导入必要的库

In [ ]:
```
from mlxtend.plotting import plot_decision_regions
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

流程

- 获取数据
- 数据清洗
- 探索数据找到规律和趋势
- 建立模型预测
- 评估模型

In [ ]:
```
#读取数据
diabetes_data = pd.read_csv('./diabetes_data.csv')

diabetes_data.head()
```

Out[ ]:

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age	diabetes
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

EDA和统计分析

获取数据类型，数据的列属性，非空值的个数，内存使用情况等等

In [ ]:
```
diabetes_data.info(verbose=True) #verbose=True 显示详细信息
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   pregnancies  768 non-null    int64  
 1   glucose      768 non-null    int64  
 2   diastolic    768 non-null    int64  
 3   triceps      768 non-null    int64  
 4   insulin      768 non-null    int64  
 5   bmi          768 non-null    float64
 6   dpf          768 non-null    float64
```

```
 7   age          768 non-null    int64
 8   diabetes     768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

In [ ]:

```
diabetes_data.describe()
#只显示数字列，要查看其他详细的数据可以设置include='all'参数
```

Out[ ]:

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000

In [ ]:

```
diabetes_data.describe().T #(转置查看)
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
pregnancies	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00000	17.00
glucose	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25000	199.00
diastolic	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00000	122.00
triceps	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00000	99.00
insulin	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25000	846.00
bmi	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60000	67.10
dpf	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62625	2.42
age	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00000	81.00
diabetes	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00000	1.00

观察describe中显示的数据,我们可以察觉到一些问题,0是没有意义的值(人体的bmi，血压等数据不应该出现0),所以,我们将0当作缺失值.

在下面的列中出现了"0":

1. glucose
2. diastolic
3. triceps
4. insulin
5. bmi

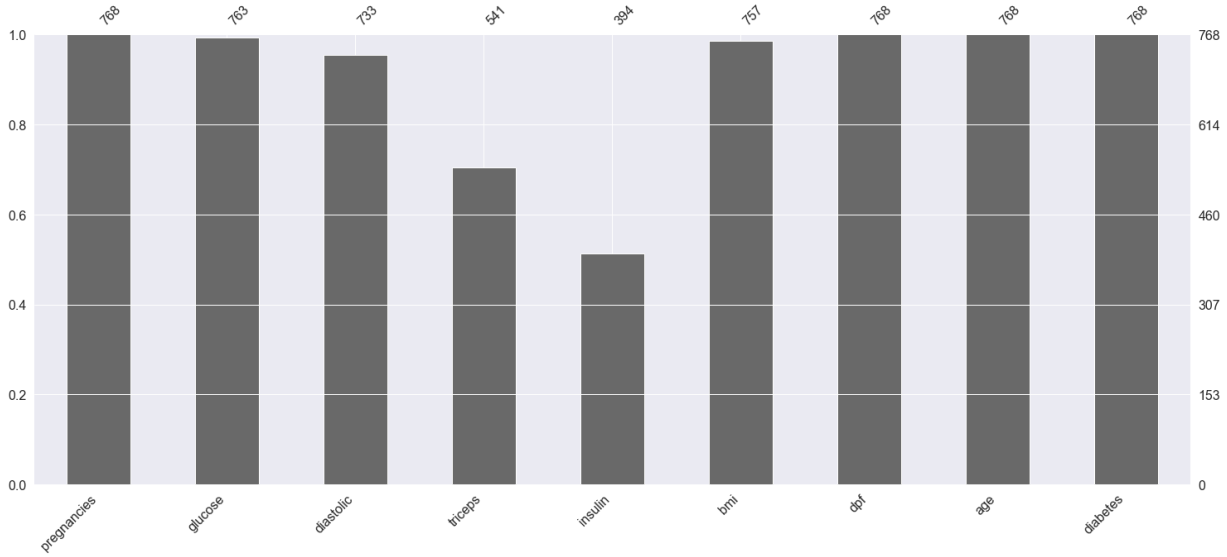
因此，可以先将这些列出现的0值改成NAN，这样比较好处理.

In [ ]:
```
diabetes_data_copy = diabetes_data.copy(deep = True)
diabetes_data_copy[['glucose','diastolic','triceps','insulin','bmi']] = diabetes_data_

## 显示每一列Nan的个数
print(diabetes_data_copy.isnull().sum())
```

```
pregnancies      0
glucose          5
diastolic       35
triceps        227
insulin        374
bmi             11
dpf              0
age              0
diabetes         0
dtype: int64
```

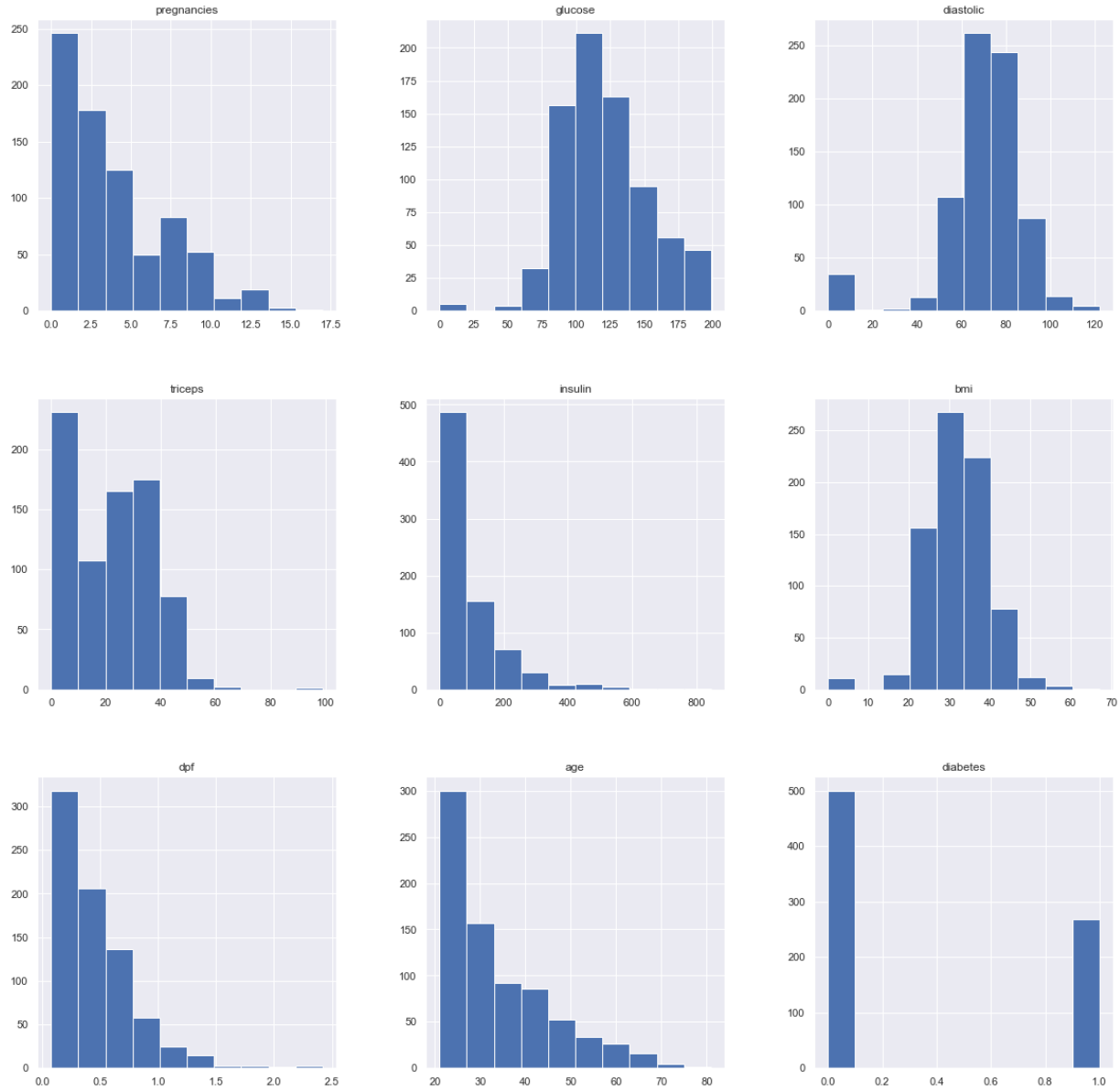
missingno库提供了一个灵活易用的可视化工具来观察数据缺失情况，是基于matplotlib的，接受pandas数据源

In [ ]:
```
## 空值个数分析
import missingno as msno
p=msno.bar(diabetes_data_copy)
```



为了填充NAN缺失值，需要了解每一列数据的分配

In [ ]:
```
#hist 用于绘制直方图
p = diabetes_data.hist(figsize = (20,20))
```



根据分布图，需要用均值或者中位数来填充缺失值

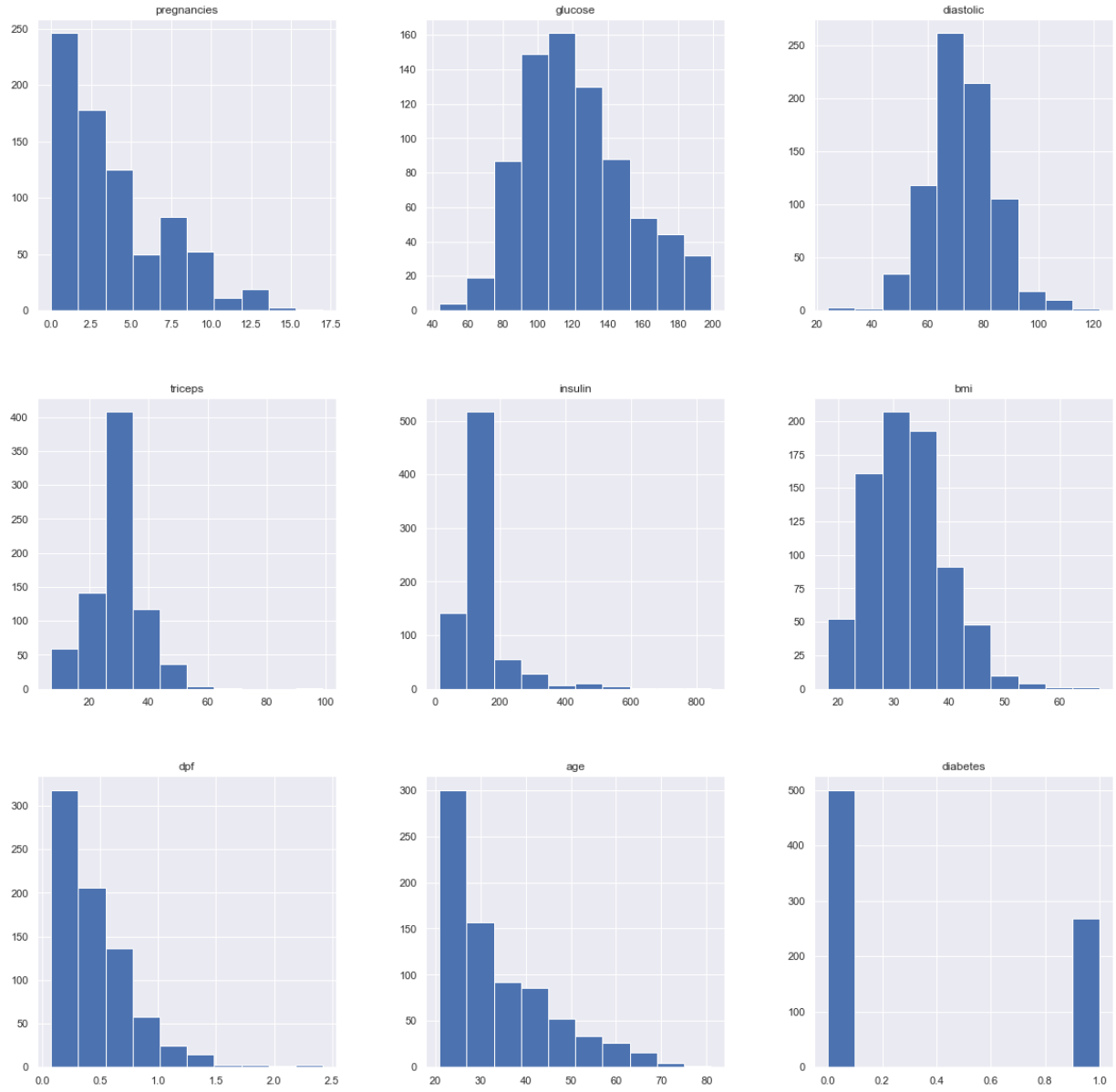
平均值：对于数据符合均匀分布，用该变量的均值填补缺失值。

中位数：对于数据存在倾斜分布的情况，采用中位数填补缺失值。

In [ ]:
```
diabetes_data_copy['glucose'].fillna(diabetes_data_copy['glucose'].mean(), inplace =
diabetes_data_copy['diastolic'].fillna(diabetes_data_copy['diastolic'].mean(), inplac
diabetes_data_copy['triceps'].fillna(diabetes_data_copy['triceps'].median(), inplace =
diabetes_data_copy['insulin'].fillna(diabetes_data_copy['insulin'].median(), inplace =
diabetes_data_copy['bmi'].fillna(diabetes_data_copy['bmi'].median(), inplace = True)
```

绘制处理完缺失值的图

In [ ]:
```
p = diabetes_data_copy.hist(figsize = (20,20))
```



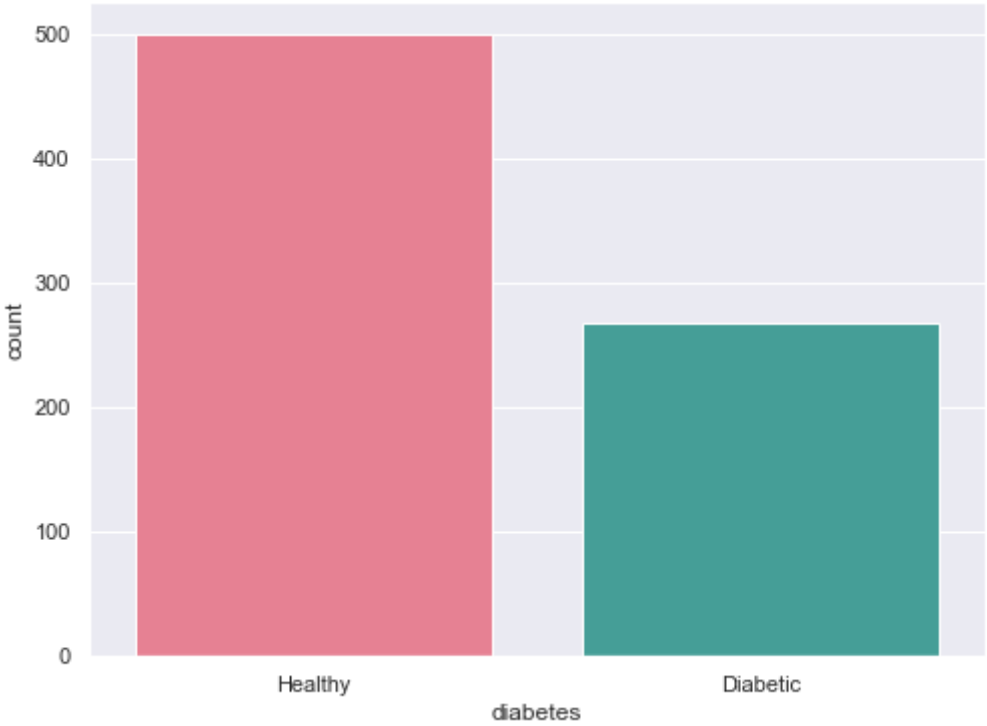
In [ ]:
```
## 观察数据的形状（行数和列数）
diabetes_data.shape
```

Out[ ]: (768, 9)

条状图查看目标特征diabetes

In [ ]:
```
from matplotlib.pyplot import figure, show

