

# 机器学习与数据挖掘-上机作业一

## 上机任务：

- 1、复现上机程序中的 topic 01, topic02 代码。熟悉 pandas、matplotlib 的基本操作。
- 2、读入 trial.csv 数据，生成以下图：欺诈类的各个 feature 特征描述性统计；欺诈类与非欺诈类的各个 feature 特征的差异性描述。
- 3、基于 trial.csv 与可视化结果，对影响是否欺诈的因素做出简单结论。提交文档与代码截图。

## 1 数据的特征描述统计

### 1.1 欺诈类的各 feature 特征描述性统计

#### 1.1.1 数据读入与总体特征获取

##### (1) 读入数据

```
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
warnings.filterwarnings('ignore')

#读入文件
df = pd.read_csv(
    r'D:\XJY\大三上\专业课程\机器学习与数据挖掘-翁克瑞\上机\trial.csv')

#展示前五行数据
df.head()
```

	Sector_score	LOCATION_ID	PARA_A	SCORE_A	PARA_B	SCORE_B	TOTAL	numbers	Marks	Money_Value	MONEY_Mark
0	3.89	23	4.18	6	2.50	2	6.68	5.0	2	3.38	
1	3.89	6	0.00	2	4.83	2	4.83	5.0	2	0.94	
2	3.89	6	0.51	2	0.23	2	0.74	5.0	2	0.00	
3	3.89	6	0.00	2	10.80	6	10.80	6.0	6	11.75	
4	3.89	6	0.00	2	0.08	2	0.08	5.0	2	0.00	

##### (2) 获取数据总体信息

```
#获取数据总体信息
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 773 entries, 0 to 772
Data columns (total 18 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   Sector_score    773 non-null   float64
 1   LOCATION_ID     773 non-null   int64  
 2   PARA_A          773 non-null   float64
 3   SCORE_A         773 non-null   int64  
 4   PARA_B          773 non-null   float64
 5   SCORE_B         773 non-null   int64  
 6   TOTAL           773 non-null   float64
 7   numbers         773 non-null   float64
 8   Marks           773 non-null   int64  
 9   Money_Value     773 non-null   float64
 10  MONEY_Marks    773 non-null   int64  
 11  District        773 non-null   int64  
 12  Loss            773 non-null   int64  
 13  LOSS_SCORE      773 non-null   int64  
 14  History         773 non-null   int64  
 15  History_score   773 non-null   int64  
 16  Score           773 non-null   float64
 17  Risk            773 non-null   int64  
dtypes: float64(7), int64(11)
memory usage: 108.8 KB
```

### (3) 获取数据总体基本统计学特征

```
#获取数据总体基本统计学特征
df.describe()
```

	Sector_score	LOCATION_ID	PARA_A	SCORE_A	PARA_B	SCORE_B	TOTAL	numbers	Marks	MoI
<b>count</b>	773.000000	773.000000	773.000000	773.000000	773.000000	773.000000	773.000000	773.000000	773.000000	7
<b>mean</b>	20.255149	14.856404	2.457983	3.518758	10.841903	3.135834	13.268062	5.067917	2.238034	
<b>std</b>	24.339709	9.891317	5.688509	1.741366	50.176308	1.699869	51.406241	0.264928	0.804941	
<b>min</b>	1.850000	1.000000	0.000000	2.000000	0.000000	2.000000	0.000000	5.000000	2.000000	
<b>25%</b>	2.370000	8.000000	0.210000	2.000000	0.000000	2.000000	0.540000	5.000000	2.000000	
<b>50%</b>	3.890000	13.000000	0.880000	2.000000	0.410000	2.000000	1.390000	5.000000	2.000000	
<b>75%</b>	55.570000	19.000000	2.480000	6.000000	4.160000	4.000000	7.760000	5.000000	2.000000	
<b>max</b>	59.850000	44.000000	85.000000	6.000000	1264.630000	6.000000	1268.910000	9.000000	6.000000	9

## 1.1.2 分类统计

我们发现除了 Risk 类之外，原数据的 feature 主要分两类：一种是数值类数据，一种是分类型数据。对于分类型数据，研究其均值、变化趋势等是没有意义的。详情可见下表。

数值型数据
Sector_score
PARA_A
SCORE_A
PARA_B
SCORE_B

TOTAL
numbers
Marks
Money_Value
LOSS_SCORE
History_score
Score

分类型数据
LOCATION_ID
Loss
History
District
Money_Marks

(1) 获取欺诈类与非欺诈类的频数和比例

```
#获取欺诈类和非欺诈类频数
df['Risk'].value_counts()
```

```
1    486
0    287
Name: Risk, dtype: int64
```

```
#获取欺诈类和非欺诈类比例
df['Risk'].value_counts(normalize=True)
```

```
1    0.628719
0    0.371281
Name: Risk, dtype: float64
```

(2) 获取欺诈类与非欺诈类各 feature 均值

可以看出这两类数据在 Sector\_score、PARA\_A、SCORE\_A、PARA\_B、SCORE\_B、TOTAL、Money\_Value、MONEY\_Marks、Loss、History、Score 这些特征值之间存在显著差异，这也将是之后我们会重点分析的 feature。

	#获取欺诈类各feature均值 df[df['Risk'] == 1].mean()	#获取非欺诈类各feature均值 df[df['Risk'] == 0].mean()	
Sector_score	13.152181	Sector_score	32.283171
LOCATION_ID	15.226337	LOCATION_ID	14.229965
PARA_A	3.732162	PARA_A	0.300314
SCORE_A	4.415638	SCORE_A	2.000000
PARA_B	17.094630	PARA_B	0.253662
SCORE_B	3.806584	SCORE_B	2.000000
TOTAL	20.776174	TOTAL	0.553976
numbers	5.108025	numbers	5.000000
Marks	2.378601	Marks	2.000000
Money_Value	22.387140	Money_Value	0.262627
MONEY_Marks	3.452675	MONEY_Marks	2.000000
District	2.806584	District	2.000000
Loss	0.047325	Loss	0.000000
LOSS_SCORE	2.098765	LOSS_SCORE	2.000000
History	0.166667	History	0.000000
History_score	2.267490	History_score	2.000000
Score	3.121811	Score	2.000000
Risk	1.000000	Risk	0.000000
dtype:	float64	dtype:	float64

(3) 查看不同的 LOCATION\_ID 中欺诈类和非欺诈类的数量，以及计算其比例并排序

可以看出，欺诈类数量比例最高的 LOCATION\_ID 为 8，占到了近 11.7%。

```
#查看不同的LOCATION_ID中欺诈类和非欺诈类的数量
L_ID = pd.crosstab(df['LOCATION_ID'], df['Risk'])
```

L\_ID

	Risk 0 1	
LOCATION_ID	0	1
1	1	10
2	6	35
3	1	2
4	18	19
5	27	17
6	22	11
7	1	3
8	19	57
9	27	26
11	9	17
12	20	27
13	14	21
14	9	11
15	10	25
16	17	35
17	1	0
18	7	9
19	21	47
20	0	5

```

#计算每个州的欺诈类比例并排序
#可以看出，比例最高的LOCATION_ID为8，占到了近11.7%
L_ID['RATE'] = L_ID[1]/486
L_ID.sort_values(by='RATE', ascending=False).head()

```

	Risk	0	1	RATE
LOCATION_ID				
8	19	57	0.117284	
19	21	47	0.096708	
2	6	35	0.072016	
16	17	35	0.072016	
12	20	27	0.055556	

(4) 查看不同的 Loss 中欺诈类和非欺诈类的数量，以及计算其比例并排序

可以看出，Loss 值为 0 时，欺诈类占比超过 95%。

```

#查看不同的Loss中欺诈类和非欺诈类的数量
#可以看出，Loss值为0时，欺诈类占比超过95%
LOSS=pd.crosstab(df['Loss'], df['Risk'])
LOSS['RATE'] = LOSS[1]/486
LOSS

```

	Risk	0	1	RATE
Loss				
0	287	465	0.956790	
1	0	19	0.039095	
2	0	2	0.004115	

(5) 查看不同的 District 中欺诈类和非欺诈类的数量，以及计算其比例并排序

可以看出，District 值为 2 时，欺诈类占比接近 75%。

```

#查看不同的District中欺诈类和非欺诈类的数量
#可以看出，District值为2时，欺诈类占比接近75%
DIST=pd.crosstab(df['District'], df['Risk'])
DIST['RATE'] = DIST[1]/486
DIST

```

	Risk	0	1	RATE
District				
2	287	363	0.746914	
4	0	50	0.102881	
6	0	73	0.150206	

(6) 查看不同的 Money\_Marks 中欺诈类和非欺诈类的数量，以及计算其比例并排序

可以看出，Money\_Marks 值为 2 时，欺诈类占比超过 58%。

```
#查看不同的Money_Marks中欺诈类和非欺诈类的数量
#可以看出, Money_Marks值为2时, 欺诈类占比超过58%
M_M=pd.crosstab(df['MONEY_MARKS'], df['Risk'])
M_M['RATE']=M_M[1]/486
M_M
```

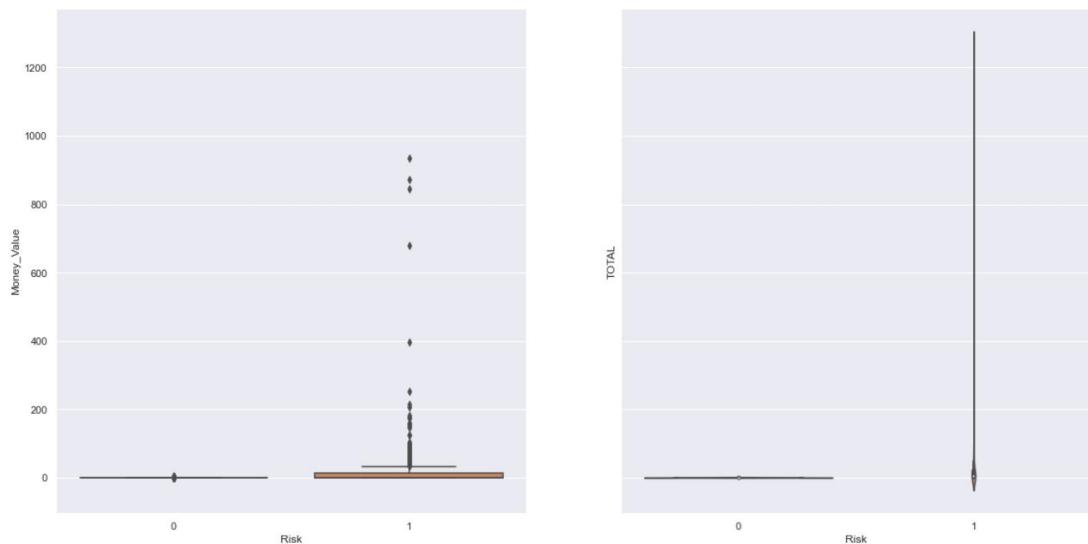
Risk	0	1	RATE
MONEY_MARKS			
2	287	284	0.584362
4	0	51	0.104938
6	0	151	0.310700

## 2 数据的可视化分析与结论

### 2.1 数据可视化

(1) 分别对 Money\_Value 和 TOTAL 这两项 feature 进行分析

发现非欺诈类的数值分布都处于较低水平。

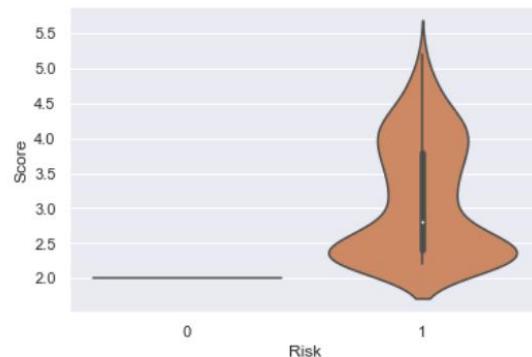


(2) 对 Score 进行分析

发现欺诈类的 Score 普遍分布于较高值域且分布广泛。

```
#对Score绘制提琴图  
#发现欺诈类的Score普遍分布于较高值域且分布广泛  
sns.violinplot(x='Risk', y='Score', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2482f0d53d0>
```

