数据科学大作业
编程练习辅助系统

何文兵 181250043@smail.nju.edu.cn &
李淳 181250068@smail.nju.edu.cn &
秦锐鑫 181250117@smail.nju.edu.cn

***Abstract*— 这篇文章包括了我们在本门课程大作业中所做的所有工作，我们用 Python 构造了一个具有以下模块的系统：代码复杂度分析、面向用例检测、题目推荐系统、题目聚类。我们认为这个系统可以很好的帮助同学们提升自己的编程能力，帮助老师们针对不同学生制定更为合适的计划。所有的代码和文件都可以在 代码仓库 这里找到。**

***Index Terms*— Python、编程、代码复杂度、机器学习**

I. 代码复杂度模块

圈复杂度 (Cyclomatic complexity) 是一种用于衡量代码复杂度的标准，具体表现为一个模块判定逻辑（条件分支、循环）的复杂度，数量上为线性无关的路径条数。圈复杂度大说明代码模块逻辑复杂，较难进行测试和维护，代码质量较低。

通过 python 中的 ast 与 astpretty 可以生成一段 python 代码的抽象语法树（AST）并以用户友好的方式打印出来，如下以程序

```
a = 1
if a > 0:
    print(a)
else:
    print(0)
```

使用下列代码可得到可视化的抽象语法树

```
import ast
import astpretty

astpretty.pprint(ast.parse('''a = 1
if a > 0:
    print(a)
else:
    print(0)
''', "exec"),indent="  ")
```

生成 ast 为

```
Module(
  body=[
    Assign(
      lineno=1,
      col_offset=0,
      targets=[Name(lineno=1, col_offset=0, id='a',
                    ctx=Store())],
      value=Num(lineno=1, col_offset=4, n=1),
    ),
    If(
      lineno=2,
      col_offset=0,
      test=Compare(
        lineno=2,
        col_offset=3,
        left=Name(lineno=2, col_offset=3, id='a',
                  ctx=Load()),
        ops=[Gt()],
        comparators=[Num(lineno=2, col_offset=7, n=0)],
      ),
      body=[
        Expr(
          lineno=3,
          col_offset=4,
          value=Call(
            lineno=3,
            col_offset=4,
            func=Name(lineno=3, col_offset=4, id='print',
                      ctx=Load()),
            args=[Name(lineno=3, col_offset=10, id='a',
                      ctx=Load())],
            keywords=[],
          ),
        ),
      ],
      orelse=[
        Expr(
          lineno=5,
          col_offset=4,
          value=Call(
            lineno=5,
            col_offset=4,
            func=Name(lineno=5, col_offset=4, id='print',
                      ctx=Load()),
            args=[Num(lineno=5, col_offset=10, n=0)],
            keywords=[],
```

Trabalho de Conclusão de Curso apresentado ao Instituto Nacional de Telecomunicações, como parte dos requisitos para a obtenção do Certificado de Pós-Graduação em Engenharia Clínica e Engenharia Biomédica. Orientador: Prof. (XXXXX). Trabalho aprovado em XX/20XX.

```
            ),
        ),
      ],
    ),
  ],
)
```

A. 模块的圈复杂度

通过遍历程序抽象语法树中的节点，递归分析抽象语法树中的条件分支表达式、循环表达式以及函数模块，分别计算各节点的圈复杂度

```
If(expr test, stmt* body, stmt* orelse)
For(expr target, expr iter, stmt* body,
    stmt* orelse)
While(expr test, stmt* body, stmt* orelse)
FunctionDef(identifier name, arguments args,
        stmt* body,expr* decorator_list,
        expr? returns)
```

