PrivacyDetect

目录

项目信息

✦小组成员：

姓名	学号	分工
金冯阳	201250069	代码分析 代码整合 项目推进
韦姚丞奕	201840187	隐私政策库自动分类 自然语言摘要
华广松	201840309	隐私政策库匹配判定 自然语言分词

摘要

通过文本扫描等手段获取代码中获取用户个人信息的部分。一方面，通过对Android应用的安装包（.apk）文件进行反编译，获取smali文件，进行安卓API扫描获取隐私行为，同时获取到根目录中的AndroidManifest.xml文件，通过扫描权限，获取应用的申请权限集，将其与通过安卓API扫描获取到的隐私行为一起进行应用的隐私行为分析找到代码中对个人信息的处理操作。另一方面，对安卓应用的隐私政策进行关键词提取，并通过调用隐私政策词典提取隐私政策中声明的隐私行为，并对这些隐私行为进行自动分类。比较隐私政策中提取的隐私行为与通过反编译等手段获取的应用隐私行为，对比代码行为是否符合隐私政策。

一. 绪论

1.1 研究背景

20世纪90年代以后，随着计算机技术关键技术的不断迭代突破，机器计算能力飞速提升，大数据时代来临，大量延伸程序诞生，并由此产生了大量的信息，包括社会信息，集体信息以及个人信息等。人们逐渐意识到信息能产生巨大的价值，大大小小的互联网企业往往是信息的直接获取者和受益者，在推动社会进步的同时带来了信息滥用的问题。2021年8月20日，十三届全国人大常委会第三十次会议表决通过《中华人民共和国个人信息保护法》。个人信息保护法自2021年11月1日起施行。其中明确：①通过自动化决策方式向个人进行信息推送、商业营销，应提供不针对其个人特征的选项或提供便捷的拒绝方式②处理生物识别、医疗健康、金融账户、行踪轨迹等敏感个人信息，应取得个人的单独同意③对违法处理个人信息的应用程序，责令暂停或者终止提供服务。可见，分析软件开发商是否非法获取个人信息以及对获取的敏感信息是否提供了安全保障是一个可研究讨论的问题。