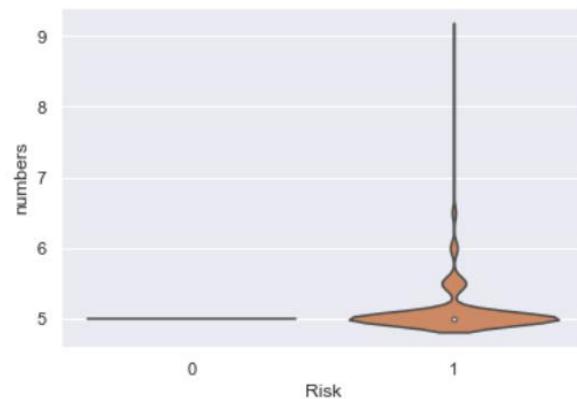


### (3) 对 numbers 进行分析

发现欺诈类的 numbers 分布值域广泛，主要集中于 5 左右。

```
#对numbers绘制提琴图  
#发现欺诈类的numbers分布值域广泛，主要集中于5左右  
sns.violinplot(x='Risk', y='numbers', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2482f3cf000>
```



### (4) 对 SCORE\_A 和 SCORE\_B 进行分析

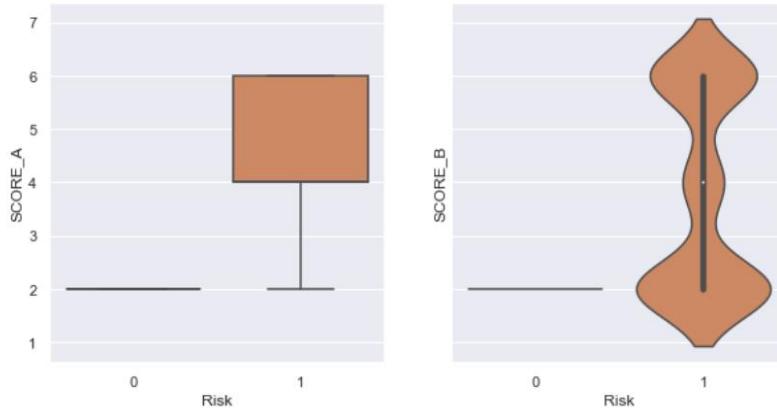
发现欺诈类的 SCORE\_A 值普遍分布于 4-6 之间，而 SCORE\_B 的值域较为广泛，在 2 和 6 左右较为集中。

```

# 分别对 SCORE_A 和 SCORE_B 这两项 feature 进行分析
# 发现欺诈类的 SCORE_A 值普遍分布于 4~6 之间，而 SCORE_B 的值域较为广泛，在 2 和 6 左右较为集中
_, axes = plt.subplots(1, 2, sharey=True, figsize=(10, 5))
sns.boxplot(x='Risk', y='SCORE_A', data=df, ax=axes[0])
sns.violinplot(x='Risk', y='SCORE_B', data=df, ax=axes[1])

```

<matplotlib.axes.\_subplots.AxesSubplot at 0x2483083f5b0>



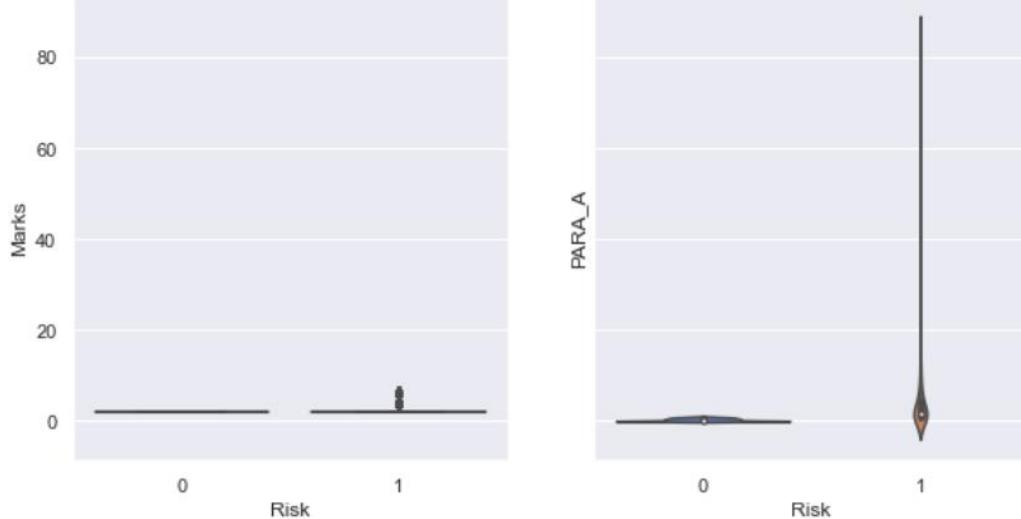
#### (4) 对 PARA\_A 和 Marks 进行分析

```

_, axes = plt.subplots(1, 2, sharey=True, figsize=(10, 5))
sns.violinplot(x='Risk', y='PARA_A', data=df, ax=axes[1])
sns.boxplot(x='Risk', y='Marks', data=df, ax=axes[0])

```

<matplotlib.axes.\_subplots.AxesSubplot at 0x211fe7e6670>



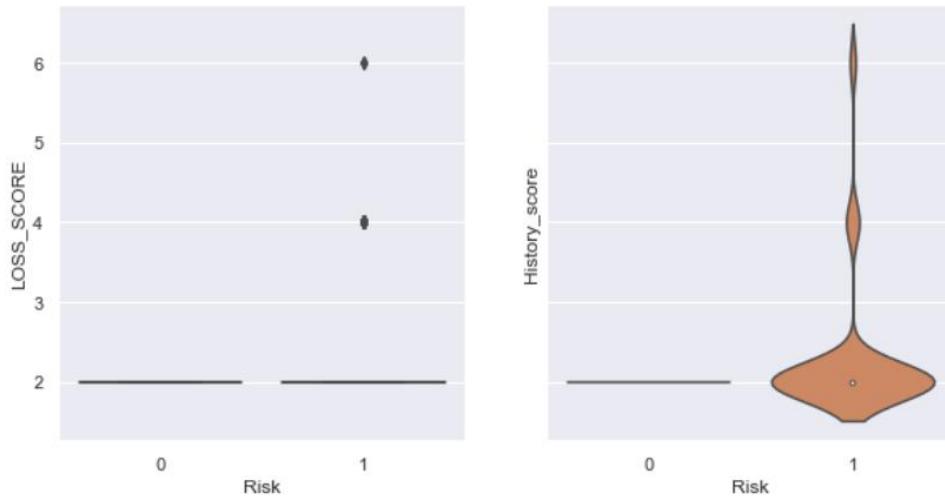
#### (5) 对 History\_score 和 LOSS\_SCORE 进行分析

发现对于非欺诈类来说，其 LOSS\_SCORE 和 History\_score 的值均为 0。

```

_, axes = plt.subplots(1, 2, sharey=True, figsize=(10, 5))
sns.violinplot(x='Risk', y='History_score', data=df, ax=axes[1])
sns.boxplot(x='Risk', y='LOSS_SCORE', data=df, ax=axes[0])
<matplotlib.axes._subplots.AxesSubplot at 0x211fe7437e0>

```



## 2.2 结论

综合以上分析，我们得出如下主要结论：

- (1) 超过 40% 的欺诈类都来自 LOCATION\_ID 为 8、19、2、16、12 的地区。
- (2) Loss 的值对欺诈类影响显著，超过 95% 的欺诈类的 Loss 值为 0。
- (3) District 的值对欺诈类影响显著，接近 75% 的欺诈类的 District 值为 2。
- (4) Money\_Marks 的值对欺诈类影响显著，超过 58 的欺诈类的 Money\_Marks 的值为 2。
- (5) Money\_Value 和 TOTAL 分布在较高值也更容易成为欺诈类。
- (6) Score 越高，尤其是分布于 2.5 和 4.0 左右的，越容易成为欺诈类。
- (7) numbers 处于较高值也容易成为欺诈类。
- (8) SCORE\_A 的值位于 4–6 的更可能为欺诈类。
- (9) SCORE\_B 的值位于 2 和 6 左右的更可能成为欺诈类。
- (10) PARA\_A 处于较高水平的更可能成为欺诈类，分布在较低（0 左右）水平时，成为欺诈类的可能性较小。
- (11) Marks 处于 2 以上的值时成为欺诈类的可能性非常大。
- (12) LOSS\_SCORE 为 4, 6 时更容易成为欺诈类。
- (13) History\_score 为 0 左右时，成为欺诈类可能性较大。若处于 3 以上的值时，基本上可以肯定为欺诈类。

## 机器学习与数据挖掘-上机作业二

上机任务：

- 1、复现 topic04
- 2、分别用 knn, logistics regression, svc 对上节课的企业欺诈数据进行分类。在调参后，输出测试数据（原数据 30%随机切割）的准确率。

### [目录]

<b>1 KNN 算法</b> .....	1
1.1 导入数据.....	1
1.2 模型训练.....	2
1.3 寻找最优超级参数 (k) .....	3
<b>2 logistics Regression 算法</b> .....	4
2.1 导入数据、数据预处理.....	4
2.2 模型训练.....	5
2.3 交叉验证寻找最优 C 值.....	7
<b>3 SVC 算法</b> .....	8
3.1 导入数据.....	8
3.2 数据处理.....	8
3.3 模型训练.....	9
3.4 参数调优.....	9

## 1 KNN 算法

- KNN 算法资料参考：[https://blog.csdn.net/weixin\\_30415801/article/details/98933030](https://blog.csdn.net/weixin_30415801/article/details/98933030)

### 1.1 导入数据

- 代码：

```
import pandas as pd
#读取数据
f = open("D:/XJY/大三上/专业课程/机器学习与数据挖掘-翁克瑞/上机/trial.csv")
df = pd.read_csv(f)
#将Risk设置为target值
y = df['Risk']
df.head()
```

- 运行结果：

	Sector_score	LOCATION_ID	PARA_A	SCORE_A	PARA_B	SCORE_B	TOTAL	numbers	Marks
0	3.41	40	10.37	6	105.56	6	115.93	9.0	6
1	2.37	1	1.13	4	3.85	6	4.98	6.5	6
2	2.72	8	5.61	6	99.33	6	104.94	6.5	6
3	2.72	7	3.04	6	67.37	6	70.41	6.5	6
4	2.72	5	1.90	4	35.04	6	36.94	6.5	6

## 1.2 训练模型

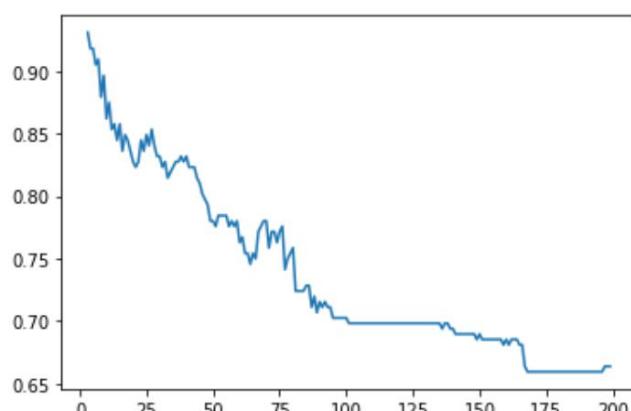
首先对数据集进行分割，按照训练数据 70%，测试数据 30% 的比例。训练模型，绘制 k 值与 accuracy 的图像，观察二者关系。

