异常 & 新奇检测：一类支持向量机

Part 2：案例分析

支持向量机（Support Vector Machine，以下简称 **SVM**）相信接触过机器学习和深度学习的读者一定不陌生了，是按监督学习（Supervised Learning）方式对数据进行二分类的广义线性分类器。笔者在这系列文章要介绍的是SVM的延申：**一类支持向量机**（One Class SVM，简称 **OCSVM**）。OCSVM的主要用途就是新奇检测（Novelty Detection）和异常检测（Outlier Detection）。

系列文章分为两部分：

- **Part 1** 主要围绕着OCSVM的一些理论进行讲解。充分理解理论才能将它放入你的武器库，做到在合适的场合使用。
- **Part 2** 会用一些玩具例子和实际例子演示OCSVM的具体实施和效果

Part 2 会和 Part 1 一样，着重介绍 ν - OCSVM 的实施案例，都是基于 Python Scikit-learn。

3. 案例实施：异常 vs 新奇检测

重温一下异常检测和新奇检测分别是什么：

1. **新奇检测**：训练集中只有一种类型A的样本，但测试集中有属于和不属于A的样本，要求检测每一个数据属不属于A。
2. **异常检测**：训练集中有一种类型A的样本和其他数量远小于A的样本，要求检测出不属于A的样本。

3.1 玩具例子：新奇检测

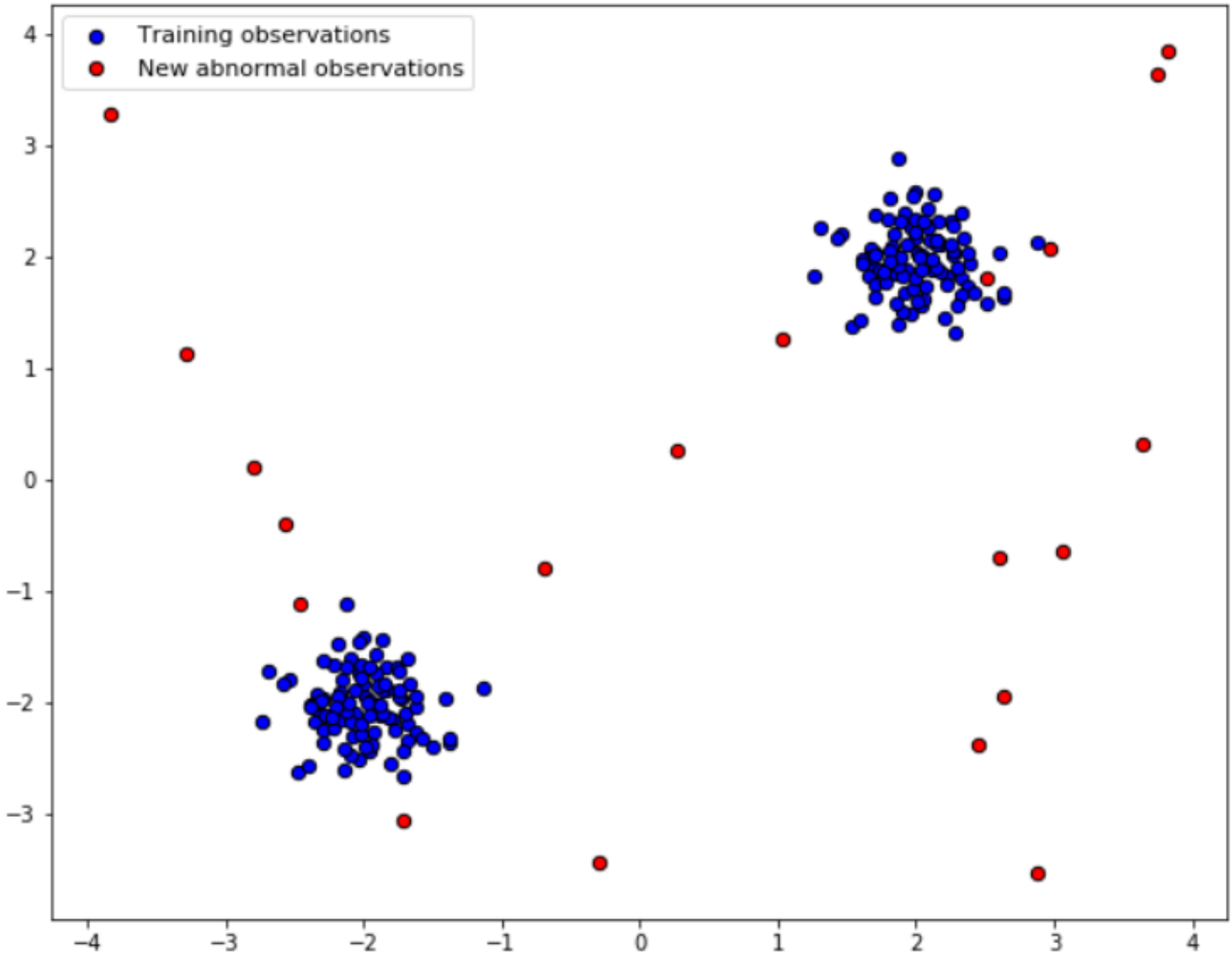
```
import numpy as np
from sklearn import svm
import matplotlib.pyplot as plt
import matplotlib.font_manager

%matplotlib inline
```

