说明

- 对于所有程序，是在python3.8实现的，都可以运行
- 如果提示缺少对应的第三方库，那么需要你先去pip3 install一下，可以加入网上一些下载比较快的镜像源。我建议是在配置pycharm里面的python解释器时，直接配置Anaconda的python环境，里面很多必要的库都有，比如sklearn机器学习库等，就可以不用去自行pip安装了，很方便
- 代码在复制的时候，请一定要注意缩进问题

规划

线性规划

scipy求解

- 需要知道目标函数(一般是求最大或者最小值)和约束条件

  求解前转化为下面的标准形式

$$\min c^T x$$
$$s.t.\begin{cases} Ax \le b \\ Aeq * x = beq \\ lb \le x \le ub \end{cases}$$

```python
from scipy import optimize
import numpy as np

# 求解函数
res = optimize.linprog(c, A, b, Aeq, beq, LB, UB, X0, OPTIONS)
# 目标函数最小值
print(res.fun)
# 最优解
print(res.x)
```

- 标准形式是<=，如果是>=，则在两边加上符号-

举例1

样例1：　求解下列线性规划问题

$$\max z = 2x_1 + 3x_2 - 5x_3$$

$$s.t. \begin{cases} x_1 + x_2 + x_3 = 7 \\ 2x_1 - 5x_2 + x_3 \geq 10 \\ x_1 + 3x_2 + x_3 \leq 12 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

- 使用scipy求解z的最大值
  - c是目标函数的系数矩阵
  - A是化成标准的<=式子的左边的系数矩阵
  - B是化成标准的<=式子的右边的数值矩阵
  - Aeq是所有=左边的系数矩阵，记得里面是[[]]二维
  - Beq是所有=右边的数值矩阵
  - 下面第11行-c加-，是因为此题求的是最大值，但是标准格式是求最小值，所以加负号
  - 另外上面的3个变量都大于0，这里可以使用`bounds=(0, None)`,bounds=(min,max)是范围，None代表无穷，如果不写bounds，那默认是(0，None)

    ```
    res = optimize.linprog(-c, A, B, Aeq, Beq, bounds=(0, None))
    ```

```
1  from scipy import optimize
2  import numpy as np
3  
4  # 确定c,A,b,Aeq,beq
5  c = np.array([2, 3, -5])
6  A = np.array([[-2, 5, -1], [1, 3, 1]])
7  B = np.array([-10, 12])
8  Aeq = np.array([[1, 1, 1]])
9  Beq = np.array([7])
10 # 求解
11 res = optimize.linprog(-c, A, B, Aeq, Beq)
12 print(res)
13 
```

  - res

```
     con: array([1.80713222e-09])
     fun: -14.57142856564506
 message: 'Optimization terminated successfully.'
     nit: 5
   slack: array([-2.24583019e-10,  3.85714286e+00])
  status: 0
 success: True
       x: array([6.42857143e+00, 5.71428571e-01, 2.35900788e-10])

Process finished with exit code 0
```

- fun是目标函数最小值
- x是最优解，即上面的x1,x2,x3的最优解

举例2

求解下列线性规划问题

$$\max z = 2x_1 + 3x_2 + x_3$$

$$s.t.\begin{cases} x_1 + 2x_2 + 4x_3 = 101 \\ x_1 + 4x_2 + 2x_3 \geq 8 \\ 3x_1 + 2x_2 \geq 6 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

```python
from scipy import optimize
import numpy as np

c = np.array([2, 3, 1])
A = np.array([[-1, -4, -2], [-3, -2, 0]])
B = np.array([-8, -6])
Aeq = np.array([[1, 2, 4]])
Beq = np.array([101])
# 求解
res = optimize.linprog(-c, A, B, Aeq, Beq)
print(res)

```

```
con: array([3.75850107e-09])
     fun: -201.9999999893402
 message: 'Optimization terminated successfully.'
     nit: 6
   slack: array([ 93.        , 296.99999998])
  status: 0
 success: True
       x: array([1.01000000e+02, 6.13324051e-10, 3.61350245e-10])

Process finished with exit code 0
```

pulp求解

- 也可以使用pulp求解，见https://www.bilibili.com/video/BV12h411d7Dm?p=4
- 但是稍微繁琐

pymprog求解

- 官方文档：http://pymprog.sourceforge.net/
  http://pymprog.sourceforge.net/intro.html#whetting

举例

```
maximize  15 x + 10 y         # profit
S.T.
             x          <=  3  # mountain bike limit
                    y   <=  4  # racer limit
             x +    y   <=  5  # frame limit
             x >=0, y >=0      # non-negative
```

```
from pymprog import *

if __name__ == '__main__':
    begin('bike production')
    x, y = var('x, y')  # variables
    maximize(15 * x + 10 * y, 'profit')
    x <= 3  # mountain bike limit
    y <= 4  # racer production limit
    x + y <= 5  # metal finishing limit
    solve()

    print('x取值：' + str(x.primal))
    print('y取值：' + str(y.primal))
    print('最优解为：' + str(vobj()))

```

- res

```
GLPK Simplex Optimizer, v4.65
1 row, 2 columns, 2 non-zeros
*     0: obj =  -0.000000000e+00 inf =   0.000e+00 (2)
*     2: obj =   6.500000000e+01 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
x取值：3.0
y取值：2.0
最优解为：65.0

Process finished with exit code 0
```

整数规划

cvxpy求解

- 和线性规划差不多，但是多了个约束，那就是部分变量被约束为整数
- 目前没有一种方法可以有效地求解一切整数规划。常见的整数规划求解算法有：
  - 分支定界法：可求纯或混合整数线性规划；
  - 割平面法：可求纯或混合整数线性规划；
  - 隐枚举法：用于求解0-1整数规划，有过滤隐枚举法和分支隐枚举法；
  - 匈牙利法：解决指派问题（0-1规划特殊情形）；
  - Monte Carlo法：求解各种类型规划。

举例1

$$min \quad z = 40x_1 + 90x_2,$$
$$s.t. \begin{cases} 9x_1 + 7x_2 \leq 56, \\ -7x_1 - 20x_2 \leq -70, \\ x_1, x_2 \geq 0\text{为整数}. \end{cases}$$

- 同理也是化成<=的标准形式
- 这里的改动只需要我们输入n,a,b,c，以及第10行的小改动，n,a,b,c含义和上面线性规划一样，如果有Aeq和Beq也是同理，加上即可，然后放进cons(下面第11行)里面
- 如果是求最大，第10行用cp.Maximize

```
1 import cvxpy as cp
2 from numpy import array
3 
4 if __name__ == '__main__':
5     n = 2  # 两个变量
6     c = array([40, 90])  # 定义目标向量
7     a = array([[9, 7], [-7, -20]])  # 定义约束矩阵
```

```
    b = array([56, -70])  # 定义约束条件的右边向量
    x = cp.Variable(n, integer=True)  # 定义两个整数决策变量
    obj = cp.Minimize(c * x)  # 构造目标函数
    cons = [a * x <= b, x >= 0]  # 构造约束条件
    prob = cp.Problem(obj, cons)  # 构建问题模型
    prob.solve(solver='GLPK_MI', verbose=True)  # 求解问题
    # prob.solve(solver=cp.CPLEX, verbose=True)  # cp.CPLEX也可以
    print("最优值为:", prob.value)
    print("最优解为:", x.value)
```

- 运行结果会有警告，但是不影响结果

```
      0: obj =   2.700000000e+02 inf =   6.250e-01 (1)
      1: obj =   3.150000000e+02 inf =   0.000e+00 (0)
Long-step dual simplex will be used
+     1: mip =     not found yet >=              -inf        (1; 0)
Solution found by heuristic: 360
+     2: >>>>>   3.500000000e+02 >=   3.500000000e+02   0.0% (1; 0)
+     2: mip =   3.500000000e+02 >=     tree is empty   0.0% (0; 1)
最优值为: 350.0
最优解为: [2. 3.]

Process finished with exit code 0
```

- 参考：https://zhuanlan.zhihu.com/p/344215929

非线性规划

- 非线性规划可分为两种，目标函数是凸函数或者是非凸函数
    - 凸函数的非线性规划：如f = x^2+y^2+x*y，可以使用scipy
    - 非凸函数的非线性规划：如求极值，可以有如下方法
        - 纯数学方法，求导求极值
        - 神经网络，深度学习(bp算法链式求导)
        - scipy.optimize.minimize

```
fun: 求最小值的目标函数
args: 常数值
method: 求极值方法，一般默认。
constraints: 约束条件
x0: 变量的初始猜测值，注意 minimize是局部最优
```

举例

- 计算1/x+x的最小值
  - 只需要改12行的系数，15行的初始猜测值，和8行的函数
  - 如果结果是True，则是找到局部最优解，若是False，则结果是错误的

```
1  from scipy.optimize import minimize
2  import numpy as np
3
4
5  # f = 1/x+x
6  def fun(args):
7      a = args
8      return lambda x: a / x[0] + x[0]
9
10
11 if __name__ == '__main__':
12     args = (1)  # a
13     # x0 = np.asarray((1.5))  # 初始猜测值
14     # x0 = np.asarray((2.2))  # 初始猜测值
15     x0 = np.asarray((2))  # 设置初始猜测值
16
17     res = minimize(fun(args), x0, method='SLSQP')
18     print('最值:', res.fun)
19     print('是否是最优解', res.success)
20     print('取到最值时，x的值(最优解)是', res.x)
21
```

```
最值: 2.00000007583235
是否是最优解 True
取到最值时，x的值(最优解)是 [1.00027541]

Process finished with exit code 0
```

举例2

- $$计算(2+x_1)/(1+x_2)-3x_1+4x_3的最小值$$
- x1,x2,x3的范围都在0.1到0.9 之间
- x是变量矩阵，如x[0]即为x1
- 需要变动的是函数fun，con,27,29,32行，x0的设置要尽在要求的0.1到0.9范围内

```
1  from scipy.optimize import minimize
```

```python
import numpy as np


# 计算(2+x1)/(1+x2)- 3*x1+4*x3
def fun(args):
    a, b, c, d = args
    return lambda x: (a + x[0]) / (b + x[1]) - c * x[0] + d * x[2]


def con(args):
    # 约束条件 分为eq 和ineq
    # eq表示 函数结果等于0
    # ineq 表示 表达式大于等于0
    x1min, x1max, x2min, x2max, x3min, x3max = args
    cons = ({'type': 'ineq', 'fun': lambda x: x[0] - x1min},
            {'type': 'ineq', 'fun': lambda x: -x[0] + x1max},
            {'type': 'ineq', 'fun': lambda x: x[1] - x2min},
            {'type': 'ineq', 'fun': lambda x: -x[1] + x2max},
            {'type': 'ineq', 'fun': lambda x: x[2] - x3min},
            {'type': 'ineq', 'fun': lambda x: -x[2] + x3max})
    return cons


if __name__ == "__main__":
    # 定义常量值
    args = (2, 1, 3, 4)  # a,b,c,d
    # 设置参数范围/约束条件
    args1 = (0.1, 0.9, 0.1, 0.9, 0.1, 0.9)  # x1min, x1max, x2min,x2max
    cons = con(args1)
    # 设置初始猜测值
    x0 = np.asarray((0.5, 0.5, 0.5))
    res = minimize(fun(args), x0, method='SLSQP', constraints=cons)
    print('最值:', res.fun)
    print('是否是最优解', res.success)
    print('取到最值时，x的值(最优解)是', res.x)

```

```
最值: -0.773684210526435
是否是最优解 True
取到最值时，x的值(最优解)是 [0.9 0.9 0.1]

Process finished with exit code 0
```

- 可以看出对于这类简单函数，局部最优解与真实最优解相差不大，但是对于复杂的函数，x0的初始值设置，会很大程度影响最优解的结果

数值逼近

一维和二维插值

- 参考https://www.bilibili.com/video/BV12h411d7Dm?p=5
- 都使得图像更加光滑

最小二乘法拟合

- 用的是

```
from scipy.optimize import leastsq
```

举例

- 一组数据：

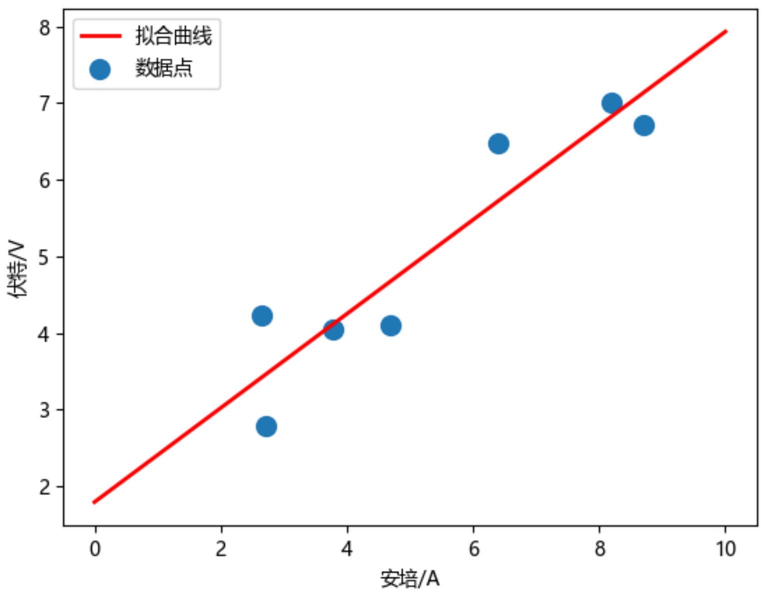
```
X = np.array([8.19, 2.72, 6.39, 8.71, 4.7, 2.66, 3.78])
Y = np.array([7.01, 2.78, 6.47, 6.71, 4.1, 4.23, 4.05])
```

- 使用leastsq

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import leastsq
from pylab import mpl

mpl.rcParams['font.sans-serif'] = ['Microsoft YaHei']  # 指定默认字体
mpl.rcParams['axes.unicode_minus'] = False  # 解决保存图像是负号'-'显示为方块的问题


# 计算以p为参数的直线与原始数据之间误差
def f(p):
    k, b = p
    return Y - (k * X + b)


if __name__ == '__main__':
    X = np.array([8.19, 2.72, 6.39, 8.71, 4.7, 2.66, 3.78])
    Y = np.array([7.01, 2.78, 6.47, 6.71, 4.1, 4.23, 4.05])
    # leastsq使得f的输出数组的平方和最小，参数初始值为[1,0]
    r = leastsq(f, [1, 0])  # 数初始值可以随便设个合理的
    k, b = r[0]
    x = np.linspace(0, 10, 1000)
    y = k * x + b

    # 画散点图，s是点的大小
    plt.scatter(X, Y, s=100, alpha=1.0, marker='o', label=u'数据点')
    # 话拟合曲线，linewidth是线宽
    plt.plot(x, y, color='r', linewidth=2, linestyle="-", markersize=20, label=u'拟合曲线')
```

```
    plt.xlabel('安培/A')  # 美赛就不用中文了
    plt.ylabel('伏特/V')
    plt.legend(loc=0, numpoints=1)  # 显示点和线的说明
    # plt.plot(X, Y)
    plt.show()

    print('k = ', k)
    print('b = ', b)

```

```
k =  0.6134953491930442
b =  1.794092543259387

Process finished with exit code 0
```



- 另外，这个线性拟合也可以使用sklearn求k和b，再去画图
  - 注意维度转换X = X.reshape(-1, 1)

```
from sklearn import linear_model
import numpy as np

if __name__ == '__main__':
    X = np.array([8.19, 2.72, 6.39, 8.71, 4.7, 2.66, 3.78])
    Y = np.array([7.01, 2.78, 6.47, 6.71, 4.1, 4.23, 4.05])
    X = X.reshape(-1, 1)
    model = linear_model.LinearRegression()
    model.fit(X, Y)
    print('k = ', model.coef_)
    print('b = ', model.intercept_)

```

```
k =  [0.61349535]
b =  1.7940925542916233

Process finished with exit code 0
```

- 更多线性和非线性问题、如多元回归、逻辑回归、其他分类问题，见我之前的sklearn blog

微分方程

- 微分方程是用来描述某一类函数与其导数之间关系的方程，其解是一个符合方程的函数。微分方程按自变量个数可分为 **常微分方程**和**偏微分方程**,

  前者表达通式 ： $f\left(\frac{d^n y}{dx^n}, \frac{d^{(n-1)}y}{dx^{(n-1)}}, \dots, \frac{dy}{dx}, y\right) = p(x)$

  后者表达通式： $\frac{\partial \emptyset}{\partial x} + x\frac{\partial \emptyset}{\partial y} = 0$

- 建议稍微复习一下高数上册最后微分方程那章再看看会更好

解析解

- 使用sympy库，但是得到的是字符形式的格式

  如下这种,如果结果是比较复杂的，可能太丑

  如

  以求解阻尼谐振子的二阶ODE为例，其表达式为：

  $$\frac{d^2x(t)}{dt^2} + 2\gamma\omega_0\frac{dx(t)}{dt} + \omega_0^2 x(t) = 0$$

  $$initial\ conditions. \begin{cases} x(0) = 1 \\ \left.\frac{dx(t)}{dt}\right|_{t=0} = 0 \end{cases}$$

