查看数据集来源，每个字段的意义为

| **字段** | **字段描述** |
| --- | --- |
| loan\_id | 贷款记录唯一标识 |
| user\_id | 借款人唯一标识 |
| total\_loan | 贷款数额 |
| year\_of\_loan | 贷款年份 |
| interest | 当前贷款利率 |
| monthly\_payment | 分期付款金额 |
| grade | 贷款级别 |
| employment\_type | 所在公司类型（世界五百强、国有企业、普通企业…） |
| industry | 工作领域（传统工业、商业、互联网、金融…） |
| work\_year | 工作年限 |
| home\_exist | 是否有房 |
| censor\_status | 审核情况 |
| Issue\_date | 贷款发放的日期 |
| use | 贷款用途类别 |
| post\_code | 贷款人申请时邮政编码 |
| region | 地区编码 |
| debt\_loan\_ratio | 债务收入比 |
| del\_in\_18month | 借款人过去18个月逾期30天以上的违约事件数 |
| scoring\_low | 借款人在贷款评分中所属的下限范围 |
| scoring\_high | 借款人在贷款评分中所属的上限范围 |
| known\_outstanding\_loan | 借款人档案中未结信用额度的数量 |
| known\_dero | 贬损公共记录的数量 |
| pub\_dero\_bankrup | 公开记录清除的数量 |
| recircle\_bal | 信贷周转余额合计 |
| recircle\_util | 循环额度利用率 |
| initial\_list\_status | 贷款的初始列表状态 |
| app\_type | 是否个人申请 |
| earlies\_credit\_mon | 借款人最早报告的信用额度开立的月份 |
| title | 借款人提供的贷款名称 |
| policy\_code | 公开可用的策略\_代码=1新产品不公开可用的策略\_代码=2 |
| f系列匿名特征 | 匿名特征f0-f4，为一些贷款人行为计数特征的处理 |
| early\_return | 借款人提前还款次数 |
| early\_return\_amount | 贷款人提前还款累积金额 |
| early\_return\_amount\_3mon | 近3个月内提前还款金额 |

其中标称字段为loan\_id,user\_id,home\_exist,policy\_code,app\_type,region,post\_code,

imployment\_type,industry, grade,censor\_status,initial\_list\_status，policy\_code。以上这些字段不做噪声处理与异常值处理。

2.数据清洗

导入所需要的相关库

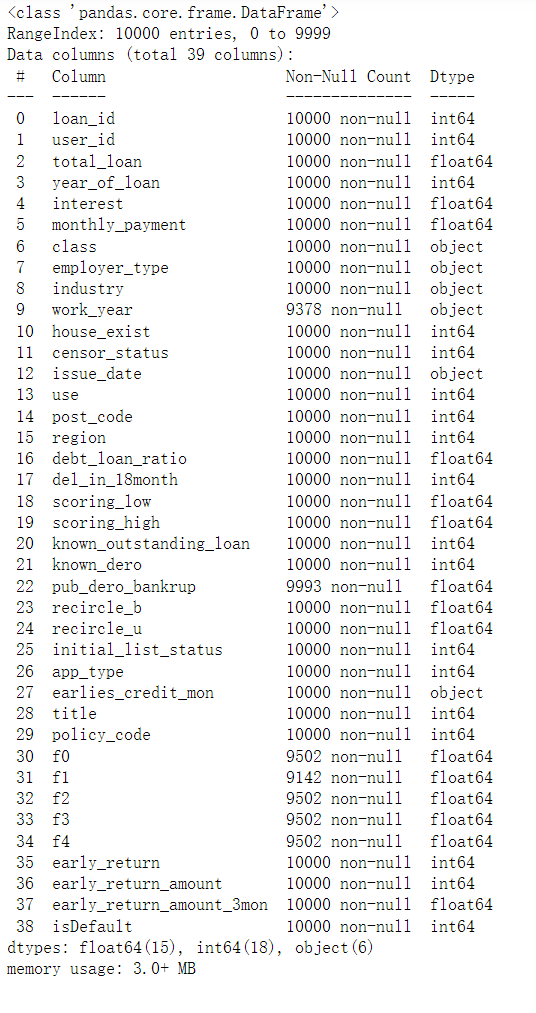
1. import numpy as np
2. import pandas as pd
3. import sklearn.preprocessing as skp
4. import scipy.stats as stats
5. import matplotlib.pyplot as plt
6. import sklearn.ensemble as es
7. import seaborn as sns

2.1数据格式查看

查看数据格式。

1. data=pd.read\_csv('./train\_public.csv')
2. pd.set\_option('display.max\_columns',None)
3. data.info()

结果如下。



class，employer\_type，Industry，work\_year，earlies\_credit\_mon，issue\_date 类型为object，

work\_year、f0、f1、f2、f3、f4、pub\_dero\_bankrup存在空缺值。

2.1缺失值处理

计算缺失值占总数据量的比例。比例不高，可以考虑删除或者填充。上述变量均为计数变量，由于work\_year为填充方式采用众数填充。

1. list\_nan=['work\_year','f0','f1','f2','f3','f4','pub\_dero\_bankrup']
2. for col in list\_nan:
3. print('%s: %f' % (col,sum(data[col].isnull())/data.shape[0]))

2.2数据去重

调用函数查看有没有重复值，结果为无重复值。

1. print('Numbers of Duplicated:',data1.duplicated().sum())



1. print('Numbers of Duplicated:',data1.drop('loan\_id',axis=1).drop('user\_id',axis=1).duplicated().sum())



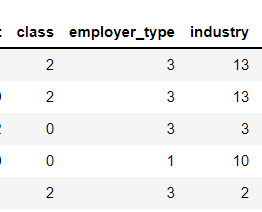
2.3数据转换——处理非数值类型数据

2.3.1数据编码

将class，employer\_type，Industry进行编码。采用序号编码。

1. skpe=skp.LabelEncoder()
2. skpe.fit(data1['class'])
3. data1['class']=skpe.transform(data1['class'])
4. skpe.fit(data1['employer\_type'])
5. data1['employer\_type']=skpe.transform(data1['employer\_type'])
6. skpe.fit(data1['industry'])
7. data1['industry']=skpe.transform(data1['industry'])
8. data1.head()

结果如图，class，employer\_type，Industry转换为数值类型。



2.3.2时序数据与计数数据处理

采用循环将work\_year转换为数字类型，其中10+ year转化为数字10，<1 year转化数字0。其余将year舍去保留年份。

1. work\_year=[]
2. for i in data1['work\_year'].values.tolist():
3. tmp=str(i).split(' ')
4. if tmp[0]=='10+':
5. work\_year.append(10)
6. elif tmp[0]=='<':
7. work\_year.append(0)
8. else:
9. work\_year.append(int(tmp[0]))
10. work\_year=pd.Series(work\_year)
11. data1['work\_year']=work\_year.values
12. data1['work\_year'].head()

处理issue\_date，将issue\_date转化为pandas自带的时间类型数据，然后将其拆分为issue\_year与issue\_month两个字段，前者为年，后者为月。

1. data1['issue\_date']=pd.to\_datetime(data1['issue\_date'],format='%Y/%m/%d')
2. data1['issue\_year']=data1['issue\_date'].dt.year
3. data1['issue\_month']=data1['issue\_date'].dt.month
4. data1=data1.drop('issue\_date',axis=1)

处理earlies\_credit\_mon，同样将其处理为earlies\_credit\_year与earlies\_credit\_mon两个字段。Earlies\_credit\_mon为两种类型格式混合，一种为年-月类型，另一种为日-月类型。对于日-月类型的数据，将其年份当作缺失值处理，采用众数2000进行填充。

2.4异常值检测与处理

采用了孤立森林与箱线图两种方法检验异常。保留了大部分检测出的异常值参与建模，仅删除了几项明显影响结果的异常值。

2.4.1所用算法介绍

一、KS检验

Kolmogorov–Smirnov 检验，简称KS检验，是统计学中的一种非参数假设检验，用来检测单样本是否服从某一分布，或者两样本是否服从相同分布。这里我们用于检测我们的数据是否符合正态分布。KS检验具体原理如下：

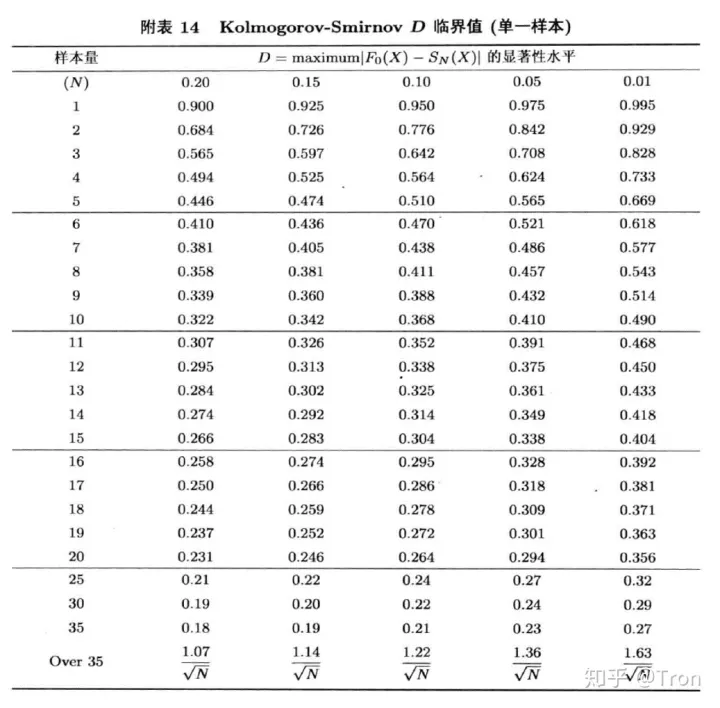
对于数据集，有两个假设，H0：数据集符合某一分布P；H1：不符合某一分布P。

计算数据集的经验分布函数，对于给定,,...个样本，其经验分布函数为计算公式如下：

KS检验结果为公式：

其中为待检验分布P的分布函数。

参考给出的KS检验D临界值来判断是否拒绝原假设，若D值大于表中对应的值，则拒绝原假设，数据集不符合分布P。



二、孤立森林

孤立森林（Isolation Forest）于2008年由西瓜书作者周志华团队提出，凭借其线性的时间复杂度与优秀的准确率被广泛应用于工业界中结构化数据的异常检测。孤立森林应用范围广泛，既能发现群异常数据，也能发现散点异常数据。同时也能处理训练数据中不包含异常数据的情况。

孤立森林成立有两个前提。一，异常数据占数据总比例要小；二异常数据与正常数据差异要大。孤立森林的基本原理为异常样本相较普通样本可以通过较少次数的随机特征分割被孤立出来。具体操作为不断选取属性划分异常值，而异常样本会较早的被划分出来。

孤立森林算法大致分为两步，第一步为构建树。

孤立森林构建单棵树的流程如下：

1.从训练数据中随机选择 Ψ 个点作为子样本，放入一棵孤立树的根节点；

2.随机指定一个维度，在当前节点数据范围内，随机产生一个切割点 p —— 切割点产生于当前节点数据中指定维度的最大值与最小值之间；

3.此切割点的选取生成了一个超平面，将当前节点数据空间切分为2个子空间：把当前所选维度下小于 p 的点放在当前节点的左分支，把大于等于 p 的点放在当前节点的右分支；

4.在节点的左分支和右分支节点递归步骤 2、3，不断构造新的叶子节点，直到叶子节点上只有一个数据（无法再继续切割） 或树已经生长到了所设定的高度 。

第二步为通过构建过的树进行预测，将要预测的样本放入孤立森林，通过计算点的孤立分数来识别出异常点。

对于每一个点x采用以下函数来计算异常分数。异常分数越接近1，其是异常点的可能性越高；异常分数都小于0.5，基本可以确定为正常数据；所有分数都在0.5附近，那么数据不包含明显的异常样本。

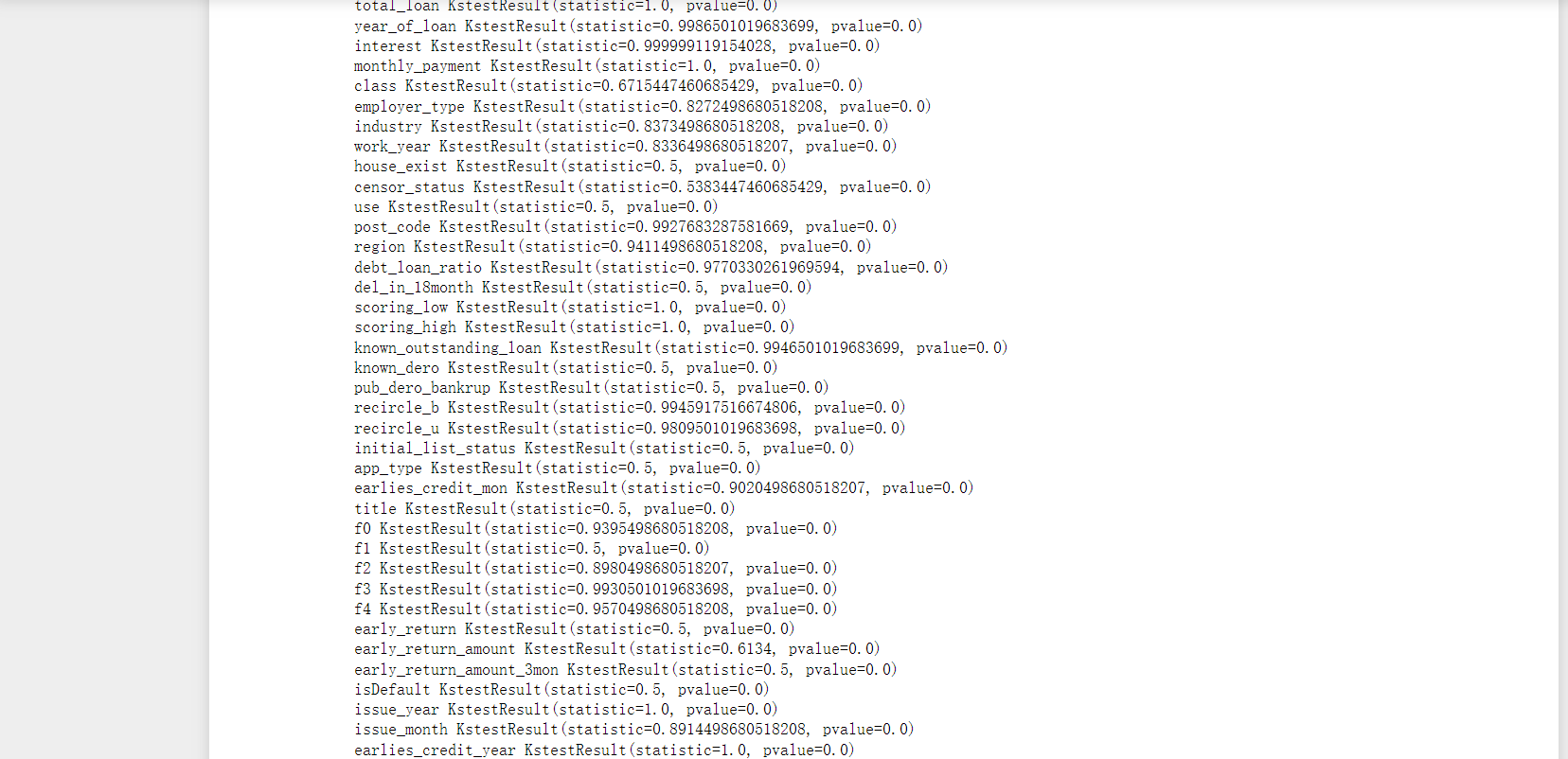
其中，E(h(x))表示点x在多棵树中的路径长度期望值，h(x)为x在树上的路径长度。其中，c(n)用来标准化记录x的路径长度。H(x)为调和数。ξ为欧拉常数，值为0.5772156649。

