数据科学大作业

小组人数：3人

分工：

201250047_殷天逸_2260122782@qq.com：任务1、任务5、提出隐私信息分类算法

201250048_沈霁昀_1377060711@qq.com：nlp部分、隐私信息分类的代码实现、提出任务2的算法

201250043_董志昂_898620751@qq.com：爬取隐私政策、构建隐私信息文本库、任务2的代码实现

目录

三、项目成果展示

1.隐私信息部分可视化

2.隐私信息定位

3.个人信息自动分类

4.信息操作判断定位

4.1 数据库操作检索

4.2 其他操作检索

5.信息加密检索

四、项目代码开源地址

一、摘要

“隐私信息扫描”这一课题，是为了解决大数据时代的信息安全问题。程序是否遵守了隐私政策，又对隐私信息做出了什么操作，操作过程中是否对某类关键信息进行加密，都是我们所关切的问题。

为了解决这些问题，我们提出了一套基于自然语言处理技术的解决方案。首先，我们通过爬虫技术生成自己的隐私信息文本库，这对后面的工作都有着很大的帮助。基于此，我们对程序代码做了 AST 处理，将变量名与文本库匹配，我们实现了程序中隐私信息的定位工作。进一步的，基于 AST，通过找到对程序中 SQL 等与数据库进行交互的语句，分析程序中函数的调用关系，我们试图定位并判断出程序对信息的操作类型，以及数据在操作过程中是否经过加密。

在生成隐私信息文本库的过程中，我们同时对隐私信息进行了分类，将设备浏览器信息同个人信息、财产信息等进行了分类，于是我们通过机器学习训练出分类器模型，对信息进行自动分类。

二、项目实现过程

1. 隐私信息文本库

我们写了一个爬虫程序，爬取了 100 余篇隐私政策，对不同类型的项目均有涉及，使得文本库尽可能完备。由于文本库的精度对后面的工作影响很大，我们首先手工处理了 10 份隐私政策，尽可能精准地提取其中的隐私信息。之后，经过自然语言处理，我们通过模式匹配文本对隐私信息的常见描述，如前面含有“your”“customer”以及“personal”等的名词块，添加文本库中没有的隐私信息表述。

2. 隐私信息定位

为了找到代码中的个人隐私信息，我们需要借助前文所提到的隐私信息文本库。注意到在程序之中，信息往往是储存在变量之中，大部分情况下，变量名并非没有意义，大部分的变量名都和其中存储的内容相关。在此基础之上，我们希望扫描整个项目，找到代码中可能用来存储个人隐私信息的变量。

尽管如此，对于代码中的每一个字符串，使用传统的字符串匹配难以判断其是一个变量，还是一些没有意义的语句(如 if、else...)。这需要我们更深入地去了解代码的结构。而 AST 能很好地帮我们解决这个问题。