```python
import sympy

def apply_ics(sol, ics, x, known_params):
    free_params = sol.free_symbols - set(known_params)
    eqs = [(sol.lhs.diff(x, n) - sol.rhs.diff(x, n)).subs(x, 0).subs(ics) for n in range(len(ics))]
    sol_params = sympy.solve(eqs, free_params)
    return sol.subs(sol_params)

```

```python

if __name__ == '__main__':
    sympy.init_printing()  # 初始化打印环境
    t, omega0, gamma = sympy.symbols("t, omega_0, gamma", positive=True)  # 标记参数，且均为正
    x = sympy.Function('x')  # 标记x是微分函数，非变量
    ode = x(t).diff(t, 2) + 2 * gamma * omega0 * x(t).diff(t) + omega0 ** 2 * x(t)
    ode_sol = sympy.dsolve(ode)  # 用diff()和dsolve得到通解
    ics = {x(0): 1, x(t).diff(t).subs(t, 0): 0}  # 将初始条件字典匹配
    x_t_sol = apply_ics(ode_sol, ics, t, [omega0, gamma])
    sympy.pprint(x_t_sol)

```

- 此段解释可见：https://www.bilibili.com/video/BV12h411d7Dm?p=6

```
                                            ⎛       _______   _______⎞
       ⎛        γ           1⎞  ω₀⋅t⋅⎝-γ - ╲╱ γ - 1 ⋅╲╱ γ + 1 ⎠  ⎛       γ
x(t) = ⎜- ───────────── + ─⎟⋅ℯ                                  + ⎜─────────────
       ⎜       ________   2⎟                                      ⎜   ________
       ⎜      ╱  2         ⎟                                      ⎜  ╱  2
       ⎝  2⋅╲╱  γ  - 1     ⎠                                      ⎝2⋅╲╱  γ  - 1

                 ⎛       _______   _______⎞
    1⎞  ω₀⋅t⋅⎝-γ + ╲╱ γ - 1 ⋅╲╱ γ + 1 ⎠
  + ─⎟⋅ℯ
    2⎟
     ⎟
     ⎠

Process finished with exit code 0
```

- 如果最后加上`print(x_t_sol)`

  结果为：

```
Eq(x(t), (-gamma/(2*sqrt(gamma**2 - 1)) + 1/2)*exp(omega_0*t*(-gamma - sqrt(gamma - 1)*sqrt(gamma + 1))) + (gamma/(2*sqrt(gamma**2 - 1)) + 1/2)*exp(omega_0*t*(-gamma + sqrt(gamma - 1)*sqrt(gamma + 1))))

```

- 结果较为简单的常微分方程
  - f(x)''+f(x)=0 二阶常系数齐次微分方程

```python
import sympy as sy

# f(x)''+f(x)=0 二阶常系数齐次微分方程
def differential_equation(x, f):
    return sy.diff(f(x), x, 2) + f(x)


if __name__ == '__main__':
    x = sy.symbols('x')  # 约定变量
    f = sy.Function('f')  # 约定函数
    print(sy.dsolve(differential_equation(x, f), f(x)))  # 打印
    sy.pprint(sy.dsolve(differential_equation(x, f), f(x)))  # 漂亮的打印

```

```
Eq(f(x), C1*sin(x) + C2*cos(x))
f(x) = C₁⋅sin(x) + C₂⋅cos(x)

Process finished with exit code 0
```

- 可以参考：https://blog.csdn.net/your_answer/article/details/79234275

数值解

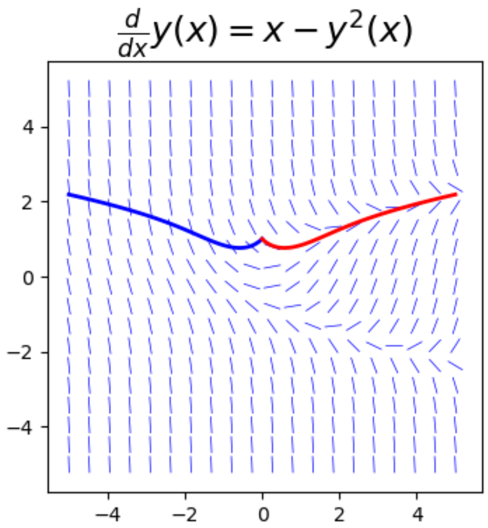
- 当ODE(常微分方程)无法求得解析解时，可以用scipy中的integrate.odeint求数值解来探索其解的部分性质,并辅以可视化，能直观地展现ODE解的函数表达

举例

- 一阶非线性常微分方程 $$\frac{dy}{dx} = x - y(x)^2$$
- plot_direction_field函数里面的参数意义：
  - y_x:也就是y(x)
  - f_xy:也就是x-y(x)^2
  - x_lim=(-5, 5), y_lim=(-5, 5)也就是在这个x，y轴的范围展示出来
- 关键需要修改的部分,29-31行，33行的y0需要适当调

```python
x = sympy.symbols('x')
y = sympy.Function('y')
f = x - y(x) ** 2
```

```python
import numpy as np
from scipy import integrate
import matplotlib.pyplot as plt
import sympy


def plot_direction_field(x, y_x, f_xy, x_lim=(-5, 5), y_lim=(-5, 5), ax=None):
    f_np = sympy.lambdify((x, y_x), f_xy, 'numpy')
    x_vec = np.linspace(x_lim[0], x_lim[1], 20)
    y_vec = np.linspace(y_lim[0], y_lim[1], 20)
    if ax is None:
        _, ax = plt.subplots(figsize=(4, 4))

    dx = x_vec[1] - x_vec[0]
    dy = y_vec[1] - y_vec[0]
    for m, xx in enumerate(x_vec):
        for n, yy in enumerate(y_vec):
            Dy = f_np(xx, yy) * dx
            Dx = 0.8 * dx ** 2 / np.sqrt(dx ** 2 + Dy ** 2)
            Dy = 0.8 * Dy * dy / np.sqrt(dx ** 2 + Dy ** 2)
            ax.plot([xx - Dx / 2, xx + Dx / 2], [yy - Dy / 2, yy + Dy / 2], 'b', lw=0.5)

    ax.axis('tight')
    ax.set_title(r'$ % s$' % (sympy.latex(sympy.Eq(y_x.diff(x), f_xy))), fontsize=18)
    return ax


if __name__ == '__main__':
    x = sympy.symbols('x')
    y = sympy.Function('y')
    f = x - y(x) ** 2
    f_np = sympy.lambdify((y(x), x), f)  # 符号表达式转隐函数
    y0 = 1 # odeint需要给个初始值
    xp = np.linspace(0, 5, 100)
    yp = integrate.odeint(f_np, y0, xp)  # 初始y0解f_np,x范围xp
    xn = np.linspace(0, -5, 100)
    yn = integrate.odeint(f_np, y0, xp)
    fig, ax = plt.subplots(1, 1, figsize=(4, 4))
    plot_direction_field(x, y(x), f, ax=ax)  # 绘制f的场线图
    ax.plot(xn, yn, 'b', lw=2)
    ax.plot(xp, yp, 'r', lw=2)
    plt.show()

```



传染病模型

- 传染病模型研究属于传染病动力学研究方向，这里只是将模型中微分方程进行了python实现
- 传染病模型包括：SI、SIS、SIR、SIRS、SEIR、SEIRS共六个模型

SI模型

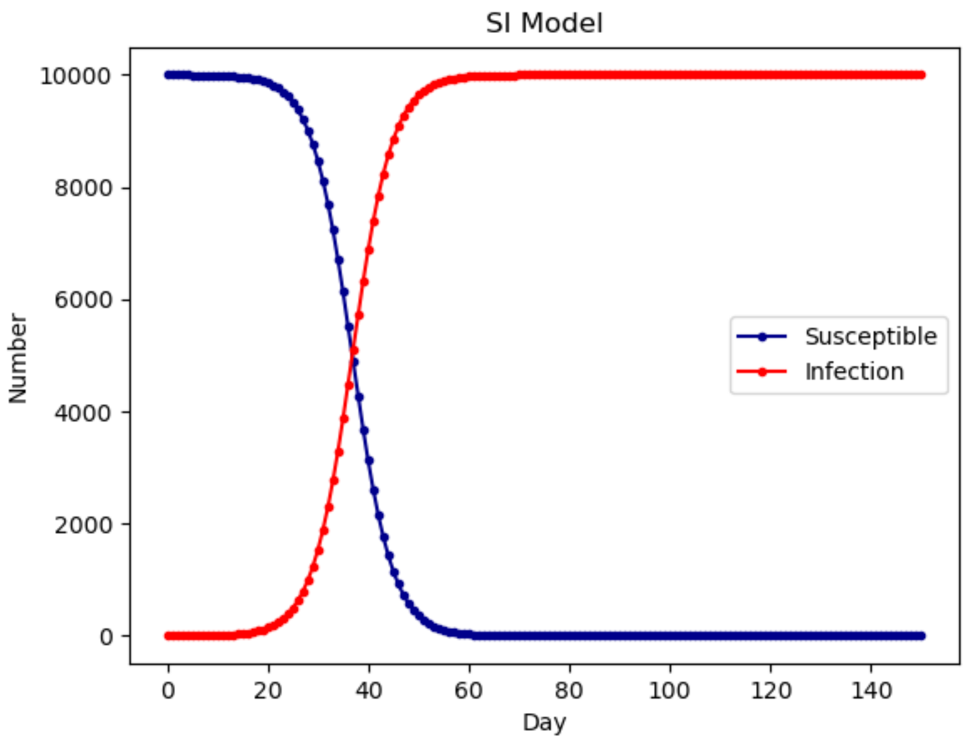
- 比如病毒传染初期，没有加防疫手段，就符合SI模型
- SI模型的表达式见网上(S:易感染，I:已感染)
- 需要修改的参数

```
N = 10000  # N为人群总数
beta = 0.25  # β为传染率系数
gamma = 0  # gamma为恢复率系数
I_0 = 1  # I_0为感染者的初始人数
S_0 = N - I_0  # S_0为易感染者的初始人数
T = 150  # T为传播时间
```

  - β为传染率系数,比如现在100个人已经传染了，在一段时间内，传染新增了25人，则β为0.25
  - gamma为恢复率系数，一开始没有抗体都是为0的，如果不为0，比如是开始有100人感染，在一个传播时间T后，治愈了6个人，则gamma取0.06
  - I_0为感染者的初始人数
  - S_0为易感染者的初始人数，这个要看情况，如果都不加干预，那就是N - I_0，一般看情况需要再考虑其他因素(交通，社交群体，航线等)，S_0考虑的越多，则越完备
  - Susceptible易感染的，Infection已经感染的
- code

```
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt

N = 10000  # N为人群总数
```

```python
beta = 0.25  # β为传染率系数
gamma = 0  # gamma为恢复率系数
I_0 = 1  # I_0为感染者的初始人数
S_0 = N - I_0  # S_0为易感染者的初始人数
T = 150  # T为传播时间
INI = (S_0, I_0)  # INI为初始状态下的数组


def funcSI(inivalue, _):
    Y = np.zeros(2)
    X = inivalue
    Y[0] = -(beta * X[0] * X[1]) / N + gamma * X[1]  # 易感个体变化
    Y[1] = (beta * X[0] * X[1]) / N - gamma * X[1]  # 感染个体变化
    return Y


if __name__ == '__main__':
    T_range = np.arange(0, T + 1)
    RES = spi.odeint(funcSI, INI, T_range)
    plt.plot(RES[:, 0], color='darkblue', label='Susceptible', marker='.')
    plt.plot(RES[:, 1], color='red', label='Infection', marker='.')
    plt.title('SI Model')
    plt.legend()
    plt.xlabel('Day')
    plt.ylabel('Number')
    plt.show()

```

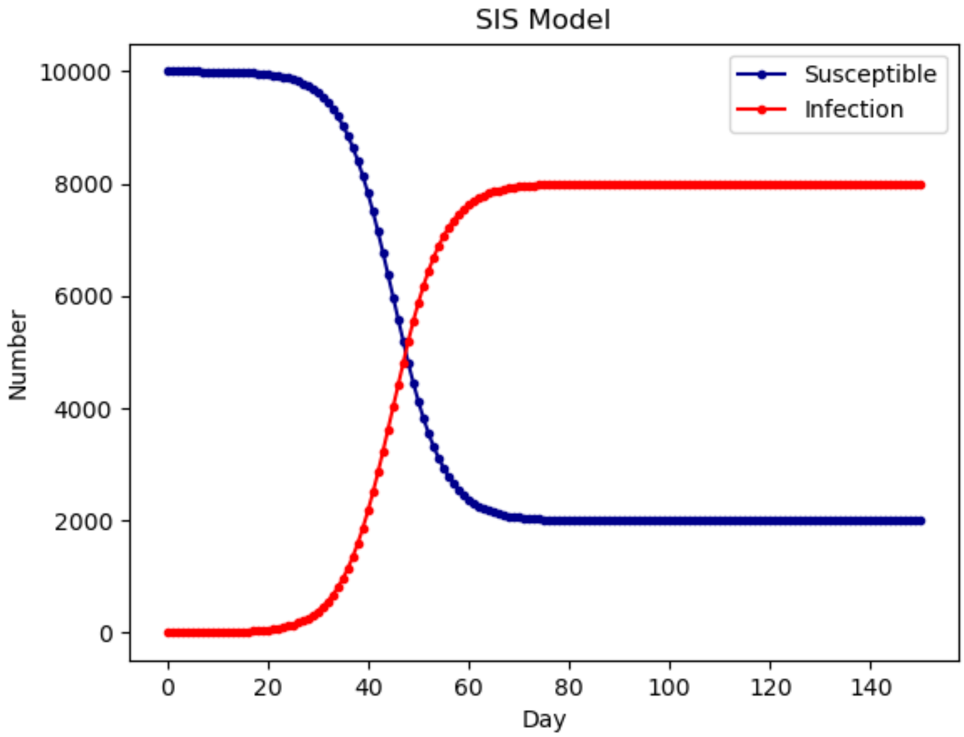


- 可以看到10000个人大概在60天左右就全部感染了

SIS模型

- 与SI区别不大，区别在于7行的gamma有初始值，以及17行的公式改变了

```python
1 import numpy as np
2 import scipy.integrate as spi
3 import matplotlib.pyplot as plt
4 
5 N = 10000  # N为人群总数
6 beta = 0.25  # β为传染率系数
7 gamma = 0.05  # gamma为恢复率系数
8 I_0 = 1  # I_0为感染者的初始人数
9 S_0 = N - I_0  # S_0为易感染者的初始人数
10 T = 150  # T为传播时间
11 INI = (S_0, I_0)  # INI为初始状态下的数组
12 
13 
14 def funcSI(inivalue, _):
15     Y = np.zeros(2)
16     X = inivalue
17     Y[0] = -(beta * X[0]) / N * X[1] + gamma * X[1]  # 易感个体变化
18     Y[1] = (beta * X[0] * X[1]) / N - gamma * X[1]  # 感染个体变化
19     return Y
20 
21 
22 if __name__ == '__main__':
23     T_range = np.arange(0, T + 1)
24     RES = spi.odeint(funcSI, INI, T_range)
25     plt.plot(RES[:, 0], color='darkblue', label='Susceptible', marker='.')
26     plt.plot(RES[:, 1], color='red', label='Infection', marker='.')
27     plt.title('SIS Model')
28     plt.legend()
29     plt.xlabel('Day')
30     plt.ylabel('Number')
31     plt.show()
32 
```



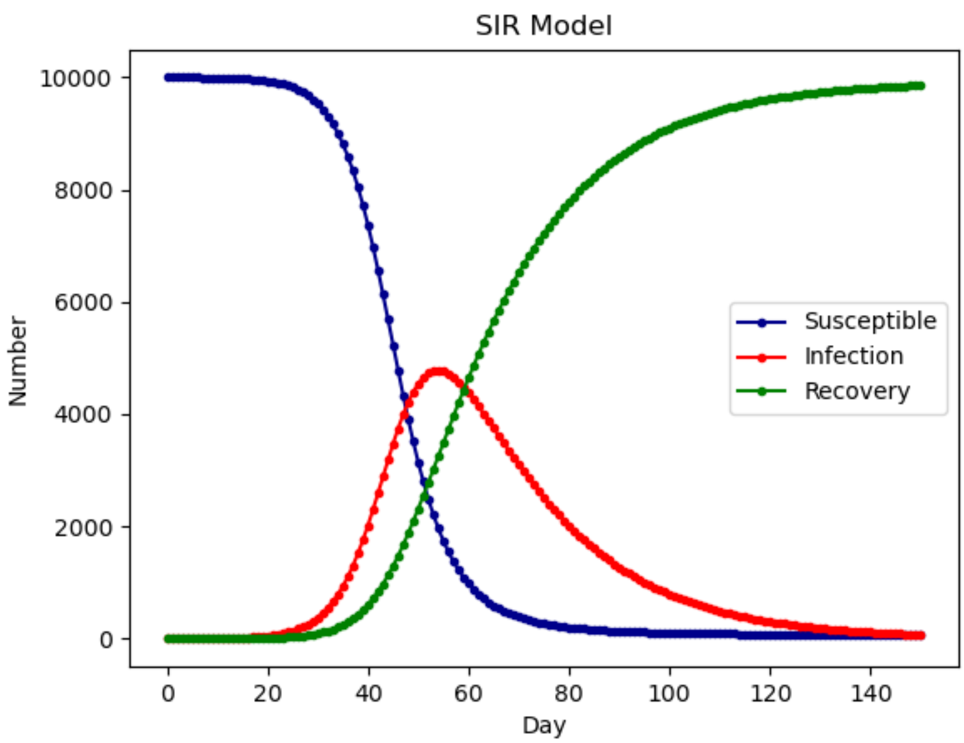
- 可以看到60-80天之间，逐渐稳定，有的人治愈后(获得抗体)活了下来，有的没治愈的就死了

SIR模型

- 多了R_0为治愈者的初始人数，即刚开始注射疫苗恢复的人
- 表达式也改变
- 注意恢复治愈包括自身产生抗体以及通过医疗手段获得抗体两种