在计算 python 程序复杂度时，使用了 mccabe 库，原理即分析程序的抽象语法树。

B. 自定义程序复杂度

由于提交的 python 程序可能涉及多个模块，之前已计算出各节点即各模块的复杂度，故需要使用一种算法通过多个模块的复杂度计算出一个值以衡量整个程序的复杂度。

由于程序由模块组成，若程序模块复杂度相同，则模块越多，组成程序的复杂度应该越高，故不适合采用简单求均值的算法；同时，由于将一个模块进行拆分时，实际上整个程序的复杂度应该要降低，故不能简单求和，例如以下两个程序

```python
# program1 demo_combine.py
dif = 0
for i in range(50):
    for j in range(25):
        for k in range(15):
            if i + j + k > 50:
                dif += 1
            else:
                dif -= 1
print(dif)

# program2 demo_separate.py
def countDif(num):
    if num > 50:
        return 1
    else:
        return -1


dif = 0
for i in range(50):
    for j in range(25):
        for k in range(15):
            dif += countDif(i+j+k)
print(dif)
```

分析得到 program1 demo_combine.py 的模块圈复杂度为 5，而 program2 的模块圈复杂度分别为 2, 4 而经过拆分,program2 demo_separate.py 应该更简单，更容易维护。

因此考虑一种对各个模块复杂度，根据复杂度大小进行比例变化后进行累加的算法。

设各个模块复杂度为 Ci，程序复杂度为 P

$$P = sum(Ci * Ci/10)s$$

以上述两个程序为例

第一个程序 demo_combine.py 程序复杂度即为 $P1 = 5 * 5/10 = 2.5$

第二个程序 demo_separate.py 程序复杂度即为 $P2 = 4 * 4/10 + 2 * 2/10 = 2$

以此作为程序复杂度

cal 将标准输出流重定向到 out.log converter 用于从 out.log 中的字符串中的有效数据提取出来并计算程序复杂度并存储为 csv 文件 csv 文件格式为 user_id,question_id, submit_id，codeComplex

C. 参考文件

Python 抽象语法树

II. 面向用例检测模块

A. 介绍

该模块启动后可以自动检测所有的提交，并且返回一个 *list*，里面是所有有面向对象嫌疑的提交

检测完后会返回有嫌疑的提交的个数和总检测数，同时会返回一个 *dict*，考虑到同学 *id* 和题目 *id* 的双重唯一性，*dict* 中会以两个 *id* 的字符串拼接为索引，值为一个 *Case* 对象，对象中包含了大量的信息

后文将用 $< userId > - < caseId > - < uploaderId >$ 这样的格式来标注出某人在某道题上的某次提交

```python
class Case:

    def __init__(self, case, uploader):
        # 用户 id
        self.uploader = uploader
        # 题目 id
        self.case_id = case['case_id']
        # 题目类型
        self.case_type = case['case_type']
        # 最终得分
        self.final_score = case['final_score']
        # 提交记录
        self.upload_records = case['upload_records']
        # 被检测出有面向用例嫌疑的提交
        self.match_records = dict()
        # 排序用的 key
        # 数值上为 len(match_records)/len(upload_records)
        self.sort_key = 0
```

助教可以随机抽查 *dict* 中的数据，然后查看某个同学在某个问题上所有的提交。

B. 面向用例检测

面向用例就是同学们在提交代码的时候投机取巧，使用了大量的 if-else 级联或者代码中出现了明文表示的输入输出来通过题目，代码描述的逻辑和解题基本上没有关系。

b.1. 面向用例代码分析

我们先来看一个面向用例代码的例子

```
nmst = input().split(' ')
num = int(nmst[1])
for i in range(num):
    input()
if nmst == ['250', '610', '204', '239']:
    print(1544)
elif nmst == ['100', '251', '88', '95']:
    print(969)
elif nmst == ['2000', '4862', '1935', '306']:
    print(1075)
elif nmst == ['2500', '6071', '1760', '669']:
    print(1159)
elif nmst == ['50', '122', '14', '3']:
    print(1215)
elif nmst == ['1000', '2450', '925', '987']:
    print(762)
elif nmst == ['7', '11', '5', '4']:
    print(7)
elif nmst == ['10', '20', '9', '4']:
    print(576)
elif nmst == ['20', '43', '11', '19']:
    print(491)
elif nmst == ['500', '1229', '5', '23']:
    print(1252)
else:
    print(nmst)
```

