常用库导入及配置

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import warnings
```

```
warnings.filterwarnings('ignore')
plt.style.use('ggplot')# 画图风格
plt.rcParams['font.sans-serif']=['SimHei'] # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus']=False # 用来正常显示负号
```

感知机

```
# 数据读取，各维名称为'one', 'two', 'three'
data = pd.read_table('mlp_data.txt', encoding='gb2312', delim_whitespace=True, names=
['one', 'two', 'three', 'label'])

X = data.iloc[:, 0:3] # X为坐标
y = data[['label']]   # y为标签
```

库函数

```
from sklearn.linear_model import Perceptron # 感知机库


# 损失阈值为1e-3,
clf = Perceptron(tol=1e-3)
clf.fit(X, y)
clf.score(X, y)
```

```
1.0
```

- 标准化处理对比

```
from sklearn.preprocessing import StandardScaler
```

```
sc = StandardScaler()
_X = sc.fit_transform(X)
clf = Perceptron(tol=1e-3, random_state=0)
clf.fit(_X, y)
clf.score(_X, y)
```

```
1.0
```

自编

```
def perception(X, y, lr=0.0002, epochs=5000):
    # 初始化权重，偏置
    w = np.zeros(X.shape[1])
    b = 0
    for epoch in range(epochs):
        for xi, yi in zip(X, y):
            if yi * (np.dot(xi, w.reshape(-1, 1)) + b) <= 0: # 样本分类错误，更新参数
                w += lr * xi * yi
                b += lr * yi
                break
    print('w: {} b: {}'.format(w, b))
    return w, b
```

- 对偶算法

```
def dual_perception(X, y, lr=0.0002, epochs=200):
    metric = np.dot(X, X.transpose())# 计算Gram矩阵，减少计算量
    alpha = np.zeros(len(X))# 实例点更新次数
    l = range(len(X))
    b = 0
    for epoch in range(epochs):
        for i in l:
            if y[i] * (np.sum([alpha[j] * y[j] * metric[i][j]
                               for j in l]) + b) <= 0: # 样本分类错误，更新参数
                alpha[i] += lr
                b += lr * y[i]
    w = sum([alpha[i] * y[i] * X[i] for i in l])

    print('alpha\n', alpha)
    print('w: {} b: {}'.format(w, b))
    return w, b
```

- 精确度计算

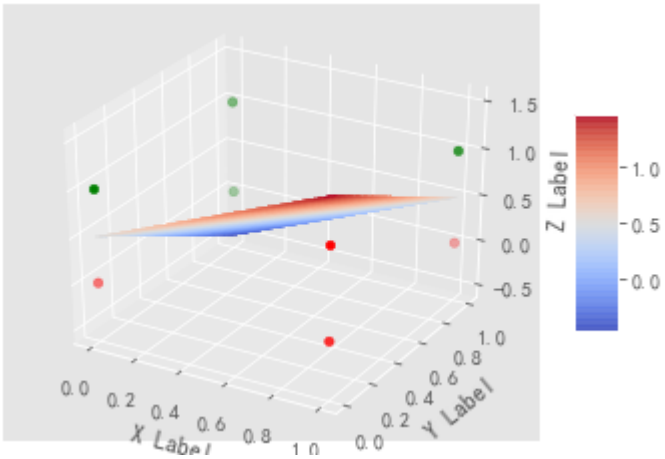
```
def score(X, y, w, b):
    n = 0
    for xi, yi in zip(X, y):
        if yi * (np.dot(xi, w.reshape(-1, 1)) + b) > 0:
            n += 1
    print("accuracy: ", n / len(X))
```

可视化

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm


def show_mlp_3d(w, b, data):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    x = np.linspace(0, 1, 100)
    y = np.linspace(0, 1, 100)
    x, y = np.meshgrid(x, y)
    z = []
    for xi, yi in zip(x, y):
        zi = (-w[0] * xi - w[1] * yi - b) / w[2]
        z.append(zi)
    z = np.array(z)

    surf = ax.plot_surface(
        x, y, z, cmap=cm.coolwarm, linewidth=0, antialiased=False, alpha=0.5)
    fig.colorbar(surf, shrink=0.5, aspect=5)
    T = data[data['label'] == 1]
    F = data[data['label'] == -1]
    ax.scatter(T['one'], T['two'], T['three'], c='green')
    ax.scatter(F['one'], F['two'], F['three'], c='red')
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()
```

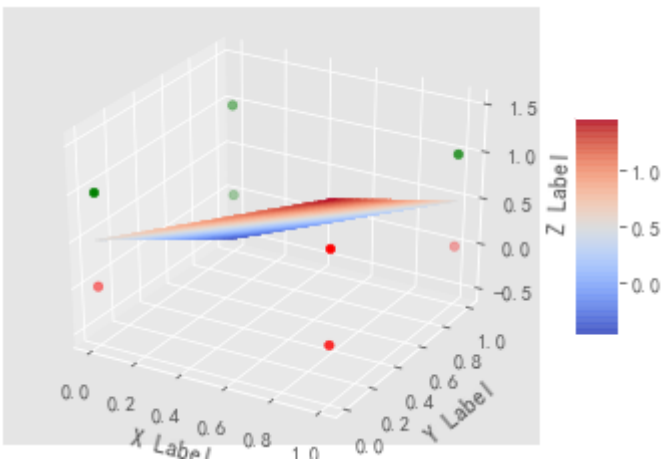
```
w, b = perception(X.values, y.values, lr=1, epochs=200)
score(X.values, y.values, w, b)

show_mlp_3d(w, b, data)
```

```
w: [-2.  2.  2.] b: [-1]
accuracy:  1.0
```




```
dual_w, dual_b = dual_perception(X.values, y.values, lr=1, epochs=200)
score(X.values, y.values, dual_w, dual_b)

show_mlp_3d(dual_w, dual_b, data)
```

```
alpha
 [4. 1. 1. 0. 3. 0. 2. 0.]
w: [-2.  2.  2.] b: [-1]
accuracy:  1.0
```



总结

- 感知机的实现原理比较简单，题中8个数据点线性可分。因此没有经过处理的原数据在自编的感知机代码中也能达到百分百的正确率。在一般情况下，若分类效果不理想，应将数据标准化（减去平均值，再除以其标准差，得到均值为0，标准差为1的服从标准正态分布的数据。
- 在具体实现中，可使用numpy的矩阵运算如点积，矩阵相乘，多维向量广播特性简化代码并且提高运算效率。
- 对偶算法中应先计算Gram矩阵，可减少计算量，代码也更加简洁明了。
- 题目中所给数据是三维的，因此可以可视化表示出来，且所得分界面也是一个平面。具体实现可以通过numpy以及感知机分界面公式求得对应坐标，继而使用mpl_toolkits的三维图形库画图。

决策树

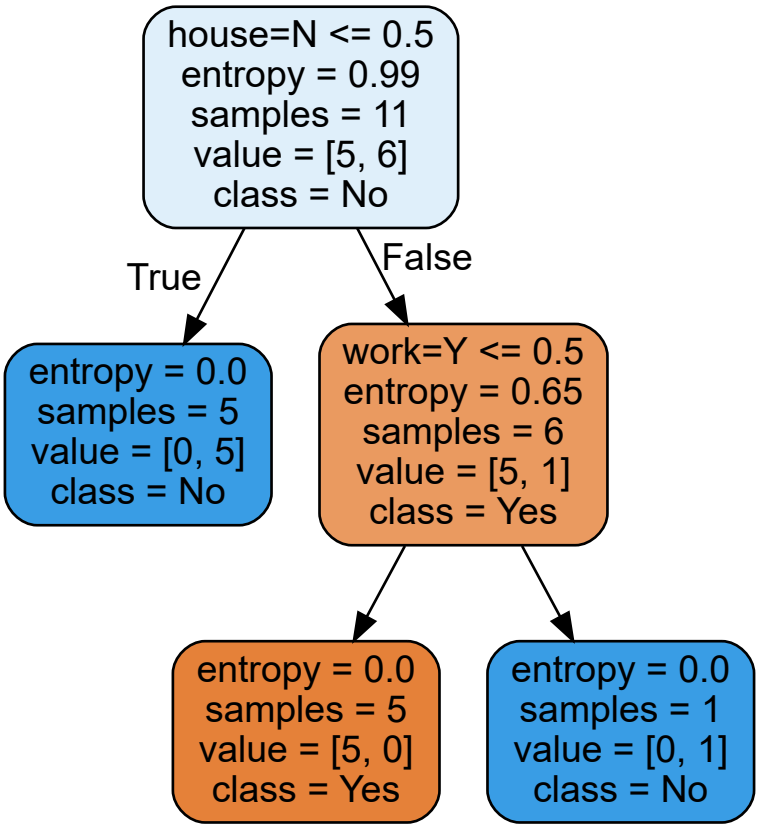
```
from sklearn.feature_extraction import DictVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from graphviz import Source
from sklearn import tree
```

```
# 读取贷款数据，第一列为索引
data = pd.read_csv("load.csv", index_col=0)

# 转化为字典格式
vec = DictVectorizer()
X = vec.fit_transform(data[['age', 'work', 'house', 'credit']].to_dict(orient="record"))
y = data['label']
```

```
# C4.5
dct = DecisionTreeClassifier(criterion='entropy')
dct.fit(X, y.values)
```

```
DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