```
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt

N = 10000  # N为人群总数
beta = 0.25  # β为传染率系数
gamma = 0.05  # gamma为恢复率系数
I_0 = 1  # I_0为感染者的初始人数
R_0 = 0  # R_0为治愈者的初始人数
S_0 = N - I_0 - R_0  # S_0为易感染者的初始人数
T = 150  # T为传播时间
INI = (S_0, I_0, R_0)  # INI为初始状态下的数组


def funcSIR(inivalue, _):
    Y = np.zeros(3)
    X = inivalue
    Y[0] = -(beta * X[0] * X[1]) / N  # 易感个体变化
    Y[1] = (beta * X[0] * X[1]) / N - gamma * X[1]  # 感染个体变化
```

```python
    Y[2] = gamma * X[1]  # 治愈个体变化
    return Y


if __name__ == '__main__':
    T_range = np.arange(0, T + 1)
    RES = spi.odeint(funcSIR, INI, T_range)
    plt.plot(RES[:, 0], color='darkblue', label='Susceptible', marker='.')
    plt.plot(RES[:, 1], color='red', label='Infection', marker='.')
    plt.plot(RES[:, 2], color='green', label='Recovery', marker='.')
    plt.title('SIR Model')
    plt.legend()
    plt.xlabel('Day')
    plt.ylabel('Number')
    plt.show()

```



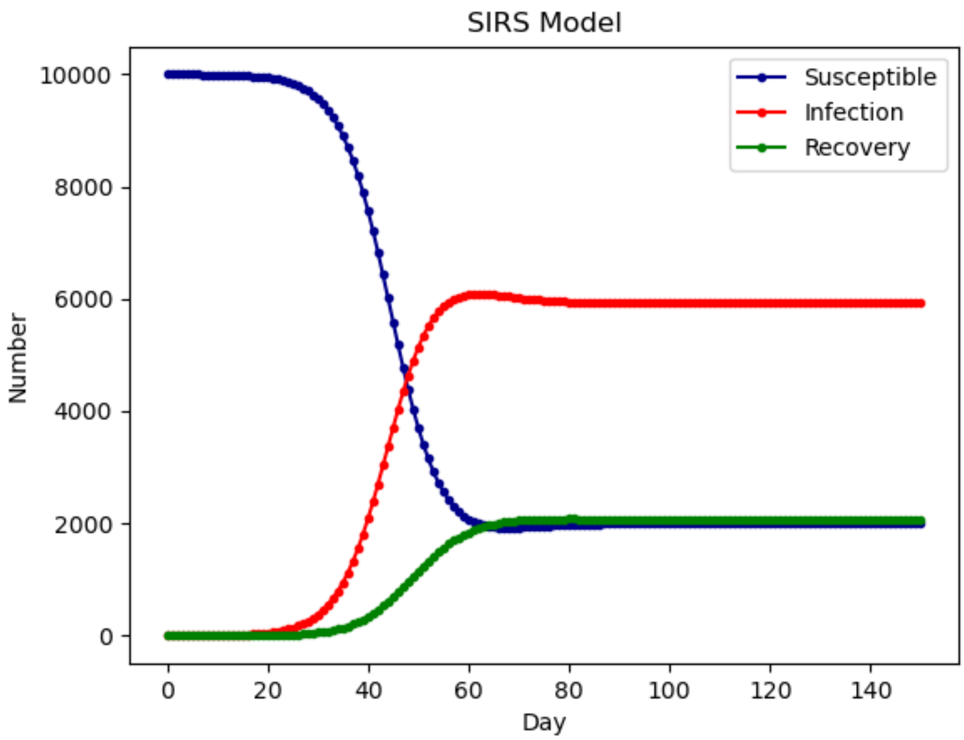
- 可以看到感染人数出现峰值，在前期已经开始使用抗体，病人逐渐治愈，最后所有人都恢复健康

SIRS模型

- 多了Ts为抗体持续时间，也就是说有了抗体一段时间后，抗体失效，又变成了易感染人群
- 公式也改变

```python
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt
```

```python

N = 10000  # N为人群总数
beta = 0.25  # β为传染率系数
gamma = 0.05  # gamma为恢复率系数
Ts = 7  # Ts为抗体持续时间
I_0 = 1  # I_0为感染者的初始人数
R_0 = 0  # R_0为治愈者的初始人数
S_0 = N - I_0 - R_0  # S_0为易感染者的初始人数
T = 150  # T为传播时间
INI = (S_0, I_0, R_0)  # INI为初始状态下的数组


def funcSIRS(inivalue, _):
    Y = np.zeros(3)
    X = inivalue
    Y[0] = -(beta * X[0] * X[1]) / N + X[2] / Ts  # 易感个体变化
    Y[1] = (beta * X[0] * X[1]) / N - gamma * X[1]  # 感染个体变化
    Y[2] = gamma * X[1] - X[2] / Ts  # 治愈个体变化
    return Y


if __name__ == '__main__':
    T_range = np.arange(0, T + 1)
    RES = spi.odeint(funcSIRS, INI, T_range)
    plt.plot(RES[:, 0], color='darkblue', label='Susceptible', marker='.')
    plt.plot(RES[:, 1], color='red', label='Infection', marker='.')
    plt.plot(RES[:, 2], color='green', label='Recovery', marker='.')
    plt.title('SIRS Model')
    plt.legend()
    plt.xlabel('Day')
    plt.ylabel('Number')
    plt.show()

```



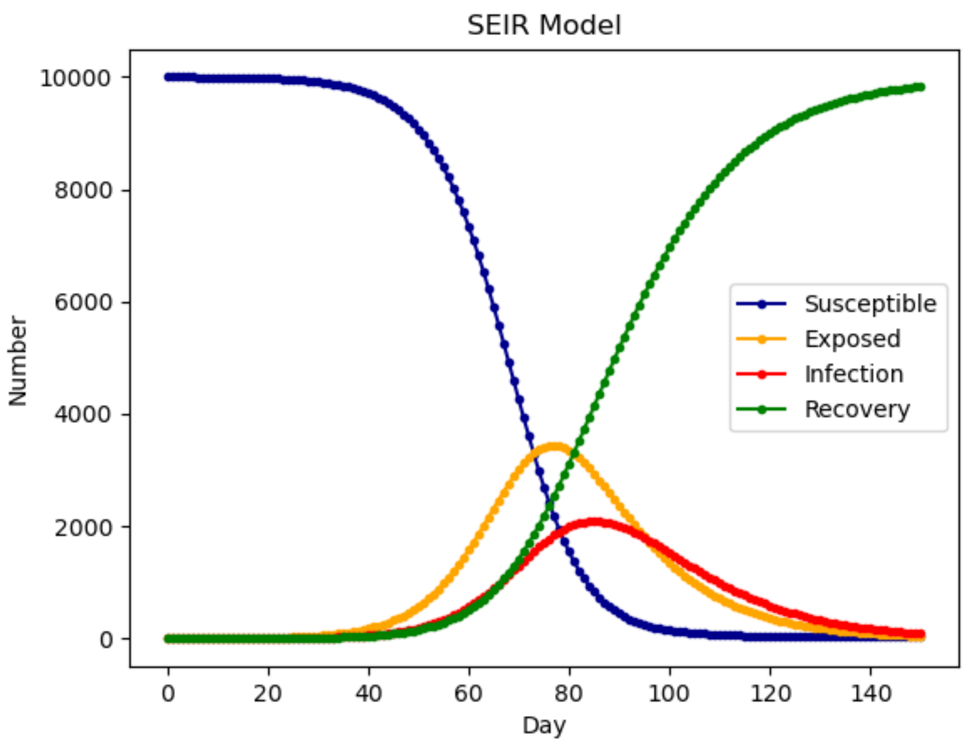
- 最终达到一个平衡

SEIR模型

- 考虑了病毒的潜伏期(潜伏人群E)，也就是感染病毒后，过了潜伏期就是感染人群了
- 多了E_0为潜伏者的初始人数，如果是0，那说明开始时有人感染，但是还没有发病，此时这类人不是易感染，但是他们携带病毒
- 只有经过潜伏期，才能被传染
- 公式有所变化

```
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt

N = 10000  # N为人群总数
beta = 0.6  # β为传染率系数
gamma = 0.1  # gamma为恢复率系数
Te = 14  # Te为疾病潜伏期
I_0 = 1  # I_0为感染者的初始人数
E_0 = 0  # E_0为潜伏者的初始人数
R_0 = 0  # R_0为治愈者的初始人数
S_0 = N - I_0 - R_0 - E_0  # S_0为易感染者的初始人数
T = 150  # T为传播时间
INI = (S_0, E_0, I_0, R_0)  # INI为初始状态下的数组


def funcSEIR(inivalue, _):
```

```python
    Y = np.zeros(4)
    X = inivalue
    Y[0] = -(beta * X[0] * X[2]) / N  # 易感个体变化
    Y[1] = (beta * X[0] * X[2] / N - X[1] / Te)  # 潜伏个体变化
    Y[2] = X[1] / Te - gamma * X[2]  # 感染个体变化
    Y[3] = gamma * X[2]  # 治愈个体变化
    return Y


if __name__ == '__main__':
    T_range = np.arange(0, T + 1)
    RES = spi.odeint(funcSEIR, INI, T_range)
    plt.plot(RES[:, 0], color='darkblue', label='Susceptible', marker='.')
    plt.plot(RES[:, 1], color='orange', label='Exposed', marker='.')
    plt.plot(RES[:, 2], color='red', label='Infection', marker='.')
    plt.plot(RES[:, 3], color='green', label='Recovery', marker='.')
    plt.title('SEIR Model')
    plt.legend()
    plt.xlabel('Day')
    plt.ylabel('Number')
    plt.show()

```

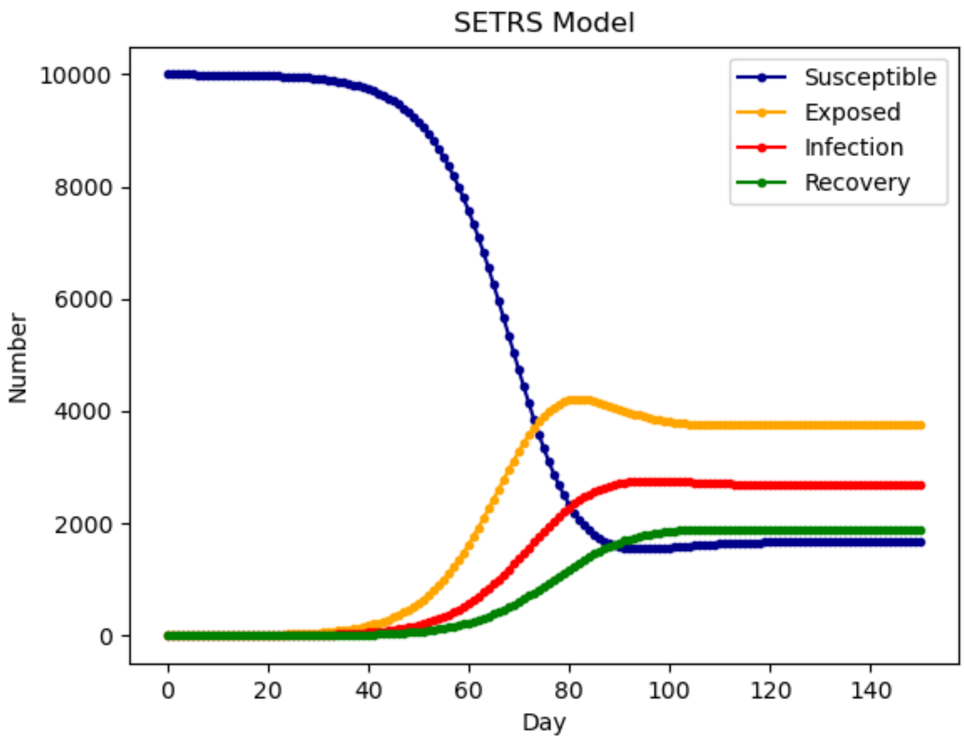


- 潜伏期的人会比感染人群先到达峰值，最终都可以治愈

SEIRS模型

- 考虑了抗体持续时间
- 一般多了潜伏期的话，传染率系数会有所增加，上面的SEIR也是同理

```python
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt

N = 10000  # N为人群总数
beta = 0.6  # β为传染率系数
gamma = 0.1  # gamma为恢复率系数
Ts = 7  # Ts为抗体持续时间
Te = 14  # Te为疾病潜伏期
I_0 = 1  # I_0为感染者的初始人数
E_0 = 0  # E_0为潜伏者的初始人数
R_0 = 0  # R_0为治愈者的初始人数
S_0 = N - I_0 - R_0 - E_0  # S_0为易感染者的初始人数
T = 150  # T为传播时间
INI = (S_0, E_0, I_0, R_0)  # INI为初始状态下的数组


def funcSEIRS(inivalue, _):
    Y = np.zeros(4)
    X = inivalue
    Y[0] = -(beta * X[0] * X[2]) / N + X[3] / Ts  # 易感个体变化
    Y[1] = (beta * X[0] * X[2] / N - X[1] / Te)  # 潜伏个体变化
    Y[2] = X[1] / Te - gamma * X[2]  # 感染个体变化
    Y[3] = gamma * X[2] - X[3] / Ts  # 治愈个体变化
    return Y


if __name__ == '__main__':
    T_range = np.arange(0, T + 1)
    RES = spi.odeint(funcSEIRS, INI, T_range)
    plt.plot(RES[:, 0], color='darkblue', label='Susceptible', marker='.')
    plt.plot(RES[:, 1], color='orange', label='Exposed', marker='.')
    plt.plot(RES[:, 2], color='red', label='Infection', marker='.')
    plt.plot(RES[:, 3], color='green', label='Recovery', marker='.')
    plt.title('SETRS Model')
    plt.legend()
    plt.xlabel('Day')
    plt.ylabel('Number')
    plt.show()

```



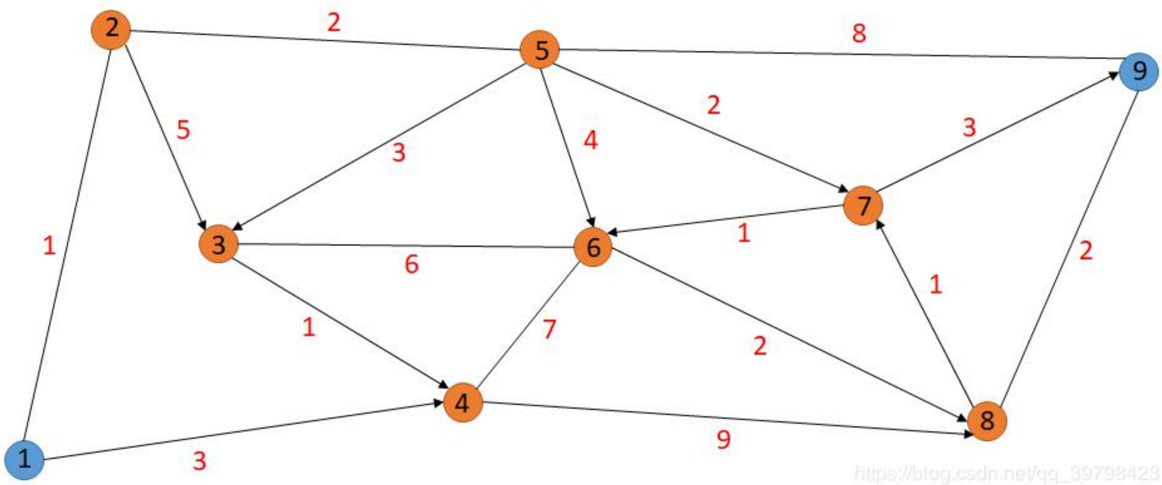
- 潜伏期的人先到峰值，然后是易感染者，然后是治愈者，他们最终会达到平衡稳定

图论

Dijkstra

解法1(常用)

- 只需要给出带有权值的邻接矩阵即可求出最短路径和最短距离
- 需要改变的第54行邻接矩阵的权值和65行的起点和终点，注意21行是从0还是1开始
- 有向边和无向边的混合均可使用
- `g = defaultdict(list)`是得到一个元素全是list类型的字典
- 举例

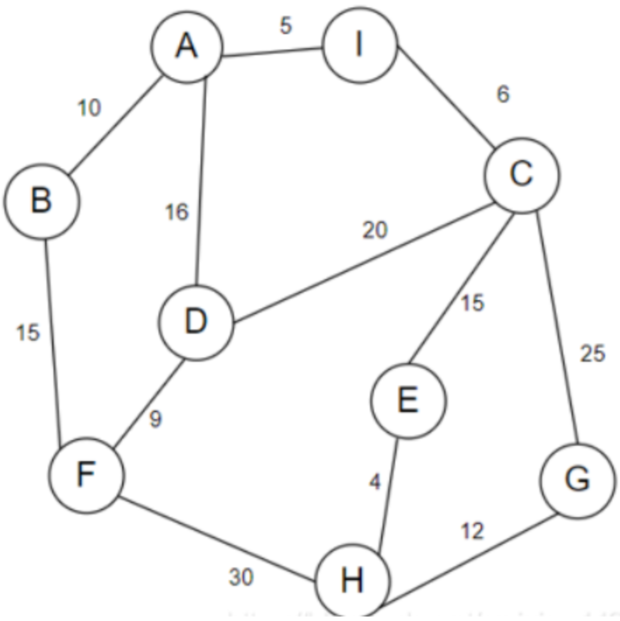