2.4.1正态分布检验

首先对数据进行正态检测分布以确保能够使用Z-score等基于正态分布的方式来检验异常值。这里采用KS方法进行检测。



以下为结果。很显然，数据完全不符合正态分布，无法采用3σ原则检验异常。

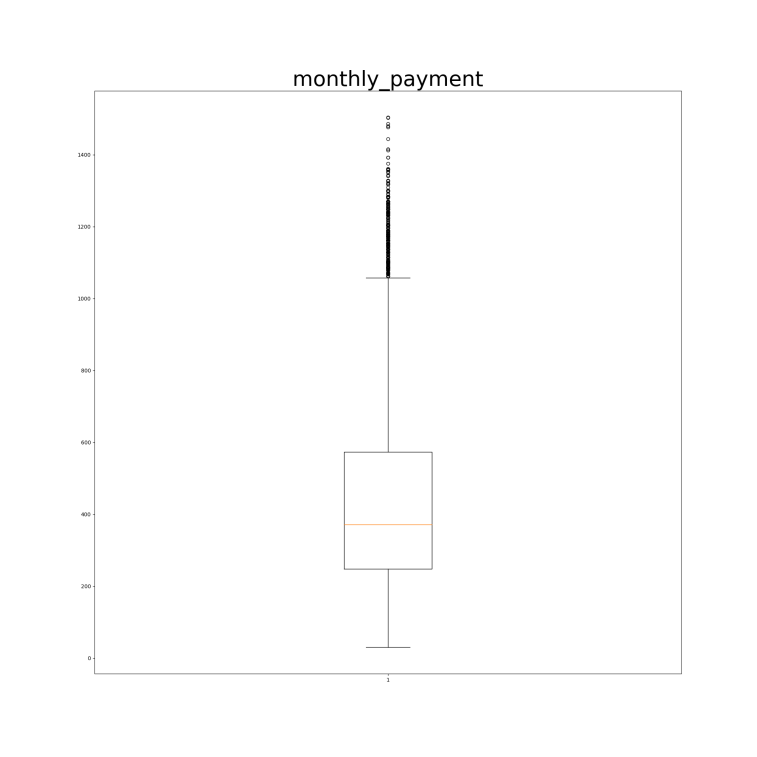
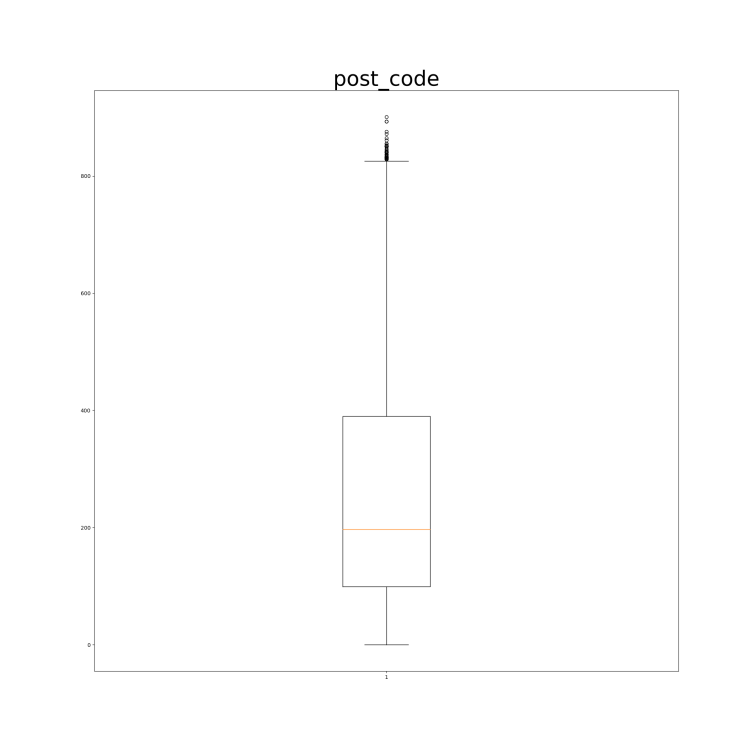
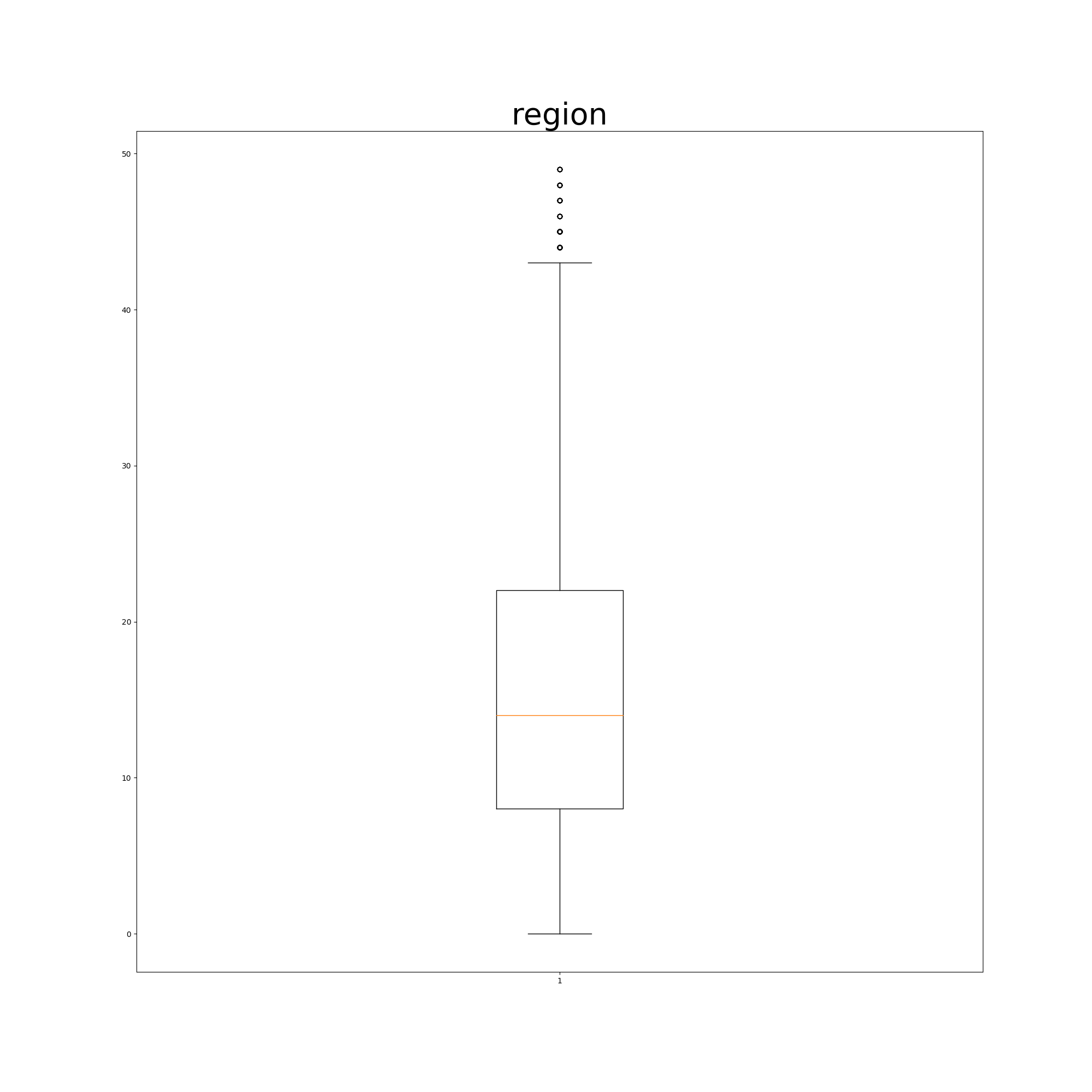
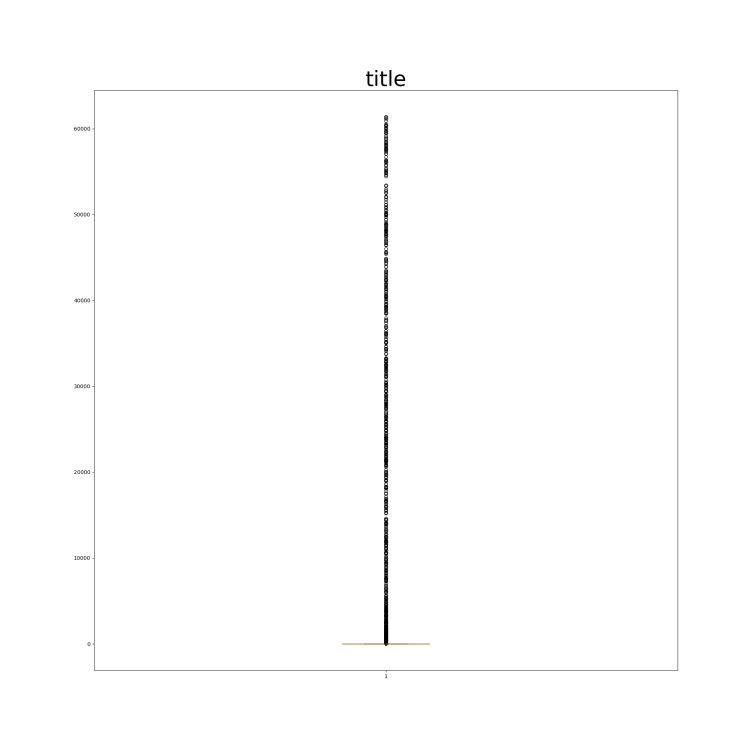
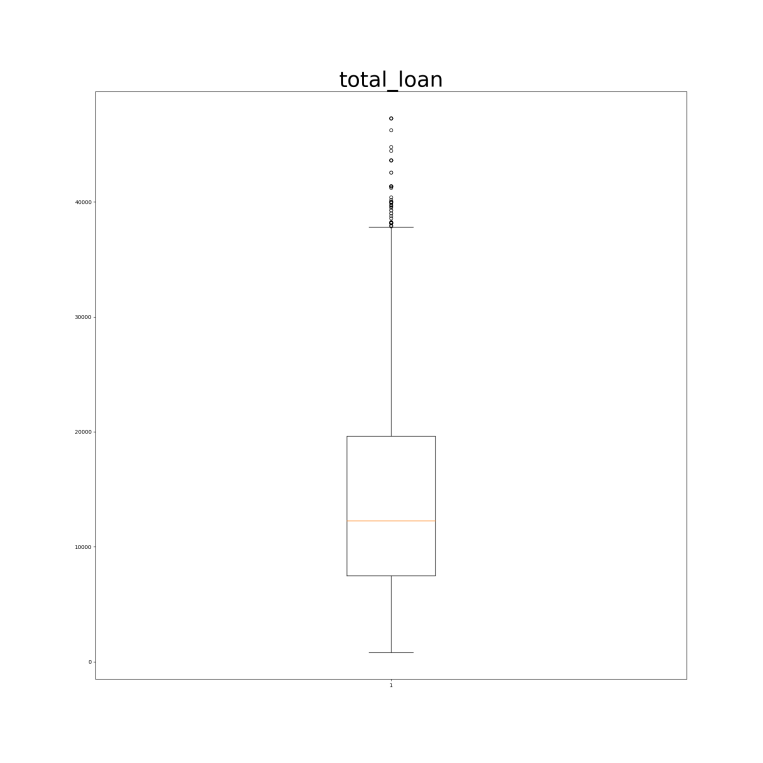
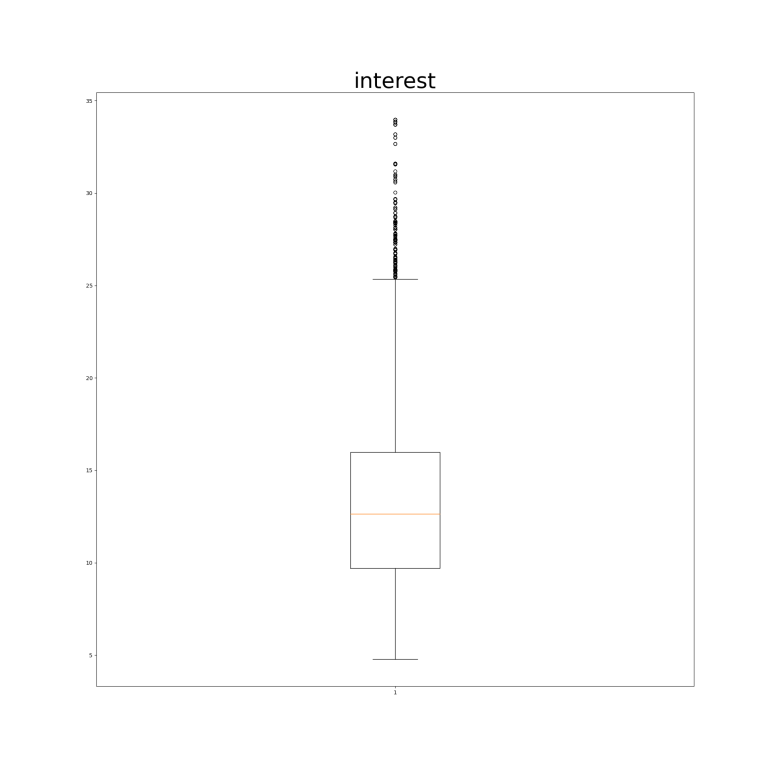