```
# 可视化
graph = Source(tree.export_graphviz(
                                    dct,
                                    out_file=None,
                                    rounded=True,proportion = False,
                                    feature_names = vec.feature_names_,
                                    precision  = 2,
                                    class_names=["Yes","No"],
                                    filled = True
                                   )
              )
display(graph)
```



总结

- 决策树具有以下优点:
  - 决策树算法中学习简单的决策规则建立决策树模型的过程非常容易理解，
  - 决策树模型可以可视化，非常直观
  - 应用范围广，可用于分类和回归，而且非常容易做多类别的分类
  - 能够处理数值型和连续的样本特征
- 缺点:
  - 很容易在训练数据中生成复杂的树结构，造成过拟合（overfitting）。剪枝可以缓解过拟合的负作用，常用方法是限制树的高度、叶子节点中的最少样本数量。
  - 学习一棵最优的决策树被认为是NP-Complete问题。实际中的决策树是基于启发式的贪心算法建立的，这种算法不能保证建立全局最优的决策树。Random Forest 引入随机能缓解这个问题。
- sklearn.tree中的DecisionTreeClassifier可通过修改参数criterion修改决策算法。
  - ID3算法十分简单，核心是根据“最大信息熵增益”原则选择划分当前数据集的最好特征，信息熵是信息论里面的概念，是信息的度量方式，不确定度越大或者说越混乱，熵就越大。在建立决策树的过程中，根据特征属性划分数据，使得原本“混乱”的数据的熵(混乱度)减少，按照不同特征划分数据熵减少的程度会不一样。然而，ID3采用的信息增益度量存在一个缺点，它一般会优先选择有较多属性值的Feature,因为属性值多的Feature会有相对较大的信息增益，并且其无法处理连续值。
  - C4.5中是用信息增益比率(gain ratio)来作为选择分支的准则。信息增益比率通过引入一个被称作分裂信息(Split information)的项来惩罚取值较多的Feature。除此之外，C4.5还弥补了ID3中不能处理特征属性值连续的问题。但是，对连续属性值需要扫描排序，会使C4.5性能下降。
  - ID3中根据属性值分割数据，之后该特征不会再起作用，这种快速切割的方式会影响算法的准确率。CART是一棵二叉树，采用二元切分法，每次把数据切成两份，分别进入左子树、右子树。而且每个非叶子节点都有两个孩子，所以CART的叶子节点比非叶子多1。相比ID3和C4.5，CART应用要多一些，既可以用于分类也可以用于回归。CART分类时，使用基尼指数（Gini）来选择最好的数据分割的特征
- 可使用该类内建函数结合graphviz可视化决策树

二叉回归树（平方误差准则）

```
def findIndex(st, nums):
    if len(nums) <= 1: return None
    ans = []
    # 计算各个可能阈值对应的平方误差
    for i in range(len(nums)):
        l = sum(nums[:i]) / i if i != 0 else 0
        r = sum(nums[i:]) / (len(nums) - i)
        sl = sum([(nums[j] - l)**2 for j in range(i)])
        sr = sum([(nums[j] - r)**2 for j in range(i, len(nums))])
        ans.append(sl + sr)
    # 获取最小误差的索引
    index = ans.index(min(ans))
    # 输出
    print(nums[:index], ' <---> ',nums[index:])
    print('index', st + index, 'value', nums[index], 'min',
          min(ans))
    # 递归计算子树
    lIndex, rIndex = findIndex(st, nums[:index]), findIndex(
        st + index, nums[index:])
```

```
nums = [4.5, 4.75, 4.91, 5.34, 5.8, 7.05, 7.9, 8.23, 8.7, 9.0]
findIndex(0, nums)
```

```
[4.5, 4.75, 4.91, 5.34, 5.8]  <--->  [7.05, 7.9, 8.23, 8.7, 9.0]
index 5 value 7.05 min 3.358719999999999
[4.5, 4.75, 4.91]  <--->  [5.34, 5.8]
index 3 value 5.34 min 0.19120000000000004
[4.5]  <--->  [4.75, 4.91]
index 1 value 4.75 min 0.012800000000000023
[4.75]  <--->  [4.91]
index 2 value 4.91 min 0.0
[5.34]  <--->  [5.8]
index 4 value 5.8 min 0.0
[7.05, 7.9]  <--->  [8.23, 8.7, 9.0]
index 7 value 8.23 min 0.6625166666666666
[7.05]  <--->  [7.9]
index 6 value 7.9 min 0.0
[8.23]  <--->  [8.7, 9.0]
index 8 value 8.7 min 0.04500000000000021
[8.7]  <--->  [9.0]
index 9 value 9.0 min 0.0
```

总结：

- 决策树分为分类树和回归树，前者用于分类，如晴天/阴天/雨天、用户性别、邮件是否是垃圾邮件，后者用于预测实数值，如明天的温度、用户的年龄等。本题为使用平方误差准则的二叉回归树

- 具体实现对于每一个结点(根节点，数据为整个数据集),遍历所有可能的分界点，找出最小的平方误差对应的分界点，切分数据，再对左右数据集继续递归调用该函数，直至完全分离或者达到停止条件。

集成学习

`定义`:通过将多个单个学习器集成/组合在一起，使它们共同完成学习任务,集成学习能够把多个单一学习模型所获得的多个预测结果进行有机地组合，从而获得更加准确、稳定和强壮的最终结果。

`分类`:

> Boosting:采用串行方式训练及分类其，各个分类器之间有依赖关系。基本思路是将基分类器层层叠加，每一层在训练的时候，对前一层基分类器分错的样本，给予更高的权重。测试时，根据跟层分类器的结果的加权得到最终结果。

> Bagging:与Boosting的串行训练方式不同，Bagging方法在训练过程中，各基分类器之间无强依赖甚至无依赖，可以进行并行训练。以随机森铃为例，为了让基分类器之间相互独立，将训练集分为若干子集。其更像是一种集体决策的过程，每个个体进行单独学习，最终通过投票的方式作出最后的决策。

- Bagging的训练集是随机的，以独立同分布选取的训练样本子集训练弱分类器，而Boosting训练集的选择不是独立的，每一次选择的训练集都依赖于上一次学习的结果，根据错误率取样，因此Boosting的分类精度在大多数数据集中要优于Bagging，但是在有些数据集中，由于过拟合的原因，Boosting的精度会退化。
- Bagging的每个预测函数(即弱假设)没有权重，而Boosting根据每一次训练的训练误差得到该次预测函数的权重；
- Bagging的各个预测函数可以并行生成，而Boosting的只能顺序生成。
- 从方差和偏差的角度再理解集成学习。
    - 基分类器的错误，是偏差和方差两种错误之和，偏差主要是由于分类器的表达你能力有限导致的系统性错误，表现在训练误差不收敛。方差是由于分类器对于样本分布过于敏感，导致在训练样本数较少时，产生过拟合。
    - Boosting 通过逐步聚焦于基分类器分错的样本，减少的集成分类器的偏差。
    - Bagging方法则是采用分而治之的策略，通过对训练样本多次采样，并分别训练处多个不同模型，然后做综合，来减少集成分类器的方差。假设所有基分类器出错的概率是独立的，在某个测试样本上，用简单多数投票方法来集成结果，超过多数基分类器出错的概率会随着基分类器的数量增加而下降。
- Adaboost 实现