AST(Abstract Syntax Tree)全称是抽象语法树，它以树状的结构的形式表现编程语言的语法结构，书上的每个节点都表示源码的一种结构。

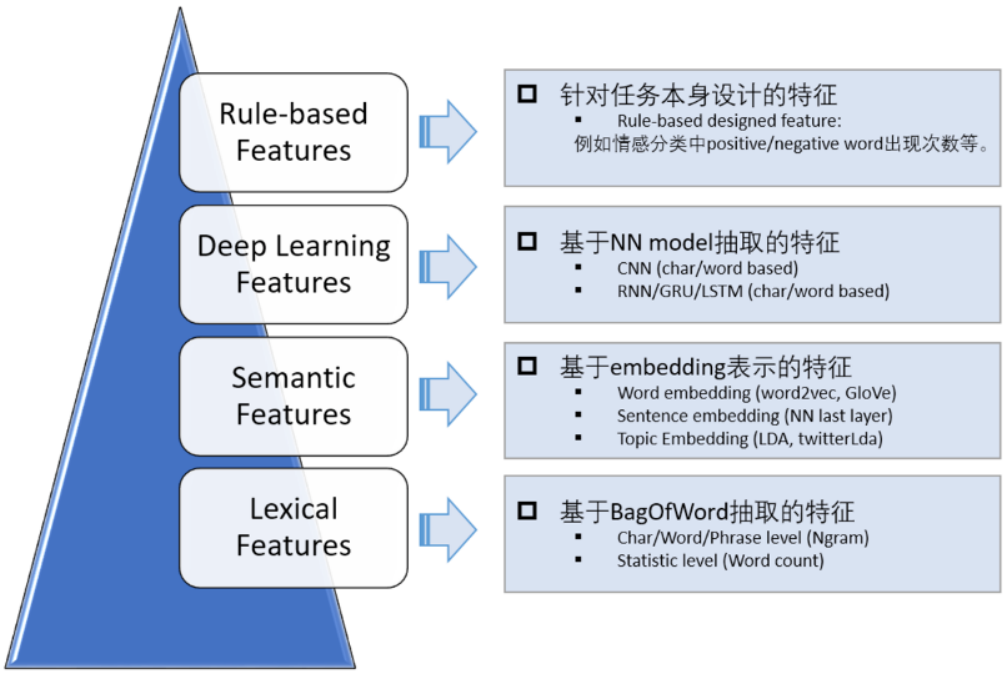
部分结构如图：

```
        type_comment=None,
    ),
    Assign(
        targets=[Name(id='country_id', ctx=Store())],
        value=Call(
            func=Attribute(
                value=Name(id='fields', ctx=Load()),
                attr='Many2one',
                ctx=Load(),
            ),
            args=[],
            keywords=[
                keyword(
                    arg='string',
                    value=Constant(value='Country', kind=None),
                ),
                keyword(
                    arg='comodel_name',
                    value=Constant(value='res.country', kind=None),
                ),
                keyword(
                    arg='help',
                    value=Constant(value='Country for which this tag is available, when applied on taxes.', kind=None),
                ),
            ],
        ),
        type_comment=None,
    ),
    FunctionDef(
        name='_get_tax_tags',
```

3. 个人信息自动分类

为了对数据进行更加深入的分析，我们对隐私信息进行了分类，将敏感个人信息比如财产信息、医疗健康信息等与普通的信息分开，有利于进一步研究程序对数据的处理。

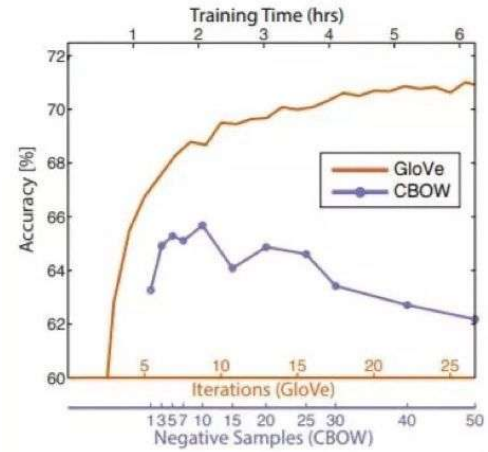
文本分类的核心都是如何从文本中抽取出能够体现文本特点的关键特征，抓取特征到类别之间的映射。 所以特征工程很重要，可以由四部分组成：



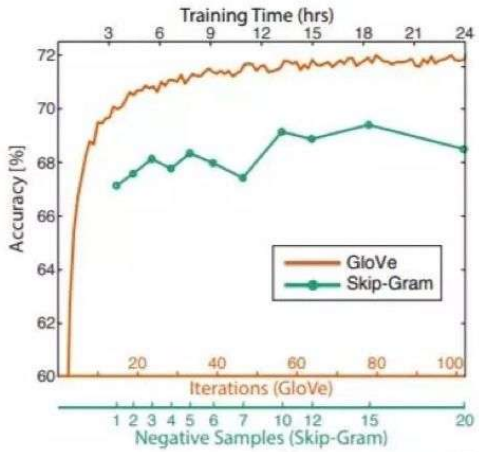
传统的词袋模型例如 TF-IDF 等，虽然比较简单直观，但是仅将词语符号化，没有考虑词之间的语义联系。比如，“麦克风”和“话筒”是不同的词，但是语义是相同的。在众多隐私政策中，不免会出现语义相同但是不是同一个词的情况。使用词袋模型，我们认为精确度可能不太乐观。

