入所需的包

```
1 import numpy as np
2 import csv
```

##### 2) 定义用索引替代字符型特征或标签的函数

```
1 # 取字符对应的索引表示该字符
2 def find_index(x, y):
3     return [i for i in range(len(y)) if y[i]==x] # Python列表解析, 返回列表
```

##### 3) 定义将原数据集中 3 种协议类型转换成数字标识的函数

```
1 def handleProtocol(inputs):
2     protocol_list=['tcp', 'udp', 'icmp']
3     if inputs[1] in protocol_list:
4         return find_index(inputs[1], protocol_list)[0]
```

##### 4) 定义将原数据集中 70 种网络服务类型转换成数字标识的函数

```
1 def handleService(inputs):
2     service_list=['aol', 'auth', 'bgp', 'courier', 'csnet_ns', 'ctf', 'daytime', 'discard', 'domain', 'domain_u',
3 'echo', 'eco_i', 'ecr_i', 'efs', 'exec', 'finger', 'ftp', 'ftp_data', 'gopher', 'harvest', 'hostnames',
4 'http', 'http_2784', 'http_443', 'http_8001', 'imap4', 'IRC', 'iso_tsap', 'klogin', 'kshell', 'ldap',
5 'link', 'login', 'mtp', 'name', 'netbios_dgm', 'netbios_ns', 'netbios_ssn', 'netstat', 'nnsdp', 'nntp',
6 'ntp_u', 'other', 'pm_dump', 'pop_2', 'pop_3', 'printer', 'private', 'red_i', 'remote_job', 'rje', 'shell',
7 'smtp', 'sql_net', 'ssh', 'sunrpc', 'supdup', 'systat', 'telnet', 'tftp_u', 'tim_i', 'time', 'urh_i', 'urp_i',
8 'uucp', 'uucp_path', 'vmnet', 'whois', 'X11', 'Z39_50']
9     if inputs[2] in service_list:
10         return find_index(inputs[2], service_list)[0]
```

##### 5) 定义将原数据集中 11 种网络连接状态转换成数字标识的函数

```

1 def handleFlag(inputs):
2     flag_list=['OTH','REJ','RSTO','RSTOSO','RSTR','SO','S1','S2','S3','SF','SH']
3     if inputs[3] in flag_list:
4         return find_index(inputs[3],flag_list)[0]

```

## 6) 定义将原数据集中 11 种网络连接状态转换成数字标识的函数

```

1 def handleLabel(inputs):
2     label_list=['normal.','buffer_overflow.','loadmodule.','perl.','neptune.','smurf.','
3         'guess_passwd.','pod.','teardrop.','portsweep.','ipsweep.','land.','ftp_write.','
4         'back.','imap.','satan.','phf.','nmap.','multihop.','warezmaster.','warezclient.','
5         'spy.','rootkit.'].
6     #在函数内部使用全局变量并修改它
7     if inputs[41] in label_list:
8         return find_index(inputs[41],label_list)[0]
9     else:
10        label_list.append(inputs[41])
11        return find_index(inputs[41],label_list)[0]

```

## 7) 读取数据集、数值化处理、写入文件

```

global label_list # 定义label_list为全局变量,
# 文件写入
data_numerization = open("kddcup.data.numerization.txt", 'w', newline='') # 新建文件用于存放数值化后的数据集
if __name__ == '__main__':
    with open('kddcup.data.original.txt','r') as data_original: # 打开原始数据集文件
        csv_reader = csv.reader(data_original) # 按行读取所有数据并返回由csv文件的每行组成的列表
        csv_writer = csv.writer(data_numerization, dialect='excel') # 先传入文件句柄
        for row in csv_reader: # 循环读取数据
            temp_line=np.array(row) # 将列表list转换为ndarray数组。
            temp_line[1] = handleProtocol(row) # 将源文件行中3种协议类型转换成数字标识
            temp_line[2] = handleService(row) # 将源文件行中70种网络服务类型转换成数字标识
            temp_line[3] = handleFlag(row) # 将源文件行中11种网络连接状态转换成数字标识
            temp_line[41] = handleLabel(row) # 将源文件行中23种攻击类型转换成数字标识
            csv_writer.writerow(temp_line) # 按行写入
        data_numerization.close()
    print('数值化done!')