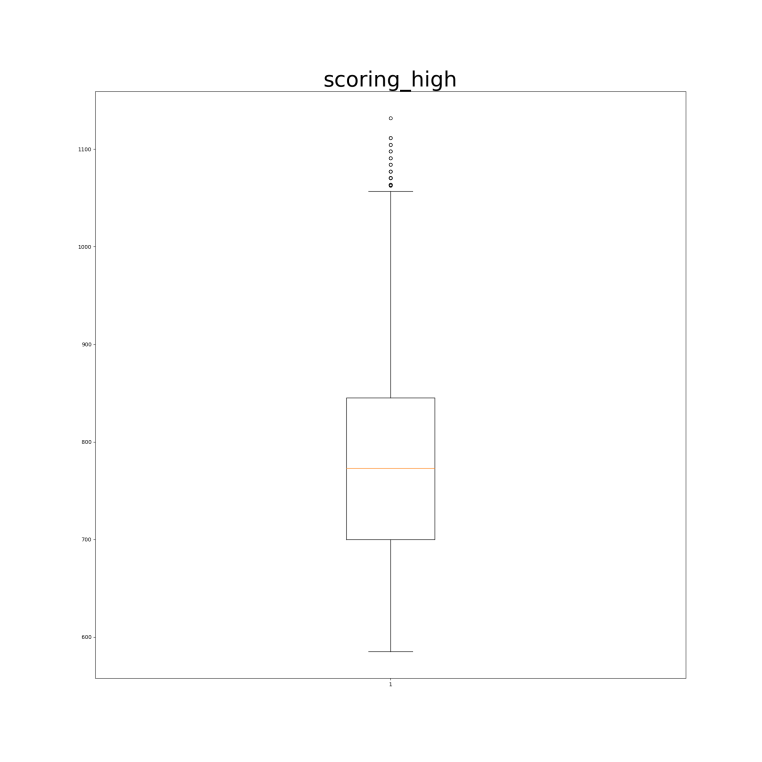
2.4.2箱线图异常

通过绘制箱线图来检测异常。由于字段的范围差距太大，不方便合入一张图中，因此每个字段绘制一张箱线图。

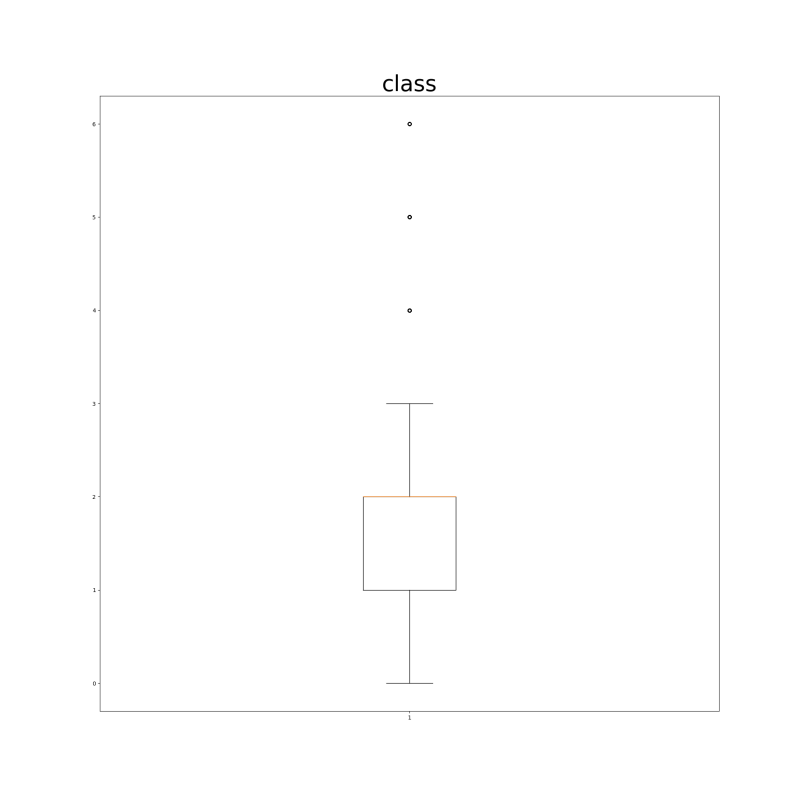
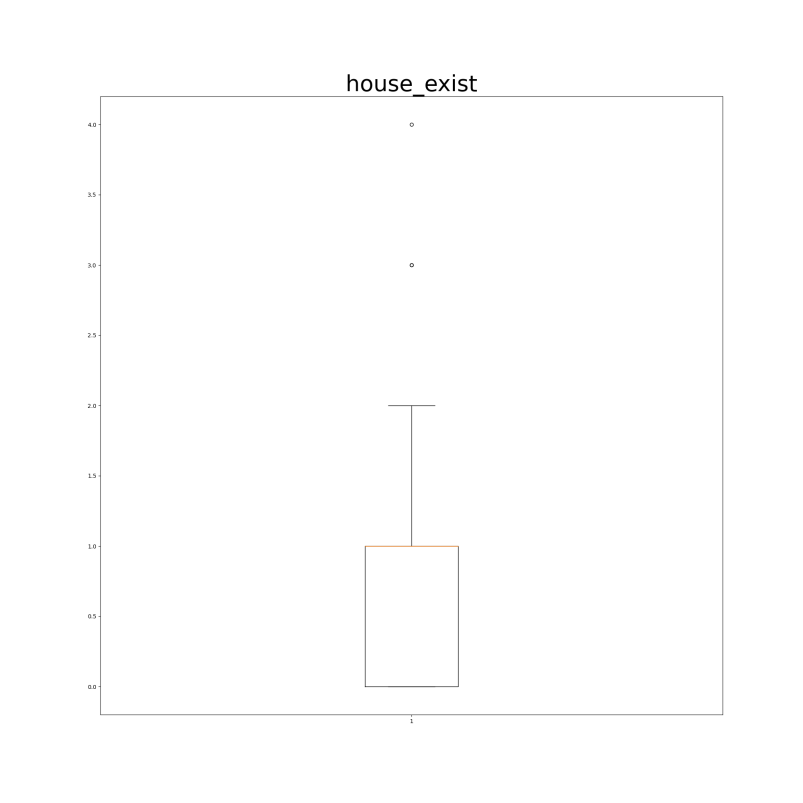
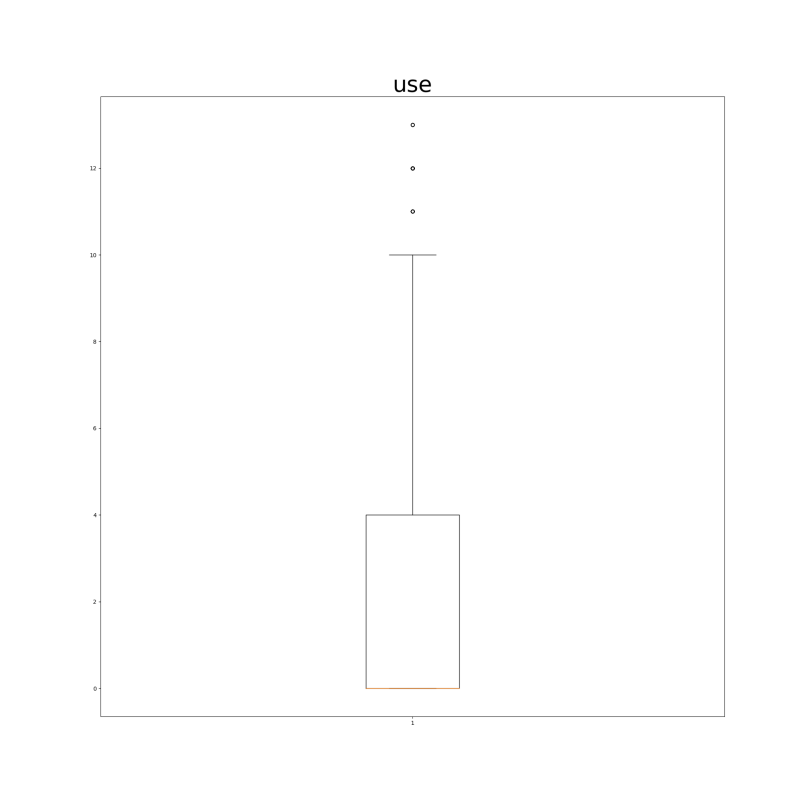
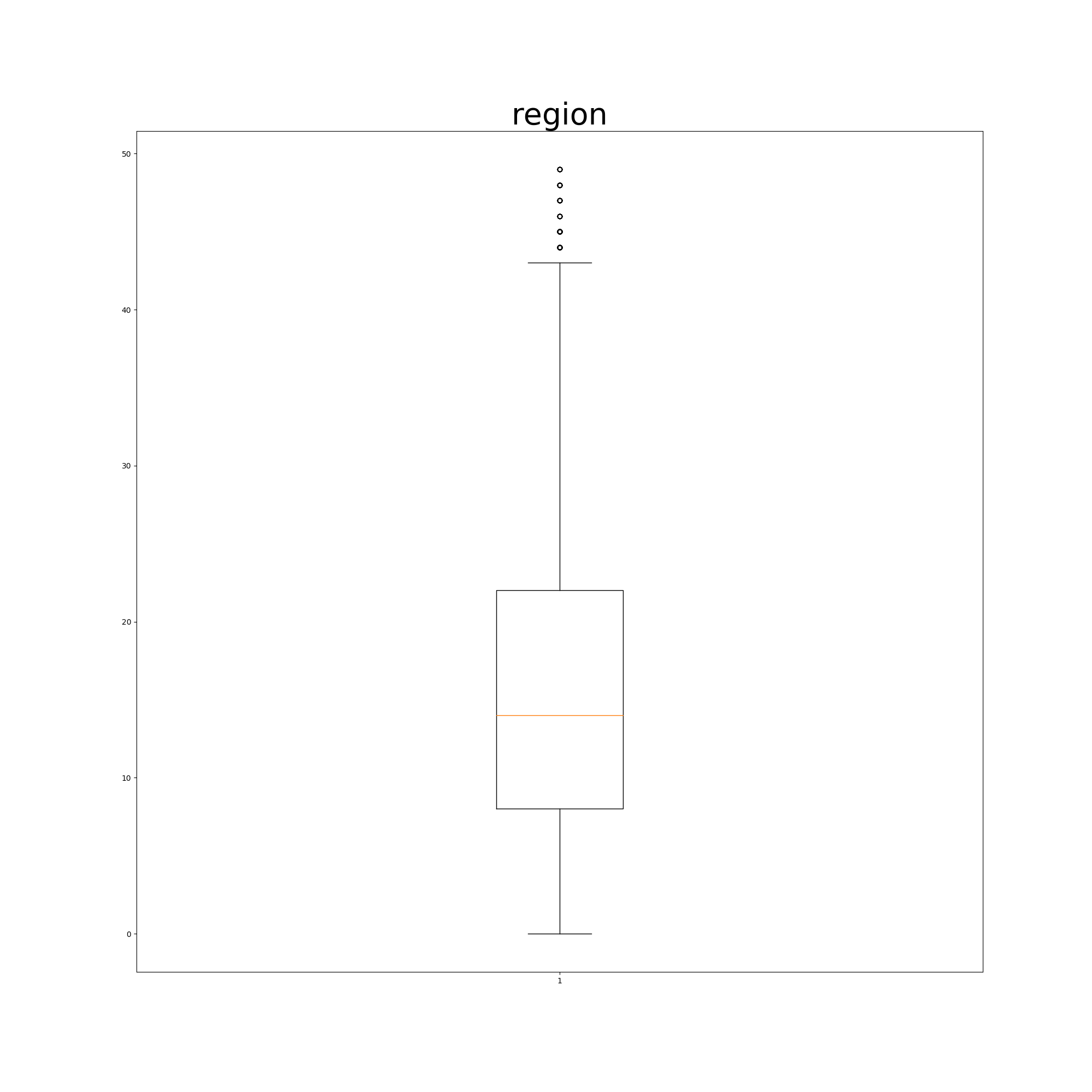
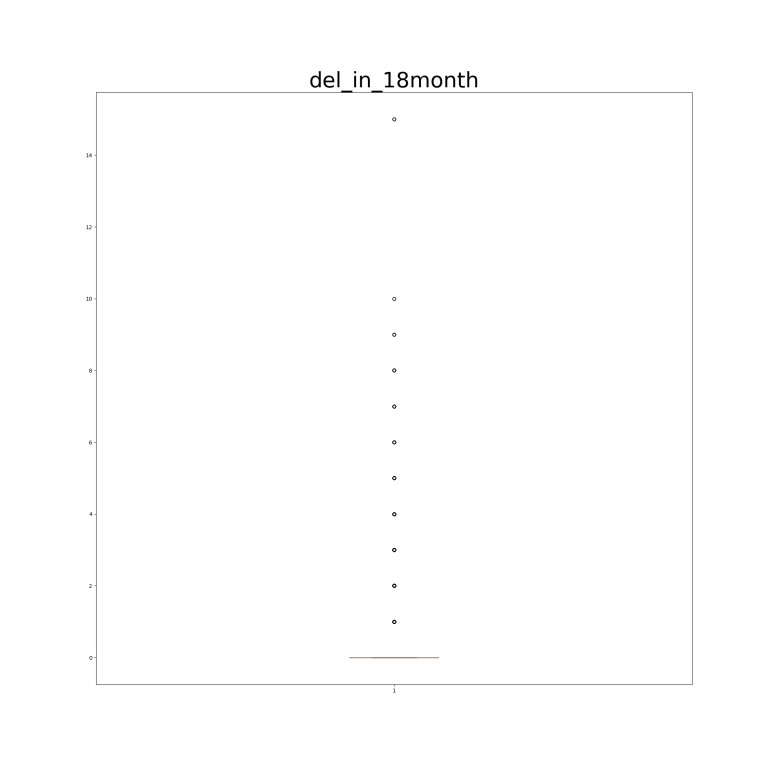