```python
# dijkstra
from collections import defaultdict
from heapq import *

inf = 99999  # 不连通值


def init_graph(mtx_graph):
    m_n = len(mtx_graph)  # 带权连接矩阵的阶数
    edges = []  # 保存连通的两个点之间的距离(点A、点B、距离)
    for i in range(m_n):
        for j in range(m_n):
            if i != j and mtx_graph[i][j] != inf:
                edges.append((i, j, mtx_graph[i][j]))

    return edges


def dijkstra(edges, from_node, to_node):
    go_path = []
    to_node = to_node - 1 # 看情况，如果是从1开始的就减一
    g = defaultdict(list)
    for l, r, c in edges:
        g[l].append((c, r))
    q, seen = [(0, from_node - 1, ())], set()
    while q:
        (cost, v1, path) = heappop(q)  # 堆弹出当前路径最小成本
        if v1 not in seen:
            seen.add(v1)
            path = (v1, path)
            if v1 == to_node:
                break
            for c, v2 in g.get(v1, ()):
                if v2 not in seen:
                    heappush(q, (cost + c, v2, path))
    if v1 != to_node:  # 无法到达
        return float['inf'], []

    if len(path) > 0:
        left = path[0]
        go_path.append(left)
        right = path[1]
        while len(right) > 0:
            left = right[0]
            go_path.append(left)
            right = right[1]
        go_path.reverse()  # 逆序变换
        for i in range(len(go_path)):  # 标号加1
            go_path[i] = go_path[i] + 1
    return cost, go_path
```

```python


if __name__ == '__main__':
    mtx_graph = [[0, 1, inf, 3, inf, inf, inf, inf, inf],
                 [1, 0, 5, inf, 2, inf, inf, inf, inf],
                 [inf, inf, 0, 1, inf, 6, inf, inf, inf],
                 [inf, inf, inf, 0, inf, 7, inf, 9, inf],
                 [inf, 2, 3, inf, 0, 4, 2, inf, 8],
                 [inf, inf, 6, 7, inf, 0, inf, 2, inf],
                 [inf, inf, inf, inf, inf, 1, 0, inf, 3],
                 [inf, inf, inf, inf, inf, inf, 1, 0, 2],
                 [inf, inf, inf, inf, 8, inf, 3, 2, 0]]

    edges = init_graph(mtx_graph)
    length, path = dijkstra(edges, 1, 9)
    print('最短距离为: ' + str(length))
    print('前进路径为: ' + str(path))

```

```
最短距离为: 8
前进路径为: [1, 2, 5, 7, 9]

Process finished with exit code 0
```

解法2

- 输入是一个包含每个点与其他点联系和权值的字典
- 下面是无向边的例子，适合无向边
- 需要修改的是51的联系，62的起点，66行的起点和终点
- 


```python
1  import heapq
2  Max = 99999999
3
```

```python

# 有点像BFS的思想
def dijktestra(graph, start):
    queue = []  # 优先队列
    heapq.heappush(queue, (0, start))
    visited = set()
    path = {start: None}  # 记录该点的上一个点

    # 先把一开始到达的所有路径距离设最大
    distance = {start: 0}
    for vertex in graph:
        if vertex != start:
            distance[vertex] = Max

    while len(queue):
        # 取出的当前在queue的第一个点
        pair = heapq.heappop(queue)
        dist = pair[0]
        vertex = pair[1]
        visited.add(vertex)

        # 该点的所有连接点
        nodes = graph[vertex].keys()
        for v in nodes:
            if v not in visited and dist + graph[vertex][v] < distance[v]:
                heapq.heappush(queue, (dist + graph[vertex][v], v))  # 优先队列会自动把值最小的放在前面
                path[v] = vertex  # 记录上一个点
                distance[v] = dist + graph[vertex][v]  # 更新最小值

    return path, distance


def show_path(path, start, end):
    shortest_path = []
    vertex = end
    while vertex != path[start]:
        vertex = path[vertex]
        shortest_path.append(vertex)

    shortest_path.reverse()
    shortest_path.pop(0)
    shortest_path.append(end)

    return shortest_path


if __name__ == '__main__':
    graph = {
        'A': {'B': 10, 'D': 16, 'I': 5},
```

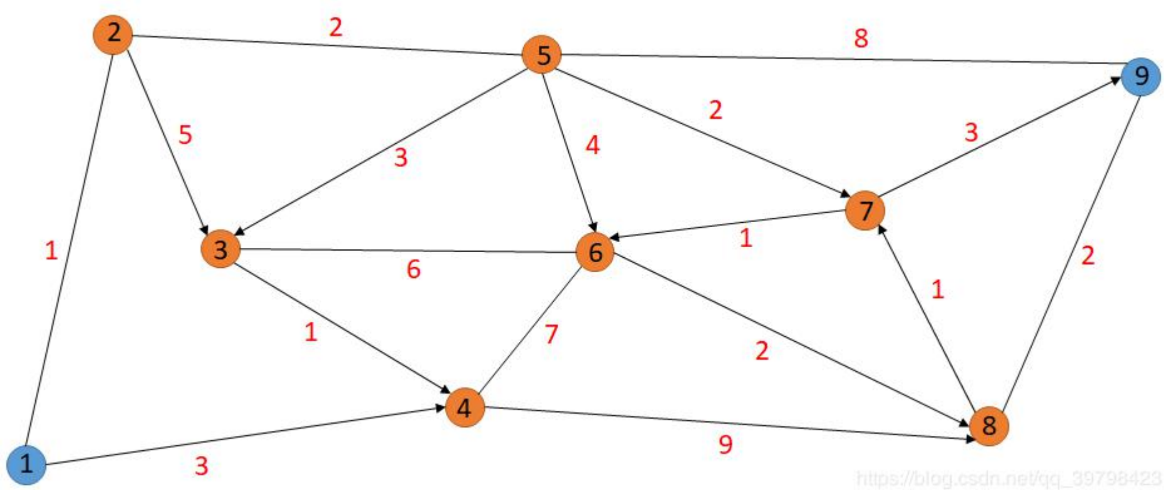
```python
        'B': {'A': 10, 'F': 15},
        'C': {'D': 20, 'E': 15, 'I': 6},
        'D': {'A': 16, 'C': 20, 'F': 9},
        'E': {'C': 15, 'H': 4},
        'F': {'B': 15, 'H': 30},
        'G': {'C': 25, 'H': 12},
        'H': {'E': 4, 'F': 9, 'G': 12},
        'I': {'A': 5, 'C': 6}
    }
    path, distance = dijktestra(graph, 'A')
    print(path)
    print(distance)

    shortest_path = show_path(path, 'A', 'H')
    print('shortest_path:', shortest_path)

```

```
该点的上一个点： {'A': None, 'B': 'A', 'D': 'A', 'I': 'A', 'C': 'I', 'F': 'B', 'E': 'C', 'H': 'E', 'G': 'H'}
起点到其他各个点的最小距离： {'A': 0, 'B': 10, 'C': 11, 'D': 16, 'E': 26, 'F': 25, 'G': 42, 'H': 30, 'I': 5}
shortest_path: ['A', 'I', 'C', 'E', 'H']
```

Floyd

- 通过动态规划求解多源最短路径问题

举例



- 图和上面的图一样，求出从每一个点到其他点的最短距离和路径

```python
import numpy as np

inf = 99999  # 不连通值


def floyd(graph):
    N = len(graph)
    A = np.array(graph)
    path = np.zeros((N, N))
    for i in range(0, N):
        for j in range(0, N):
            if A[i][j] != inf:
                path[i][j] = j

    for k in range(0, N):
        for i in range(0, N):
            for j in range(0, N):
                if A[i][k] + A[k][j] < A[i][j]:
                    A[i][j] = A[i][k] + A[k][j]
                    path[i][j] = path[i][k]

    for i in range(0, N):
        for j in range(0, N):
            path[i][j] = path[i][j] + 1

    print('距离 = \n', A)
    print('路径 = \n', path)


if __name__ == '__main__':
    mtx_graph = [[0, 1, inf, 3, inf, inf, inf, inf, inf],
                 [1, 0, 5, inf, 2, inf, inf, inf, inf],
                 [inf, inf, 0, 1, inf, 6, inf, inf, inf],
                 [inf, inf, inf, 0, inf, 7, inf, 9, inf],
                 [inf, 2, 3, inf, 0, 4, 2, inf, 8],
                 [inf, inf, 6, 7, inf, 0, inf, 2, inf],
                 [inf, inf, inf, inf, inf, 1, 0, inf, 3],
                 [inf, inf, inf, inf, inf, inf, 1, 0, 2],
                 [inf, inf, inf, inf, 8, inf, 3, 2, 0]]
    floyd(mtx_graph)

```

```
距离 =
 [[ 0  1  6  3  3  6  5  8  8]
 [ 1  0  5  4  2  5  4  7  7]
 [21 20  0  1 18  6  9  8 10]
 [22 21 13  0 19  7 10  9 11]
 [ 3  2  3  4  0  3  2  5  5]
 [15 14  6  7 12  0  3  2  4]
```

```
 [14 13  7  8 11  1  0  3  3]
 [13 12  8  9 10  2  1  0  2]
 [11 10 10 11  8  4  3  2  0]]
路径 =
 [[1. 2. 2. 4. 2. 2. 2. 2. 2.]
 [1. 2. 3. 1. 5. 5. 5. 5. 5.]
 [6. 6. 3. 4. 6. 6. 6. 6. 6.]
 [8. 8. 6. 4. 8. 6. 8. 8. 8.]
 [2. 2. 3. 3. 5. 7. 7. 7. 7.]
 [8. 8. 3. 4. 8. 6. 8. 8. 8.]
 [9. 9. 6. 6. 9. 6. 7. 6. 9.]
 [9. 9. 7. 7. 9. 7. 7. 8. 9.]
 [5. 5. 7. 7. 5. 7. 7. 8. 9.]]

Process finished with exit code 0
```

- 距离，比如1到9的距离为8，即从第1行看到第9列
- 怎么看最短路径看解释见下图

```
距离 =
 [[ 0  1  6  3  3  6  5  8  8]
 [ 1  0  5  4  2  5  4  7  7]
 [21 20  0  1 18  6  9  8 10]
 [22 21 13  0 19  7 10  9 11]
 [ 3  2  3  4  0  3  2  5  5]
 [15 14  6  7 12  0  3  2  4]
 [14 13  7  8 11  1  0  3  3]
 [13 12  8  9 10  2  1  0  2]
 [11 10 10 11  8  4  3  2  0]]
路径 =
 [[1. 2. 2. 4. 2. 2. 2. 2. 2.]
 [1. 2. 3. 1. 5. 5. 5. 5. 5.]
 [6. 6. 3. 4. 6. 6. 6. 6. 6.]
 [8. 8. 6. 4. 8. 6. 8. 8. 8.]
 [2. 2. 3. 3. 5. 7. 7. 7. 7.]
 [8. 8. 3. 4. 8. 6. 8. 8. 8.]
 [9. 9. 6. 6. 9. 6. 7. 6. 9.]
 [9. 9. 7. 7. 9. 7. 7. 8. 9.]
 [5. 5. 7. 7. 5. 7. 7. 8. 9.]]

Process finished with exit code 0
```

1 2 5 7 9

机场航线设计

- 图的可视化

- 数据清洗分析，可参考Kaggle练习
- 城市可作为图节点
- 这种一般考虑
  - 找到最密集的点，作为交通枢纽，考虑其他成本、时效性、盈利因素之类的...

回归

- 多元回归、逻辑回归见我之前的blog：https://www.cnblogs.com/jmchen/p/13550573.html

差分方程

递推关系

- 差分方程建模的关键在于如何得到第**n**组数据与第**n+1**组数据之间的关系

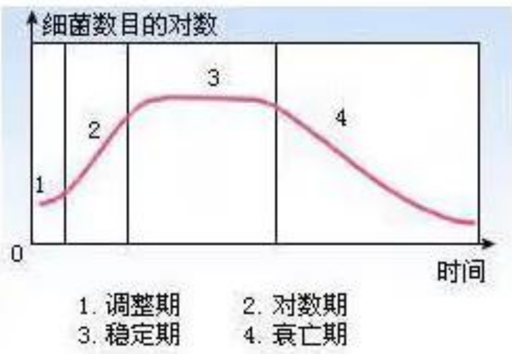
举例

酵母菌生长模型

- 相类比的还有比如兔子(其他生物)繁殖模型等
- 如图所示我们用培养基培养细菌时，其数量变化通常会经历这四个时期。

  这个模型针对前三个时期建一个大致的模型：

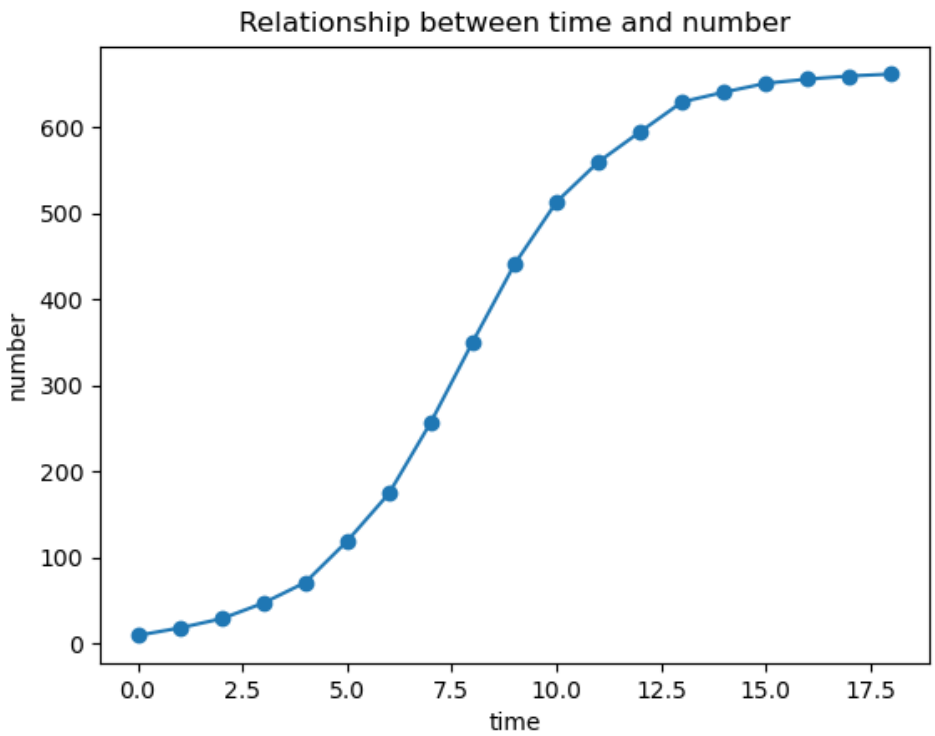
  调整期、对数期、稳定期



- 数据可以也从文件读入，这里直接写了

```python
import matplotlib.pyplot as plt

if __name__ == '__main__':
    time = [i for i in range(0, 19)]
    number = [9.6, 18.3, 29, 47.2, 71.1, 119.1, 174.6,
              257.3, 350.7, 441.0, 513.3, 559.7, 594.8,
              629.4, 640.8, 651.1, 655.9, 659.6, 661.8]
    plt.title('Relationship between time and number')  # 创建标题
    plt.xlabel('time')  # X轴标签
    plt.ylabel('number')  # Y轴标签
```

