💳信用卡💳欺诈检测--逻辑回归

9 1.1 k 74

本篇notebook使用SMOTE平衡样本，Logistic Regression建模，Confusion Matrix与ROC进行模型评估。

小科 · 玩转K-Lab

Fork

内容 数据集 Fork记录 74 评论 3

内容

版本 3 2018-11-26 00:08

使用逻辑回归进行信用卡欺诈检测

该项目使用的数据集是脱敏过的且经过PCA处理的数据，所以我们会先进行数据的探索，然后进行数据的预处理，其中包括不平衡样本如何处理，我们在这里会使用SMOTE进行不平衡样本的处理；接着会应用Logistic Regression来建模并使用confusion matrix和ROC来评估模型。

如果想查看图表相关的数据可视化及绘制代码，可以先Fork小科的这篇项目，运行本项目后，在想要查看代码的cell上点击右上角朝下的小箭头来展开代码cell即可查看。

目录
1.数据探索
1.1.探索交易时间分布
1.2.探索交易金额分布
2. 数据预处理
2.1. 标准化
2.2. 如何平衡样本
2.3.使用SMOTE平衡样本
3. 建模与模型评估

In [1]:

```
# 查看当前挂载的数据集目录
!ls ../input/fraud_detection
```

```
creditcardfraud.csv
```

In [2]:

```
import numpy as np
import pandas as pd
```

In [3]:

```
from plotly import __version__
print (__version__)

import plotly.offline as offline
from plotly.offline import init_notebook_mode, iplot

init_notebook_mode(connected=True)
from plotly.graph_objs import *
import colorlover as cl
from plotly import tools
```

```
2.0.1
```

In [4]:

```
import matplotlib.pyplot as plt
import matplotlib as mpl
```

1. 数据探索

In [5]:

```
data = pd.read_csv('../input/fraud_detection/creditcardfraud.csv')
```

In [6]:

```
data.shape
```

Out[6]:

```
(284807, 31)
```

检查是否有空值

In [7]:

```
data.isnull().values.sum()
```

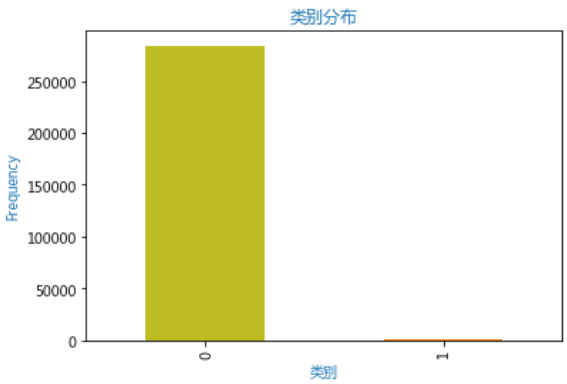
Out[7]:

```
0
```

查看前四行数据

Out[8]:

V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V2
8321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.02105
0018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.01472
3198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.05975
0309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.06145
7193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.21515



正常交易记录有284315条
异常交易记录有492条
异常交易记录比率0.173%

1.1.探索交易时间分布

正常交易的时间描述信息

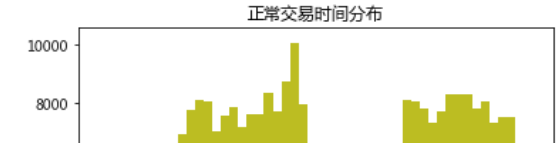
```
 count    284315.000000
mean      94838.202258
std       47484.015786
min           0.000000
25%       54230.000000
50%       84711.000000
75%      139333.000000
max      172792.000000
Name: Time, dtype: float64
```