1.2 研究意义

1.产出了一个安卓软件隐私扫描工具

2.帮助解决目前隐私泄露泛滥的情况

3.提升用户对软件的信任度与安全感

1.3 研究目标

目标1：找到代码中的个人信息

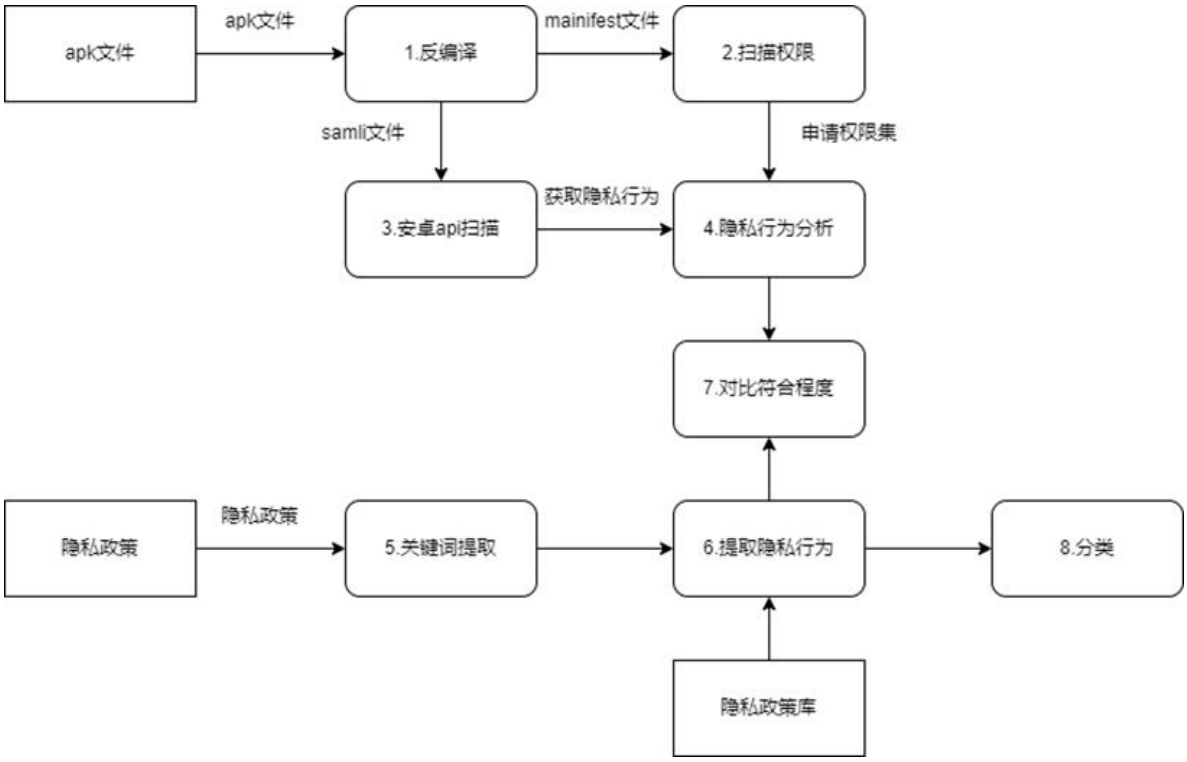
目标2：找到代码中对个人信息的处理操作

目标3：对比代码行为是否符合隐私政策

目标4：对获取隐私政策中的个人信息进行自动分类

目标5：判断用户数据是否经过安全处理

1.4 研究思路



二. 数据获取

2.1 数据概况

本次研究数据主要来源于官方公布的隐私政策，用户输入软件的apk文件和对应公布的隐私政策，输出扫描后的分析结果。

2.2 数据源

腾讯隐私政策：https://privacy.qq.com/policy/apps-privacypolicy

华为隐私政策：https://www.huawei.com/cn/privacy-policy

苹果隐私政策：https://www.apple.com.cn/legal/privacy/szh/?cid

阿里隐私政策：https://privacy.alibabagroup.com/#/home

.....

三. 研究方法

3.1 安卓代码分析方法

3.1.1 总体思路

首先通过反编译工具对apk进行反编译。随后对提取出的AndroidManifest.xml进行分析提取出activity和permission。再对代码分析提取隐私内容和行为，与前者合并后再通过permission进行过滤，得到最终在代码中进行的隐私行为。

```
Set<Activity> activities=scanActivity();
Set<Permission> permissions=scanPermission();
activities.addAll(scanFunction());
Set<Activity> newActivities=filterActivity(activities,permissions);
```

3.1.2 Apk反编译

对apk进行反编译已经是网络上十分常见的操作。github上有著名的反编译器Apktool，它可以实现对包含运行代码的classes.dex的解包，将其分解为smali文件的集合，而每个smali文件里都包含了一个类的字节码，且文件都是按照包名进行组织。同时Apktool还会对AndroidManifest.xml文件进行解码，使其可以被按照正常方式阅读，也能被输入流读取。在反编译完成后，我们便能得到apk对应的Source，其中包含AndroidManifest.xml以及所有的smali文件的引用。

由于Apk反编译常常被人滥用，所以一些公司（如腾讯、360）开发出了Apk加密（壳）套件。其原理是将业务代码写入库文件，classes.dex只包含外壳代码。运行时外壳代码将库文件中的业务代码动态解密出来交给dalvik虚拟机使用。Apktool对库文件都是采用直接忽视的办法，因而只会解包出外壳代码。目前网络上对于脱壳处理都是通过安卓设备在运行时将外壳代码解密出的业务代码dump出来，都是基于运行时的操作。解密的动态性使得静态分析无法实现脱壳。但即使拿不到业务代码，通过对AndroidManifest的分析也能获取一些行为信息。

在程序的开始阶段，我们会调用Windows的cmd对提供的apk进行反编译。由于一些代码量较大的软件反编译时间较长，且为了对多个apk反编译后的内容进行组织，我们使用了一个cache来缓存每一个Source。如果开始时发现cache中已有对应文件则直接完成反编译部分。

3.1.3 AndroidManifest分析

Manifest直译为清单，其列举了一个应用所能执行的操作。AndroidManifest.xml是所有应用一定要有的内容，没有的话应用无法运行。该文件中会定义若干属性：例如user-permission，其标识了应用会使用到的权限，应用的任何行为如果没有permission便无法完成；activities，其标识了应用会进行的activity。

activity在安卓App中的含义是代表了一个具有界面的屏幕，其往往表示前端与用户的交互行为，比如某个按键、某个界面等等。将其含义延伸就是在前端的行为，其可能会涉及用户隐私。应用中要使用到的activity必须在清单中写明，未写明的会被系统所忽略。因此即使应用进行了加壳处理，运行时需要的activity也一定得写出来。activity的名字必须与对应的类名一致，且出于开发与沟通的需要，程序员不会在activity相关模块中修改变量名和类名实现代码混淆，在真实性上能够保证。

在ManifestScanner模块中，我们通过正则匹配的方式收集user-permission和activity。Google在设计权限时，将一部分权限声明为“危险”权限，这些权限包含读取用户通讯录、地理位置、短信箱等与用户隐私高度敏感的内容，我们收集的权限也是这些部分，同时会根据Google的文档的分类方法将这些权限进行分类。对于收集到的Activity，因为其名称的真实性，我们尝试通过搜索关键词，即寻找隐私相关的名词以及和操作有关的动词构造动宾词组。如果成功构建则其会被收集起来，等待下一步进行分析。

在对manifest文件分析完毕后，所有的Activity和Permission会在manifest.txt中显示出来。

权限组名	权限名
CALENDAR	READ_CALENDAR WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE

3.1.4 代码分析

Dalvik虚拟机是Jvm的变种，其并不使用Jvm的栈式计算架构，而是使用寄存器来进行计算，因而其字节码会与jvm字节码有所不同。但Dalvik字节码依然有统一的格式，能被统一的处理。Dalvik字节码指令繁多，但其中对分析有必要的指令并不多。