创建一组数据：

- 训练集：由两个均属点生成的高斯分布数据，无异常值，共200个

- 测试集：两部分组成。由两个均数点生成的相同高斯分布数据，非异常值，共40个；均态分布生成的异常值，共20个。

```
X = 0.3 * np.random.randn(100,2)
X_train = np.r_[X+2, X-2]
X = 0.3 * np.random.randn(20,2)
X_test = np.r_[X+2, X-2]
# create some outliers
X_outliers = np.random.uniform(low = -4, high = 4, size = (20,2))
```

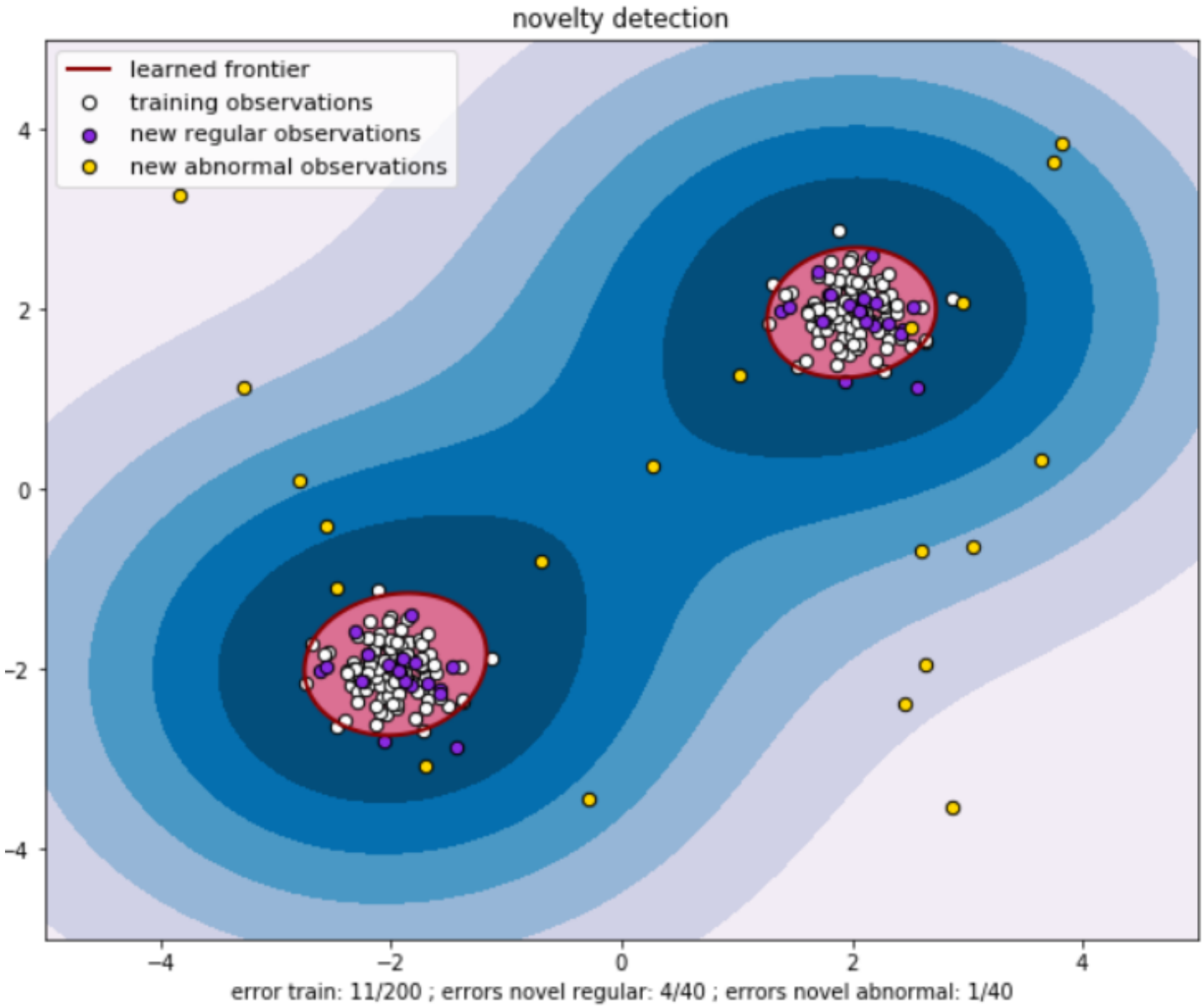


图中，蓝色的点是训练集，红色的点是测试集中的异常值。看图估算一下，异常值占比大概不到10%。我们取 $\nu = 0.05$。至于核函数，因为我们使用高斯分布生成数据集的，我们也自然用高斯函数核。

接下来我们训练模型：

```
clf = svm.OneClassSVM(nu = 0.05, kernel = 'rbf', gamma = 'scale')
clf.fit(X_train)
Y_train = clf.predict(X_train)
Y_test = clf.predict(X_test)
Y_outliers = clf.predict(X_outliers)
```

我们把训练的决策边界画出来：



图中，深红色代表了决策边界。在新奇检测中，**决策边界是不会变的**，因为OCSVM是最大间隔超平面，具有唯一解。除此之外，我们还会关心这些问题：

- FP，也就是假正值，是5%，40个异常值里分类错了1个。这是一个还不错的数字。我们训练样本、测试样本并没有特别多，只分错一个异常值是可以接受的。