figure(figsize=(8,6))
ax = sns.countplot(x=diabetes_data_copy['diabetes'], data=diabetes_data_copy,palette=
ax.set_xticklabels(["Healthy","Diabetic"])
healthy, diabetics = diabetes_data_copy['diabetes'].value_counts().values
print("Samples of diabetic people: ", diabetics)
print("Samples of healthy people: ", healthy)
```

```
Samples of diabetic people:  268
Samples of healthy people:  500
```

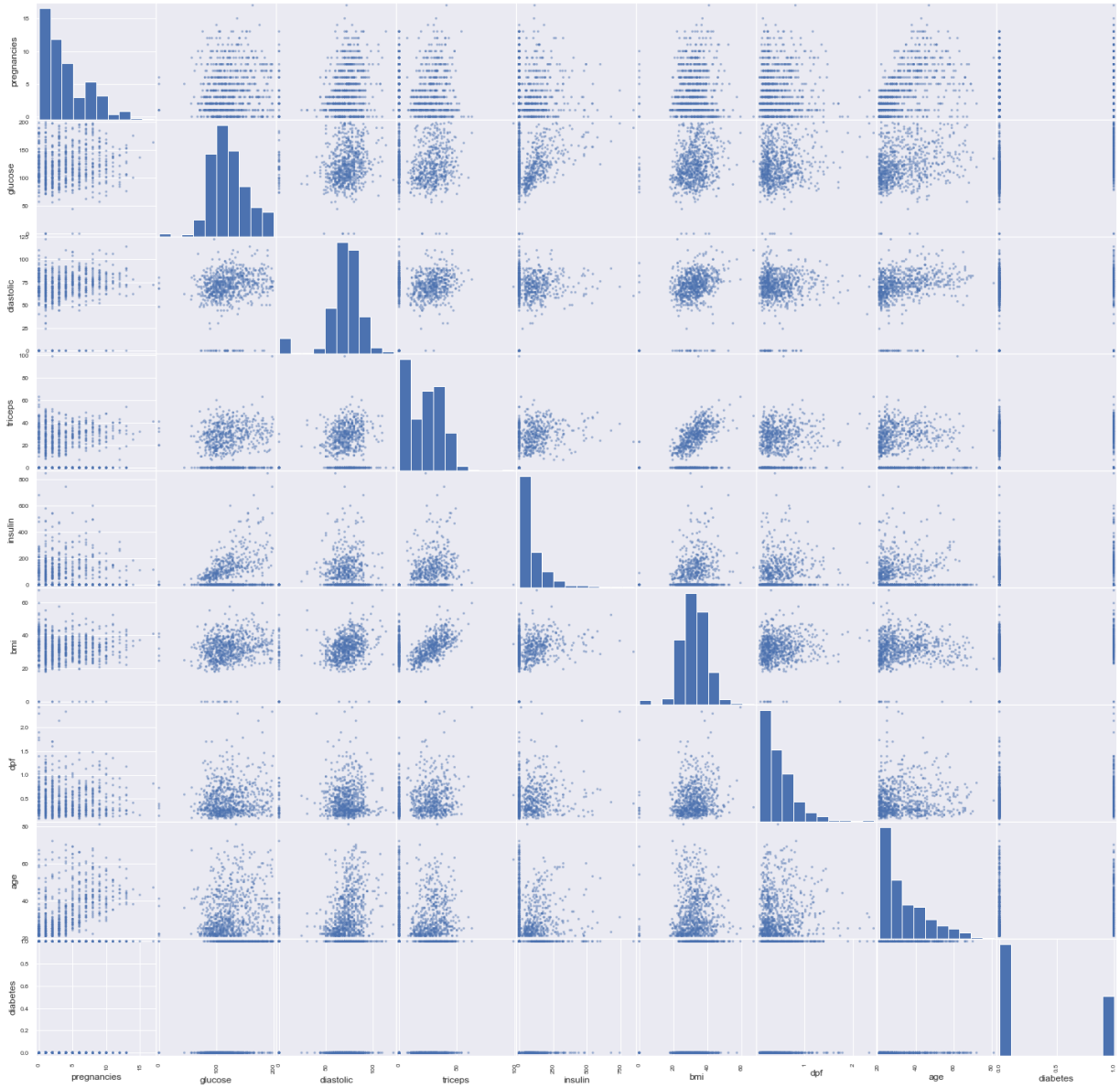


上图显示，数据偏向于结果值为0的数据点，这意味着实际上没有糖尿病患者的人数几乎是糖尿病患者的两倍

未清理数据的散点矩阵

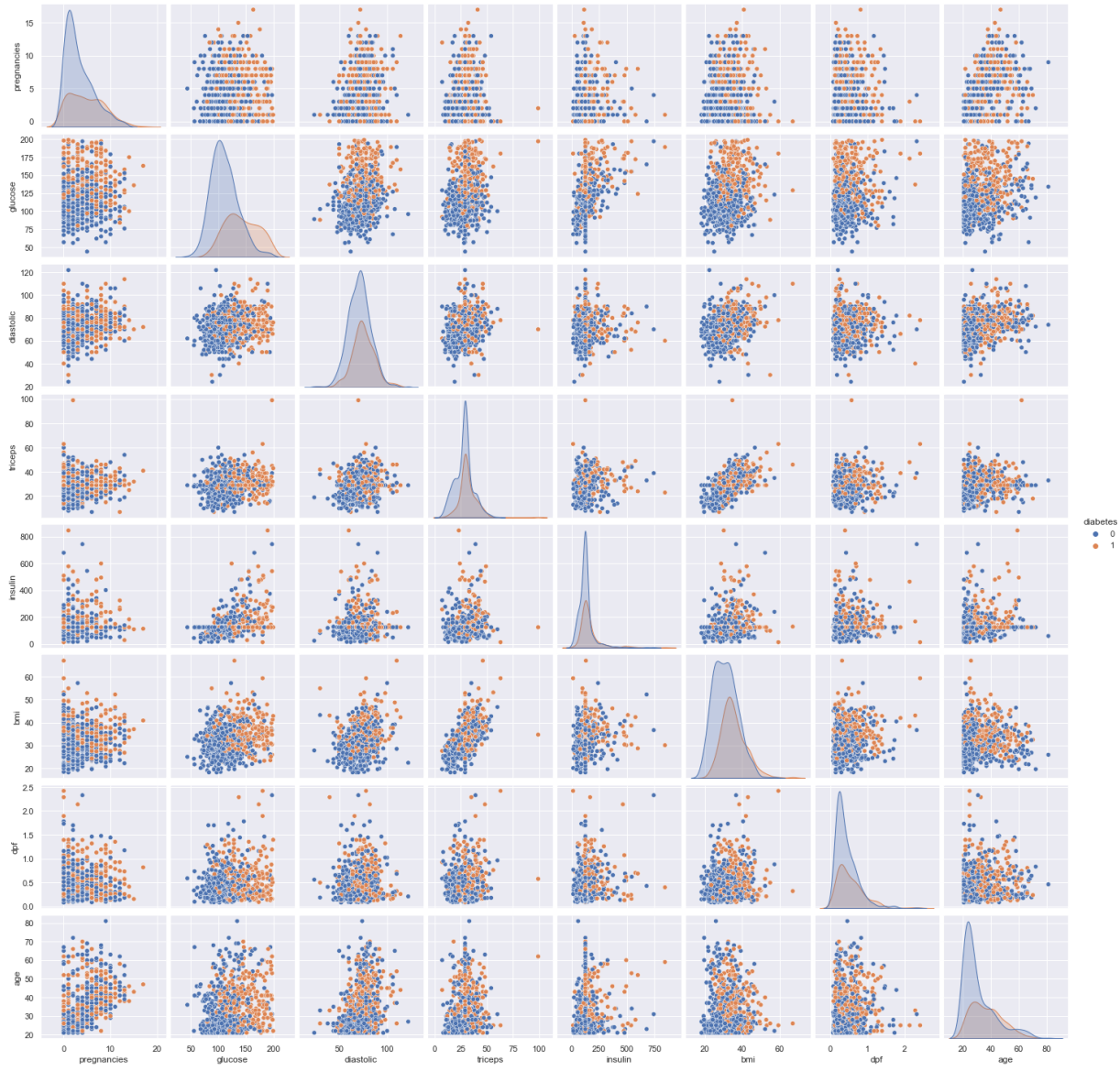
In [ ]:

```
#散点图矩阵图，可以两两发现特征之间的联系
from pandas.plotting import scatter_matrix
p=scatter_matrix(diabetes_data,figsize=(25, 25))
```



清洗完的数据 Pair plot

In [ ]:
```
#pairplot 主要展现的是变量两两之间的关系（线性或非线性，有无较为明显的相关关系）
p=sns.pairplot(diabetes_data_copy, hue = 'diabetes')
```

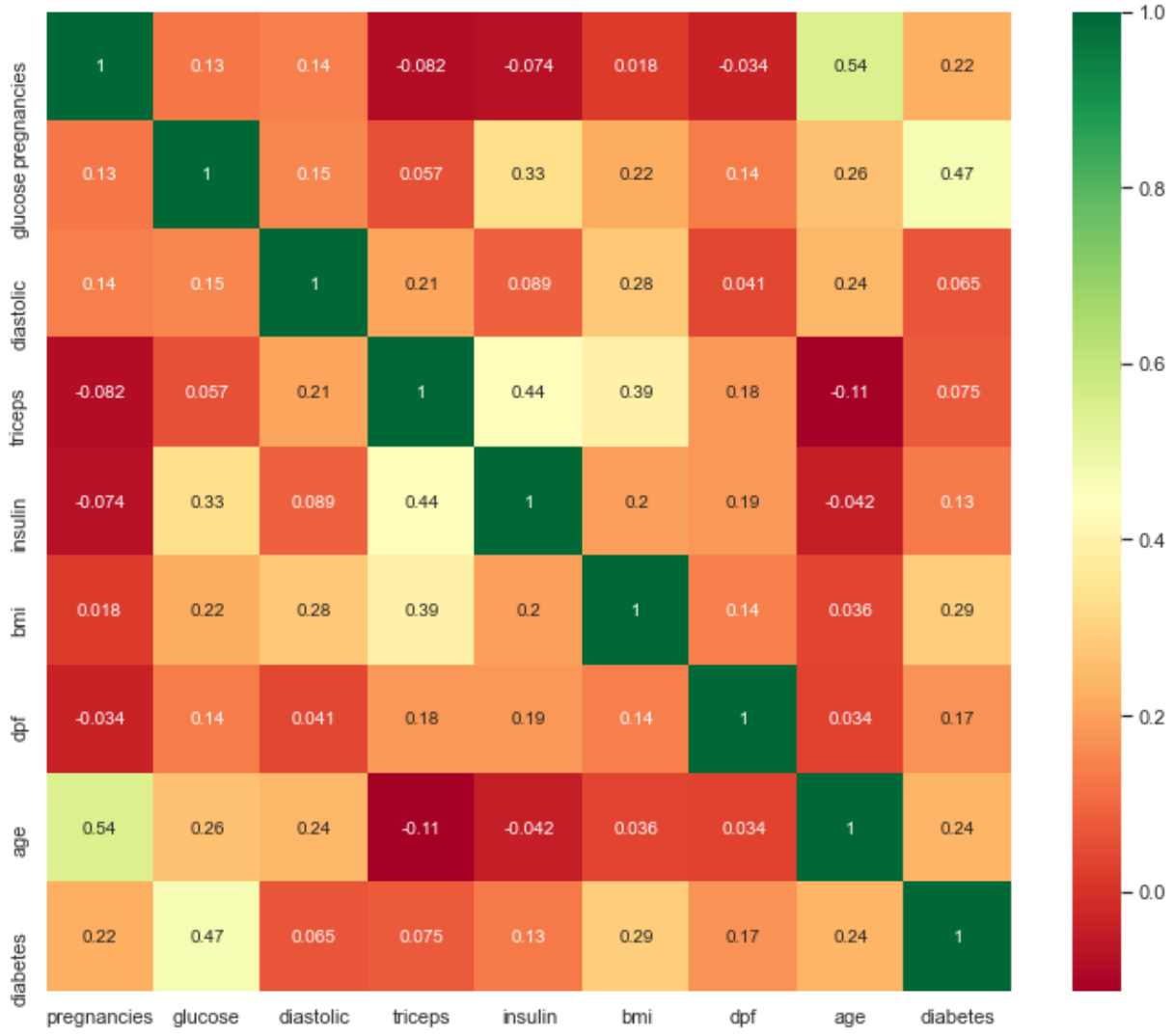


Heatmap

热图是借助颜色对信息的二维表示。热图可以帮助用户可视化简单或复杂的信息。

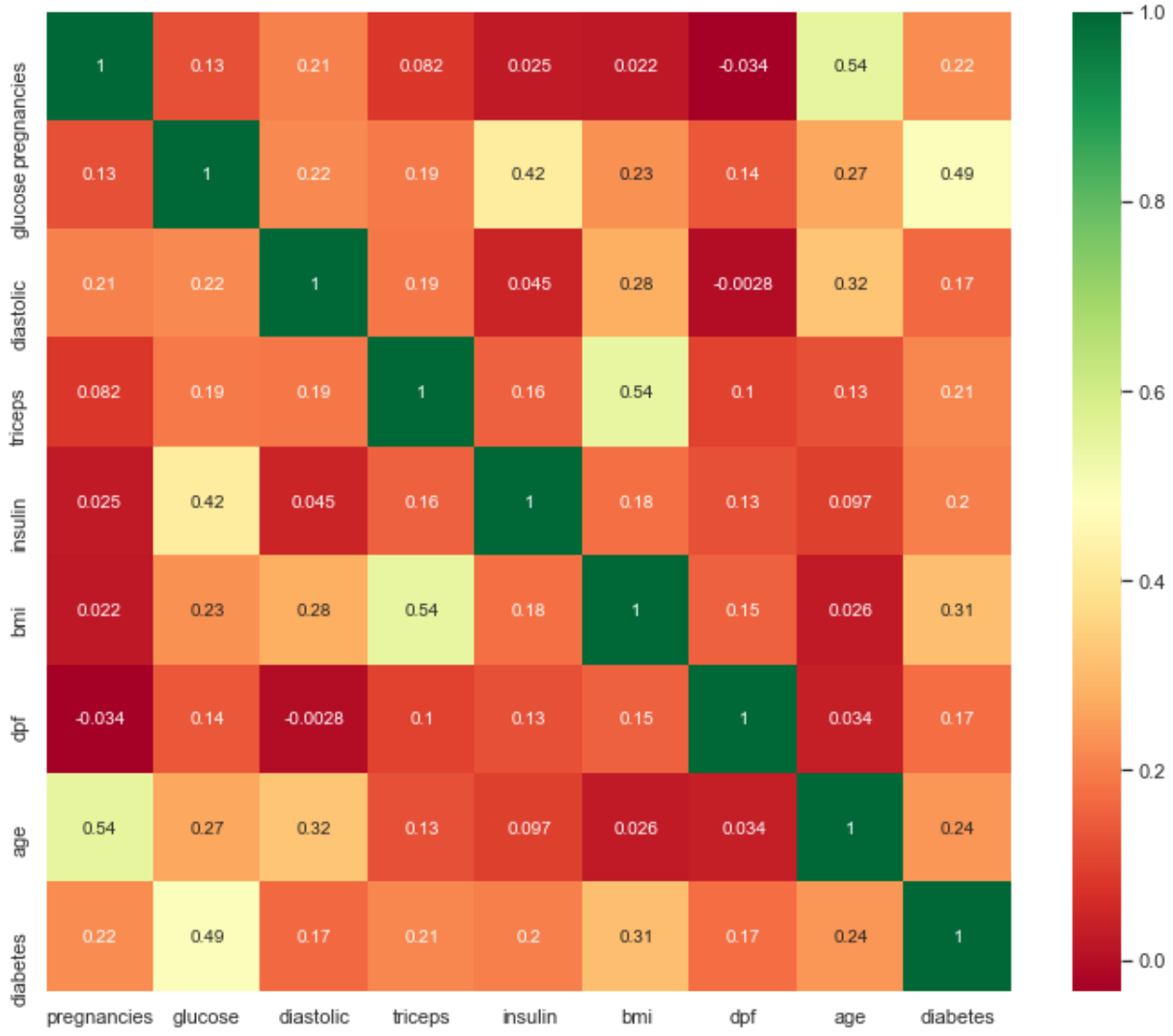
heatmap(热力图)是识别预测变量与目标变量相关性的方法，同时，也是发现变量间是否存在多重共线性的好方法。

In [ ]:
```
# 主要看最后一行和最后一列的数据
plt.figure(figsize=(12,10))
p=sns.heatmap(diabetes_data.corr(), annot=True,cmap ='RdYlGn')  # seaborn has very s
```



In [ ]:

```
plt.figure(figsize=(12,10))
p=sns.heatmap(diabetes_data_copy.corr(), annot=True,cmap ='RdYlGn')
```



多个模型的比较

1. 建立模型
2. 模型评估

1.决策树

导入必要的库

In [ ]:

```
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
```

数据集划分（未标准化）

In [ ]:
```
x_train,x_test,Y_train,Y_test = train_test_split(\
    diabetes_data_copy.loc[:,diabetes_data.columns !='diabetes'],\
    diabetes_data_copy['diabetes'],stratify=diabetes_data['diabetes'],test_size=0.3)
```

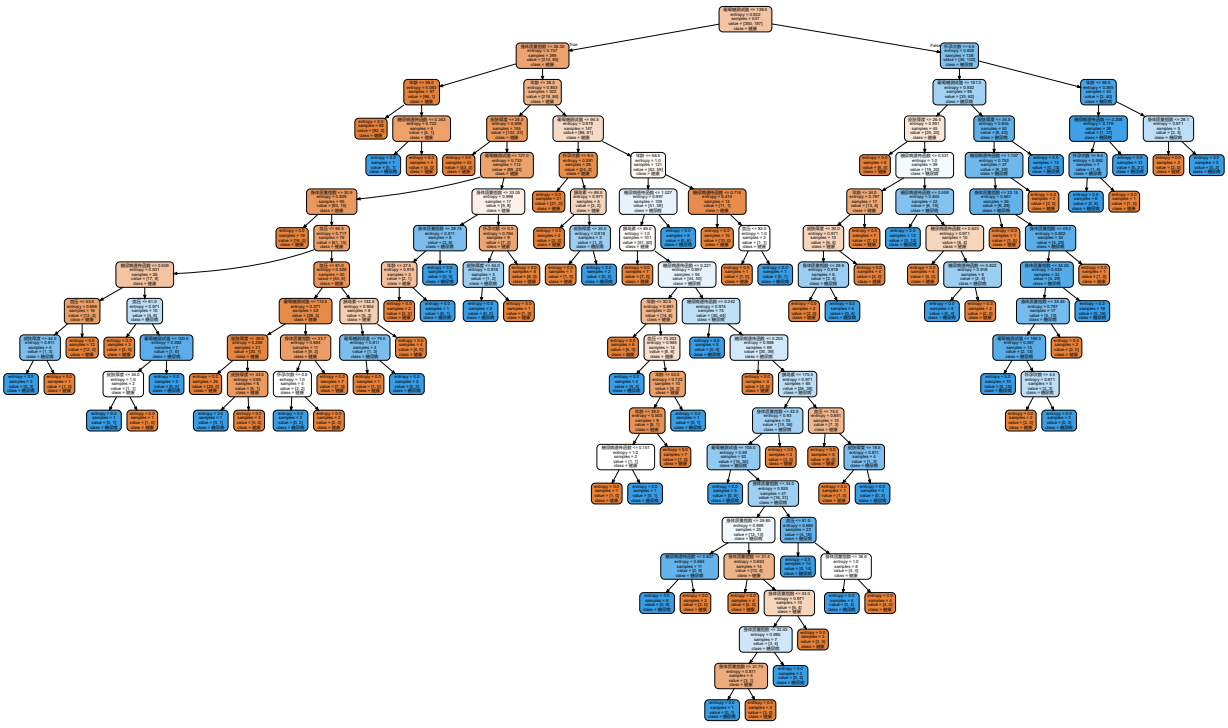
In [ ]:
```
clf=tree.DecisionTreeClassifier(criterion='entropy')
clf=clf.fit(x_train,Y_train)
score=clf.score(x_test,Y_test)
score
```

Out[ ]: 0.6536796536796536

没有标准化的数据便于可视化

In [ ]:
```
feature_name = ['怀孕次数','葡萄糖测试值','血压','皮肤厚度','胰岛素','身体质量指数','
import graphviz
dot_data = tree.export_graphviz(clf
                                ,out_file = None
                                ,feature_names= feature_name
                                ,class_names=["健康","糖尿病"] # 标签名
                                ,filled=True #填充颜色（不同颜色不同分类）
                                ,rounded=True # 圆角
                                )
graph = graphviz.Source(dot_data) # 可视化
graph
```

Out[ ]:



In [ ]:
```
#特征重要性
clf.feature_importances_    # 特征重要性
[*zip(feature_name,clf.feature_importances_)]
```

Out[ ]:
```
[('怀孕次数', 0.06490425317728087),
 ('葡萄糖测试值', 0.2639411789373503),
 ('血压', 0.08326527831678089),
 ('皮肤厚度', 0.09165012387945522),
 ('胰岛素', 0.03377306240953782),
```

```
('身体质量指数', 0.2145860755735277),
('糖尿病遗传函数', 0.1457431646316514),
('年龄', 0.10213686307441582)]
```

数据标准化

Z-score（standardization）

1. 思路：把所有数据归一到均值为0方差为1的分布中；
2. 公式：Xscale = (X - Xmean) / σ

- Xmean：特征的均值（均值就是平均值）；
- σ：每组特征值的标准差；
- X：每一个特征值；
- Xscale：归一化后的特征值；

1. 特点1：使用于数据分布没有明显的边界；（有可能存在极端的数据值）

- 归一化后，数据集中的每一种特征的均值为0，方差为1；

1. 优点（相对于最值归一化）：即使原数据集中有极端值，归一化有的数据集，依然满足均值为0方差为1，不会形成一个有偏的数据；

In [ ]:
```
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X =  pd.DataFrame(sc_X.fit_transform(diabetes_data_copy.drop(["diabetes"],axis = 1),)
        columns=['pregnancies', 'glucose', 'diastolic', 'triceps', 'insulin',
       'bmi', 'dpf', 'age'])
X.head()
```

Out[ ]:

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age
0	0.639947	0.865108	-0.033518	0.670643	-0.181541	0.166619	0.468492	1.425995
1	-0.844885	-1.206162	-0.529859	-0.012301	-0.181541	-0.852200	-0.365061	-0.190672
2	1.233880	2.015813	-0.695306	-0.012301	-0.181541	-1.332500	0.604397	-0.105584
3	-0.844885	-1.074652	-0.529859	-0.695245	-0.540642	-0.633881	-0.920763	-1.041549
4	-1.141852	0.503458	-2.680669	0.670643	0.316566	1.549303	5.484909	-0.020496

In [ ]:
```
#X = diabetes_data.drop("Outcome",axis = 1)
y = diabetes_data_copy.diabetes
print(y)
y.shape[0]
```

```
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
```

```
766    1
767    0
Name: diabetes, Length: 768, dtype: int64
```

Out[ ]:
```
768
```

标准化后的数据集划分

In [ ]:
```
#importing train_test_split
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3)
```

random_state

random_state用来设置分枝中的随机模式的参数，默认None，在高维度时随机性会表现更明显，低维度的数据（比如鸢尾花数据集），随机性几乎不会显现

In [ ]:
```
clf = tree.DecisionTreeClassifier(criterion="entropy",random_state=22)
clf = clf.fit(X_train, y_train)
score = clf.score(X_test, y_test) #返回预测的准确度
score
```

Out[ ]:
```
0.7359307359307359
```

剪枝

max_depth

限制树的最大深度，超过设定深度的树枝全部剪掉 这是用得最广泛的剪枝参数，在高维度低样本量时非常有效。决策树多生长一层，对样本量的需求会增加一倍，所以限制树深度能够有效地限制过拟合。在集成算法中也非常实用。

min_samples_leaf

限定一个节点在分枝后的每个子节点都必须包含至少min_samples_leaf个训练样本，否则分枝就不会发生，或者分枝会朝着满足每个子节点都包含min_samples_leaf个样本的方向去发生

确定最优的剪枝参数

GridSearch和CV，即网格搜索和交叉验证。GridSearchCV可以保证在指定的参数范围内找到精度最高的参数，但是这也是网格搜索的缺陷所在，他要求遍历所有可能参数的组合，在面对大数据集和多参数的情况下，非常耗时。

In [ ]:
```
dt = DecisionTreeClassifier()
para_dt = {'criterion':['gini','entropy'],'max_depth':np.arange(1, 50), 'min_samples_
grid_dt = GridSearchCV(dt, param_grid=para_dt, cv=5) #grid search decision tree for 5
#"gini" for the Gini impurity and "entropy" for the information gain.
#min_samples_leaf: The minimum number of samples required to be at a leaf node, have t
grid_dt.fit(X_train, y_train)
print("Best parameters for Decision Tree:", grid_dt.best_params_)
```

```
Best parameters for Decision Tree: {'criterion': 'entropy', 'max_depth': 5, 'min_sampl
es_leaf': 40}
```

模型评估

In [ ]:
```
dt = DecisionTreeClassifier(criterion='entropy', max_depth=5, min_samples_leaf=40)
dt = dt.fit(X_train, y_train)
score = dt.score(X_test, y_test) #返回预测的准确度
score
```

Out[ ]: 0.7878787878787878

In [ ]:
```
y_pred_dt = dt.predict(X_test)
accuracy = accuracy_score(y_test,y_pred_dt)
print('{:s} : {:.2f}'.format('Decision Tree的准确率为:', accuracy))
```

```
Decision Tree的准确率为: : 0.79
```

建立混淆矩阵

在分类任务下，预测结果(Predicted Condition)与正确标记(True Condition)之间存在四种不同的组合，构成混淆矩阵(适用于多分类)

预测结果

真实结果	正例	假例
正例	真正例TP	伪反例FN
假例	伪正例FP	真反例TN

In [ ]:
```
from sklearn.metrics import confusion_matrix
m_dt = confusion_matrix(y_test,y_pred_dt)
print('预测结果混淆矩阵：\n',m_dt)
```

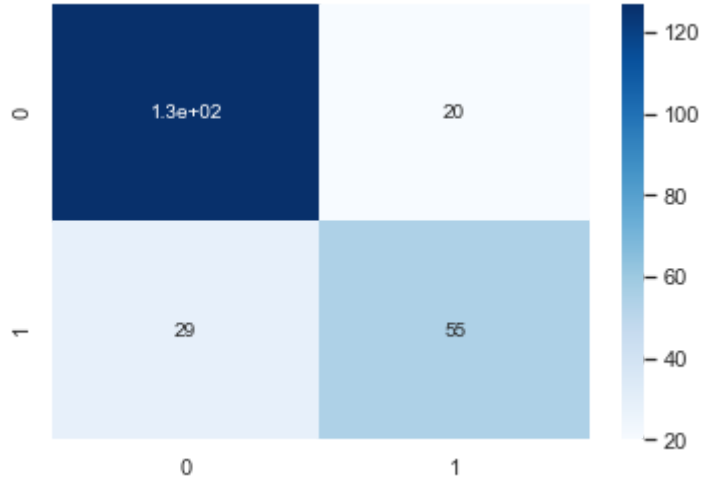
```
预测结果混淆矩阵：
 [[127  20]
 [ 29  55]]
```

可视化混淆矩阵

In [ ]:
```
%pylab inline
import seaborn as sns
sns.heatmap(m_dt,cmap='Blues',annot=True)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[ ]: <AxesSubplot:>



分类评估报告

sklearn.metrics.classification_report(y_true, y_pred, labels=[], target_names=None)

y_true：真实目标值

y_pred：估计器预测目标值

labels:指定类别对应的数字

target_names：目标类别名称

return：每个类别精确率与召回率

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_dt, labels=[0, 1], target_names=['健康', '
```

```
              precision    recall  f1-score   support

          健康       0.81      0.86      0.84       147
         糖尿病       0.73      0.65      0.69        84

    accuracy                           0.79       231
   macro avg       0.77      0.76      0.77       231
weighted avg       0.78      0.79      0.79       231
```

精确率(Precision)与召回率(Recall)

精确率：预测结果为正例样本中真实为正例的比例

召回率：真实为正例的样本中预测结果为正例的比例

总结

- 优点：
  - 简单的理解和解释，树木可视化。
- 缺点：
  - 决策树学习者可以创建不能很好地推广数据的过于复杂的树，会出现过拟合。
- 改进：
  - 剪枝cart算法
  - 随机森林

2.朴素贝叶斯算法

模型建立

In [ ]:
```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train,y_train)
score_gnb = gnb.score(X_test, y_test) #返回预测的准确度
score_gnb
```

Out[ ]: 0.7792207792207793

模型评估

In [ ]:
```
y_pred_gnb = gnb.predict(X_test)
accuracy_gnb = accuracy_score(y_test,y_pred_gnb)
print('{:s} : {:.2f}'.format('GaussianNB的准确率为:', accuracy_gnb))
```

GaussianNB的准确率为: : 0.78

建立混淆矩阵

In [ ]:
```
from sklearn.metrics import confusion_matrix
m_gnb = confusion_matrix(y_test,y_pred_gnb)
print('预测结果混淆矩阵：\n',m_gnb)
```

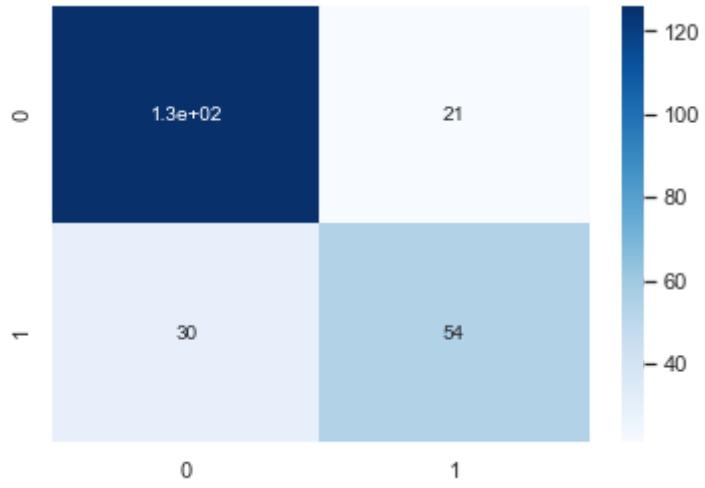
```
预测结果混淆矩阵：
 [[126  21]
 [ 30  54]]
```

可视化混淆矩阵

In [ ]:
```
%pylab inline
import seaborn as sns
sns.heatmap(m_gnb,cmap='Blues',annot=True)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[ ]: <AxesSubplot:>