- 代码

```
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
#分割数据集
X_train, X_holdout, y_train, y_holdout = train_test_split(df.values, y, test_size=0.3,
                                                          random_state=17)
#绘制k值与accuracy的图像，寻找最优k值
k_list=[]
acc_list=[]
for k in range(3, 200):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    knn_pred = knn.predict(X_holdout)
    k_list.append(k)
    acc_list.append(accuracy_score(y_holdout, knn_pred))
plt.plot(k_list, acc_list)
plt.show()
```

可以看到，accuracy 随着 k 值的增大在波动减小。

- 运行结果：



### 1.3 寻找最优超级参数 (k)

使用 GridSearchCV 网格搜索算法寻找最优参数。

- 属性值说明：

属性值	说明
param_grid	是一个列表，列表里是算法对象的超参数的取值，用字典存储
n_jobs	使用电脑的 CPU 个数，-1 代表全部使用
verbose	每次 CV 时输出的格式
weights	有两种参数：'uniform'和'distance'，前者表示最原始的不带距离权重的 KNN，后者指带有距离权重的 KNN
n_neighbors	邻居个数 (k 值)
p	p=1 表示曼哈顿距离，p=2 表示欧式距离，p 可以大于 2，注意 p 参数只有在 weights='distance'时才有
cv	交叉验证时的折叠度

- 代码：

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV, cross_val_score

param_grid = [
    {
        'weights': ['uniform'],
        'n_neighbors': [i for i in range(1, 20)]
    },
    {
        'weights': ['distance'],
        'n_neighbors': [i for i in range(1, 20)],
        'p': [i for i in range(1, 6)]
    }
]
grid_search = GridSearchCV(knn, param_grid, verbose=2, cv=10)
grid_search.fit(X_train, y_train)
```

GridSearch 会自动找到一组最优的参数组合。

- 运行结果（部分）：

```
[CV] END ..... n_neighbors=19, p=5, weights=distance; total time= 0.0s
[CV] END ..... n_neighbors=19, p=5, weights=distance; total time= 0.0s
[CV] END ..... n_neighbors=19, p=5, weights=distance; total time= 0.0s

GridSearchCV(cv=10, estimator=KNeighborsClassifier(n_neighbors=19),
            param_grid=[{'n_neighbors': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                      13, 14, 15, 16, 17, 18, 19],
                         'weights': ['uniform']},
                        {'n_neighbors': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                      13, 14, 15, 16, 17, 18, 19],
                         'p': [1, 2, 3, 4, 5], 'weights': ['distance']}],
            verbose=2)
```

此时最好的一组参数组合{'n\_neighbors': 2, 'p': 1, 'weights': 'distance'}准确率达到了 0.9796。

```
#取出最好的超参数组合对应的准确率  
grid_search.best_score_
```

```
0.9796296296296296
```

```
#取出最好的一组超参数  
grid_search.best_params_  
{'n_neighbors': 2, 'p': 1, 'weights': 'distance'}
```

```
#取出参数最好的一组对应的分类器  
grid_search.best_estimator_  
KNeighborsClassifier(n_neighbors=2, p=1, weights='distance')
```

## 2 logistics Regression 算法

### 2.1 导入数据、数据预处理

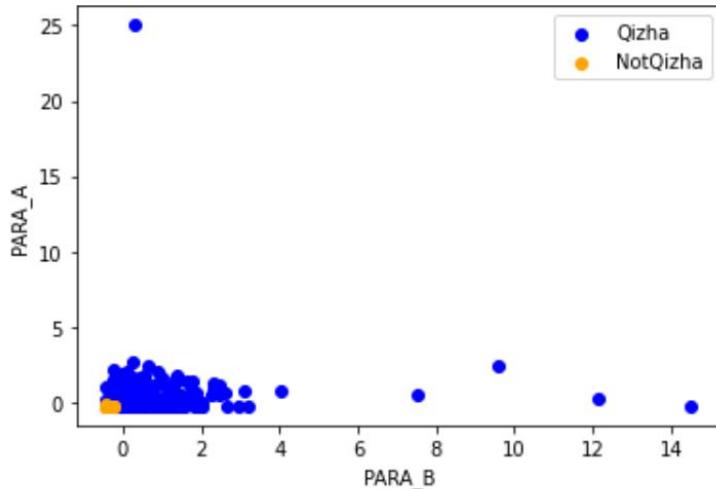
为了便于画出图像，我们在此选择了两个特征值： PARA\_A 和 PARA\_B。

• 代码：

```
import pandas as pd  
from matplotlib import pyplot as plt  
from sklearn.preprocessing import StandardScaler  
#读取数据  
f = open("D:/XJY/大三上/专业课程/机器学习与数据挖掘-翁克瑞/上机/trial.csv")  
df = pd.read_csv(f)  
  
#去除多余的特征值  
df.drop(['LOCATION_ID', 'SCORE_B', 'numbers', 'Sector_score', 'TOTAL', 'Money_Value', 'Marks',  
         'Score', 'SCORE_A', 'History'], axis=1, inplace=True)  
  
#设置特征值和target value  
X = df.iloc[:, :2].values  
y = df.iloc[:, 2].values  
  
#对数据进行标准化  
sc = StandardScaler()  
X = sc.fit_transform(X)  
  
#绘制图像  
plt.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', label='Qizha')  
plt.scatter(X[y == 0, 0], X[y == 0, 1], c='orange', label='NotQizha')  
plt.xlabel("PARA_B")  
plt.ylabel("PARA_A")  
plt.title('2 tests of microchips. Logit with C=1')  
plt.legend()
```

对其进行标准化之后，画出图像。

• 运行结果：



## 2.2 训练模型

首先我们定义一个函数，用来显示分类器的分界线。

• 代码：

```
# 定义一个函数，显示分类器的分界线
def plot_boundary(clf, X, y, grid_step=.01, poly_featurizer=None):
    x_min, x_max = X[:, 0].min() - .1, X[:, 0].max() + .1
    y_min, y_max = X[:, 1].min() - .1, X[:, 1].max() + .1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, grid_step),
                          np.arange(y_min, y_max, grid_step))

    # 在 [x_min, x_max]x[y_min, y_max] 的每一点都用它自己的颜色来对应
    Z = clf.predict(poly_featurizer.transform(np.c_[xx.ravel(), yy.ravel()]))
    Z = Z.reshape(xx.shape)
    plt.contour(xx, yy, Z, cmap=plt.cm.Paired)
```

进行特征值构造，我们在此定义 degree 为 5。

• 特征值构造的相关资料：<https://blog.csdn.net/hushenming3/article/details/80500364>

```
# 特征的构造，指定degree为5
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=5)
X_poly = poly.fit_transform(X)
X_poly.shape
```

(773, 21)

为了对模型进行调优，我们在此首先尝试一些正则化优化模型。

(1) 首先，取 C=1。

```

C = 1
logit = LogisticRegression(C=C, random_state=17)
logit.fit(X_poly, y)

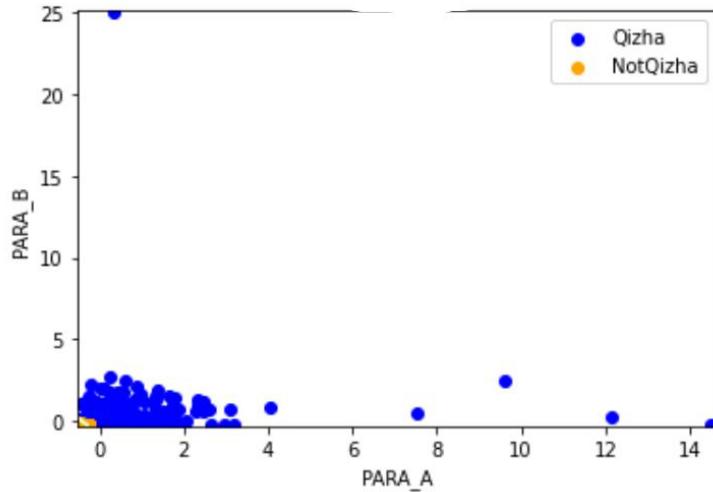
plot_boundary(logit, X, y, grid_step=.005, poly_featurizer=poly)

plt.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', label='Qizha')
plt.scatter(X[y == 0, 0], X[y == 0, 1], c='orange', label='NotQizha')
plt.xlabel("PARA_A")
plt.ylabel("PARA_B")
plt.title('2 tests of microchips. Logit with C=%s' % C)
plt.legend()

print("Accuracy on training set:",
      round(logit.score(X_poly, y), 3))

```

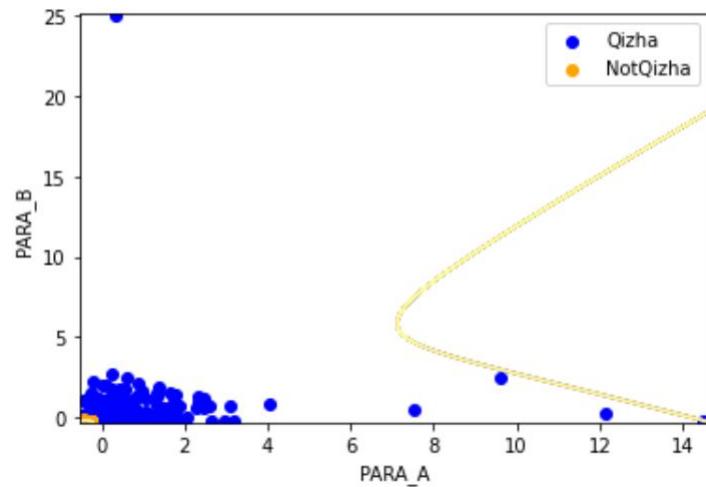
Accuracy on training set: 0.836



在此我们发现，当 C=1 时，模型的精确率为 0.836。让我尝试放大 C 值。

(2) 放大 C 值，取 1000。  
这时，精确度上升到了 0.878。

Accuracy on training set: 0.878



## 2.3 交叉验证寻找最优 C 值

我们依然使用网格搜索法，利用 LogisticRegressionCV() 方法进行网格搜索参数后再交叉验证，LogisticRegressionCV() 是专门为逻辑回归设计的。如果想对其他模型进行同样的操作，可以使用 GridSearchCV() 或 RandomizedSearchCV() 等超参数优化算法。

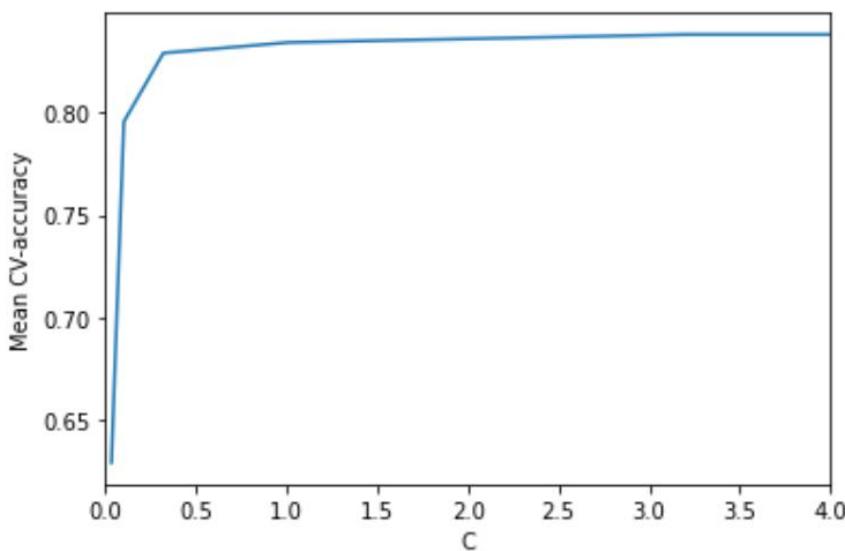
```
from sklearn.model_selection import StratifiedKFold
# 使用 LogisticRegressionCV() 方法进行网格搜索参数后再交叉验证,
# LogisticRegressionCV() 是专门为逻辑回归设计的。
# 如果想对其他模型进行同样的操作, 可以使用 GridSearchCV() 或 RandomizedSearchCV()
# 等超参数优化算法。
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=17)
# 下方结尾的切片是为了在线上环境搜索更快
c_values = np.logspace(-2, 3, 500)[50:450:50]

logit_searcher = LogisticRegressionCV(
    Cs=c_values, cv=skf, verbose=1)
logit_searcher.fit(X_poly, y)
```

绘制准确度与 C 值的图像，我们发现当 C 取到 1~3 就可以取得比较不错的结果。

