Assignment #2: 深度学习与大语言模型

Updated 2204 GMT+8 Feb 25, 2025

2025 spring, Complied by <mark>同学的姓名、院系</mark>

姓名：李彦臻
学号：2300010821
学院：数学科学学院

作业的各项评分细则及对应的得分

标准	等级	得分
按时提交	完全按时提交：1 分 提交有请假说明：0.5 分 未提交：0 分	1 分
源码、耗时（可选）、解题思路（可选）	提交了 4 个或更多题目且包含所有必要信息：1 分 提交了 2 个或以上题目但不足 4 个：0.5 分 少于 2 个：0 分	1 分
AC 代码截图	提交了 4 个或更多题目且包含所有必要信息：1 分 提交了 2 个或以上题目但不足 4 个：0.5 分 少于：0 分	1 分
清晰头像、PDF 文件、MD/DOC 附件	包含清晰的 Canvas 头像、PDF 文件以及 MD 或 DOC 格式的附件：1 分 缺少上述三项中的任意一项：0.5 分 缺失两项或以上：0 分	1 分
学习总结和个人收获	提交了学习总结和个人收获：1 分 未提交学习总结或内容不详：0 分	1 分
总得分： 5	总分满分：5 分	

>
>
>
> **说明：**
>
> 1. **解题与记录：**
> - 对于每一个题目，请提供其解题思路（可选），并附上使用 Python 或 C++编写的源代码（确保已在 OpenJudge， Codeforces，LeetCode 等平台上获得 Accepted）。请将这些信息连同显示“Accepted”的截图一起填写到下方的作业模板中。（推荐使用 Typora https://typoraio.cn 进行编辑，当然你也可以选择 Word。）无论题目是否已通过，请标明每个题目大致花费的时间。
>
>2. **课程平台与提交安排：**
>
> - 我们的课程网站位于 Canvas 平台（https://pku.instructure.com ）。该平台将在第 2 周选课结束后正式启用。在平台启用前，请先完成作业并将作业妥善保存。待 Canvas 平台激活后，再上传你的作业。

>
> - 提交时，请首先上传 PDF 格式的文件，并将.md 或.doc 格式的文件作为附件上传至右侧的“作业评论”区。确保你的 Canvas 账户有一个清晰可见的头像，提交的文件为 PDF 格式，并且“作业评论”区包含上传的.md 或.doc 附件。
>
>3. **延迟提交：**
>
> - 如果你预计无法在截止日期前提交作业，请提前告知具体原因。这有助于我们了解情况并可能为你提供适当的延期或其他帮助。
>
>请按照上述指导认真准备和提交作业，以保证顺利完成课程要求。

1. 题目

18161: 矩阵运算

matrices, http://cs101.openjudge.cn/practice/18161

思路：做法很直接，先判断是否能够进行矩阵运算：只需要验证第一个矩阵的列数是否等于第二个矩阵的行数、以及第一个矩阵的行数是否等于第三个矩阵的行数、以及第一个矩阵的列数是否等于第三个矩阵的列数即可；
随后，按部就班的相乘和相加就行了~

代码：

```python
```
```
def calc(A, B, C, x1, x2, y2):

    D = [] #A 与 B 的乘积
    for i in range(x1):
        row = [] #A 与 B 乘积的第 i 行
        for j in range(y2):
            num = sum(A[i][k]*B[k][j] for k in range(x2)) #第 i 行第 j 个数
            row.append(num)
        D.append(row)

    E = [] #D 与 C 的和
    for i in range(x1):
        row = [] #D 与 C 和的第 i 行
```

```
        for j in range(y2):
            num = D[i][j] + C[i][j] #第 i 行第 j 个数
            row.append(num)
        E.append(row)

    return E

x1, y1 = map(int,input().split())
A = []
for i in range(x1):
    A.append(list(map(int,input().split())))
x2, y2 = map(int,input().split())
B = []
for i in range(x2):
    B.append(list(map(int,input().split())))
x3, y3 = map(int,input().split())
C = []
for i in range(x3):
    C.append(list(map(int,input().split())))
if x2 != y1 or x1 != x3 or y2 != y3:
    print("Error!")
else:
    Z = calc(A, B, C, x1, x2, y2)
    for row in Z:
        print(" ".join(map(str,row)))
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

19942: 二维矩阵上的卷积运算

matrices, http://cs101.openjudge.cn/practice/19942/

思路：先定义一个函数 calc(a, b)，用于计算“以原矩阵中第 a 行第 b 列个数为左上角的 p 乘 q 矩阵”与卷积矩阵得出的卷积结果，为了简化接下来的计算过程；
随后，注意到结果矩阵有 m-p+1 行、n-q+1 列，因此挨个儿计算出原矩阵中左上角的 m-p+1 行 n-q+1 列的数对应出的卷积结果，并将其填入结果矩阵即可。

代码：