```
class AdaBoost:
    def __init__(self, n_estimators=50, learning_rate=1.0):
        self.clf_num = n_estimators
        self.learning_rate = learning_rate

    def init_args(self, datasets, labels):

        self.X = datasets
        self.Y = labels
        self.M, self.N = datasets.shape
        # 弱分类器数目和集合
        self.clf_sets = []
        # 初始化weights
        self.weights = [1.0/self.M] * self.M
```

```
        # G(x)系数 alpha
        self.alpha = []

    def _G(self, features, labels, weights):
        m = len(features)
        error = 100000.0  # 无穷大
        best_v = 0.0
        features_min = min(features)
        features_max = max(features)
        n_step = (features_max - features_min +
                  self.learning_rate) // self.learning_rate
        direct, compare_array = None, None
        for i in range(1, int(n_step)):
            v = features_min + self.learning_rate * i
            if v not in features:
                # 误分类计算
                compare_array_positive = np.array(
                    [1 if features[k] > v else -1 for k in range(m)])
                weight_error_positive = sum([weights[k] for k in range(
                    m) if compare_array_positive[k] != labels[k]])

                compare_array_nagetive = np.array(
                    [-1 if features[k] > v else 1 for k in range(m)])
                weight_error_nagetive = sum([weights[k] for k in range(
                    m) if compare_array_nagetive[k] != labels[k]])

                if weight_error_positive < weight_error_nagetive:
                    weight_error = weight_error_positive
                    _compare_array = compare_array_positive
                    _direct = 'positive'
                else:
                    weight_error = weight_error_nagetive
                    _compare_array = compare_array_nagetive
                    _direct = 'nagetive'

                if weight_error < error:
                    error = weight_error
                    compare_array = _compare_array
                    best_v = v
                    direct = _direct

        return best_v, direct, error, compare_array

    # 计算alpha
    def _alpha(self, error):
        return 0.5 * np.log((1-error)/error)

    # 规范化因子
    def _Z(self, weights, a, clf):
        return sum([weights[i]*np.exp(-1*a*self.Y[i]*clf[i]) for i in range(self.M)])

    # 权值更新
    def _w(self, a, clf, Z):
```

```
        for i in range(self.M):
            self.weights[i] = self.weights[i]*np.exp(-1*a*self.Y[i]*clf[i]) / Z

    # G(x)的线性组合
    def _f(self, alpha, clf_sets):
        pass

    def G(self, x, v, direct):
        if direct == 'positive':
            return 1 if x > v else -1
        else:
            return -1 if x > v else 1

    def fit(self, X, y):
        self.init_args(X, y)

        for epoch in range(self.clf_num):
            best_clf_error, best_v, clf_result = 100000, None, None
            # 根据特征维度，选择误差最小的
            for j in range(self.N):
                features = self.X[:, j]
                # 分类阈值，分类误差，分类结果
                v, direct, error, compare_array = self._G(
                    features, self.Y, self.weights)

                if error < best_clf_error:
                    best_clf_error = error
                    best_v = v
                    final_direct = direct
                    clf_result = compare_array
                    axis = j
                if best_clf_error == 0:
                    break

            a = self._alpha(best_clf_error)
            self.alpha.append(a)
            self.clf_sets.append((axis, best_v, final_direct))
            Z = self._Z(self.weights, a, clf_result)
            self._w(a, clf_result, Z)
        print(self.clf_sets)

    def predict(self, feature):
        result = 0.0
        for i, clf_set in enumerate(self.clf_sets):
            axis, clf_v, direct = clf_set
            f_input = feature[axis]
            result += self.alpha[i] * self.G(f_input, clf_v, direct)
        return 1 if result > 0 else -1

    def score(self, X_test, y_test):
        right_count = 0
        for i in range(len(X_test)):
            feature = X_test[i]
```

```
            if self.predict(feature) == y_test[i]:
                right_count += 1

        return right_count / len(X_test)
```

```
X = np.arange(10).reshape(10, 1)
y = np.array([1, 1, 1, -1, -1, -1, 1, 1, 1, -1])

clf = AdaBoost(n_estimators=3, learning_rate=0.5)
clf.fit(X, y)
```

```
[(0, 2.5, 'nagetive'), (0, 8.5, 'nagetive'), (0, 5.5, 'positive')]
```

```
clf.score(X, y)
```

```
1.0
```

总结

- Adaboost的原理比较简单，然而代码实现有一些需要注意的地方。
  - 基分类器的定义和保存:课本习题对应的基分类器应至少包括3个属性，阈值以及方向（左边为正样本还是负样本
  - 权值更新，可使用numpy简化计算

EM算法

```
class EM(object):
    def __init__(self, init_p):
        self.A, self.B, self.C = init_p

    def expect(self, i):  # E步
        pt = self.A * (self.B**data[i]) * (1 - self.B)**(1 - data[i])
        pf = (1 - self.A) * (self.C**data[i]) * (1 - self.C)**(1 - data[i])
        return pt / (pt + pf)

    def fit(self, data, f=1e-6):
        l = len(data)
        print('init ', self.A, self.B, self.C)
        for d in range(l):
            # M步
            p = np.array([self.expect(i) for i in range(l)])
            pa = 1. / l * sum(p)
            pb = p.dot(data) / sum(p)
            pc = (1 - p).dot(data) / sum(1 - p)
            print('{} / {} A: {:.4f}, B: {:4f} C: {:.4f}'.\
                format(d+1, l, pa, pb, pc))
            if np.max([self.A-pa, self.B-pb, self.C-pc]) < f:
```

```
                break
            self.A = pa
            self.B = pb
            self.C = pc
```

```
data = [1, 1, 1, 0, 0, 1, 0, 1, 0, 0]
em = EM(init_p=[.5, .5, .5])
f = em.fit(data)
```

```
init  0.5 0.5 0.5
1 / 10 A: 0.5000, B: 0.500000 C: 0.5000
```

总结

- 仍然可以使用numpy简化计算

SVM && ME

数据集读入

```
from sklearn.datasets import load_wine
import seaborn as sns


iris = sns.load_dataset('iris')
iris_x = iris.drop('species', axis=1)
iris_y = iris['species']