分类评估报告

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_gnb))
```

```
              precision    recall  f1-score   support

           0       0.81      0.86      0.83       147
           1       0.72      0.64      0.68        84

    accuracy                           0.78       231
   macro avg       0.76      0.75      0.76       231
weighted avg       0.78      0.78      0.78       231
```

总结

- 优点:
  - 朴素贝叶斯模型发源于古典数学理论，有稳定的分类效率。
  - 对缺失数据不太敏感，算法也比较简单，常用于文本分类。
  - 分类准确度高，速度快
- 缺点:
  - 由于使用了样本属性独立性的假设，所以如果特征属性有关联时其效果不好

3.随机森林

模型建立(找到最优的超参数)

In [ ]:
```
rf = RandomForestClassifier()
params_rf = {'n_estimators':[100, 350, 500], 'min_samples_leaf':[2, 10, 30]}
grid_rf = GridSearchCV(rf, param_grid=params_rf, cv=5)
grid_rf.fit(X_train, y_train)
print("Best parameters for Random Forest:", grid_rf.best_params_)
```

```
Best parameters for Random Forest: {'min_samples_leaf': 2, 'n_estimators': 500}
```

模型评估

In [ ]:
```
rf = RandomForestClassifier(n_estimators=500, min_samples_leaf=2, random_state=42)
rf.fit(X_train, y_train)
```

Out[ ]:
```
RandomForestClassifier(min_samples_leaf=2, n_estimators=500, random_state=42)
```

In [ ]:
```
y_pred_rf = rf.predict(X_test)
accuracy_rf = accuracy_score(y_test,y_pred_rf)
print('{:s} : {:.2f}'.format('RandomForest的准确率为:', accuracy_rf))
```

```
RandomForest的准确率为: : 0.79
```

建立混淆矩阵

In [ ]:
```
from sklearn.metrics import confusion_matrix
m_rf = confusion_matrix(y_test,y_pred_rf)
print('预测结果混淆矩阵：\n',m_rf)
```

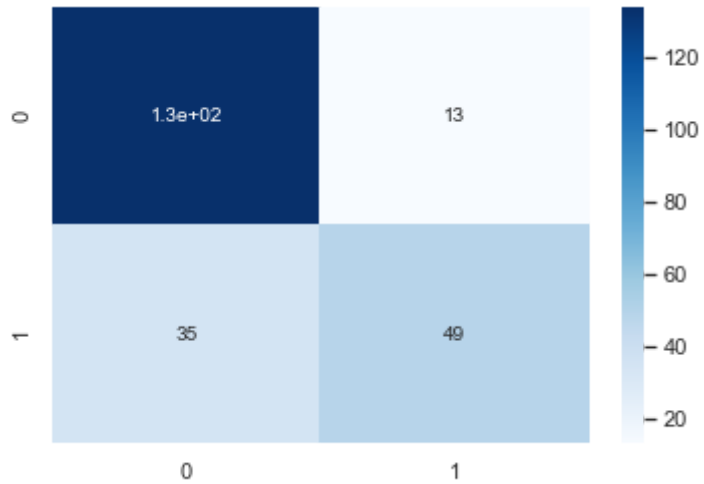
```
预测结果混淆矩阵：
 [[134  13]
 [ 35  49]]
```

可视化混淆矩阵

In [ ]:
```
%pylab inline
import seaborn as sns
sns.heatmap(m_rf,cmap='Blues',annot=True)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[ ]:
```
<AxesSubplot:>
```



分类评估报告

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_rf))
```

```
              precision    recall  f1-score   support

           0       0.79      0.91      0.85       147
           1       0.79      0.58      0.67        84

    accuracy                           0.79       231
   macro avg       0.79      0.75      0.76       231
weighted avg       0.79      0.79      0.78       231
```

总结

- 在当前所有算法中，具有极好的准确率
- 能够有效地运行在大数据集上，处理具有高维特征的输入样本，而且不需要降维
- 能够评估各个特征在分类问题上的重要性

4. 逻辑斯蒂回归

In [ ]:
```
lr = LogisticRegression(random_state=42)
lr.fit(X_train,y_train)
```

Out[ ]:
```
LogisticRegression(random_state=42)
```

In [ ]:
```
y_pred_lr = lr.predict(X_test)
accuracy_lr = accuracy_score(y_test,y_pred_lr)
print('{:s} : {:.2f}'.format('LogisticRegression的准确率为:', accuracy_lr))
```

```
LogisticRegression的准确率为: : 0.78
```

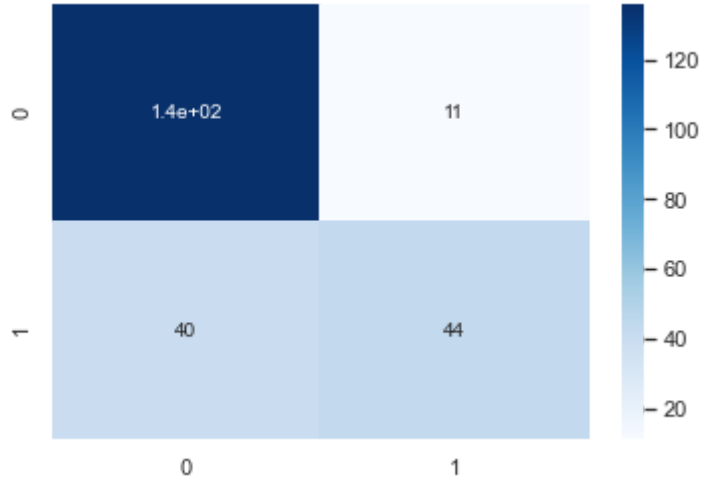
In [ ]:
```
from sklearn.metrics import confusion_matrix
m_lr = confusion_matrix(y_test,y_pred_lr)
print('预测结果混淆矩阵：\n',m_lr)
```

```
预测结果混淆矩阵：
 [[136  11]
 [ 40  44]]
```

In [ ]:
```
## 可视化混淆矩阵
%pylab inline
import seaborn as sns
sns.heatmap(m_lr,cmap='Blues',annot=True)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[ ]:
```
<AxesSubplot:>
```



分类评估报告

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_lr))
```

```
              precision    recall  f1-score   support

           0       0.77      0.93      0.84       147
           1       0.80      0.52      0.63        84

    accuracy                           0.78       231
   macro avg       0.79      0.72      0.74       231
weighted avg       0.78      0.78      0.77       231
```

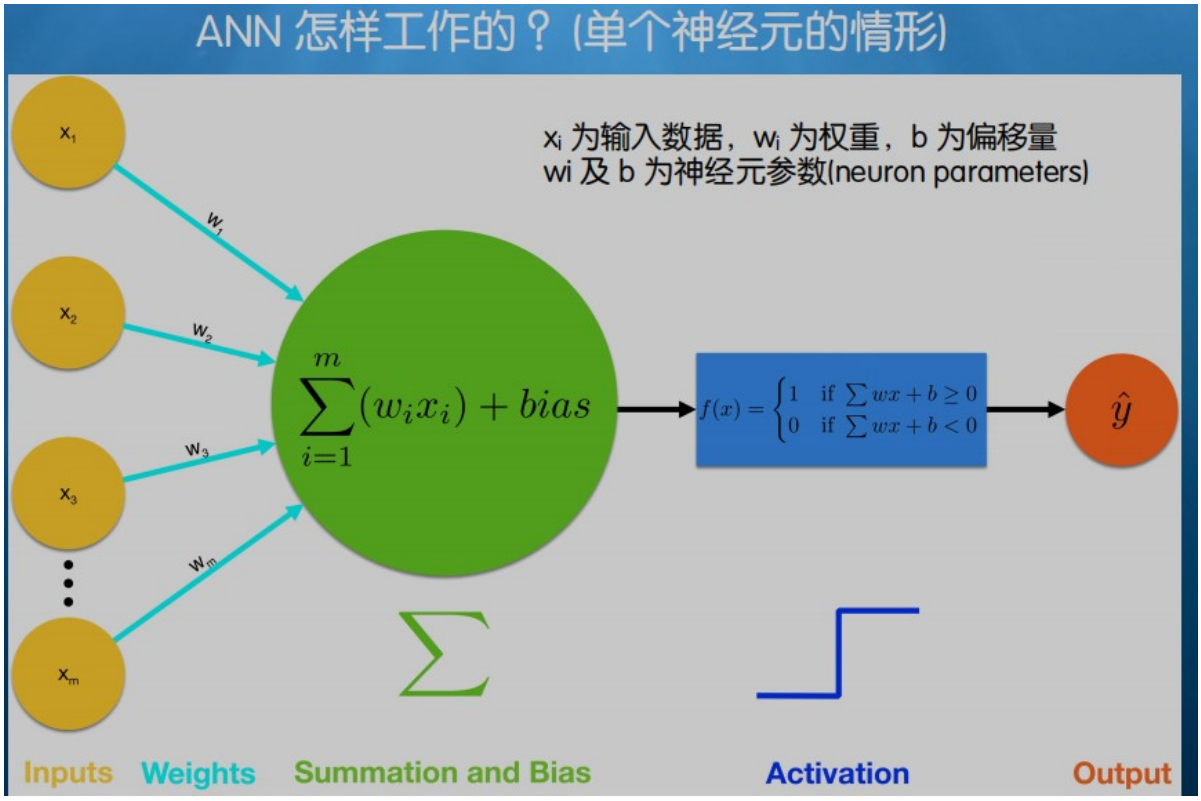
5. ANN 人工神经网络

人工神经网络(ANN) 是人脑神经系统的简单抽象，它像人类大脑一样处理信息。它由大量相互连接的神经元(neuron)组成并协同处理某个特定问题。

In [ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import optimizers,losses,activations,metrics
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.callbacks import LambdaCallback
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import OneHotEncoder, StandardScaler
```

In [ ]:
```
import sys
print(sys.version)
print("tf version=",tf.__version__)
```

```
3.7.8 (tags/v3.7.8:4b47a5b6ba, Jun 28 2020, 08:53:46) [MSC v.1916 64 bit (AMD64)]
tf version= 2.4.1
```