根据这个例子我们可以观察出这样的几个事实：

1. 面向用例编程可能会产生大量的 if-else
2. 面向用例编程可能需要获得所有的输入
3. 面向用例编程可能会直接输出结果

b.2. 基于输出的面向用例检测

基于第三个事实，我们做出了一个非常严苛的面向用例检测代码，其中核心代码如下

```
# 所有用例的输出
out_ = case['output']
outs = out_.split('\n')
# _ 表示每个用例的输出
for _ in outs:
    if _ == '':
        continue
    # prints 是代码中所有的 print 语句
    for __ in prints:
        if _ in __:
            count += 1
            print(_+" "+__)
            break
```

这段代码会将代码里面所有的 *print* 函数提取出来，然后检测被 *print* 的数据是否含有正确答案。为什么我们要使用这样的方法呢？原因很简单，我们的在线测评系统完全是通过 *print* 函数来打印答案，然而很多题目的答案往往不会是打印一个常量，而是打印一个表达式。如果你在答案中打印出来一个常量，那么我们认为你有面向用例的嫌疑。当然，对于一些比较简单的题目，往往答案就是 True 或者 False，所以大家最后也会在 *print* 函数中写个 True 或者 False。这部分内容在后面讲，下面我们看一下基于第二个事实检测出面向用例的情况。

b.3. 基于输入的面向用例检测

在基于面向用例的检测之后，我们核对了所有有面向用例嫌疑的例子，在 48117-2081-249325 提交中出现了这样的代码，分数为 100 分

```
a = input()
b = input()
a.lower()
b.lower()
if a.count(b) == 2:
    print(3, end='')
else:
    print(a.count(b), end='')
```

在讨论是否这段代码是否面向用例之前，我们来看一下题目

题目描述

这是一道模板题。

给定一个字符串 A 和一个字符串 B，求 B 在 A 中的出现次数。A 和 B 中的字符均为英语大写字母或小写字母。

A 中不同位置出现的 B 可重叠。

输入描述

> 输入共两行，分别是字符串 A 和字符串 B。

输出描述

> 输出一个整数，表示 A 在 B 中的出现次数。

测试样例

样例 1：输入-输出-解释

```
zyzyzyz
zyz
```

```
3
```

数据范围与提示
> 1<=` A,B 的长度` <=100000, ` A、B` 仅包含大小写字母

在这次提交中，我们可以分析出写作业的人没有考虑到字符串重合的问题，直接使用了 *count* 函数来判断一个字符串在另一个字符串中出现的次数，官方文档中对这个函数的描述是这样的

```
`str.count`(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences
of substring *sub* in the range [*start*, *end*].
Optional arguments *start* and *end* are interp-
-reted as in slice notation.
```

其中指出了这个方法只针对 $non-overlapping$。因此，我们可以判断出来，在这次提交中，虽然提交者并没有将输入直接放到答案里面，但是提交者靠着对输入用例的分析，成功找到了输入的一些特点，那就是只有一个用例是有字符串重合的情况，其他的用例都是不重合了。

这个例子给了我们启发，那就是面向用例或许不一定需要直接将用例的输入写出来，而是可以结合题目，合理分析用例的结构，进而通过取巧的方式输出答案。另一方面，在 *json* 文件中的输入和真正转换到 python 中可以使用的变量的时候，用例的输入已经发生了变化，可能从字符串变成了一个列表。不过对于大部分题目而言，检测输入是否存在于代码中还是非常好用的。但是我们注意到，如果输入中存在一些 1，0，或者同学们经常使用的常量的时候，那就很容易发生误判了。

针对输入用例进行算法修改后，我们的核心代码变成了下面这样

```
# 所有用例的输入
in_ = case['input']
ins = in_.split('\n')
# _ 表示每个用例的输入
for _ in ins:
    if _ == '':
        continue
    # lines 是每一行代码
    for __ in lines:
        if _ in __:
            ic += 1
            si_ += _ + ' ' + __ + '\n'
            break

# 所有用例的输出
out_ = case['output']
outs = out_.split('\n')
# _ 表示每个用例的输出
for _ in outs:
    if _ == '':
        continue
    # prints 是代码中所有的 print 语句
    for __ in prints:
        if _ in __:
            count += 1
            print(_+" "+__)
            break
```