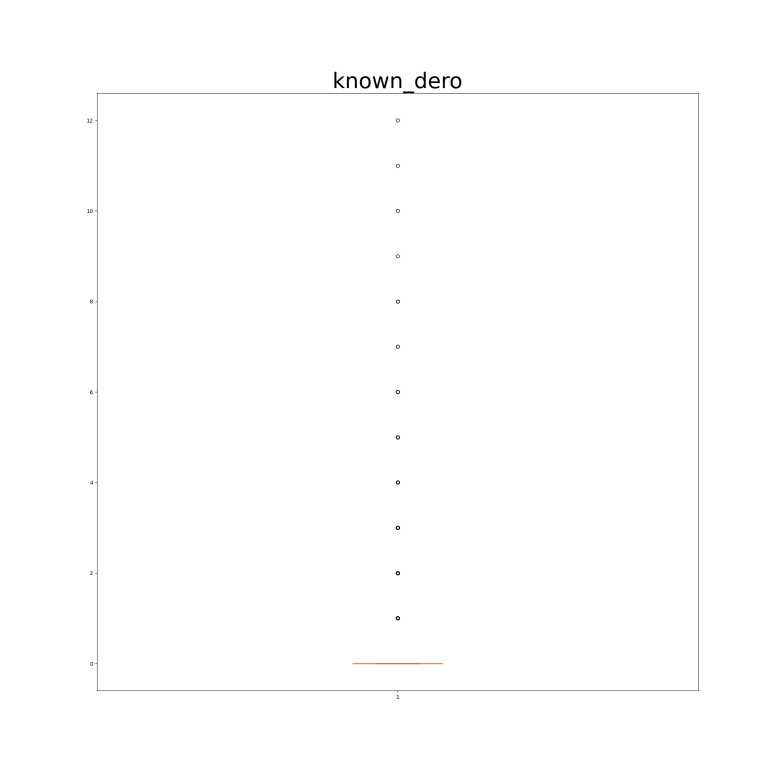
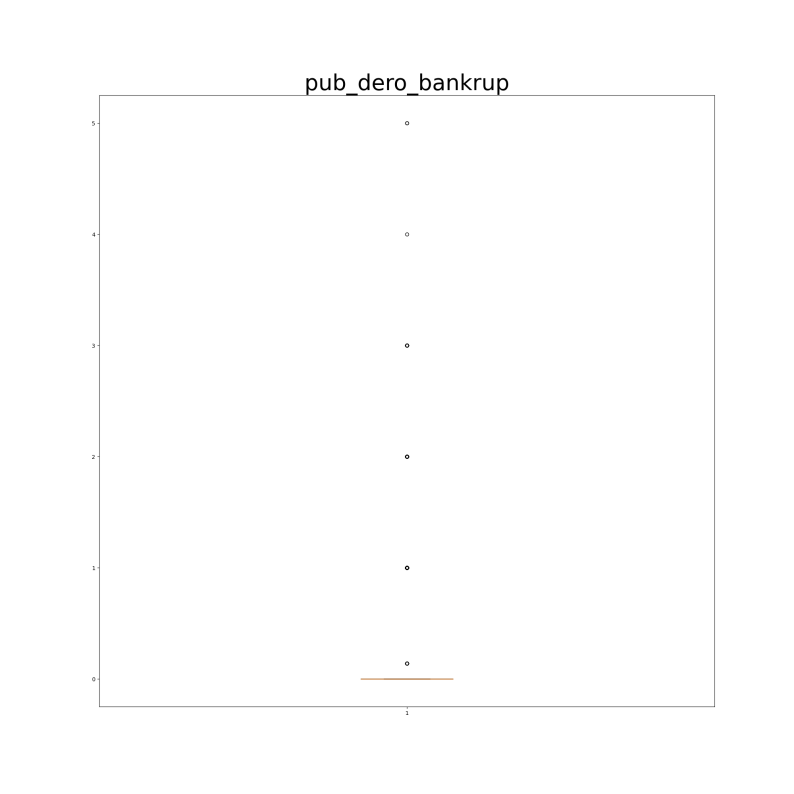
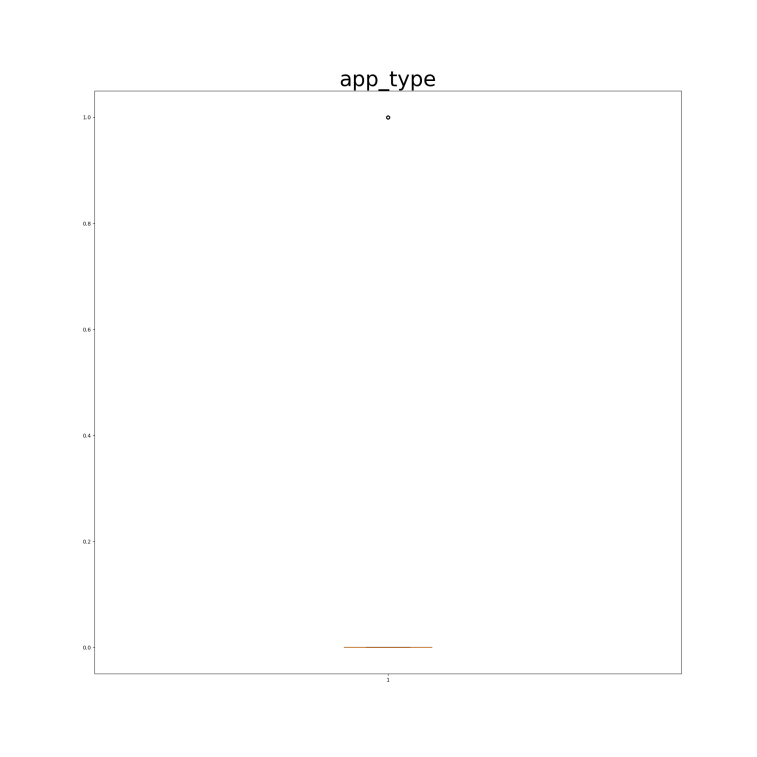
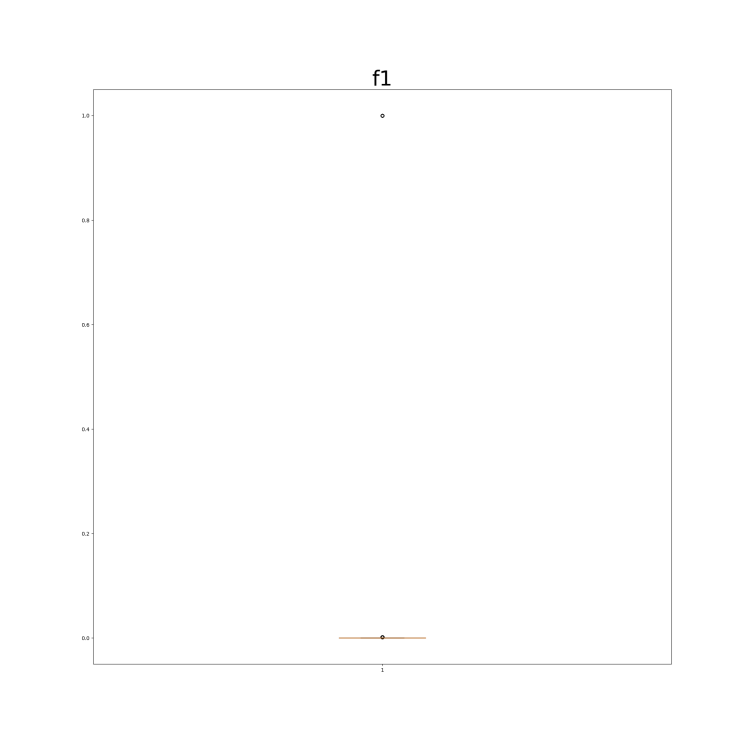
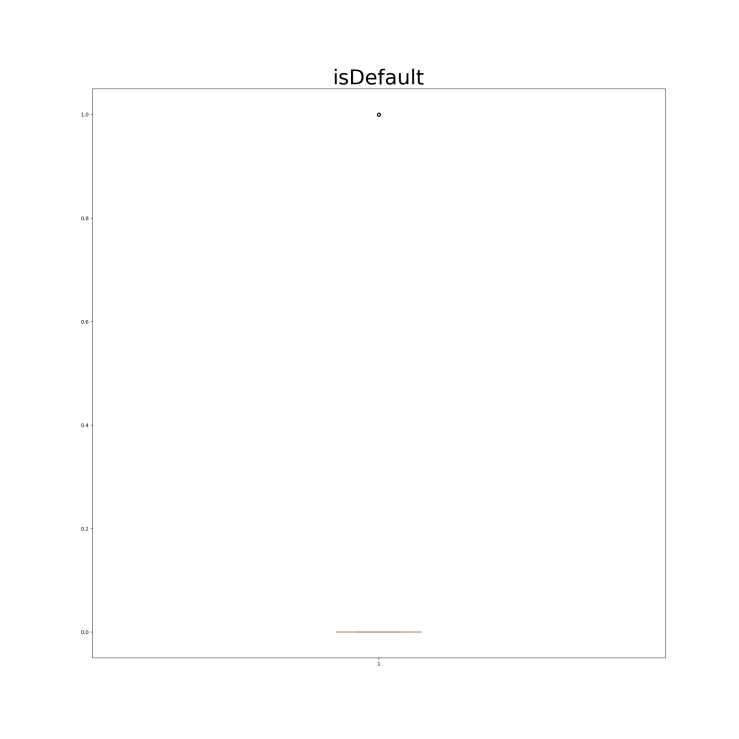
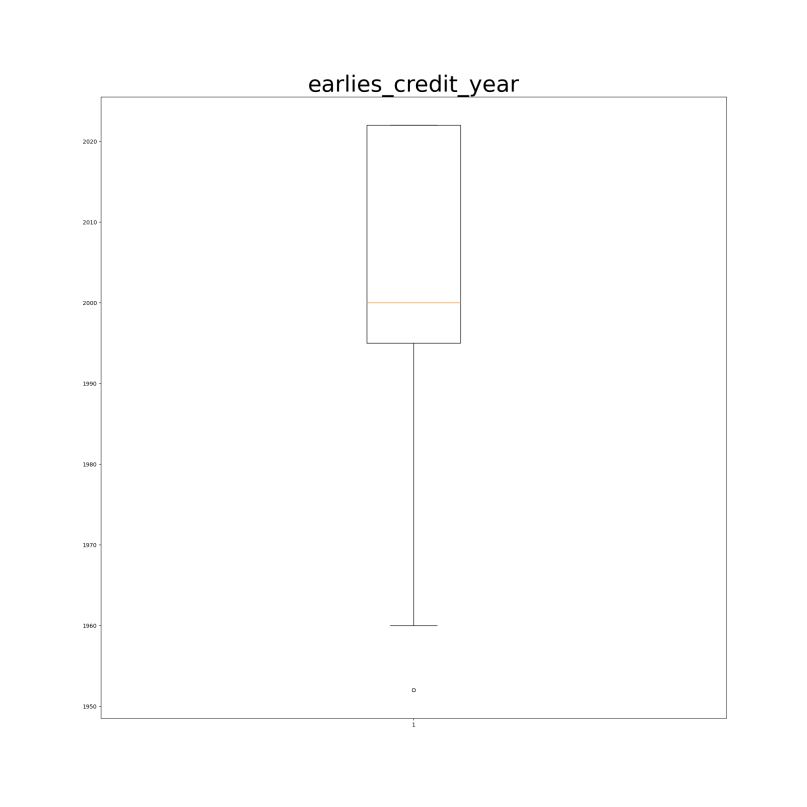
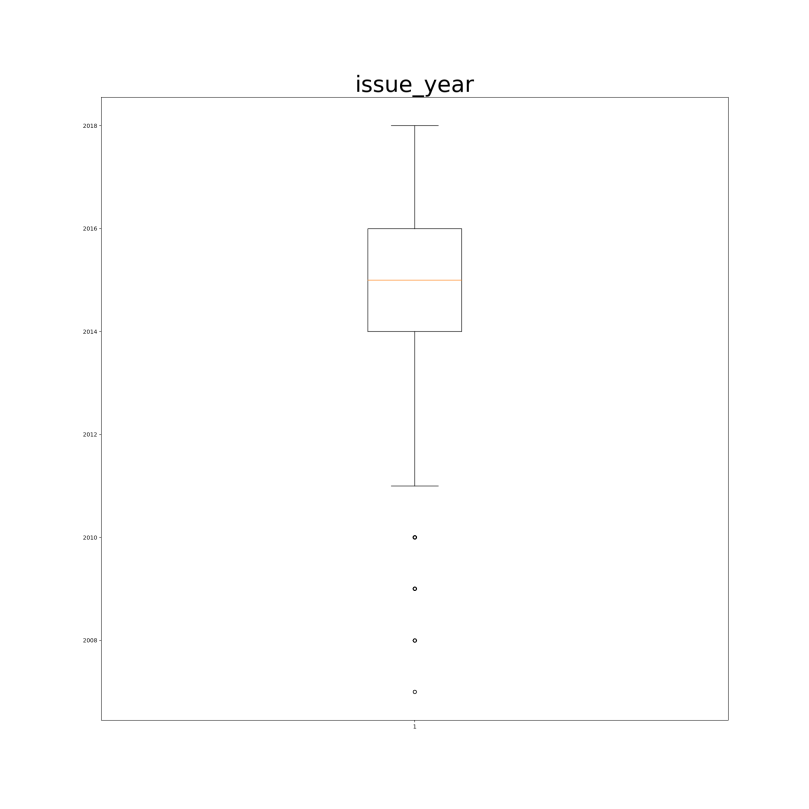
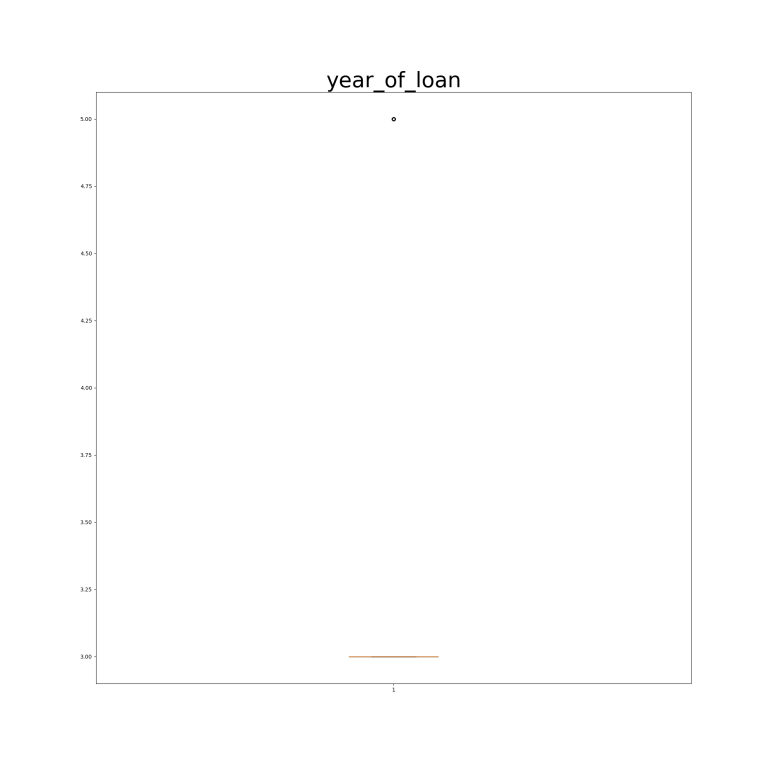
结果如下。