- 训练错误率是11/200，虽然看上去很高，但并没有关系。$\nu > 0$ 的时候，根据OCSVM的性质，有 ν 比例的训练集数据会被作为异常值。这是有助于防止“过拟合”的。100%的训练正确率会导致很高

的预测错误率，会将测试集里的大部分正常数据判断成异常。

3.2 玩具例子：异常检测

在这个例子中，我们会用四种不同的**异常检测**算法，比较它们在五组2维数据里的表现。此部分参考了sklearn官网上的实例。

```
import numpy as np

import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
matplotlib.rcParams['contour.negative_linestyle'] = 'solid'

import time

from sklearn import svm
from sklearn.datasets import make_moons, make_blobs
from sklearn.covariance import EllipticEnvelope
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
```

每组数据集里正常样本有300个，其中15%是由均态分布生成的的异常值。（这15%也会作为 ν -OCSVM$ 中的参数。）数据集的具体介绍如下：

1. **数据一**：高斯分布，一个均数点，标准方差为1
2. **数据二**：高斯分布，两个均数点，标准方差相同，为0.5
3. **数据三**：高斯分布，两个均数点（和数据二的均属点一致），标准方差不同，一个为1.5，一个为0.3
4. **数据四**：双月牙型数据
5. **数据五**：均态分布