```
plt.plot(c_values, np.mean(logit_searcher.scores_[1], axis=0))
plt.xlabel('C')
plt.ylabel('Mean CV-accuracy')
plt.xlim((0, 4))
```

(0.0, 4.0)



## 3 SVC

### 3.1 导入数据

为了减少模型复杂度，避免过拟合，我们在此减少一些特征值。

```
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.preprocessing import StandardScaler
#读取数据
f = open("D:/XJY/大三上/专业课程/机器学习与数据挖掘-翁克瑞/上机/trial.csv")
df = pd.read_csv(f)

#去除多余的特征值
df.drop(['LOCATION_ID', 'numbers', 'Marks', 'History_score', 'MONEY_Marks', 'District', 'Loss'],
        df.head()
```

	Sector_score	PARA_A	SCORE_A	PARA_B	SCORE_B	TOTAL	Money_Value	Score	Risk
0	3.41	10.37	6	105.56	6	115.93	52.13	4.8	1
1	2.37	1.13	4	3.85	6	4.98	20.89	5.0	1
2	2.72	5.61	6	99.33	6	104.94	76.47	4.8	1
3	2.72	3.04	6	67.37	6	70.41	77.47	4.8	1
4	2.72	1.90	4	35.04	6	36.94	16.19	4.6	1

### 3.2 数据处理

- **关于随机数种子：**其实就是该组随机数的编号，在需要重复试验的时候，保证得到一组一样的随机数。比如你每次都填 1，其他参数一样的情况下你得到的随机数组是一样的。但填 0 或不填，每次都会不一样。随机数的产生取决于种子，随机数和种子之间的关系遵从以下两个规则：种子不同，产生不同的随机数；种子相同，即使实例不同也产生相同的随机数。

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.svm import SVC
from sklearn.svm import LinearSVC

#将目标特征与其他特征分离
X = df.iloc[:, :-1] # 数据前17列
y = df.iloc[:, -1] # 最后一列

# 划分训练集train_X, train_y和测试集test_X, test_y (7: 3)
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size = 0.3, random_state =
scaler = StandardScaler()
scaler.fit(train_X)
scaled_train_X = scaler.transform(train_X)

scaled_train_X
```

下图是标准化之后的数据结果：

```
array([[-0.76236409,  0.21510947,  1.43869183, ..., -0.3309882 ,  
       -0.22071495, -0.35068185],  
      [ 1.44557665, -0.26932439, -0.87777329, ..., -0.45057999,  
       -0.22284134, -0.81970257],  
      [-0.67851824,  0.41159313,  1.43869183, ...,  1.67363778,  
       0.89670294,  1.99442175],  
      ...,  
      [ 1.44557665, -0.34724032, -0.87777329, ..., -0.46605165,  
       -0.22284134, -0.81970257],  
      [ 1.44557665, -0.3252206 , -0.87777329, ..., -0.44890737,  
       -0.22284134, -0.81970257],  
      [-0.67851824, -0.41160566, -0.87777329, ..., -0.48235962,  
       -0.2107918 , -0.81970257]])
```

### 3.3 模型训练

```
# 构建支持向量机模型  
clf = SVC()  
  
# 模型训练  
clf.fit(scaled_train_X, train_y)  
  
# 测试集标准化  
scaled_test_X = scaler.transform(test_X)  
  
# 使用模型返回预测值  
pred_y = clf.predict(scaled_test_X)  
  
print("训练数据准确率: ", clf.score(scaled_train_X, train_y))  
print("测试数据准确率: ", clf.score(scaled_test_X, test_y))  
## 打印支持向量的个数，返回结果为列表，[-1标签的支持向量，+1标签的支持向量]  
# print(clf.n_support_)
```

训练数据准确率： 0.9685767097966729

测试数据准确率： 0.9525862068965517

在没有进行任何参数调优的结果下，我们的模型在测试数据中已经可以达到了 0.953。

### 3.4 参数调优

```
# 构建惩罚系数为0.3的模型，并与之前的模型做比较  
print('----- 加入正则化项: C = 2 -----')  
clf_1 = SVC(C=2)  
clf_1.fit(scaled_train_X, train_y)  
pred_y1 = clf_1.predict(scaled_test_X)  
print(pred_y1)  
print("训练数据准确率: ", clf_1.score(scaled_train_X, train_y))  
print("测试数据准确率: ", clf_1.score(scaled_test_X, test_y))
```

```

print('----- 线性核函数: linear -----')
clf_2 = SVC(kernel="linear")
clf_2.fit(scaled_train_X, train_y)
pred_y2 = clf_2.predict(scaled_test_X)
print(pred_y2)
print("训练数据准确率: ", clf_2.score(scaled_train_X, train_y))
print("测试数据准确率: ", clf_2.score(scaled_test_X, test_y))

print('----- 正则化+线性核函数: linear -----')
clf_3 = SVC(C=2, kernel="linear")
clf_3.fit(scaled_train_X, train_y)
pred_y3 = clf_3.predict(scaled_test_X)
print(pred_y3)
print("训练数据准确率: ", clf_3.score(scaled_train_X, train_y))
print("测试数据准确率: ", clf_3.score(scaled_test_X, test_y))

```

从运行结果中可以看导，加入正则化项 C=2 之后，测试数据的准确率从 0.953 上升到了 0.957。加入线性核函数之后，则上升到了 0.983。

- 运行结果：

---

----- 加入正则化项: C = 2 -----

```
[1 0 1 1 0 1 1 0 1 1 1 0 0 0 1 0 0 0 1 1 0 0 1 1 0 1 1 0 0 1 0 0 1 1 1 1
 1 0 1 1 1 1 0 1 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 0 1 0 1
 0 1 1 0 1 0 1 0 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 1 1 0 1 1 0 0 1 0 1
 1 0 1 1 1 1 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 0 0 1 1 0
 1 1 1 0 1 0 0 0 1 1 1 1 0 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 1 0 0 1
 0 0 1 1 1 1 1 1 1 0 1 1 0 0 1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1
 0 0 1 0 0 1 1 1 1]
```

训练数据准确率： 0.9759704251386322  
 测试数据准确率： 0.9568965517241379

---

----- 线性核函数: linear -----

```
[1 0 1 1 0 1 1 0 1 1 1 0 0 0 1 0 0 0 1 1 0 0 1 1 0 1 1 0 0 1 0 0 1 1 1 1
 1 0 1 1 1 1 0 1 0 0 1 1 1 0 0 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 0 1 0 1
 0 1 1 0 1 0 1 0 0 1 0 1 1 1 0 0 1 1 0 0 0 0 1 1 1 0 1 1 0 0 1 0 0 1 0 1
 1 0 1 1 1 1 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1 1 0
 1 1 1 0 1 0 0 0 1 1 1 1 0 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 1 1 0 1 0 0 1
 0 0 1 1 1 1 1 1 1 0 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 0 1 0 0 1 1 1 1]
```

训练数据准确率： 0.9963031423290203  
 测试数据准确率： 0.9827586206896551

---

----- 正则化+线性核函数: linear -----

```
[1 0 1 1 0 1 1 0 1 1 1 0 0 0 1 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 0 0 1 1 1 1
 1 0 1 1 1 1 0 1 0 0 1 1 1 0 0 1 1 1 1 1 1 0 0 0 1 0 1 1 0 1 1 0 1 0 1
 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 1 0 1 1 0 0 1 0 0 1 0 1
 1 0 1 1 1 1 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1 1 0
 1 1 1 1 1 0 0 0 1 1 1 1 0 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 1 1 0 1 0 0 1
 0 0 1 1 1 1 1 1 1 0 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 0 1 0 0 1 1 1 1]
```

训练数据准确率： 1.0  
 测试数据准确率： 1.0

---

上面只是我们对于参数组合的初步尝试，为了更有说服力，我们在此使用 `GridSearchCV` 寻找最优参数组合。

- 参考资料： <https://blog.csdn.net/k411797905/article/details/95856514>

- SVM 模型的参数：SVM 模型有两个非常重要的参数 `C` 与 `gamma`。`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。

最后，自动选择出最优参数组合，以及测试数据的准确率。

```
from sklearn import svm
from sklearn.model_selection import GridSearchCV

parameters={'kernel':('rbf','linear'), 'C':[i for i in range(1,10)]}

clf_4=GridSearchCV(clf_new,parameters)
clf_4.fit(scaled_train_X,train_y)
print(clf_4.best_estimator_)
# #输出:
# SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
#      decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
#      max_iter=-1, probability=False, random_state=None, shrinking=True,
#      tol=0.001, verbose=False)

SVC(C=2, kernel='linear')
```

```
clf_4.best_score_
```

1.0

## 机器学习与数据挖掘-上机作业二

上机任务：

- (1) 复现 topic03,topic04 的代码
- (2) 对 cancer 数据包，分割 30% 测试，70% 训练和调参，分别给出测试数据在 logistics regression, svc, decisiontree,knn 调参后的测试结果。
- (2) 对 boston 房价数据，分割 30% 测试，70% 训练和调参，分别给出测试数据在线性回归，svr, decisiontree 调参后的测试结果。

### 目录

<b>1 Breast cancer 数据包</b> .....	1
1.1 导入数据.....	1
1.2 数据可视化.....	3
1.3 分割数据集.....	5
1.4 KNN 算法.....	5
1.5 决策树算法.....	7
1.6 Logistics regression 算法.....	8
1.7 SVC 算法.....	9
1.8 模型对比.....	11
<b>2 Boston 房价数据</b> .....	12
2.1 导入数据.....	12
2.2 数据预处理.....	13
2.3 线性回归模型.....	14
2.4 SVR 模型.....	15
2.5 决策树模型.....	16

### 1 Breast cancer 数据包

- 参考资料：<https://blog.csdn.net/wangyuankl123/article/details/102777014>

#### 1.1 导入数据

导入数据，并输出相关数据信息。

- 代码：

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.datasets import load_breast_cancer

#加载数据
cancer = load_breast_cancer()

#输出数据相关信息
print("数据: ", cancer)
print("target name: ", cancer['target_names'])
print("target: ", cancer['target'])
print("备注: ", cancer['DESCR'])
print("feature names: ", cancer['feature_names'])
print("shape: ", cancer['data'].shape)
```

#### • 运行结果 (部分) :

## 将数据转化为 DataFrame

```
#将数据转化为Dataframe  
df = pd.DataFrame(np.c_[cancer['data'], cancer['target']], columns= np.append(cancer['fea  
#查看数据  
df.head()
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	di
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	