```

## 第二步:

### 1) 导入所需的包

### 2) 读取数据集

```

1 global x_data # 定义全局变量
2 begin_time = time() # 读取文件开始时间
3 data_numerization = open("kddcup.data.numerization_corrected.txt")
4 lines = data_numerization.readlines()
5 line_nums = len(lines)
6 x_data = np.zeros((line_nums, 42)) # 创建line_nums行 para_num列的矩阵
7 for i in range(line_nums):
8     line = lines[i].strip().split(',')
9     x_data[i, :] = line[0:42] # 获取42个特征
10 data_numerization.close()
11 print('数据集大小: ', x_data.shape)
12
13 # 耗时分析
14 end_time = time() # 读取文件结束时间
15 total_time = end_time-begin_time # 读取文件耗时
16 print('读取文件耗时: ', total_time, 's')

```

### 3) 指定 0 号 GPU

```

1 import torch
2 from torch import nn
3 print(torch.cuda.is_available())      # 查看GPU是否可用
4 print(torch.cuda.device_count())     # GPU数量, 1
5 print(torch.cuda.current_device())   # 当前GPU的索引
6 print(torch.cuda.get_device_name(0)) # 输出GPU名称
7
8 device = torch.device('cuda:0')     # 指定device为0号GPU, 若使用CPU则填写"cpu"

```

#### 4) 定义数据标准化的函数

```

1 # 在GPU上并行加速, 并利用pytorch的Tensor的广播机制做矩阵计算
2 def Zscore_Normalization(x, n):
3     if np.std(x) == 0:
4         x_data[:, n] = 0
5     else:
6         mean = torch.tensor(np.mean(x), device='cuda:0')
7         std = torch.tensor(np.std(x), device='cuda:0')
8         x = torch.tensor(x, device='cuda:0').view(-1,1)
9         x_data[:, n] = ((x - mean) / std).cpu().numpy().T
10    print("The ", n, "feature is normalizing.")

```

#### 5) 数据标准化处理并分析耗时

```

1 begin_time = time()                    # 标准化开始时间
2 for i in range(42):
3     Zscore_Normalization(x_data[:, i], i)
4
5 # 耗时分析
6 end_time = time()                     # 标准化结束时间
7 total_time = end_time - begin_time    # 标准化耗时
8 print(' 标准化耗时: ', total_time, 's')

```

#### 6) 将标准化后的数据集写入文件

```

1 data_normalizing = open("kddcup.data.numerization_corrected_normalizing_GPU.txt", 'w', newline='')
2 csv_writer = csv.writer(data_normalizing)
3 i = 0
4 while i < len(x_data[:, 0]):
5     csv_writer.writerow(x_data[i, :])
6     i = i + 1
7 data_normalizing.close()
8 print(' 数据标准化done! ')

```

### 第三步:

#### 1) 导入所需的包

#### 2) 读取数据集

如果在这一步调用 sklearn 包的库函数 StandardScaler()对数据进行标准化,则可以省略第二步。我在这里设置了选择采用全部特征进行训练和选择 3,4,5,6,8,10,13,23,24,37 这 10 个特征进行训练两种方式(下图红框所示)。



```

1 fr=open("kddcup.data.numerization_corrected_normalizing_StandardScaler.txt") # 打开数值化、修正后的数据集
2 data = fr.readlines() # 读取所有行(直到结束符EOF)并返回列表
3 line_nums = len(data)
4 # data_feature = np.zeros((line_nums, 41)) # 创建line_nums行 41列的矩阵
5 data_feature = np.zeros((line_nums, 10)) # 创建line_nums行 10列的矩阵
6 data_labels = []
7 for i in range(line_nums): # 依次读取每行
8     line = data[i].strip().split(',') # 去掉每行头尾空白, 分隔符对字符串进行切片
9     # data_feature[i, :] = line[0:41] # 选择前41个特征 划分数据集特征和标签
10    feature = [3, 4, 5, 6, 8, 10, 13, 23, 24, 37] # 选择第3, 4, 5, 6, 8, 10, 13, 23, 24, 37这10个特征分类
11    for j in feature:
12        data_feature[i, feature.index(j)] = line[j]
13    data_labels.append(line[-1]) # 标签
14 fr.close() # 关闭文件
15
16 data_feature = StandardScaler().fit_transform(data_feature) # 标准化, 利用Sklearn库的StandardScaler实现数据标准化
17 data_labels = StandardScaler().fit_transform(np.array(data_labels).reshape(-1, 1))
18 # data_feature = MinMaxScaler().fit_transform(data_feature) # 归一化: 利用Sklearn库的MinMaxScaler实现数据归一化, 返回[0, 1]区间的数据
19 # data_labels = MinMaxScaler().fit_transform(np.array(data_labels).reshape(-1, 1))
20
21 print('数据集特征大小: ', data_feature.shape)
22 print('数据集标签大小: ', len(data_labels))