每一组数据的**异常值**完全相同，均为 -6 到 6 的均态分布。

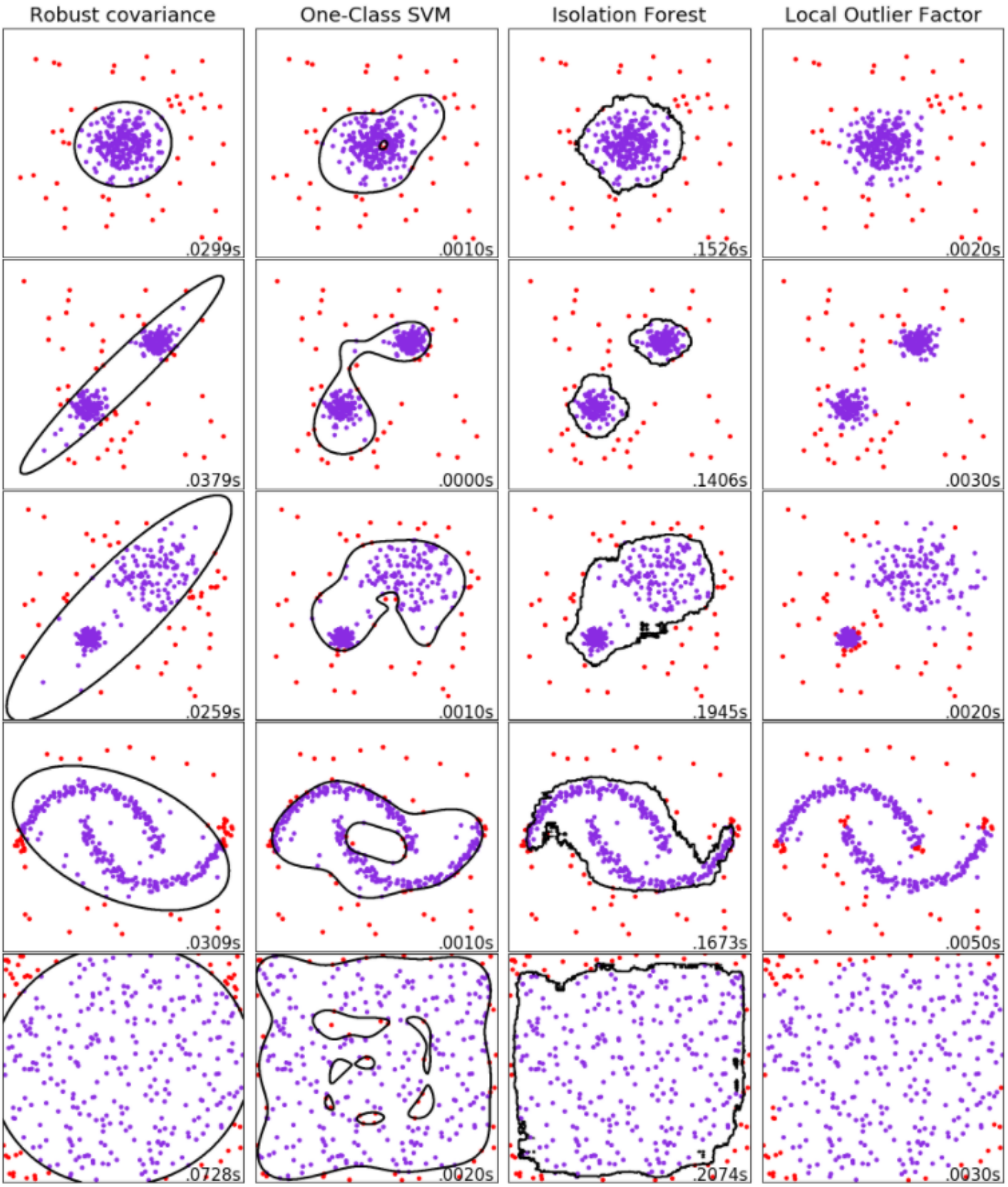
```
# example setting
n_samples = 300
outliers_fraction = 0.15
n_outliers = int(n_samples * outliers_fraction)
n_inliers = n_samples - n_outliers

#Define datasets
blobs_params = dict(random_state = 13192, n_samples = n_inliers, n_features = 2)
datasets = [
    make_blobs(centers=[[0,0], [0,0]], cluster_std=1, **blobs_params)[0],
    make_blobs(centers=[[2,2], [-2,-2]], cluster_std=[0.5,0.5], **blobs_params)[0],
    make_blobs(centers=[[2,2], [-2,-2]], cluster_std=[1.5,0.3], **blobs_params)[0],
    4 * (make_moons(n_samples=n_samples, noise = 0.05, random_state = 13192)[0] - np.array([0.5,
    14 * (np.random.RandomState(42).rand(n_samples, 2) - 0.5)
]
```