wine = load_wine()
wine_x = pd.DataFrame(wine.data, columns=wine.feature_names)
wine_y = pd.DataFrame(wine.target, columns=['label'])['label']
```

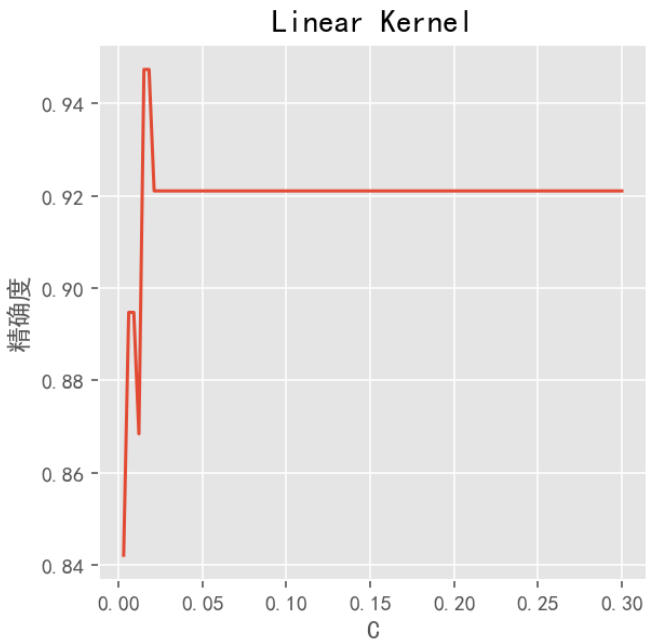
实现

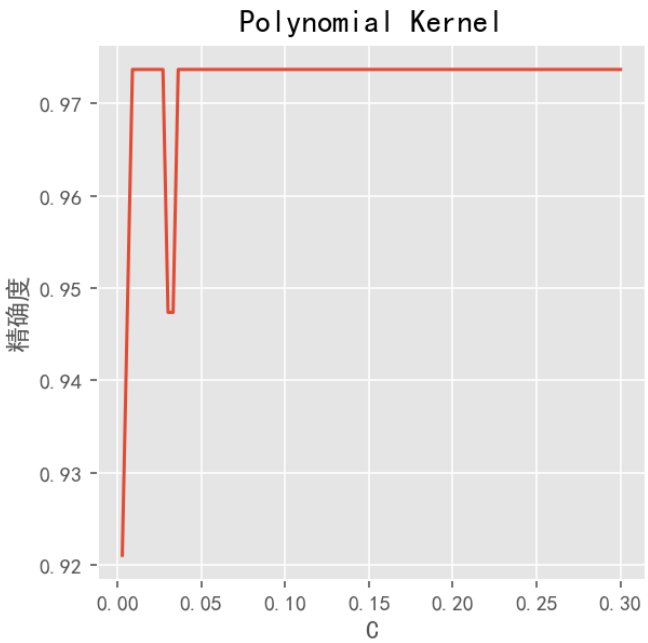
支持向量机

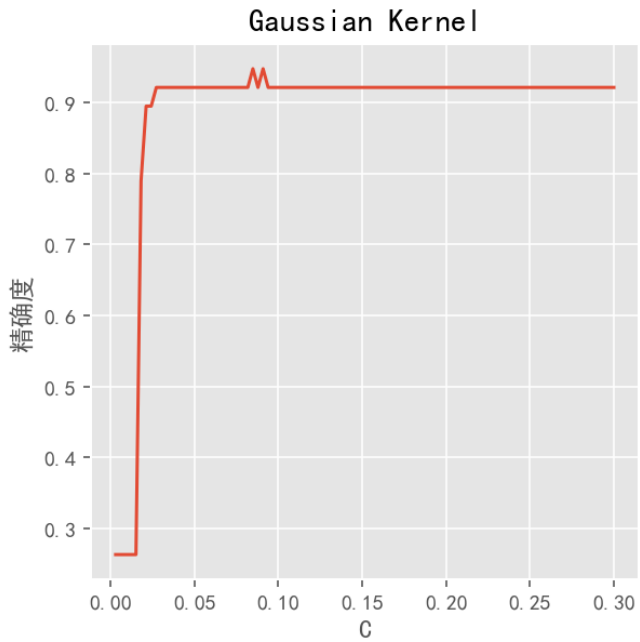
```
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn import svm


def svm_solve(x, y, C_l=0.01, C_r=1):
    x_train, x_test, y_train, y_test = train_test_split(x, y)  # 切分训练集，测试集
    titles = ['Linear Kernel', 'Polynomial Kernel', 'Gaussian Kernel']
    fig = plt.figure(figsize=(10, 10), dpi=144)
    C_space = np.linspace(C_l, C_r, 100)[1:]
    for i, kernel in enumerate(('linear', 'poly', 'rbf')):
```

```
        accuracy = []
        for C in C_space:
            clf = svm.SVC(C=C, kernel=kernel)
            clf.fit(x_train, y_train)
            y_predict = clf.predict(x_test)
            score = metrics.accuracy_score(y_test, y_predict)
            accuracy.append(score)
        plt.subplot(2, 2, i + 1)
        plt.title(titles[i])
        plt.xlabel('C')
        plt.ylabel('精确度')
        plt.plot(C_space, accuracy)
    plt.show()
```

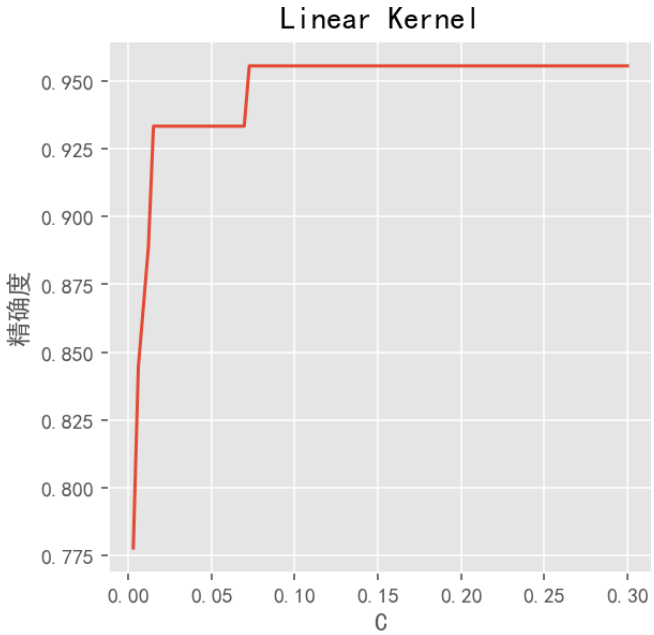
```
svm_solve(iris_x.values, iris_y.values, 0, 0.3)
```

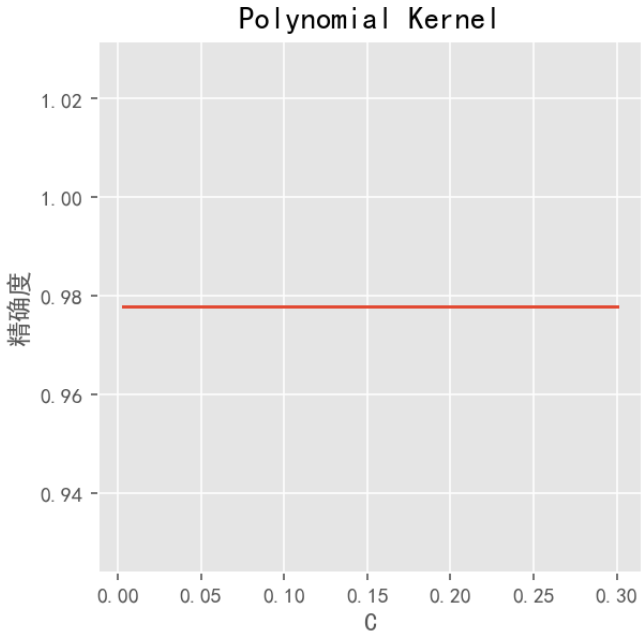


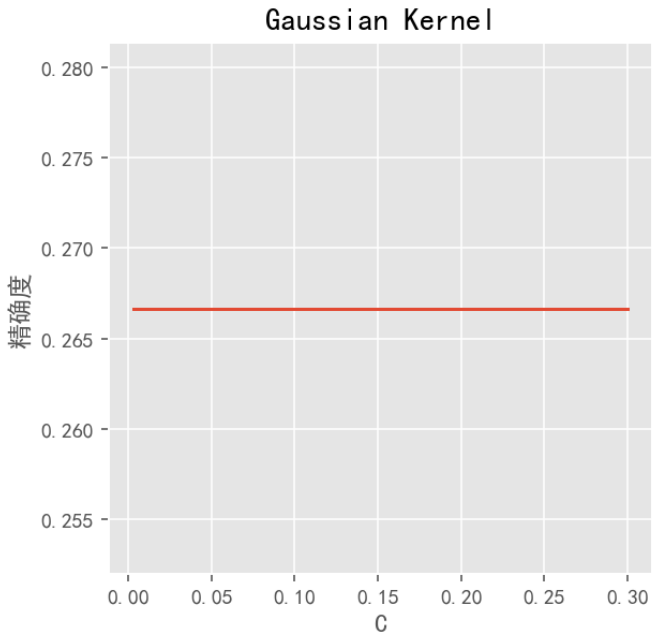