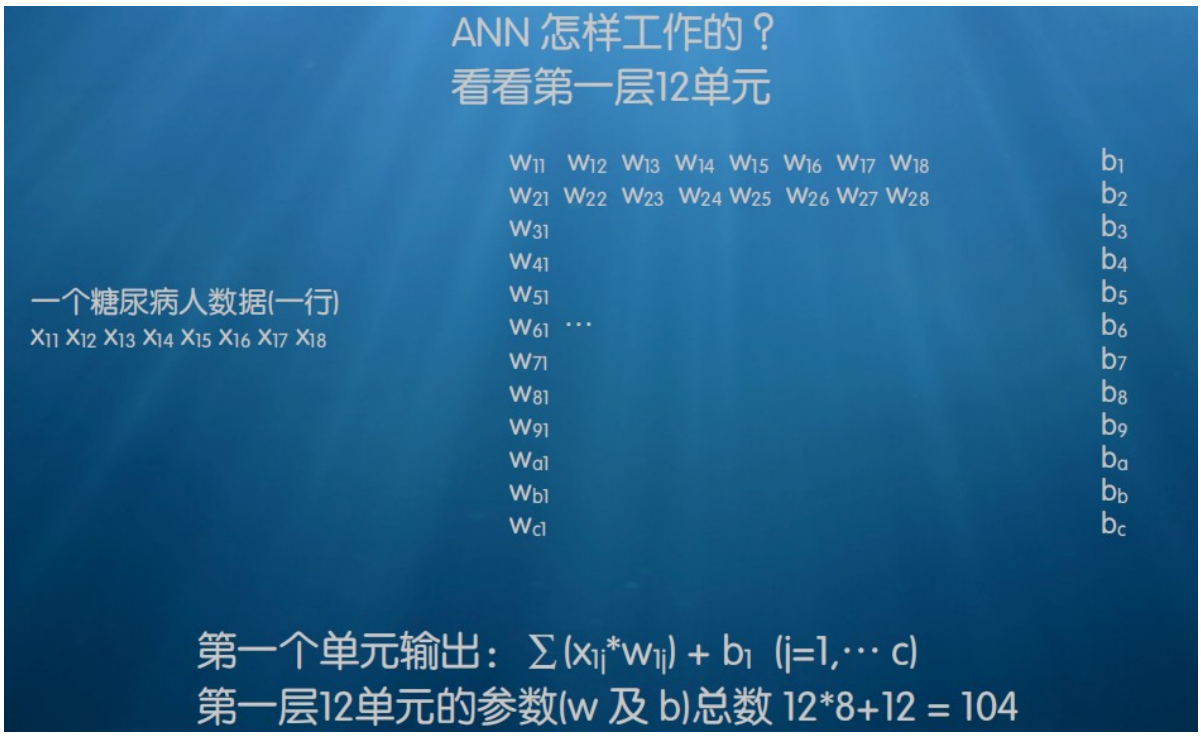


In [ ]:

```
#创建模型
model = Sequential()

#输入层
model.add(Dense(12, input_dim=8, activation='relu'))
#12个神经元和8个变量（数据有8个特征值）


#隐藏层

model.add(Dense(8,activation='relu'))


# 输出层是有一个神经元来预测它的类别
model.add(Dense(1, activation='sigmoid'))
model.summary()


adam = optimizers.Adam(lr=0.001)   # 默认 learning rate 0.001
loss = losses.binary_crossentropy

# 编译模型
# 需要接受三个参数
  # 优化器 optimizer。它可以是现有优化器的字符串标识符，如 rmsprop 或 adagrad，用的较多
  # 损失函数 loss，模型试图最小化的目标函数。它可以是现有损失函数的字符串标识符，
  # 多分类时为 categorical_crossentropy 或 mse，二分类时为 binary_crossentropy，也可以
  # 评估标准 metrics。对于任何分类问题，你都希望将其设置为 metrics = ['accuracy']。
  # 评估标准可以是现有的标准的字符串标识符，也可以是自定义的评估标准函数。

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 12)                108
_________________________________________________________________
dense_1 (Dense)              (None, 8)                 104
```

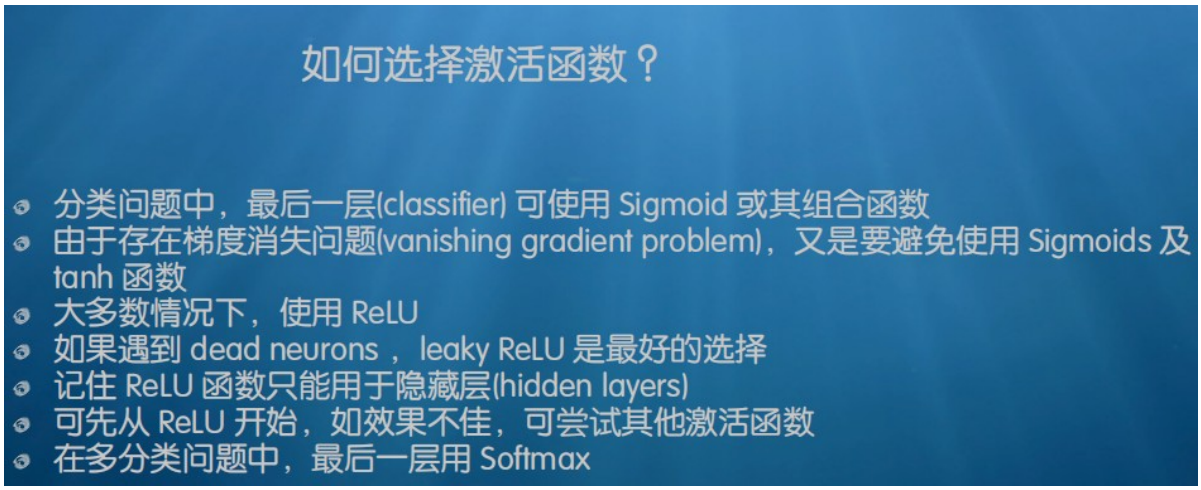
```
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 9         
=================================================================
Total params: 221
Trainable params: 221
Non-trainable params: 0
_________________________________________________________________
```



激活函数的作用是决定本神经元的输出是否传递到下一单元

使用激活函数的原因：

- 各层的输出wx+b是线性变换，当网络层数很大时，各神经元输出的总和会非常大，会引起严重的计算问题
- 激活函数最大的作用是将非线性引入ANN，这对分类问题很重要，0、1表明类别，0~1之间值表示了可能性
- 非线性激活函数作用于输入数据可使ANN学习和完成更复杂的任务
- 没有激活函数的模型就是一个线性回归模型，它使多层多单元的网络行为如同一个神经元