异常检测算法包括：

1. ν **- OCSVM**
2. **Robust Covariance**：这个模型假设我们的数据是按照多元高斯分布，训练出重心和方差参数，然后根据高斯概率密度估算每个点被分配到重心的概率，判断异常。
3. **Isolation Forest 孤立森林**：孤立森林由很多孤立树（isolation tree）组成。孤立树在特征空间里从不同维度随机2切分，从而让每一个数据点都落在一个叶子节点上。其核心思想是异常点所需的切分次数比正常点要少，所以在孤立树里的深度也相对要浅。多次创建孤立树组成孤立森林，根据平均深度对异常点进行有效判断。
4. **Local Outlier Factor**：基于密度的经典算法，核心思想和DBSCAN一类的聚类算法很相像。一句话概括：赋予每个数据点一个“局部离群因子”：该数据点周围最近K个数据点所处位置的平均密度比上该数据点所处位子的密度。以下简称LOF。

```
anomaly_algorithms = [
    ("Robust covariance", EllipticEnvelope(contamination=outliers_fraction)),
    ("One-Class SVM", svm.OneClassSVM(nu=outliers_fraction, kernel = 'rbf', gamma = 0.1)),
    ("Isolation Forest", IsolationForest(contamination = outliers_fraction, random_state=129)),
    ("Local Outlier Factor", LocalOutlierFactor(n_neighbors=35, contamination=outliers_fraction)
]
```

训练，并记录下训练时长，然后逐个画出决策边界：



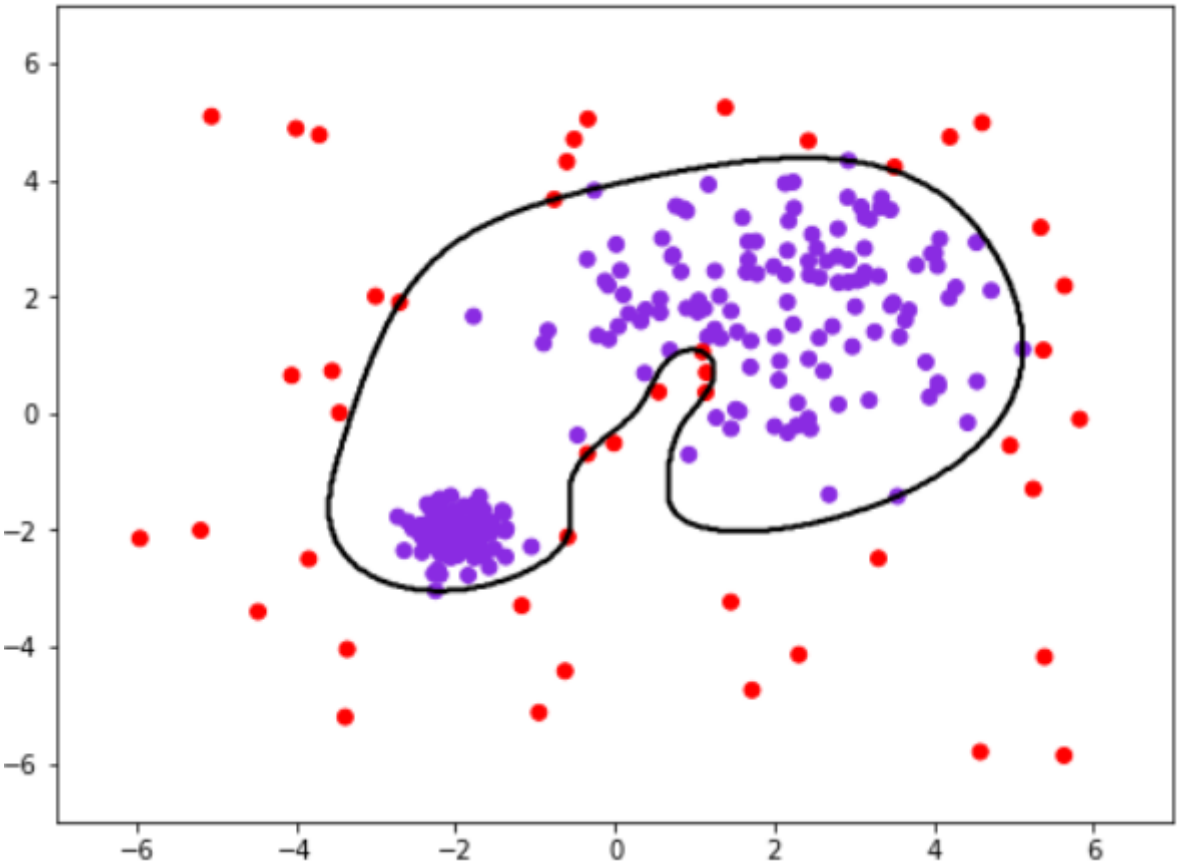
- 异常检测场景中，LOF是不能用 `predict()` 函数的，所以没有决策边界。
- Robust Covariace 因为假设数据集是单重心的高斯分布，所以一旦数据点拥有两个以上均属点，效果就没有那么好了。但我们可以观察到 Robust Covariance 是最这四种算法模型里对异常值最具有鲁棒性的，不会因为离群值的分布改变自己的决策边界的形状。