为什么在输入的时候我们选择扫描每一行代码呢？因为我们注意到 *input* 里面是不能填写明文的，因此输入用例会出现的地方其实是没有办法遇见的，在大量的 if-else 级联中，可能会有用例均匀分布在代码中。这种情况应该是像下面的代码一样

```
a = input()
if a == '9 1':
    a = input()
    if a == '1 2 3 -3':
        a = input()
        if a == '2 4 5 3':
            a = input()
            if a == '4 0 0 1':
                a = input()
                if a == '5 8 9 0':
                    a = input()
                    if a == '8 0 0 1':
                        a = input()
                        if a == '9 0 0 6':
                            a = input()
                            if a == '3 6 7 -9':
                                a = input()
                                if a == '6 0 0 2':
                                    a = input()
                                    if a == '7 0 0 1':
                                        a = input()
                                        if a == '3':
                                            print(2)
                                        else:
                                            if a== '6':
                                                print(4)
                                            else:

                                                print(1)
                                else:
                                    print(a)
                        else:
                            print(a)
                else:
                    print(a)
            else:
                print(a)
        else:
            print(a)
    else:
        print(a)
elif a == '35 29':
    print(1)
else:
    print(a)
```

因为存在很多用例，他们的前缀是完全一样的，所以 if-else 的深度也会随之增加，这样每一行都有可能是具有输入用例的。

总的来说，不管是基于输出还是基于输入，他们在非常简单的情况下都会失效，因此在使用的时候需要仔细甄别，我们也在检测环节打印出了被检测到的代码片段来帮助助教进行判断。

b.4. 大量出现的 if-else

我们不打算在代码中直接分析 if-else 的数量和级联情况，因为他对我们检测的性能有很大的折损，同时，大量的 if-else 是为了什么？其实还是为了慢慢筛选用例里面的输入，本质上还是基于输入面向用例。如果有一大堆 if-else 但是没有相应的明文的输入，那么你面向用例的嫌疑只会因为输出产生，如果你有一大堆明文的输入，那基本上已经可以肯定你面向用例了。所以我们认为，检测 if-else 得到的有嫌疑的代码应该是检测输入得到的有嫌疑的代码的子集。因此不需要对 if-else 进行重新的检测。

但是并不是所有的代码都是基于输入来进行面向用例的，他们有可能发现了输入的特点，然后利用输出来面向用例，就像 b.3 最开始给的例子一样，所以基于输入和基于输出他们并不是完全重合的。

另一方面，我们认为 if-else 的大量出现反应了一个事实，那就是大部分学生往往需要多次的提交、实验，才能获取所有的用例，因此我们总结出了另一个重要的检测指标，那就是提交次数。当然并不是提交次数多就一定是面向用例编程，他有可能提交了很多次只是因为找不到 bug，也有可能是因为用例太少而不需要多次的提交来获得。但是我们认为这是一个比较直观的判断的依据。

C. 辅助检测

在上文中，我们提到了很多数据和指标，但是因为数据和指标比较多，展示的时候将会有一些优先级，不然会降低助教判断代码面向用例的效率。我们数据展示的顺序将会是以下顺序

1. 有嫌疑的提交次数和总提交次数
2. 所有有嫌疑的提交中他们的 [提交者 id，题目 id，在该题上的提交次数] 将会被打印出来，以提交次数和用例数的比值进行从大到小的排序。
3. 助教可以选择提交者 id，题目 id 来查看他在本题上所有提交的编号，以及每次提交的分数，哪些提交是有面向用例嫌疑的
4. 指定编号后可以查看在这个提交中被判断有嫌疑的那些 < 用例，代码 > 对和具体的代码

III. 题目推荐

A. 介绍

对于编程新手来说，我们认为提供针对性的训练是非常重要的，因为对于新人来说，如果各个方面都浅尝则止，那么他将没有办法体会到不同类型题目的特点和套路。因此，我们设计了题目推荐系统，并且参考了比较流行的几个推荐系统模型。在数据预处理阶段，我们为了方便后面推荐系统模型的训练，将 userId 和 caseId 分别映射到从 0 开始的序列，同时因为 test_data 包含所有 sample 的数据，所以由 test_data 构造出来的矩阵包括了 sample 的内容，基于此，我们可以将 test_data 中的 80% 作为训练集，20% 作为测试集。