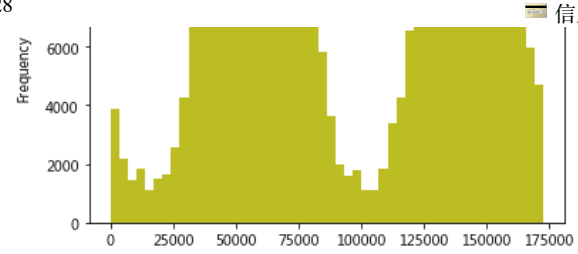
异常交易的时间描述信息

```
 count       492.000000
mean      80746.806911
std       47835.365138
min         406.000000
25%       41241.500000
50%       75568.500000
75%      128483.000000
max      170348.000000
Name: Time, dtype: float64
```

Out[12]:

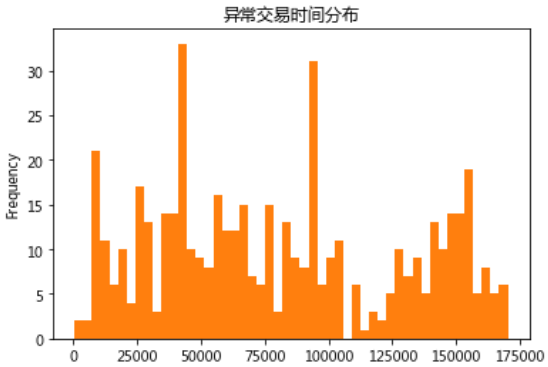
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa770ae96a0>
```





Out[13]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa76ea0d828>
```



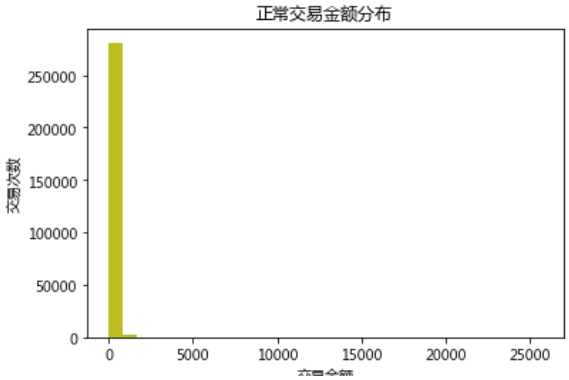
可以看出正常的交易记录是呈周期性分布的，而异常交易分布较平均。所以可以从交易周期的低频段入手欺诈交易的检测。

1.2.探索交易金额分布

```
正常交易的金额描述信息
 count    284315.000000
mean         88.291022
std         250.105092
min           0.000000
25%           5.650000
50%          22.000000
75%          77.050000
max       25691.160000
Name: Amount, dtype: float64
异常交易的金额描述信息
 count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64
```

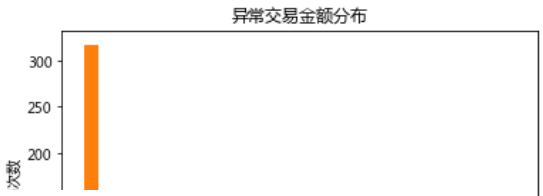
Out[15]:

```
Text(0, 0.5, '交易次数')
```



Out[16]:

```
Text(0, 0.5, '交易次数')
```



2. 数据预处理

2.1. 标准化

鉴于Amount列特征值的取值范围相比于PCA处理过的其他28列（V1至V28）特征值取值范围相差很大，需统一标准即**标准化**处理，以此来消除内部构成不同造成的对结果的影响。

In [17]:

```
from sklearn.preprocessing import StandardScaler

# reshape(-1,1) 将data['Amount']变成只有一列，行数不限定的np.array
data['normAmount'] = StandardScaler().fit_transform(data['Amount'].values.reshape(-1,1))

print('未标准化的Amount: ',data['Amount'].values.reshape(-1,1))
print('标准化后的Amount: ',StandardScaler().fit_transform(data['Amount'].values.reshape(-1,1)))

# 删除不需要使用到的两列数据
new_data = data.drop(['Time','Amount'], axis = 1)
new_data.head()
```

```
未标准化的Amount:  [[149.62]
 [  2.69]
 [378.66]
 ...
 [ 67.88]
 [ 10.  ]
 [217.  ]]
标准化后的Amount:  [[ 0.24496426]
 [-0.34247454]
 [ 1.16068593]
 ...
 [-0.0818393 ]
 [-0.31324853]
 [ 0.51435531]]
```