- 相反，OCSVM 对异常值十分敏感，所以对于相比于之前的新奇检测，异常检测的效果就没有那么好了。不过，他相比其他算法的优势在于对维度的适应性非常高。
- 孤立森林和LOF对于多均数的高斯分布数据有着很亮眼的表现。LOF对第三组数据的表现尤为出色，是它基于密度的体现。
- 对于最后一组数据来说，异常值的存在没有那么明显，因为数据本身就是均态分布的。我们不难发现 OCSVM 有一点过拟合现象。

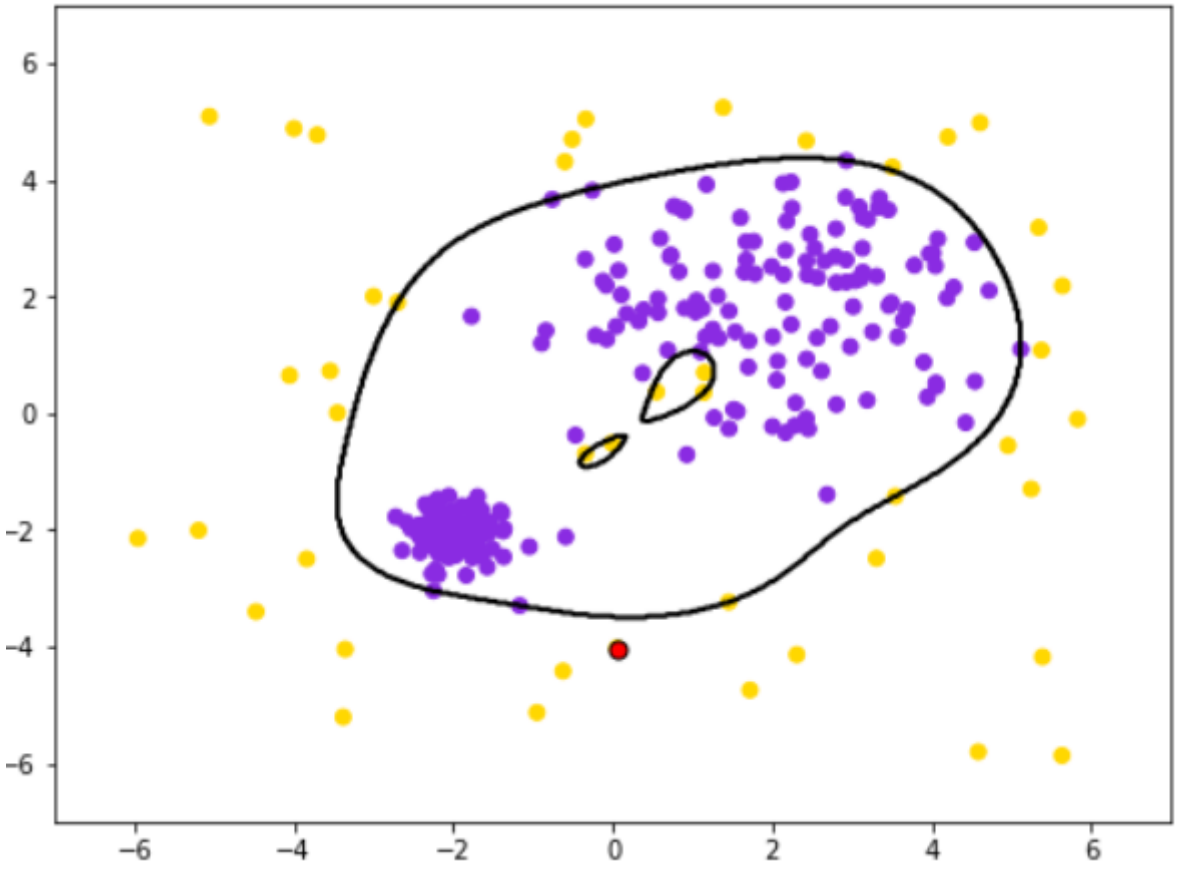
3.3 OCSVM对异常值有多敏感?

