STAT243 Lecture 8.3 Calculating with Integers vs. Floating Points

> **Logic**
>
> 关于 Calculating with integers vs. floating points 的详细描述, 见 DDA3005 Lecture 3

1 计算机算术 ≠ 数学算术 (Computer Arithmetic Is Not Mathematical Arithmetic)

在数学中，实数的加、减、乘、除是**封闭的运算**，但计算机中的算术却不是。
由于数值在内存中只能被近似表示，计算机运算可能：

- 溢出 (overflow)：结果超出可表示范围
- 下溢 (underflow)：结果太小而被当作 0
- 违反代数性质，例如结合律与分配律

例如：

Python

```python
val1 = 1/10; val2 = 0.31; val3 = 0.57
res1 = val1 * val2 * val3
res2 = val3 * val2 * val1
res1 == res2
# False

dg(res1)
# 0.01766999999999999821
dg(res2)
# 0.01767000000000000168
```

结果可能不同，因为浮点数运算存在微小的舍入误差，导致 $(a+b)+c \neq a+(b+c)$ 或 $a(b+c) \neq ab+ac$。

---

2 整数与浮点运算的差异 (Integer vs Floating-Point Calculations)

- **整数计算**：
  精确且快速（但可能溢出）。
  在 Python 的 `int` 类型中不会溢出，但在其他语言或 `numpy` 的定长整数中可能出现溢出。
- **浮点数计算**：
  近似且较慢。
  每个浮点运算 (addition/multiplication) 称为一次 **flop (floating point operation)**。
  计算强度常用 flops 数量衡量。

> **Remark**
>
> 在线性代数中（如矩阵乘法、LU 分解等），flops 是衡量算法复杂度的主要单位。

---

3 浮点比较问题 (Comparisons)

在计算机中，**不要轻易使用 == 比较浮点数**，除非满足以下条件：

1. `x` 和 `y` 都是整数；
2. `x` 和 `y` 是整数值，但以 double 存储且足够小以精确表示；
3. 两个数是通过**相同的计算方式**得到的，例如：

Python

```python
0.4 - 0.3 == 0.4 - 0.3   # True
0.4 - 0.3 == 0.1         # False
```

3.1 示例：浮点数比较陷阱

Python

```python
4 - 3 == 1          # True
4.0 - 3.0 == 1.0    # True
4.1 - 3.1 == 1.0    # False!
```

3.2 使用机器精度判断“近似相等”

一个更可靠的判断方式是：
如果两个数之间的**相对差距小于机器精度 ε**，则认为它们“近似相等”。

Python

```python
def approx_equal(a, b):
    if abs(a - b) < np.finfo(np.float64).eps * abs(a + b):
        print("approximately equal")
    else:
        print("not equal")

approx_equal(1234567812345678, 1234567812345677)
# not equal
```

通常可使用比 ε 略大的常数（例如 10ε）以增强稳健性。

3.3 缺失值检测的注意事项

数据中常用特殊整数（如 `-9999`）代表缺失值：

Python

```python
x[x == -9999] = np.nan
```

这种比较是安全的，因为整数 `-9999` 可被浮点数精确表示。
但更稳妥的做法是：在读入时使用整数或字符串类型判断，再转为浮点数。

---

4 常见数值误差 (Common Numerical Errors)

4.1 大数相减的灾难性抵消 (Catastrophic Cancellation)

当两个接近的数相减时，大部分有效位会被抵消，剩余的位数全是舍入误差。

Python

```python
dg(123456781234.56 - 123456781234.00)
# 0.55999755859375000000
```

误差估计：

$$\epsilon x = 2.2 \times 10^{-16} \times 10^{11} \approx 10^{-5}$$

因此结果仅保留约 5 位有效数字。

示例：小数的灾难性抵消

Python

```python
x = .00000000000000123412341234
y = .00000000000000123412340000
dg(x - y, '.35f')
# 0.00000000000000000000000123399999315140
```

理论值：0.0000000000000000000001234
实际结果略有偏差，仅保留约 28 位有效数字（而非理想的 36 位）。

> ⚠ **Remark: 解决思路**
>
> 避免直接相减两个近似相等的大数，可通过数值重构或代数变换降低误差。

4.2 计算方差的示例 (Variance Computation)

计算方差的两种写法：

Python

```python
x = np.array([-1.0, 0.0, 1.0])
n = len(x)
np.sum(x**2) - n*np.mean(x)**2          # 方法一
# np.float64(2.0)
np.sum((x - np.mean(x))**2)             # 方法二
# np.float64(2.0)
```

当数据加上一个大常数：

Python

```python
x = x + 1e8
np.sum(x**2) - n*np.mean(x)**2          # 错误!
# np.float64(0.0)
np.sum((x - np.mean(x))**2)             # 正确
# np.float64(2.0)
```