Out[17]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V22
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	...	-0.018307	0.277838
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.225775	-0.638672
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.247998	0.771679
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	...	-0.108300	0.005274
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	...	-0.009431	0.798278

5 rows × 30 columns

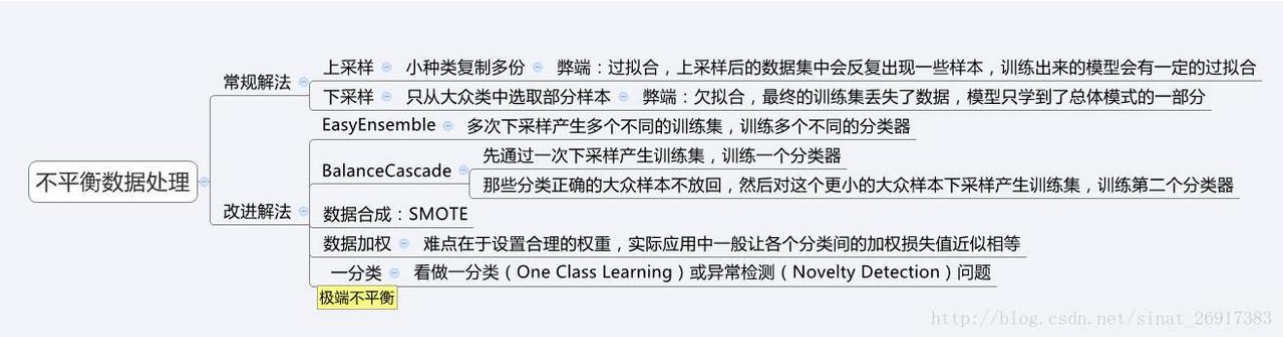
2.2 如何平衡样本

在进行平衡样本的数据预处理以前，先来谈谈非平衡样本的影响以及常用的一些平衡样本的方法，及适用场景。

不平衡样本

不平衡的样本会影响模型的评估效果，严重的会带来过拟合的结果。所以我们需要让正负样本在训练过程中拥有相同话语权或权重。在这里，称数据集中样本较多的一类称为“大众类”（majority class），样本较少的一类称为“小众类”（minority class）。

对于不平衡样本的处理做法总结如下[1] (https://blog.csdn.net/sinat_26917383/article/details/75890859)：



图片来源：素质云博客

常规做法是进行**上采样**与**下采样**，也就是下采样（**Undersampling**,欠采样）大众类，上采样（**Oversampling**，过采样）小众类。但是这样也会有相应的弊端出现。因为上采样是复制多份小众类，也就是下采样是选取部分大众类，所以上采样中小众类会反复出现一些样本，这会导致过拟合；下采样会由于丢失信息而导致欠拟合。

所以针对**下采样信息丢失**的问题，有**EasyEnsemble**，与**BalanceCascade**两种**改进**方法。

对于**上采样的改进方法**，可以通过**数据合成**方法来基于已有的数据生成更多的样本，其中数**SMOTE**最为常见；或者可以通过**加权**的方式来解决问题，但其难点在于如何合理设置权重。

同样的，我们可以换一种角度，对于正负样本极不平衡的情况下，我们也可以视其为**异常值检测**（Outlier Detection）或**一分类**（One Class Learning）问题。经典的工具包有One-class SVM等。

以上方法着重于处理数据，但同时也有适用于不平衡样本的模型比如XGBoost 。

所以解决不平衡样本的问题有很多种方法，那如何选择？ [1] (https://blog.csdn.net/sinat_26917383/article/details/75890859)

- 在**正负样本都非常之少**的情况下, 采用**数据合成**的方式
- 在**负样本足够多，正样本非常之少**且比例及其悬殊的情况下, 考虑**一分类**方法
- 在**正负样本都足够多**且比例不是特别悬殊的情况下, 应该考虑**采样**或者**加权**的方法