在代码分析中，因为反编译出来的smali文本格式规整，我们也通过正则匹配的方法来搜索敏感代码。首先是文件的第一行是类的声明，从中可以提取出该类的全限定名。然后是方法的声明。方法的声明以".method"开头，以".end"结尾，通过检测这两个行的出现可以构造出每个方法的代码区间。另外我们也可以进行正则捕获来获取这个方法的方法名、参数表、返回值。在函数收集时通过其类名、方法名、参数表来判断两个方法是否相同。

```
.method private final c()Ljava/lang/String;
    .locals 4

    .line 47
    invoke-static {}, Lcom/tencent/mobileqq/kandian/base/utils/TraceUtil;->a()V

    .line 48
    new-instance v0, Ljava/lang/StringBuilder;

    const-string/jumbo v1, "{"

    invoke-direct {v0, v1}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V
```

在发现方法声明后，便会进入对方法体内部代码分析的模式。我们主要关注两类指令：const-() 常量声明指令和invoke-() 方法调用指令，而常量声明指令中关注字符串常量const-string和类常量const-class。

首先是方法调用指令，通过正则捕获可以获取到被调用函数的全类名、方法名和参数表，如果在已扫描到的方法集合里找到这个方法，则这两个方法间便产生了一条调用关系。若是没有，则会生成一个方法来构建调用关系。当然，对于android和java的库函数的调用不会构建调用关系，因为库函数的内容并不是分析的重点。但是，部分安卓的api会包含涉及用户隐私的操作，因而它们会被特别监视。如果某个方法内调用这些敏感函数，则它会被标注上一种敏感类型，成为需要注意的函数。另外我们也会通过调用的方法名和类名判断这个方法是否调用了加密函数，是否会对获取到的信息进行加密处理。

对于常量声明代码，首先要提到对于敏感行为的另一种启动方式。一些行为被封装成了Android自带的Activity，而要启动活动则要通过Intent（意图）。Intent的构造有多种方式。一种是通过字符串常量的声明并setType来修改Intent的类型，Android系统会根据其类型来启动与其相适应的活动，例如调用照相机必须要用“android.media.action.IMAGE_CAPTURE”常量构造Intent，那么这个常量的声明就会被监视。另一种是对应用内声明的活动的启用。通过声明类常量并将其储存入寄存器中，然后将该常量传入Intent的构造后便能启动应用内声明的活动。应用声明的活动列表和敏感性分析在之前已经扫描完成，通过匹配就能注意到这个方法的可能涉及隐私的行为。另外还有一种字符串常量，其以"content://"开头，它代表了应用将访问的地址，如"content://sms"是访问短信的地址，通过对这些内容的监视可以得知方法会访问哪些内容。

```
String constString = constStringInstruction.group(1);
if (constString.contains("content://sms")) {
    if (!function.privacyWord.visitSms) {
        function.privacyWord.visitSms = true;
        writer.append("Detected app visiting sms in ").append(String.valueOf(function)).append("\n");
    }
} else if (constString.equals("android.media.action.IMAGE_CAPTURE")) {
    if (!function.privacyWord.usesCamera) {
        function.privacyWord.usesCamera = true;
        writer.append("Detected app starting a camera in ").append(String.valueOf(function)).append("\n");
    }
```

通过对这些指令的分析，我们便能得知这个方法会涉及哪些隐私内容以及是否加密。但仅仅这些内容并不够。我们仍然会对方法的声明内容进行分析，使用类似的构造词组的方式，如果构造成功则会为该方法添加隐私行为。当然这对于修改名称的混淆手段无能为力，但这些代码在项目中一般占比不大。通过方法名提取行为仍然是重要手段之一。

所有包含隐私内容获取行为和加密行为的方法都会在code.txt中进行标注。

3.1.5 Call Graph构建

在所有方法扫描完毕后，我们可以得到所有方法间的调用关系。有了调用关系和敏感点，就可以在调用图上进行敏感性传播。该传播算法思想基于队列的worklist算法，首先将所有敏感方法加入worklist中，对每个方法A处理时，如果调用它的某个方法B的敏感性和其不一样，也就是A有某个敏感点是B所没有的，那么B的敏感性就会刷新，即B=B∪A，并将B加入队列。由于每次方法的刷新都相当于在格上进行上升，因此这个算法一定能够停止。在分析中主要选择了朝根函数方向进行传播，原先还有向下传播的分析，但后来发现没有必要。每个没有应用内没有调用者的方法被看作是根方法，它们可能会被安卓系统调用，但不会被业务代码调用。在早期的计划中，曾考虑过通过敏感点和传播距离为方法进行评分，但由于这种方法因为缺少有效的数据衡量手段和模型而放弃了。

```
for (Function caller : f.beCalledBy) {
    if (scanActivity && !caller.activities.equals(f.activities)) {
        caller.activities.addAll(f.activities);
        queue.add(new Node(caller, ScanDirection.BACK));
    }
    if (scanPrivacy && (caller.privacyWord.differ(f.privacyWord) || (f.privacyWord.isEncrypted && !cal
        caller.privacyWord.merge(f.privacyWord);
        if (f.privacyWord.isEncrypted) caller.privacyWord.isEncrypted = true;
        if (scanActivity && !caller.haveSpecialActivity()) {
            caller.activities.add(new Activity(caller.toString()));
        }
        queue.add(new Node(caller, ScanDirection.BACK));
    }
    if (caller.beCalledBy.size() == 0 && (caller.usesPrivacy() || caller.haveSpecialActivity())) {
        rootFunctions.add(caller);
    }
```