```

### 3) 划分训练集和测试集

利用 sklearn 包的 train\_test\_split 模块对划分训练集和测试集。

```

1 data_label = []
2 for i in data_labels:
3     data_label.append(int(float(i)))
4 data_label = np.array(data_label, dtype = int) # list转换数组
5 train_feature, test_feature, train_label, test_label = train_test_split(data_feature, data_label, test_size=0.4, random_state=4) # 测试集40%
6 print('训练集特征大小: {}, 训练集标签大小: {}'.format(train_feature.shape, train_label.shape))
7 print('测试集特征大小: {}, 测试集标签大小: {}'.format(test_feature.shape, test_label.shape))

```

### 4) 模型训练、预测

方法一：采用决策树 DT 算法对数据集分类，使用基尼系数 gini 选择特征，即采用 CART 算法，决策树最大深度 20。

```

1 begin_time = time() # 训练预测开始时间
2 if __name__ == '__main__':
3     print('Start training DT: ', end='')
4     dt = sklearn.tree.DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=20, min_samples_split=2, min_samples_leaf=1)
5     dt.fit(train_feature, train_label)
6     print(dt)
7     print('Training done! ')
8
9     print('Start prediction DT: ')
10    test_predict = dt.predict(test_feature)
11    print('Prediction done! ')
12
13    print('预测结果: ', test_predict)
14    print('实际结果: ', test_label)
15    print('正确预测的数量: ', sum(test_predict==test_label))
16 end_time = time() # 训练预测结束时间
17 total_time = end_time - begin_time
18 print('训练预测耗时: ', total_time, 's')

```

方法二：采用支持向量机 SVM 算法对数据集分类，选择高斯核函数 rbf，核函数系数 0.5，迭代精度 1e-2。

```

1 begin_time = time() # 训练预测开始时间
2 if __name__ == '__main__':
3     print('Start training SVM: ',end='')
4     svm = sklearn.svm.SVC(kernel='rbf', C=1, gamma=0.5, tol=1e-2) # 选择高斯核函数rbf, 正则化系数为1, 核函数系数0.5, SMO迭代精度1e-2
5     svm.fit(train_feature, train_label) # 开始训练SVM
6     print(svm)
7     print('Training done! ')
8
9     print('Start prediction SVM: ')
10    test_predict = svm.predict(test_feature) # 对测试集进行类别预测
11    print('Prediction done! ')
12
13    print(' 预测结果: ',test_predict)
14    print(' 实际结果: ',test_label)
15    print(' 正确预测的数量: ',sum(test_predict==test_label))
16    end_time = time() # 训练预测结束时间
17    total_time = end_time - begin_time
18    print(' 训练预测耗时: ',total_time,'s')

```

## 5) 输出分类报告

```

1 print(' 准确率:', metrics.accuracy_score(test_label, test_predict)) # 预测准确率输出
2 print(' 宏平均精确率:', metrics.precision_score(test_label, test_predict, average='macro')) # 预测宏平均精确率输出
3 print(' 微平均精确率:', metrics.precision_score(test_label, test_predict, average='micro')) # 预测微平均精确率输出
4 print(' 宏平均召回率:', metrics.recall_score(test_label, test_predict, average='macro')) # 预测宏平均召回率输出
5 print(' 平均F1-score:', metrics.f1_score(test_label, test_predict, average='weighted')) # 预测平均f1-score输出
6 print(' 混淆矩阵输出:', metrics.confusion_matrix(test_label, test_predict)) # 混淆矩阵输出
7
8 # 从精确率:precision、召回率:recall、调和平均f1值:f1-score和支持度:support四个维度进行衡量
9 print(' 分类报告:', metrics.classification_report(test_label, test_predict)) # 分类报告输出

```

## 五、实验结果

当选择所有特征进行分类时,采用决策树的 CART 算法训练模型后的预测结果准确率达到 0.9999 以上,如下图所示。混淆矩阵对角线元素的值最大,非对角线元素的值基本为 0,说明正确预测的数量很大,错误预测的数量可以忽略不计。当选择第 3,4,5,6,8,10,13,23,24,37 这 10 个特征分类时准确率达到 0.994 以上,这说明并不需要选择全部的特征就可以实现 kdd 数据集的分类,从而减小计算量。