上文提到：OCSVM 对异常值十分敏感，所以对于相比于之前的新奇检测，异常检测的效果就没有那么好了。我们来看一下 ν - OCSVM 对异常值的敏感是如何体现的。

先做一次异常检测，数据集为章节2.2中的**数据三**：高斯分布，两个均数点（和数据二的均属点一致），标准方差不同，一个为1.5，一个为0.3。异常点是从 -6 到 6 的均态分布。结果如下



然后我们在这个数据集的基础上，随机再从 -6 到 6 的均态分布中生成**一个点**，加入其中，再训练和预测一次。分离边界如下：



其中红色的点是新加入的点。可以看出仅仅一点只差，分类边界就完全被改变了。这就是所谓的**敏感**，也就是说 **OCSVM在异常检测的场景中是不具有鲁棒性的（Robustness）** 。当然，这种敏感也取决于新加入的异常点在什么位置：红点离决策原边界越近（不论内外），对其影响就会越大。

而我们通常会更偏向具有鲁棒性的算法模型，因为这种模型的泛用性和稳定性都非常好。于异常检测相反，**OCSVM在新奇检测中是具有鲁棒性的**，因为我们的训练集是一个一组正常数据，不会有变动，训练出来的分离边界也就不会变，非常稳定。

3.4 实例：异常vs新奇检测

最后，我们用一个实例数据来分别体现 ν - OCSVM 在异常检测和新奇检测场景下的效果。

首先介绍一下我们使用的数据。数据来源于 1999年的KDD CUP：The 1999 KDD intrusion detection contest。MIT的实验室和美国空军合作，模拟7周的军事网络入侵得到的数据源，并加以修改后放出。KDD CUP上用了其中2周的数据，达2百万条网络信息。

注：下文提到的所有特征都可以在上面的网站中查询到。

笔者截取了这2百多万条其中的10%作为这次实验的数据。

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline


# read data
raw_data = pd.read_csv("kddcup_data_10_percent_corrected.csv")
print(raw_data.shape)
```

```
(494021, 42)
```

一共有50万数据，41个特征。在这些网络连接（network connection）行为中，有正常的行为，被标为'normal'；也有网络攻击行为，用攻击类型作为标签。我们这次的目标是训练出一个能供检测网络攻击行为的模型。

数据处理：

训练之前还是要先处理数据，这部分对于异常检测和新奇检测是一样的。我处理数据的大致步骤如下：

- 我们这次实验主要是看网络攻击，所以在 `service` 这个特征里去掉不是'http'的数据。
- 我们也不关心没有成功登录的数据，所以 `logged_in = 0` 的数据都删除
- 把所有数据集的标签分为 `+1` 和 `-1` ： 'normal'为 `+1` ，其余的攻击类型皆为 `-1` 。

```
data = raw_data[raw_data['logged_in'] == 1]
data = data[data['service'] == 'http']

data.loc[data['class_label'] == 'normal.', 'behavior'] = int(1)
data.loc[data['class_label'] != 'normal.', 'behavior'] = int(-1)

data = data.drop(['service','logged_in','class_label'],axis=1)

prop_of_attack_data = 100*(data[data['class_label']!='normal.'].shape[0])/(data.shape[0])
print('proportion of attacks in the new dataset is: %.2f '%prop_of_attack_data)
print(data.shape)
```

```
proportion of attacks in the new dataset is: 3.76
(58725, 40)
```

我们将数据减少到了接近6万个，并且得知有3.76%的数据是网络攻击，也就是我们这次模型里的异常值。

但是39个特征显然太多了。经过测试，30个特征放入OCSVM已经运行得相当慢了。读者可以通过Exploratory Data Analysis和Feature Engineering压缩特征数量。为了节省时间，笔者这里就选择了以

下三个特征作为我们的重点：

1. `duration` ： 连接到服务器的时常，以秒为单位。
2. `src_bytes` ： 从本地上传到远端的数据数量，以bytes为单位
3. `dst_bytes` ： 从远端下载到本地的数据数量，以bytes为单位

接下来我们来看看这几个特征的统计性质：