```python
    plt.scatter(time, number)
    plt.plot(time, number)  # 画图
    plt.show()  # 显示

```



➢ 分析：

酵母菌数量增长有一个这样的规律：当某些资源只能支撑某个最大限度的种群数量，而不能支持种群数量的无限增长，当接近这个最大值时，种群数量的增长速度就会慢下来。

- 1. 两个观测点的值差Δp来表征增长速度
- 2. Δp与目前的种群数量有关，数量越大，增长速度越快
- 3. Δp还与剩余的未分配的资源量有关，资源越多，增长速度越快
- 4. 然后以极限总群数量与现有种群数量的差值表征剩余资源量

➢ 模型：

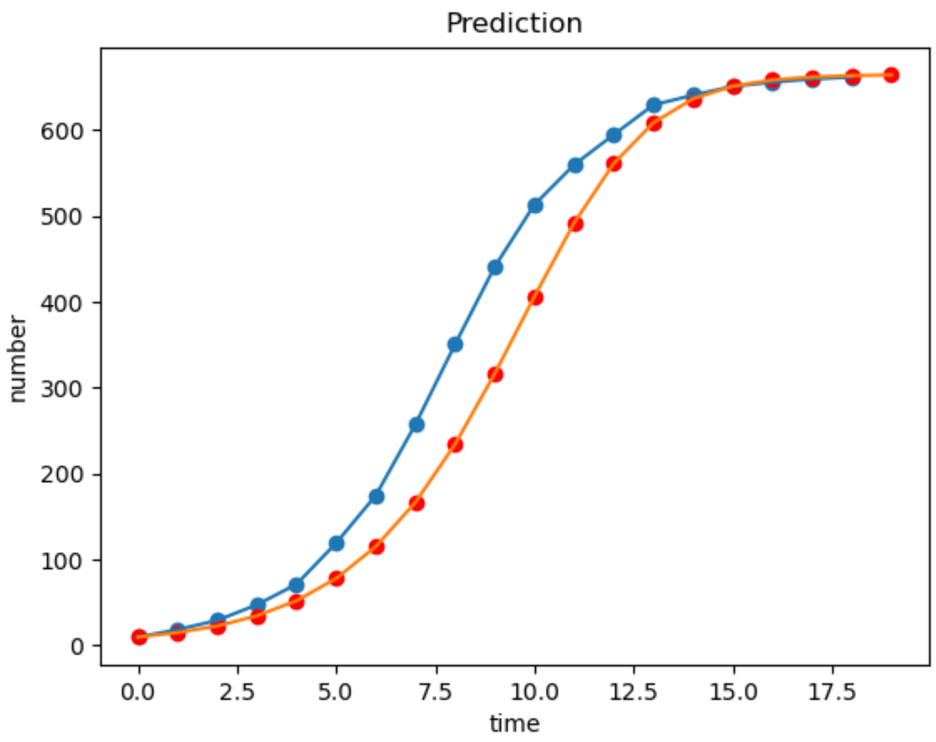
$$\Delta p = p_{n+1} - p_n = k(665 - p_n)p_n$$

- Δp：因为横坐标间隔是1，所以相邻纵坐标之差可以当成增速
- 665是极限总群数量
- 要求的是k,然后预测下一年
- 需要修改的是4，17，18行，如果不只是算下一年，那要改40，46行

```python
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 Max = 665
5
```

```python

# 获取相邻纵坐标的差值
def get_delta(y_num: list):
    delta_y = []
    for i in range(len(y_num) - 1):
        delta_y.append(y_num[i + 1] - y_num[i])

    return delta_y


if __name__ == '__main__':
    time = [_ for _ in range(0, 19)]
    number = [9.6, 18.3, 29, 47.2, 71.1, 119.1, 174.6,
              257.3, 350.7, 441.0, 513.3, 559.7, 594.8,
              629.4, 640.8, 651.1, 655.9, 659.6, 661.8]

    plt.title('Relationship between time and number')  # 创建标题
    plt.xlabel('time')  # X轴标签
    plt.ylabel('number')  # Y轴标签
    plt.scatter(time, number)
    plt.plot(time, number)  # 画图
    # plt.show()  # 显示, 注释掉后, 实际曲线和预测曲线泛在同一个图里面对比

    delta_p = get_delta(number)
    number.pop(-1)
    pn = np.array(number)
    f = pn * (Max - pn)
    res = np.polyfit(f, delta_p, 1)
    print(res)
    print('k = ', res[0])

    # 预测
    p0 = number[0]
    p_list = []
    for i in range(len(time) + 1):
        p_list.append(p0)
        p0 = res[0] * (Max - p0) * p0 + p0
    plt.xlabel('time')  # X轴标签
    plt.ylabel('number')  # Y轴标签
    plt.title('Prediction')  # 创建标题
    plt.scatter([_ for _ in range(0, len(time) + 1)], p_list, c='r')
    plt.plot(p_list)
    plt.show()

```

```
[ 0.00081448 -0.30791574]
k =  0.0008144797937893836

Process finished with exit code 0
```



显式差分

- 热传导方程，见https://www.bilibili.com/video/BV12h411d7Dm?p=8

马尔科夫链

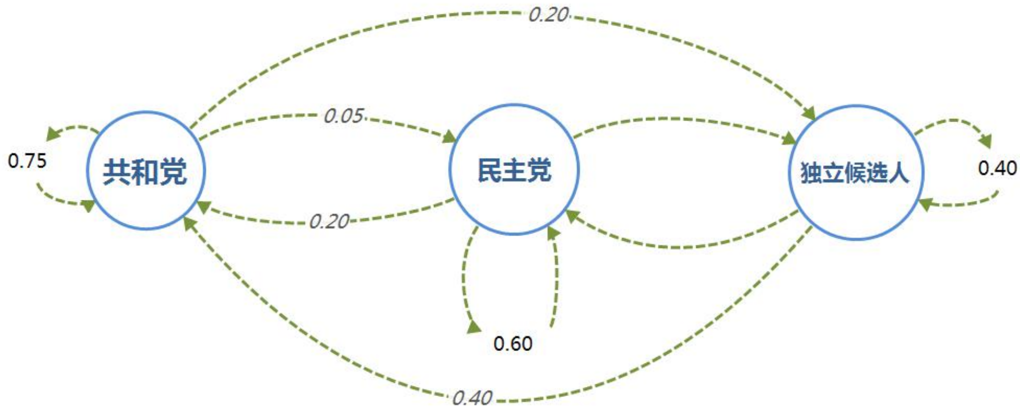
选举投票预测

- 马尔科夫链是由具有以下性质的一系列事件构成的过程：
    - 一个事件有有限多个结果，称为状态，该过程总是这些状态中的一个；
    - 在过程的每个阶段或者时段，一个特定的结果可以从它现在的状态转移到任何状态，或者保持原状；
    - 每个阶段从一个状态转移到其他状态的概率用一个转移矩阵表示，矩阵每行的各元素在0到1之间，每行的和为1。
- 选举投票趋势预测
    - 以美国大选为例，首先取得过去十次选举的历史数据，然后根据历史数据得到选民意向的转移矩阵，转移矩阵如下

下一状态 当前状态	共和党	民主党	独立候选人
共和党	0.75	0.05	0.20
民主党	0.20	0.60	0.20
独立候选人	0.40	0.20	0.40

- 比如，当前状态的共和党转移到下一状态的共和党的概率是0.75，以此类推

  即如下关系：



- 然后我们可以构造出差分表达式(共和党R，民主党D，独立候选人I)：

  也就是下个状态等于前一个状态的所有可能之和

$$
\begin{aligned}
R_{n+1} &= 0.75R_n + 0.20D_n + 0.40I_n \\
D_{n+1} &= 0.05R_n + 0.60D_n + 0.20I_n \\
I_{n+1} &= 0.20R_n + 0.20D_n + 0.40I_n
\end{aligned}
$$

- 通过求解差分方程组，预测出选民投票意向的长期趋势
  - plt.annotate是标记文本，如

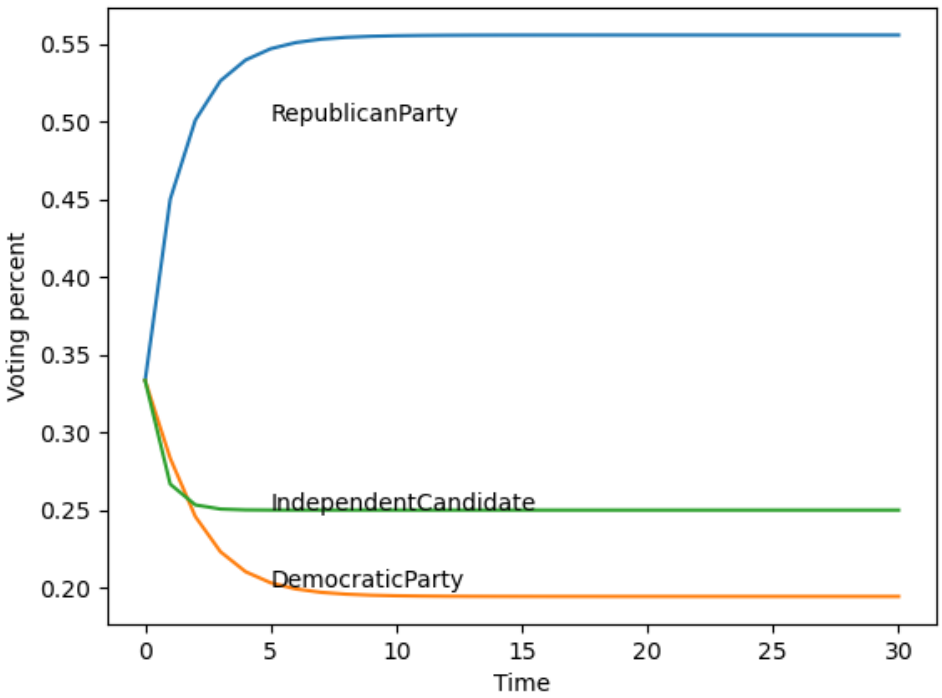
    `plt.annotate('DemocraticParty', xy=(5, 0.2))`中，xy=(a, b)是文字的位置，需要自己多试几次调一下

```python
import matplotlib.pyplot as plt

if __name__ == '__main__':

    RLIST = [1 / 3]
    DLIST = [1 / 3]
    ILIST = [1 / 3]
    for i in range(40):
        R = RLIST[i] * 0.75 + DLIST[i] * 0.20 + ILIST[i] * 0.40
        RLIST.append(R)
        D = RLIST[i] * 0.05 + DLIST[i] * 0.60 + ILIST[i] * 0.20
        DLIST.append(D)
        I = RLIST[i] * 0.20 + DLIST[i] * 0.20 + ILIST[i] * 0.40
        ILIST.append(I)
        plt.plot(RLIST)
        plt.plot(DLIST)
        plt.plot(ILIST)
        plt.xlabel('Time')
```

```
        plt.ylabel('Voting percent')
        plt.annotate('DemocraticParty', xy=(5, 0.2))
        plt.annotate('RepublicanParty', xy=(5, 0.5))
        plt.annotate('IndependentCandidate', xy=(5, 0.25))
        plt.show()
        print(RLIST, DLIST, ILIST)

    print('预测的最后一年：RLIST: {}, DLIST: {}, ILIST: {}'.format(RLIST[-1],
DLIST[-1], ILIST[-1]))

```

- 遍历画出每一年，这是最后一年的图

```
......
......
预测的最后一年：RLIST: 0.5555555555483689, DLIST: 0.1944444444516318, ILIST:
0.2500000000000002

Process finished with exit code 0
```



- 最后得到的长期趋势是：

  56%的人选共和党、

  19%的人选民主党、

  25%的人选独立候选人

灰色与模糊

多层模糊评价

- 见相关资料

模糊c均值聚类

- 原理见资料
- 聚类的算法实现参考我之前sklearn里面的聚类算法实现就行了

灰色预测(经典常用)

- 灰色预测是用灰色模式GM(1,1)来进行定量分析的，通常分为以下几类：
  - 灰色时间序列预测。用等时距观测到的反映预测对象特征的一系列数量（如产量、销量、人口数量、存款数量、利率等）构造灰色预测模型，预测未来某一时刻的特征量，或者达到某特征量的时间。
  - 畸变预测（灾变预测）。通过模型预测异常值出现的时刻，预测异常值什么时候出现在特定时区内。
  - 波形预测，或拓扑预测，通过灰色模型预测事物未来变动的轨迹。
  - 系统预测，对系统行为特征指标建立一族相互关联的灰色预测理论模型，在预测系统整体变化的同时，预测系统各个环节的变化。

- 上述灰色预测方法的共同特征是：
  - **允许少数据预测**；
  - 允许对灰因果律实践进行预测，例如：
    - 灰因白果律事件：粮食生产预测(就是结果产量是已知的，中间受什么因素影响是未知的)
    - 白因灰果律事件：开放项目前景预测(过程已知，但是结果前景未知)
  - 具有可检验性（事前检验：建模可行性级比检验；模型检验：建模精度检验；

  预测检验：预测滚动检验）

- 模型理论部分见资料或者网上看

算法步骤

- 要使用灰色预测模型，首先看看适不适用
- 如果级比都落在可容覆盖范围内，就直接用

  否则做平移变换

（1）数据的级比检验

为了保证灰色预测的可行性，需要对原始序列数据进行级比检验。

对原始数据列

$$X^{(0)} = \left(x^{(0)}(1), x^{(0)}(2), \cdots, x^{(0)}(n)\right)$$

计算序列的级比：

$$\lambda(k) = \frac{x^0(k-1)}{x^0(k)}, k = 2, \cdots, n$$

若所有的级比$\lambda(k)$都落在可容覆盖$\Theta = \left(e^{-2/(n+1)}, e^{2/(n+2)}\right)$内，则可进行灰色预测；否则需要对$X^{(0)}$做平移变换，$Y^{(0)} = X^{(0)} + c$，使得$Y^{(0)}$满足级比要求。

（2）建立GM（1,1）模型，计算出预测值列。

（3）检验预测值：

①相对残差检验，计算

$$\varepsilon(k) = \frac{x^{(0)}(k+1)}{x^{(0)}(k)}, \quad k = 2, \cdots, n$$

若$\varepsilon(k) < 0.2$，则认为达到一般要求，若$\varepsilon(k) < 0.1$，则认为达到较高要求；

②级比偏差值检验

根据前面计算出来的级比 和发展系数 ，计算相应的级比偏差：

若 ，则认为达到一般要求，若 ，则认为达到较高要求。

（4）利用模型进行预测

- 代码部分是可以使用cuda加速的
- 只需改87行的输入数据和93行的预测个数m

```
1  import torch as th
2  import numpy as np
3
4
5  class GM:
6      def __init__(self):
7          # 判断是否可用 gpu 编程 , 大量级计算使用GPU
8          self._is_gpu = False  # th.cuda.is_available()
9
10     def fit(self, dt: list or np.ndarray):
11         self._df: th.Tensor = th.from_numpy(np.array(dt, dtype=np.float32))
12         if self._is_gpu:
13             self._df.cuda()
14         self._n: int = len(self._df)
15         self._x, self._max_value = self._sigmod(self._df)
16         z: th.Tensor = self._next_to_mean(th.cumsum(self._x, dim=0))
17         self.coef: th.Tensor = self._coefficient(self._x, z)
18         del z
19         self._x0: th.Tensor = self._x[0]
20         self._pre: th.Tensor = self._pred()
21
```

```python
    # 归一化
    def _sigmod(self, x: th.Tensor):
        _maxv: th.Tensor = th.max(x)
        return th.div(x, _maxv), _maxv

    # 计算紧邻均值数列
    def _next_to_mean(self, x_1: th.Tensor):
        z: th.Tensor = th.zeros(self._n - 1)
        if self._is_gpu:
            z.cuda()
        for i in range(1, self._n):  # 下标从0开始，取不到最大值
            z[i - 1] = 0.5 * x_1[i] + 0.5 * x_1[i - 1]
        return z

    # 计算系数 a,b
    def _coefficient(self, x: th.Tensor, z: th.Tensor):
        B: th.Tensor = th.stack((-1 * z, th.ones(self._n - 1)), dim=1)
        Y: th.Tensor = th.tensor(x[1:], dtype=th.float32).reshape((-1, 1))
        if self._is_gpu:
            B.cuda()
            Y.cuda()

        # 返回的是a和b的向量转置，第一个是a 第二个是b；
        return th.matmul(th.matmul(th.inverse(th.matmul(B.t(), B)), B.t()), Y)

    def _pred(self, start: int = 1, end: int = 0):
        les: int = self._n + end
        resut: th.Tensor = th.zeros(les)

        if self._is_gpu:
            resut.cuda()
        resut[0] = self._x0
        for i in range(start, les):
            resut[i] = (self._x0 - (self.coef[1] / self.coef[0])) * \
                       (1 - th.exp(self.coef[0])) * th.exp(-1 * self.coef[0] * (i))
        del les
        return resut

    # 计算绝对误差
    def confidence(self):
        return round((th.sum(th.abs(th.div((self._x - self._pre), self._x))) / self._n).item(), 4)

    # 预测个数，默认个数大于等于0,
    def predict(self, m: int = 1, decimals: int = 4):
        y_pred: th.Tensor = th.mul(self._pre, self._max_value)
        y_pred_ = th.zeros(1)
        if m < 0:
            return "预测个数需大于等于0"
        elif m > 0:
```

```python
            y_pred_: th.Tensor = self._pred(self._n, m)[-m:].mul(self._max_value)
        else:
            if self._is_gpu:
                return list(map(lambda _: round(_, decimals), y_pred.cpu().numpy().tolist()))
            else:
                return list(map(lambda _: round(_, decimals), y_pred.numpy().tolist()))

        # cat 拼接 0 x水平拼接，1y垂直拼接
        result: th.Tensor = th.cat((y_pred, y_pred_), dim=0)
        del y_pred, y_pred_
        if self._is_gpu:
            return list(map(lambda _: round(_, decimals), result.cpu().numpy().tolist()))
        return list(map(lambda _: round(_, decimals), result.numpy().tolist()))


if __name__ == "__main__":
    ls = np.arange(91, 100, 2)  # ls是原始的值
    print(type(ls))
    gm = GM()
    gm.fit(ls)
    print('绝对误差: ', gm.confidence())
    print('原始值: ', ls)
    print('预测: ', gm.predict(m=2))  # m是2代表要预测后面两个值

```