在训练阶段，我们选择了 BiasSVD 作为模型，并且在 Google 提供的 colab 上进行训练，最后的结果为使用 RMSE 进行评估，结果为 10.7595，我们认为这个结果是可以接受的

B. 模型选择

在开始之前，我们先规定一下后文中使用到的一些变。M_{mn} 表示一个 m 行，n 列的矩阵，矩阵里的每一个元素表示第 m 个用户在第 n 道题目上的得分，e_{ui} 表示第 u 个用户在第 i 道题上的预测分数和实际分数的偏差，Q_{mk} 表示一个 m 行，k 列的矩阵，其中每行都表示一个用户，每列表示题目不同的特征，P_{nk} 表示一个 n 行，k 列的矩阵，其中每行表示一个题目，每列表示题目不同的特征。

虽然最基础和最流行的机器学习推荐系统模型是协同过滤模型，但是我们都知道，如果使用协同过滤模型，在 user 和 case 两个矩阵中，他们的列都是题目的特征，它训练的公式为

$$e_{ui} = m_{ui} - q_u p_i^T$$

但是这个模型忽略了一件事情，那就是不同用户在不同题目上的表现其实不能完全由一个题目的特征决定。

b.1. 关于偏差

我们先从电影推荐这个例子来考虑，如果泰坦尼克号的分数为 5 分，但是因为这部电影比较经典，在所有电影里面排名比较前，用户给他打分的时候会多打几分。但是如果一个用户比较具有批判心理，那么他会倾向于给电影打一个比平均分稍微低一点的分数。同样的，有部分同学水平可能特别高，那么这部分同学所做的题目的特征就会带上这部分同学的印记，他们会呈现出一种系统性的趋势，也会有一部分题目可能天生就会得到比别的题目更高的分数，又或者有一部分题目的翻译有问题（这个是真实存在的问题），这部分题目的得分可能会表现出低于同类型题目的倾向。在这里我们认为题目的特征仅仅是题目的分类，不包括题目的难度。因此我们选择了 BiasSVD 作为我们的训练模型。我们引入几个新的变量。b_{ui} 表示偏差在 m_{ui} 上的贡献，b_u 表示用户的偏差，b_i 表示题目的偏差，μ 表示平均分数

所以最后我们训练的公式为

$$e_{ui} = m_{ui} - q_u p_i^T - b_{ui} b_{ui} = b_u + b_i + \mu$$

在编码上来说，BiasSVD 是比较简单的，所以我们认为这个算法集中了便捷性和适用性。

最后，因为我们是矩阵分解算法，所以我们一开始并不知道题目的特征是多少，这个我们交给了模型自己来训练，尽管在数据中是包含了题目的特征的，但是数据中的特征相对来说比较局限，比如字符串中又可以分出动态规划 + 字符串，或者自动机 + 字符串等。所以我们选择了 k=15 来进行训练。

C. 结论

最后我们使用了 RMSE 来进行评估

$$RMSE(H, X) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(x_i) - y_i)^2}$$

他常用来作为机器学习模型预测结果衡量的标准。在 RMSE 下我们的结果为 10.7595，我们认为这个结果是可以接受的。因为 RMSE 衡量了所有偏差的平均值。这表明我们所有的预测出来的结果和真实值的差距只有 10 分，如果这道题的用例有 10 个，那么相当于只错了 1 个用例，如果用例有 5 个，相当于有一半的概率多做对或者多错一个用例。这表明我们的预测和实际情况已经非常接近了。

根据我们的结果重新计算一轮，然后对每个用户都按照题目的得分从小到大排序，从这个序列中，我们需要进一步考虑推荐的策略。这里我们给出一个比较简单但是实用的策略

1. 首先计算和最低分题目特征欧式距离最短的题目，这些题目和最低分题目是类型比较接近的
2. 其次在这些题目里我们按照题目难度和用户在这些题目上的最终得分进行排序
3. 最后我们再选出最适合用户的题目