第一类箱线图。

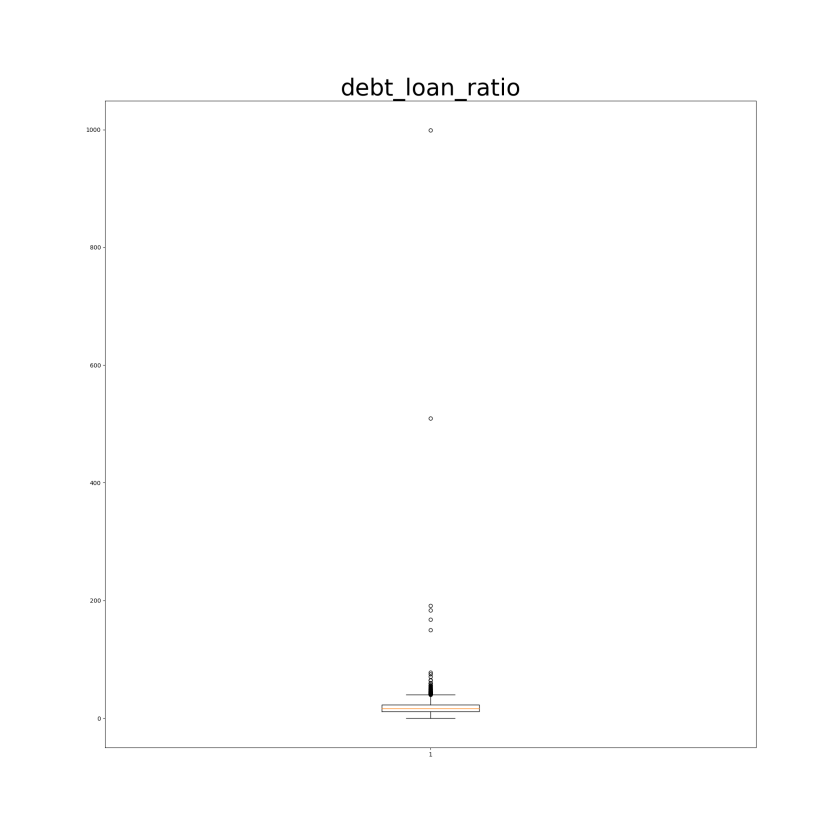
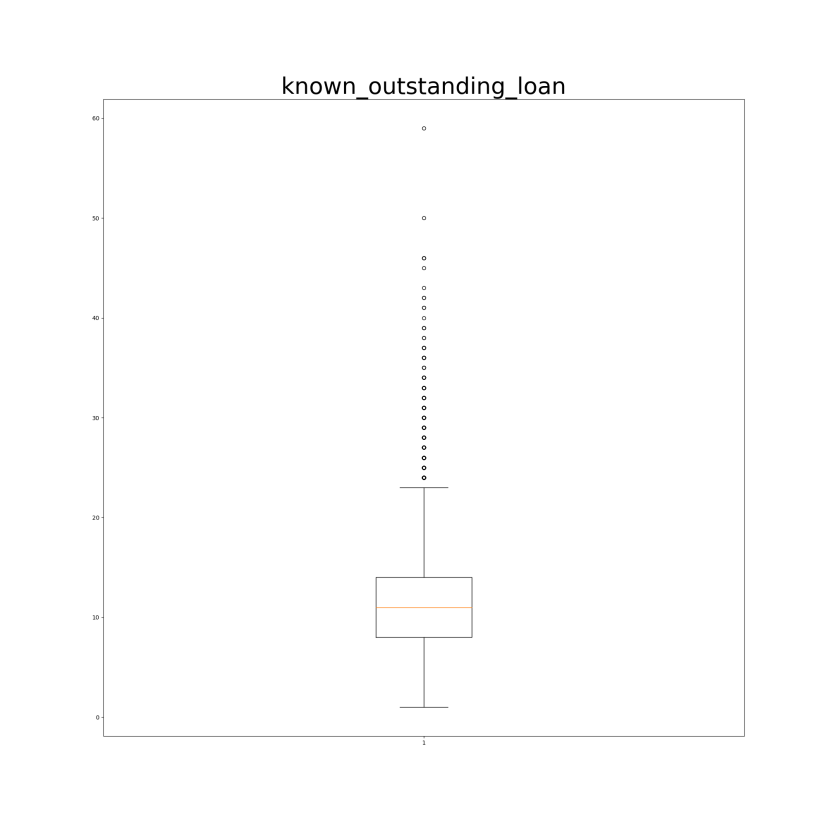
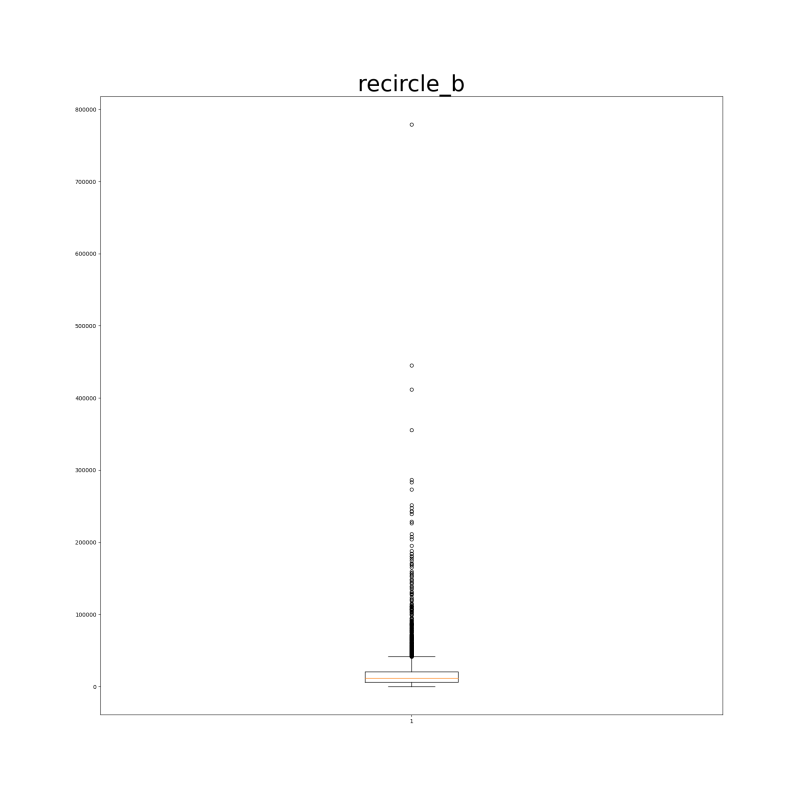
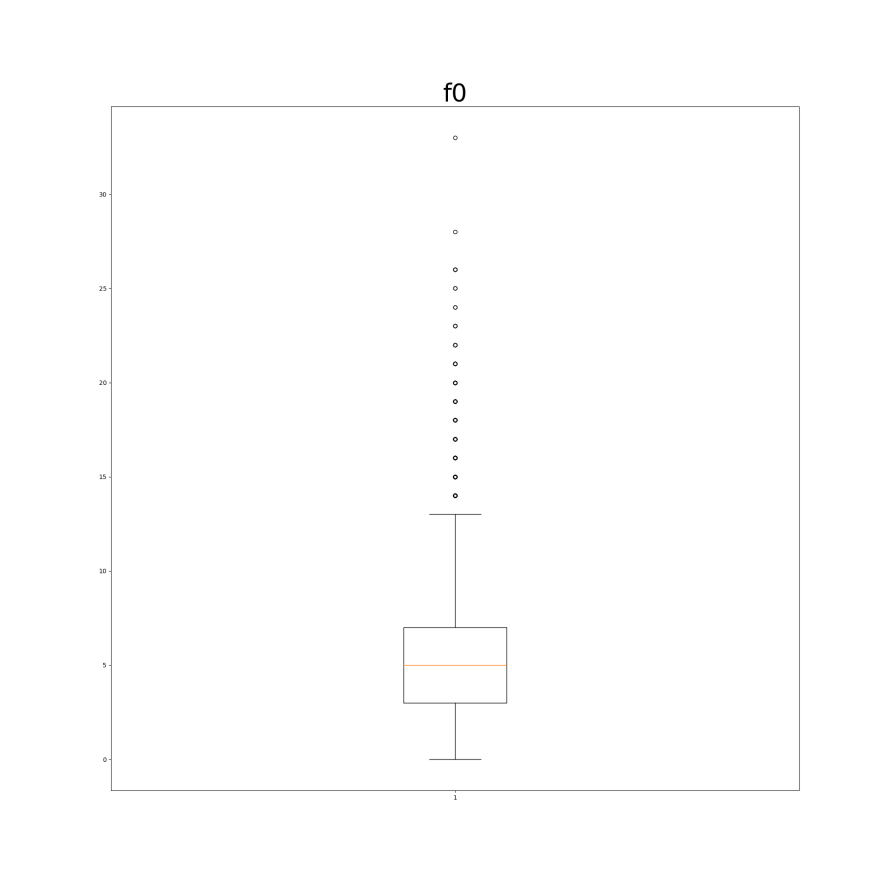
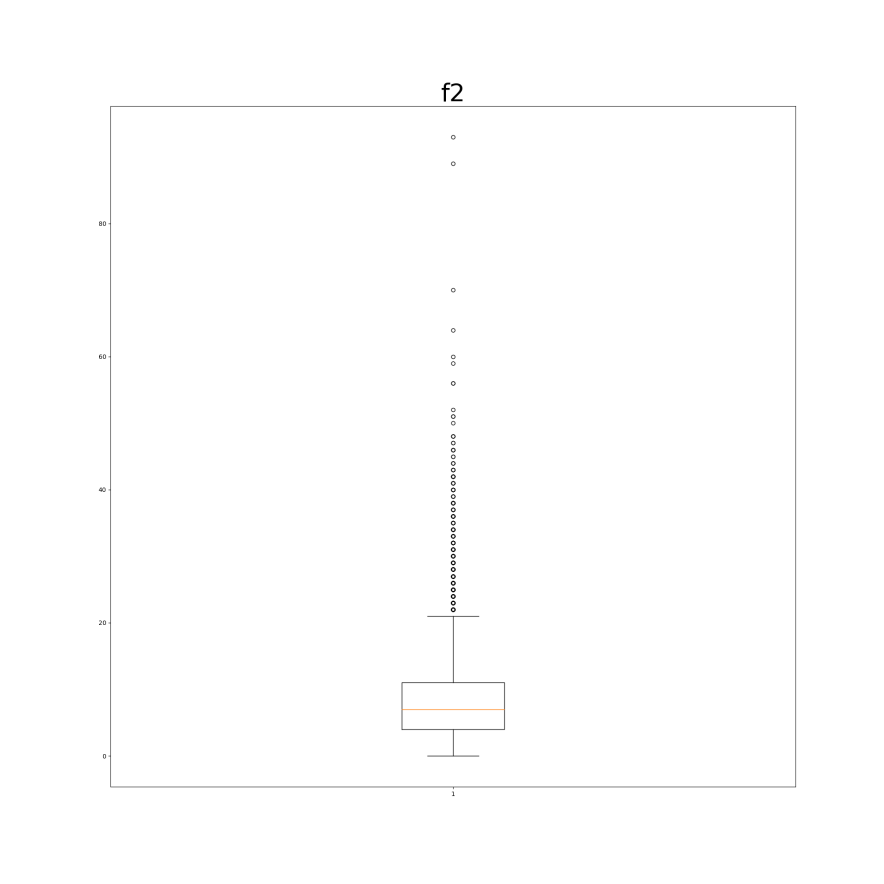
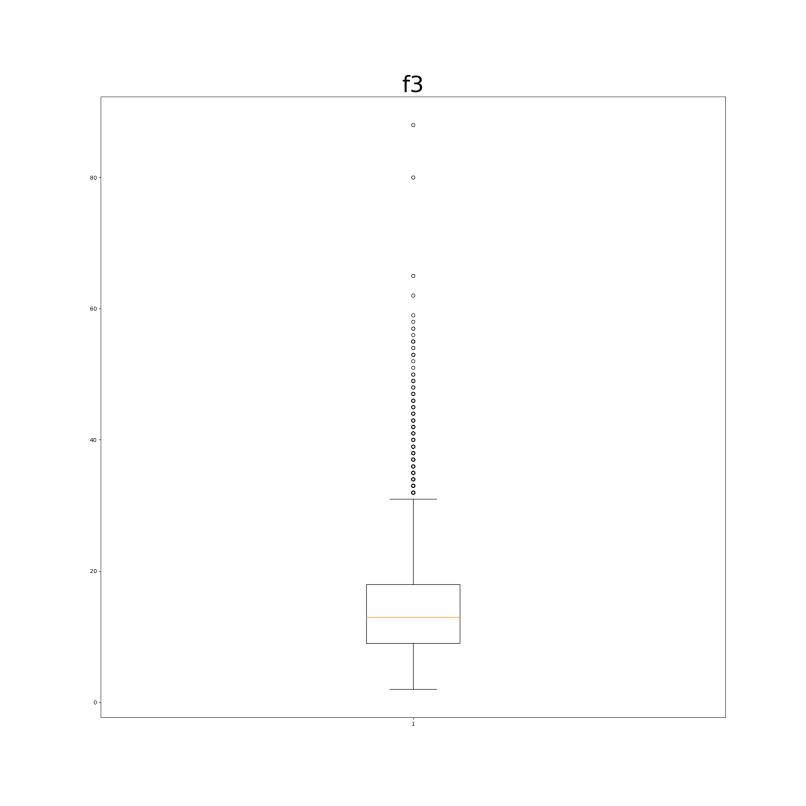
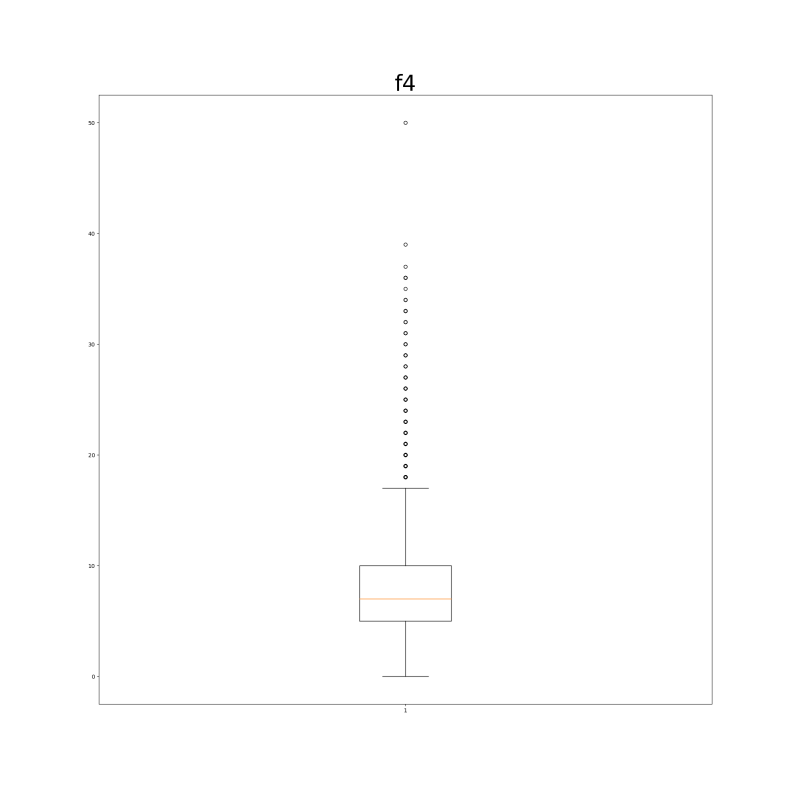
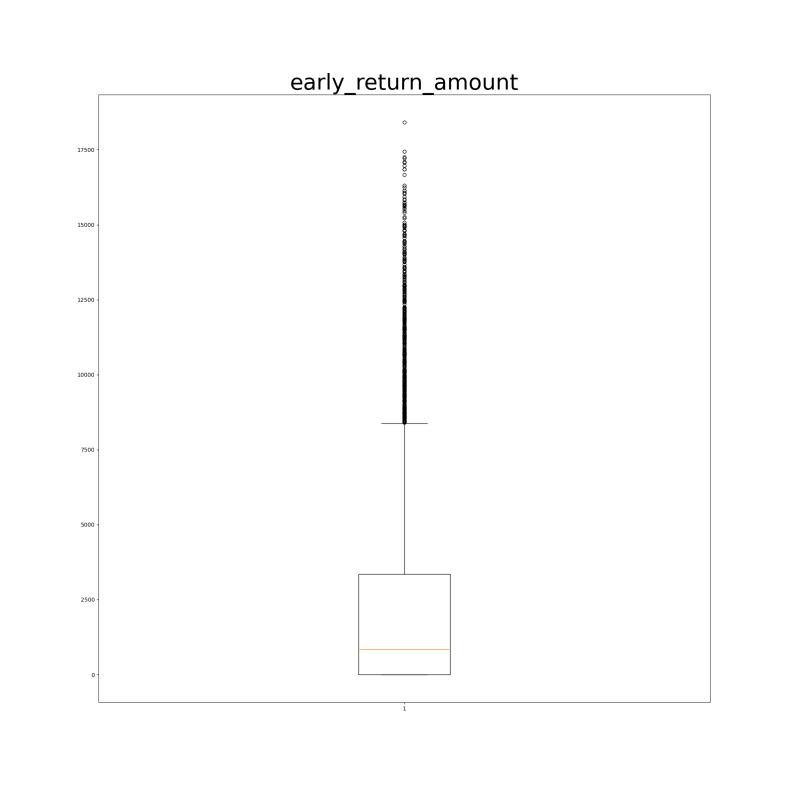
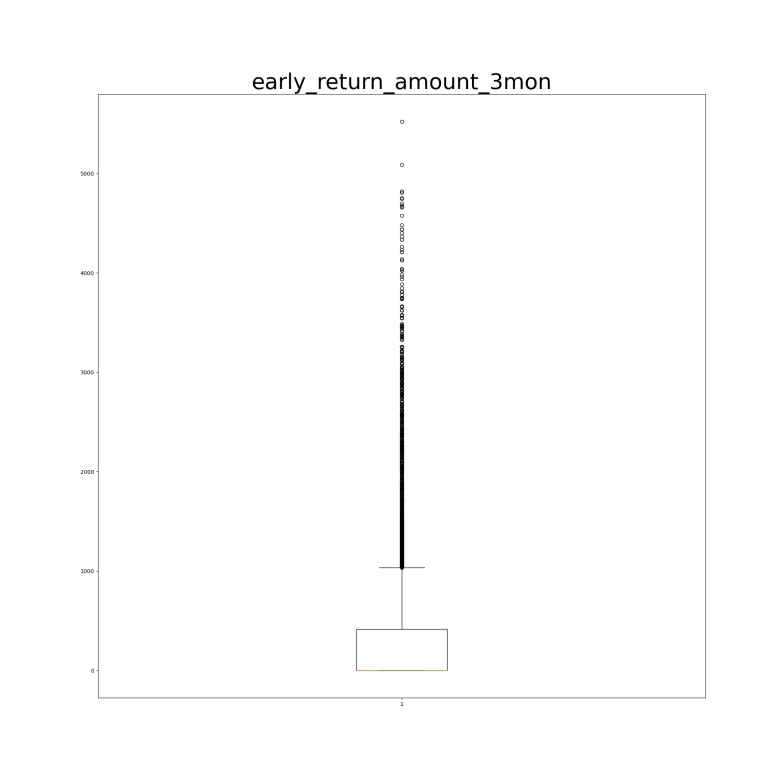




第二类箱线图。



第三类箱线图。



剩余特征箱线图无异常点，不在关注范围内。

可见，带有异常值的箱线图大致分为三类，一种是箱线图上限外有大量密集的异常点；另一种为箱线图上限外有零星几个异常点；第三种为箱线图上限外有大量密集的异常点，在这些密集异常点外还存在明显的离群点。

对于第一类箱线图对应的特征，我们认为其与我们将要检测的异常相关，不进行处理。对于第二类箱线图对应的特征，我们可以发现其中大部分均为标称数据，以箱线图检测异常无太大意义，所以仅考虑其中的计数数据。删除了house\_exist中的最高值，其余未进行处理。

对于第三类数据，我们可以发现其中密集离群点外的离群点距离密集离群点并不远，由于任务为异常检测，因此我们选择不处理这些离群点。仅有debt\_loan\_radio与recircle\_b两项，最高值对应的离群点距离下方密集离群点较远，影响较大。因此仅删除这两项的最高值，删除debt\_loan\_radio的次高值。经检验，删除上述几项后交叉检验模型auc值有所提升。

2.4.3孤立森林检测异常

使用孤立森林检测异常。对每一个属性都使用孤立森林检测异常，将每一个属性检测出的异常的index放入一个集合，最后统计集合的大小。

1. iforest=es.IsolationForest(n\_estimators=100, max\_samples='auto',
2. contamination=0.0005,
3. bootstrap=False,)
4. index=set('')
5. for i in data1.columns.values.tolist():
6. if (i!='loan\_id') and (i!='user\_id'):
7. tmp=pd.Series(iforest.fit\_predict(data1[[i]].values))
8. index.update(tmp[tmp==-1].index.values)
9. print('outliers number:',len(index))
10. print('outliers number ratio',len(index)/data1.shape[0])
11. #data1=data1.drop(list(index))

当异常比例为0.05时，得到的结果总共有6372条数据，占到了数据集的63%左右，显然结果较差，缩小范围再次尝试。



当异常比例为0.005时，得到889条异常数据，占比仍然略高，再次缩小异常比例范围。



当异常比例为0.0005时，得到63条异常数据，尝试进行删除，发现模型预测结果有几个百分点的下降，因此孤立森林仅用于异常检测，其检测出的异常值不进行处理。



删除后EasyEnsemble结果。

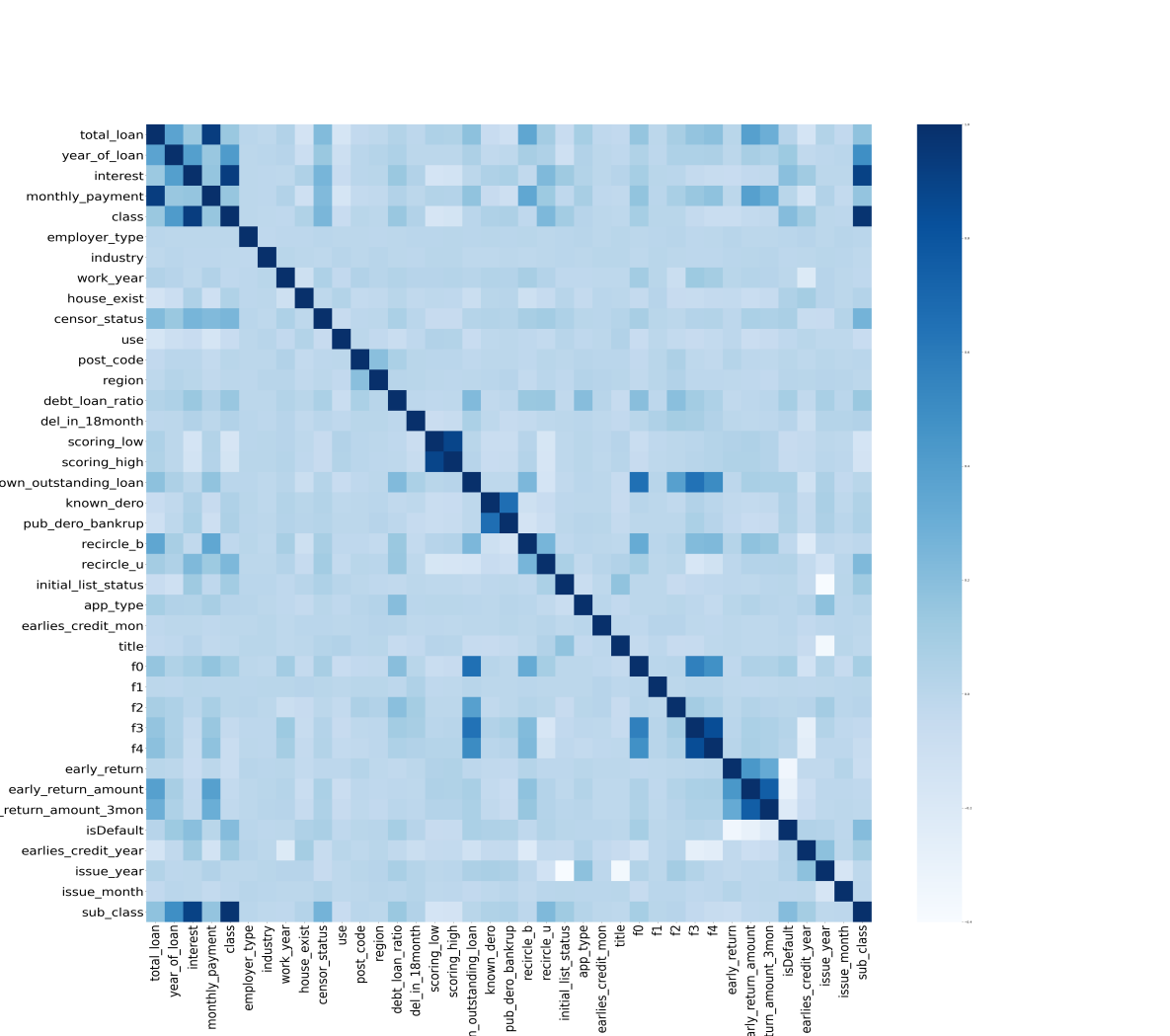
2.4.4异常处理

因此我们以箱线图结果为基础，删除了debt\_loan\_radio的最高值，house\_exist的最高值，recircle\_b的最高值。特别一提，EasyEnsemble模型删除了debt\_loan\_radio次高值后最低的结果由0.867提升到了0.88。

