Lab2_report

姓名：徐宇鸣

学号：PB17111636

实验内容

P1:监督学习问题--学生表现预测

main部分:

主要做的预处理方面，测试集和训练集的分割就是简单地一刀切3:7，在main部分主要就是初始化和预处理了一些参数，并没有做训练集测试集划分随机化

读取数据，将标签和数据分开，并且决定前多少个为训练集

```
def Preprocessing(Path,train_factor):
  allset = []
  labelset = []
```

```
  with open(Path,encoding='utf-8')as dataset:
    data = csv.reader(dataset,delimiter=';')
    count = 0;
    print("preprocessing")
    for row in data :
      count+=1
      if count == 1 :
        continue
      #print(row)
      temp = []
      ###preprocessing
      for appendnum in appendlist :
        if codelists[appendnum] :
          ###need to transform
          temp_series = pd.Series(row[appendnum])
          #print(temp_series)
          temp.append(list(Le_list[appendnum].transform(temp_series))[0])
        else :
          ###needn't
          temp.append(int(row[appendnum]))
      if int(row[32]) >= 10 :
        labelset.append(1)
      else :
        labelset.append(-1)
      #print(temp)
      allset.append(temp)
    setlen = len(allset)
    trainlen = setlen//10*train_factor
  return allset,labelset,setlen,trainlen
```

对于字符类数据就简单地用labelencoder进行标记，在这里先用一个列表存入了所有的划分:

```
def Init():
  for codelist in codelists :
    le = preprocessing.LabelEncoder()
    if codelist :
      le = le.fit(codelist)
    else :
      le = None
    Le_list.append(le)
```

除此之外就是为决策树准备的attrlist，主要是记录每个属性有多少个值，这里就是纯粹手造，没有设计函数

KNN部分：

算法介绍：

K近邻算法要做的就是对每一个测试集中的样本点，计算其与训练集的样本差距，然后排序获得差距最小的K的点，根据他们的结果判断测试样本点的分类

使用欧式距离：

```
def calculate_distance(lista,listb,len):
  #using eural distance
  total = 0;
  for i in range(0,len) :
    total += (lista[i] - listb[i])**2
  total = math.sqrt(total)
  return total
```

在这里我选择简化了输入，将训练集和测试集在一个list里面作为输入，而训练集和测试集的划分也是简单的一刀切，并没有进行随机划分，由训练集长度和样本总长度来划分训练集和测试集

函数的结果就是返回了K从1-20的情况下的结果

```
def KNN_lab2(allset,labelset,trainlen,alllen):
  count = trainlen;
  result = []
  result_without = []
  while count < alllen :
    rank = {}
    rank_without = {}
    for i in range(0,trainlen) :
      #calculate the difference of the count and the first [0,trainlen]
examples
      #include with G1 G2 and without G1,G2
      dist = calculate_distance(allset[i],allset[count],len(allset[i]))
      dist_without =
calculate_distance(allset[i],allset[count],len(allset[i])-2)
      rank[i] = dist
      rank_without[i] = dist_without
    #sort the dict with the distance
    ranksort = sorted(rank.items(),key=lambda rank:rank[1],reverse=False)
    ranksort_without = sorted(rank_without.items(),key=lambda
rank_without:rank_without[1],reverse=False)
    temp = []
    #find the result with k from 1 to 20
    for k in range(1,21):
      ###different result of k
      passlen = 0
      for m in range(0,k):
        if labelset[ranksort[m][0]] == 1 :
          passlen += 1
      if passlen > k//2 :
        temp.append(1)
      else :
```

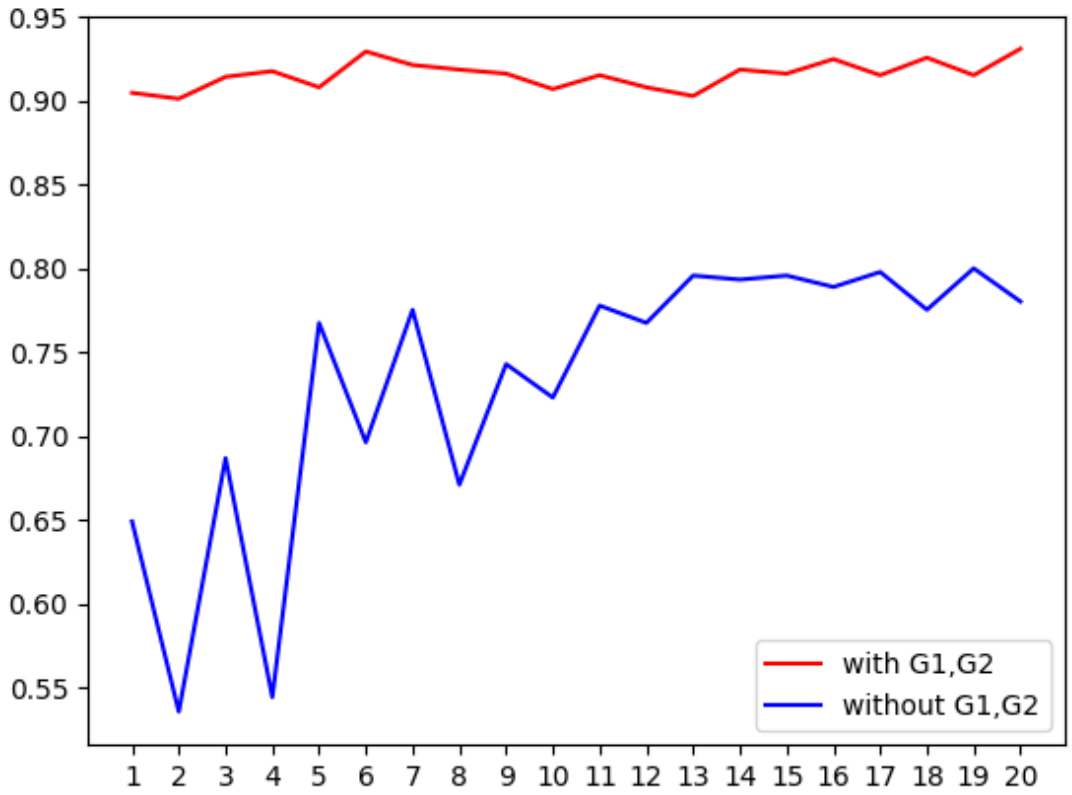
```
            temp.append(-1)
        result.append(temp)
        temp = []
        for k in range(1,21):
          ###different result of k
          passlen = 0
          for m in range(0,k):
            if labelset[ranksort_without[m][0]] == 1 :
              passlen += 1
          if passlen > k//2 :
            temp.append(1)
          else :
            temp.append(-1)
        result_without.append(temp)
        count += 1
    return result,result_without
```

算法结果：

首先是对mat预测的结果：

K	带G1、G2	不带G1、G2
1	0.9047619047619048	0.6490066225165563
2	0.9012345679012346	0.5354330708661417
3	0.9142857142857143	0.6867469879518073
4	0.9176470588235294	0.5441176470588236
5	0.9080459770114941	0.7674418604651162
6	0.9294117647058824	0.6962025316455697
7	0.9213483146067416	0.7752808988764045
8	0.9186046511627908	0.6708860759493671
9	0.9162011173184357	0.7428571428571428
10	0.9069767441860465	0.7228915662650603
11	0.9152542372881357	0.7777777777777778
12	0.9080459770114941	0.7674418604651162
13	0.9028571428571429	0.7956989247311828
14	0.9186046511627908	0.7932960893854748
15	0.9162011173184357	0.7956989247311828
16	0.9248554913294799	0.7888888888888889
17	0.9152542372881357	0.7978142076502733
18	0.9257142857142857	0.7752808988764045
19	0.9152542372881357	0.7999999999999999
20	0.9310344827586207	0.7802197802197801

