第五次作业总结

- 共性问题
  - 第6问理解有误，不是返回长度恰好等于seq_len的句子的tokens，而应该是返回所有句子的tokens并利用trim函数将编码长度限制为seq_len
  - 题意理解错误，部分同学直接添加"PAD”到向量
  - encode_all方法可以用encode和trim组合实现
  - 字典不保证顺序(python3.5之前)
  - 有同学没有写类，直接写的函数，或者把构建字典的部分写到类外的函数

School of Economics ang Management, Beihang University

现代程序设计技术

赵吉昌
jichang@buaa.edu.cn

本周内容

- 面向对象编程
  - 代理模式(Proxy)
  - 装饰器

代理模式

- Proxy Pattern
  - 在访问某个对象之前执行一个或多个重要的额外操作
  - 访问敏感信息或关键功能前需要具备足够的权限
  - 将计算成本较高的对象的创建过程延迟到用户首次真正使用时才进行
    - 惰性求值

代理模式

- 常见类型
  - 远程代理：实际存在于不同地址空间（如网络服务器）的对象在本地的代理者
    - 对使用者透明
  - 虚拟代理：用于惰性求值，将一个大计算量对象的创建延迟到真正需要的时候进行
  - 保护/防护代理：控制对敏感对象的访问
  - 智能（引用）代理：在对象被访问时执行额外的动作，如引用计数或线程安全检查等
  - `Demo: pdp.py`

装饰器

- 函数式编程(Functional Programming)
  - 面向过程，但更接近于数学计算
  - 一种抽象程度很高的编程范式
  - 允许将函数作为参数传入另一个函数
  - 允许返回另一个函数
  - python支持部分的函数式编程

装饰器

- 高阶函数
  - 接收另一个函数作为参数的函数
    - `list(map(f,[x1, x2, x3, x4]))=[f(x1),f(x2),f(x3),f(x4)]`
    - `reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)`
    - `list(filter(f,[x1,x2,x3,x4]))=[x for x in [x1,x2,x3,x4] if f(x)]`
  - `Demo: fp.py`

装饰器

- 返回函数
  - def lazy_sum(*args):
  - def sum():
  - ax = 0
  - for n in args:
  - ax = ax + n
  - return ax
  - return sum
  - f = lazy_sum(1, 3, 5, 7, 9)
  - f()
  - 注意对“惰性”的理解

装饰器

- 偏函数
  - 为了简化多参数函数的调用，可通过固定某参数来返回新函数，以实现简化调用

```
def growth(step=1,limit=200):
  g=0
  while(True):
        if g+step>limit:
              break
        g+=step
  return g
print(growth())
print(growth(step=3))
growth3=functools.partial(growth,step=3)
print(growth3())
print(growth3(limit=300))
```

装饰器

- 闭包
  - 返回函数不宜引用任何循环变量，或者后续会发生变化的变量

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i
        fs.append(f)
    return fs
f1, f2, f3 = count()
print(f1())
print(f2())
print(f3())
```

Demo: fp.py

```
def count():
    def f(j):
        def g():
            return j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i))
    return fs
f1,f2,f3=count()
print(f1())
print(f2())
print(f3())
# f(i)立刻被执行，因此i的当前值被传入f
```

装饰器

- 装饰器(Decorator)
  - 在不修改原始代码的前提一下增强或扩展既有功能
    - 在核心功能的基础上增加额外的功能如
      - 授权(Authorization)
      - 日志(Logging)

```
def log(func):
  def wrapper(*args,**kwargs):
        print("call "+func.__name__)#额外的功能
        return func(*args,**kwargs) #调用原功能
  return wrapper
@log #使用装饰器
def now():
    pass
now()#相当于log(now)()
```

装饰器

- 装饰器
  - 如果装饰器本身需要参数，则需要通过高阶函数实现

```
def log(text):
  def decorator(func):#高阶函数
        def wrapper(*args,**kwargs):
              print(text+' '+'call '+func.__name__)
              return func(*args,**kwargs)
        return wrapper
  return decorator
@log('日志：')
def now():
  print(time.strftime('%Y-%m-%d',time.localtime(time.time())))
now()#相当于log('日志：')(now)()
```

装饰器

- 装饰器
  - 通过装饰后函数的名称发生了变化
    - 如前例`now.__name__`变成了`wrapper`
  - 如何保持函数名称不发生变化？
    - `functools.wraps(func)`
    - `@wraps`复制了函数名称、注释文档、参数列表等，使得能够在装饰器里访问在装饰之前的函数属性
  - 在实现装饰器时应在`wrapper`函数前加入`@wraps`
    - 避免因函数内部功能逻辑对函数属性的依赖而导致功能错误

装饰器

- 装饰器类