在构建过程中，每一次传播都会在code.txt中以调用箭头的形式进行标注。同时文件中也会汇总根方法，统计其获取的隐私内容、可能存在的处理行为以及是否有加密。当然由于方法职责的分化，一般一个方法的敏感点不会过多，而且其内容都是比较相近的。

获取隐私内容本身也代表了一种隐私行为，因此如果一个方法中获取了某类隐私内容，那么对应的隐私行为也会被自动附带上。

构建结束后，我们便完成了对代码中个人信息的收集、处理操作的寻找以及判断是否进行过安全处理，收集到的行为会被存储起来准备与隐私政策中的内容进行比对。

3.1.6 隐私行为的处理

在前面的代码扫描中，我们获得了基于manifest的activity、从代码中收集到的隐私内容扫描和推断出的行为。前三者会先进行合并，然后在通过permission集合进行过滤。因为没有获得permission的隐私行为是无法进行的。在进行过滤时，会对一些近义词进行合并。例如share、send、publish等表示网络上传的行为会被统一为upload。随后根据每一条permission将旧行为中能对应的加入新的行为集合中。

```
if(permissions.contains(new Permission( descriptor: "SEND_SMS"))){
    for(Activity activity:transformedOld){
        if(activity.isNormal()) continue;
        if(activity.target.equals("sms")&&(activity.action.equals("send"))) neu.add(activity);
    }
}
if(permissions.contains(new Permission( descriptor: "RECORD_AUDIO"))){
    for(Activity activity:transformedOld){
        if(activity.isNormal()) continue;
        if(activity.target.equals("audio")&&(activity.action.equals("record"))) neu.add(activity);
    }
}
```

3.2 隐私文本分析方法

3.2.1 HanLP 实现文本分词

3.2.1.1 概要

中文分词工具HanLP在github上排名靠前，因此我们一开始便考虑直接使用HanLP处理隐私政策文本，将文本进行分词后再分析。但在查阅了大量资料后，发现此事并不容易，导入过程十分艰辛，甚至试过hancs的在线文本分词，但满足不了我们的需求。最后终于在助教的帮助下，修改了maven的配置以及.pom文件的依赖，将maven换源为阿里云，成功导入了HanLP。附上添加的依赖和配置的修改：

```
<dependencies>
  <dependency>
    <groupId>com.hankcs</groupId>
    <artifactId>hanlp</artifactId>
    <version>portable-1.7.8</version>
  </dependency>
</dependencies>

<properties>
  <maven.compiler.source>8</maven.compiler.source>
  <maven.compiler.target>8</maven.compiler.target>
</properties>
```

添加完HanLP包后，将隐私政策文本文件按行读取，然后直接调用HanLP.segment方法分词即可

3.2.1.2 关键代码实现

```
boolean isUse = false;

    //得到含有标签的句子
    SearchSentence s = new SearchSentence();
    s.getInfo(filename);
    StringBuilder str = new StringBuilder();

    //分词
    for (String sentence :
Objects.requireNonNull(transformation(stoken.toString()))) {
      str.append(sentence);
      writer.append("包含该信息的句子：  ").append(sentence).append("\n");  //原始句
子
      writer.append("去除停用词后：
").append(String.valueOf(NotionalTokenizer.segment(sentence))).append("\n");//去
除停用词
      List<Term> terms = NotionalTokenizer.segment(sentence);//标准分词
      writer.append("提取动词后：  ");
      writer.newLine();
      Term verb = null;
      for (Term t : terms) {
        if (t.nature == Nature.vn) {
          isUse = true;
          writer.append(String.valueOf(t)).append(" ");
          for (Stoken stokenv : Stoken.values()) {
            for (String w : stokenv.words) {
              if (t.word.contains(w))
                policyActions.add(new PolicyAction(t.word));
            }
          }
        } else if (t.nature == Nature.v) {
```

```
            isUse = true;
            verb = t;
            writer.append(String.valueOf(t)).append("  ");
          } else if ((t.nature == Nature.n || t.nature == Nature.nz) && verb !=
null) {
            for (Stoken stokenv : Stoken.values()) {
              for (String w : stokenv.words) {
                if (t.word.contains(w))
                  policyActions.add(new PolicyAction(verb.word + t.word));
              }
            }
          }
        }
      }
      writer.newLine();
      writer.newLine();
```