得到的结果由matplotlib画图得到：

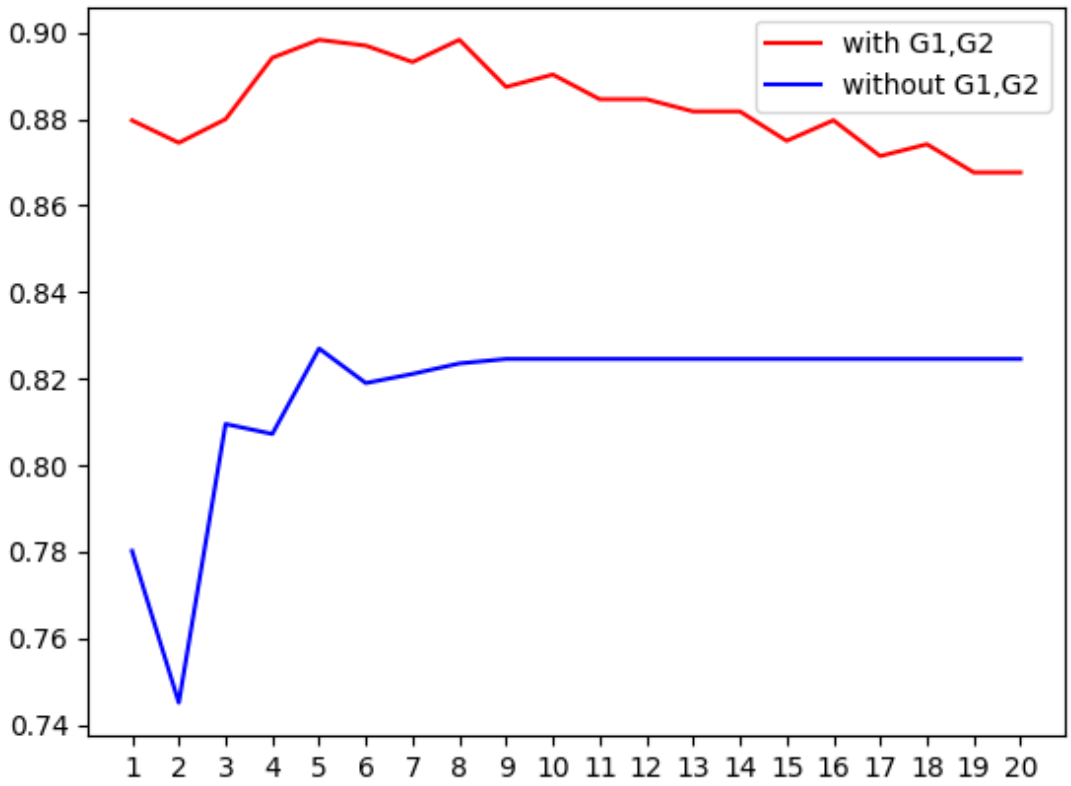


可以看到当对于这样的数据集，K近邻在1-20的情况下都还还是保持着F1值有上涨的趋势，对于不含G1、G2的预测最好的效果在K=19，对于含G1、G2最好的效果在K=20

接着是对por的结果：

K	带G1、G2	不带G1、G2
1	0.8797250859106529	0.7801857585139319
2	0.8745519713261649	0.7450980392156863
3	0.88	0.8095238095238094
4	0.89419795221843	0.8072289156626506
5	0.8983606557377048	0.8269794721407624
6	0.8970099667774086	0.8189910979228486
7	0.8932038834951456	0.8211143695014663
8	0.8983606557377048	0.823529411764706
9	0.8874598070739549	0.8245614035087719
10	0.8903225806451612	0.8245614035087719
11	0.8846153846153846	0.8245614035087719
12	0.8846153846153846	0.8245614035087719
13	0.8817891373801916	0.8245614035087719
14	0.8817891373801916	0.8245614035087719
15	0.875	0.8245614035087719
16	0.879746835443038	0.8245614035087719
17	0.8714733542319749	0.8245614035087719
18	0.8742138364779874	0.8245614035087719
19	0.8676923076923077	0.8245614035087719
20	0.8676923076923077	0.8245614035087719

得到的结果由matplotlib画图得到：



可以看到对于这样的训练集+测试集，对于含G1，G2的最好的结果在K=8，接下来就在逐渐下降，而对于不含G1，G2的结果，最好的结果在K=5，有趣的是当K=9直到K=20所有的F1值都相等

SVM部分：

算法介绍

SVM算法所需要做的就是找到这么一个超平面，使得：

$$min_{w,b}(\frac{1}{2}w^T w) + C\sum_i \xi_i, \forall i, y_i(w^T x_i + b) \geq 1 - \xi_i, \xi \geq 0$$

而在使用了拉格朗日函数之后，最终其实要求的就是获得如下的凸二项规划：

$$max_a \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

$$\forall i, 0 \leq \alpha \leq C, \sum_{i=1}^{n} \alpha_i y_i = 0$$

在不允许调库的情况下，我选择使用smo算法来解决这个问题，smo算法简单来说其实就是每轮迭代的时候只改变一个变量，对该变量求偏导，得到极值，而对于这个凸二次规划，因为存在限制，所以我们需要每次选择一对变量，在这里我根据John Platt的论文中实现了带启发式选择变量对的方法。

伪代码

以下首先是smo算法在论文中的伪代码：

```
target = desired output vector
point = training point matrix
procedure takeStep(i1,i2)
  if (i1 == i2) return 0
  alph1 = Lagrange multiplier for i1
  y1 = target[i1]
  E1 = SVM output on point[i1] – y1 (check in error cache)
  s = y1*y2
  Compute L, H via equations (13) and (14)
  if (L == H)
    return 0
  k11 = kernel(point[i1],point[i1])
  k12 = kernel(point[i1],point[i2])
  k22 = kernel(point[i2],point[i2])
  eta = k11+k22-2*k12
  if (eta > 0)
  {
    a2 = alph2 + y2*(E1-E2)/eta
    if (a2 < L) a2 = L
    else if (a2 > H) a2 = H
  }
  else
  {
    Lobj = objective function at a2=L
    Hobj = objective function at a2=H
    if (Lobj < Hobj-eps)
      a2 = L
    else if (Lobj > Hobj+eps)
      a2 = H
    else
      a2 = alph2
  }
  if (|a2-alph2| < eps*(a2+alph2+eps))
    return 0
  a1 = alph1+s*(alph2-a2)
  Update threshold to reflect change in Lagrange multipliers
  Update weight vector to reflect change in a1 & a2, if SVM is linear
  Update error cache using new Lagrange multipliers
  Store a1 in the alpha array
  Store a2 in the alpha array
  return 1
endprocedure

procedure examineExample(i2)
  y2 = target[i2]
  alph2 = Lagrange multiplier for i2
  E2 = SVM output on point[i2] – y2 (check in error cache)
  r2 = E2*y2
  if ((r2 < -tol && alph2 < C) || (r2 > tol && alph2 > 0))
```

```
  {
    if (number of non-zero & non-C alpha > 1)
    {
      i1 = result of second choice heuristic (section 2.2)
      if takeStep(i1,i2)
        return 1
    }
    loop over all non-zero and non-C alpha, starting at a random point
    {
      i1 = identity of current alpha
      if takeStep(i1,i2)
        return 1
    }
    loop over all possible i1, starting at a random point
    {
      i1 = loop variable
      if (takeStep(i1,i2)
        return 1
    }
  }
  return 0
endprocedure

main routine:
  numChanged = 0;
  examineAll = 1;
  while (numChanged > 0 | examineAll)
  {
    numChanged = 0;
    if (examineAll)
      loop I over all training examples
      numChanged += examineExample(I)
    else
      loop I over examples where alpha is not 0 & not C
      numChanged += examineExample(I)
    if (examineAll == 1)
      examineAll = 0
    else if (numChanged == 0)
      examineAll = 1
  }
```

首先就是smo算法的框架：

将smo算法模型作为一个对象，每个变量的作用都加上了注释因此不再说明，指出我们在初始化的时候就先把训练集的核函数提前计算好

```
class PlattSMO:
    def __init__(self,dataMat,classlabels,C,toler,maxIter,**kernel):
        self.x = np.array(dataMat)
```

```
        self.label = np.array(classlabels).transpose()
        #soft margin
        self.C = C
        #epsilon in smo
        self.toler = toler
        #the max count of iteration
        self.maxIter = maxIter
        #m = trainlen
        self.m = np.shape(dataMat)[0]
        #n = the dims of vector
        self.n = np.shape(dataMat)[1]
        self.alpha = np.array(np.zeros(self.m),dtype='float64')
        self.b = 0.0
        #EK in smo
        self.eCache = np.array(np.zeros((self.m,2)))
        #K(i,j)
        self.K = np.zeros((self.m,self.m),dtype='float64')
        #Kernel func, is a dict
        self.kwargs = kernel
        #svm vector
        self.SV = ()
        #svm vector in [0:trainlen]
        self.SVIndex = None
        #initial calculate
        for i in range(self.m):
            for j in range(self.m):
                self.K[i,j] = kernel_func(kernel,self.x[i,:],self.x[j,:])
```

kernel function

在核函数方面，这次选取了4个核函数进行测试：

```
def kernel_func(kernel,x,y) :
  if kernel['name'] == 'linear' :
    return np.inner(x, y)
  elif kernel['name'] == 'gaussian' :
    return np.exp(-np.sqrt(la.norm(np.array(x)-np.array(y)) ** 2 / (2 *
kernel['sigma'] ** 2)))
  elif kernel['name'] == 'rbf' :
    return np.exp(-kernel['gamma']*la.norm(np.subtract(x, y)))
  elif kernel['name'] == 'poly':
    return (kernel['offset'] + np.inner(x, y)) ** kernel['dimension']
```