5 rows × 31 columns

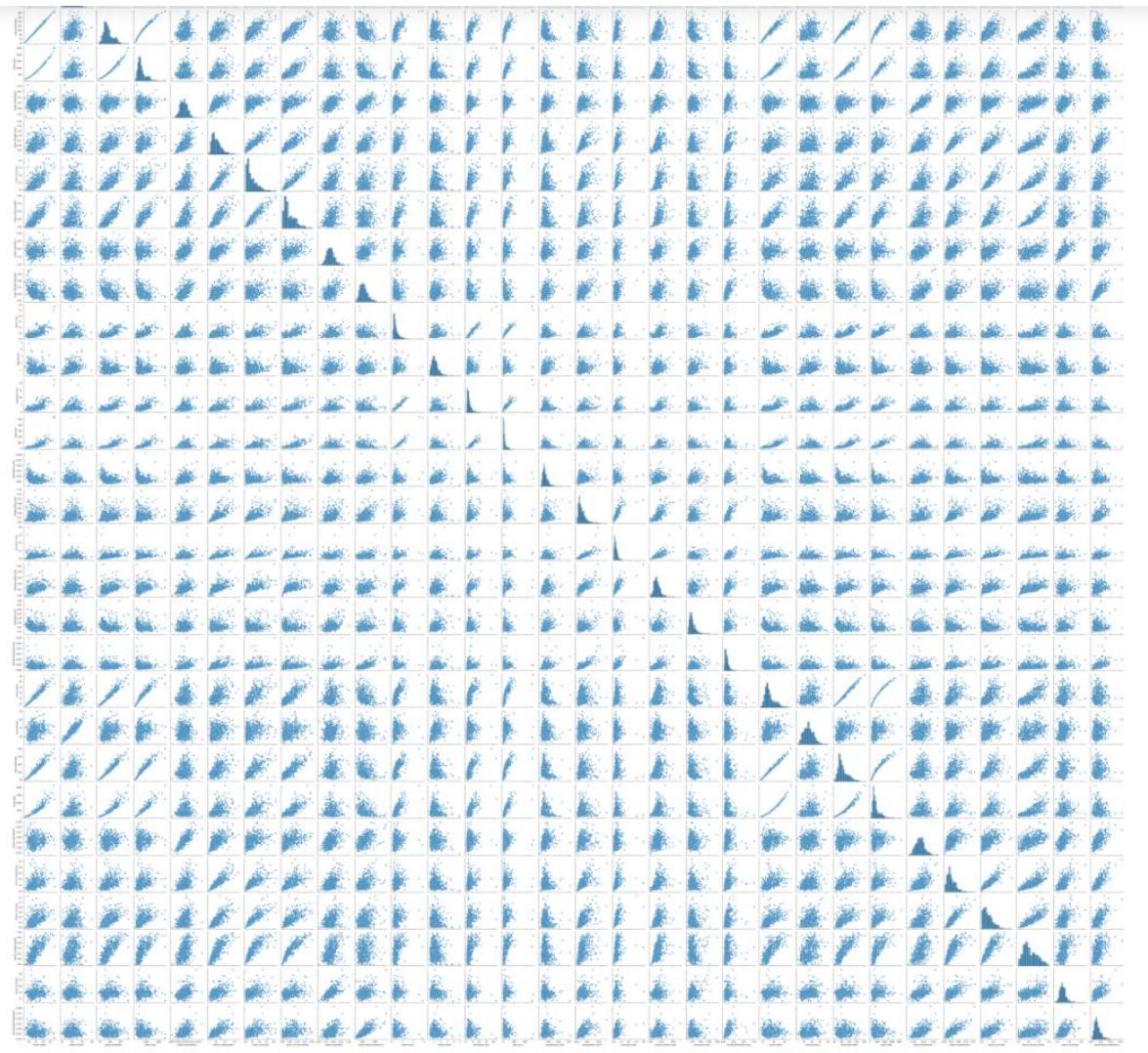
## 1.2 数据可视化

查看样本总体特征。

• 代码：

```
#查看样本总体特征
sns.pairplot(df, vars =['mean radius', 'mean texture', 'mean perimeter', 'mean area',
 'mean smoothness', 'mean compactness', 'mean concavity',
 'mean concave points', 'mean symmetry', 'mean fractal dimension',
 'radius error', 'texture error', 'perimeter error', 'area error',
 'smoothness error', 'compactness error', 'concavity error',
 'concave points error', 'symmetry error', 'fractal dimension error',
 'worst radius', 'worst texture', 'worst perimeter', 'worst area',
 'worst smoothness', 'worst compactness', 'worst concavity',
 'worst concave points', 'worst symmetry', 'worst fractal dimension'])
```

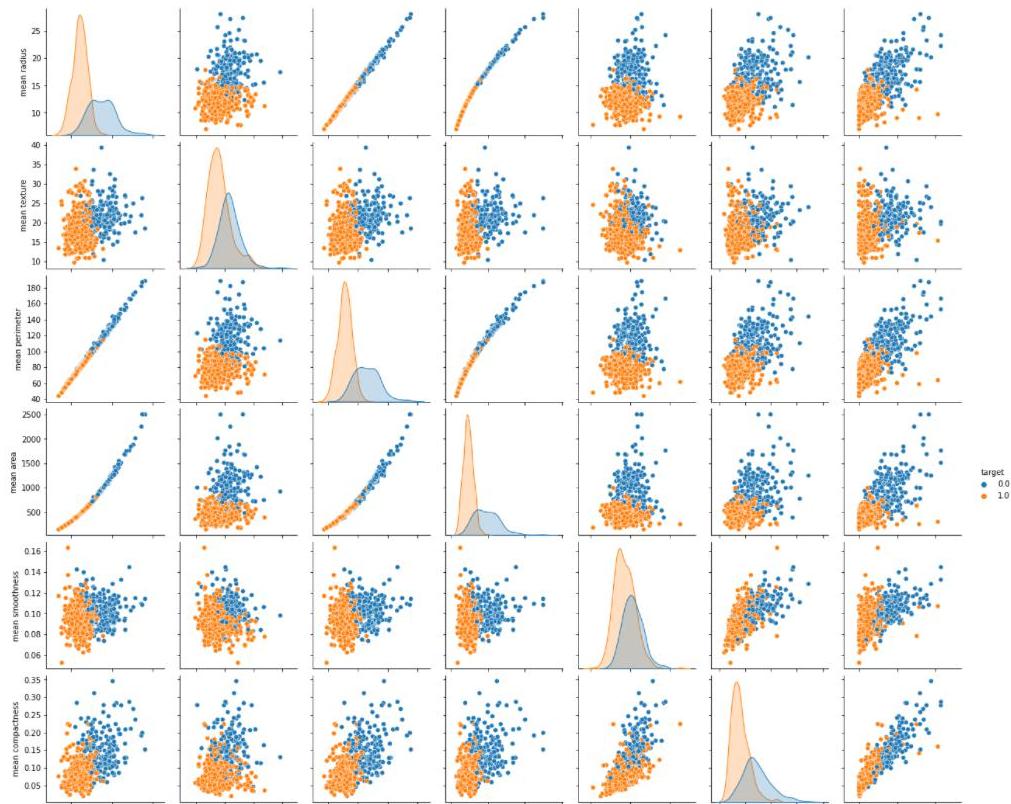
• 运行结果：



挑选出部分特征：

```
#选出部分特征  
sns.pairplot(df, hue = 'target', vars =['mean radius', 'mean texture', 'mean perimeter', 'me  
'mean smoothness', 'mean compactness', 'mean concavity'])
```

<seaborn.axisgrid.PairGrid at 0x244f18a96a0>

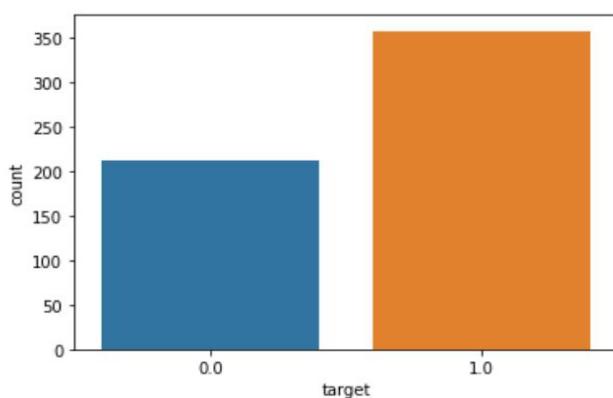


查看 target 值

```
: #查看tagert值数量  
sns.countplot(df['target'])
```

D:\ProgramData\Anaconda3\lib\site-packages\seaborn\\_decorators.py:36: FutureWarning: Pa  
ss the following variable as a keyword arg: x. From version 0.12, the only valid posit  
ional argument will be `data`, and passing other arguments without an explicit keyword w  
ill result in an error or misinterpretation.  
warnings.warn(

```
: <AxesSubplot:xlabel='target', ylabel='count'>
```



### 1.3 分割数据集

分割数据集：以训练数据与测试数据 7: 3 的比例，随机种子选择 17

```
from sklearn.model_selection import train_test_split

#设置x, y变量
x = df.drop(['target'], axis=1)
y = df['target']

#分割数据集：以训练数据与测试数据7: 3的比例，随机种子选择17
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=17)
```

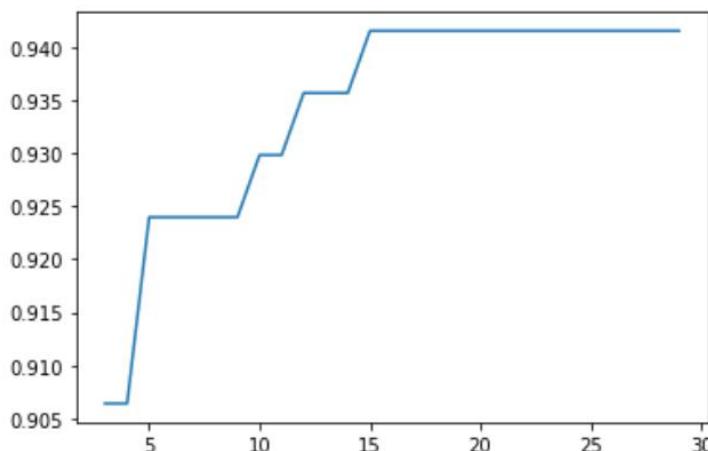
### 1.4 KNN 算法

训练模型，绘制出 K 值与 accuracy 的图像

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

#模型训练

#绘制k值与accuracy的图像，简单观察一下k值与accuracy的趋势
k_list = []
acc_list = []
for k in range(3, 30):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    knn_pred = knn.predict(X_test)
    k_list.append(k)
    acc_list.append(accuracy_score(y_test, knn_pred))
plt.plot(k_list, acc_list)
plt.show()
```



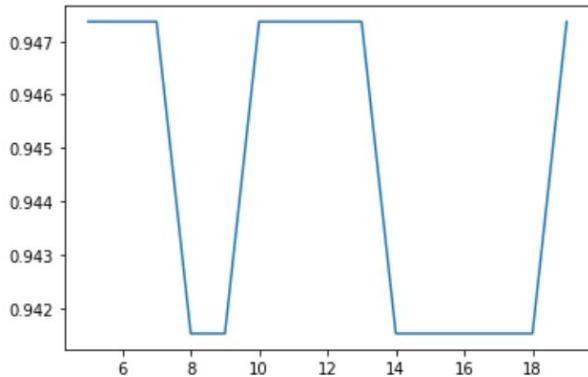
利用网格搜索法，寻找最优超级参数，可以看到，当 k=10 的时候，准确率达到 0.95 左右



## 1.5 决策树算法

训练模型，绘制树的 depth 值与 accuracy 图像