```
svm_solve(wine_x.values, wine_y.values, C_l=0, C_r=0.3)
```



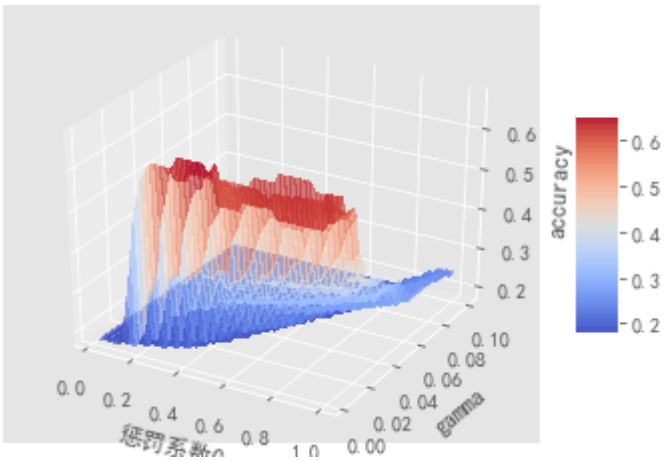




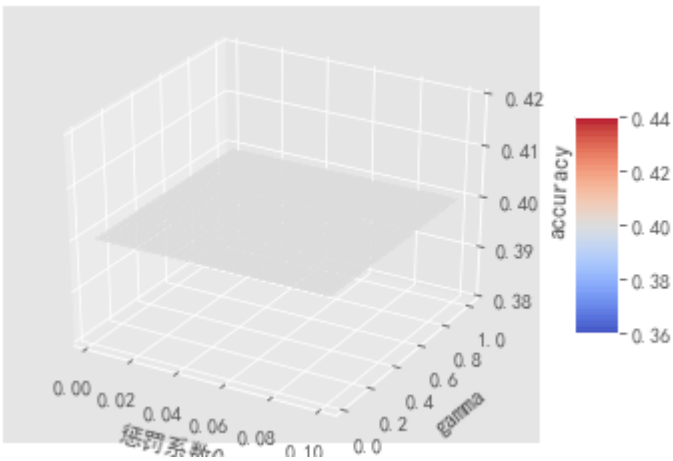

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm


def Guassian(X, y, C_l, C_r, g_l, g_r):
    x_train, x_test, y_train, y_test = train_test_split(X, y)  # 切分训练集，测试集
    C_space = np.linspace(C_l, C_r, 50)[1:]
    g_space = np.linspace(g_l, g_r, 50)[1:]
    accuracy = []
    C_space, g_space = np.meshgrid(C_space, g_space)
    for c, g in zip(np.ravel(C_space), np.ravel(g_space)):
        clf = svm.SVC(C=c, kernel='rbf', gamma=g)
        clf.fit(x_train, y_train)
        y_predict = clf.predict(x_test)
```

```
        score = metrics.accuracy_score(y_test, y_predict)
        accuracy.append(score)
    accuracy = np.array(accuracy).reshape(C_space.shape)
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(
        C_space,
        g_space,
        accuracy,
        cmap=cm.coolwarm,
        linewidth=0,
        antialiased=False,
        alpha=0.6)
    fig.colorbar(surf, shrink=0.5, aspect=5)
    ax.set_xlabel('惩罚系数C')
    ax.set_ylabel('gamma')
    ax.set_zlabel('accuracy')
    plt.show()
```

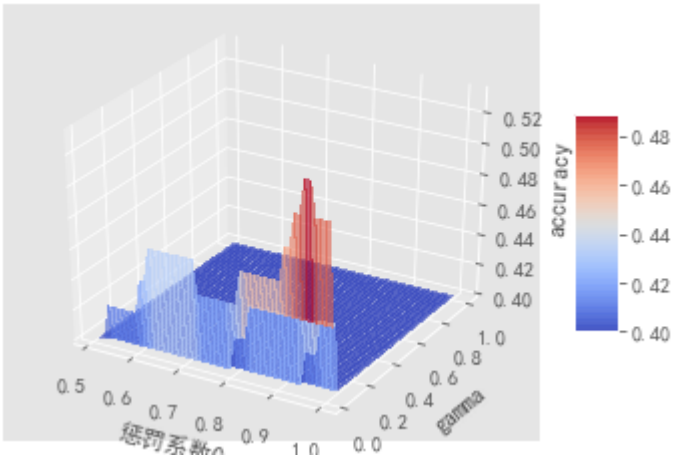
```
Guassian(wine_x.values, wine_y.values, C_l=0, C_r=1, g_l=0, g_r=0.1)
```




```
Guassian(wine_x.values, wine_y.values, C_l=0, C_r=0.1, g_l=0, g_r=1)
```

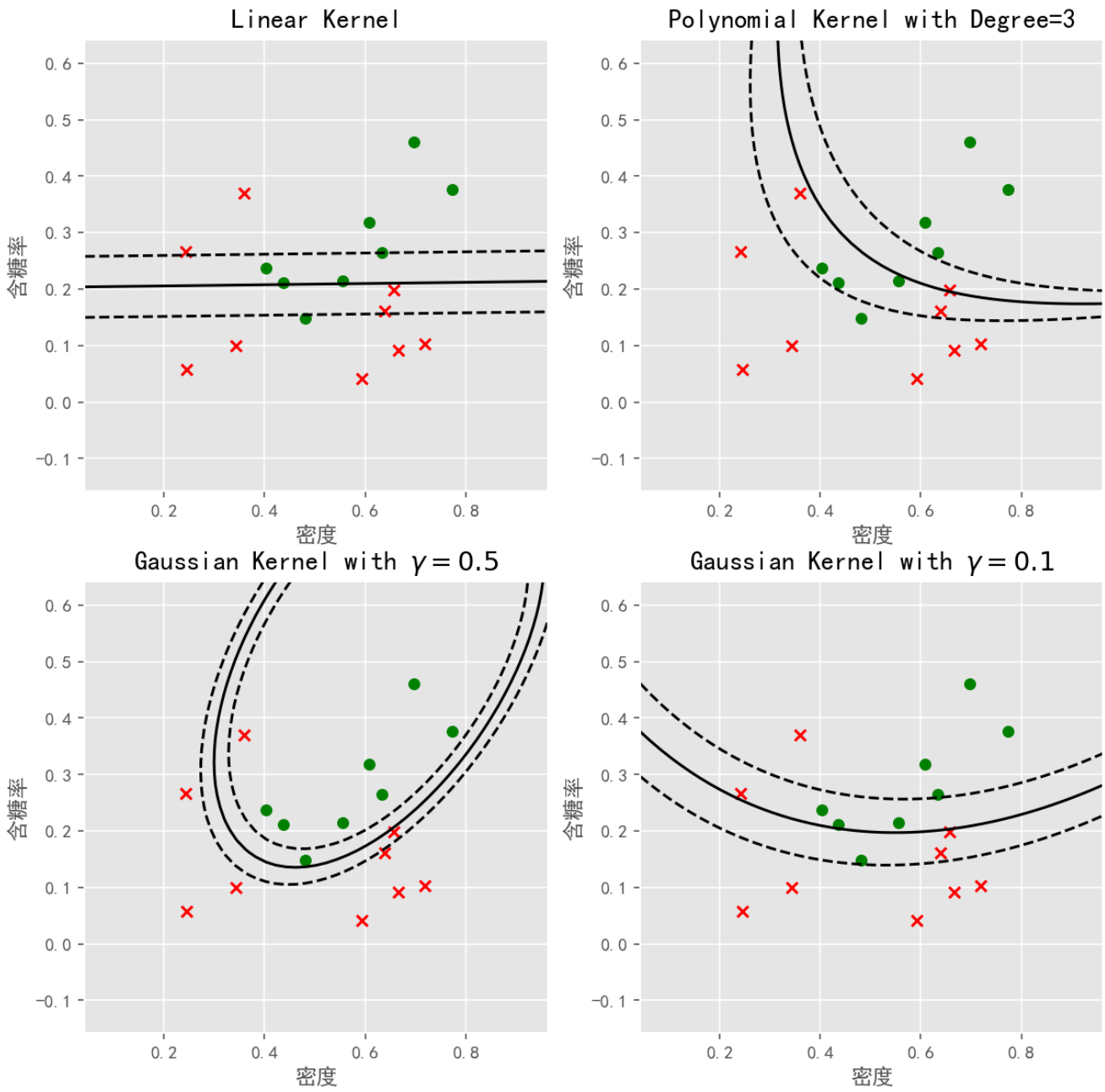



```
Guassian(wine_x.values, wine_y.values, C_l=0.5, C_r=1, g_l=0, g_r=1)
```




```
data = pd.read_table('wmdata.txt', encoding='gb2312', delim_whitespace=True, index_col=0)


clf_linear = svm.SVC(C=1000, kernel='linear')
clf_poly = svm.SVC(C=1000, kernel='poly', degree=3)
clf_rbf = svm.SVC(C=1000, kernel='rbf', gamma=0.5)
clf_rbf2 = svm.SVC(C=1000, kernel='rbf', gamma=0.1)
#创建画布
fig = plt.figure(figsize=(10, 10), dpi=144)

clfs = [clf_linear, clf_poly, clf_rbf, clf_rbf2]
#子图标题
titles = [
    'Linear Kernel', 'Polynomial Kernel with Degree=3',
    'Gaussian Kernel with $\gamma=0.5$', 'Gaussian Kernel with $\gamma=0.1$'
]
#提取数据
X, y = data[['密度', '含糖率']].values, data['好瓜'].values

for clf, i in zip(clfs, range(len(clfs))):
    clf.fit(X, y)  #训练
    subfig = plt.subplot(2, 2, i + 1)
    plt.scatter(
        data[data['好瓜'] == '是']['密度'],
        data[data['好瓜'] == '是']['含糖率'],
        c='green')
    plt.scatter(
        data[data['好瓜'] == '否']['密度'],
        data[data['好瓜'] == '否']['含糖率'],
        c='red',
        marker='x')
    plt.xlabel('密度')
    plt.ylabel('含糖率')

    x_min, x_max = X[:, 0].min() - 0.2, X[:, 0].max() + 0.2
    y_min, y_max = X[:, 1].min() - 0.2, X[:, 1].max() + 0.2
    XX, YY = np.meshgrid(
        np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))
    Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()]).reshape(XX.shape)
```

```
    plt.contour(
        XX,
        YY,
        Z,
        colors=['k', 'k', 'k'],
        linestyles=['--', '-', '--'],
        levels=[-.5, 0, .5])
    plt.title(titles[i])
plt.show()
```