因此我们考虑采用词嵌入模型。最著名的词嵌入模型是 Google 的 Word2Vec(2013)，其他的还有斯坦福大学的 GloVe(2014)和 Facebook 的 FastText(2016)。

Word2Vec 是基于局部语料库训练的，最大的缺点是没有充分利用所有的语料。相比之下，GloVe 则借助了全局词频统计，在计算语义相似度上的表现更加优秀，我们决定采用 GloVe（Global Vectors for Word Representation）计算词语相似度来实现我们的分类工作。



GloVe vs CBOW



GloVe vs Skip-Gram

要理解 GloVe，首先要理解词向量。词语，是人类的抽象总结，是符号形式的（比如中文、英文、拉丁文等等），为了让机器理解，需要把他们转换成数值形式，或者说——嵌入到一个数学空间里，这种嵌入方式，就叫词嵌入，词向量就是词语与数值转化的一种方式。

GloVe 词向量模型基本步骤如下：

- 基于词共现矩阵收集词共现信息。假设 X_{ij} 表示词汇 i 出现在词汇 j 上下文的概率。首先对语料库进行扫描，对于每个词汇，我们定义一个 window_size，即每个单词向两边能够联系到的距离，在一句话中如果一个词距离中心词越远，我们给予这个词的权重越低。
- 对于每一组词对，都有：

$$w_i^T w_j + b_i + bj = \log X_{ij}$$

两个词相似，即在类似的语义环境下，两个词出现表征的含义是类似的，从数学的角度，即计算两个词向量的余弦值，夹角越小，两个词就越相似。

举例：

- 对于一句话，“她们 夸 吴彦祖 帅 到 没朋友”，如果输入 x 是“吴彦祖”，那么 y 可以是“她们”、“夸”、“帅”、“没朋友”这些词

- 现有另一句话：“她们 夸 助教 帅 到 没朋友”，如果输入 x 是“助教”，那么不难发现，这里的上下文 y 跟上面一句话一样
- 从而 f(吴彦祖) = f(助教) = y，所以大数据告诉我们：助教 ≈ 吴彦祖

我们使用了 spacy 预训练的 GloVe 模型，将对于一个文本，求与每一类词的相似度。这里一开始，我们参考了 Privacy Information Classification: A Hybrid Approach ，文中对于分类使用的是求与每组相似度的算术平均值，但在实践后，我们发现效果并不是很好，于是我们调整了策略，采用求相似度最大值的方法进行分类，结果证明分类效果较为显著。

4. 信息操作判断定位

接下来，我们开始尝试对程序代码进行静态分析，寻找其中对隐私信息的操作。我们首先考虑将一些典型处理模块（比如：分享，上云，加密，存进数据库，可视化等），通过 AST 处理，与项目代码的 AST 进行匹配。但我们很快意识到这很难行得通，处于不同的目的，对于不同的信息，代码无疑是多样的，很难像这样去用一些模板去匹配。

于是我们研究了一些项目代码，发现对信息数据的操作与数据库交互操作的关系是紧密的。从数据流的角度来看，程序可能从获取数据到处理加工数据，使用公开数据，再到存储数据。也就是说，数据库的交互操作，往往接近处理数据的起点或终点。

事实上，比如要将信息存进数据库之前，通常会先进行比如分割、加密，这些操作通常是放在同一个函数中，作为程序的最细粒度。这样处理立即存储，保证信息的安全与操作流程的整体性，操作不会被分割导致信息泄露或遗失。

意识到数据库操作的重要性之后，我们考虑首先从程序与数据库的交互入手，检索对数据库操作的语句，判断程序对数据库的增删查改操作，便可识别定位程序对信息的操

作。而在读取数据库之后或是存进数据库之前的操作，则是对数据的使用、加工或公开等，是下一步研究的工作。

4.1 数据库操作检索

在实际操作的过程中，我们发现一个大的项目，往往会有一些基础设施的建设，将 SQL 操作进行封装，就是其中很常见的一种，以我们研究的 Odoo 项目为例，将数据库操作封装为了 `Environment` 类，含有 `create, delete, execute, unlink` 等一系列封装了 SQL 语句的与数据库交互的函数。