想了解更多内容可以参考以下列表：

- 如何解决机器学习中数据不平衡问题 (https://mp.weixin.qq.com/s?__biz=MzA4NzE1NzYyMw==&mid=2247492055&idx=3&sn=76e4216a997199a6b2b76daa403ef000&chksm=903f1fcfa74896d92218c41814a7

2.3.使用SMOTE平衡样本

In [18]:

```
X = np.array(new_data.iloc[:, new_data.columns != 'Class']) # 选取特征列数据
y = np.array(new_data.iloc[:, new_data.columns == 'Class']) # 选取类别label
print('X shape:',X.shape, '\ny shape:',y.shape)
```

```
X shape: (284807, 29)
y shape: (284807, 1)
```

In [19]:

```
# !pip install imblearn # 安装包
```

```
Requirement already satisfied: imblearn in /opt/conda/lib/python3.5/site-packages (0.0)
Requirement already satisfied: imbalanced-learn in /opt/conda/lib/python3.5/site-packages (from imblearn) (0
Requirement already satisfied: scipy>=0.13.3 in /opt/conda/lib/python3.5/site-packages (from imbalanced-lear
Requirement already satisfied: numpy>=1.8.2 in /opt/conda/lib/python3.5/site-packages (from imbalanced-learn
Requirement already satisfied: scikit-learn>=0.20 in /opt/conda/lib/python3.5/site-packages (from imbalanced
You are using pip version 18.0, however version 18.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

In [22]:

```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
```

Step 1. 对所有数据进行训练集与测试集的切分

训练集：测试集 = 7：3

Step 2. 先对数据进行上采样，然后对上采样后的数据进行训练集与测试集的切分

In [42]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

print('训练集的交易记录条数：',X_train.shape[0])
print('测试集的交易记录条数：',X_test.shape[0])
print('交易记录总数：',X_train.shape[0] + X_test.shape[0])
print('上采样前，类别为‘1’的共有{}个，类别为‘0’的共有{}个。'.format(sum(y_train==1),sum(y_train==0)))
print('------------------------')

# 对训练集进行上采样处理
smote = SMOTE(random_state=2)
X_train_os,y_train_os = smote.fit_sample(X_train, y_train.ravel()) # ravel(): change the shape of y to (n_s

print('上采样后，训练集的交易记录条数：', len(X_train_os))
print('其中，训练集X的shape:',X_train_os.shape,'，y的shape:',y_train_os.shape)
print('交易记录总数：',X_train_os.shape[0] + X_test.shape[0])
print('上采样后，类别为‘1’的共有{}个，类别为‘0’的共有{}个。'.format(sum(y_train_os==1),sum(y_train_os==0)))
```

```
训练集的交易记录条数： 199364
测试集的交易记录条数： 85443
交易记录总数： 284807
上采样前，类别为‘1’的共有[345]个，类别为‘0’的共有[199019]个。
------------------------
上采样后，训练集的交易记录条数： 398038
其中，训练集X的shape: (398038, 29) ，y的shape: (398038,)
交易记录总数： 483481
上采样后，类别为‘1’的共有199019个，类别为‘0’的共有199019个。
```

3. 建模与模型评估

In [70]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix,roc_curve, auc, recall_score, classification_report
```

In [ ]:

```
# 定义正则化权重参数，用以控制过拟合
paramaters = {'C':np.linspace(1,10, num=10)} # generate sequnce: start = 1, stop = 10
paramaters
# C_param_range = [0.01,0.1,1,10,100]
```

Tip: CV的运行过程可在页面下方的【日志监控】处查看

In [52]:

```
lr = LogisticRegression()
# 5 folds, 3 jobs run in parallel
lr_clf = GridSearchCV(lr, paramaters, cv=5, n_jobs=3, verbose=5)
lr_clf.fit(X_train_os, y_train_os.ravel())
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=3)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=3)]: Done  12 tasks      | elapsed:   44.3s
/opt/conda/lib/python3.5/site-packages/sklearn/externals/joblib/externals/loky/process_executor.py:706: User