D. 参考文献

Matrix Factorization Techniques for Recommender Systems

IV. 题目难度评估模块

A. 简介

在之前的模块中我们获得了一些统计数据，例如学生在该题目上的最终代码的程序复杂度、该题目采取面向用例的学生比例，该题目的学生平均得分，该题目的平均提交次数等等。于是在这个模块中我们尝试利用这些数据维度来评估一道题目在同学们心中的难度如何。

B. 数据匹配

data_match.py 文件对两份数据文件 csv_test_by_pkun.csv,csv_test_by_Qin.py 进行匹配整理，最终得到了 600 多道题目统计数据，并导入了 final.csv。

final.csv 文件的格式如下:

```
case_id, case_oriented_ratio, final_average_score
full_score_ratio, upload_times, average_complexity
classification
```

其中所有面向用例的提交，我们均视为同学作弊，在统计时最终成绩计为 0 分。

C. 主成分分析法降维

由于数据维度太多，我们无法对数据进行可视化，所以选择用主成分分析法 (Principal Components Analysis) 对数据进行降维处理

```
pca = PCA(n_components='mle', svd_solver='full',
          copy=True, whiten=True)
newX = pca.fit_transform(X)
print(pca.explained_variance_ratio_)
```

$n\_components = mle$ 意味着模型将根据计算结果自行选择最终维度，$svd\_solver = full$ 意味着采取完全的奇异值分解，$whiten = True$ 意味着会将降维后的数据进行归一化处理，使得各维度方差相当。

通过可解释方差比重的结果我们发现，选取的第一个向量占据了 51.6% 的信息值，第二个向量占据了 27.3% 的信息值，第三个向量占据了 15.1% 的信息值，其余两个维度总计只占 6% 左右，所以我们选择舍弃调其他向量，仅保留这三个向量

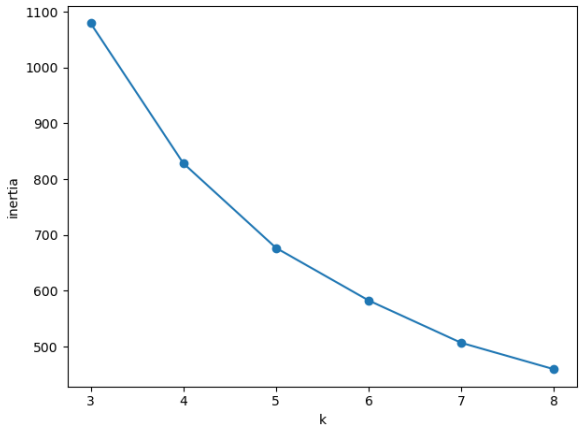
D. 聚类分析

得到了主向量之后，我们采用 K-means 算法进行聚类分析，采用了距离作为相似性的评价指标，认为两个对象的距离越近，其相似度越大。

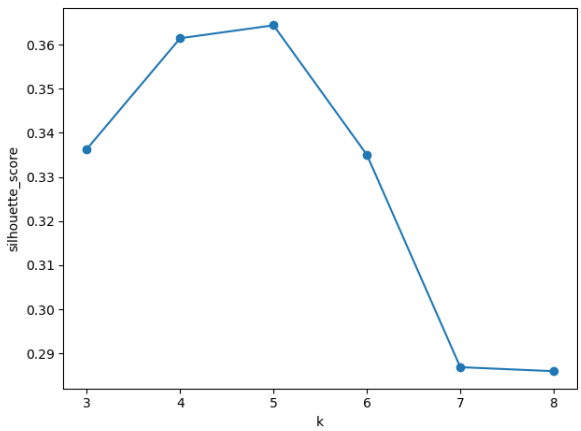
为了选取合适的簇数量，我们在 KmeansAssessment.py 中采取了三种评价指标:

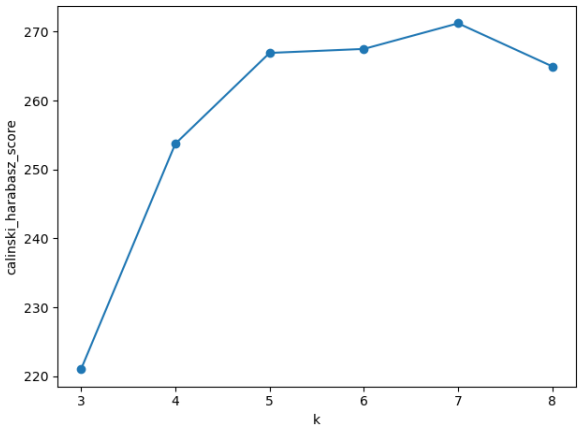
$$inertia\_ \ silhouette\_score \ calinski\_harabasz\_score$$

并绘制了当簇数目从 3 增加到 8 时，三项评价指标的走势图



inertia 是每个点到其簇的质心的距离之和，图形斜率越大，代表增加类别数量越能改善分类效果，从图中可以看出 $k = 4, k = 5$ 的分类效果变化不明显。





silhouette_score 是轮廓系数，针对样本空间中的一个特定样本，计算它与所在聚类其它样本的平均距离 a，以及该样本与距离最近的另一个聚类中所有样本的平均距离 b，该样本的轮廓系数为 (b-a)/max(a,b)，将整个样本空间中所有样本的轮廓系数取算数平均值，作为聚类划分的性能指标 s。轮廓系数的区间为：[-1,1]。-1 代表分类效果差，1 代表分类效果好。0 代表聚类重叠，没有很好的划分聚类。因此从该图中我们可以看出 $k = 4$ 和 $k = 5$ 的聚类效果是比较好的。

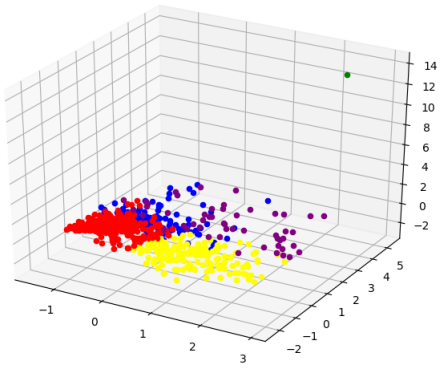
CH 指标是通过计算类中各点与类中心的距离平方和来度量类内的紧密度，通过计算各类中心点与数据集中心点距离平方和来度量数据集的分离度，CH 指标由分离度与紧密度的比值得到。从而，CH 越大代表着类自身越紧密，类与类之间越分散，即更优的聚类结果。

从 CH 指标可以发现，$k = 5$ 及之后的 CH 指标比较不错，但之前效果较好的 $k = 4$ 则一般

通过三张走势图的分析，我们可以发现当 $k = 5$ 时，能够在三项评价指标中都取得还不错的效果，因此最终我们设定簇的数量为 5。

```
kmeans = KMeans(n_clusters=6, copy_x=True).fit(newX)
```

然后将分类结果导入 final.csv 文件中，在 classification 列中列出了该题目所属类别，并对分类结果进行了可视化。



E. 评估结果

分类完成后，我们打开 final.csv 文件，通过 classification 我们可以观察到题目被分为了以下 5 类：

第 0 类：简单题，同学们面向用例的比例很小，并且代码复杂度也比较低，用例简单，大家提交次数都不多，得分都很高。一共有 289 道题目

第 1 类：中档题，虽然代码复杂低，但可能用例比较刁钻或者情景比较复杂，同学们的提交次数变得比较多（说明大家 debug 变困难了），但也侧面说明这一类题目是大家都在认认真真写的题目。这时已经有一部分同学开始感觉吃力因此选择面向用例的作弊手法了。

第 2 类：难题，由于题目比较难，同学们面向用例的比例大幅增加，此时差不多每道题都有超过 2/3 的同学选择了面向用例，但用例不太复杂，大家的代码复杂度也不高。我们推测应该是一些涉及到了高级的数据结构和算法的题目，过程并不复杂，但背后的原理比较困难。一部分同学选择自学后解决问题，剩下的同学就采取了面向用例的作弊方法。

第 3 类：也是难题，题目的逻辑相当复杂的题目，因此同学们的代码复杂度很高。这一类题目的得分情况相当极端，要么就几乎所有的同学都面向用例作弊了，要么就只有很少同学作弊。