总结：

- 当使用高斯核时，默认参数情况下SVM在葡萄酒数据集下的测试集正确率只有不到50%，然而在训练集中的正确率接近100%,因此我怀疑是过拟合。查看了SVM的文档，发现了两个相关的参数如下：

> C : float, optional (default=1.0) Penalty parameter C of the error term. gamma : float, optional (default='auto') Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. Current default is 'auto' which uses 1 / n_features, if gamma='scale' is passed then it uses 1 / (n_features * X.std()) as value of gamma. The current default of gamma, 'auto', will change to 'scale' in version 0.22. 'auto_deprecated', a deprecated version of 'auto' is used as a default indicating that no explicit value of gamma was passed.

可以看出，

- C是惩罚系数，即对误差的宽容度。c越高，说明越不能容忍出现误差,容易过拟合。C越小，容易欠拟合。C过大或过小，泛化能力变差
- gamma是选择RBF函数作为kernel后，该函数自带的一个参数。隐含地决定了数据映射到新的特征空间后的分布，gamma越大，支持向量越少，gamma值越小，支持向量越多。支持向量的个数影响训练与预测的速度。RBF公式里面的sigma和gamma的关系如下：

$$k(x,z)=\exp(-\frac{d(x,z)^2}{2*\sigma^2})=\exp(-gamma\cdot d(x,z)^2)\Rightarrow gamma=\frac{1}{2\cdot\sigma^2}$$

gamma的物理意义，可考虑RBF的幅宽，它会影响每个支持向量对应的高斯的作用范围，从而影响泛化性能。我的理解：如果gamma设的太大，theta会很小，theta很小的高斯分布长得又高又瘦， 会造成只会作用于支持向量样本附近，对于未知样本分类效果很差，存在训练准确率可以很高，如果让theta无穷小，则理论上，高斯核的SVM可以拟合任何非线性数据，但容易过拟合)而测试准确率不高的可能，就是通常说的过训练；而如果设的过大，则会造成平滑效应太大，无法在训练集上得到特别高的准确率，也会影响测试集的准确率。 可在以上图中看出区别。

以下有两个结论：

- 样本数目少于特征维度并不一定会导致过拟合，这可以参考余凯老师的这句评论：“这不是原因啊，呵呵。用RBF kernel, 系统的dimension实际上不超过样本数，与特征维数没有一个trivial的关系。”
- RBF核应该可以得到与线性核相近的效果（按照理论，RBF核可以模拟线性核），可能好于线性核，也可能差于，但是，不应该相差太多。 当然，很多问题中，比如维度过高，或者样本海量的情况下，大家更倾向于用线性核，因为效果相当，但是在速度和模型大小方面，线性核会有更好的表现。

最大熵

库函数

```
from nltk.classify import MaxentClassifier


def nltk_max_ent(X, y):
    def train_and_test(data, algorithm, X_test, y_test):
        clf = MaxentClassifier.train(data, algorithm)
        correct = 0
        for x, y in zip(X_test, y_test):
            predict = clf.prob_classify(x)
            p = []
            for yi in np.unique(y_test.values):
                p.append((predict.prob(yi), yi))
            p.sort(reverse=True)
            if (p[0][1] == y):
                correct += 1
        print('{} : accuracy : {:.4}'.format(algorithm,
                                             1. * correct / len(y_test)))
```

```
    X = X.to_dict('records')
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    data = [(xi, yi) for xi, yi in zip(X_train, y_train)]
    train_and_test(data, 'GIS', X_test, y_test)
    train_and_test(data, 'IIS', X_test, y_test)
```

```
nltk_max_ent(iris_x, iris_y)
```

```
GIS : accuracy : 0.9474
IIS : accuracy : 0.9211
```

```
nltk_max_ent(wine_x, wine_y)
```

```
GIS : accuracy : 0.7111
IIS : accuracy : 0.7333
```

自编

```
from collections import defaultdict


class MaxEntropy(object):
    def __init__(self):
        self.X = []
        self.Y = set()
        self.w = []
        self.pairs = defaultdict(int)

    def _initparams(self, X, Y):
        self.n = X.shape[0]
        self.M = np.max([len(xi) for xi in X])

        for x, y in zip(X, Y):
            if len(x) < 2:
                continue
            self.X.append(x)
            self.Y.add(y)

            for xi in x:
                self.pairs[(xi, y)] += 1

        self.pairs_size = len(self.pairs)
        self.w = np.zeros(self.pairs_size)
        self._calcu_sample_ep()

    def train(self, X, Y, epochs=200, eps=1e-2):
        self._initparams(X, Y)
        delta_w = None
        for epoch in range(epochs):
```

```
            self._calcu_model_ep()
            delta_w = ([
                np.log(self.sample_ep[i] / self.model_ep[i])
                for i in range(self.pairs_size)
            ]) / self.M
            self.w += delta_w
            if self._convergence(delta_w, eps):
                print("误差范围内")
                break

    def _calcu_sample_ep(self):
        self.sample_ep = np.zeros(self.pairs_size)
        for i, pair in enumerate(self.pairs):
            self.sample_ep[i] += self.pairs[pair] / self.n
            self.pairs[pair] = i

    def _calcu_model_ep(self):
        self.model_ep = np.zeros(self.pairs_size)
        for x in self.X:
            prob = self._condictional_p(x)
            for xi in x:
                for pyx, yi in prob:
                    if (xi, yi) in self.pairs:
                        id = self.pairs[(xi, yi)]
                        self.model_ep[id] += 1. / self.pairs_size * pyx

    def _condictional_p(self, x):
        p = [self._calcu_condictional_p(x, yi) for yi in self.Y]
        return list(zip(p / sum(p), self.Y))

    def _calcu_condictional_p(self, x, yi):
        sum = 0.0
        for xi in x:
            if (xi, yi) in self.pairs:
                sum += self.w[self.pairs[(xi, yi)]]
        return np.exp(sum)

    def predict(self, x):
        p = self._condictional_p(x)
        p.sort(reverse=True)
        return p

    def score(self, X, y):
        correct = 0
        for xi, yi in zip(X, y):
            if self.predict(xi)[0][1] == yi:
                correct += 1
#             else:
#                 print('features:{} label:{}'.format(xi, yi))
        print("accuracy: {:.4}".format(1. * correct / len(y)))

    def _convergence(self, delta_w, eps):
        return np.max(delta_w) <= eps
```

```
maxEnt = MaxEntropy()
x_train, x_test, y_train, y_test = train_test_split(iris_x.values, iris_y)
maxEnt.train(x_train, y_train, epochs=1000, eps=1e-2)
maxEnt.score(x_test, y_test)
```

```
accuracy: 0.8158
```

```
def to_str(data):
    q = data
    for i, feature in enumerate(data.columns):
        data[feature] = data[feature].apply(lambda x: str(i)+'_'+str(x))
    return data
```

```
wine_x_str = to_str(wine_x)
x_train, x_test, y_train, y_test = train_test_split(wine_x_str.values, wine_y)
```

```
maxEnt = MaxEntropy()
maxEnt.train(x_train, y_train, epochs=200, eps=1e-8)
maxEnt.score(x_test, y_test)
```

```
accuracy: 0.8
```