```
from sklearn.tree import DecisionTreeClassifier
# max_depth参数限制决策树的深度from sklearn.tree import export_graphviz
depth_list=[]
tree_acc_list=[]
for i in range(5, 20):
    clf_tree = DecisionTreeClassifier(criterion='entropy', max_depth=3,
                                       random_state=i)
    # 训练决策树
    clf_tree.fit(X_train, y_train)
    # 测试集上的准确度
    tree_pred = clf_tree.predict(X_test)
    tree_score = accuracy_score(y_test, tree_pred)
    depth_list.append(i)
    tree_acc_list.append(tree_score)
plt.plot(depth_list, tree_acc_list)
plt.show()
```



• 决策树调参优化参考资料：<https://www.cnblogs.com/jin-liang/p/9638197.html>

使用网格搜索法寻找最优参数

决策树调参优化参考资料：<https://www.cnblogs.com/jin-liang/p/9638197.html>

```
# 用GridSearchCV寻找最优参数（字典）
# 定义参数列表
tree_param = {'criterion':['gini'],
              'max_depth':[i for i in range(1, 20)]}
#           'min_samples_leaf':[8],
#           'min_samples_split':[3]}
# max_depth: 树的深度
# min_samples_split: 拆分内部节点所需的最小样本数
# min_samples_leaf: 是叶节点所需的最小样本数
# max_features: 表示查找最佳拆分时要考虑的要最大特征数量。

# 交叉验证
tree_grid = GridSearchCV(clf_tree, param_grid=tree_param, cv=2)
tree_grid.fit(X_train, y_train)

# 得到最优的参数和分值
print('最优分类器:\n', tree_grid.best_params_, '\n最优分数:', tree_grid.best_score_)
```

最优分类器：

{'criterion': 'gini', 'max\_depth': 2}

最优分数： 0.9422110552763818

决策树的精确度约为 0.942

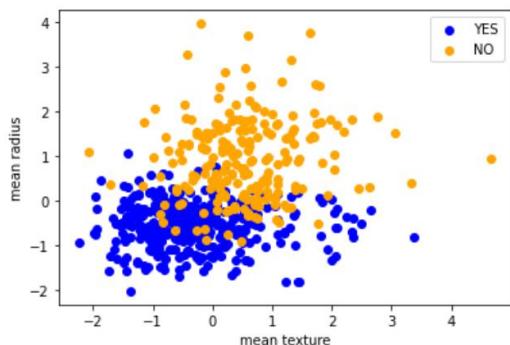
## 1.6 Logistics regression 算法

在此使用两个特征进行计算， mean texture 和 mean radius

```
from sklearn.preprocessing import StandardScaler  
  
#-----数据预处理-----  
  
#设置特征值和target value  
lr_X = df[['mean texture', 'mean radius']]  
  
#对数据进行标准化  
sc = StandardScaler()  
lr_X = sc.fit_transform(lr_X)  
  
#绘制图像  
plt.scatter(lr_X[y == 1, 0], lr_X[y == 1, 1], c='blue', label='YES')  
plt.scatter(lr_X[y == 0, 0], lr_X[y == 0, 1], c='orange', label='NO')  
plt.xlabel("mean texture")  
plt.ylabel("mean radius")  
plt.legend()
```

绘制散点图

<matplotlib.legend.Legend at 0x24484b08dc0>



```
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV  
#定义正则化项的参数C  
C = 0.5  
  
#拟合logistic回归模型  
logit = LogisticRegression(C=C, class_weight=None, dual=False,  
                           fit_intercept=True, random_state=17,  
                           intercept_scaling=1, max_iter=100,  
                           multi_class='ovr', n_jobs=1, penalty='l2',  
                           solver='liblinear', tol=0.0001, verbose=0, warm_start=False)  
logit.fit(X_train, y_train)  
  
print("Accuracy on training set:",  
      round(logit.score(X_train, y_train), 3))
```

Accuracy on training set: 0.965

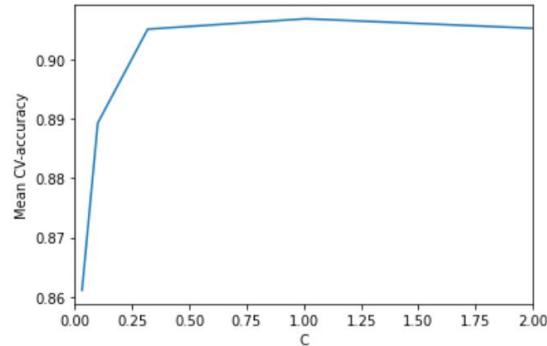
在此可以看到当 C=0.5 时，logistics 回归得到的准确率为 0.965

寻找最优参数。

```
from sklearn.model_selection import StratifiedKFold  
  
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=17)  
# 下方结尾的切片是为了在线上环境搜索更快  
c_values = np.logspace(-2, 3, 500)[50:450:50]  
  
logit_searcher = LogisticRegressionCV(  
    Cs=c_values, cv=skf, verbose=1)  
logit_searcher.fit(X_poly, y)
```

从下图可以看到，当 C 取到 1.0 左右时，就可以得到比较不错的准确率。

```
plt.plot(c_values, np.mean(logit_searcher.scores_[1], axis=0))  
plt.xlabel('C')  
plt.ylabel('Mean CV-accuracy')  
plt.xlim((0, 2.0))  
  
(0.0, 2.0)
```

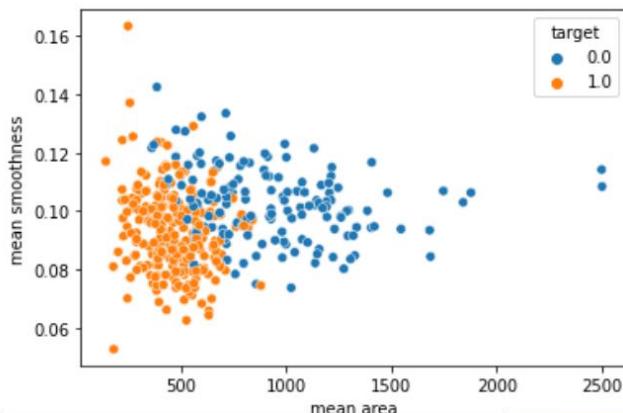


## 1.7 SVC 算法

训练模型，对数据标准化，绘制原始数据散点图。在此我们选择 mean area 和 mean smoothness 两个特征。

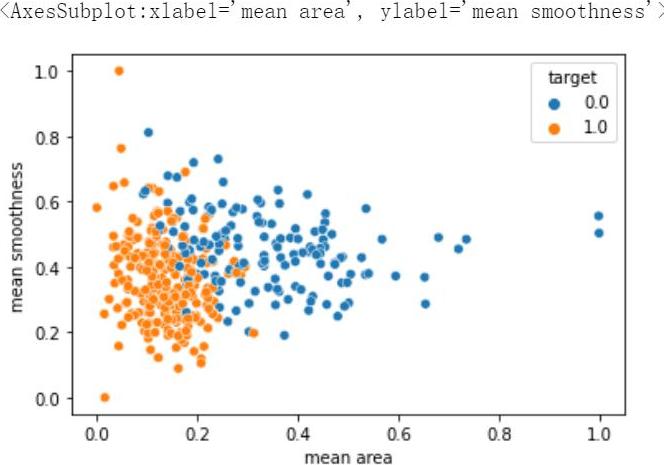
```
#模型训练  
from sklearn.svm import SVC  
from sklearn.metrics import classification_report, confusion_matrix  
svc_model= SVC()  
svc_model.fit(X_train,y_train)  
y_predict = svc_model.predict(X_test)  
  
#标准化  
min_train = X_train.min()  
range_train =(X_train - min_train).max()  
x_train_scaled =(X_train-min_train)/range_train  
#画出原始数据散点图(在此我们选择“mean area”和“mean smoothness”两个特征)  
sns.scatterplot(x = X_train['mean area'], y= X_train['mean smoothness'], hue =y_train)
```

```
<AxesSubplot:xlabel=' mean area', ylabel=' mean smoothness'>
```



数据标准化之后的散点图

```
#画出标准化后数据散点图(在此我们选择“mean area”和“mean smoothness”两个特征)
sns.scatterplot(x = x_train_scaled['mean area'], y= x_train_scaled['mean smoothness'], hue
```



```
#标准化测试数据
min_test = X_test.min()
range_test = (X_test - min_test).max()
x_test_scaled = (X_test - min_test) / range_test
svc_model.fit(x_train_scaled, y_train)
y_predict = svc_model.predict(x_test_scaled)
```

```
print(classification_report(y_test, y_predict))
```

	precision	recall	f1-score	support
0.0	0.88	0.98	0.93	61
1.0	0.99	0.93	0.96	110
accuracy			0.95	171
macro avg	0.94	0.96	0.94	171
weighted avg	0.95	0.95	0.95	171

使用网格搜索算法，寻找最优参数。

```
: from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.1, 1, 10, 100], 'gamma': [1, 0.1, 0.01, 0.001], 'kernel': ['rbf']}
grid=GridSearchCV(SVC(), param_grid, refit=True, verbose=4)
grid.fit(x_train_scaled, y_train)
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[CV 1/5] END ..... C=0.1, gamma=1, kernel=rbf;, score=0.938 total time= 0.0s
[CV 2/5] END ..... C=0.1, gamma=1, kernel=rbf;, score=0.950 total time= 0.0s
[CV 3/5] END ..... C=0.1, gamma=1, kernel=rbf;, score=0.950 total time= 0.0s
[CV 4/5] END ..... C=0.1, gamma=1, kernel=rbf;, score=0.949 total time= 0.0s
[CV 5/5] END ..... C=0.1, gamma=1, kernel=rbf;, score=0.962 total time= 0.0s
[CV 1/5] END ..... C=0.1, gamma=0.1, kernel=rbf;, score=0.863 total time= 0.0s
[CV 2/5] END ..... C=0.1, gamma=0.1, kernel=rbf;, score=0.938 total time= 0.0s
[CV 3/5] END ..... C=0.1, gamma=0.1, kernel=rbf;, score=0.912 total time= 0.0s
[CV 4/5] END ..... C=0.1, gamma=0.1, kernel=rbf;, score=0.899 total time= 0.0s
[CV 5/5] END ..... C=0.1, gamma=0.1, kernel=rbf;, score=0.899 total time= 0.0s
[CV 1/5] END ..... C=0.1, gamma=0.01, kernel=rbf;, score=0.613 total time= 0.0s
[CV 2/5] END ..... C=0.1, gamma=0.01, kernel=rbf;, score=0.625 total time= 0.0s
[CV 3/5] END ..... C=0.1, gamma=0.01, kernel=rbf;, score=0.625 total time= 0.0s
[CV 4/5] END ..... C=0.1, gamma=0.01, kernel=rbf;, score=0.620 total time= 0.0s
[CV 5/5] END ..... C=0.1, gamma=0.01, kernel=rbf;, score=0.620 total time= 0.0s
[CV 1/5] END .... C=0.1, gamma=0.001, kernel=rbf;, score=0.613 total time= 0.0s
[CV 2/5] END .... C=0.1, gamma=0.001, kernel=rbf;, score=0.625 total time= 0.0s
[CV 3/5] END .... C=0.1, gamma=0.001, kernel=rbf;, score=0.625 total time= 0.0s
[CV 4/5] END .... C=0.1, gamma=0.001, kernel=rbf;, score=0.620 total time= 0.0s
[CV 5/5] END .... C=0.1, gamma=0.001, kernel=rbf;, score=0.620 total time= 0.0s
[CV 1/5] END ..... C=1, gamma=1, kernel=rbf;, score=0.950 total time= 0.0s
[CV 2/5] END ..... C=1, gamma=1, kernel=rbf;, score=0.975 total time= 0.0s
[CV 3/5] END ..... C=1, gamma=1, kernel=rbf;, score=0.975 total time= 0.0s
[CV 4/5] END ..... C=1, gamma=1, kernel=rbf;, score=0.975 total time= 0.0s
[CV 5/5] END ..... C=1, gamma=1, kernel=rbf;, score=0.975 total time= 0.0s
```

```
print("最优参数组合: ", grid.best_params_)
print("最优分数: ", grid.best_score_)
grid_predictions=grid.predict(x_test_scaled)
```

```
最优参数组合: {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
最优分数: 0.9748417721518987
```

可以看到，SVC 算法的准确率达到了 0.975

## 1.8 模型对比

综合以上四个算法，我们对比了模型精确度。

模型	精确度
KNN 算法	0.95
决策树算法	0.942
logistics 回归算法	0.965
SVC 算法	0.975

## 2 Boston 房价数据

### 2.1 导入数据

导入数据，并查看数据基本信息

```
from sklearn.datasets import load_boston
import pandas as pd
import numpy as np
#载入数据集
boston = load_boston()

#查看数据基本信息
print(boston.DESCR)
```

---

```
.. _boston_dataset:

Boston house prices dataset
-----
**Data Set Characteristics:**

:Number of Instances: 506

:Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 1
4) is usually the target.