```python
```
```
m, n, p, q = map(int,input().split())
mat = [] # 原矩阵
for i in range(m):
    row = list(map(int,input().split()))
    mat.append(row)
con = [] # 卷积矩阵
for i in range(p):
    row = list(map(int,input().split()))
    con.append(row)

def calc(a, b): # 从第 a 行第 b 列个数开始的卷积结果
    num = 0
    for i in range(p):
        row = sum(mat[a + i][b + j]*con[i][j] for j in range(q)) # 卷积计算结果的第 i 行
        num += row
    return num

finale = [] # 结果矩阵，有 m-p+1 行，n-q+1 列
for i in range(m - p + 1):
    row = [] # 结果矩阵的第 i 行，依次从第 i 行第 j 个数开始
    for j in range(n - q + 1):
        row.append(calc(i, j))
```

```
    finale.append(row)

for row in finale: # 把结果矩阵打印出来
    print(" ".join(map(str,row)))
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

04140: 方程求解

牛顿迭代法，http://cs101.openjudge.cn/practice/04140/

请用<mark>牛顿迭代法</mark>实现。

因为大语言模型的训练过程中涉及到了梯度下降（或其变种，如 SGD、Adam 等），用于优化模型参数以最小化损失函数。两种方法都是通过迭代的方式逐步接近最优解。每一次迭代都基于当前点的局部信息调整参数，试图找到一个比当前点更优的新点。理解牛顿迭代法有助于深入理解基于梯度的优化算法的工作原理，特别是它们如何利用导数信息进行决策。

> **牛顿迭代法**
>
> - **目的**：主要用于寻找一个函数 $f(x)$ 的根，即找到满足 $f(x)=0$ 的 x 值。不过，通过适当变换目标函数，它也可以用于寻找函数的极值。
> - **方法基础**：利用泰勒级数的一阶和二阶项来近似目标函数，在每次迭代中使用目标函数及其导数的信息来计算下一步的方向和步长。
> - **迭代公式**：$ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} $ 对于求极值问题，这可以

转化为$ x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} $，这里 $f'(x)$ 和 $f''(x)$ 分别是目标函数的一阶导数和二阶导数。
> - **特点**：牛顿法通常具有更快的收敛速度（尤其是对于二次可微函数），但是需要计算目标函数的二阶导数（Hessian 矩阵在多维情况下），并且对初始点的选择较为敏感。
>
> **梯度下降法**
>
> - **目的**：直接用于寻找函数的最小值（也可以通过取负寻找最大值），尤其在机器学习领域应用广泛。
> - **方法基础**：仅依赖于目标函数的一阶导数信息（即梯度），沿着梯度的反方向移动以达到减少函数值的目的。
> - **迭代公式**：$ x_{n+1} = x_n - \alpha \cdot \nabla f(x_n) $ 这里 α 是学习率，$\nabla f(x_n)$ 表示目标函数在 x_n 点的梯度。
> - **特点**：梯度下降不需要计算复杂的二阶导数，因此在高维空间中相对容易实现。然而，它的收敛速度通常较慢，特别是当目标函数的等高线呈现出椭圆而非圆形时（即存在条件数大的情况）。
>
> **相同与不同**
>
> - **相同点**：两者都可用于优化问题，试图找到函数的极小值点；都需要目标函数至少一阶可导。
> - **不同点**：
> - 牛顿法使用了更多的局部信息（即二阶导数），因此理论上收敛速度更快，但在实际应用中可能会遇到计算成本高、难以处理大规模数据集等问题。
> - 梯度下降则更为简单，易于实现，特别是在高维空间中，但由于只使用了一阶导数信息，其收敛速度可能较慢，尤其是在接近极值点时。
>

代码：

```python
```

```
def fun(x):
    return x**3 - 5*x**2 + 10*x - 80

def dif_fun(x):
    return 3*x**2 - 10*x + 10

# 由于 f(5)=-30，f(6)=16，因此根在 5~6 之间；故可以选取 5 作为初始点
queue = [5]
while True:
    x = queue[-1]
    y = x - fun(x) / dif_fun(x) # x 在牛顿迭代法里的下一项
    queue.append(y)
```

```
    if abs(y - x) < 10**(-10): # 当误差已经小于10^10时，可以结束
        break
    else:continue

a = queue[-1] # 最终解
x = "{:.9f}".format(a) # 用format函数保留小数点后九位（记住！）
print(x)
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

06640: 倒排索引

data structures, http://cs101.openjudge.cn/practice/06640/

思路：想法很朴素，就是对于每个需要检查的单词，直接挨个检查这个单词是否在每个列表（文档）里，如果在某个列表里那么就把这个文档的编号记录下来即可~
（p.s.标准答案的方法更好，已经学习并理解了~）

代码：