总结：

- 一些问题: 最大熵应该是此次实验中最麻烦的一个实验。当然前提是SVM直接调库实现，毕竟SMO优化算法我实在不会，寒假自己再尽量试试能否编出来吧。一开始我自己编的最大熵所使用的数据集是离散型的，效果不错，代码应该是没有错误的。然而在使用iris以及wine数据集时，正确率不到10%,比随机猜的33%正确率还差，我便怀疑是代码的问题。于是先去使用nltk库中的classify.MaxentClassifier 最大熵模型验证数据集是否有误，结果发现其正确率只有3%，于是我有理由怀疑是数据集的问题。在思考了最大熵GIS优化算法的原理后，我怀疑最大熵不适合连续性数据，因为其需要统计各维特征数据的出现次数，并以此为依据分类。而连续性数据甚至可能连重复的数据都没有。直到我与同学的讨论中被告知同学的最大熵在iris数据集（连续型）上的实现效果不坏后，我才重新思考这个问题。再输出了数据的出现次数后，我发现在不少数据集中，我的担心显然有点过了—— 重复出现的数据还是很多的。再进一步检查后，我才发现是我标签格式的问题，在改正过后，其正确率终于回归80%以上,在nltk中更有更高的正确率。
- nltk最大熵模型训练函数的参数train_toks格式不同于常见的特征数据，标签数据分开，因此需要特别处理，可使用pandas中的to_dict函数简单完成数据集到字典的转换。

隐马尔可夫模型

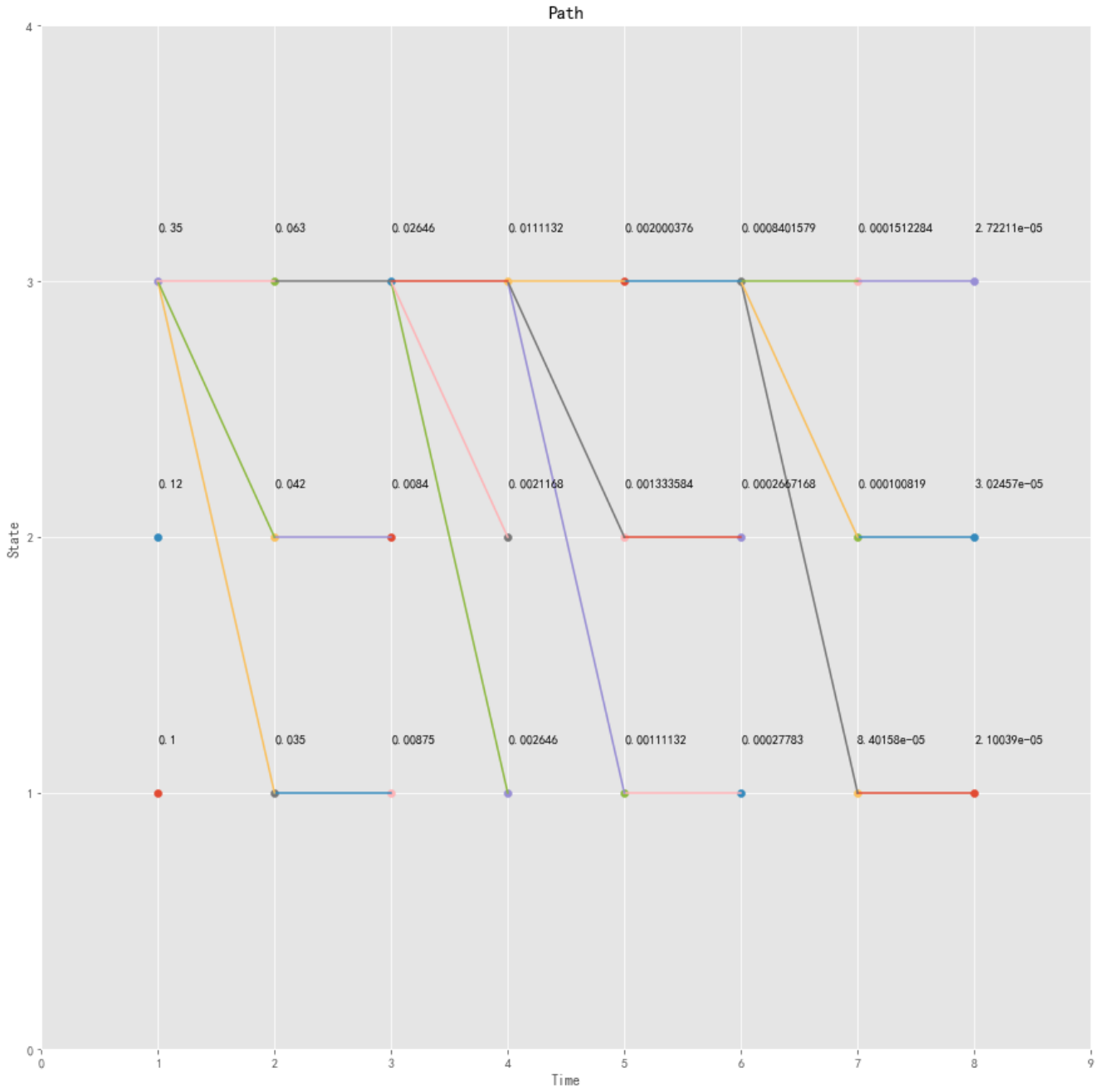
```
class HiddenMarkov(object):
    def forward(self, Q, V, A, B, O, PI):
        '''
            前向算法
        '''
        N, M = len(Q), len(O)
```

```
        alpha = np.zeros((M, N))
        for t in range(M):
            index = V.index(O[t])
            if t == 0:
                alpha[t] = PI * B.T[index]
            else:
                alpha[t] = [np.dot(alpha[t-1], A.T[j])
                            for j in range(N)] * B.T[index]
        p = np.sum(alpha[M-1])
        print('Forward matric:\n', alpha)
        print('P(O|model) = ', p)
        return alpha

    def backward(self, Q, V, A, B, O, PI):
        '''
            后向算法
        '''
        N, M = len(Q), len(O)
        betas = np.zeros((M, N))
        betas[M-1] = [1, ] * N
        for t in range(M-2, -1, -1):
            index = V.index(O[t+1])
            betas[t] = [np.dot(betas[t+1] * A[j], B.T[index])
                        for j in range(N)]
        p = np.dot(PI * B.T[V.index(O[0])], betas[0])
        print('Backward matric:\n', betas)
        print('P(O|model) = ', p[0])
        return betas

    def viterbi(self, Q, V, A, B, O, PI):
        '''
            维特比算法求最优路径
        '''
        N, M = len(Q), len(O)
        state = np.zeros((M, N), dtype=np.int8)
        delta = np.zeros((M, N))

        for t in range(M):
            index = V.index(O[t])
            if t == 0:
                delta[t] = PI * B.T[index]
            else:
                p = np.array([(delta[t-1] * A.T[j] * B[j][index])
                              for j in range(N)])
                state[t] = (p.argmax(axis=1))
                delta[t] = p.max(axis=1)

        print('Possibility matric :\n', delta)
        print("Last most possible state matric :\n", state)

        self.optimal_path(delta, state, M, N)
        self.show_sequence(delta, state, M, N)
```

```
    def calcu_p(self, Q, V, A, B, O, PI, t, s):
        '''计算P(i_t = q_s|Q, lambda)
            params:
                t: 时刻
                s: 状态
        '''
        alpha = self.forward(Q, V, A, B, O, PI)
        betas = self.backward(Q, V, A, B, O, PI)
        print("alpha\n{}\nbetas\n{}".format(alpha, betas))
        t -= 1
        s -= 1
        ans = alpha[t][s] * betas[t][s] / (alpha[t].dot(betas[t]))
        print('P(i{} = q{} | O, lambda): {}'.format(t+1, s+1, ans))

    def optimal_path(self, delta, state, M, N):
        '''
            求最优路径
        '''
        path = []
        end = delta.argmax(axis=1)[-1]
        path.append(end)
        best = end
        for row in state[-1:0:-1]:
            best = row[best]
            path.append(best)
        path = np.array(path[::-1]) + 1
        print('Optimal path is ', path)
        return path

    def show_sequence(self, pmatric, state, M, N):
        '''
            可视化
        '''
        plt.figure(figsize=(15, 15))
        np.set_printoptions(precision=4)
        for i in range(M):
            for j in range(N):
                plt.scatter(i+1, j+1)                    # 画点
                plt.annotate(str(round(pmatric[i][j], 10)), xy=(   # 标注概率
                    i+1, j+1), xytext=(i+1, j+1.2))

        for i in range(M-1, 0, -1):
            for j in range(N):
                plt.plot([i+1, i], [j+1, state[i][j]+1])    # 连线

        plt.xticks(range(pmatric.shape[0]+2))     # 横坐标范围
        plt.yticks(range(pmatric.shape[1]+2))     # 纵坐标范围
        plt.title("Path")
        plt.xlabel("Time")
        plt.ylabel('State')
        plt.show()
```

```
"""
Q:状态集合
V:所有可能观测的集合
A:状态转移概率矩阵
B:观测概率矩阵
O:观测序列
Pi:初始状态概率矩阵
"""

Q = [1, 2, 3]
V = ['red', 'white']
A = np.array([[0.5, 0.1, 0.4], [0.3, 0.5, 0.2], [0.2, 0.2, 0.6]])
B = np.array([[0.5, 0.5], [0.4, 0.6], [0.7, 0.3]])
O = ['red', 'white', 'red', 'red', 'white', 'red', 'white', 'white']
PI = np.array([[0.2, 0.3, 0.5]])

HMM = HiddenMarkov()
HMM.viterbi(Q, V, A, B, O, PI)
HMM.calcu_p(Q, V, A, B, O, PI, 3, 1)
```

```
'\nQ:状态集合\nV:所有可能观测的集合\nA:状态转移概率矩阵\nB:观测概率矩阵\nO:观测序列\nPi:初始状态概率矩阵
\n'

Possibility matric :
 [[1.00000000e-01 1.20000000e-01 3.50000000e-01]
 [3.50000000e-02 4.20000000e-02 6.30000000e-02]
 [8.75000000e-03 8.40000000e-03 2.64600000e-02]
 [2.64600000e-03 2.11680000e-03 1.11132000e-02]
 [1.11132000e-03 1.33358400e-03 2.00037600e-03]
 [2.77830000e-04 2.66716800e-04 8.40157920e-04]
 [8.40157920e-05 1.00818950e-04 1.51228426e-04]
 [2.10039480e-05 3.02456851e-05 2.72211166e-05]]
Last most possible state matric :
 [[0 0 0]
 [2 2 2]
 [0 1 2]
 [2 2 2]
 [2 2 2]
 [0 1 2]
 [2 2 2]
 [0 1 2]]
Optimal path is  [3 3 3 3 3 3 2 2]
```




```
Forward matric:
 [[0.1    0.12   0.35  ]
 [0.078  0.084  0.0822]
 [0.0403 0.0265 0.0681]
 [0.0209 0.0124 0.0436]
 [0.0114 0.0102 0.0111]
 [0.0055 0.0034 0.0093]
 [0.0028 0.0025 0.0025]
 [0.0013 0.0012 0.0009]]
P(O|model) =  0.0034767094492823996
Backward matric:
 [[0.0063 0.0068 0.0058]
 [0.0148 0.0123 0.0157]
 [0.0256 0.0234 0.0268]
 [0.0459 0.0528 0.0428]
```

```
 [0.1055 0.1009 0.1119]
 [0.1861 0.2415 0.1762]
 [0.43   0.51   0.4   ]
 [1.     1.     1.    ]]
P(O|model) =  0.0034767094492824
alpha
[[0.1    0.12   0.35  ]
 [0.078  0.084  0.0822]
 [0.0403 0.0265 0.0681]
 [0.0209 0.0124 0.0436]
 [0.0114 0.0102 0.0111]
 [0.0055 0.0034 0.0093]
 [0.0028 0.0025 0.0025]
 [0.0013 0.0012 0.0009]]
betas
[[0.0063 0.0068 0.0058]
 [0.0148 0.0123 0.0157]
 [0.0256 0.0234 0.0268]
 [0.0459 0.0528 0.0428]
 [0.1055 0.1009 0.1119]
 [0.1861 0.2415 0.1762]
 [0.43   0.51   0.4   ]
 [1.     1.     1.    ]]
P(i3 = q1 | O, lambda): 0.2964750117651477
```