```
class Log:
  def __init__(self,logfile='out.log'):
        self.logfile=logfile
  def __call__(self,func):
        @wraps(func)
        def wrapper(*args,**kwargs):
              info="INFO: "+func.__name__+" was called"
              with open(self.logfile,'a') as file:
                    file.write(info+'\n')
              return func(*args,**kwargs)
        return wrapper
@Log('test.log')#@Log()
def myfunc():
  pass
myfunc()#相当于Log('test.log')(myfunc)()
```

装饰器

- 装饰器的顺序
  - 装饰顺序：就近原则
    - 从下往上装饰
  - 调用顺序：就远原则
    - 从上往下调用
  - `@a`
  - `@b`
  - `@c`
  - `def f ():`
  - `    pass`
  - 相当于`f=a(b(c(f)))`

装饰器

- property
  - 使实例方法的使用如同实例属性
  - @property 读取属性
  - @方法名.setter 修改属性
  - @方法名.deleter 删除属性
    - Demo: fp.py
  - 也可通过property()方法
    - 获取，设置，删除，描述文档
    - id = property(get_id, set_id, del_id, 'id is ...')

装饰器

- 类方法与静态方法
  - 实例方法需要通过`self`参数隐式的传递当前类对象的实例
  - 用`@classmethod`修饰的方法需要通过`cls`参数传递当前类对象，称为类方法
  - 用`@staticmethod`修饰的方法定义与普通函数一样，称为静态方法
  - 类方法和静态方法均可通过类对象或实例对象调用

装假器

- 类方法的使用场景
  - 常作为工厂方法创建实例对象(fp.py)
- 静态方法的使用场景
  - 方法不需要访问任何实例方法和属性，仅需通过传入参数就可返回结果
  - 节省了实例化对象的开销成本
  - 等价于类外的普通函数，但可能仅为该类服务，因此搁进类中
  - 往往写在类外可能更合适

本周作业

- 当进行大批量数据读取、模型训练等，往往需要花费大量时间。这类耗时长的程序有一些通用的功能需求，可以通过装饰器实现，具体如下：
- 1. 实现一个类，在其中提供一些方法模拟耗时耗内存的一些操作，以测试如下的装饰器（用类或函数实现），如大的数据结构生成、遍历、写入文件序列化等。
- 2. 如果需要知道程序的运行时间、运行进度、内存占用情况，请利用line_profiler、memory_profiler、tqdm等装饰器实现相关功能，要求在程序执行结束后，打印程序的内存占用和运行时间。
- 3. 在程序处理结束后，通常需要将模型或者数据处理结果保存下来。但是，有时会因为路径设置错误（忘记新建文件夹）等原因导致文件无法存储，浪费大量的时间重复运行程序。一种解决方法是在执行程序前对参数中的路径进行检查。要求利用装饰器函数实现这一功能，接收函数的路径参数，检查路径对应文件夹是否存在，若不存在，则给出提示，并在提示后由系统自动创建对应的文件夹。
- 4. 在程序运行结束后，可以给用户发送一个通知，比如播放一段音乐等。要求实现对应的装饰器类，在被装饰的函数执行结束后，可以主动播放声音（了解并使用一下playsound或其他声音文件处理的库）。

SI: 装饰器

- 代码的内存占用分析
  - memory_profiler库
  - 通过装饰器实现
  - from memory_profiler import profile
  - @profile
  - def my_func():
  - a = [1] * (10 ** 6)
  - b = [2] * (2 * 10 ** 7)
  - del b
  - return a
  - my_func()
  - Demo: md.py

SI: 装饰器

- 代码的执行时长分析
  - line_profiler库
  - 通过装饰器实现
  - import time
  - @profile
  - def test_time():
  - for i in range(100):
  - a=[1]*(10**8)
  - b=[2]*(10*6)
  - pass
  - test_time()
  - kernprof -lv ld.py

SI: 代码运行频率的分析

- heartrate
  - https://github.com/alexmojaki/heartrate/tree/master/heartrate
  - `Demo: hd.py`
- pyheat
  - https://github.com/csurfer/pyheat
  - `Demo: heat.py`

SI: 代码执行进度的可视化

- tqdm包
  - Demo: td.py