接着是smo算法的运行函数：

smoP

首先我加上了一个maxIter的变量来限制迭代次数，而对于smo算法，我们选取第一个参数的做法不是随机选取，而是先考虑选取变量处于0到C之间的值，进行判断，另外在我们的innerL函数的返回值会判断是否迭代更新了alpha（也就是拉格朗日乘子），如果没有更新的情况下，说明启发式做法已经不管用了，我们就需要遍历alpha，如果还没有更新，说明迭代收敛，也就会跳出循环了，这里就是完全参照smo算法的伪代码实现的，就是额外增加了一个最大迭代次数

```
def smoP(self):
        iter = 0
        entrySet = True
        alphaPairChanged = 0
        while iter < self.maxIter and ((alphaPairChanged > 0) or (entrySet)):
            #print("current iter:",iter)
            alphaPairChanged = 0
            if entrySet:
                for i in range(self.m):
                    alphaPairChanged+=self.innerL(i)
                iter += 1
            else:
                nonBounds = np.nonzero((self.alpha > 0)*(self.alpha < self.C))
[0]
                for i in nonBounds:
                    alphaPairChanged+=self.innerL(i)
                iter+=1
            if entrySet:
                entrySet = False
            elif alphaPairChanged == 0:
                entrySet = True
        self.SVIndex = np.nonzero(self.alpha)[0]
        self.SV = self.x[self.SVIndex]
        self.SVAlpha = self.alpha[self.SVIndex]
        self.SVLabel = self.label[self.SVIndex]
        self.x = None
        self.K = None
        self.label = None
        self.alpha = None
        self.eCache = None
```

innerL

接着就是innerL函数：当我们选择了第一个alpha参数的时候，我们首先先计算现在这个alpha的值的样本误差Ei（这里就是论文中的Ei，不多进行描述），如果其不满足KKT条件（在这里我们没有使用理论值0，而是用了一个误差值toler来进行判断，这里也是根据论文实现的），就需要更新alpha

```
    def innerL(self,i):
        Ei = self.calcEK(i)
        #can't keep for KKT
        if (self.label[i] * Ei < -self.toler and self.alpha[i] < self.C) or \
                (self.label[i] * Ei > self.toler and self.alpha[i] > 0):
            self.updateEK(i)
```

```
            #find j
            j,Ej = self.selectJ(i,Ei)
            #the old value
            alphaIOld = self.alpha[i].copy()
            alphaJOld = self.alpha[j].copy()
            #just as smo, if yi != yj, L = max(0,alphaj - alphai), H =
min(C,C+alphaj-alphai)
            if self.label[i] != self.label[j]:
                L = max(0,self.alpha[j]-self.alpha[i])
                H = min(self.C,self.C + self.alpha[j]-self.alpha[i])
            else:
            #the same
                L = max(0,self.alpha[j]+self.alpha[i] - self.C)
                H = min(self.C,self.alpha[i]+self.alpha[j])
            if L == H:
                return 0
            #eta = K(x1,x1) K(x2,x2) - 2*K(x1,x2)
            eta = self.K[i,i] + self.K[j,j] - 2*self.K[i,j]
            if eta <= 0:
                #don't care about kernel function that can't keep for Mercer
                return 0
            self.alpha[j] += self.label[j]*(Ei-Ej)/eta
            #cut, or cliped
            self.alpha[j] = clipAlpha(self.alpha[j],H,L)
            self.updateEK(j)
            #change is small
            if abs(alphaJOld-self.alpha[j]) < 0.00001:
                return 0
            #update
            self.alpha[i] +=  self.label[i]*self.label[j]*(alphaJOld-
self.alpha[j])
            self.updateEK(i)
            b1 = self.b - Ei - self.label[i] * self.K[i, i] * (self.alpha[i] -
alphaIOld) - \
                 self.label[j] * self.K[i, j] * (self.alpha[j] - alphaJOld)
            b2 = self.b - Ej - self.label[i] * self.K[i, j] * (self.alpha[i] -
alphaIOld) - \
                 self.label[j] * self.K[j, j] * (self.alpha[j] - alphaJOld)
            #calculate b
            if 0<self.alpha[i] and self.alpha[i] < self.C:
                self.b = b1
            elif 0 < self.alpha[j] and self.alpha[j] < self.C:
                self.b = b2
            else:
                self.b = (b1 + b2) /2.0
            return 1
        else:
            return 0
```

selectJ

对于选择第二个alpha：这里是根据论文中的设计来选择的：

> 一旦第一个乘数选定，SMO开始选择第二个乘数来最大化优化的步伐。现在，计算核函数很耗时间，所以SMO通过（16）中的|E1-E2|来估计步长。SMO为训练集的每一个非边界样本记录一个错误值E，然后选择一个错误值来估计最大步长。如果E1是正，SMO选择一个最小错误值E2；如果E1是负，SMO选择一个最大错误值E2.

如果找不到E非0的alpha(注意这里需要判断非0长度大于1的情况，因为第一个参数的E会是非0值，所以长度一定是大于等于1的），那么就随机选一个

```
def selectJ(self,i,Ei):
        #select the second alpha, first search for nonzero,
        #if not then random select
        maxE = 0.0
        selectJ = 0
        Ej = 0.0
        validECacheList = np.nonzero(self.eCache[:,0])[0]
        if len(validECacheList) > 1:
            for k in validECacheList:
                if k == i:continue
                Ek = self.calcEK(k)
                deltaE = abs(Ei-Ek)
                if deltaE > maxE:
                    selectJ = k
                    maxE = deltaE
                    Ej = Ek
            return selectJ,Ej
        else:
            selectJ = selectJrand(i,self.m)
            Ej = self.calcEK(selectJ)
            return selectJ,Ej
```

之后就根据smo算法中的几个公式计算得到结果

```
 #just as smo, if yi != yj, L = max(0,alphaj - alphai), H = min(C,C+alphaj-
alphai)
            if self.label[i] != self.label[j]:
                L = max(0,self.alpha[j]-self.alpha[i])
                H = min(self.C,self.C + self.alpha[j]-self.alpha[i])
            else:
            #the same
                L = max(0,self.alpha[j]+self.alpha[i] - self.C)
                H = min(self.C,self.alpha[i]+self.alpha[j])
            if L == H:
                return 0
            #eta = K(x1,x1) K(x2,x2) - 2*K(x1,x2)
            eta = self.K[i,i] + self.K[j,j] - 2*self.K[i,j]
```

```
            if eta <= 0:
                #don't care about kernel function that can't keep for Mercer
                return 0
            self.alpha[j] += self.label[j]*(Ei-Ej)/eta
```

clipAlpha

同时也根据论文，计算出来的新的alpha需要进行截断：

```
def clipAlpha(a_j,H,L):
    if a_j > H:
        a_j = H
    if L > a_j:
        a_j = L
    return a_j
```

我们更新了alpha，自然也需要更新其现在的样本误差：

```
self.updateEK(i)

def updateEK(self,k):
        #update E of training sample k
        Ek = self.calcEK(k)
```

之后如果的差异太小，就默认这两个值是相等的，就不进行b的更新，否则也是根据论文中的描述进行更新：

```
            if abs(alphaJOld-self.alpha[j]) < 0.00001:
                return 0
            #update
            self.alpha[i] +=  self.label[i]*self.label[j]*(alphaJOld-
self.alpha[j])
            self.updateEK(i)
            b1 = self.b - Ei - self.label[i] * self.K[i, i] * (self.alpha[i] -
alphaIOld) - \
                 self.label[j] * self.K[i, j] * (self.alpha[j] - alphaJOld)
            b2 = self.b - Ej - self.label[i] * self.K[i, j] * (self.alpha[i] -
alphaIOld) - \
                 self.label[j] * self.K[j, j] * (self.alpha[j] - alphaJOld)
            #calculate b
            if 0<self.alpha[i] and self.alpha[i] < self.C:
                self.b = b1
            elif 0 < self.alpha[j] and self.alpha[j] < self.C:
                self.b = b2
            else:
                self.b = (b1 + b2) /2.0
```