3.2.2 LDA模型实现关键词抽取

3.2.2.1 概要

LDA模型是一种能够体现文本语义关系的关键词提取算法，包含词、主题和文档三层结构。LDA 最早是由 Blei 等，以pLSI 为基础，提出的服从 Dirichlet 分布的 *K* 维隐含随机变量表示文档主题概率分布、模拟文档的一个产生过程。后来 Griffiths 等对参数*β*施加了 Dirichlet 先验分布，使得 LDA 模型成为一个完整的生成模型。

3.2.2.2 原理

LDA模型认为一篇文档的生成过程是：先挑选若干主题，再为每个主题挑选若干词语。最终，这些词语就组成了一篇文章。所以主题对于文章是服从多项分布的，同时单词对于主题也是服从多项分布的。基于这样的理论，我们可以知道，如果一个单词 *w* 对于主题 *t* 非常重要，而主题 *t* 对于文章 d又非常重要，那么单词 *w* 对于文章 *d* 就很重要，并且在同主题的词 (= 1,2,3, ...)里面，单词 *w* 的权重也会比较大。而这正好可以解决我们上面所描述的问题。所以下面就要计算两个概率：单词对于主题的概率和主题对于文档的概率。这里我们

选择 Gibbs 采样法来进行概率的计算。具体公式如下：

主题Tk 下各个词Wi 的权重计算公式：

$$P(w_i|T_k) = \frac{C_{ik} + \beta}{\sum_{i=1}^{N} C_{ik} + N \cdot \beta} = \varphi_i^{t=k}$$

文档Dm下各个词Tk 的权重计算公式：

$$P(T_k|D_m) = \frac{C_{km} + \alpha}{\sum_{k=1}^{K} C_{km} + K \cdot \alpha} = \theta_{t=k}^{m}$$

现在得到了指定文档下某主题出现的概率，以及指定主题下、某单词出现的概率。那么由联合概率分布可以知道，对于指定文档某单词出现的概率可以由如下公式计算得到：

$$P(w_i|D_m) = \sum_{k=1}^{K} \varphi_i^{t=k} \cdot \theta_{t=k}^{m}$$

基于上述公式，我们就可以算出单词 *i* 对于文档 *m* 的主题重要性。

3.2.2.3 关键代码实现

```
for (int j = 0; j < strArray.length; j++) {
String word = strArray[j];//获得当前要计算的单词
int i = wordIndexMap.get(word);//获得给定单词在单词-主题分布矩阵中的行号
//计算单词在指定文档 m 中的主题概率权重
double word2TopicWeightSum = 0.0;
//遍历所有主题,计算其对于单词 i 的频次总和
for (int k = 0; k < topicSize; k++) {
 word2TopicWeightSum += word2TopicCount[i][k];
}
double topic2DocWeightSum = 0.0;
//遍历所有主题,计算其对于文档 m 的频次总和
for (int k = 0; k < topicSize; k++) {
 topic2DocWeightSum += topic2DocCount[m][k];
}
double weightSum = 0.0;
//遍历所有主题,计算其单词 i 对于文档 m 的主题概率权重
for (int k = 0; k < topicSize; k++) {
 double word2TopicWeight = (word2TopicCount[i][k] + beta)/(word2TopicWeightSum
+wordSize*beta);
 double topic2DocWeight = (topic2DocCount[m][k] + alpha)/(topic2DocWeightSum +
topicSize*alpha);
 weightSum += word2TopicWeight*topic2DocWeight;
}
resultMap.put(word, weightSum);
 }
```

3.2.2.4 缺陷

缺陷：基于 LDA 主题概率模型的关键词提取方法的准确度，会严重依赖于基础语料库，而这个语料库还需要有一定的丰富性，这样才可以使得计算的概率值具有一定的准确度。这就会造成比较大的人力投入，对于模型的灵活扩展就会受到限制。

解决办法：借助于知网，或者同义词林的方法获得单词的准确语义关系

3.2.3 基于信息熵实现短语抽取

3.2.3.1 概要

信息熵是对于分布纯净度的一个度量，这个值随着分布的纯净度增加而降低。所以当分布中只有一种情况时，那么信息熵就为最小，假设为0。既然信息熵有这样的特性，那就可以用来衡量两个词是不是经常组合在一起的情况，因为如果这两个词是固定搭配的时候，那么它们互为对方的唯一连接，也就是从任何一个词的角度来说其连接词的分布都是唯一的。

3.2.3.2 原理

互信息是体现两个或者多个元素之间的依赖关系。以两个元素为例（以下没有特殊说

明，均以两个元素之间的关系为例进行说明），互信息计算公式：

$$\mathrm{MI} = P(X,Y)\log\frac{P(X,Y)}{P(X)\cdot P(Y)}$$

其中

P(X, Y)：表示元素 *X* 和元素 *Y* 共同出现的概率。P(X)：表示元素 *X* 单独出现的概率。P(Y)：表示元素 *Y* 单独出现的概率。而信息熵是用来描述一个变量的确定性的，如果一变量的确定性越高，那么它的信息熵就越小，相反则越大。对应到我们的场景来说，如果单词 *X* 和单词 *Y* 组合的确定性越大（即这两次固定搭配的概率越大），那