A worker stopped while some jobs were given to the executor. This can be caused by a too short worker timeou

[Parallel(n_jobs=3)]: Done  50 out of  50 | elapsed:  2.9min finished
/opt/conda/lib/python3.5/site-packages/sklearn/linear_model/logistic.py:433: FutureWarning:

Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
```

Out[52]:

```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False),
```

```
        fit_params=None, iid='warn', n_jobs=3,
        param_grid={'C': array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])},
        pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
        scoring=None, verbose=5)
```

In [55]:

```
print('最好的参数: ',lr_clf.best_params_)
```

```
最好的参数:  {'C': 4.0}
```

In [56]:

```
lr1 = LogisticRegression(C=4, penalty='l1',verbose=5)
lr1.fit(X_train_os, y_train_os.ravel())
```

```
/opt/conda/lib/python3.5/site-packages/sklearn/linear_model/logistic.py:433: FutureWarning:

Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.


[LibLinear]
```

Out[56]:

```
LogisticRegression(C=4, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l1', random_state=None, solver='warn',
          tol=0.0001, verbose=5, warm_start=False)
```

混淆矩阵的绘图function

In [58]:

```
import itertools

def plot_confusion_matrix(cm, classes,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

关于模型评估

二元分类的混淆矩阵形式如下：

		实际值	
	样本总数 N	Positive	Negative
预测值	Positive	实际是Positive，预测成Positive的样本数，又叫true positive (TP)	实际是Negative，预测成Positive的样本数，又叫false positive（FP）
	Negative	实际是Positive，预测成Negative的样本数，又叫false negative(FN)	实际是Negative，预测成Negative的样本数，又叫true negative(TN)

		实际Positive样本数 =TP+FN	实际Negative样本数 =FP+TN

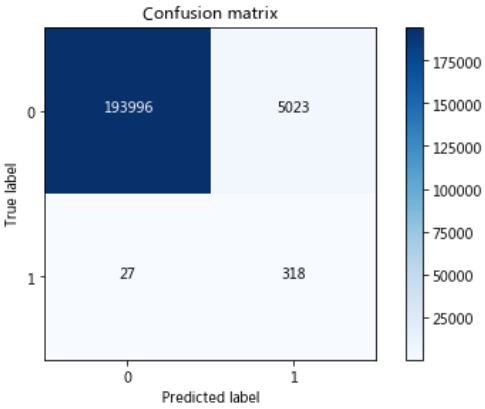
图源 (https://www.xuebuyuan.com/3239374.html)

> 由于我们是要尽量将所有信用卡欺诈的数据找出来，所以有个很重要的衡量标准： 召回率： $\boldsymbol{Recall} = \frac{TP}{TP+FN}$ 也就是说，假设1000条信用卡交易记录中，有10条是欺诈交易，如果最后识别出4条，那么召回率就为 $\frac{4}{10} = 0.4$

In [63]:

```
# 对原训练集X进行预测
y_train_pre = lr1.predict(X_train)

# 训练集的混淆矩阵
cnf_matrix_train = confusion_matrix(y_train, y_train_pre)

print("Recall metric in the train dataset: {}%".format(100*cnf_matrix_train[1,1]/(cnf_matrix_train[1,0]+cn

class_names = [0,1]
plt.figure()
plot_confusion_matrix(cnf_matrix_train , classes=class_names, title='Confusion matrix')
plt.show()
```

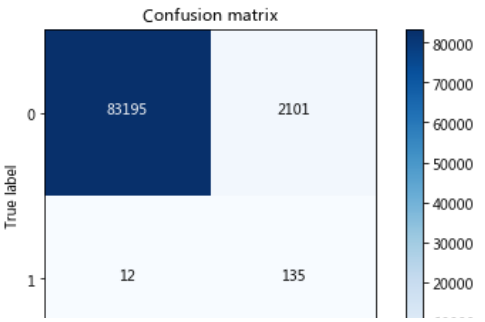
```
Recall metric in the train dataset: 92.17391304347827%
```