:Attribute Information (in order):
 - CRIM    per capita crime rate by town
 - ZN      proportion of residential land zoned for lots over 25,000 sq.ft.
 - INDUS   proportion of non-retail business acres per town
 - CHAS    Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
 - NOX    nitric oxides concentration (parts per 10 million)
 - RM     average number of rooms per dwelling
 - AGE    proportion of owner-occupied units built prior to 1940
```

查看关键字、将数据集转化为 dataframe

```
#查看关键字
boston.keys()

dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename', 'data_module'])

#将数据集转化为dataframe
bostonDf_X = pd.DataFrame(boston.data, columns=boston.feature_names)
bostonDf_y = pd.DataFrame(boston.target, columns=['houseprice']) #注意加列名称

#合并dataframe
bostonDf = pd.concat([bostonDf_X, bostonDf_y], axis=1) #axis=1为横向操作
bostonDf.shape #(506, 14)
bostonDf.head() #看一下数据集
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

#查看基本信息

```
print("最大值: ", np.max(boston.target))
print("最小值: ", np.min(boston.target))
print("平均值: ", np.mean(boston.target))
bostonDf.describe()
```

最大值: 50.0

最小值: 5.0

平均值: 22.532806324110677

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000

## 2.2 数据预处理

对数据集进行分割，依然按照 7: 3 划分数据集

```
#进行数据预处理
from sklearn.preprocessing import StandardScaler
#切分数据集
from sklearn.model_selection import train_test_split
#划分训练集train_X, train_y和测试集train_X, train_y (7: 3)
X_train, X_test, y_train, y_test = train_test_split(bostonDf_X, bostonDf_y, test_size = 0.3)
```

```
print(X_test.shape)
print(y_test.shape)
print(X_train.shape)
print(y_train.shape)
```

(152, 13)

(152, 1)

(354, 13)

(354, 1)

对训练数据和测试数据进行标准化

```
#标准化
from sklearn.preprocessing import StandardScaler

#特征值
std_x = StandardScaler()
X_train = std_x.fit_transform(X_train)
X_test = std_x.transform(X_test)

#目标值
std_y = StandardScaler()
y_train = std_y.fit_transform(y_train)
y_test = std_y.transform(y_test)
```

## 2.3 线性回归模型

- 参考资料：<http://blog.17baishi.com/8271/>

模型训练

```
#模型训练
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train, y_train)

y_lr_predict = lr.predict(X_test)
```

使用梯度下降算法，寻找最优解

```
#梯度下降
from sklearn.linear_model import SGDRegressor

sgd = SGDRegressor()
sgd.fit(X_train, y_train)

y_sgd_predict = sgd.predict(X_test)

D:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:993: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

带有正则化的岭回归

```
# 带有正则化的岭回归
from sklearn.linear_model import Ridge

rd = Ridge(alpha=0.01)
rd.fit(x_train, y_train)
y_rd_predict = rd.predict(x_test)
y_rd_predict = std_y.inverse_transform(y_rd_predict)
print(rd.coef_)
```

```
[[ -0.05463627 -0.09609018 -0.03417016  0.01490075  0.19886761 -0.02573944  
   0.03537826  0.13540342 -0.0554492   0.03556055 -0.061569   -0.05282228  
  -0.053717 ]]
```

模型评估：

- **解析：**可能是由于数据量较少，所以 linear regression 的方差要比 SGD 的方差小。

```
print("lr的均方误差为: ", mean_squared_error(std_y.inverse_transform(y_test), y_lr_predict))  
print("SGD的均方误差为: ", mean_squared_error(std_y.inverse_transform(y_test), y_sgd_predict))  
print("Ridge的均方误差为: ", mean_squared_error(std_y.inverse_transform(y_test), y_rd_predict))  
lr的均方误差为:  0.32084140701786895  
SGD的均方误差为:  0.3353607382012472  
Ridge的均方误差为:  1.041363130238943
```

## 2.4 SVR 模型

模型训练，分别使用线性核函数和多项式核函数配置支持向量机。

```
#模型训练  
from sklearn.svm import SVR  
  
# 线性核函数配置支持向量机  
linear_svr = SVR(kernel="linear")  
  
# 训练  
linear_svr.fit(x_train, y_train)  
  
# 预测 保存预测结果  
linear_svr_y_predict = linear_svr.predict(x_test)  
  
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:993: DataConversionWarning:  
A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().  
y = column_or_1d(y, warn=True)
```

```
# 多项式核函数配置支持向量机  
poly_svr = SVR(kernel="poly")  
  
# 训练  
poly_svr.fit(x_train, y_train)  
  
# 预测 保存预测结果  
poly_svr_y_predict = linear_svr.predict(x_test)
```

```
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:993: DataConversionWarning:  
A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().  
y = column_or_1d(y, warn=True)
```

- SVR 模型评估参考资料: <https://blog.csdn.net/u013090676/article/details/99697628>

```
# 线性核函数 模型评估
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

print("线性核函数支持向量机的默认评估值为: ", linear_svr.score(x_test, y_test))
print("线性核函数支持向量机的R_squared值为: ", r2_score(y_test, linear_svr_y_predict))
print("线性核函数支持向量机的均方误差为: ", mean_squared_error(ss_y.inverse_transform(y_te
ss_y.inverse_transform(linear_svr_y_predict)))
print("线性核函数支持向量机的平均绝对误差为: ", mean_absolute_error(ss_y.inverse_transform
ss_y.inverse_transform(linear_svr_y_predict)))

# 对多项式核函数模型评估
print("对多项式核函数的默认评估值为: ", poly_svr.score(x_test, y_test))
print("对多项式核函数的R_squared值为: ", r2_score(y_test, poly_svr_y_predict))
print("对多项式核函数的均方误差为: ", mean_squared_error(ss_y.inverse_transform(y_test),
ss_y.inverse_transform(poly_svr_y_predict)))
print("对多项式核函数的平均绝对误差为: ", mean_absolute_error(ss_y.inverse_transform(y_te
```

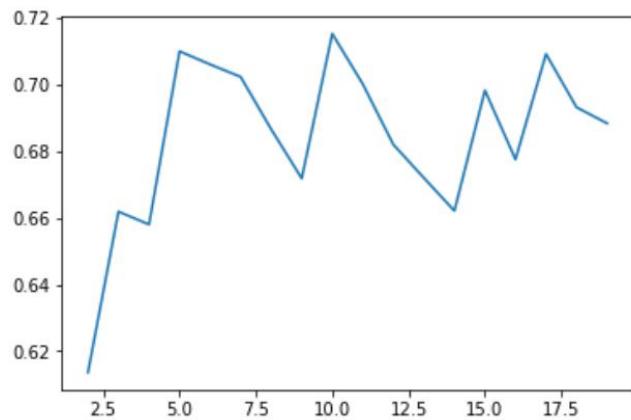
线性核函数支持向量机的默认评估值为: 0.651717097429608  
 线性核函数支持向量机的R\_squared值为: 0.651717097429608  
 线性核函数支持向量机的均方误差为: 27.0063071393243  
 线性核函数支持向量机的平均绝对误差为: 3.426672916872753  
 对多项式核函数的默认评估值为: 0.40445405800289286  
 对多项式核函数的R\_squared值为: 0.651717097429608  
 对多项式核函数的均方误差为: 27.0063071393243  
 对多项式核函数的平均绝对误差为: 3.426672916872753

## 2.5 决策树模型

```
# 导入相关模块
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import load_boston
import matplotlib.pyplot as plt
import random
'''注: 回归树的叶节点的数据类型是连续型, 节点的值是‘一系列’数的均值’'''

depth = [i for i in range(2, 20)]
n = 0
d_list = []
score_list = []
for i in depth:
    n = n + 1
    decision_i = DecisionTreeRegressor(criterion='mse', max_depth=i)
    # 训练回归树模型
    decision_i.fit(X_train, y_train)
    # 使用测试数据检验回归树结果
    y_pre_decision_i = decision_i.predict(X_test)
    decision_score = decision_i.score(X_test, y_test)
    print('回归树深度为 {} 预测的准确性'.format(i), decision_score)
    d_list.append(i)
    score_list.append(decision_score)
plt.plot(d_list, score_list)
plt.show()
```

由下图可知，当树深度为 10 时，可以取得比较好的准确度。



---

回归树深度为 2 预测的准确性 0.613613004828102  
回归树深度为 3 预测的准确性 0.6617956915007512  
回归树深度为 4 预测的准确性 0.6579498100171595  
回归树深度为 5 预测的准确性 0.7098188670365968  
回归树深度为 6 预测的准确性 0.7058786912075699  
回归树深度为 7 预测的准确性 0.7021575491498087  
回归树深度为 8 预测的准确性 0.6864674934059856  
回归树深度为 9 预测的准确性 0.6717406550672335  
回归树深度为 10 预测的准确性 0.7150716175963556  
回归树深度为 11 预测的准确性 0.6999979067476387  
回归树深度为 12 预测的准确性 0.6818916625456921  
回归树深度为 13 预测的准确性 0.6718934753334852  
回归树深度为 14 预测的准确性 0.6620266384440447  
回归树深度为 15 预测的准确性 0.698116030885003  
回归树深度为 16 预测的准确性 0.6774177653121556  
回归树深度为 17 预测的准确性 0.7090086217757551  
回归树深度为 18 预测的准确性 0.6930289406878578  
回归树深度为 19 预测的准确性 0.6882343972259009

## 机器学习与数据挖掘-上机作业四

上机任务：

复现 topic05、topic06 的代码，在特征工程的基础上，分别用一个集成学习（如 xgb 或 LGB）预测 cancer、boston 房价的测试数据。

采用随机森林算法实现 breast cancer 和 boston 房价的数据预测。

·**随机森林（Random Forest）**：是 Bagging 的一种，但是有一点区别是随机森林在构建决策树的时候，会随机选择样本特征中的一部分来进行划分。由于随机森林的二重随机性，它具有良好的学习性能。以随机森林为代表的装袋法的训练过程旨在降低方差，即降低模型复杂度。

### 目录

1 随机森林预测 breast cancer 数据.....	1
1.1 数据导入.....	1
1.2 模型训练.....	2
1.3 模型调优.....	2
2 随机森林预测 Boston 房价数据.....	5
2.1 导入数据、查看数据基本信息.....	5
2.2 切分数据集.....	5
2.3 寻找最优超参数.....	5

## 1 随机森林预测 breast cancer 数据

· 参考资料：<https://blog.csdn.net/sjjsaaaa/article/details/110201530>

### 1.1 数据导入