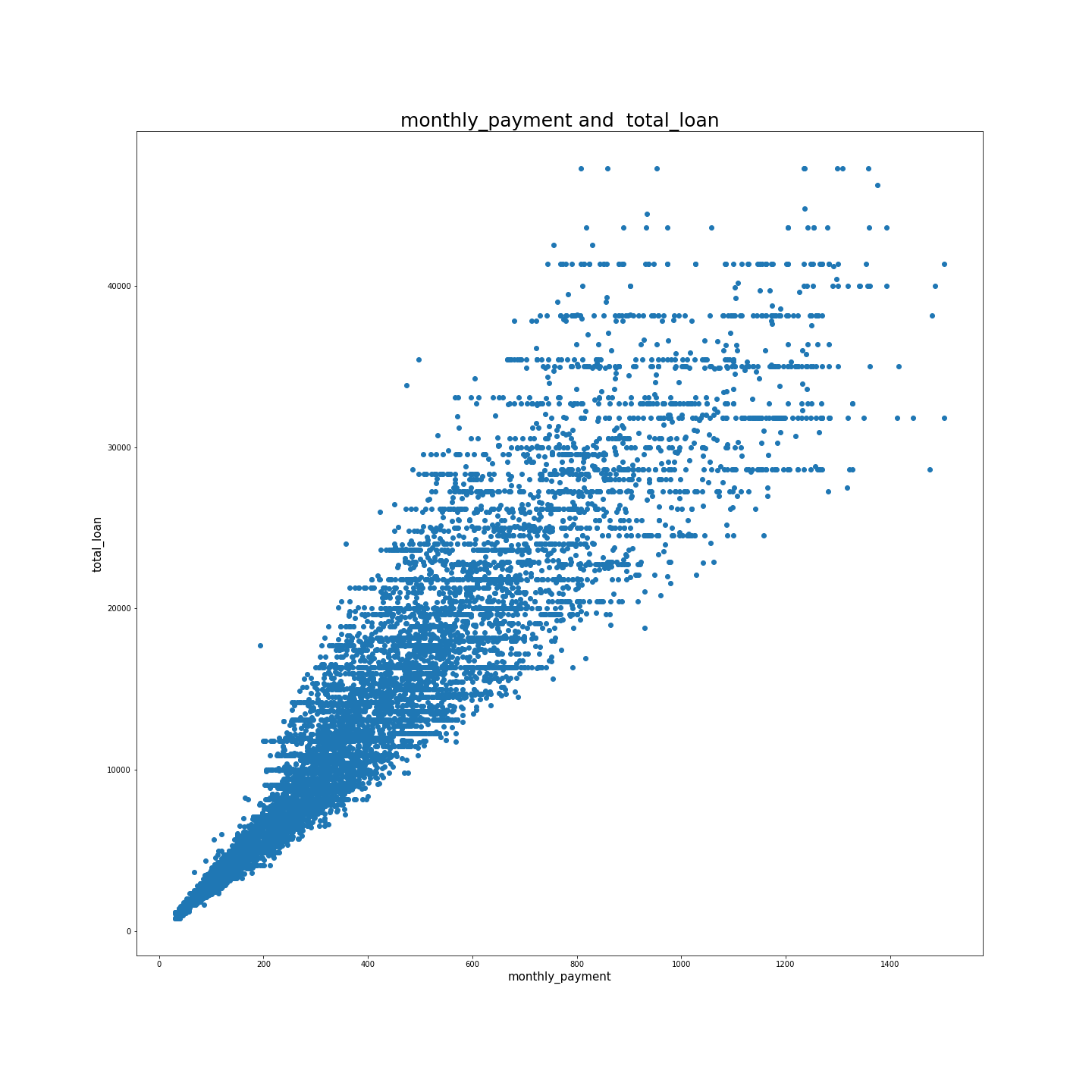
2.5噪音检测与处理

2.5.1检测属性间的相关性

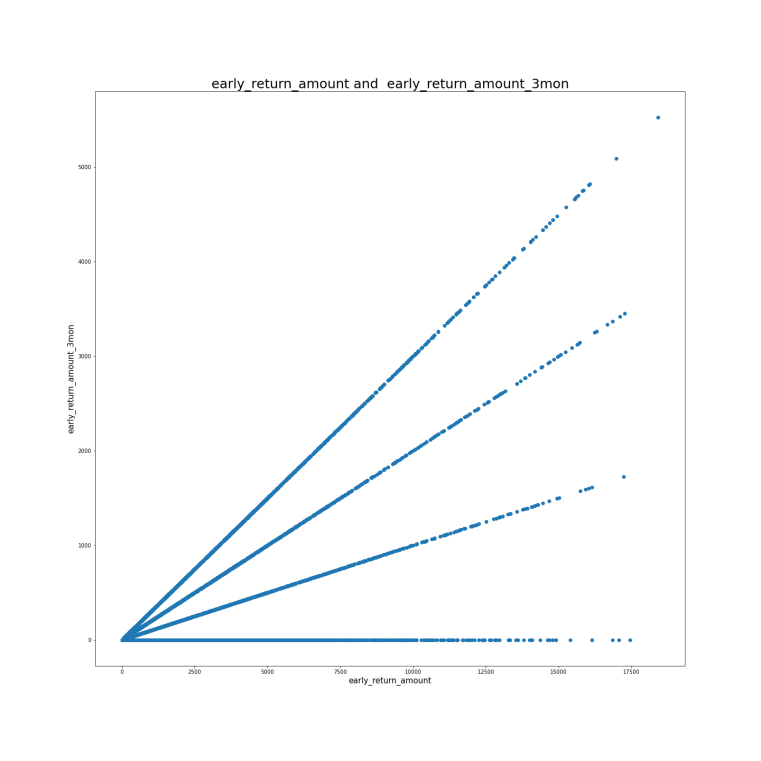
使用皮尔森相关系数检测特征间的相关性。结果如图。



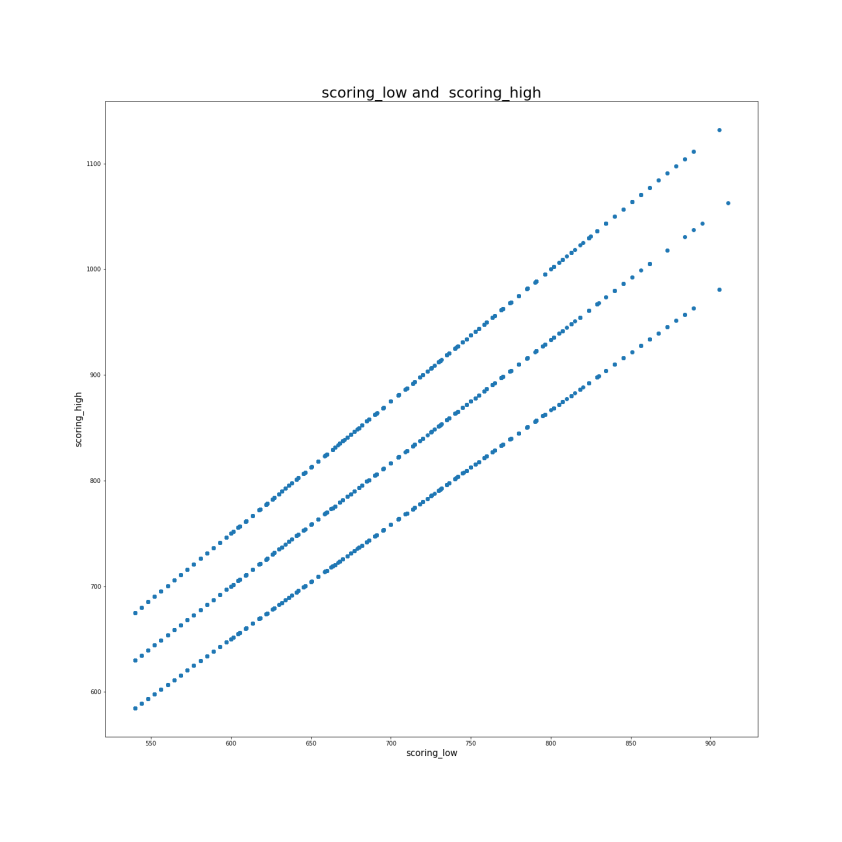
根据热力图，monthly\_payment与total\_loan，scoring\_high与scoring\_low，class与interest，f3与f4,early\_return\_amount与early\_return\_amount\_3mon有较强线性相关性。sub\_class从class中构建出，无疑与class和interest有较强相关性。绘制相关性较强的几个散点图寻找规律。



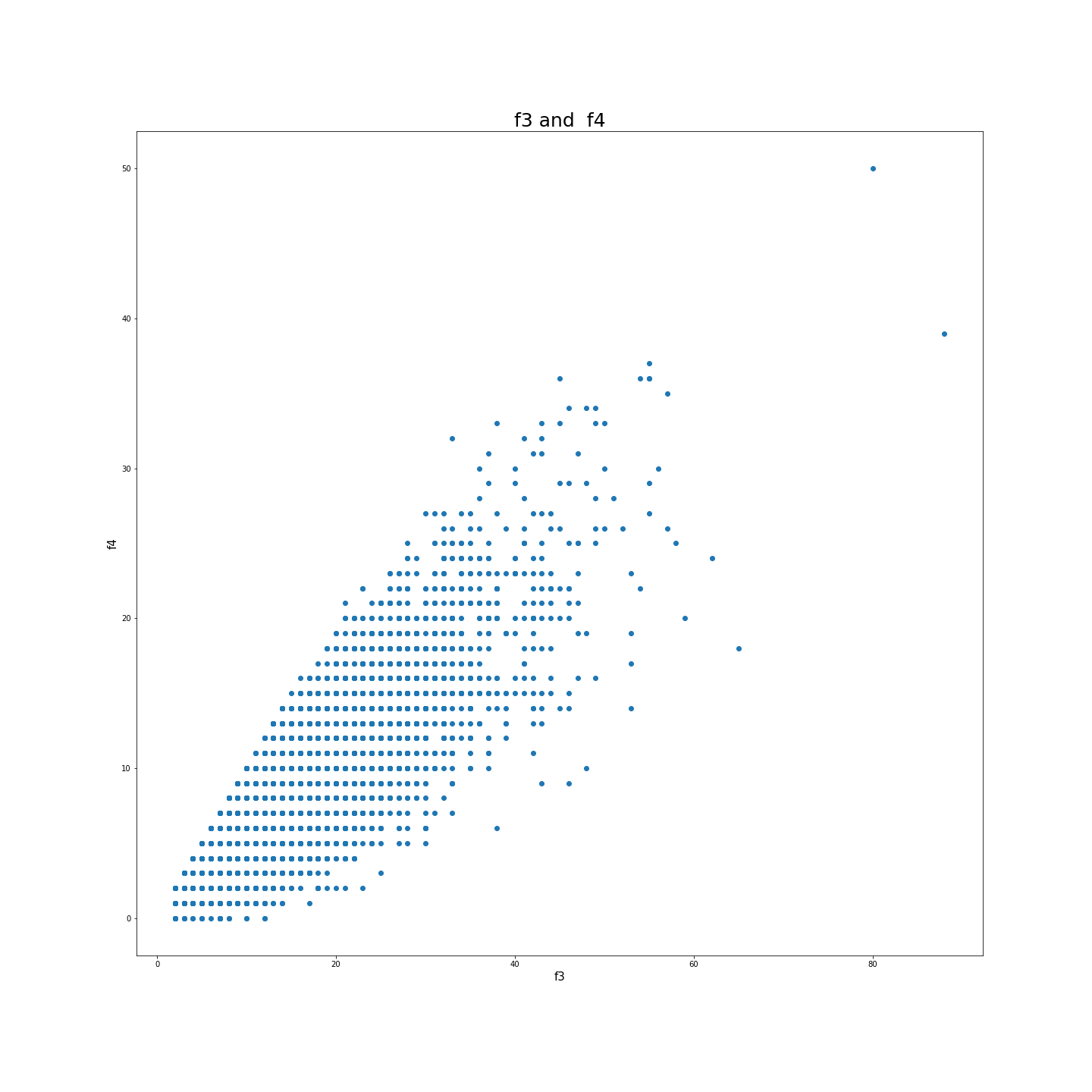
以monthly\_payment与total\_loan为纵横轴绘制散点图，可以发现在数值较小时，二者基本呈线性相关，但当数值较大时，数据随机性增大，方差增大。



以early\_return\_amount\_3mon与early\_return\_amount为横纵轴绘制散点图。可以发现散点图中存在明显的四条线。



以scoring\_high与scoring\_low为横纵轴绘制散点图。可以发现散点图中存在三条线且无明显异常点。



以f3、f4为横纵轴绘制散点图。可发现二者大致呈线性关系但有较大方差存在且存在明显离群点。

2.6数据规约

将数据采用Z-score进行归一化，方便特征构建与主成分分析。

1. data2:pd.DataFrame=(data1-data1.mean())/data1.std()

2.8特征选取

2.8.1概念介绍

特征选取是特征工程里的重要步骤，可以去除冗余的属性，提高模型精度。常用的特征选取方式有三类：Filter，Wrapper，Embedded。

Filter的大致思路为对每个特征 x\_i ，计算 x\_i 相对于类别标签 y 的信息量 S(i)，之后将S(i)按从大到小排序选取前k个特征。Filter包括Pearson相关系数，卡方验证，互信息和最大信息系数，距离相关系数，方差选择法这几种方法。

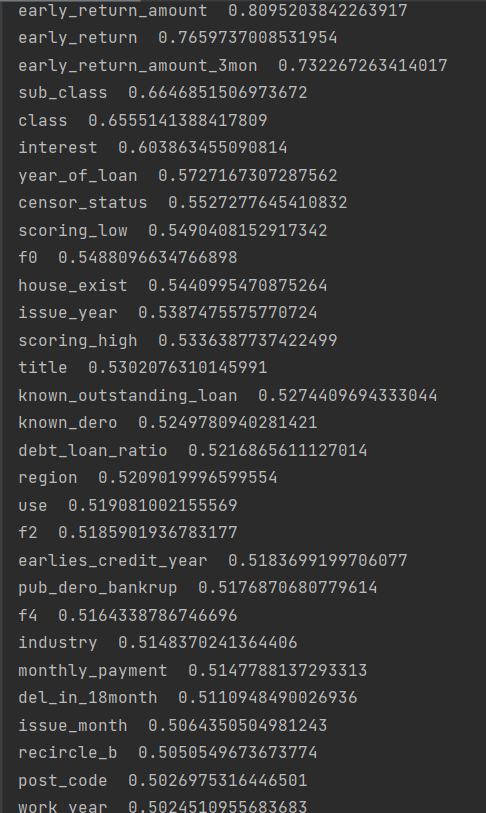
Wrapper为将训练集的待选取特征子集放入模型训练，根据误差大小选取适合的特征子集。

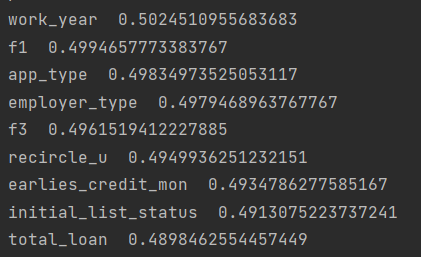
Embedded为先使用某些机器学习的模型进行训练，得到各个特征的权值系数，根据系数从大到小选择特征。