么它们的信息熵就越小，反之则越大。所以针对这个问题，我们分别对单词对中的单词加上信息熵，这样就可以对上述两种情况进一步区分。但是如上所述，固定搭配概率越大，其信息熵越小；而我们希望其作为关键短语的权值越大，但信息熵是一个 0～1 之间的数，所以我们最终使用：1-信息熵，作为计算的权值。其计算公式如下：

左单词的信息熵：

$$E_{L-X} = 1 - (-P(X)_Y \log P(X)_Y)$$

其中

P(X)y：表示在单词 *Y* 左边的单词集合中，*X* 的概率。

右单词的信息熵：

$$E_{R-Y} = 1 - (-P(Y)_X \log P(Y)_X)$$

其中

P(Y)x：表示在单词 *X* 左边的单词集合中，*Y* 的概率。上述两个概率，相对于单词对 *X*-*Y*，正好又可以描述为左右信息熵。所以结合第一步的互信息，这里就组成了基于互信息和左右信息熵的计算方法。但是完整的权重计算还需

要再补充一些东西。上面的左右信息熵只是计算了一个词的，而一般一个词组中每个单词的左右单词都不止一个，所以一个词组的左右信息熵是上面单个词左右信息熵的和，具体的计算如下。

$$E_L = \sum_{X \in Left(Y)} E_{L-X} = \sum_{X \in Left(Y)} 1 + P(X)_Y \log P(X)_Y$$

$$E_R = \sum_{Y \in Right(X)} E_{R-Y} = \sum_{Y \in Right(X)} 1 + P(Y)_X \log P(Y)_X$$

如上所述，我们在互信息的基础上加上信息熵，是为了提高“纯净”词组的概率。所以对于这三个概率，我们最终采用相加的策略进行。因为我们希望最终的权重值和这三个值中任何一个值都是单调增的关系，那么在基础的实现方式里面，有加法和乘法可选，但是如上对于信息熵的描述，我们知道当词组非常“纯净”的时候，即左右词是唯一互连时，其信息熵值为 1，如果采用乘法，则其最终的权重值就等于互信息，这样就没有提高“纯净”词组的概率，所以这里我们采用加法策略进行，这样就可以实现，三个权重值任意一个值高的时候，整体权重是高的，同时对于“纯净”词组的概率也有提升，所以最终的权重计算公式如下：

$$w_{X-Y} = E_L + E_R + \text{MI}$$

其中

Wx-y：表示由词 *X* 和词 *Y* 组成的词组的关键短语权重。

3.2.3.3 关键代码实现

```
for (Entry<String, MIAndEntropyBean> entry : wordsMap.entrySet()) {
//计算互信息
String wordArrayString = entry.getKey();
//获得总的词语计数
int wordArrayCount = entry.getValue().getWordCount();
String rightWordString = entry.getValue().getRightWord();
String leftWordString = entry.getValue().getLeftWord();
double leftWordCount = wordCountMap.get(leftWordString);
double rightWordCount = wordCountMap.get(rightWordString);
double MI = (wordArrayCount/totalWordArrays)/((leftWordCount/totalWords)*
(rightWordCount/totalWords));
//计算左信息熵
double leftSum = getEntropy(leftEntropyMap, rightWordString);
//计算右信息熵
double rightSum = getEntropy(rightEntropyMap, leftWordString);
//最终的权值
double weightSum = MI + leftSum + rightSum;
resultMap.put(wordArrayString, weightSum);

for (Entry<String, MIAndEntropyBean> entry : wordsMap.entrySet()) {
 //计算互信息
 String wordArrayString = entry.getKey();
 //获得总的词语计数
 int wordArrayCount = entry.getValue().getWordCount();
 String rightWordString = entry.getValue().getRightWord();
 String leftWordString = entry.getValue().getLeftWord();
 double leftWordCount = wordCountMap.get(leftWordString);
 double rightWordCount = wordCountMap.get(rightWordString);
 double MI =
(wordArrayCount/totalWordArrays)/
((leftWordCount/totalWords)*
```

```
(rightWordCount/totalWords));
 //计算左信息熵
 double leftSum = getEntropy(leftEntropyMap, rightWordString);
 //计算右信息熵
 double rightSum = getEntropy(rightEntropyMap, leftWordString);
 //最终的权值
 double weightSum = MI + leftSum + rightSum;
 resultMap.put(wordArrayString, weightSum);
}
private static double getEntropy(
 Map<String, Map<String, Double>> leftEntropyMap,
 String rightWordString) {
 double leftSum = 0.0;
 Map<String, Double> leftMap = leftEntropyMap.get(rightWordString);
 double total = leftMap.get(WORD_COUNT);
 //计算信息熵
 for (Entry<String, Double> entry3 : leftMap.entrySet()) {
 leftSum += 1.0 +
entry3.getValue()/total*Math.log(entry3.getValue()/total);
 }
 return leftSum;
 }
```