准确率: 0.9999101753010659  
 宏平均精确率: 0.8371495825320061  
 微平均精确率: 0.9999101753010659  
 宏平均召回率: 0.835555771621358  
 平均F1-score: 0.9999090457658912  
 混淆矩阵输出:

```

[[ 388599      11      1      8      4      6      6      0      4
      4 1552588      0      0      2      0      1      0      0
      3      0    532      0      0      0      0      0      0
      16      2      0   9049      0      0      1      4      0
      4      0      0      0      4      0      0      0      0
      3      0      0      0      0      875      0      0      0
      20      1      0      2      0      0    6273      0      0
      16      3      0      8      0      0      0    908      0
      40      0      0      0      0      1      0      0    368
      4      0      0      0      0      0      0      0      0
      0]]

```

分类报告:

	precision	recall	f1-score	support
-1	1.00	1.00	1.00	388640
0	1.00	1.00	1.00	1552595
1	1.00	0.99	1.00	535
2	1.00	1.00	1.00	9072
3	0.40	0.50	0.44	878
4	0.99	1.00	0.99	878
5	1.00	1.00	1.00	6296
6	1.00	0.97	0.98	935
7	0.99	0.90	0.94	409
8	0.00	0.00	0.00	4
accuracy			1.00	1959372
macro avg	0.84	0.84	0.84	1959372
weighted avg	1.00	1.00	1.00	1959372

图 1 混淆矩阵

图 2 分类报告





```

1 # 在上CPU, 循环计算
2 def Zscore_Normalization(x, n):
3     if np.std(x) == 0:
4         x_data[:, n] = 0
5     else:
6         i = 0
7         while i < len(x):
8             x_data[i][n] = (x[i] - np.mean(x)) / np.std(x)
9             i = i + 1
10    print("The ", n, "feature is normalizing.")

```

## 方案二:

因为方案一的数据标准化计算太慢, 所以想到了一个办法, 利用 GPU 做加速并行计算, 函数如下图所示。利用 Pytorch 框架实现 GPU 计算功能, 直接在 GPU 上创建 Tensor, 计算完成后再将结果转移到 CPU 上, 并将 Tensor 转化为 numpy 数组, 再赋给数据集中相对应的特征。

```

1 # 在GPU上并行加速, 循环计算
2 def Zscore_Normalization(x, n):
3     if np.std(x) == 0:
4         x_data[:, n] = 0
5     else:
6         mean = torch.tensor(np.mean(x), device='cuda:0')
7         std = torch.tensor(np.std(x), device='cuda:0')
8         x = torch.tensor(x, device='cuda:0')
9         i = 0
10        while i < len(x):
11            x_data[i][n] = ((x[i] - mean) / std).cpu().numpy()
12            i = i + 1
13    print("The ", n, "feature is normalizing.")

```

## 方案三:

因为方案二数据标准化的速度没有提高多少, 分析原因是因为循环赋值非常耗时。所以又改进了标准化函数, 在 GPU 上并行加速, 并且利用 pytorch 的 Tensor 的广播机制做矩阵计算, 函数如下图。

```

1 # 在GPU上并行加速, 并利用pytorch的Tensor的广播机制做矩阵计算
2 def Zscore_Normalization(x, n):
3     if np.std(x) == 0:
4         x_data[:, n] = 0
5     else:
6         mean = torch.tensor(np.mean(x), device='cuda:0')
7         std = torch.tensor(np.std(x), device='cuda:0')
8         x = torch.tensor(x, device='cuda:0').view(-1, 1)
9         x_data[:, n] = ((x - mean) / std).cpu().numpy().T

```

采用方案三标准化整个数据集耗时大约 16.6s (如下图所示), 速度提高 1000 倍以上。

```

1 begin_time = time() # 标准化开始时间
2 for i in range(42):
3     Zscore_Normalization(x_data[:, i], i)
4
5 # 耗时分析
6 end_time = time() # 标准化结束时间
7 total_time = end_time - begin_time # 标准化耗时
8 print(' 标准化耗时: ', total_time, 's')

```

标准化耗时: 16.6126127243042 s

#### 方案四：

在查广播机制的时候发现，发现 numpy 数组计算也有广播机制的功能。然后在 CPU 上，利用 numpy 数组计算的广播功能做矩阵计算，对比一下。

```
1 # 在上CPU，并利用numpy的ndarray数组的广播机制做矩阵计算
2 def Zscore_Normalization(x, n):
3     if np.std(x) == 0:
4         x_data[:, n] = 0
5     else:
6         x_data[:, n] = (x - np.mean(x)) / np.std(x)
```