对于innerL如果变化了就返回1，没有变化就返回0，用来更新外面的alphapairchange变量

predict

在预测方面，我们有了alpha和b，可以获得w以及支持变量，因此就直接根据课上ppt的公式获得结果：

$$y* = sign(\sum \alpha_i y_i K(x_i, x') + b)$$

```
    def predict(self,testData):
        test = np.array(testData)
        result = []
        m = np.shape(test)[0]
        for i in range(m):
            tmp = self.b
            for j in range(len(self.SVIndex)):
                tmp += self.SVAlpha[j] * self.SVLabel[j] *
kernel_func(self.kwargs,self.SV[j],test[i,:])
            while tmp == 0:
                tmp = random.uniform(-1,1)
            if tmp > 0:
                tmp = 1
            else:
                tmp = -1
            result.append(tmp)
        return result
```

算法结果：

所使用的一系列参数：

```
Cs = [0.0001,0.001,0.01,0.1,1,10,100,1000,10000]
Gammas = [0.001,0.01,0.1,1,10,100]
Sigmas = [0.25,0.5,1,2,4,8]
Dimensions = [1,2,3,4]
```

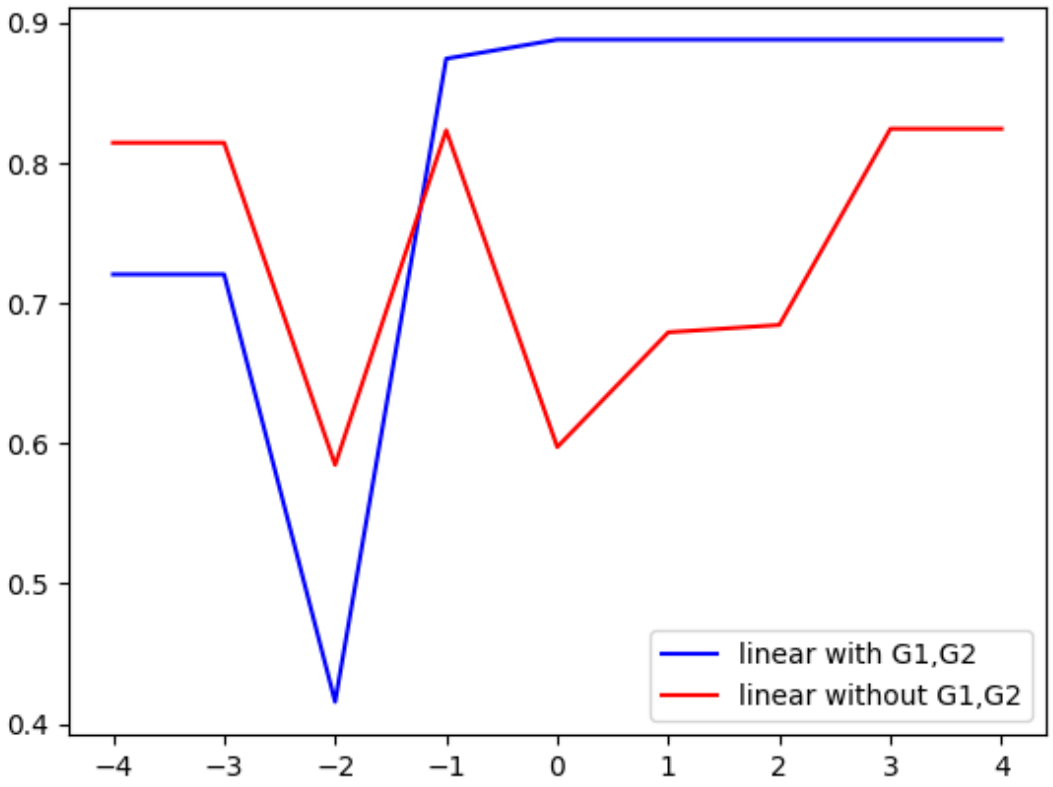
线性

```
para = {'name':'linear'}
  for C in Cs:
    start = time.time()
    result,result_without =
SVM_lab2_no_cvxopt(allset,labelset,trainlen,setlen,C,10000,para)
    F1,F1_without = SVM_test(result,result_without,labelset,trainlen,setlen)
    end = time.time()
    runtime.append(end-start)
    F1_list.append(F1)
    F1_without_list.append(F1_without)
```

线性结果：

por结果：

C	with G1,G2	without G1,G2
0.0001	0.720	0.814
0.001	0.720	0.814
0.01	0.415	0.584
0.1	0.874	0.823
1	0.888	0.597
10	0.888	0.679
100	0.888	0.684
1000	0.888	0.824
10000	0.888	0.824

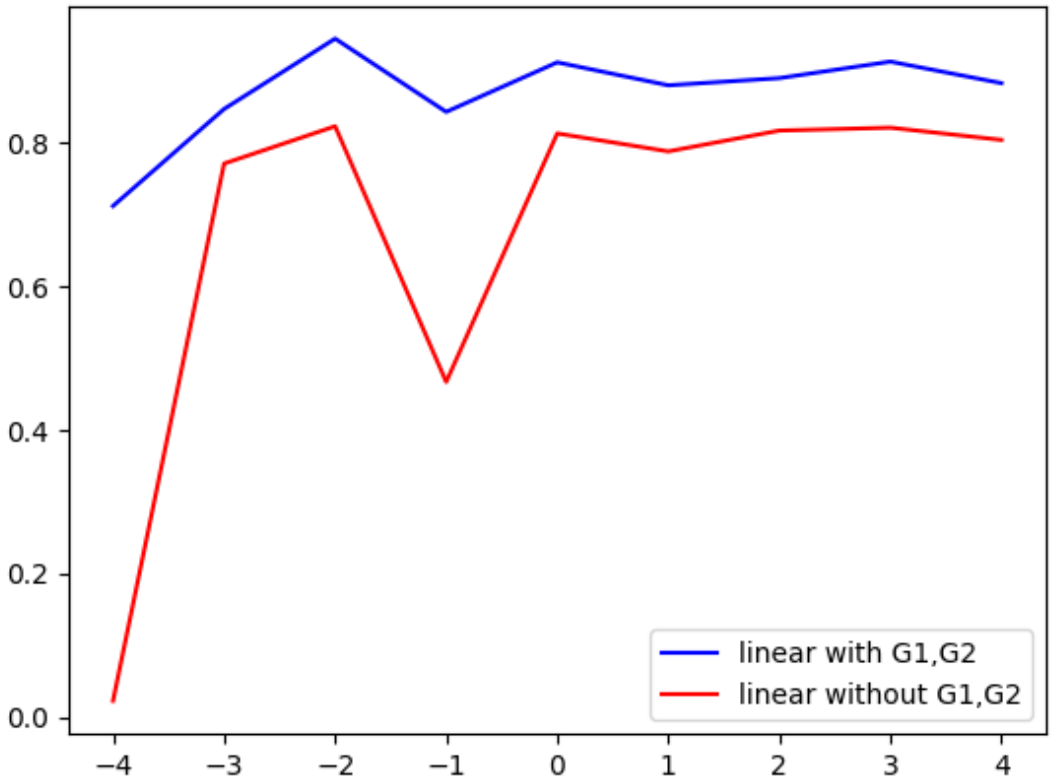


其中横坐标代表logC

可以看到当C的值很小的时候，F1值波动是很大的，我认为主要原因是C的小导致惩罚小，导致支持变量在[-1,1]之间的几率变大，而又因为smo算法其实还是带有一定随机性的（如果没有非0alpha因子启发式失效，随机选择一个alpha开始运算），所以当C小的时候结果也时好时坏，但是当C变大时结果开始趋于稳定，同时呈上升趋势

mat结果：

C	with G1,G2	without G1,G2
0.0001	0.712	0.023
0.001	0.847	0.771
0.01	0.945	0.823
0.1	0.843	0.467
1	0.912	0.813
10	0.880	0.788
100	0.890	0.817
1000	0.913	0.821
10000	0.883	0.804