```python

```

```
n = int(input())
all = [] # 包含所有单词的列表
for i in range(n):
    words = list(map(str,input().split()))
    all.append(words)

m = int(input())
for i in range(m):
    word = input()
    yes = [] # 所有包含这个单词的文档编号
    for j in range(n):
        if word in all[j]: # 挨个儿检查这个单词是否在每个文档里
            yes.append(j + 1)
    if yes: # 如果这本书里确实有这个单词
        print(" ".join(map(str,yes))) # 把 yes 列表打印出来
    else:
        print("NOT FOUND")
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

04093: 倒排索引查询

data structures, http://cs101.openjudge.cn/practice/04093/

思路：对于每一次查询，先找出所有的 1，并对这些需要被包含的单词所对应的文档取一个共同的交集，记为 a；然后再找出所有的-1，并对所有这些不能被包含的单词的文档取一个并集，记为 b；最后 a-b 的差集即可满足题意！

代码：

```python

```
```
n = int(input())
files = [] # 储存所有单词的文档归属数据
for i in range(n):
    data = list(map(int,input().split()))
    files.append(set(data[1:])) # 将每个单词对应的文档数据转化为集合，方便后续的操作

m = int(input())
for i in range(m):

    data = list(map(int,input().split()))
    t = data.index(1) # 第一个 1 要特殊处理，作为初始值
    a = set(files[t]) # 所有有可能合题的文档（同样要储存在集合内）
    for j in range(t + 1,n):
        if data[j] == 1:
            a = a & files[j] # 取交集（集合之间的∩、-、∪运算时间复杂度极低！）

    b = set() # 所有一定不合题的文档（同样要储存在集合内）
    for j in range(n):
        if data[j] == -1:
            b = b | files[j] # 取并集

    # 注意：一定要先把所有要包含的文档全部放入 a，所有不能包含的文档统一放入 b，最后再 a-b！
    # 否则会损失一些 b 中的信息，例如如果是“无所谓、不能包含 2、要包含 12”，
    # 如果一步一步使用交集和差集的话，则 1 和 2 都合题，这与第二个条件矛盾！因此不能一步一步来~
    c = a - b # a 与 b 的差集
    finale = list(c)
    if finale:
        finale.sort()
```

```
        print(" ".join(map(str,finale)))
    else:print("NOT FOUND")
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

Q6. Neural Network 实现鸢尾花卉数据分类

在 http://clab.pku.edu.cn 云端虚拟机，用 Neural Network 实现鸢尾花卉数据分类。

参考链接，https://github.com/GMyhf/2025spring-cs201/blob/main/LLM/iris_neural_network.md

2. 学习总结和个人收获

<mark>如果发现作业题目相对简单，有否寻找额外的练习题目，如“数算 2025spring 每日选做”、LeetCode、Codeforces、洛谷等网站上的题目。</mark>

对于这次作业，第一道题很常规也很直接，直接利用矩阵的运算规则对其进行计算即可，是一道很简单的题目（如果没记错的话，上学期的计概作业中应该布置过这一道习题哈哈）；

第二题的思路也比较直接，只需要定义一个函数 calc(a, b)用于计算“以原矩阵中第 a 行第 b 列个数为左上角的 p 乘 q 矩阵”与卷积矩阵得出的卷积结果；而又因为结果矩阵有 m-p+1 行、n-q+1 列，因此只需要挨个儿计算出原矩阵中左上角的 m-p+1 行 n-q+1 列的数对应出的卷积结果即可；可以说，前两题的难度相对较小。

第三题则是一道很有趣的题目，里面提到的牛顿迭代法我还是第一次学到。我觉得这个方法不仅非常的有效，而且里面蕴含的数学原理也十分直观，可以说是一个非常精妙的方法。在做这道题的过程中，我们只需要观察到根在 5~6 这个范围之内，然后把初始值设为 5，一步一步迭代即可。这个题目的主要难点在于该如何设定结束的法则，我也是在这个地方犹豫了一会。经过思考后，我认为只需控制“误差”在 10 的-10 次方内，应该就一定能够满足精确度到后 9 位小数，因此我就把上一项与下一项的差的绝对值小于 10 的-10 次方作为了能够结束迭代的判断法则；最终也是用此方法成功的 ac 了。

不仅如此，在这道题里面，我还学习到了保留若干位小数的方法（如下图）：