2.8.2特征选取具体实现

本次采用了Embedded方法，将所有特征以单个变量的形式放入随机森林中，以roc\_auc为评估标准进行5折交叉验证，只保留auc分数高于0.5的特征。

1. rfc=es.RandomForestClassifier(n\_estimators=60)
2. feature\_list=[]
3. for i in data1.columns.values:
4. if(i!='isDefault'):
5. score = sklearn.model\_selection.cross\_val\_score(rfc,data2[i].to\_numpy().reshape(-1,1),data1['isDefault'] , scoring='roc\_auc', cv=5)
6. feature\_list.append(score.mean())
7. tmp\_columns:list=data1.columns.values.tolist()
8. tmp\_columns.remove('isDefault')
9. feature:dict=dict(zip(tmp\_columns,feature\_list))
10. feature=dict(sorted(feature.items(),key=lambda x:x[1],reverse=True))
11. for i in feature:
12. print(i,"",feature[i])





2.8.3选取前后结果比较

特征选取前的结果



特征选取后的结果



对于单独的随机森林，特征选取能够小幅度降低了auc值。但对于EasyEnsemble

模型，特征选取提升了千分之一左右auc值。

5.3 随机森林+欠采样集成算法

5.3.1 欠采样集成算法介绍

欠采样是指选择投入模型的数据时减少一些类别数量占比较高的数据，使得投入模型的数据各类别比例都相等。欠采样的主要方法有随机欠采样，基于集成的欠采样。

随机欠采样指在抽样时随机减少多数类别数据已让各类别数据比例相等，其优点很明显，简单，但缺点也很明显，由于随机减少多数类数据，因此很可能让模型丢失重要特征。因此有了基于集成的欠采样。

基于集成的欠采样核心思路为在基于Bagging或Boosting构建基分类器时，采用一定的方式进行欠采样，使得进入基分类器的数据各分类占比相同，最后对基分类器结果进行集成。这样对于每个基分类器来说数据经过了欠采样处理，但总体来看多个基分类器集成避免了丢失重要特征。能够较好的解决欠采样方法的缺点。

5.3.2算法原理

基于集成的欠采样模型有多种，实现原理大同小异。这里仅简单介绍我们所尝试过的模型。包括RUSBoost,BalancedRandomForest,EasyEnsemble三种模型。

RUSBoost

RUSBoost是一种基于Adaboost与随机欠采样算法的模型。其主要原理为在使用Adaboost方法构建每一个基分类器时，采用随机欠采样的方式构建出训练集的子集，将该子集用于训练基分类器。

BalancedRandomForest

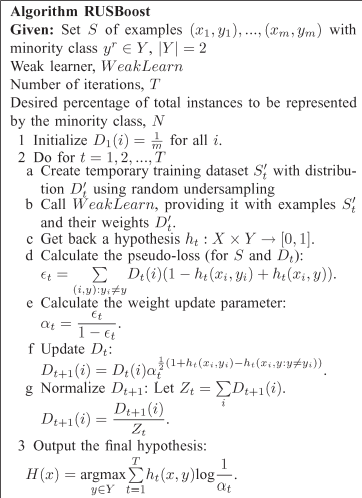
BalancedRandomForest是一种随机森林的改进模型。其与随机森林的区别在于在构建基分类器的时候，平衡随机森林对少数类进行有放回的抽样而对于多数类随机选取等量数据；而随机森林在构建基分类器时对整个数据集进行随机抽样。

EasyEnsemble

EasyEnsemble基于Bagging与随机欠采样构建。其主要原理为在使用Bagging方法构建基分类器时，随机从多数类中进行不放回采样构建一个与少数类集合大小相等的子集。将该子集与少数类用于训练集分类器。

5.3.3算法描述

RUSBoost



论文原文给出算法，以下以中文进行理解解释。

1.初始化每个基分类器权重

2.重复3步到8步直到构建出T个基分类器

3.采用随机欠采样构建一个数据集

4.训练基分类器

5.计算损失函数，公式如图2.d步

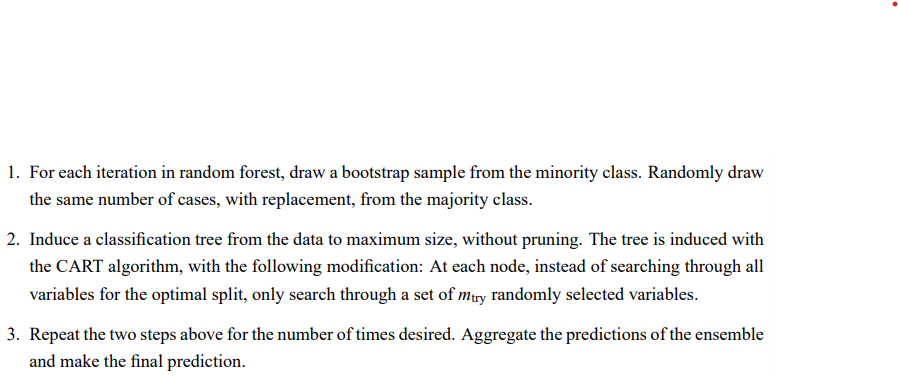
6.计算属性权重，公式如图2.e步

7.采用图示2.f步公式更新基分类器权重

8.对分类器权重进行标准化

9.输出如图第3步，其中为基分类器结果。

BalancedRandomForest



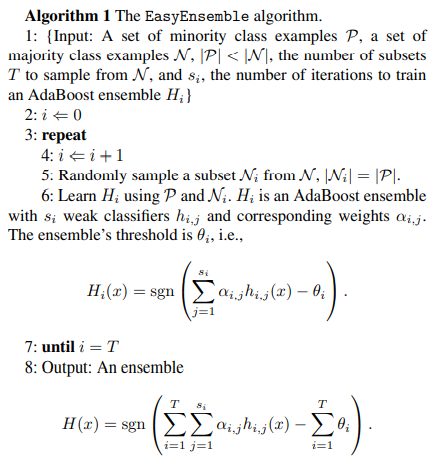
以上为论文原文给出算法流程，这里用中文进行理解。

1.对于随机森林的每棵决策树，随机从多数类样本中抽取出与少数样本等量的样本作为决策树的训练集

2.采用CART算法生成决策树，对于该决策树每个节点的判定属性，仅在一个随机选取的属性集中进行选择

3.在预定时间内重复上述两步骤。

EasyEnsemble



这是论文原文给出的具体算法流程，由于论文原文的基分类器采用了AdaBoost，上述算法给出的集成函数AdaBoost相关，因此下面给出以任意模型作为基分类器的算法流程。

1.{输入：少数类别集合P，多数类别集合N，基分类器个数T}

2.0

3.重复4到6步直到i=T：

4.

5.对多数类别集合N进行随机欠采样构建子集Ni，其中

6.用集合Ni和集合P训练基分类器，基分类器的输出为

7.输出：

5.3.4算法实现

由于具体工程中均采用调用imblearn来应用这些算法，这里以imblearn库的源代码来介绍这些算法的实现。

RUSboost方法实现。

首先可以看到该类继承了AdaBoostClassifier类



以下为该类的fit训练函数

1. def fit(self, X, y, sample\_weight=None):
2. check\_target\_type(y)
3. self.samplers\_ = []
4. self.pipelines\_ = []
5. super().fit(X, y, sample\_weight)
6. return self

其中check\_target\_type检查方法用于检查输入y的类型，与具体算法实现无关，可以发现调用了父类的fit方法。

以下是其父类AdaBoostClassifier的fit方法，我们发现在验证了algorithm是否合法的时候，其返回其父类fit方法

1. def fit(self, X, y, sample\_weight=None):
2. if self.algorithm not in ("SAMME", "SAMME.R"):
3. raise ValueError(
4. "Algorithm must be 'SAMME' or 'SAMME.R'."
5. f" Got {self.algorithm!r} instead."
6. )
7. return super().fit(X, y, sample\_weight)

查看AdaBoostClassifier的父类 的fit方法，可以发现该方法实现了用Boosting方法进行集成。其中有调用self.\_validate\_estimator()方法，由于该方法在AdaBoostClassifier与RUSBoost

Classifier中均被重写过，所以我们接下来查看RUSBoostClassifier的该方法。

1. def fit(self, X, y, sample\_weight=None):
2. """Build a boosted classifier/regressor from the training set (X, y).
3. *# Validate scalar parameters*
4. check\_scalar(
5. self.n\_estimators,
6. "n\_estimators",
7. target\_type=numbers.Integral,
8. min\_val=1,
9. include\_boundaries="left",
10. )
11. check\_scalar(
12. self.learning\_rate,
13. "learning\_rate",
14. target\_type=numbers.Real,
15. min\_val=0,
16. include\_boundaries="neither",
17. )
18. X, y = self.\_validate\_data(
19. X,
20. y,
21. accept\_sparse=["csr", "csc"],
22. ensure\_2d=True,
23. allow\_nd=True,
24. dtype=None,
25. y\_numeric=is\_regressor(self),
26. )
27. sample\_weight = \_check\_sample\_weight(
28. sample\_weight, X, np.float64, copy=True, only\_non\_negative=True
29. )
30. sample\_weight /= sample\_weight.sum()
31. # Check parameters
32. self.\_validate\_estimator()
33. # Clear any previous fit results
34. self.estimators\_ = []
35. self.estimator\_weights\_ = np.zeros(self.n\_estimators, dtype=np.float64)
36. self.estimator\_errors\_ = np.ones(self.n\_estimators, dtype=np.float64)
37. # Initialization of the random number instance that will be used to
38. *# generate a seed at each iteration*
39. random\_state = check\_random\_state(self.random\_state)
40. for iboost in range(self.n\_estimators):
41. *# Boosting step*
42. sample\_weight, estimator\_weight, estimator\_error = self.\_boost(
43. iboost, X, y, sample\_weight, random\_state
44. )
45. # Early termination
46. if sample\_weight is None:
47. break
48. self.estimator\_weights\_[iboost] = estimator\_weight
49. self.estimator\_errors\_[iboost] = estimator\_error
50. # Stop if error is zero
51. if estimator\_error == 0:
52. break
53. sample\_weight\_sum = np.sum(sample\_weight)
54. if not np.isfinite(sample\_weight\_sum):
55. warnings.warn(
56. "Sample weights have reached infinite values,"
57. f" at iteration {iboost}, causing overflow. "
58. "Iterations stopped. Try lowering the learning rate.",
59. stacklevel=2,
60. )
61. break
62. # Stop if the sum of sample weights has become non-positive
63. if sample\_weight\_sum <= 0:
64. break
65. if iboost < self.n\_estimators - 1:
66. *# Normalize*
67. sample\_weight /= sample\_weight\_sum
68. return self

以下是RUSBoostClassifier的\_validate\_estimator，很明显base\_sampler\_参数被替换为了随机欠采样模型。

1. def \_validate\_estimator(self):
2. """Check the estimator and the n\_estimator attribute, set the
3. `base\_estimator\_` attribute."""
4. super().\_validate\_estimator()
5. self.base\_sampler\_ = RandomUnderSampler(
6. sampling\_strategy=self.sampling\_strategy,
7. replacement=self.replacement,
8. )

以下是AdaBoostClassifier的\_boost方法，其中返回的\_boost\_real与\_boost\_discrete方法均于被重写。

1. def \_boost(self, iboost, X, y, sample\_weight, random\_state):
2. if self.algorithm == "SAMME.R":
3. return self.\_boost\_real(iboost, X, y, sample\_weight, random\_state)
4. else:  # elif self.algorithm == "SAMME":
5. return self.\_boost\_discrete(iboost, X, y, sample\_weight, random\_state)

以下为RUSBoostClassifier的\_boost\_real方法，第5行可以看到采用随机欠采样对投入基分类器决策树的样本进行了处理，而第7行则训练了一棵决策树。

1. def \_boost\_real(self, iboost, X, y, sample\_weight, random\_state):
2. """Implement a single boost using the SAMME.R real algorithm."""
3. estimator, sampler = self.\_make\_sampler\_estimator(random\_state=random\_state)
4. X\_res, y\_res = sampler.fit\_resample(X, y)
5. sample\_weight\_res = \_safe\_indexing(sample\_weight, sampler.sample\_indices\_)
6. estimator.fit(X\_res, y\_res, sample\_weight=sample\_weight\_res)
7. y\_predict\_proba = estimator.predict\_proba(X)
8. if iboost == 0:
9. self.classes\_ = getattr(estimator, "classes\_", None)
10. self.n\_classes\_ = len(self.classes\_)
11. y\_predict = self.classes\_.take(np.argmax(y\_predict\_proba, axis=1), axis=0)
12. *# Instances incorrectly classified*
13. incorrect = y\_predict != y
14. *# Error fraction*
15. estimator\_error = np.mean(np.average(incorrect, weights=sample\_weight, axis=0))
16. *# Stop if classification is perfect*
17. if estimator\_error <= 0:
18. return sample\_weight, 1.0, 0.0
19. *# Construct y coding as described in Zhu et al [2]:*
20. *#*
21. *#    y\_k = 1 if c == k else -1 / (K - 1)*
22. *#*
23. *# where K == n\_classes\_ and c, k in [0, K) are indices along the second*
24. *# axis of the y coding with c being the index corresponding to the true*
25. *# class label.*
26. n\_classes = self.n\_classes\_
27. classes = self.classes\_
28. y\_codes = np.array([-1.0 / (n\_classes - 1), 1.0])
29. y\_coding = y\_codes.take(classes == y[:, np.newaxis])
30. *# Displace zero probabilities so the log is defined.*
31. *# Also fix negative elements which may occur with*
32. *# negative sample weights.*
33. proba = y\_predict\_proba  *# alias for readability*
34. np.clip(proba, np.finfo(proba.dtype).eps, None, out=proba)
35. *# Boost weight using multi-class AdaBoost SAMME.R alg*
36. estimator\_weight = (
37. -1.0
38. \* self.learning\_rate
39. \* ((n\_classes - 1.0) / n\_classes)
40. \* (y\_coding \* np.log(y\_predict\_proba)).sum(axis=1)
41. )
42. *# Only boost the weights if it will fit again*
43. if not iboost == self.n\_estimators - 1:
44. *# Only boost positive weights*
45. sample\_weight \*= np.exp(
46. estimator\_weight \* ((sample\_weight > 0) | (estimator\_weight < 0))
47. )
48. return sample\_weight, 1.0, estimator\_error

EasyEnsemble算法实现

EasyEnsemble继承了BaggingClassifier类。



EasyEnsemble重写了\_validate\_estimator方法，可以看到19行到30行将基分类器替换为了管道类型，将随机欠采样的结果作为了基分类器的输入。

1. def \_validate\_estimator(self, default=AdaBoostClassifier()):
2. """Check the estimator and the n\_estimator attribute, set the
3. `base\_estimator\_` attribute."""
4. if not isinstance(self.n\_estimators, (numbers.Integral, np.integer)):
5. raise ValueError(
6. f"n\_estimators must be an integer, " f"got {type(self.n\_estimators)}."
7. )
8. if self.n\_estimators <= 0:
9. raise ValueError(
10. f"n\_estimators must be greater than zero, " f"got {self.n\_estimators}."
11. )
12. if self.base\_estimator is not None:
13. base\_estimator = clone(self.base\_estimator)
14. else:
15. base\_estimator = clone(default)
16. self.base\_estimator\_ = Pipeline(
17. [
18. (
19. "sampler",
20. RandomUnderSampler(
21. sampling\_strategy=self.\_sampling\_strategy,
22. replacement=self.replacement,
23. ),
24. ),
25. ("classifier", base\_estimator),
26. ]
27. )
28. *# Stop if classification is perfect*
29. if estimator\_error <= 0:
30. return sample\_weight, 1.0, 0.0
31. n\_classes = self.n\_classes\_
32. classes = self.classes\_
33. y\_codes = np.array([-1.0 / (n\_classes - 1), 1.0])
34. y\_coding = y\_codes.take(classes == y[:, np.newaxis])
35. proba = y\_predict\_proba  *# alias for readability*
36. np.clip(proba, np.finfo(proba.dtype).eps, None, out=proba)
37. *# Boost weight using multi-class AdaBoost SAMME.R alg*
38. estimator\_weight = (
39. -1.0
40. \* self.learning\_rate
41. \* ((n\_classes - 1.0) / n\_classes)
42. \* (y\_coding \* np.log(y\_predict\_proba)).sum(axis=1)
43. )
44. *# Only boost the weights if it will fit again*
45. if not iboost == self.n\_estimators - 1:
46. *# Only boost positive weights*
47. sample\_weight \*= np.exp(
48. estimator\_weight \* ((sample\_weight > 0) | (estimator\_weight < 0))
49. )
50. return sample\_weight, 1.0, estimator\_error