其中横坐标代表logC

可以看到对于C比较小的情况，也存在着一定的波动，而对于C增大时，F1值也逐渐稳定

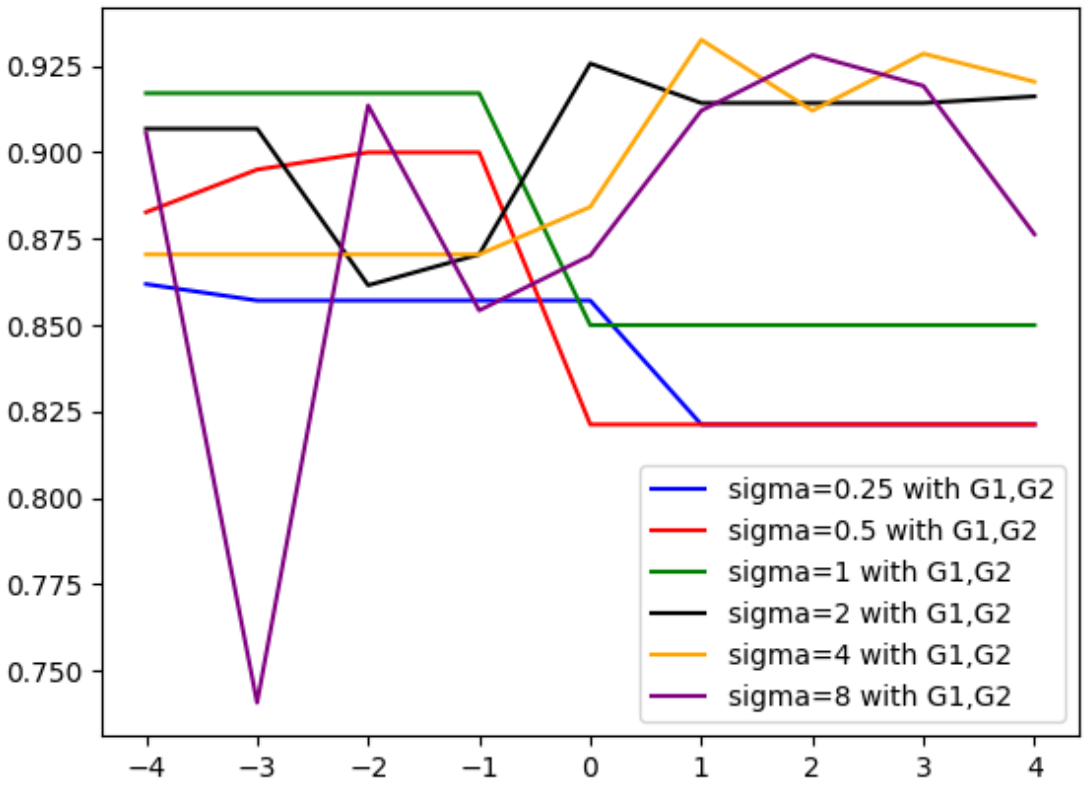
高斯

```
para = {'name':'gaussian'}
  for sigma in Sigmas :
    para['sigma'] = sigma
    runtime_single_C = []
    F1_single_C = []
    F1_without_single_C = []
    for C in Cs :
      start = time.time()
      result,result_without =
SVM_lab2_no_cvxopt(allset,labelset,trainlen,setlen,C,10000,para)
      F1,F1_without = SVM_test(result,result_without,labelset,trainlen,setlen)
      end = time.time()
      runtime_single_C.append(end-start)
      F1_single_C.append(F1)
      F1_without_single_C.append(F1_without)
    runtime.append(runtime_single_C)
    F1_list.append(F1_single_C)
    F1_without_list.append(F1_without_single_C)
```

高斯结果：

mat结果：

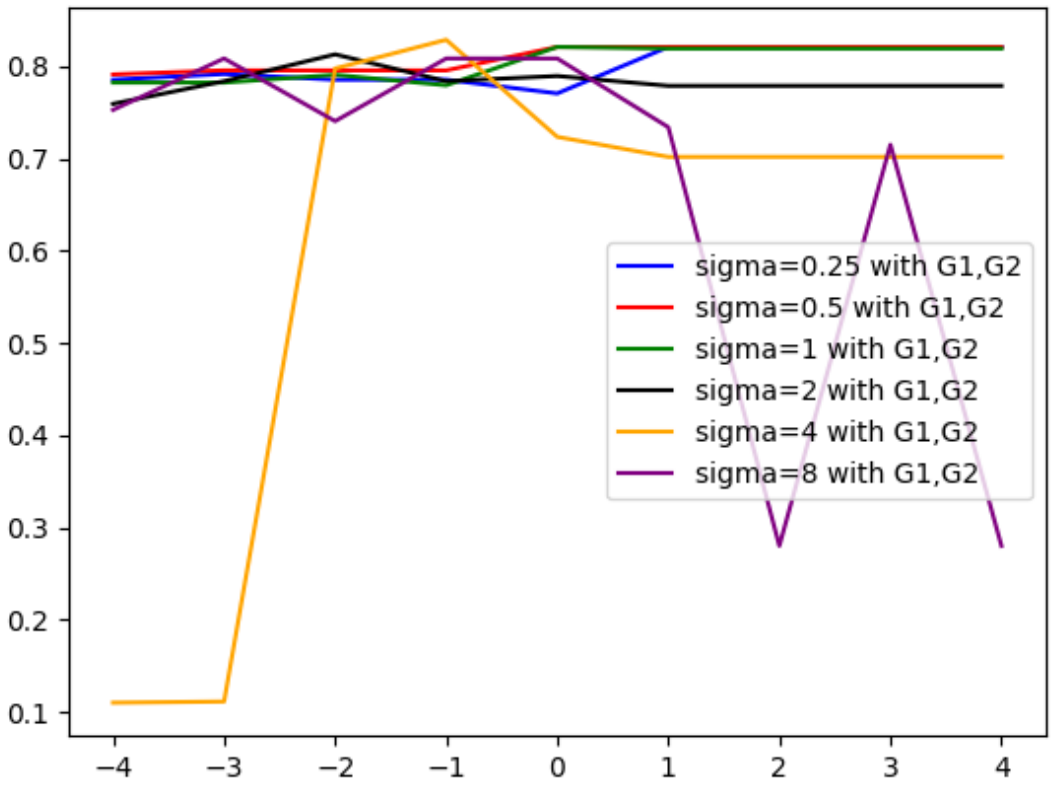
使用G1，G2:



C	sigma=0.25	sigma=0.5	sigma=1	sigma=2	sigma=4	sigma=8
0.0001	0.861	0.882	0.917	0.906	0.870	0.905
0.001	0.857	0.895	0.917	0.906	0.879	0.740
0.01	0.857	0.9	0.917	0.861	0.870	0.913
0.1	0.857	0.9	0.917	0.870	0.870	0.854
1	0.857	0.821	0.85	0.925	0.884	0.870
10	0.821	0.821	0.85	0.914	0.932	0.912
100	0.821	0.821	0.85	0.914	0.912	0.928
1000	0.821	0.821	0.85	0.914	0.928	0.919
10000	0.821	0.821	0.85	0.916	0.920	0.876

可以看到对于高斯核，sigma=4时当C足够大的情况下得到的结果普遍比其他结果要来的好，其中最好的结果在sigma=4，C=10

不使用G1，G2(在代码中因为标签设置出错所以这里写成了with，其实应该是不含G1，G2的):