因此，经过一系列研究，我们认为对于数据库操作的检索，应该既要考虑直接寻找 SQL 语句，也要观察项目本身是否有相关的基础设施函数。而具体选用哪一种方法是需要视具体情况而定的。对于一个开源的、整体封装较好的大型项目，我们只需要关注项目本身用于封装数据库的基础设施函数，通过检索整个项目里基础设施函数的调用即可分析得到项目对数据库操作的检索。

以 odoo 项目为例，odoo 项目将对数据库的操作统一封装成了(.*).self.env[(.*)].(.*)形式，这表明了 odoo 项目中所有与数据库的交互操作都是通过这一函数形式实现的,那我们只需要在项目源码中检索类似所有(.*).self.env[(.*)].(.*)形式的函数，并对其进行相应的处理，便可得出 odoo 项目中所有与数据库的交互操作。

在最初，我们考虑的是通过字符串匹配的形式去实现定位封装函数的功能。但是由于字符串匹配算法过于机械，尽管算法运行起来很快，但是输出时只能定位到包含封装函数的某一行或某几行，不能完全精准的定位封装函数本身，这对后续的处理工作造成了较大的影响。

因此，斟酌再三之后，我们选择了基于 AST 的子树匹配算法，通过对程序源码的 AST 进行遍历，搜寻它的子树，从而精确的匹配出了 odoo 项目与数据库进行交互的操作。搜寻到子树后，我们便能针对子树输出数据库交互操作函数的操作名与被操作的变量。

4.2其他操作检索

完成了对与数据库的交互函数的检索，接下来我们试图定位到项目对隐私信息变量的其他操作。这些操作的实现手段往往因为其针对的数据形式各异而大相径庭，因此，我们提出的方案是检索函数名与相关注释。

在 4.1 中，对于存在数据库交互操作的函数，我们已经对它的 AST 进行了检索，找到了所有数据库操作的调用方法。而对于数据的处理不仅仅只包含与数据库进行交互这一项处理，还包含对信息的收集、存储、使用、加工、传输、提供、公开、删除等基本操作。由于函数名通常都有其实际含义，我们试图从函数名和注释入手，判断它的处理类型。

比如对数据进行各类加工的函数，常常被命名为"AtoB""A2B"等，删除操作则含有"discard""remove"等关键词。以这种方式，我们便可以从复杂的项目代码中提取出其他操作过程。

在阅读了部分项目源码后，参考 GDPR 的相关法律规定，关于项目对隐私信息变量的其他操作，我们抽象出了共五种类型的操作，我们将其分别命名为 acquire, delete, process, public ,save 操作。其中各类操作中包含了 4-9 个不等的关键字，通过对每类操作的关键字在代码中进行检索，便可以获得代码中实现的对隐私信息变量的其他操作。

在实际操作的时候我们注意到，对隐私信息变量的其他操作**往往**与数据库检索操作有着极高的相关性，这点不难理解：项目代码所做的信息工作基本上遵循着“检索数据——提取数据——处理数据——返回数据”这一流程，检索、提取、返回这三项工作基本上是由数据库交互操作来完成的，而其中的处理数据工作则大多由4.2所检索到的对隐私信息变量的其他操作完成的。

因此，在实际实现中，我们基于 4.1 中已匹配到的相关数据库操作的 AST 子树，向上层回溯一定的层级，再从这个层级向下检索对隐私信息变量的其他操作，便有较大希望检索到对隐私信息变量进行处理的相关操作。

5. 信息加密检索

对于一些重要的信息等，我们希望程序能够将密码加密在进行操作，而非明文操作或存储。于是我们希望研究程序是否有对一些关键信息进行过加密处理。假如使用 python 自带的加密方法，需要引入相应的模块，例如 hashlib，hmac，base64 等。而这些加密模块的加密函数都是固定的。因此，我们使用了字符串匹配，在前文找到的信息处理操作中找到加密操作。