In [ ]:
```
history=model.fit(X_train, y_train, epochs=150, batch_size=32,validation_data=(X_test
```

```
Epoch 1/150
17/17 [==============================] - 0s 18ms/step - loss: 0.3149 - accuracy: 0.856
```

```
6 - val_loss: 0.6069 - val_accuracy: 0.7576
Epoch 2/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3145 - accuracy: 0.856
6 - val_loss: 0.6140 - val_accuracy: 0.7619
Epoch 3/150
17/17 [==============================] - 0s 12ms/step - loss: 0.3141 - accuracy: 0.858
5 - val_loss: 0.6142 - val_accuracy: 0.7576
Epoch 4/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3137 - accuracy: 0.860
3 - val_loss: 0.6119 - val_accuracy: 0.7662
Epoch 5/150
17/17 [==============================] - 0s 11ms/step - loss: 0.3135 - accuracy: 0.860
3 - val_loss: 0.6135 - val_accuracy: 0.7619
Epoch 6/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3131 - accuracy: 0.860
3 - val_loss: 0.6121 - val_accuracy: 0.7619
Epoch 7/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3134 - accuracy: 0.856
6 - val_loss: 0.6145 - val_accuracy: 0.7576
Epoch 8/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3129 - accuracy: 0.856
6 - val_loss: 0.6132 - val_accuracy: 0.7619
Epoch 9/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3133 - accuracy: 0.8603
- val_loss: 0.6145 - val_accuracy: 0.7619
Epoch 10/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3129 - accuracy: 0.8603
- val_loss: 0.6154 - val_accuracy: 0.7576
Epoch 11/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3120 - accuracy: 0.8566
- val_loss: 0.6110 - val_accuracy: 0.7662
Epoch 12/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3123 - accuracy: 0.8585
- val_loss: 0.6180 - val_accuracy: 0.7619
Epoch 13/150
17/17 [==============================] - 0s 11ms/step - loss: 0.3118 - accuracy: 0.858
5 - val_loss: 0.6155 - val_accuracy: 0.7576
Epoch 14/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3111 - accuracy: 0.860
3 - val_loss: 0.6160 - val_accuracy: 0.7619
Epoch 15/150
17/17 [==============================] - 0s 11ms/step - loss: 0.3112 - accuracy: 0.860
3 - val_loss: 0.6153 - val_accuracy: 0.7662
Epoch 16/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3108 - accuracy: 0.8603
- val_loss: 0.6194 - val_accuracy: 0.7662
Epoch 17/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3108 - accuracy: 0.858
5 - val_loss: 0.6178 - val_accuracy: 0.7619
Epoch 18/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3114 - accuracy: 0.8547
- val_loss: 0.6120 - val_accuracy: 0.7619
Epoch 19/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3101 - accuracy: 0.8585
- val_loss: 0.6203 - val_accuracy: 0.7619
Epoch 20/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3103 - accuracy: 0.8566
- val_loss: 0.6150 - val_accuracy: 0.7576
Epoch 21/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3098 - accuracy: 0.8585
- val_loss: 0.6194 - val_accuracy: 0.7706
Epoch 22/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3099 - accuracy: 0.8529
- val_loss: 0.6217 - val_accuracy: 0.7619
Epoch 23/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3092 - accuracy: 0.8566
- val_loss: 0.6147 - val_accuracy: 0.7532
Epoch 24/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3093 - accuracy: 0.8566
```

```
- val_loss: 0.6173 - val_accuracy: 0.7576
Epoch 25/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3091 - accuracy: 0.8585
- val_loss: 0.6196 - val_accuracy: 0.7662
Epoch 26/150
17/17 [==============================] - 0s 11ms/step - loss: 0.3087 - accuracy: 0.856
6 - val_loss: 0.6179 - val_accuracy: 0.7532
Epoch 27/150
17/17 [==============================] - 0s 11ms/step - loss: 0.3088 - accuracy: 0.854
7 - val_loss: 0.6154 - val_accuracy: 0.7619
Epoch 28/150
17/17 [==============================] - 0s 15ms/step - loss: 0.3089 - accuracy: 0.854
7 - val_loss: 0.6256 - val_accuracy: 0.7662
Epoch 29/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3087 - accuracy: 0.854
7 - val_loss: 0.6193 - val_accuracy: 0.7662
Epoch 30/150
17/17 [==============================] - 0s 12ms/step - loss: 0.3075 - accuracy: 0.858
5 - val_loss: 0.6231 - val_accuracy: 0.7662
Epoch 31/150
17/17 [==============================] - 0s 22ms/step - loss: 0.3075 - accuracy: 0.856
6 - val_loss: 0.6219 - val_accuracy: 0.7662
Epoch 32/150
17/17 [==============================] - 0s 14ms/step - loss: 0.3072 - accuracy: 0.854
7 - val_loss: 0.6245 - val_accuracy: 0.7619
Epoch 33/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3073 - accuracy: 0.8547
- val_loss: 0.6205 - val_accuracy: 0.7662
Epoch 34/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3071 - accuracy: 0.8566
- val_loss: 0.6196 - val_accuracy: 0.7706
Epoch 35/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3069 - accuracy: 0.8566
- val_loss: 0.6206 - val_accuracy: 0.7619
Epoch 36/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3067 - accuracy: 0.8547
- val_loss: 0.6247 - val_accuracy: 0.7662
Epoch 37/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3061 - accuracy: 0.8547
- val_loss: 0.6230 - val_accuracy: 0.7619
Epoch 38/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3066 - accuracy: 0.8585
- val_loss: 0.6233 - val_accuracy: 0.7619
Epoch 39/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3056 - accuracy: 0.8585
- val_loss: 0.6237 - val_accuracy: 0.7706
Epoch 40/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3057 - accuracy: 0.8585
- val_loss: 0.6236 - val_accuracy: 0.7749
Epoch 41/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3052 - accuracy: 0.8566
- val_loss: 0.6262 - val_accuracy: 0.7662
Epoch 42/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3054 - accuracy: 0.8547
- val_loss: 0.6302 - val_accuracy: 0.7662
Epoch 43/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3052 - accuracy: 0.8566
- val_loss: 0.6267 - val_accuracy: 0.7706
Epoch 44/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3047 - accuracy: 0.8566
- val_loss: 0.6269 - val_accuracy: 0.7706
Epoch 45/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3057 - accuracy: 0.8585
- val_loss: 0.6277 - val_accuracy: 0.7706
Epoch 46/150
17/17 [==============================] - 0s 16ms/step - loss: 0.3043 - accuracy: 0.854
7 - val_loss: 0.6299 - val_accuracy: 0.7662
Epoch 47/150
17/17 [==============================] - 0s 13ms/step - loss: 0.3046 - accuracy: 0.854
```

```
7 - val_loss: 0.6273 - val_accuracy: 0.7706
Epoch 48/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3039 - accuracy: 0.854
7 - val_loss: 0.6313 - val_accuracy: 0.7662
Epoch 49/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3052 - accuracy: 0.8603
- val_loss: 0.6262 - val_accuracy: 0.7619
Epoch 50/150
17/17 [==============================] - 0s 10ms/step - loss: 0.3036 - accuracy: 0.860
3 - val_loss: 0.6311 - val_accuracy: 0.7749
Epoch 51/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3035 - accuracy: 0.8585
- val_loss: 0.6325 - val_accuracy: 0.7662
Epoch 52/150
17/17 [==============================] - 0s 11ms/step - loss: 0.3036 - accuracy: 0.858
5 - val_loss: 0.6315 - val_accuracy: 0.7706
Epoch 53/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3045 - accuracy: 0.8603
- val_loss: 0.6279 - val_accuracy: 0.7749
Epoch 54/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3029 - accuracy: 0.8585
- val_loss: 0.6337 - val_accuracy: 0.7749
Epoch 55/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3029 - accuracy: 0.8547
- val_loss: 0.6337 - val_accuracy: 0.7749
Epoch 56/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3031 - accuracy: 0.8566
- val_loss: 0.6289 - val_accuracy: 0.7706
Epoch 57/150
17/17 [==============================] - 0s 7ms/step - loss: 0.3024 - accuracy: 0.8603
- val_loss: 0.6339 - val_accuracy: 0.7749
Epoch 58/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3025 - accuracy: 0.8603
- val_loss: 0.6364 - val_accuracy: 0.7706
Epoch 59/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3021 - accuracy: 0.8603
- val_loss: 0.6343 - val_accuracy: 0.7706
Epoch 60/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3023 - accuracy: 0.8603
- val_loss: 0.6355 - val_accuracy: 0.7749
Epoch 61/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3018 - accuracy: 0.8585
- val_loss: 0.6326 - val_accuracy: 0.7792
Epoch 62/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3015 - accuracy: 0.8603
- val_loss: 0.6379 - val_accuracy: 0.7749
Epoch 63/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3021 - accuracy: 0.8622
- val_loss: 0.6349 - val_accuracy: 0.7749
Epoch 64/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3013 - accuracy: 0.8622
- val_loss: 0.6382 - val_accuracy: 0.7792
Epoch 65/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3011 - accuracy: 0.8622
- val_loss: 0.6400 - val_accuracy: 0.7749
Epoch 66/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3012 - accuracy: 0.8603
- val_loss: 0.6358 - val_accuracy: 0.7792
Epoch 67/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3014 - accuracy: 0.8622
- val_loss: 0.6399 - val_accuracy: 0.7792
Epoch 68/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3017 - accuracy: 0.8603
- val_loss: 0.6317 - val_accuracy: 0.7749
Epoch 69/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3018 - accuracy: 0.8585
- val_loss: 0.6401 - val_accuracy: 0.7706
Epoch 70/150
17/17 [==============================] - 0s 7ms/step - loss: 0.3002 - accuracy: 0.8566
```

```
- val_loss: 0.6339 - val_accuracy: 0.7835
Epoch 71/150
17/17 [==============================] - 0s 7ms/step - loss: 0.3002 - accuracy: 0.8641
- val_loss: 0.6359 - val_accuracy: 0.7792
Epoch 72/150
17/17 [==============================] - 0s 9ms/step - loss: 0.3000 - accuracy: 0.8641
- val_loss: 0.6381 - val_accuracy: 0.7792
Epoch 73/150
17/17 [==============================] - 0s 8ms/step - loss: 0.3002 - accuracy: 0.8641
- val_loss: 0.6406 - val_accuracy: 0.7792
Epoch 74/150
17/17 [==============================] - 0s 7ms/step - loss: 0.2996 - accuracy: 0.8622
- val_loss: 0.6388 - val_accuracy: 0.7792
Epoch 75/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2994 - accuracy: 0.864
1 - val_loss: 0.6391 - val_accuracy: 0.7835
Epoch 76/150
17/17 [==============================] - 0s 7ms/step - loss: 0.3005 - accuracy: 0.8622
- val_loss: 0.6409 - val_accuracy: 0.7792
Epoch 77/150
17/17 [==============================] - 0s 7ms/step - loss: 0.2991 - accuracy: 0.8659
- val_loss: 0.6381 - val_accuracy: 0.7749
Epoch 78/150
17/17 [==============================] - 0s 13ms/step - loss: 0.2992 - accuracy: 0.864
1 - val_loss: 0.6419 - val_accuracy: 0.7792
Epoch 79/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2988 - accuracy: 0.8641
- val_loss: 0.6379 - val_accuracy: 0.7792
Epoch 80/150
17/17 [==============================] - 0s 12ms/step - loss: 0.2988 - accuracy: 0.862
2 - val_loss: 0.6424 - val_accuracy: 0.7792
Epoch 81/150
17/17 [==============================] - 0s 11ms/step - loss: 0.2993 - accuracy: 0.862
2 - val_loss: 0.6432 - val_accuracy: 0.7792
Epoch 82/150
17/17 [==============================] - 0s 11ms/step - loss: 0.2986 - accuracy: 0.864
1 - val_loss: 0.6420 - val_accuracy: 0.7792
Epoch 83/150
17/17 [==============================] - 0s 11ms/step - loss: 0.2986 - accuracy: 0.862
2 - val_loss: 0.6414 - val_accuracy: 0.7835
Epoch 84/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2982 - accuracy: 0.8659
- val_loss: 0.6408 - val_accuracy: 0.7792
Epoch 85/150
17/17 [==============================] - 0s 12ms/step - loss: 0.2984 - accuracy: 0.864
1 - val_loss: 0.6461 - val_accuracy: 0.7792
Epoch 86/150
17/17 [==============================] - 0s 11ms/step - loss: 0.2973 - accuracy: 0.867
8 - val_loss: 0.6445 - val_accuracy: 0.7792
Epoch 87/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2971 - accuracy: 0.8641
- val_loss: 0.6435 - val_accuracy: 0.7835
Epoch 88/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2968 - accuracy: 0.8659
- val_loss: 0.6450 - val_accuracy: 0.7835
Epoch 89/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2968 - accuracy: 0.8678
- val_loss: 0.6452 - val_accuracy: 0.7792
Epoch 90/150
17/17 [==============================] - 0s 12ms/step - loss: 0.2966 - accuracy: 0.867
8 - val_loss: 0.6435 - val_accuracy: 0.7879
Epoch 91/150
17/17 [==============================] - 0s 11ms/step - loss: 0.2966 - accuracy: 0.865
9 - val_loss: 0.6486 - val_accuracy: 0.7792
Epoch 92/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2958 - accuracy: 0.864
1 - val_loss: 0.6469 - val_accuracy: 0.7792
Epoch 93/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2958 - accuracy: 0.8659
```

```
- val_loss: 0.6416 - val_accuracy: 0.7835
Epoch 94/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2950 - accuracy: 0.8678
- val_loss: 0.6445 - val_accuracy: 0.7879
Epoch 95/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2958 - accuracy: 0.8641
- val_loss: 0.6452 - val_accuracy: 0.7835
Epoch 96/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2951 - accuracy: 0.8641
- val_loss: 0.6454 - val_accuracy: 0.7879
Epoch 97/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2947 - accuracy: 0.8641
- val_loss: 0.6464 - val_accuracy: 0.7835
Epoch 98/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2943 - accuracy: 0.8678
- val_loss: 0.6486 - val_accuracy: 0.7835
Epoch 99/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2942 - accuracy: 0.8659
- val_loss: 0.6469 - val_accuracy: 0.7792
Epoch 100/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2942 - accuracy: 0.8659
- val_loss: 0.6464 - val_accuracy: 0.7879
Epoch 101/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2935 - accuracy: 0.8678
- val_loss: 0.6472 - val_accuracy: 0.7835
Epoch 102/150
17/17 [==============================] - 0s 12ms/step - loss: 0.2945 - accuracy: 0.865
9 - val_loss: 0.6498 - val_accuracy: 0.7792
Epoch 103/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2932 - accuracy: 0.865
9 - val_loss: 0.6466 - val_accuracy: 0.7879
Epoch 104/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2936 - accuracy: 0.8678
- val_loss: 0.6452 - val_accuracy: 0.7922
Epoch 105/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2931 - accuracy: 0.8659
- val_loss: 0.6513 - val_accuracy: 0.7792
Epoch 106/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2926 - accuracy: 0.8641
- val_loss: 0.6543 - val_accuracy: 0.7792
Epoch 107/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2924 - accuracy: 0.8678
- val_loss: 0.6524 - val_accuracy: 0.7792
Epoch 108/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2925 - accuracy: 0.8659
- val_loss: 0.6542 - val_accuracy: 0.7792
Epoch 109/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2920 - accuracy: 0.865
9 - val_loss: 0.6517 - val_accuracy: 0.7835
Epoch 110/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2917 - accuracy: 0.8659
- val_loss: 0.6527 - val_accuracy: 0.7792
Epoch 111/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2921 - accuracy: 0.865
9 - val_loss: 0.6489 - val_accuracy: 0.7835
Epoch 112/150
17/17 [==============================] - 0s 13ms/step - loss: 0.2915 - accuracy: 0.864
1 - val_loss: 0.6530 - val_accuracy: 0.7835
Epoch 113/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2914 - accuracy: 0.8641
- val_loss: 0.6546 - val_accuracy: 0.7835
Epoch 114/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2905 - accuracy: 0.8715
- val_loss: 0.6555 - val_accuracy: 0.7792
Epoch 115/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2911 - accuracy: 0.871
5 - val_loss: 0.6575 - val_accuracy: 0.7792
Epoch 116/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2910 - accuracy: 0.864
```

```
1 - val_loss: 0.6598 - val_accuracy: 0.7792
Epoch 117/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2898 - accuracy: 0.867
8 - val_loss: 0.6516 - val_accuracy: 0.7792
Epoch 118/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2897 - accuracy: 0.8696
- val_loss: 0.6540 - val_accuracy: 0.7835
Epoch 119/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2903 - accuracy: 0.8696
- val_loss: 0.6548 - val_accuracy: 0.7879
Epoch 120/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2893 - accuracy: 0.8678
- val_loss: 0.6589 - val_accuracy: 0.7792
Epoch 121/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2896 - accuracy: 0.8696
- val_loss: 0.6573 - val_accuracy: 0.7879
Epoch 122/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2887 - accuracy: 0.8696
- val_loss: 0.6586 - val_accuracy: 0.7749
Epoch 123/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2900 - accuracy: 0.8678
- val_loss: 0.6571 - val_accuracy: 0.7879
Epoch 124/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2889 - accuracy: 0.8715
- val_loss: 0.6593 - val_accuracy: 0.7835
Epoch 125/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2889 - accuracy: 0.8696
- val_loss: 0.6605 - val_accuracy: 0.7749
Epoch 126/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2884 - accuracy: 0.8696
- val_loss: 0.6597 - val_accuracy: 0.7835
Epoch 127/150
17/17 [==============================] - 0s 14ms/step - loss: 0.2882 - accuracy: 0.869
6 - val_loss: 0.6611 - val_accuracy: 0.7835
Epoch 128/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2891 - accuracy: 0.865
9 - val_loss: 0.6621 - val_accuracy: 0.7749
Epoch 129/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2896 - accuracy: 0.869
6 - val_loss: 0.6506 - val_accuracy: 0.7792
Epoch 130/150
17/17 [==============================] - 0s 12ms/step - loss: 0.2876 - accuracy: 0.873
4 - val_loss: 0.6605 - val_accuracy: 0.7835
Epoch 131/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2875 - accuracy: 0.8696
- val_loss: 0.6608 - val_accuracy: 0.7835
Epoch 132/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2870 - accuracy: 0.8715
- val_loss: 0.6618 - val_accuracy: 0.7835
Epoch 133/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2865 - accuracy: 0.8715
- val_loss: 0.6635 - val_accuracy: 0.7792
Epoch 134/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2867 - accuracy: 0.8678
- val_loss: 0.6581 - val_accuracy: 0.7835
Epoch 135/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2858 - accuracy: 0.8696
- val_loss: 0.6635 - val_accuracy: 0.7879
Epoch 136/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2861 - accuracy: 0.8678
- val_loss: 0.6644 - val_accuracy: 0.7879
Epoch 137/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2860 - accuracy: 0.8659
- val_loss: 0.6630 - val_accuracy: 0.7879
Epoch 138/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2862 - accuracy: 0.8696
- val_loss: 0.6647 - val_accuracy: 0.7879
Epoch 139/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2855 - accuracy: 0.8696
```

```
- val_loss: 0.6612 - val_accuracy: 0.7792
Epoch 140/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2849 - accuracy: 0.8696
- val_loss: 0.6656 - val_accuracy: 0.7879
Epoch 141/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2857 - accuracy: 0.8659
- val_loss: 0.6673 - val_accuracy: 0.7792
Epoch 142/150
17/17 [==============================] - 0s 7ms/step - loss: 0.2845 - accuracy: 0.8659
- val_loss: 0.6663 - val_accuracy: 0.7792
Epoch 143/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2845 - accuracy: 0.8678
- val_loss: 0.6703 - val_accuracy: 0.7879
Epoch 144/150
17/17 [==============================] - 0s 7ms/step - loss: 0.2842 - accuracy: 0.8696
- val_loss: 0.6643 - val_accuracy: 0.7835
Epoch 145/150
17/17 [==============================] - 0s 10ms/step - loss: 0.2841 - accuracy: 0.871
5 - val_loss: 0.6647 - val_accuracy: 0.7835
Epoch 146/150
17/17 [==============================] - 0s 9ms/step - loss: 0.2839 - accuracy: 0.8715
- val_loss: 0.6640 - val_accuracy: 0.7835
Epoch 147/150
17/17 [==============================] - 0s 7ms/step - loss: 0.2845 - accuracy: 0.8696
- val_loss: 0.6654 - val_accuracy: 0.7792
Epoch 148/150
17/17 [==============================] - 0s 12ms/step - loss: 0.2839 - accuracy: 0.869
6 - val_loss: 0.6694 - val_accuracy: 0.7835
Epoch 149/150
17/17 [==============================] - 0s 8ms/step - loss: 0.2834 - accuracy: 0.8659
- val_loss: 0.6696 - val_accuracy: 0.7835
Epoch 150/150
17/17 [==============================] - 0s 11ms/step - loss: 0.2835 - accuracy: 0.869
6 - val_loss: 0.6723 - val_accuracy: 0.7879
```