而bagging算法的实现则在BaggingClassifier的\_fit方法中，具体如下：

1. def \_fit(
2. self,
3. X,
4. y,
5. max\_samples=None,
6. max\_depth=None,
7. sample\_weight=None,
8. check\_input=True,
9. ):
10. random\_state = check\_random\_state(self.random\_state)
11. if sample\_weight is not None:
12. sample\_weight = \_check\_sample\_weight(sample\_weight, X, dtype=None)
13. *# Remap output*
14. n\_samples = X.shape[0]
15. self.\_n\_samples = n\_samples
16. y = self.\_validate\_y(y)
17. *# Check parameters*
18. self.\_validate\_estimator()
19. if max\_depth is not None:
20. self.base\_estimator\_.max\_depth = max\_depth
21. *# Validate max\_samples*
22. if max\_samples is None:
23. max\_samples = self.max\_samples
24. elif not isinstance(max\_samples, numbers.Integral):
25. max\_samples = int(max\_samples \* X.shape[0])
26. if not (0 < max\_samples <= X.shape[0]):
27. raise ValueError("max\_samples must be in (0, n\_samples]")
28. *# Store validated integer row sampling value*
29. self.\_max\_samples = max\_samples
30. *# Validate max\_features*
31. if isinstance(self.max\_features, numbers.Integral):
32. max\_features = self.max\_features
33. elif isinstance(self.max\_features, float):
34. max\_features = self.max\_features \* self.n\_features\_in\_
35. else:
36. raise ValueError("max\_features must be int or float")
37. if not (0 < max\_features <= self.n\_features\_in\_):
38. raise ValueError("max\_features must be in (0, n\_features]")
39. max\_features = max(1, int(max\_features))
40. *# Store validated integer feature sampling value*
41. self.\_max\_features = max\_features
42. *# Other checks*
43. if not self.bootstrap and self.oob\_score:
44. raise ValueError("Out of bag estimation only available if bootstrap=True")
45. if self.warm\_start and self.oob\_score:
46. raise ValueError("Out of bag estimate only available if warm\_start=False")
47. if hasattr(self, "oob\_score\_") and self.warm\_start:
48. del self.oob\_score\_
49. if not self.warm\_start or not hasattr(self, "estimators\_"):
50. *# Free allocated memory, if any*
51. self.estimators\_ = []
52. self.estimators\_features\_ = []
53. n\_more\_estimators = self.n\_estimators - len(self.estimators\_)
54. if n\_more\_estimators < 0:
55. raise ValueError(
56. "n\_estimators=%d must be larger or equal to "
57. "len(estimators\_)=%d when warm\_start==True"
58. % (self.n\_estimators, len(self.estimators\_))
59. )
60. elif n\_more\_estimators == 0:
61. warn(
62. "Warm-start fitting without increasing n\_estimators does not "
63. "fit new trees."
64. )
65. return self
66. *# Parallel loop*
67. n\_jobs, n\_estimators, starts = \_partition\_estimators(
68. n\_more\_estimators, self.n\_jobs
69. )
70. total\_n\_estimators = sum(n\_estimators)
71. *# Advance random state to state after training*
72. *# the first n\_estimators*
73. if self.warm\_start and len(self.estimators\_) > 0:
74. random\_state.randint(MAX\_INT, size=len(self.estimators\_))
75. seeds = random\_state.randint(MAX\_INT, size=n\_more\_estimators)
76. self.\_seeds = seeds
77. all\_results = Parallel(
78. n\_jobs=n\_jobs, verbose=self.verbose, \*\*self.\_parallel\_args()
79. )(
80. delayed(\_parallel\_build\_estimators)(
81. n\_estimators[i],
82. self,
83. X,
84. y,
85. sample\_weight,
86. seeds[starts[i] : starts[i + 1]],
87. total\_n\_estimators,
88. verbose=self.verbose,
89. check\_input=check\_input,
90. )
91. for i in range(n\_jobs)
92. )
93. *# Reduce*
94. self.estimators\_ += list(
95. itertools.chain.from\_iterable(t[0] for t in all\_results)
96. )
97. self.estimators\_features\_ += list(
98. itertools.chain.from\_iterable(t[1] for t in all\_results)
99. )
100. if self.oob\_score:
101. self.\_set\_oob\_score(X, y)
102. return self

可以看到其中调用了\_validate\_estimator方法。

平衡随机森林实现与EasyEnsemble类似，在继承RandomForestClassifier后将采样器属性替换成了随机下采样方法。



1. def \_validate\_estimator(self, default=DecisionTreeClassifier()):
2. """Check the estimator and the n\_estimator attribute, set the
3. `base\_estimator\_` attribute."""
4. if not isinstance(self.n\_estimators, (numbers.Integral, np.integer)):
5. raise ValueError(
6. f"n\_estimators must be an integer, " f"got {type(self.n\_estimators)}."
7. )
8. if self.n\_estimators <= 0:
9. raise ValueError(
10. f"n\_estimators must be greater than zero, " f"got {self.n\_estimators}."
11. )
12. if self.base\_estimator is not None:
13. self.base\_estimator\_ = clone(self.base\_estimator)
14. else:
15. self.base\_estimator\_ = clone(default)
16. self.base\_sampler\_ = RandomUnderSampler(
17. sampling\_strategy=self.\_sampling\_strategy,
18. replacement=self.replacement,
19. )

5.3.5应用过程

异常检测环节中，采用的数据集为进行了数据预处理异常值处理前所有环节，未进行特征选取，未进行噪音处理的原始数据集。

1.导入实现以上算法所需要的imblearn包与sklearn包。

1. import sklearn.ensemble as es
2. from sklearn.model\_selection import StratifiedShuffleSplit
3. from imblearn.ensemble import EasyEnsembleClassifier
4. from imblearn.ensemble import BalancedRandomForestClassifier
5. from imblearn.ensemble import RUSBoostClassifier
6. from imblearn.ensemble import BalancedBaggingClassifier
7. from sklearn.model\_selection import GridSearchCV

2.初始化模型，设置模型参数。统一设置random\_state控制模型随机划分方式，方便比较模型间的好坏。

1. rfc=es.RandomForestClassifier(n\_estimators=168,max\_depth=75,random\_state=1024)
2. eec=EasyEnsembleClassifier(base\_estimator=rfc,n\_estimators=20,random\_state=1024)
3. brfc=BalancedRandomForestClassifier(n\_estimators=168,random\_state=1024)
4. rusb=RUSBoostClassifier(base\_estimator=rfc,random\_state=1024)
5. bbc=BalancedBaggingClassifier(base\_estimator=rfc,random\_state=1024)

3.选取模型参数

使用GridSearchCV函数在一定范围内以一定步长选取模型最优的参数。这里以EsayEnsemble的基分类器个数为例。GridSearchCV的estimator参数为需要选取参数的模型，param\_grid为该模型需要选取的参数，scoring为评估标准。由于具体使用时为直接更改GridSearchCV参数，每评估一次就运行一次，因此源代码中仅保留了最后一次运行的代码。

1. param = {"n\_estimators": range(15,25)}
2. gscv=GridSearchCV(estimator=eec,param\_grid=param,scoring='roc\_auc')
3. gscv.fit(x,y)
4. print(gscv.best\_params\_,gscv.best\_score\_)

4.使用StratifiedShuffleSplit函数划分训练集与测试集进行5折抽样交叉验证。设置random\_state以控制划分方式。

1. for flod\_id,(trn\_idx, val\_idx) in enumerate(kfold.split(x, y)):
2. X\_train = x.iloc[trn\_idx]
3. Y\_train = y.iloc[trn\_idx]
4. X\_val = x.iloc[val\_idx]
5. Y\_val = y.iloc[val\_idx]

5.每次训练模型

1. eec\_model=eec.fit(X\_train,Y\_train)
2. brfc\_model=brfc.fit(X\_train,Y\_train)
3. rusb\_model=rusb.fit(X\_train,Y\_train)
4. bbc\_model=bbc.fit(X\_train,Y\_train)
5. rfc\_model = rfc.fit(X\_train, Y\_train)

6.以EsayEnsemble为例。对每个模型进行模型评估，使用predict\_proba预测为异常的概率用于计算auc值。使用predict预测异常值用于计算召回率等评估标准。

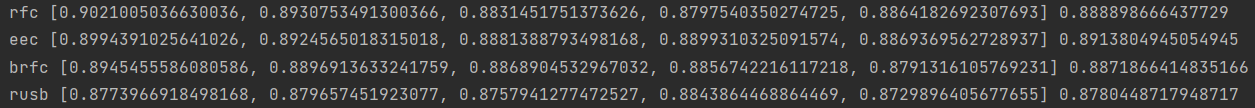
1. y\_val\_predict\_proba=eec\_model.predict\_proba(X\_val)
2. roc\_score=sklearn.metrics.roc\_auc\_score(Y\_val,y\_val\_predict\_proba[:,1])
3. roc\_auc.append(roc\_score.mean())
4. y\_val\_predict=eec\_model.predict(X\_val)
5. print('eec/n',sklearn.metrics.classification\_report(y\_true=Y\_val,y\_pred=y\_val\_predict))
6. print(roc\_score)

4到6步完整代码如下：

1. roc\_auc = []
2. accuracy=[]
3. roc\_auc\_brfc=[]
4. roc\_auc\_rusb=[]
5. roc\_auc\_bbc=[]
6. roc\_auc\_rfc=[]
7. ks=[]
8. kfold = StratifiedShuffleSplit(n\_splits=5,random\_state=1901)
9. for flod\_id,(trn\_idx, val\_idx) in enumerate(kfold.split(x, y)):
10. X\_train = x.iloc[trn\_idx]
11. Y\_train = y.iloc[trn\_idx]
12. X\_val = x.iloc[val\_idx]
13. Y\_val = y.iloc[val\_idx]
14. eec\_model=eec.fit(X\_train,Y\_train)
15. brfc\_model=brfc.fit(X\_train,Y\_train)
16. rusb\_model=rusb.fit(X\_train,Y\_train)
17. bbc\_model=bbc.fit(X\_train,Y\_train)
18. rfc\_model = rfc.fit(X\_train, Y\_train)
20. y\_val\_predict\_proba = rfc\_model.predict\_proba(X\_val)
21. roc\_score = sklearn.metrics.roc\_auc\_score(Y\_val, y\_val\_predict\_proba[:, 1])
22. roc\_auc\_rfc.append(roc\_score.mean())
23. y\_val\_predict = rfc\_model.predict(X\_val)
24. print('rfc/n', sklearn.metrics.classification\_report(y\_true=Y\_val, y\_pred=y\_val\_predict))
25. print(roc\_score)
27. y\_val\_predict\_proba=eec\_model.predict\_proba(X\_val)
28. roc\_score=sklearn.metrics.roc\_auc\_score(Y\_val,y\_val\_predict\_proba[:,1])
29. roc\_auc.append(roc\_score.mean())
30. y\_val\_predict=eec\_model.predict(X\_val)
31. print('eec/n',sklearn.metrics.classification\_report(y\_true=Y\_val,y\_pred=y\_val\_predict))
32. print(roc\_score)
33. y\_val\_predict\_proba = brfc\_model.predict\_proba(X\_val)
34. roc\_score = sklearn.metrics.roc\_auc\_score(Y\_val, y\_val\_predict\_proba[:, 1])
35. roc\_auc\_brfc.append(roc\_score.mean())
36. y\_val\_predict = brfc\_model.predict(X\_val)
37. print('brfc/n', sklearn.metrics.classification\_report(y\_true=Y\_val, y\_pred=y\_val\_predict))
38. print(roc\_score)
39. y\_val\_predict\_proba = rusb\_model.predict\_proba(X\_val)
40. roc\_score = sklearn.metrics.roc\_auc\_score(Y\_val, y\_val\_predict\_proba[:, 1])
41. roc\_auc\_rusb.append(roc\_score.mean())
42. y\_val\_predict = rusb\_model.predict(X\_val)
43. print('rusb/n', sklearn.metrics.classification\_report(y\_true=Y\_val, y\_pred=y\_val\_predict))
44. print(roc\_score)
45. y\_val\_predict\_proba = bbc\_model.predict\_proba(X\_val)
46. roc\_score = sklearn.metrics.roc\_auc\_score(Y\_val, y\_val\_predict\_proba[:, 1])
47. roc\_auc\_bbc.append(roc\_score.mean())
48. y\_val\_predict = bbc\_model.predict(X\_val)
49. print('bbc/n', sklearn.metrics.classification\_report(y\_true=Y\_val, y\_pred=y\_val\_predict))
50. print(roc\_score)
51. del eec\_model,test\_y, X\_train, Y\_train, X\_val, Y\_val,brfc\_model,rusb\_model,rfc\_model,bbc\_model
52. gc.collect()
53. print('rfc',roc\_auc,np.mean(roc\_auc))
54. print('eec',roc\_auc,np.mean(roc\_auc))
55. print('brfc',roc\_auc,np.mean(roc\_auc))
56. print('bbc',roc\_auc,np.mean(roc\_auc))
57. print('rusb',roc\_auc,np.mean(roc\_auc))

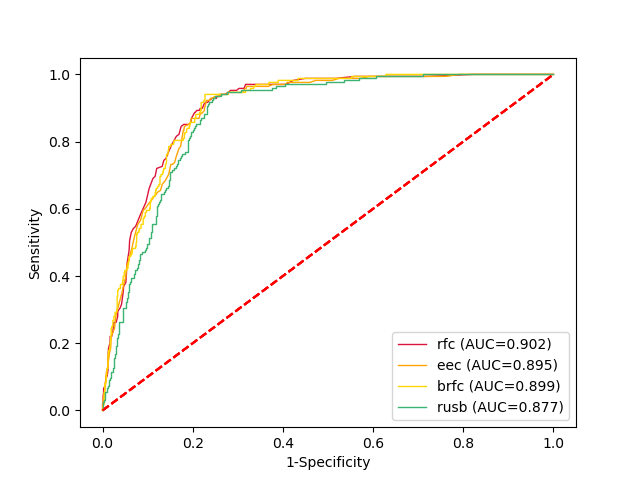
5.3.6结果分析与对比

以下是4种模型的auc值比较，其中rfc是随机森林的结果，brfc是平衡随机森林的结果，rusb是RUSBoost的结果，eec是EasyEnsemble的结果。列表中数据为5折抽样交叉验证每次的AUC值，最右侧为抽样交叉验证结果的均值。



显然，仅就AUC值这一评估标准而言，EasyEnsemble模型效果最好，随机森林次之，平衡随机森林再次，RUSBoost结果最差。而细看每一次交叉验证的结果，随机森林对部分数据集效果超过EasyEnsemble而部分结果较差。

绘制出ROC曲线，显然除了RUSBoost性能较弱外其余模型性能不相上下。



以下4种模型的召回率、精确率、准确率指标，其中的值均为5折抽样交叉验证的均值。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 精确率(0) | 召回率(0) | 精确率(1) | 召回率(1) | 准确率 |
| 随机森林 | 0.882 | 0.954 | 0.618 | 0.358 | 0.854 |
| EasyEnsemble | 0.976 | 0.756 | 0.432 | 0.918 | 0.782 |
| 平衡随机森林 | 0.974 | 0.756 | 0.428 | 0.912 | 0.786 |
| RUSBoost | 0.898 | 0.904 | 0.508 | 0.488 | 0.836 |

由上述结果可以看出，使用抽样交叉验证后，随机森林虽然预测值为1的召回率比随机划分数据集的召回率好些，但较前述经过STOME处理过的数据集仍然差很多。说明随机森林严重偏向于正常样本，分辨出异常值的能力较差。采用欠采样集成方法改进模型后，虽然AUC值变化幅度不大，分类准确率下降，分类为0的召回率下降，模型将正常样本判断为异常值的可能提高，但分类为1的召回率大幅度提高。考虑到任务为异常检测，要求尽可能的检测出异常值，因此分类为1的召回率权重应该高于精确率，这样的结果可以接受。

比较这三种欠采样集成模型，可以发现RUSBoost在提高召回率能力上不如其它两种，而其余两种模型除AUC值外能力相近。因此在展示时选用了EasyEnsemble进行展示。推测RUSBoost平衡数据能力不如其余两种的原因为RUSBoost采用AdaBoost算法进行集成，原始的AdaBoost算法构建基分类器始终仅采用一套数据集，而从RUSBoost具体实现中我们可以看到RUSBoost每构建一棵决策树就要对样本进行随机欠采样，这可能影响了AdaBoost算法样本权重的计算进而影响到了基分类器的权重。推测本次案例EasyEnsemble结果略好于平衡随机森林主要因为应用EasyEnsemble实际上对随机森林再进行了一次集成，而平衡随机森林仅在决策树上集成，整个模型随机性强于EasyEnsemble。

3.3.7未来挑战

数据不平衡对少数类别的召回率影响较大，调整模型权重与分层抽样对提升少数类别召回率影响较小。可以通过过采样方式与欠采样方式进行数据平衡。但是，以上方式尽管能够提高少数类别的召回率，但明显降低多数类别的召回率与整个模型的准确率，且对提升auc值没有太大影响。当任务为异常检测时，这可以被接受。但当其它任务时，可能这会成为一种缺陷。