```
a = queue[-1] # 最终解
x = "{:.9f}".format(a)
print(x)
```

（虽然在计概里学过，但现在已经完全忘了；上一次学的时候也只是背了一下代码，这次重新学的时候还仔细思考了一下为什么要这么写，理解了一些新的内涵；也算是温故而知新了吧~

第四题的话则颇为简单，用常规的思路也能很快 ac；但是我觉得如果这道题只用我那个方法简单的 ac 了然后不接着思考了，感觉就没有什么收获，肯定还有值得挖掘的东西。于是我打开 gpt，问了一下 AI 是怎么做的，然后他给我的那个方法就是标准答案的方法，我觉得那个方法相比于我之前的方法要有用的多（就是用字典来储存每个字母所对应到的文档）；这个方法的时间复杂度比我之前所使用的那个普通方法要低得多，我觉得在做一些更加困难的问题的时候会非常的有效；通过做这道题，我也算是学习到了一个新的思路，蛮有收获。

第五题的思路看似不难想到，但是实际做起来时却花了我不少时间。首先，这道题目显然涉及到了集合的交集、并集与差集的运算，因此我就问了一下 gpt 该使用什么方法才能够让这些运算的时间复杂度尽量降低；然后，他就给了我以下内容：

如何快速的取两堆数字的交集

在 Python 中，最快速的方法是使用 **集合（set）的交集运算**，因为 `set` 使用哈希表，求交集的时间复杂度接近 **O(min(len(A), len(B)))**，远比 `list` 逐个遍历快。

方法 1：使用 `set` 交集运算 `&`

如果两堆数字存储在**列表（list）**或集合（set）**中，可以使用 `&` 运算符：

python 复制 编辑

```python
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

intersection = A & B  # 交集
print(intersection)  # 输出 {4, 5}
```

✅ **时间复杂度**：O(min(len(A), len(B)))
✅ **适用于**：当 `A` 和 `B` 都是 **集合（set）** 时。

包含在a中且不包含在b中的数该怎么求呢

如果想要找出**在 `A` 中但不在 `B` 中的数**，可以使用**集合的差集运算**（`-` 或 `set.difference()`），这在 Python 中非常高效，时间复杂度为 **O(N)**（其中 `N` 是 `A` 的大小）。

方法 1：使用 `-` 计算差集

python 复制 编辑

```python
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

difference = A - B  # 计算 A 中但不在 B 中的元素
print(difference)  # 输出 {1, 2, 3}
```

✅ **时间复杂度**：O(len(A))
✅ **适用于**：当 `A` 和 `B` 都是 `set` 时，速度最快！

同上，如何取并集呢

在 Python 中，**并集**（Union）表示取**两个集合中的所有元素（去重）**。可以使用 `|` 运算符或者 `set.union()` 方法，高效地求并集，时间复杂度为 **O(N + M)**（其中 `N` 和 `M` 分别是 `A` 和 `B` 的大小）。

方法 1：使用 `|` 运算符

python 复制 编辑

```python
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

union_set = A | B  # 计算 A 和 B 的并集
print(union_set)  # 输出 {1, 2, 3, 4, 5, 6, 7, 8}
```

✅ **时间复杂度**：O(N + M)
✅ **适用于**：当 `A` 和 `B` 都是 `set` 时，最快！

看完了这些内容之后，我感觉大有收获！因为学会了这些集合之间的运算之后，这一类题目的方法的时间复杂度就会大幅降低，可以说是非常有效的。

除了学习这些运算法则以外，我还在这道题目的另外一个地方卡了很久，就是：这道题我一开始使用的方法是，每录入一个数据，就对当下所有可能的文档的集合进行一次运算。比如如果出现一次 1，那么就对当下这个文档所对应的集合与原集合进行一次交集运算；如果出现了一个-1，就反过来，进行一次差集运算。这个思路看起来很合理，但我仔细思考之后，才发现原来有很大的漏洞：问题在于，如果每录入一个数据就进行一次对应的运算的话，那么很有可能会损失掉一些重要的“信息”，导致一些本来应该被剔除掉的文档没有被剔除出去。比如，如果一开始第一个判断是无所谓（0），第二个判断是不要（-1），那么因为“无所谓”是不会对原本的所有可能合题的集合（目前暂时是空集）进行任何处理的，而对于负一而言，因为原集合是空，所以负一也不会有任何处理；这就相当于是白白损失掉了这个负一的信息，显然会导致 wa。

因此，在花了很长时间终于发现了这个错误之后，我赶紧修改了代码，采用了上述提到的“先把所有的 1 包含的单词对应到的文档取一个共同的交集，记为 A；再把所有的-1 对应到的文档取一个共同的并集，记为 B；最后取 A 和 B 的差集即可”的方法，才最终 AC，还是花了不少时间；可以说，这道题目如果陷入了这个陷阱中，还是蛮难检查出来的，要花一些时间仔细研究才能发现这个小漏洞。。

总的来讲，在这次作业中，并没有什么本质上特别困难的题目，他们的思想和方法基本上看了几分钟大致都能想到，但是在做这几道题的过程中涉及到的很多细节促使我去回顾并复习了很多内容，也让我去了解了很多新的方法和思想（如牛顿迭代法、倒排表等等），还是蛮有收获的；希望自己能够继续加油！