```
<class 'numpy.ndarray'>
绝对误差:  0.0002
原始值:  [91 93 95 97 99]
预测:  [91.0, 93.0178, 94.9758, 96.9751, 99.0164, 101.1007, 103.2289]

Process finished with exit code 0
```

蒙特卡罗

蒙特卡罗算法

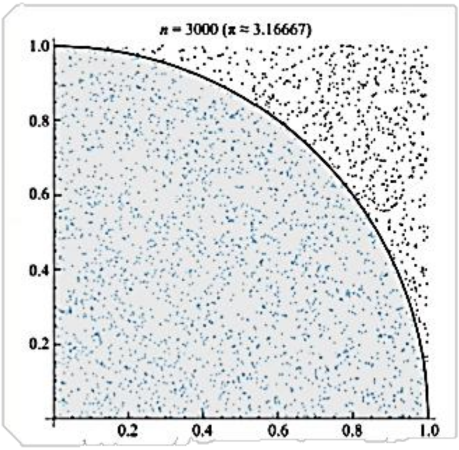
- 由冯.诺依曼提出来的
- 蒙特·卡罗(Monte Carlo method)又称统计模拟方法，一种以概率统计理论为指导的数值计算方法。是指使用随机数(或者伪随机数)来解决很多计算问题的方法。
- 基本思想

  当所求解问题是某种随机事件出现的概率，或者是某个随机变量的期望值时，通过某种“实验”的方法，以这种事件出现的频率估计这一随机事件的概率，或者得到这个随机变量的某些数字特征，并将其作为问题的解。

举例

- **蒙特卡罗方法求圆周率**

  基本思想：在图中区域产生足够多的随机数点，然后计算落在圆内的点的个数与总个数的比值再乘以4，就是圆周率。



- random.random()得到的是0到1之间的(伪)随机数，若要某个整数范围，则用random.randint(a, b)，是有包括a和b的

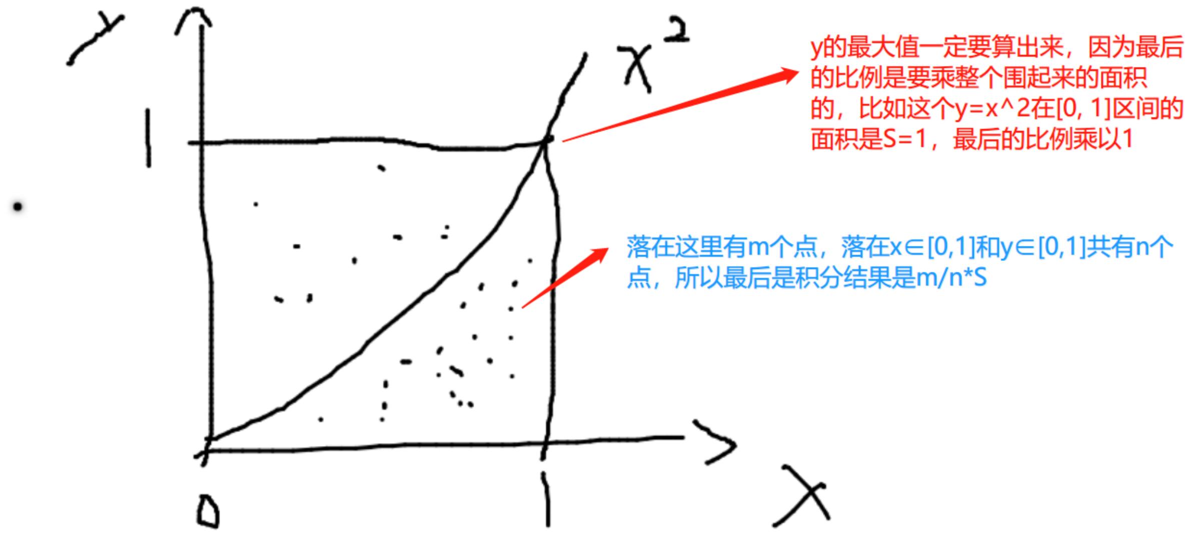
```python
import math
import random

if __name__ == '__main__':

    M = input('请输入一个较大的整数')
    N = 0
    for i in range(int(M)):
        x = random.random()
        y = random.random()
        if math.sqrt(x ** 2 + y ** 2) < 1:
            N += 1
            pi = 4 * N / int(M)
            # print(pi)
    print(pi)

```

```
请输入一个较大的整数9999999
3.1417339141733915

Process finished with exit code 0
```

- **蒙特卡罗求定积分**

  利用python计算函数y=x**2在[0,1]区间的定积分

基本思想：和上例相似




```python
"""
求f(x) = x^2的定积分
"""

import random

if __name__ == '__main__':
    n = int(input('请输入一个较大的整数:'))
    m = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if y < x ** 2:  # 找到落在f(x)下面的点
            m += 1
    R = m / n # 这里因为总面积是1所以省略乘以1了
    print(R)

```

```
请输入一个较大的整数:999999
0.33292533292533294

Process finished with exit code 0
```

三门问题

- 背景

三门问题（Monty Hall probelm）亦称为蒙提霍尔问题，出自美国电视游戏节目Let’s Make a Deal。参赛者会看见三扇关闭了的门，其中一扇的后面有一辆汽车，选中后面有车的那扇门可赢得该汽车，另外两扇门则各藏有一只山羊。当参赛者选定了一扇门，但未去开启它的时候，节目主持人开启剩下两扇门的其中一扇，露出其中一只山羊。主持人其后问参赛者要不要换另一扇仍然关上的门。问题是：换另一扇门是否会增加参赛者赢得汽车的几率？如果严格按照上述条件，即主持人清楚地知道，自己打开的那扇门后面是羊，那么答案是会。不换门的话，赢得汽车的几率是1/3,换门的话，赢得汽车的几率是2/3

- 应用蒙特卡罗重点在使用随机数来模拟类似于赌博问题的赢率问题，通过多次模拟得到所要计算值的模拟值
- 解决思路：

  在三门问题中，用0、1、2分代表三扇门的编号，在[0,2]之间随机生成一个整数代表奖品所在门的编号prize，再次在[0,2]之间随机生成一个整数代表参赛者所选择的门的编号guess。用变量change代表游戏中的换门(true)与不换门(false)


```python
import random


def play(change):
    prize = random.randint(0, 2)
    guess = random.randint(0, 2)
    if prize == guess:
        if change:
            return False
        else:
            return True
    else:
        if change:
            return True
        else:
            return False


def winRate(change, N):
    win = 0
    for i in range(0, N):
        if play(change):
            win = win + 1
```

```python
            # print('中奖率为: ')
            # print(win / N)
    print('中奖率为: ')
    print(win / N)


if __name__ == '__main__':
    N = 100000
    print('玩' + str(N) + '次，每一次都换门:')
    winRate(True, N)
    print()
    print('玩' + str(N) + '次，每一次都不换门:')
    winRate(False, N)

```

```
玩100000次，每一次都换门:
中奖率为:
0.66748

玩100000次，每一次都不换门:
中奖率为:
0.33008

Process finished with exit code 0
```

巧克力豆问题

- 见相关资料

时间序列

- 时序问题也可使用神经网络里面的LSTM(长短时记忆)
- 预测的是近期的，不是预测长远的(长远预测需要挖掘更多特征，深度学习那块的)
- 均方差:

  ```python
  from sklearn.metrics import mean_squared_error
  from math import sqrt
  rms = sqrt(mean_squared_error(test, pred))# 把实际和预测的放进去
  print(rms)
  ```

- 见https://www.bilibili.com/video/BV12h411d7Dm?p=10

简单指数平滑法

- 导入数据
- 切分数据
- 代码适当修改和测试

```python
from statsmodels.tsa.api import SimpleExpSmoothing
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from math import sqrt

if __name__ == '__main__':
    df = pd.read_csv('train.csv')
    train = df[0:10392]
    test = df[10392:]
    pred = test.copy()
    fit = SimpleExpSmoothing(np.asarray(train['列名1'])).fit(smoothing_level=0.6, optimized=False)
    pred['列名2'] = fit.forecast(len(test))  # 需要预测多长

    # 画出来
    plt.figure(figsize=(16, 8))
    plt.plot(train['列名1'], label='Train')
    plt.plot(test['列名1'], label='Test')
    plt.plot(pred['列名2'], label='列名2')
    plt.legend(loc='best')
    plt.show()

    # 评估
    rms = sqrt(mean_squared_error(test['列名1'], pred))
    print(rms)

```

霍尔特线性趋势法

- 考虑到数据集变化趋势的方法就叫做霍尔特线性趋势法，任何呈现某种趋势(比如商上升趋势)的数据集都可以用霍尔特线性趋势法用于预测
- 先看看是否呈现某种趋势

  ```python
  import statsmodels.api as sm sm.tsa.seasonal_decompose(train['xxx']).plot()
  result = sm.tsa.stattools.adfuller(train['xxx'])
  plt.show()
  ```

  - 如果是的话就可以使用
- 代码相对上面只需改动这两句，其他适当改即可

```python
from statsmodels.tsa.api import Holt
fit = Holt(np.asarray(train['xxx'])).fit(smoothing_level=0.3, smoothing_slope=0.1)
```

Holt-Winters季节性预测模型

- 体现在季节性
    - 比如一个水果店的销售情况，在夏季收入远高于其他季节
- 只需改变一点代码,选择了seasonal_period=7作为每周重复数据

```
from statsmodels.tsa.api import ExponentialSmoothing
fit1 = ExponentialSmoothing(np.asarray(train['xxx']), seasonal_periods=7, trend='add', seasonal='add', ).fit()
```

自回归移动平均模型（ARIMA）

- 指数平滑模型都是基于数据中的趋势和季节性的描述，而自回归移动平均模型的目标是描述数据中彼此之间的关系。ARIMA的一个优化版就是季节性ARIMA。它像Holt-Winters季节性预测模型一样，也把数据集的季节性考虑在内。

```
import statsmodels.api as sm
pred = test.copy()
fit1 = sm.tsa.statespace.SARIMAX(train.xxx, order=(2, 1, 4), seasonal_order=(0, 1, 1, 7)).fit() pred['SARIMA']=fit1.predict(start="20xx-xx-xx",end="20xx-xx-xx",dynamic=True)
```

SVM

```
clf = svm.SVC(C=0.8, kernel='rbf', gamma=20, decision_function_shape='ovr')
clf.fit(x_train, y_train.ravel())
```

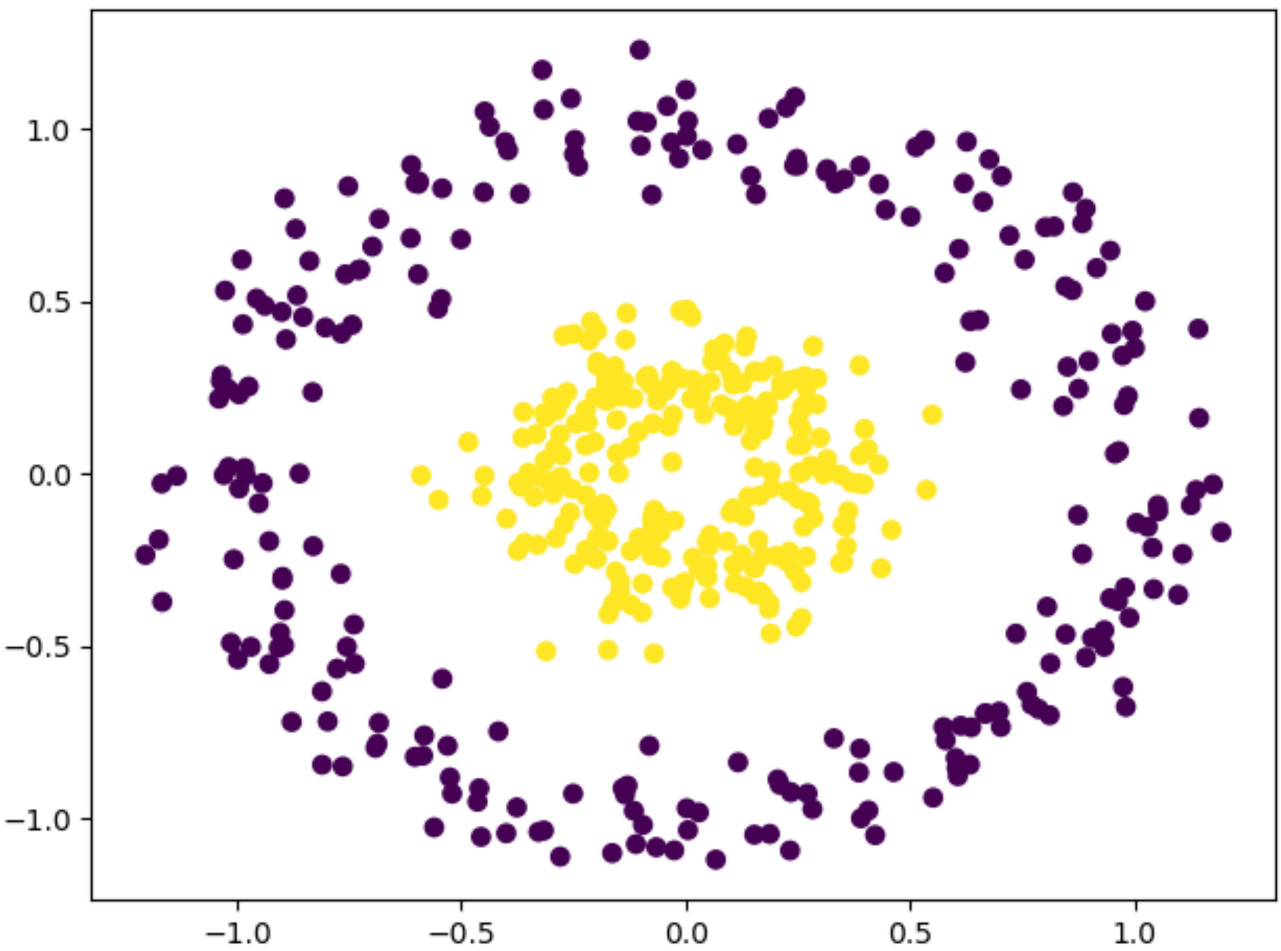
- 调参，参数见资料
- 参考：https://www.bilibili.com/video/BV12h411d7Dm?p=12

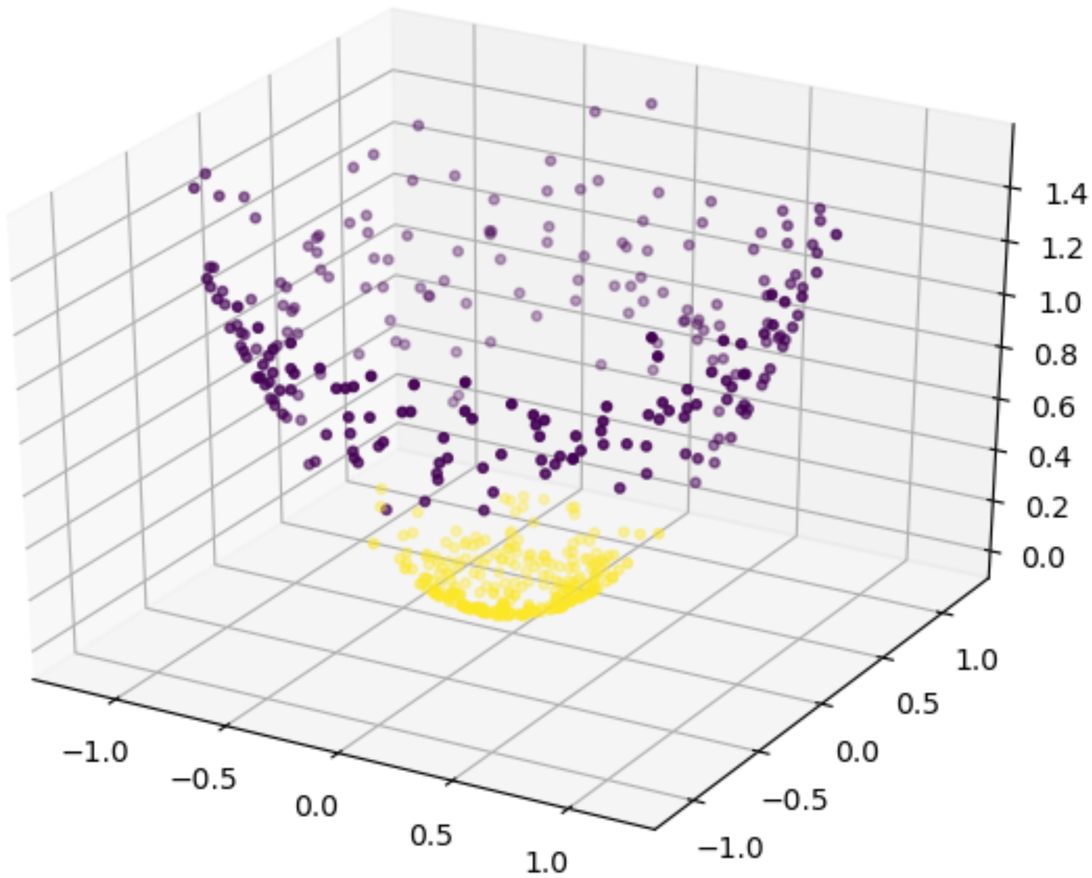
svm处理非线性问题，低维映射到高维

- 如2维转3维，找到切平面，在投影到2维平面，可能是个圆或者椭圆的分界线

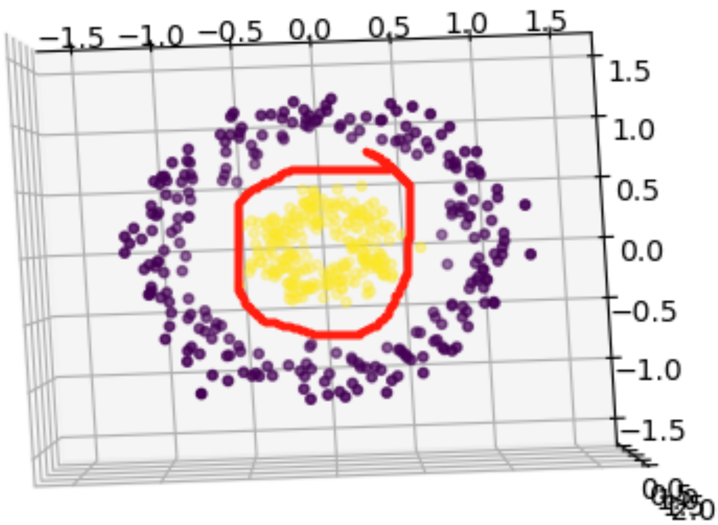
```
""""
# @Time    :  2020/8/22
# @Author  :  Jimou Chen
""""
import matplotlib.pyplot as plt
from sklearn import datasets
```