总结:

- 这是我实现得最为完整的一个模型，包括概率计算中的前向算法，后向算法，求解最优路径的维特比算法，特定时刻特定状态的概率以及路径的可视化。虽然是在刚教隐马模型时实现的，但也花费了3个多小时的实现，当然收获更大，不仅对运算库的使用技巧有更深入的了解，对模型的理解也更加透彻。
- 具体实现中，仍然使用numpy简化计算，另外，对标准格式的观测概率矩阵，在运算时可取其转置矩阵中的一行进行运算，达到简化作用
- 在维特比函数中，通过记录可能路径对应概率以及前一状态，可记录得完整路径，以此画出对应图像，更可逆向计算出最优路径

线性链条件随机场

```
from itertools import product


class CRF(object):
    def encode(self, start, stop, time, state):
        # 获得所有可能序列
        s = list(
            map(lambda x: list(x),
                list(product(list(range(state)), repeat=time - 1))))
        for si in s:
            si.insert(0, start - 1)
            si.append(stop - 1)
        return np.array(s, dtype=np.int8)
```

```
    def forward(self, start, stop, *args):
        '''
            前向算法
        '''
        M = np.array(args)
        time, state, _ = M.shape

        sequences = self.encode(start, stop, time, state)
        p = 1 + np.zeros(len(sequences))
        for i, s in enumerate(sequences):
            for j in range(len(s) - 1):
                p[i] *= M[j][s[j]][s[j + 1]]
        print('sum', sum(p))
        p /= sum(p)

        for i, pi in enumerate(p):
            print('{} --> {:.4f}'.format(sequences[i] + 1, pi))

        print("The most possible sequence: ", sequences[p.argmax()] + 1)
```

```
M1 = [[0, 0], [.3, .7]]
M2 = [[.3, .7], [.7, .3]]
M3 = [[.5, .5], [.6, .4]]
M4 = [[0, 1], [0, 1]]
crf = CRF()
crf.forward(2, 2, M1, M2, M3, M4)
```

```
 sum 0.9999999999999999
 [2 1 1 1 2] --> 0.0450
 [2 1 1 2 2] --> 0.0450
 [2 1 2 1 2] --> 0.1260
 [2 1 2 2 2] --> 0.0840
 [2 2 1 1 2] --> 0.2450
 [2 2 1 2 2] --> 0.2450
 [2 2 2 1 2] --> 0.1260
 [2 2 2 2 2] --> 0.0840
 The most possible sequence:  [2 2 1 1 2]
```

总结：

- 条件随机场的最可能序列效率最高的算法是维特比，考虑题中要求列出所有可能序列的对应概率，因此遍历所有序列是必要的。
- 上述模型的函数实现了一般化，可求解任意符合规范的转移概率矩阵，先通过itertools中的product求得所有可能的序列，再以此遍历转移概率矩阵，相乘求得该序列的概率

大作业总结

> 在总结了各小题后，总结一下此次大作业的收获。虽然大多数代码都是以前实现过得，但是所谓温故知新，还是大有收获。

- 首先是发现之前代码上的一些小问题，比如在感知机中的对偶算法中发现了一个小bug，在之前的数据集中并没有体现出来，终于在这一次中出问题了。
- 然后是对旧代码的一些整理，不仅是对代码风格的改进，更是用上了自己新学的一些代码技巧，库函数技巧。在提高代码可读性的同时也提高了运行效率。
- 对于调库实现的模型，会去深入探究其参数的意义，学习其内部优化，由此对模型的理解，函数的封装，异常的处理也更加了解，也明白了编写易于理解的代码的重要性。
- 明白了和同学讨论交流的重要性，通过与同学的交流学习，可以从更多角度理解算法，模型，既能学习他人，也能更正自己理解上的错误。