第 4 类：奇葩题，因为分类显示这个类别下只有一道题，这道题大家的平均代码复杂度达到了惊人的 55.5，因此我们单独提取出了这道题目，并检查了大家的代码，下面我们就来展示一下这道题目

题目描述

将非负整数转换为其对应的英文表示。可以保证给定输入小于 2<sup

示例
输入:1234567
输出:"One Million Two Hundred Thirty Four Thousand Five Hu

```
class Solution(object):
    def numberToWords(self, num):
        """
        :type num: int
        :rtype: str
        """
        d1 = ['', 'One', 'Two', 'Three', 'Four', 'Five',
              'Thirteen', 'Fourteen', 'Fifteen', 'Sixteen
              'Seventeen', 'Eighteen', 'Nineteen', 'Twent
        d2 = ['', 'Ten', 'Twenty', 'Thirty', 'Forty',
            'Fifty', 'Sixty', 'Seventy', 'Eighty', 'Ninet
        if num == 0: return 'Zero'
        if num <= 20: return d1[num]
        if num < 100:
            t, d = num // 10, num % 10
            return d2[t] + ' ' + d1[d] if d > 0
                else d2[t]

        if num < 1000:
            h = num // 100
            if num % 100 == 0: return d1[h] + ' Hundred'
```

```
            return d1[h] + ' Hundred ' +
                self.numberToWords(num % 100)

        if num < 10 ** 6:
            th = num // 10 ** 3
            if num % 10 ** 3 == 0:
                return self.numberToWords(th) +
                    ' Thousand'
            return self.numberToWords(th) +
                ' Thousand ' +
                self.numberToWords(num % 10 ** 3)

        if num < 10 ** 9:
            mi = num // 10 ** 6
            if num % 10 ** 6 == 0:
                return self.numberToWords(mi) +
                    ' Million'
            return self.numberToWords(mi) +
                ' Million ' +
                self.numberToWords(num % 10 ** 6)

        if num < 10 ** 12:
            bi = num // 10 ** 9
            if num % 10 ** 9 == 0:
                return d1[num // 10 ** 9] +
                    ' Billion'
            return self.numberToWords(bi) +
                ' Billion ' +
                self.numberToWords(num % 10 ** 9)
a = input()
s = Solution()
print(s.numberToWords(int(a)))
```

从题目来看这应该是一道比较简单的题目才对，只需要利用表驱动进行简单地对应就可以进行转换了，为什么大家的复杂度如此之高呢？让我们来看一位同学的代码

```
str = int(input())

count = 0

answer = ""
while str >= 1000:
    part = ""
    left = str%1000
    lst = []
    str = int(str/1000)
    while left > 0:
        lefted = left % 10
        lst.insert(0, lefted)
        left = int(left / 10)
    if lst[0] == 1:
        part += "One Hundred "
    elif lst[0] == 2:
        part += "Two Hundred "
    elif lst[0] == 3:
        part += "Three Hundred "
    elif lst[0] == 4:
        part += "Four Hundred "
    elif lst[0] == 5:
        part += "Five Hundred "
    elif lst[0] == 6:
        part += "Six Hundred "
    elif lst[0] == 7:
        part += "Seven Hundred"
......
```

论文中我们只展示了代码的一小部分，但相信大家已经看明白了。这位同学一拿到题目就不考虑任何技巧，直接用 if else 来暴力解题以至于最终代码复杂度高达 865.6! 所以看来同学们的代码能力还是亟待提高，需要多多练习。

F. 代码执行流程

1. 先执行 $data_match.py$ 文件，将两份源文件得到的题目分析数据进行匹配整理
2. 再执行 $PCA.py$ 文件，利用 PCA 进行降维，并进行 $KMeans$ 聚类分析，得到题目难度分类结果和可视化图形，关于 k 值的选取，运用了 $inertia$（肘部法则）、$silhouette\_score$（轮廓系数）、$calinski\_harabasz\_score$（方差比准则）进行评估，选取最佳的 k 值

G. 参考文件

sklearn 文档
sklearn 中 PCA 参数的详细介绍
K-means 聚类分析参数介绍