三、项目成果展示

1.隐私信息部分可视化

这是隐私信息文本库中的部分内容可视化之后的结果。之后隐私信息文本库将作为词典贯穿整个项目。

2.隐私信息定位

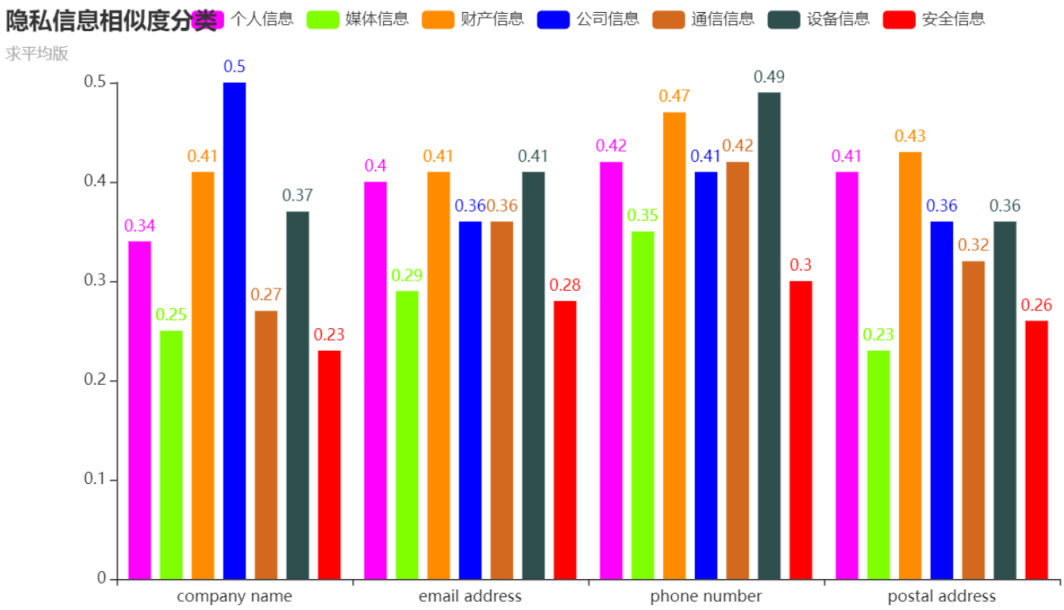
我们扫描了整个项目，对每个.py 文件，将其解析成抽象语法树，然后以 BFS 遍历整棵树，找到类型为 ast.Name 类型的结点 node。该节点则代表着一个变量，而 node.id 则是变量名。我们借助 nlp 将变量与词典中的所有词比较，若相似度大于某个值，则判断其是存有隐私信息的变量。

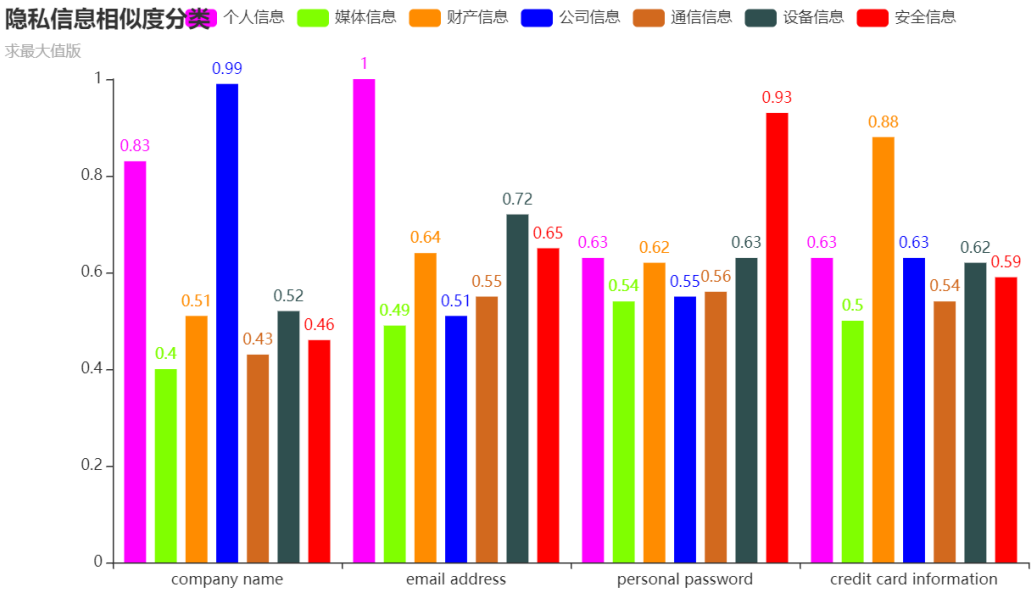
```
def check(varName):
    global doc1
    global res
    varname = " ".join(re.split("[^A-Za-z0-9]", varName))
    docVar = nlp(varname)
    for doc in doc1:
        if (doc.similarity(docVar) >= min_similarity):
            res.write(varname.strip() + os.linesep)
            break


def search(filepath):
    file = open(filepath, encoding='utf-8')
    code = file.read()
    file.close()
    ast_root = ast.parse(code, mode="exec")
    for node in ast.walk(ast_root):
        if (isinstance(node, ast.Name)):
            if (node.id in hashset): continue
            hashset.append(node.id)
            check(node.id)
```