```python

'''用于解决非线性问题'''

# 制造数据
x_data, y_data = datasets.make_circles(n_samples=500, factor=0.3, noise=0.1)
# 画出来看看
plt.scatter(x_data[:, 0], x_data[:, 1], c=y_data)
plt.show()

'''接下来把2维映射到3维'''

z_data = x_data[:, 0] ** 2 + x_data[:, 1] ** 2
# 画3d图
ax = plt.figure().add_subplot(111, projection='3d')
ax.scatter(x_data[:, 0], x_data[:, 1], z_data, c=y_data, s=10)  # s是大小
plt.show()

```




1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0
1.0
0.5
0.0
−0.5
−1.0

投影到二维平面


−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5

核函数

我们可以构造核函数使得运算结果等同于非线性映射，同时运算量要远远小于非线性映射。

$$K(X_i, X_j) = \phi(X_i) \cdot \phi(X_j)$$

$$h\text{ 次多项式核函数}: K(X_i, X_j) = (X_i \cdot X_j + 1)^h$$

$$\text{高斯径向基函数核函数}: K(X_i, X_j) = e^{-\|X_i - X_j\|^2 / 2\sigma^2}$$

$$\text{S 型核函数}: K(X_i, X_j) = \tanh(\kappa X_i \cdot X_j - \delta)$$

svm推导过程

- 推导过程理解起来较为复杂，见其他机器学习教材或者相关资料

svm处理多分类问题

- 一般svm是处理二值分类问题的，如果处理多个类别的，使用下面三种方法

```
model = svm.SVC(decision_function_shape='ovo')
model = svm.SVC(decision_function_shape='ovr')
model = svm.SVC(probability=True)
```

- 以下是先用pca降维后，再用svm进行分类的例子

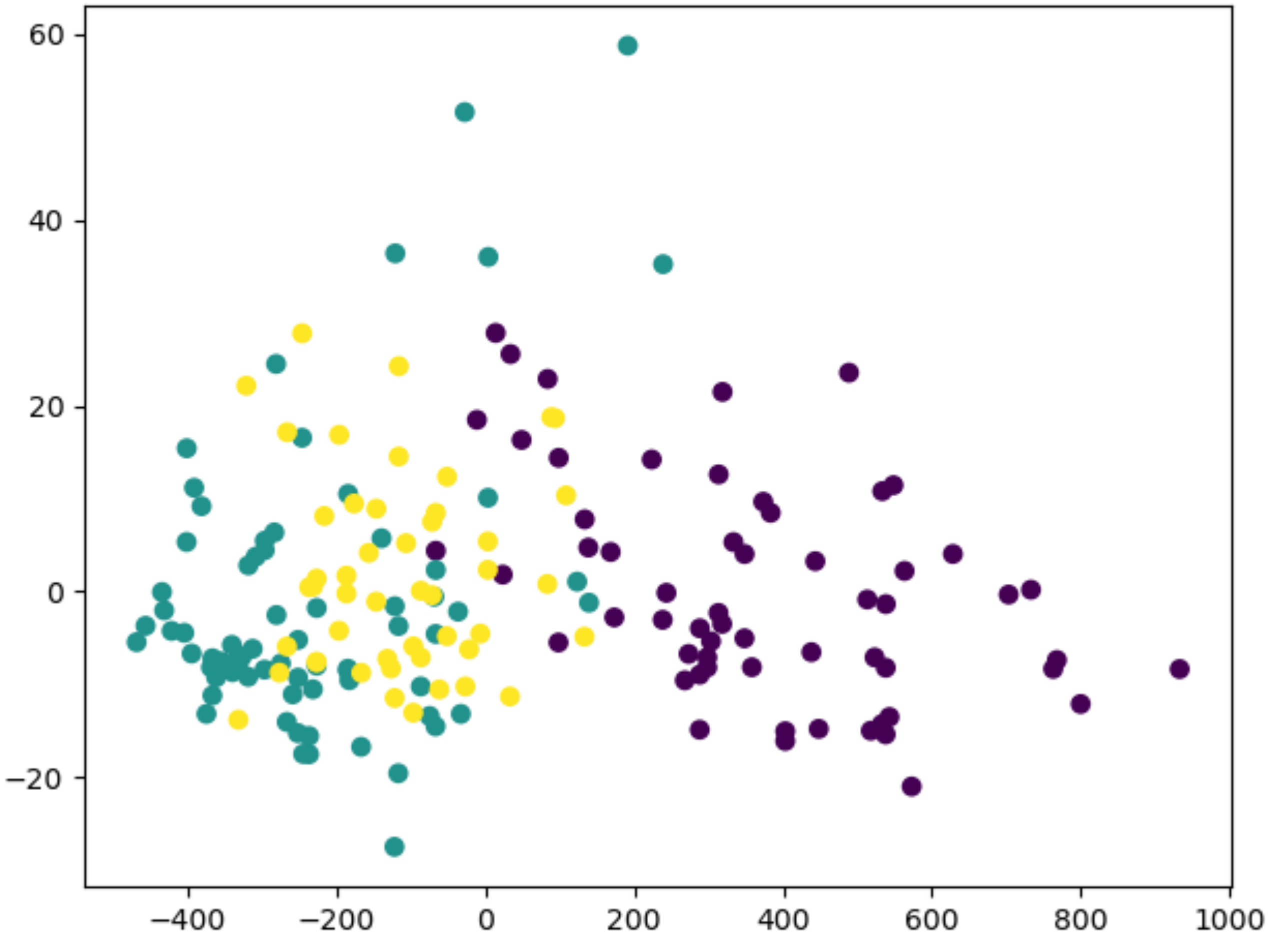
```
"""
# @Time    :  2020/8/30
# @Author  :  Jimou Chen
"""
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import svm
from sklearn.metrics import classification_report


data = pd.read_csv('data/wine.csv')
y_data = data.iloc[:, 0]
x_data = data.iloc[:, 1:]

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data)
pca = PCA(n_components=2)
```

```python
new_data = pca.fit_transform(x_data)

# 画出来看一下
plt.scatter(new_data[:, 0], new_data[:, 1], c=y_data)
plt.show()

# 建模预测,多分类的三种方法，有时候会警告,不影响
# model = svm.SVC(decision_function_shape='ovo')
model = svm.SVC(decision_function_shape='ovr')
# model = svm.SVC(probability=True)
model.fit(x_train, y_train)

prediction = model.predict(x_data)

print(model.score(x_test, y_test))
print(classification_report(y_data, prediction))

# 画出预测的
plt.scatter(new_data[:, 0], new_data[0:, 1], c=prediction)
plt.show()
```

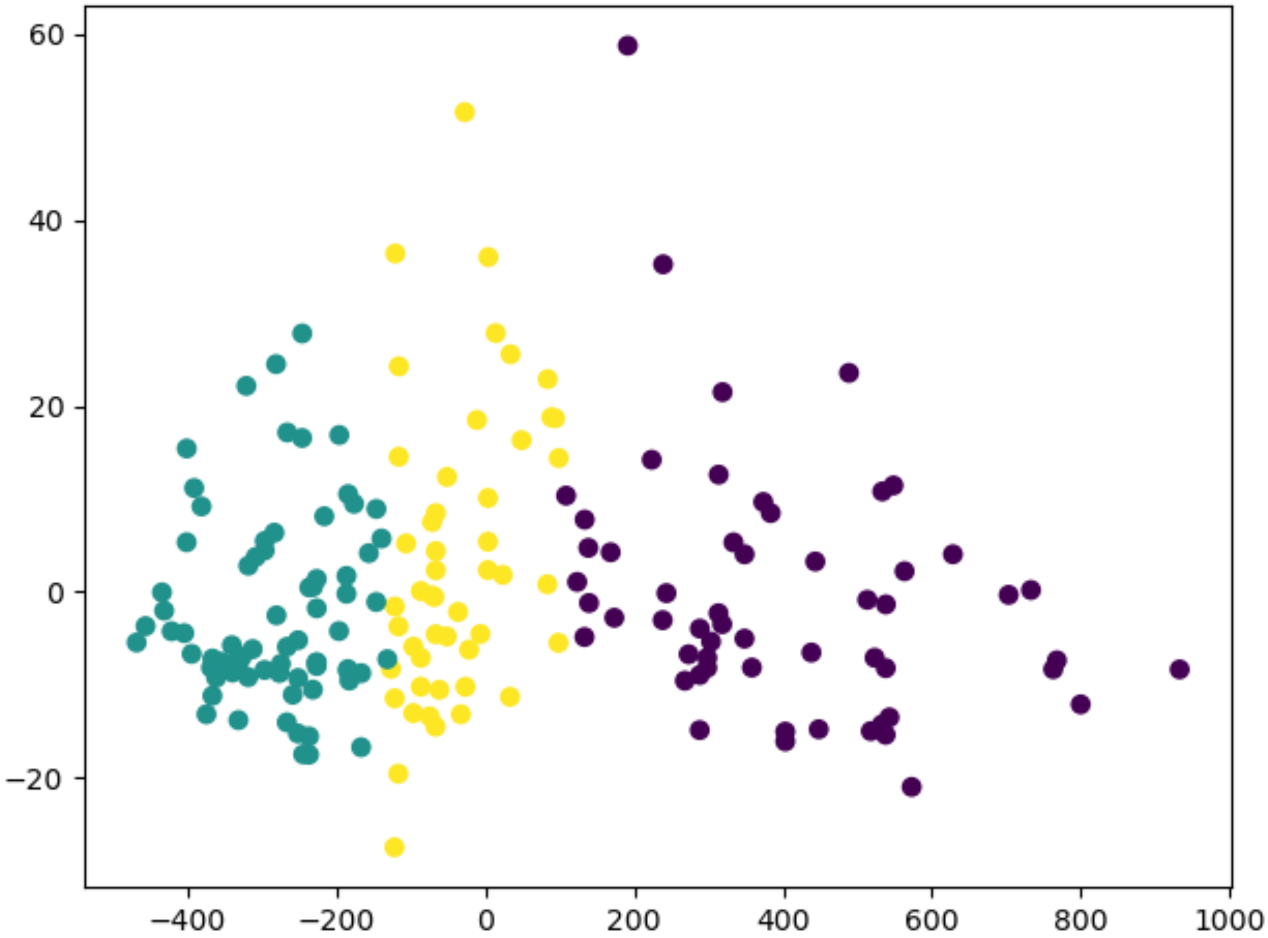
- 结果:

```
0.6888888888888889
              precision    recall  f1-score   support

           1       0.89      0.85      0.87        59
           2       0.70      0.72      0.71        71
           3       0.49      0.50      0.49        48

    accuracy                           0.70       178
   macro avg       0.69      0.69      0.69       178
weighted avg       0.71      0.70      0.70       178


Process finished with exit code 0
```

- **原始的**



- 预测的



- 改进
  上面这个题用随机森林的效果会更好，无论降到2/3/5维，预测效果几乎完美
  `model = RandomForestClassifier(n_estimators=100)`

聚类

K-Means模型

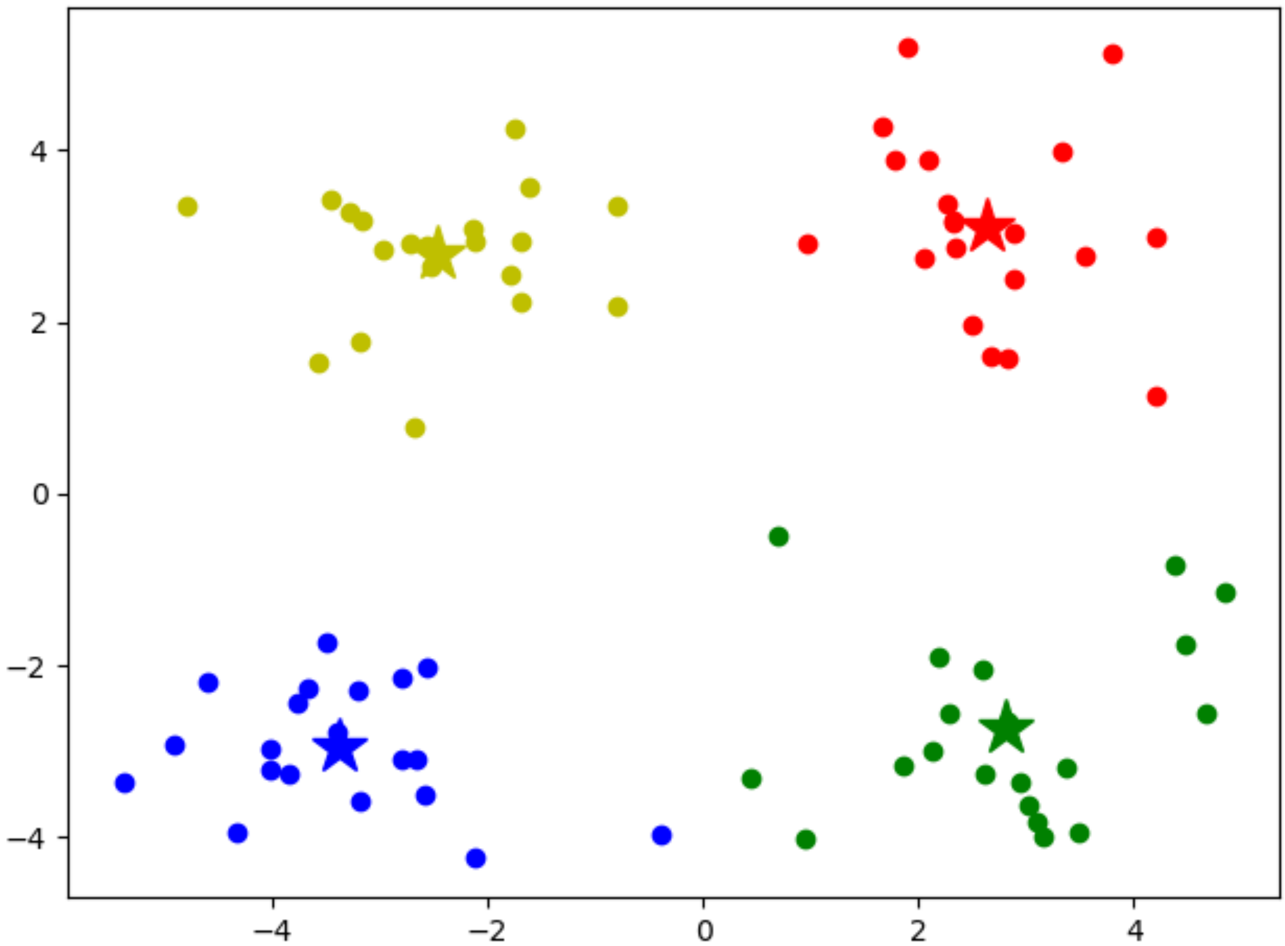
- 算法思想：以空间中k个点为中心进行聚类，对最靠近他们的对象归类。通过迭代的方法，逐次更新各聚类中心的值，直至得到最好的聚类结果
- 跟分类相比，没有给定已知标签
- 用sklearn自带的KMeans效果要比自己实现的Kmeans效果好
- 用sklearn实现如下，以下是以二维为例子
  - 修改第6行的文件文件即可，kmeans.txt的格式见：
    - https://wwr.lanzoui.com/iykw2re7qxc
      密码:hmse
  - 这里的文件可以灵活处理，如果是csv格式的需要稍微处理