```
from sklearn.datasets import load_breast_cancer#数据
from sklearn.ensemble import RandomForestClassifier#分类器
from sklearn.model_selection import GridSearchCV#网格搜索
from sklearn.model_selection import cross_val_score#交叉验证
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

乳腺癌数据集有 569 条记录，30 个特征。虽然维度不太高，但是样本量少。可能存在过拟合的情况。

```
#数据集
data = load_breast_cancer()
data
data.data.shape#查看数据维度
data.target#标签
```

## 1.2 模型训练

进行一次简单的建模，看看模型本身在数据集上的效果。

```
rfc = RandomForestClassifier(n_estimators=100, random_state=90)#实例化
score_pre = cross_val_score(rfc, data.data, data.target, cv=10).mean()#交叉验证 取平均数
score_pre#准确度
0.9648809523809524
```

## 1.3 模型调优

首先调 `n_estimators`，画出学习曲线。

在调试过程中我们遇到了一些问题：将 `fit_params` 传递给 `sklearn` 中的 `cross_val_score`，它总是返回 `nan`。

后来发现，需要调整一些函数参数，可以参考如下资料：

- 参考资料：<https://www.pythonheidong.com/blog/article/861651/81abd43653c065519da1/>

```
scorel = []
for i in range(0, 200, 10):
    rfc = RandomForestClassifier(n_estimators=i+1, random_state = 90)
    parameters = {'n_estimators':i+1}
    score = cross_val_score(rfc, data.data, data.target, cv = 10).mean()
    scorel.append(score)

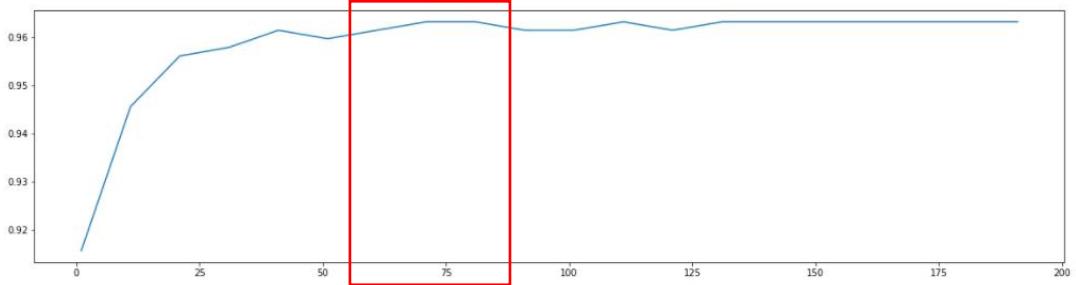
print(max(scorel), (scorel.index(max(scorel))*10)+1)#打印出做大的值，并输出其索引位置
print(scorel)
plt.figure(figsize=[20, 5])#画布
plt.plot(range(1, 201, 10), scorel)
plt.show()
```

观察 `n_estimators` 在何时开始变得平稳，是否一直推动模型整体准确率的上升等信息，第一次的学习曲线，可以先用来帮助我们划定范围，我们取每十个数作为一个阶段，来观察 `n_estimators` 的变化如何引起模型整体准确率的变化。

```

0.9631265664160402 71
[0.9156641604010025, 0.9455513784461151, 0.9560463659147869, 0.9578320802005011, 0.9613
721804511279, 0.9596177944862155, 0.9613721804511279, 0.9631265664160402, 0.96312656641
60402, 0.9613721804511279, 0.9613721804511279, 0.9631265664160402, 0.9613721804511279,
0.9631265664160402, 0.9631265664160402, 0.9631265664160402, 0.9631265664160402, 0.96312
65664160402, 0.9631265664160402, 0.9631265664160402]

```



从上图中我们可以将范围缩小到 60~80 之间，然后缩小范围，进行更精确的搜寻。

```

score1_2 = []

for i in range(60, 80):
    rfc = RandomForestClassifier(n_estimators=i+1, random_state = 90)
    parameters = {'n_estimators':i+1}
    score = cross_val_score(rfc, data.data, data.target, cv = 10).mean()
    score1_2.append(score)

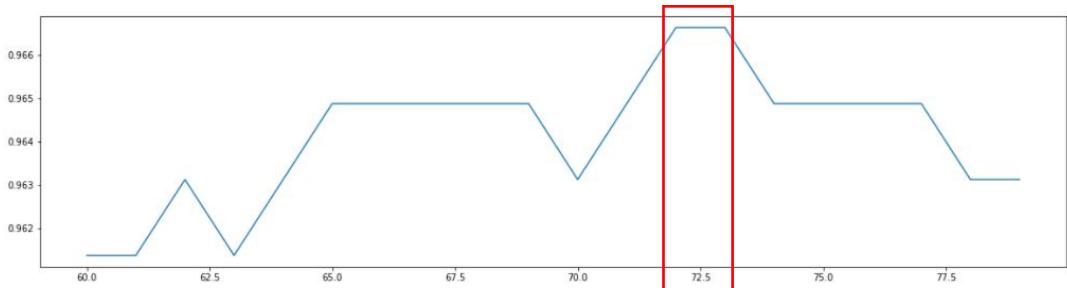
print(max(score1_2), (score1_2.index(max(score1_2))*10)+1)#打印出做大的值，并输出其索引位置
print(score1_2)
plt.figure(figsize=[20, 5])#画布
plt.plot(range(60, 80), score1_2)
plt.show()

```

```

0.9666353383458647 121
[0.9613721804511279, 0.9613721804511279, 0.9631265664160402, 0.9613721804511279, 0.9631
265664160402, 0.9648809523809524, 0.9648809523809524, 0.9648809523809524, 0.9648809523
09524, 0.9648809523809524, 0.9631265664160402, 0.9648809523809524, 0.9666353383458647,
0.9666353383458647, 0.9648809523809524, 0.9648809523809524, 0.9648809523809524, 0.9648809523809524,
0.9648809523809524, 0.96312656641604, 0.963126566416042]

```



从这里可以看出，当 n\_estimators 取 73 左右时，效果较好。

之后，进行网格搜索算法，进行参数的调整。

## (1) 调整深度

```

garam_grid = {"max_depth":np.arange(1, 20, 1)}

rfc = RandomForestClassifier(n_estimators=73, random_state = 90)

GS = GridSearchCV(rfc, garam_grid, cv=10)
GS.fit(data.data, data.target)

print("查看最高评分:", GS.best_score_)
print("查看最高评分的数:", GS.best_params_)

```

查看最高评分: 0.9666353383458647  
 查看最高评分的数: {'max\_depth': 8}

## (2) 调整 max\_feature

### 调整max\_feature

```

garam_grid = {"max_features":np.arange(8, 70, 1)}

rfc = RandomForestClassifier(n_estimators=73, random_state = 90)

GS = GridSearchCV(rfc, garam_grid, cv=10)
GS.fit(data.data, data.target)

print("查看最高评分:", GS.best_score_)
print("查看最高评分的数:", GS.best_params_)

```

查看最高评分: 0.9666666666666668  
 查看最高评分的数: {'max\_features': 24}

## (3) 调整 min\_samples\_leaf

### 调整min\_samples\_leaf参数

```

garam_grid = {"min_samples_leaf":np.arange(1, 1+10, 1)}

rfc = RandomForestClassifier(n_estimators=73, random_state = 90)

GS = GridSearchCV(rfc, garam_grid, cv=10)
GS.fit(data.data, data.target)

print("查看最高评分:", GS.best_score_)
print("查看最高评分的数:", GS.best_params_)

```

查看最高评分: 0.9666353383458647  
 查看最高评分的数: {'min\_samples\_leaf': 1}

## (4) 最后，总结出最佳模型参数（精确度为 0.9666）

### 总结出模型的最佳参数

```

rfc = RandomForestClassifier(n_estimators=73
                            , random_state=90
                            , max_features=24)

score = cross_val_score(rfc, data.data, data.target, cv=10).mean()
print("评分:", score)

```

评分: 0.9666666666666668

## 2 随机森林预测 Boston 房价数据

### 2.1 导入数据、查看数据基本信息

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score

boston = load_boston()
#将数据集转化为dataframe
bostonDf_X = pd.DataFrame(boston.data, columns=boston.feature_names)
bostonDf_y = pd.DataFrame(boston.target, columns=['houseprice']) #注意加列名称

#合并dataframe
bostonDf = pd.concat([bostonDf_X, bostonDf_y], axis=1) #axis=1为横向操作
bostonDf.shape #(506, 14)
bostonDf.dropna(inplace=True) #消除空值
bostonDf.head() #看一下数据集
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

### 2.2 切分数据集

按照 7: 3 切分数据集，随机种子选择 17

```
x = bostonDf.drop(["houseprice"], axis = 1) #x选取特征值
y = bostonDf["houseprice"] #y选取房价

#切分数据集
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 17)
```

### 2.3 寻找最优超参数

```
#定义网格搜索
param_grid = {
    "n_estimators": [5, 10, 20, 100, 200], #数值均可预设
    "max_depth": [3, 5, 7],
    "max_features": [0.6, 0.7, 0.8, 1]
}
rf = RandomForestRegressor()
grid = GridSearchCV(rf, param_grid=param_grid, cv = 3) #在网格搜索前提下训练，调参助手——grid
grid.fit(x_train, y_train) #训练
```

```
GridSearchCV(cv=3, estimator=RandomForestRegressor(),
            param_grid={'max_depth': [3, 5, 7],
                        'max_features': [0.6, 0.7, 0.8, 1],
                        'n_estimators': [5, 10, 20, 100, 200]})
```

得出，最优的参数组合为{'max\_depth': 7, 'max\_features': 0.6, 'n\_estimators': 200}

```
grid.best_params_ #查看最好参数  
{'max_depth': 7, 'max_features': 0.6, 'n_estimators': 200}
```

```
model = grid.best_estimator_ #选中最好的参数作为模型参数  
model
```

```
RandomForestRegressor(max_depth=7, max_features=0.6, n_estimators=200)
```

```
model.feature_importances_ #特征重要度分析，数值越大，影响越大  
array([0.04734209, 0.00421687, 0.02603037, 0.00334109, 0.04640768,  
       0.30673189, 0.02391322, 0.05043885, 0.00467932, 0.01899062,  
       0.04111584, 0.01460455, 0.41218762])
```

```
model.predict(x_test) #预测  
array([28.40442125, 22.37424939, 19.46417455, 43.96882784, 24.75359184,  
      20.41817838, 46.25312567, 22.58619847, 9.61020576, 27.5204651,  
      18.86072607, 21.4836993, 21.17809754, 15.33038656, 21.32131705,  
      21.1118148, 21.53763488, 15.17907356, 19.73712307, 19.90476747,  
      28.87448453, 22.66892757, 15.92348782, 14.79868115, 25.58179192,  
      37.85557318, 15.37639702, 10.2388908, 18.55420256, 22.41103936,  
      22.82894908, 20.36724423, 23.01395715, 21.30410975, 26.03952481,  
      18.69089546, 30.92707482, 11.33321114, 20.16053681, 44.55566753,  
      9.36086939, 22.78966117, 23.97863409, 23.56364876, 19.63977783,  
      16.06352924, 31.13266014, 24.96711594, 22.53487087, 14.7253649]
```

计算 mse 均分误差

```
#计算mse均分误差，开根号得均方根误差  
MSE = metrics.mean_squared_error(y_test, model.predict(x_test))  
MSE
```

```
8.472541885582093
```