3.2.3.4 缺陷

缺陷：该方法对于长尾词、异常组合的词会给予比较大的权重，但是这在大部分情况下是有问题的。一方面是因为我们对于词组的选择只是基于滑动的窗口，所以会有："造成-比如"这样的词出现，而这样的词最终计算的结果会比较大，特别是当整个单词集合非常大的时候。

解决办法：根据词出现的频率过滤掉低频的词 调整不同词的权重

3.2.4 抽取句子主干

3.2.4.1 贝叶斯算法

在项目之初,我们考虑使用朴素贝叶斯算法实现抽取文本主干，记录如下：

贝叶斯算法是基于贝叶斯理论提出的一种分类算法，贝叶斯学派是统计学的一个重要学派，其与经典概率学派的一个最大的区别就是提出了先验概率的概念。在经典概率学派的理论中，同一个事件发生的概率是固定的，比如扔硬币这件事情，你扔和我扔得到正面的概率是一样的。但是贝叶斯学派认为不一样。比如一个有经验的人扔和一个新手扔得到正面的概率就会有很大的区别。并且对于先验概率的计算，也给出了一个简单的计算方式：即为这个事件在历史上出现的概率作为先验概率的一个估计值。比如要计算我扔硬币得到正面概率的先验值，就把我历史上投硬币的情况进行统计，出现正面的概率即为我扔硬币得到正面的先验概率值。

而通常实现朴素贝叶斯分类需要三个阶段：

第一阶段：准备工作阶段。这个阶段的一项工作是为统计工作准备素材，即确定要统计的指标。另一个是对训练数据按照确定的指标进行统计。这个步骤也是传说中的特征工程部分。对于指标的确定既可以完全由人工确定，也可以确定一些规则、实现半自动化的指标提取。比如确定了一个规则为：统计元素的位置信息。那么对于单词、句子、高频词等只要类型为元素的都要统计其位置，当然对于位置的类型也可以预先定义，这样就可以实现自动的特征提取。

第二阶段：分类器训练阶段。这个阶段的任务就是生成分类器，主要工作是计算每个类别在训练样本中出现的频率及每个特征属性划分对每个类别的条件概率估计，并将结果记录下来。

第三阶段：应用阶段。这个阶段就是对待分类的数据，先根据在训练阶段确定的指标进行统计，获得其在各个指标下的具体值，然后根据具体的值计算其属于各个类别的概率。再然后将其归到概率值最大的类别中。

但由于在第二阶段我们没有找到足够的语料库以及时间限制等原因，最后项目没有采用该实现方式

3.2.4.2 自动摘要文本

经过不断地探索，我们在github上发现了HanLP这一强大的工具，查阅文档得知其已经实现了文本自动摘要功能，轮子已经造好了，所以在将maven换源为阿里云以及导入对应的数据包和接口后，我们首先把包含个人信息的句子归类，然后再进行动词判定后再将该类下的所有句子合起来进行摘要，若仍然包含该个人信息，则可确定隐私政策文本中声明使用了该信息。每一句可能涉及隐私的语句都会标注词性并分类后加入policy.txt中。此外我们通过在语料库中搜索相近的动词和名词构造表示行为的词组的方式，为后面和代码中提取的行为比对做准备。

```
 public static boolean UseOrNot(Stoken stoken,String filename)throws IOException
{

    boolean isUse = false;

    //得到含有标签的句子
    SearchSentence s = new SearchSentence();
    s.getInfo(filename);
    StringBuilder str = new StringBuilder();

    //分词
    for (String sentence :
Objects.requireNonNull(transformation(stoken.toString()))) {
      str.append(sentence);
      writer.append("包含该信息的句子：  ").append(sentence).append("\n");  //原始句
子
      writer.append("去除停用词后：
").append(String.valueOf(NotionalTokenizer.segment(sentence))).append("\n");//去
除停用词
      List<Term> terms = NotionalTokenizer.segment(sentence);//标准分词
      writer.append("提取动词后：  ");
      writer.newLine();
      Term verb = null;
      for (Term t : terms) {
        if (t.nature == Nature.vn) {
          isUse = true;
          writer.append(String.valueOf(t)).append(" ");
          for (Stoken stokenv : Stoken.values()) {
            for (String w : stokenv.words) {
              if (t.word.contains(w))
                policyActions.add(new PolicyAction(t.word));
            }
          }
        } else if (t.nature == Nature.v) {
          isUse = true;
          verb = t;
          writer.append(String.valueOf(t)).append("  ");
        } else if ((t.nature == Nature.n || t.nature ==
Nature.nz||t.nature==Nature.gp||t.nature==Nature.gi) && verb != null) {
          for (Stoken stokenv : Stoken.values()) {
            for (String w : stokenv.words) {
              if (t.word.contains(w))
```

```
                policyActions.add(new PolicyAction(verb.word + t.word));
            }
          }
        }
      }
```

3.2.5 敏感信息自动分类

3.2.5.1 概要

根据国家互联网信息办公室、工业和信息化部、公安部、国家市场监督管理总局印发《常见类型移动互联网应用程序必要个人信息范围规定》，该《规定》明确了39种常见类型APP的必要个人信息范围，可见，app涉及到的个人信息种类并不多，因而基于个人词典实现分类也具有一定的可行性。