```
data['duration'].describe()
```

```
count    58725.000000
mean         0.686641
std         14.394064
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max       1440.000000
Name: duration, dtype: float64
```

```
data['src_bytes'].describe()
```

```
count    58725.000000
mean      2282.454593
std      10259.946866
min          0.000000
25%        218.000000
50%        253.000000
75%        309.000000
max      54540.000000
Name: src_bytes, dtype: float64
```

```
data['dst_bytes'].describe()
```

```
count    5.872500e+04
mean     4.658200e+03
std      1.982101e+04
min      0.000000e+00
25%      6.860000e+02
50%      1.724000e+03
75%      5.053000e+03
max      3.916592e+06
Name: dst_bytes, dtype: float64
```

从以上数据我们可以得出，这三个数据的分布是完完全全不一样的。OCSVM和SVM一样，是一个**以数据点之间的距离为核心的模型**（distance based method）。完全不同的分布对训练的结果产生很大的偏倚，影响OCSVM模型的准确性和泛用性。因此，在开始真正训练模型之前，需要将每个特征缩放到一个固定区间里，例如 $(0, 1)$（此部分代码会放在后面一起展示）。

2.3.1 异常检测

那数据处理到这一步就已经符合异常检测了，因为异常值是和正常数据混在一起的。我们接下来就用OCSVM来进行训练。为了公平比较异常检测和新奇检测，我们都将 ν 设置成精确的异常值比例，也就是刚刚算出来的 0.0376。

```
# Feature Scaling using Min-Max
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range = (0,1))

scaler.fit(sub_data)
X = scaler.transform(sub_data)

# normal implementation
from sklearn.svm import OneClassSVM
import time

nu = prop_of_attack_data/100
clf = OneClassSVM(nu = nu, gamma = 'scale', kernel = 'rbf')

t0 = time.time()
clf.fit(X)
t1 = time.time()

fit_time = t1 - t0
print('Model fit in %.2f seconds' % fit_time)
```

```
Model fit in 7.71 seconds
```

最后，我们计算整体的预测准确率以及真正值和假正值：

```
y_pred = clf.predict(X)
print('Accuracy is: %.3f' % (100*(y_pred == ground_truth).sum()/len(y_pred)))

TP = 100*(y_pred[ground_truth==1] == 1).sum()/len(y_pred[ground_truth==1])
FP = 100*(y_pred[ground_truth==-1] == 1).sum()/len(y_pred[ground_truth==-1])

print('True positive rate is %.2f ' % TP)
print('False positive rate is %.2f ' % FP)
```

```
Accuracy is: 98.072
True positive rate is 98.01
False positive rate is 0.23
```

2.3.2 新奇检测

我们再来看看新奇检测的情况。想要进行新奇检测，我们只需要先把异常值从训练集里剔除，再从剩下的正常数据中选择20%，和异常值一起作为测试集。

```
selected_features = ['duration','src_bytes', 'dst_bytes', 'behavior']
sub_data = data[selected_features]

normal_behavior = sub_data[sub_data['behavior'] == 1]
attack_behavior = sub_data[sub_data['behavior'] == -1]

X = normal_behavior[['duration','src_bytes','dst_bytes']]
y = normal_behavior['behavior']

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 235135

X_outlier = attack_behavior[['duration','src_bytes','dst_bytes']]

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range = (0,1))
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
X_outlier = scaler.transform(X_outlier)
```

训练和预测：

```
from sklearn.svm import OneClassSVM
import time

nu = prop_of_attack_data/100
clf = OneClassSVM(nu = nu, gamma = 'scale', kernel = 'rbf')

t0 = time.time()
clf.fit(X_train)
t1 = time.time()

fit_time = t1 - t0
print('Model fit in %.2f seconds' % fit_time)
```

```
Model fit in 4.46 seconds
```

```
print('True Positive Rate is %.2f'% (100*y_test_pred[y_test_pred == -1].size/X_test.shape[0]))
print('False Positive Rate is ', y_outlier_pred[y_outlier_pred == 1].size/X_outlier.shape[0])
```

```
True Positive Rate is 96.04
False Positive Rate is  0.0
```