结论：

- 第一种写法会发生灾难性抵消（因 $\bar{x}^2$ 与 $\sum x_i^2$ 数值巨大且相近）
- 第二种写法稳定得多

经验法则：

- 在运算中**先减去均值**或相似数量级的值，能显著减少误差。

4.3 数量级差异太大的加减法

当两个数差距过大时，小数部分的贡献可能被完全舍入掉。

Python

```python
dg(123456781234.2)
# 123456781234.19999694824218750000

dg(123456781234.2 - 0.1)     # truth: 123456781234.1
# 123456781234.09999084472656250000

dg(123456781234.2 - 0.000001)   # truth: 123456781234.199999
# 123456781234.19999694824218750000

123456781234.2 - 0.000001 == 123456781234.2
# True
```

此时较小的数已被“吞没”，因为：

$$x = 10^{11}, \quad \epsilon x \approx 10^{-5}$$

任何小于 10^{-5} 的变化都不会影响结果。

改进策略：

- 若要对大量不同数量级的数求和，**先排序再相加（从小到大）**
- 或采用**树状求和 (pairwise summation)**，使每次加法在相近数量级间进行。

---

5 对数运算的稳定性 (Log-Scale Computations)

当涉及大量数的连乘或连除时，直接计算容易溢出或下溢。
解决方案：**在对数尺度上进行计算**。

例如：

$$\frac{\prod_i x_i}{\prod_j y_j} = \exp\left(\sum_i \log x_i - \sum_j \log y_j\right)$$

多数情况下我们甚至无需取指数（如最大化对数似然时）。

5.1 log-sum-exp 技巧

问题背景

我们经常需要计算如下形式的表达式：

$$\log\left(\sum_{i=1}^{n} e^{x_i}\right)$$

这是在机器学习、统计建模中极常见的一类操作，比如：

- Softmax 函数
- 对数似然（log-likelihood）
- Bayesian predictive densities
- Partition function in probabilistic models

问题在于：

- 如果某个 x_i 很大（如 $x_i = 1000$），那么 e^{x_i} 会导致溢出 (overflow)；
- 如果所有 x_i 都很小（如 $x_i = -1000$），那么 e^{x_i} 会接近 0，导致下溢 (underflow)。

基本思路

我们利用一个简单的代数恒等式来“拉回尺度”：

$$\log\left(\sum_i e^{x_i}\right) = c + \log\left(\sum_i e^{x_i - c}\right)$$

其中

$$c = \max_i x_i$$

常见应用场景

1. **Softmax 函数**

   $$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

   实现时通常写作：

   Python

   ```python
   def softmax(x):
       c = np.max(x)
       exp_x = np.exp(x - c)
       return exp_x / np.sum(exp_x)
   ```

2. **对数似然 (Log-Likelihood)**
   例如在分类任务中计算：

   $$\log p_j = x_j - \log\left(\sum_k e^{x_k}\right)$$

   这里的第二项正是 log-sum-exp。
3. **数值积分 / 预测密度**
   在计算

   $$\log \frac{1}{m} \sum_{j=1}^{m} e^{v_j}$$

   时使用：

   $$\log \frac{1}{m} \sum_{j=1}^{m} e^{v_j} = -\log m + c + \log \sum_j e^{v_j - c}$$

延伸：稳定实现

Python 中的 NumPy 和 SciPy 已经内置了高效稳定的实现：

Python

```python
from scipy.special import logsumexp

logsumexp(x)
```

它自动执行减去最大值的操作，并支持多维数组及权重选项。

5.2 实例 1：多分类逻辑回归 (Multiclass Logistic Regression)

$$p_j = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}$$

若 z_k 很大或很小，会导致数值溢出。
可通过以下变形稳定计算：

$$p_j = \frac{\exp(z_j - c)}{\sum_k \exp(z_k - c)}, \quad c = \max_k z_k$$

该技巧即为 **log-sum-exp trick** 的核心思想。

5.3 实例 2：预测密度计算 (Predictive Density in Bayesian Models)

$$\begin{aligned} f(y^*|y,x) &= \int f(y^*|y,x,\theta)\pi(\theta|y,x)d\theta \\ &\approx \frac{1}{m}\sum_{j=1}^{m}\prod_{i=1}^{n} f(y_i^*|x,\theta_j) \\ &= \frac{1}{m}\sum_{j=1}^{m}\exp\sum_{i=1}^{n}\log f(y_i^*|x,\theta_j) \\ &\equiv \frac{1}{m}\sum_{j=1}^{m}\exp(v_j) \end{aligned}$$

若 v_j 很小（如 -1000），直接取 $\exp(v_j)$ 将导致下溢。
正确做法：

$$\log f(y^*|y,x) \approx \log\frac{1}{m}\sum_j \exp(v_j) = c + \log\frac{1}{m}\sum_j \exp(v_j - c)$$

其中 $c = \max_j v_j$。
这同样利用了 log-sum-exp 技巧，避免了浮点下溢。

---

6 数值线性代数中的误差 (Numerical Errors in Linear Algebra)

某些矩阵在数学上是正定的，但数值上可能出现**负特征值**。
例如平方指数核矩阵：

Python

```python
xs = np.arange(100)
dists = np.abs(xs[:, np.newaxis] - xs)
corr_matrix = np.exp(-(dists/10)**2)
scipy.linalg.eigvals(corr_matrix)[80:99]
```

虽然该相关矩阵在理论上正定，但由于舍入误差，部分特征值可能略为负数。
在高维协方差估计、核方法中，这类现象极为常见。