3.2.5.2 原理

通过将39种的个人信息插入到hanlp提供的用户词典，信息类型存储为词性，从而巧妙地通过分词，然后筛选词性的方式将隐私政策中的个人信息过滤出，保存至policy.txt文件中

3.2.5.3 关键代码实现

```
public void Classification() throws IOException {
    // 将个人信息关键词插入到用户词典
    Initialize();
    BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream("哔哩哔哩隐私政策.txt")));
    String data = "";
    while((data = br.readLine())!=null){
      StandardTokenizer.SEGMENT.enablePartOfSpeechTagging(true);  // 支持隐马词性
标注
      List<Term> termList = HanLP.segment(data);
      for (Term term : termList){
        Add(term);
      }
    }
  }
public void Add(Term term){
    if(term.nature == Nature.create("行踪轨迹信息")){
        TrackInformation.add(term.word);
    }
    if(term.nature == Nature.create("金融账户信息")){
      FinancialAccountInformation.add(term.word);
    }
    if(term.nature == Nature.create("医疗健康信息")){
      MedicalAndHealthInformation.add(term.word);
    }
    if(term.nature == Nature.create("特定身份信息")){
      SpecificIdentityInformation.add(term.word);
    }
    if(term.nature == Nature.create("宗教信仰信息")){
      ReligiousBeliefInformation.add(term.word);
    }
    if(term.nature == Nature.create("生物识别信息")){
      BiometricInformation.add(term.word);
    }
    if(term.nature == Nature.create("未满14周岁未成年人的个人信息")){
```

```
            MinorInformation.add(term.word);
        }
```

3.2.5.4 缺陷

缺陷：基于词典进行匹配方式通常都具有维护效果不佳的问题，若增加了新词，则对应词典也要同步添加

解决方法：扩充词典容量，从个人信息语料库中添加进词典 或者通过神经网络模型训练出一个分类器（如果时间充裕）

3.2.6 行为内容比对

通过对代码分析和对隐私政策的分析，我们得到了两套针对隐私的处理行为，但这两套行为毕竟在语言上是不同的，中间翻译存在困难。但隐私政策作为有固定体系的文章，其行为动词范围是比较小的。因此我们通过寻找隐私政策中的高频动词，将其翻译为若干可能对应的英文动词，构造一个匹配词典。在最后的比对阶段，我们对每一个activity在政策中提到的行为里尝试匹配，如果成功在类别和动作上匹配，则可以认为二者是符合的。当然由于翻译词典始终会存在缺漏，我们也会提供原始的两套行为供用户进行比对。在比对结束后，整套扫描流程便结束了。用户可以在控制台看到两套行为和比对结果，并在文件夹下的manifest.txt,code.txt,policy.txt中看到扫描过程中的中间结果。

```
public void check(Set<Activity> activities,Set<PolicyAction> policyActions){
        for(Activity activity: activities){
            Stoken stoken=activity.toStoken();//Stoken表示隐私信息的类别
            ACTION:for(PolicyAction p:policyActions){
                if(p.stoken.equals(stoken)){
                    String[] a=p.translate();
                    for(String s:a) {
                        if (activity.action.equals(s)) {
                            System.out.println(activity + "符合");
                            break ACTION;
                        }
                    }
                }
            }
        }
    }
```

四. 成果展示与收获

4.1 成果展示

我们对若干app进行了检测，分别有qq、微信、哔哩哔哩、京东、哈啰单车、南京大学app、U净洗衣、永安行、扫雷F。对部分App的扫描结果：

App名称	代码隐私行为数	符合数	符合比例
QQ	9	8	88%
微信	6	5	83%
哔哩哔哩	5	5	100%
京东	7	3	43%
哈啰单车	11	9	82%

通过人工复检京东的隐私政策，确认京东确实存在与隐私政策不符的行为。而南京大学app、U净洗衣、永安行对Apk进行了加壳处理，难以推断其具体行为。而扫雷F没有隐私政策提供，但通过行为分析可以发现其存在对地理位置、照相机的获取行为，这是非常令人怀疑的。

App名称	代码隐私行为数	符合数	符合比例
南京大学	3	2	67%
U净	0	0	0%
永安行	4	2	50%

对这些加壳的app程序表现不佳，这是静态分析框架的固有缺陷。在经过多个app的反编译后，得出规律：一般代码量非常小（几mb）的的应用和非常大（数百mb到1g以上）的应用不会选择加壳，在几十mb的应用会选择加壳。不选择加壳的考虑其一可能是核心业务逻辑并不储存在app中，而是储存在web服务器上；其二可能是业务逻辑过大，无法加壳封装成库。

4.2 未来展望

在整个程序中仍然存在一些不足，例如安卓api可能收集不全、隐私语料库容量小等问题。未来可能会考虑通过统计手段获取大量隐私政策文本中的动词，完善语料库和对照词典。同时在过滤行为时减小对部分词的合并操作，获取更高的准确性。