In [ ]:

```
y_pred_ann=model.predict_classes(X_test)
accuracy_ann = accuracy_score(y_test,y_pred_ann)
print('{:s} : {:.2f}'.format('ANN的准确率为:', accuracy_ann))
```

```
ANN的准确率为: : 0.79
```

In [ ]:

```
from sklearn.metrics import confusion_matrix
m_ann = confusion_matrix(y_test,y_pred_ann)
print('预测结果混淆矩阵: \n',m_ann)
```

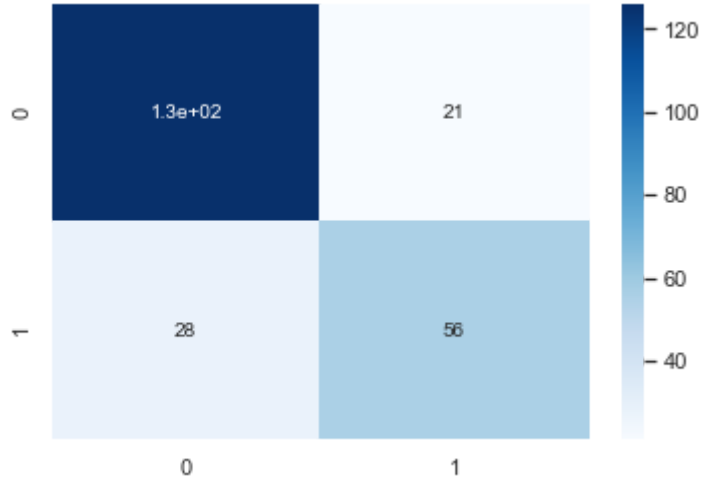
```
预测结果混淆矩阵:
 [[126  21]
 [ 28  56]]
```

In [ ]:

```
%pylab inline
import seaborn as sns
sns.heatmap(m_ann,cmap='Blues',annot=True)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[ ]: <AxesSubplot:>

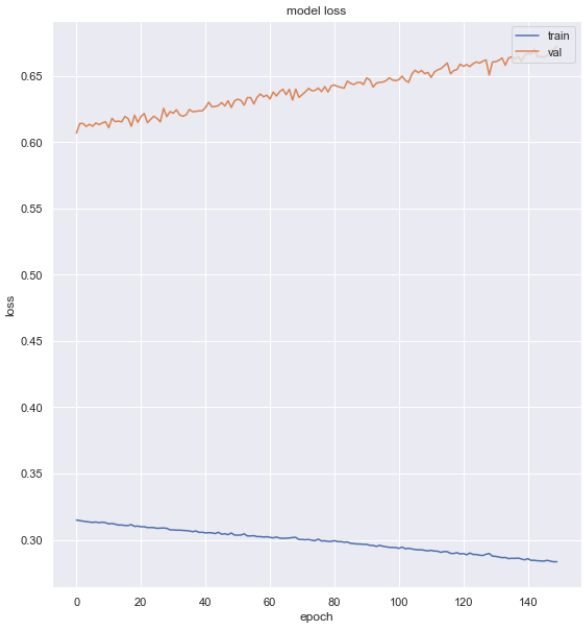


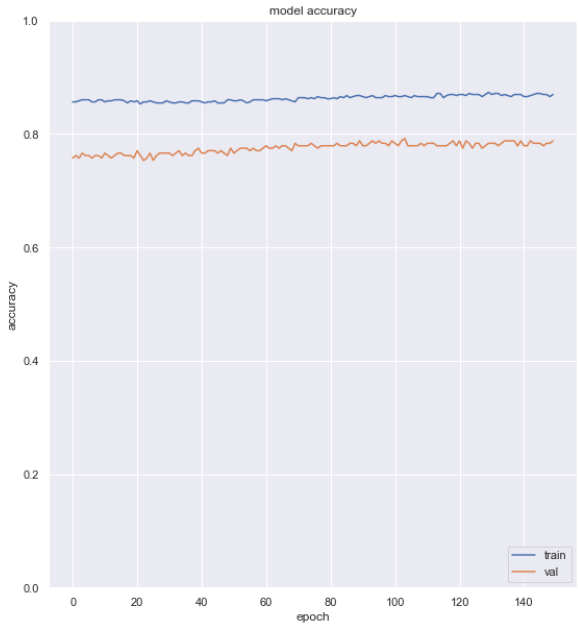
分类评估报告

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_ann))
```

```
              precision    recall  f1-score   support

           0       0.82      0.86      0.84       147
           1       0.73      0.67      0.70        84

    accuracy                           0.79       231
   macro avg       0.77      0.76      0.77       231
weighted avg       0.79      0.79      0.79       231
```

In [ ]:
```
# summarize history for accuracy or loss
plt.figure(figsize=(20,10))
plt.subplot(121)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
#plt.ylim(0.0,6.0)
plt.legend(['train','val'], loc='upper right')

plt.subplot(122)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.ylim(0.0,1.0)
plt.legend(['train','val'], loc='lower right')
plt.show()
```





使用ROC-AUC进行评估

ROC曲线与AUC指标

- TPR = TP / (TP + FN)
  - 所有真实类别为1的样本中，预测类别为1的比例
- FPR = FP / (FP + FN)
  - 所有真实类别为0的样本中，预测类别为1的比例

In [ ]:

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve,auc
plt.figure(figsize=[20,10])
for clf,title in zip([dt,gnb,rf,lr],['DecisionTree','GaussianNB','RandomForest','Logi
    probas_ = clf.fit(X_train,y_train).predict_proba(X_test)
    fpr,tpr,threshold = roc_curve(y_test,probas_[:,1])
    plt.plot(fpr,tpr,label='%s - AUC:%.2f'%(title,auc(fpr,tpr)))
plt.plot([0,1],[0,1],'k--')
plt.xlim([0.0,1.0])
plt.ylim([0.0,1.0])
plt.xlabel('False Positive Rate',fontsize=20)
plt.ylabel('True Positive Rate',fontsize=20)
plt.title('Receiver Operating Characteristic example',fontsize=20)
plt.legend(loc='lower right',fontsize=20)
plt.show()
```

ROC曲线的横轴就是FPRate，纵轴就是TPRate，当二者相等时，表示的意义则是：对于不论真实类别是1还是0的样本，分类器预测为1的概率是相等的，此时AUC为0.5

- AUC的概率意义是随机取一对正负样本，正样本得分大于负样本的概率
- AUC的最小值为0.5，最大值为1，取值越高越好
- AUC=1，完美分类器，采用这个预测模型时，不管设定什么阈值都能得出完美预测。绝大多数预测的场合，不存在完美分类器。
- 0.5<AUC<1，优于随机猜测。这个分类器（模型）妥善设定阈值的话，能有预测价值。

> 最终AUC的范围在[0.5, 1]之间，并且越接近1越好

- AUC只能用来评价二分类
- AUC非常适合评价样本不平衡中的分类器性能

评估特征重要性

列出各个特征的重要性

In [ ]:
```
rf.feature_importances_
```

Out[ ]:
```
array([0.08835209, 0.25937727, 0.07612149, 0.07124658, 0.08186008,
       0.162648  , 0.12920395, 0.13119053])
```

特征重要性从小到大返回其位置信息,并倒序从大到小输出

In [ ]:
```
rf.feature_importances_.argsort()[::-1]
```

Out[ ]:
```
array([1, 5, 7, 6, 0, 4, 2, 3], dtype=int64)
```

In [ ]:
```
diabetes_data_copy.columns[rf.feature_importances_.argsort()[::-1]]
```

Out[ ]:
```
Index(['glucose', 'bmi', 'age', 'dpf', 'pregnancies', 'insulin', 'diastolic',
       'triceps'],
      dtype='object')
```

特征的重要性可视化

In [ ]:
```
import matplotlib.pyplot as plt
importance = rf.feature_importances_
names = diabetes_data_copy.columns
plt.title('Feature Importance')
plt.bar(range(1,len(names)),importance[importance.argsort()[::-1]])
plt.xticks(range(1,len(names)),names[importance.argsort()[::-1]],rotation=90)
plt.show()
```