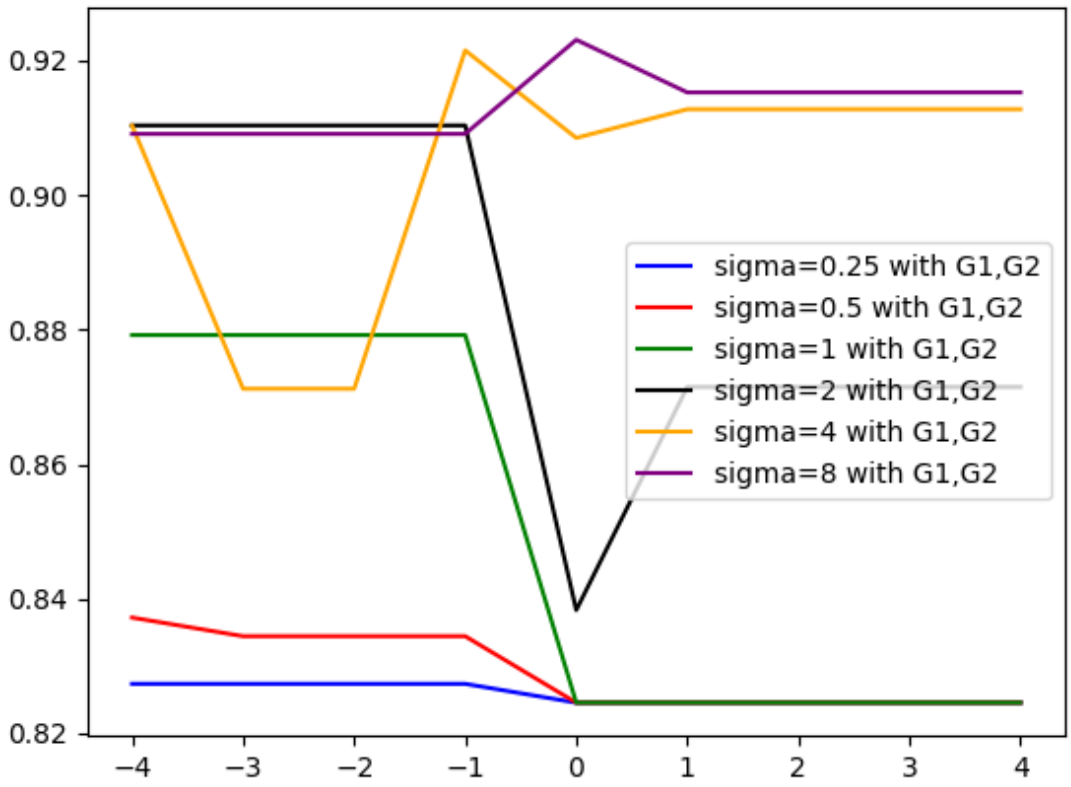
C	sigma=0.25	sigma=0.5	sigma=1	sigma=2	sigma=4	sigma=8
0.0001	0.785	0.791	0.782	0.759	0.109	0.752
0.001	0.791	0.795	0.782	0.784	0.111	0.808
0.01	0.785	0.795	0.790	0.813	0.797	0.740
0.1	0.785	0.795	0.779	0.784	0.829	0.808
1	0.770	0.821	0.821	0.789	0.723	0.808
10	0.821	0.821	0.819	0.779	0.701	0.734
100	0.821	0.821	0.819	0.779	0.701	0.279
1000	0.821	0.821	0.819	0.779	0.701	0.715
10000	0.821	0.821	0.819	0.779	0.701	0.279

可以看到对于高斯核，结果大部分维持在0.8上下，有少部分结果会偏低，总体结果和线性核相当，较优的结果出现在sigma比较小的情况下

por结果：

使用G1，G2

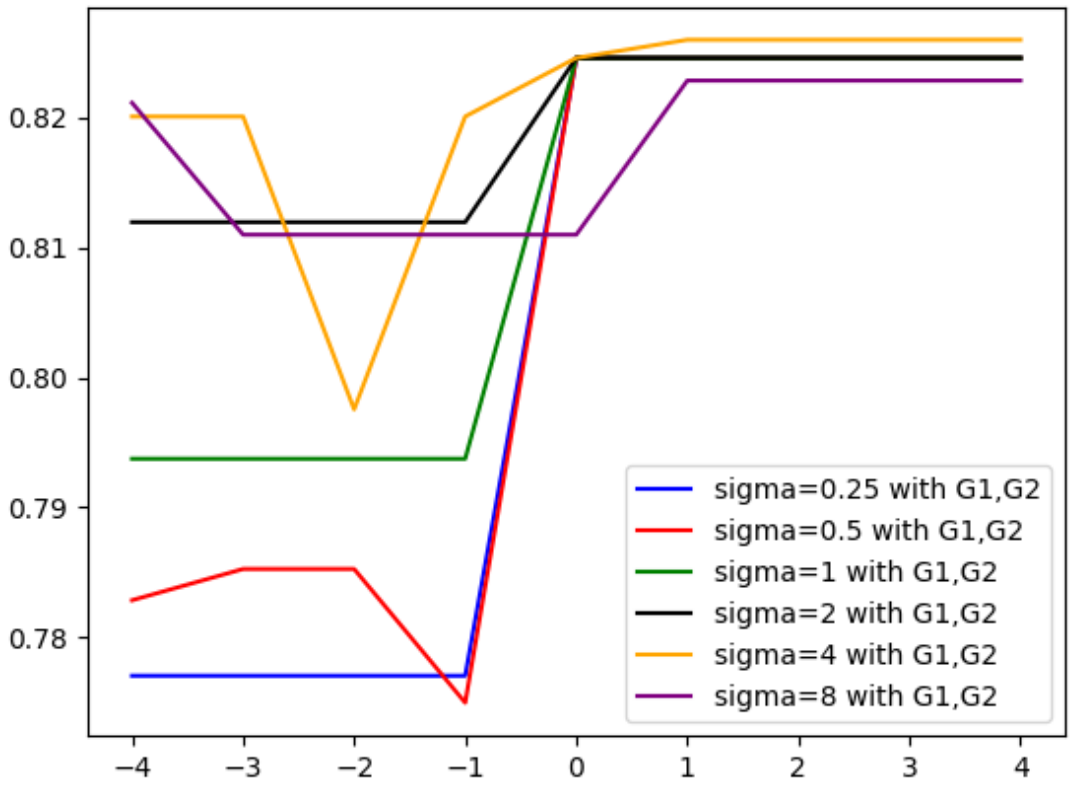
C	sigma=0.25	sigma=0.5	sigma=1	sigma=2	sigma=4	sigma=8
0.0001	0.827	0.837	0.879	0.910	0.910	0.909
0.001	0.827	0.834	0.879	0.910	0.871	0.909
0.01	0.827	0.834	0.879	0.910	0.871	0.909
0.1	0.827	0.834	0.879	0.910	0.921	0.909
1	0.824	0.824	0.824	0.838	0.908	0.923
10	0.824	0.824	0.824	0.871	0.912	0.915
100	0.824	0.824	0.824	0.871	0.912	0.915
1000	0.824	0.824	0.824	0.871	0.912	0.915
10000	0.824	0.824	0.824	0.871	0.912	0.915



可以看到，针对mat的数据来说总体情况是随着sigma增大F1值增大的，而最高点出现在sigma=8，C=1的情况

不使用G1，G2(在代码中因为标签设置出错所以这里写成了with，其实应该是不含G1，G2的):

C	sigma=0.25	sigma=0.5	sigma=1	sigma=2	sigma=4	sigma=8
0.0001	0.777	0.782	0.793	0.811	0.820	0.821
0.001	0.777	0.785	0.793	0.811	0.820	0.810
0.01	0.777	0.785	0.793	0.811	0.797	0.810
0.1	0.777	0.775	0.793	0.811	0.820	0.810
1	0.824	0.824	0.824	0.824	0.824	0.810
10	0.824	0.824	0.824	0.824	0.825	0.822
100	0.824	0.824	0.824	0.824	0.825	0.822
1000	0.824	0.824	0.824	0.824	0.825	0.822
10000	0.824	0.824	0.824	0.824	0.825	0.822



对于不使用G1，G2的结果，在我们可调的sigma参数范围内，最大值都只有在0.825左右，而且基本都是随着C的增大F1值增大

rbf

```
para = {'name':'rbf'}
  for gamma in Gammas :
    para['gamma'] = gamma
    runtime_single_C = []
    F1_single_C = []
    F1_without_single_C = []
    for C in Cs :
```

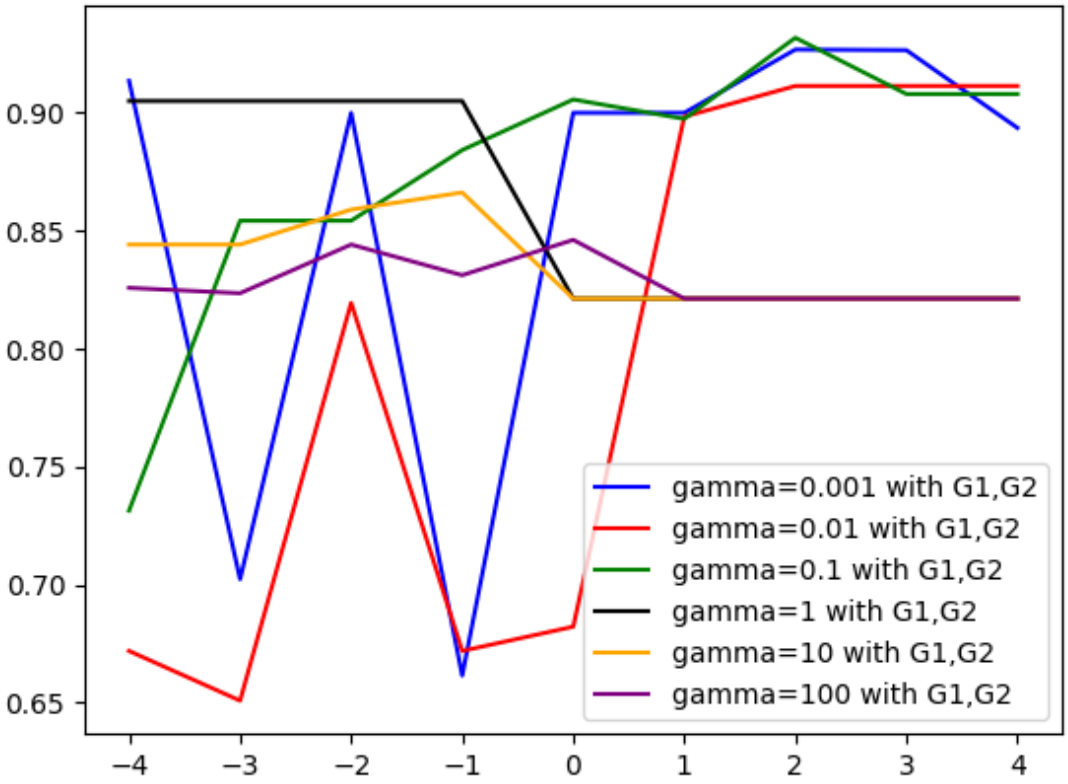
```
        start = time.time()
        result,result_without =
SVM_lab2_no_cvxopt(allset,labelset,trainlen,setlen,C,10000,para)
        F1,F1_without = SVM_test(result,result_without,labelset,trainlen,setlen)
        end = time.time()
        runtime_single_C.append(end-start)
        F1_single_C.append(F1)
        F1_without_single_C.append(F1_without)
    runtime.append(runtime_single_C)
    F1_list.append(F1_single_C)
    F1_without_list.append(F1_without_single_C)
```