人工统计在阈值(min_similarity)为 0.82 时准确度约为 70%。

3. 个人信息自动分类

前文提到，我们在计算相似度时，考虑了求平均值和最大值两种方法。两种方法效果对比统计图：





图表解释：我们采取了欧盟一般数据保护条例（GDPR）的分类标准，目前将信息大致分成了 7 类

1. 个人一般信息（地址、电话、邮箱等）
2. 媒体信息（照片、视频、录音等）
3. 财产信息（消费账单、银行账户、信用卡记录等）
4. 公司信息（erp 项目多见，公司名、职位、简历、面试等）
5. 通讯信息（通话记录，聊天记录等）
6. 设备信息（IP，browser，设备型号等）
7. 密码安全信息

对于一个隐私信息，我们通过计算它与各组的相似度来进行分类，如果与其中一组的相似度比较高，即可判定它属于这一类别。那么不难发现图二中相似度区分度比较大，结果来看分类也比较精准。

4. 信息操作判断定位

4.1 数据库操作检索

文件名：ast_self_env_process.py

我们扫描了整个项目，对每个.py 文件，将其解析成抽象语法树，对于该抽象语法树，以 BFS 扫描抽象语法树的各个节点，如果一个节点的构型符合如下代码中的要求构型，也即这个节点的源码函数符合(.*).self.env[(.*)].(.*)构型，那我们便可以返回该函数的操作名与变量名。

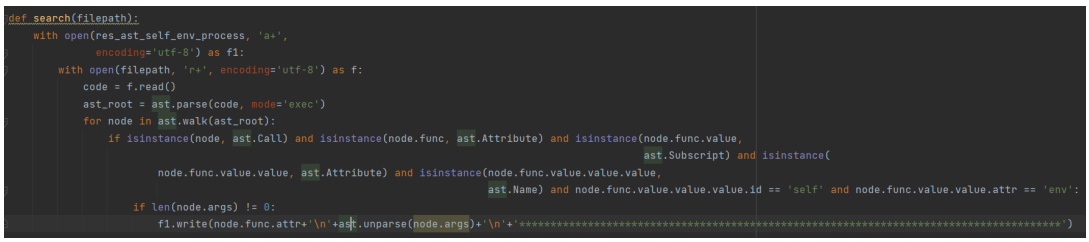


图 ast_self_env_process.py

4.2 其他操作检索

文件名：

get_other_functions.py

sort_other_functions.py

基于上面实现的对数据库操作检索代码的实现，我们所要完成的任务便是对数据库操作检索代码的相关节点进行追溯，在具体的项目实现中，我们选取的追溯范围是追溯到数据库操作检索节点的上一个 DefFunction 节点，也即源码中数据库操作检索函数的上一个 def 定义函数，再从这个节点向下匹配对隐私信息的其他操作，从而获得了相关的其他操作