采用方案四标准化整个数据集耗时大约 14.1s（如下图所示），比方案三还快 2s 多。分析其原因可能是数据从 CPU 转移到 GPU，然后又从 GPU 转移到 CPU 会有很大的开销，而 GPU 的专长矩阵计算优势在这小规模计算中无法明显体现出来。

```
1 begin_time = time() # 标准化开始时间
2 for i in range(42):
3     Zscore_Normalization(x_data[:, i], i)
4
5 # 耗时分析
6 end_time = time() # 标准化结束时间
7 total_time = end_time - begin_time # 标准化耗时
8 print(' 标准化耗时: ', total_time, 's')
```

标准化耗时: 14.087749719619751 s

#### 方案五：

比较坑的是，我突然发现 sklearn 库中有专门做数据集特征处理的包，使用 sklearn.preprocessing 的 StandardScaler 一行代码就可以完成数据处理了……而且速度更快，标准化整个数据集耗时大约 7.9s，如下图所示。

```
1 # 利用Sklearn库的StandardScaler实现数据标准化
2 begin_time = time() # 标准化开始时间
3 x_data = StandardScaler().fit_transform(x_data) # 标准化，返回值为标准化后的数据
4
5 # 耗时分析
6 end_time = time() # 标准化结束时间
7 total_time = end_time - begin_time # 标准化耗时
8 print(' 标准化耗时: ', total_time, 's')
```

标准化耗时: 7.943350076675415 s

### 3. 读取数据、数值化、删除错误行、标准化、归一化，写进一个程序

这里重新写了一段数据集处理的程序，比之前的更加简单高效，直接读取原始数据集，统一处理后就可以直接用于训练。这里采用 pandas 包的 read\_csv

读取数据集，并设置参数 `error_bad_lines=False` 来删除错误的那一行数据。然后采用 `sklearn` 包中 `preprocessing` 类的 `LabelEncoder` 和 `OrdinalEncoder` 分别对字符型标签和特征进行编码。统一编码后采用 `StandardScaler().fit_transform()` 标准化数据集，或者采用 `MinMaxScaler().fit_transform()` 归一化数据集。然后再循环提取数据集特征和标签，可以选择使用全部特征训练，也可以选择第 3,4,5,6,8,10,13,23,24,37 这 10 个特征训练。

```
1 # 定义读取、处理数据集函数
2 def data_processing(file, all_features=True):
3     fr = pd.read_csv(file, encoding='utf-8', error_bad_lines=False, nrows=None)
4     data = np.array(fr)
5     print('数据集大小: ', data.shape)
6
7     data[:, -1] = LabelEncoder().fit_transform(data[:, -1]) # 标签的编码
8     data[:, 0:-1] = OrdinalEncoder().fit_transform(data[:, 0:-1]) # 特征的分类编码
9     data = StandardScaler().fit_transform(data) # 标准化: 利用Sklearn库的StandardScaler对数据标准化
10
11 # 选取特征和标签
12 line_nums = len(data)
13 data_label = np.zeros(line_nums)
14 if all_features == True:
15     data_feature = np.zeros((line_nums, 41)) # 创建line_nums行 41列的矩阵
16     for i in range(line_nums): # 依次读取每行
17         data_feature[i, :] = data[i][0:41] # 选择前41个特征 划分数据集特征和标签
18         data_label[i] = int(data[i][-1]) # 标签
19 else:
20     data_feature = np.zeros((line_nums, 10)) # 创建line_nums行 10列的矩阵
21     for i in range(line_nums): # 依次读取每行
22         feature = [3, 4, 5, 6, 8, 10, 13, 23, 24, 37] # 选择第3, 4, 5, 6, 8, 10, 13, 23, 24, 37这10个特征分类
23         for j in feature:
24             data_feature[i, feature.index(j)] = data[i][j]
25         data_label[i] = int(data[i][-1]) # 标签
26
27 print('数据集特征大小: ', data_feature.shape)
28 print('数据集标签大小: ', len(data_label))
29 return data_feature, data_label
30
31 data_feature, data_label = data_processing(file="kddcup.data.txt", all_features=False)
```

## 七、参考文献

[1] <https://blog.csdn.net/Eastmount/article/details/103189405>

[2] [http://pytorch123.com/SecondSection/what\\_is\\_pytorch/](http://pytorch123.com/SecondSection/what_is_pytorch/)

[3] PyTorch 版《动手学深度学习》