rbf结果

mat结果：

使用G1，G2

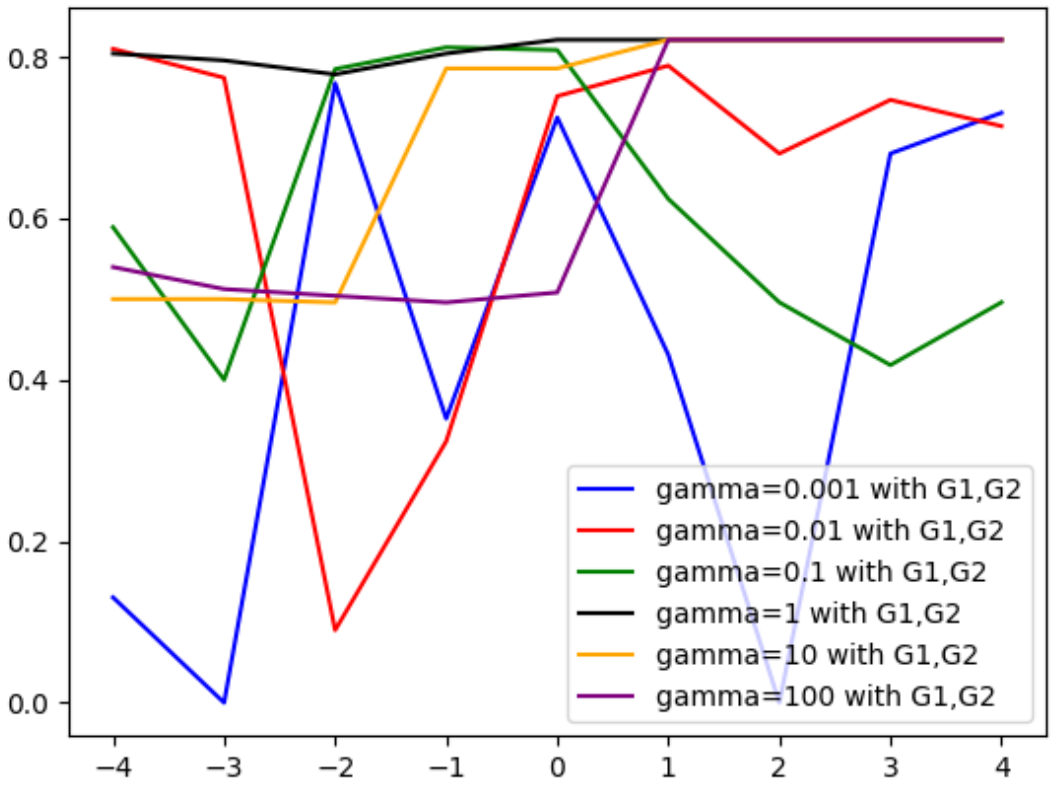
C	sigma=0.25	gamma=0.01	gamma=0.1	gamma=1	gamma=10	gamma=100
0.0001	0.913	0.671	0.731	0.905	0.844	0.825
0.001	0.702	0.650	0.854	0.905	0.844	0.823
0.01	0.899	0.819	0.854	0.905	0.858	0.844
0.1	0.661	0.671	0.884	0.905	0.866	0.831
1	0.899	0.682	0.905	0.821	0.821	0.846
10	0.899	0.898	0.897	0.821	0.821	0.821
100	0.926	0.911	0.931	0.821	0.821	0.821
1000	0.926	0.911	0.907	0.821	0.821	0.821
10000	0.893	0.911	0.907	0.821	0.821	0.821



对于mat生成的训练测试集来说，gamma=0.1、C=100的时候F1值最高

不使用G1，G2(在代码中因为标签设置出错所以这里写成了with，其实应该是不含G1，G2的):

C	gamma=0.001	gamma=0.01	gamma=0.1	gamma=1	gamma=10	gamma=100
0.0001	0.130	0.809	0.589	0.804	0.5	0.539
0.001	0	0.773	0.399	0.795	0.5	0.512
0.01	0.767	0.089	0.784	0.778	0.495	0.504
0.1	0.351	0.323	0.811	0.804	0.785	0.495
1	0.725	0.751	0.808	0.821	0.785	0.508
10	0.431	0.789	0.624	0.821	0.821	0.821
100	0	0.680	0.496	0.821	0.821	0.821
1000	0.680	0.746	0.418	0.821	0.821	0.821
10000	0.730	0.714	0.496	0.821	0.821	0.821

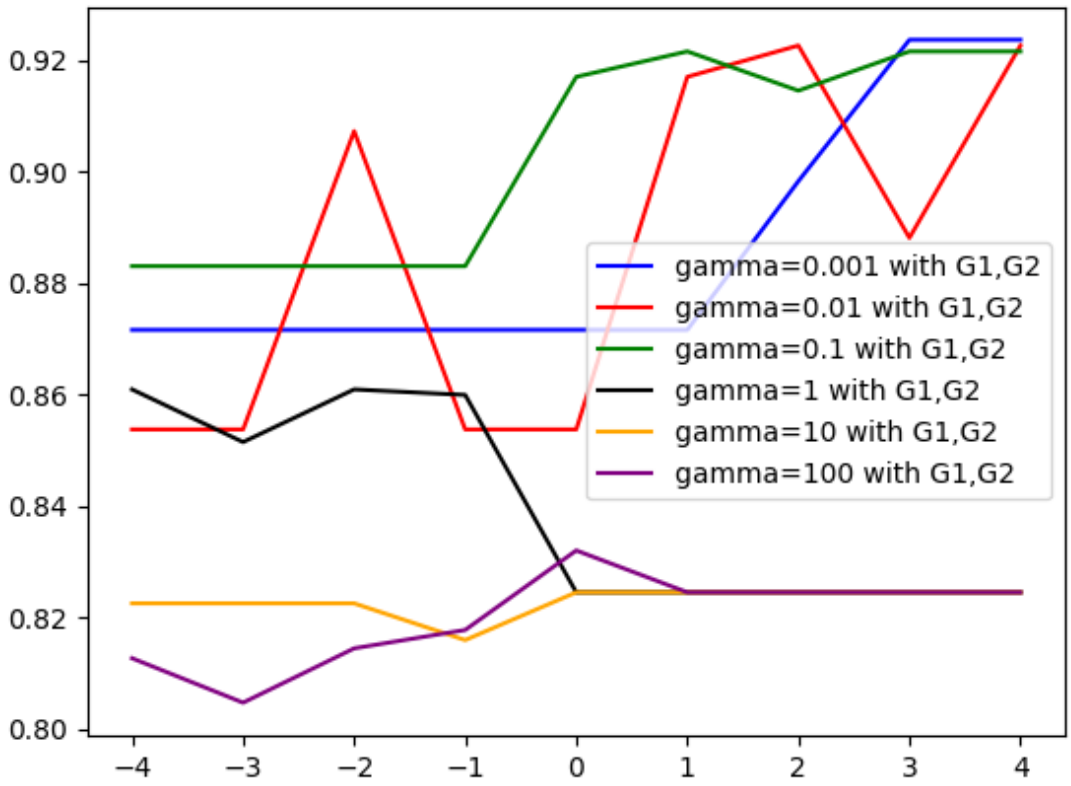


最高F1值为0.821，此时gamma=1、10、100，C=1、10、100、1000，10000，注意到0.821这个结果在我们前面用的几个核函数的C值高的时候也频繁出现，但是因为没有输出划分结果等数据，不能断定其划分是一样的