```
import numpy as np
```

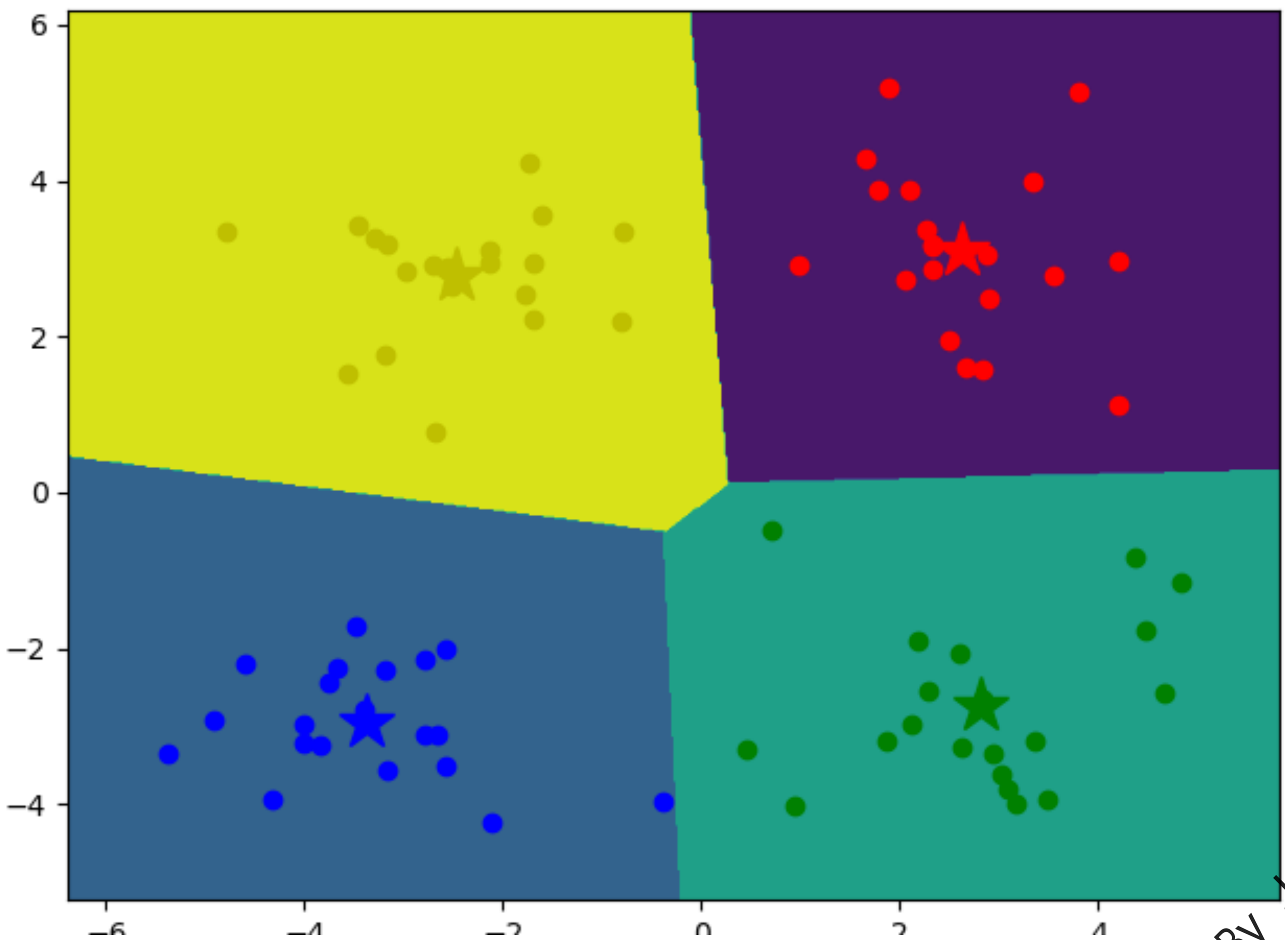
```python
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans  # 导入KMeans

# 载入数据
data = np.genfromtxt('kmeans.txt', delimiter=' ')
# 设置k值
k = 4

# 建模
model = KMeans(n_clusters=k)
model.fit(data)

# 各分类的中心点坐标
centers = model.cluster_centers_
print(centers)

# 预测结果
prediction = model.predict(data)
print(prediction)

# 画图
colors = ['or', 'ob', 'og', 'oy']
for i, d in enumerate(data):
    plt.plot(d[0], d[1], colors[prediction[i]])

# 画出各个分类的中心点
mark = ['*r', '*b', '*g', '*y']
for i, center in enumerate(centers):
    plt.plot(center[0], center[1], mark[i], markersize=20)

plt.show()

# 获取数据值所在的范围
x_min, x_max = data[:, 0].min() - 1, data[:, 0].max() + 1
y_min, y_max = data[:, 1].min() - 1, data[:, 1].max() + 1

# 生成网格矩阵
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))

z = model.predict(np.c_[xx.ravel(), yy.ravel()])  # ravel与flatten类似，多维数据转一维。flatten不会改变原始数据，ravel会改变原始数据
z = z.reshape(xx.shape)
# 等高线图
cs = plt.contourf(xx, yy, z)
# 显示结果
# 画出各个数据点，用不同颜色表示分类
mark = ['or', 'ob', 'og', 'oy']
for i, d in enumerate(data):
    plt.plot(d[0], d[1], mark[prediction[i]])
```

```python

# 画出各个分类的中心点
mark = ['*r', '*b', '*g', '*y']
for i, center in enumerate(centers):
    plt.plot(center[0], center[1], mark[i], markersize=20)

plt.show()

```

```
[[ 2.6265299   3.10868015]
 [-3.38237045 -2.9473363 ]
 [ 2.80293085 -2.7315146 ]
 [-2.46154315  2.78737555]]
[0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0
 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0 3
 2 1 0 3 2 1]

Process finished with exit code 0
```



4
2
0
−2
−4
−4
−2
0
2
4



6
4
2
0
−2
−4

> model.fit(data) 只传一个

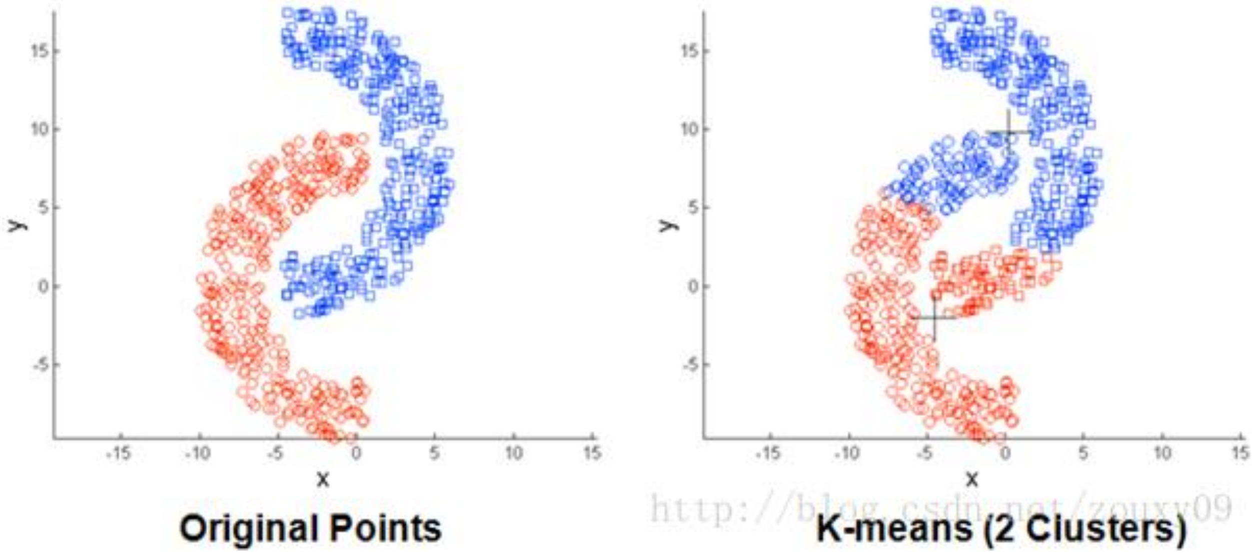
MiniBatchKMeans模型

- Mini Batch K-Means算法是K-Means算法的变种
- 与K均值算法相比，数据的更新是在每一个小的样本集上。Mini Batch K-Means比K-Means有更快的 收敛速度，但同时也降低了聚类的效果，但是在实际项目中却表现得不明显

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MiniBatchKMeans

'''适用于数据多的情况，但是一般情况还是用KMeans'''

data = np.genfromtxt('kmeans.txt', delimiter=' ')
k = 4

model = MiniBatchKMeans(n_clusters=k)
model.fit(data)

# 分类中心坐标
centers = model.cluster_centers_
print(centers)

# 预测结果
pred_res = model.predict(data)
print(pred_res)

# 画图
colors = ['or', 'ob', 'og', 'oy']
for i, d in enumerate(data):
    plt.plot(d[0], d[1], colors[pred_res[i]])

# 画出各个分类的中心点
mark = ['*r', '*b', '*g', '*y']
for i, center in enumerate(centers):
    plt.plot(center[0], center[1], mark[i], markersize=20)

plt.show()

```

- KMeans的4个问题，可参考解释： https://www.bilibili.com/video/BV1Rt411q7WJ?p=65
  - 对k个初始质心的选择比较敏感，容易陷入局部最小值
  - k值的选择是用户指定的，不同的k得到的结果会有挺大的不同

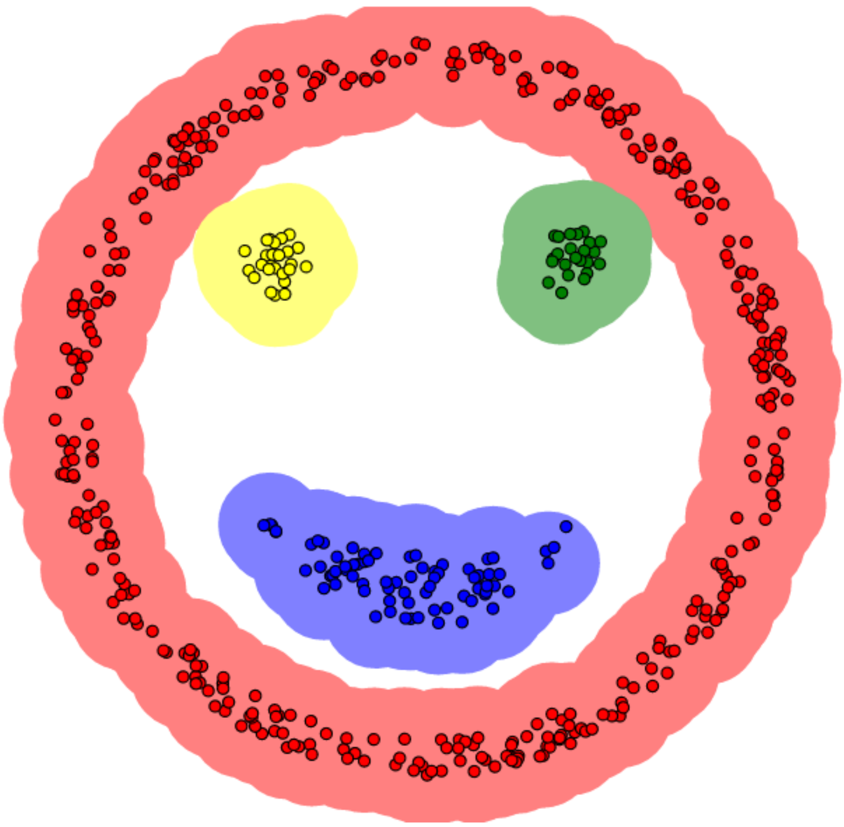
- 存在局限性，如下面这种非球状的数据分布就搞不定了



- 数据比较大的时候，收敛会比较慢

DBSCAN模型

- 本算法将具有足够高密度的区域划分为簇，并可以发现任何形状的聚类

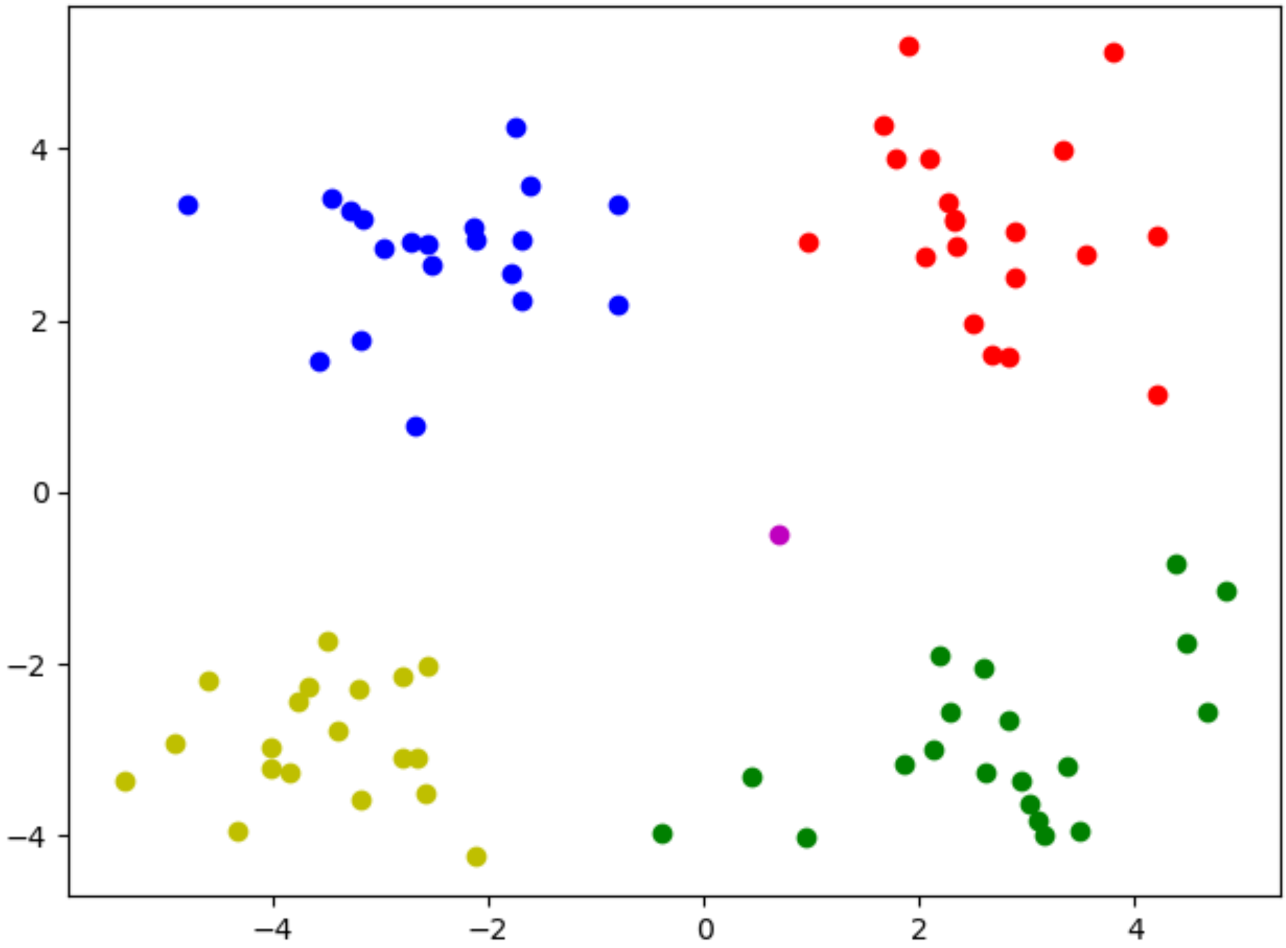


- 算法思想： https://www.bilibili.com/video/BV1Rt411q7WJ?p=69
- 缺点：
  - 当数据量增大时，要求较大的内存支持I/O消耗也很大。

    - 当空间聚类的密度不均匀、聚类间距差相差很大时，聚类质量较差。
- DBSCAN和K-MEANS比较：
    - DBSCAN不需要输入聚类个数。
    - 聚类簇的形状没有要求。
    - 可以在需要时输入过滤噪声的参数。
- 关键参数
    - esp
    - min_point

> 即通过调整eps=，min_samples=来找到一个最好的效果

```python
from sklearn.cluster import DBSCAN
import numpy as np
import matplotlib.pyplot as plt

data = np.genfromtxt('kmeans.txt', delimiter=' ')
# 建模
# esp距离阈值，min_samples在esp领域里面的样本数
model = DBSCAN(eps=1.5, min_samples=4)
model.fit(data)

# 预测,fit_predict是先拟合后预测，DBSCAN没有predict方法
pred = model.fit_predict(data)
print(pred)  # 预测值为-1的是噪点

# 画出各个数据点
mark = ['or', 'ob', 'og', 'oy', 'ok', 'om']
for i, d in enumerate(data):
    plt.plot(d[0], d[1], mark[pred[i]])
plt.show()

```




```
[ 0  1  2  3  0  1  2  3  0  1  2  3  0  1 -1  2  0  1  2  3  0  1  2  3
  0  1  2  3  0  1  2  3  0  1  2  3  0  1  2  3  0  1  2  3  0  1  2  3
  0  1  2  3  0  1  2  3  0  1  2  3  0  1  2  3  0  1  2  3  0  1  2  3
  0  1  2  3  0  1  2  3]
```

- 预测值是-1 的代表噪点