图 get_other_functions.py

在得到了其他操作之后，由于代码的空间局部性原理，代码中对隐私信息变量的其他操作往往有着极高的重复性，这给分析其他操作带来了干扰，因此，我们在 sort_other_functions.py 文件里实现了对得到结果的去重与排序，以下是得到的最终结果的部分截图：

acquire 类的其他操作：

```
Department.create({'name': 'Research and devlopment'})

DocumentModel._get_mail_message_access(doc_ids, operation)

ERRORS.get(result[filename]['error'], {'error': result[filename]['error'], 'blocking_level': 'error'})

EU_TAX_MAP.get((company.account_fiscal_country_id.code, domestic_tax.amount, country.code), False)

Employee.search([('department_id', '=', department.id)])

EnglishUoMCateg.search([('name', '=', 'Unsorted/Imported Units')])

Equipment.with_user(equipment_manager).with_context(allowed_company_ids=cids).search_count([])

Equipment.with_user(user).search_count([])

Fields.search([('model', '=', 'res.country'), ('name', '=', 'code')])

Fields.search([('model', '=', 'res.country'), ('name', '=', 'name')])

Fields.search([('model', '=', 'res.partner'), ('name', '=', 'category_id')])

Fields.search([('model', '=', 'res.partner'), ('name', '=', 'child_ids')])

Fields.search([('model', '=', 'res.partner'), ('name', '=', 'city')])

Fields.search([('model', '=', 'res.partner'), ('name', '=', 'country_id')])

Fields.search([('model', '=', 'res.partner'), ('name', '=', 'name')])
```

delete 类的其他操作：

```
clean(result.get(id, default))

clean(self._get(name, model))

clean(values[id])

clean_context(self._context)

clean_context(self.env.context)

cleaned_vals.update({'amount_currency': vals.get('debit', 0.0) - vals.get('credit', 0.0)})

credit_aml.remove_move_reconcile()

custom_view.unlink()

data.get('to_delete')

delete(records[:half_size])

delete(records[half_size:])

delete(self.env['ir.model'].browse(unique(model_ids)))

delete(self.env['ir.model.constraint'].browse(unique(constraint_ids)))

delete(self.env['ir.model.fields'].browse(unique(field_ids)))

delete(self.env['ir.model.fields.selection'].browse(unique(selection_ids)).exists())
```

process 类型的其他操作：

```
self._check_correct_order(self.p2 + self.p1 + self.p3 + self.p4)

self._check_correct_order(self.p2 + self.p3 + self.p4 + self.p1)

self._check_currencies()

self._check_delay(action, record, record_dt)

self._check_double_validation_rules(employee_id, values.get('state', False))

self._check_double_validation_rules(employees, values['state'])

self._check_end_of_rating_tours()

self._check_holidays_status(holiday_status, 2.0, 0.0, 2.0, 0.0)

self._check_holidays_status(holiday_status, 2.0, 0.0, 2.0, 2.0)

self._check_holidays_status(holiday_status, 2.0, 2.0, 0.0, 0.0)

self._check_if_no_draft_orders()

self._check_multiwarehouse_group()

self._check_order_search(self.sale_order, [('invoice_ids', '=', False)], self.env['sale.order'])

self._check_order_search(self.sale_order, [('invoice_ids', '=', False)], self.sale_order)

self._check_payment_method_ids()
```