por结果：

使用G1、G2

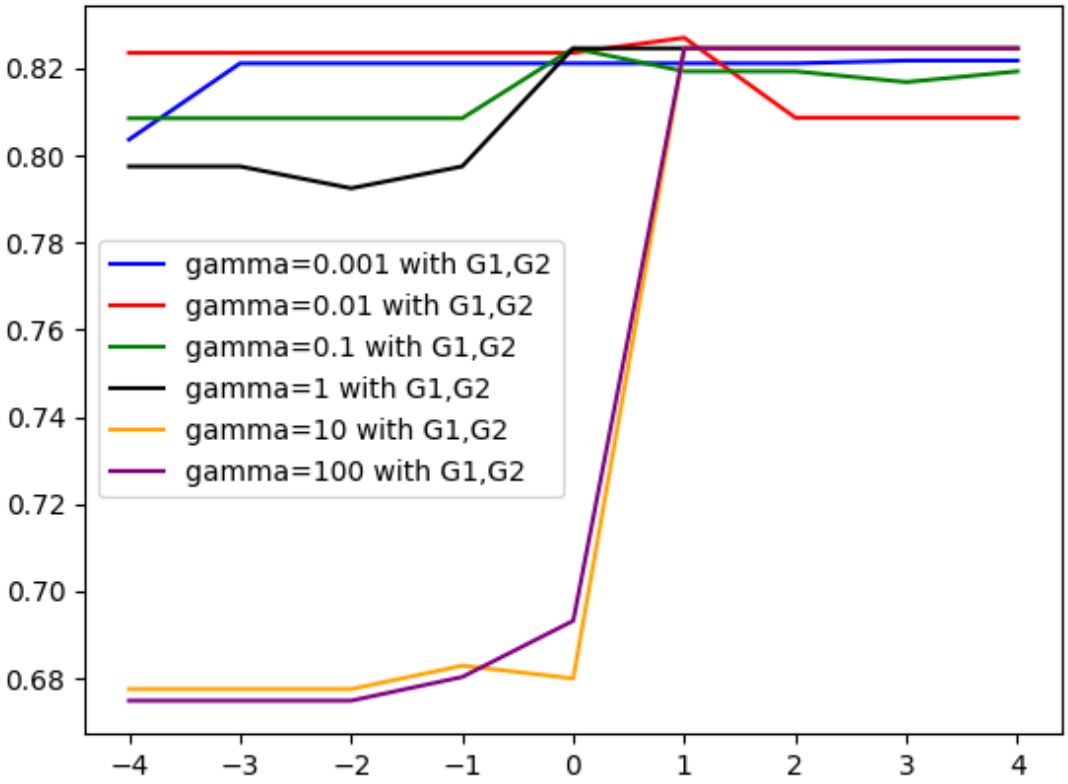
C	gamma=0.001	gamma=0.01	gamma=0.1	gamma=1	gamma=10	gamma=100
0.0001	0.871	0.853	0.883	0.860	0.822	0.812
0.001	0.871	0.853	0.883	0.851	0.822	0.804
0.01	0.871	0.907	0.883	0.860	0.822	0.814
0.1	0.871	0.853	0.883	0.859	0.816	0.817
1	0.871	0.853	0.916	0.824	0.824	0.832
10	0.871	0.916	0.921	0.824	0.824	0.824
100	0.898	0.922	0.914	0.824	0.824	0.824
1000	0.923	0.888	0.921	0.824	0.824	0.824
10000	0.923	0.922	0.921	0.824	0.824	0.824



对于使用G1，G2的por的测试训练集来说，反而是gamma的值比较小的情况下por得到的结果更好

不使用G1，G2(在代码中因为标签设置出错所以这里写成了with，其实应该是不含G1，G2的):

C	gamma=0.001	gamma=0.01	gamma=0.1	gamma=1	gamma=10	gamma=100
0.0001	0.803	0.823	0.808	0.797	0.677	0.674
0.001	0.821	0.823	0.808	0.797	0.677	0.674
0.01	0.821	0.823	0.808	0.792	0.677	0.674
0.1	0.821	0.823	0.808	0.797	0.682	0.680
1	0.821	0.823	0.824	0.824	0.68	0.693
10	0.821	0.826	0.819	0.824	0.824	0.824
100	0.821	0.808	0.819	0.824	0.824	0.824
1000	0.821	0.808	0.816	0.824	0.824	0.824
10000	0.821	0.808	0.819	0.824	0.824	0.824



在这个测试数据集下，不使用

算法最优结果汇总及总结

	mat：含G1，G2	mat：不含G1，G2	por：含G1，G2	por：不含G1，G2
线性	0.945(C=0.01)	0.823(C=0.01)	0.888(C=1~10000)	0.824(C=1000/10000)
高斯	0.932(C=10,sigma=4)	0.829(C=0.1,sigma=4)	0.923(C=1,sigma=8)	0.825(C=10~10000,sigma=4)
rbf	0.931(C=100,gamma=0.1)	0.821(C=1~10000,gamma=1~100)	0.922(C=100,gamma=0.01)	0.826(C=10,gamma=0.01)

可以看到，结果上并不是C越大最后结果也就越好（我想主要问题出在C越大只是使得划分可能更加准确，但是这样的划分不一定能使得我们的预测更加准确，反而是C小的时候因为划分不是很准确，带有一定随机性，所以可能会roll出一个更高的结果）

总体上SVM的结果比起KNN的结果来的好

决策树部分：

在这里我另外实现的一个机器学习的算法就是决策树算法，这里就是按照ppt的算法进行设计并实现的

run

就是按照ppt上的算法实现的，在这里default值我用在样本里面的出现最频繁的结果作为返回值

```
def run(self,examples,attrs,default):
    if examples == []:
      return default
    result,classification = self.testExample(examples)
```

```
    if result:
      return classification
    if attrs == []:
      return self.mode(examples)
    best = self.chooseAttr(examples,attrs)
    #print(best)
    tree = DTL_tree(best)
    attrs_i = attrs
    attrs_i.remove(best)
    default_val = self.mode(examples)
    for value in self.attrlist[best]:
      example_i = []
      for example in examples:
        if example[best] == value:
          example_i.append(example)
      subtree = self.run(example_i,attrs_i,default_val)
      tree.addsubtree(subtree,value)
    return tree
```

chooseAttr

选择属性的时候，是计算每个属性在这个样本输入下的IG值，进行排序，返回第一个

```
def chooseAttr(self,examples,attrs):
    #choose the best
    attrlen = len(examples[0])
    attr_rank_dict = {}
    pos = 0
    neg = 0
    for example in examples:
      if example[attrlen-1] == 1 :
        pos+=1
      else :
        neg+=1
    IG = self.calcI(pos,neg)
    for attr in attrs:
      remainder = 0
      pos_i = {}
      neg_i = {}
      for value in self.attrlist[attr]:
        pos_i[value] = 0
        neg_i[value] = 0
      for example in examples:
        #print(self.attrlist[attr])
        #print(attr,example[attr])
        if example[attrlen-1] == 1 :
          pos_i[example[attr]] += 1
        else :
          neg_i[example[attr]] += 1
```

```
      for value in self.attrlist[attr]:
        if pos_i[value]+neg_i[value] != 0:
          remainder += self.calcI(pos_i[value],neg_i[value]) *
(pos_i[value]+neg_i[value])/(pos+neg)
      attr_rank_dict[attr] = IG - remainder
    #print(attr_rank_dict)
    attrrank = sorted(attr_rank_dict.items(),key=lambda
attr_rank_dict:attr_rank_dict[1],reverse=True)
    return attrrank[0][0]
```

算法结果

因为在决策树部分如果测试集训练集一定，构造出来的结果是确定的，因此能调参的地方是在训练集测试集的选择上，而我又没有做划分随机化，因此只有一个固定结果：

```
with G1,G2:0.8737201365187712
without G1,G2:0.8086419753086419
```

实验总结

通过本次实验我进一步熟悉了svm算法和knn算法的实现，同时进行了一定量的测试，感受到了数据科学的魅力（写作魅力念做玄学）

本次实验中我还有很多的不足如没有考虑随机划分测试集训练集而是选择使用一刀切的方法