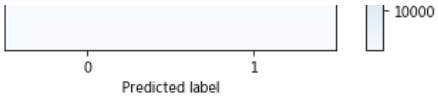


In [82]:

```
# 对原测试集进行预测
y_pre = lr1.predict(X_test)

# 测试集的混淆矩阵
cnf_matrix = confusion_matrix(y_test, y_pre)

print("Recall metric in the test dataset: {}%".format(100*cnf_matrix[1,1]/(cnf_matrix[1,0]+cnf_matrix[1,1]

class_names = [0,1]
plt.figure()
plot_confusion_matrix(cnf_matrix , classes=class_names, title='Confusion matrix')
plt.show()
```

```
Recall metric in the test dataset: 91.83673469387755%
```





用上采样处理后的训练集训练模型

In [84]:

```
model = lr1.fit(X_train_os, y_train_os.ravel())
```

```
/opt/conda/lib/python3.5/site-packages/sklearn/linear_model/logistic.py:433: FutureWarning:

Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.


[LibLinear]
```

In [85]:

```
# decision_function(): Predict confidence scores for samples X_test.
y_pred_sample_score = model.decision_function(X_test)
y_pred_sample_score
```

Out[85]:

```
array([-1.62412655, -1.54930668, -1.63358588, ..., -2.9664152 ,
       -2.1298146 , -1.86854419])
```

ROC 评估

FPR，TPR

FPR（False Positive Rate，假阳性率）, TPR(True Positive Rate，真阳性率)分别对应ROC曲线的横坐标与纵坐标

- **真阳性率**（灵敏度）= **召回率**（Recall）：
  如果一个实例类别是positive，分类器预测结果的类别也是positive的比例。这个指标也叫敏感度（sensitivity）或召回率（recall），描述了分类器对positive类别的**敏感程度**。
  $$TPR = \frac{TP}{TP+FN}$$
- **假阳性率** = **错检率**（fallout）：
  如果一个实例类别是negative，分类器预测结果的类别是positive的比例。这个指标也叫错检率（fallout）。
  $$FPR = \frac{FP}{FP+TN}$$

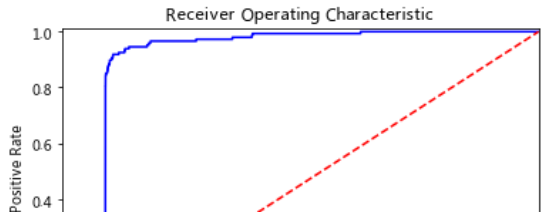
In [81]:

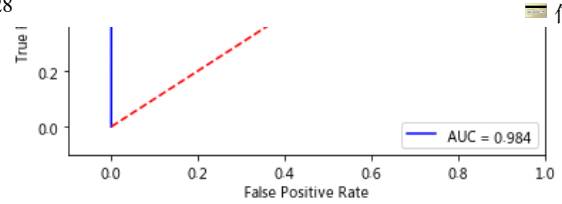
```
fpr, tpr, thresholds = roc_curve(y_test, y_pred_sample_score)

roc_auc = auc(fpr, tpr)
print('准确率: ',roc_auc)
```

```
准确率:  0.9839169911070786
```

In [79]:

```
# Plot ROC

plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b',label='AUC = %0.3f'% roc_auc)
plt.legend(loc='lower right') # 设置legend的位置
plt.plot([0,1],[0,1],'r--') # red, --
plt.xlim([-0.1,1.0])
plt.ylim([-0.1,1.01])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```





从图可以看出，ROC曲线非常接近左上角，即AUC(Area Under Curve)面积很大，说明准确性很高，模型很好。最靠近左上角的ROC曲线的点是错误最少的最好阈值（thresholds），其假阳性与假阴性的总数最少。

参考

- 非平衡数据处理方式与评估 (https://blog.csdn.net/sinat_26917383/article/details/75890859)
- Smote with imbalance data (https://www.kaggle.com/qianchao/smote-with-imbalance-data)