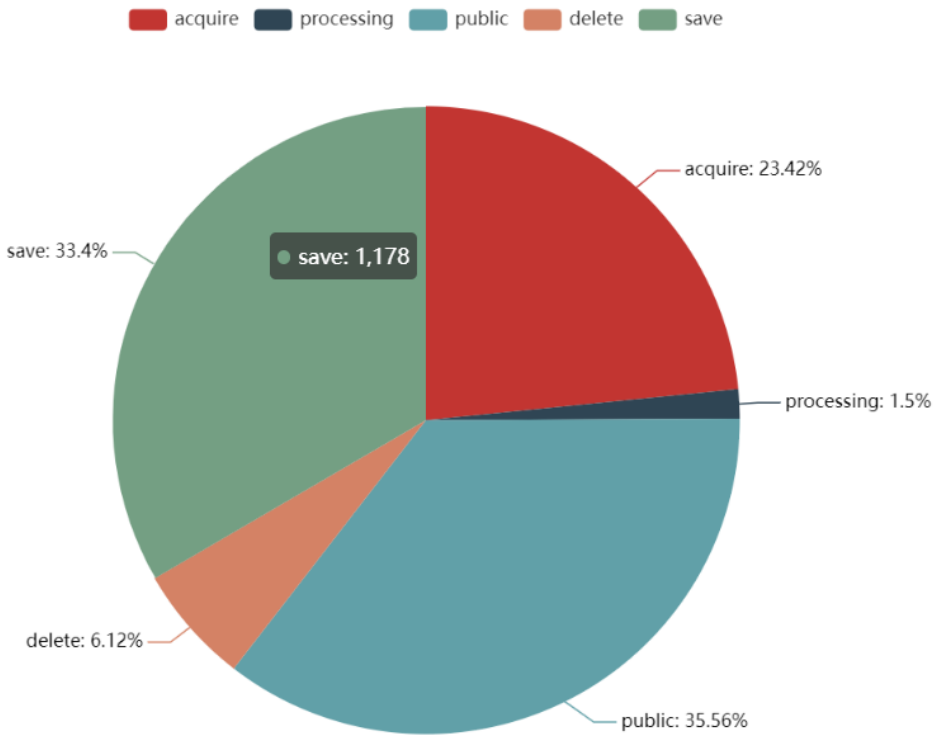
public 类型的其他操作：

```
payment_a.action_post()

payment_b.action_post()

payment_c.action_post()

payment_move._post()

payments.action_post()

payments.filtered(lambda r: r.state == 'draft').action_post()

payments.filtered(lambda r: r.state == 'posted' and (not r.is_move_sent))

payments.filtered(lambda r: r.state == 'posted' and (not r.is_move_sent)).write({'is_move_sent': True})

pkgutil.get_data(self.__module__, 'contacts.json')

pkgutil.get_data(self.__module__, 'contacts.json').decode('utf-8')

pkgutil.get_data(self.__module__, 'contacts_big.json')

pkgutil.get_data(self.__module__, 'contacts_big.json').decode('utf-8')

plaintext2html(alarm.body)

plaintext2html(html2plaintext(body))

plus_child_tags.write({'name': '+' + tag_name_postponed})
```

save 类型的其他操作：

```
BomLine.create({'product_id': self.kit_3.id, 'product_qty': 2.0, 'bom_id': bom_kit_parent.id})
Category.with_user(equipment_manager).create({'name': 'Phones - Test', 'company_id': company_a.id, 'technician_user_id': equipment
Category.with_user(equipment_manager).with_context(allowed_company_ids=cids).create({'name': 'Computers - Test', 'company_id': com
Category.with_user(equipment_manager).with_context(allowed_company_ids=cids).create({'name': 'Monitors - Test', 'company_id': comp
Category.with_user(user).create({'name': 'Computers', 'company_id': company_b.id, 'technician_user_id': user.id})
Category.with_user(user).create({'name': 'Software', 'company_id': company_b.id, 'technician_user_id': user.id})
Command.create(dest_vals)
Command.create(new_move[0])
Command.create(source_vals)
Command.create(vals)
Command.create(values)
Command.create({'acc_number': 'FO042'})
Command.create({'author': user_demo.id, 'body': 'two'})
Command.create({'author': user_root.id, 'body': 'one'})
Command.create({'author': user_root.id, 'body': 'three'})
```

对以上五种类型的其他操作进行统计分析，可以得到一个简略的饼状图来表示各类操作在总其他操作中所占据的份额，图表如下：



5. 信息加密检索

```
def check(line):
    if ("hashlib" in line) or ("hmac" in line) or ("b64encode" in line):
        return True
    return False
```

以上我们对任务 2 的结果进行字符串匹配，找到每类操作中的加密操作。结果如下：

然而这种方法只能找到有限的加密操作，一些没有加密得操作就难以定位。对于像 odoo 这样复杂的架构难以对数据进行溯源，但是我们认为，利用数据流分析可以实现信息安全检测。

四、项目代码开源地址

Stanton-Morgan/Data-Sci-Project (github.com)