Decision Trees, Ensemble Methods and Hyper-parameter tuning

Dr Rob Collins

Version 3, 23rd August 2023

Introduction

This tutorial starts with a treatment of 'Decision Trees'. Decision Trees are an example of a supervised-learning, classification algorithm.

Decision Trees have a number of advantages and disadvantages. In terms of advantages, the are easy to understand and are 'explainable': A human being reviewing the model can understand how decisions are being made. The core algorithm is rather simple to explain and write: In class we use a spread-sheet implementation which demonstrates the algorithm step-by-step. Finally, the algorithm not only operates on numerical data - but can also be programmed to operate on category data directly (having said that, the specific library function we will be using from sklearn is restricted to numerical data only. But this is a limitation of the library function - not of the fundamental algorithm).

In terms of disadvantages, Decision Tree models can grow rather large. They also have a tendency to 'over-fit' data. That is, whilst the fit a given data-set they may not generalise so well.

To deal with the disadvantage of the tendency of decision trees to over-fit, we will look at a number of methods for dealing with that.

In this workshop we will work with a data-set relating to Diabetes. The data-set contains a number of descriptive features for individuals and a single output variable (label) that indicate if they acquired Diabetes. We will use that data to build a Decision-Tree model of the type that might be used to help predict the onset of Diabetes for individuals.

Instructions for Students

This workbook includes some empty code-blocks. In those cases you are required to create the code for the block based on your previous learning on this course. This activity is indicated in the workbook below with a '**Student Task**' indicator.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

1 Loading and reviewing the data-set

As always we start by loading key library functions:

In [1]:
```
import numpy as np
import pandas as pd
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.model_selection import GridSearchCV
from sklearn.tree import plot_tree
```

.. and loading the data-set:

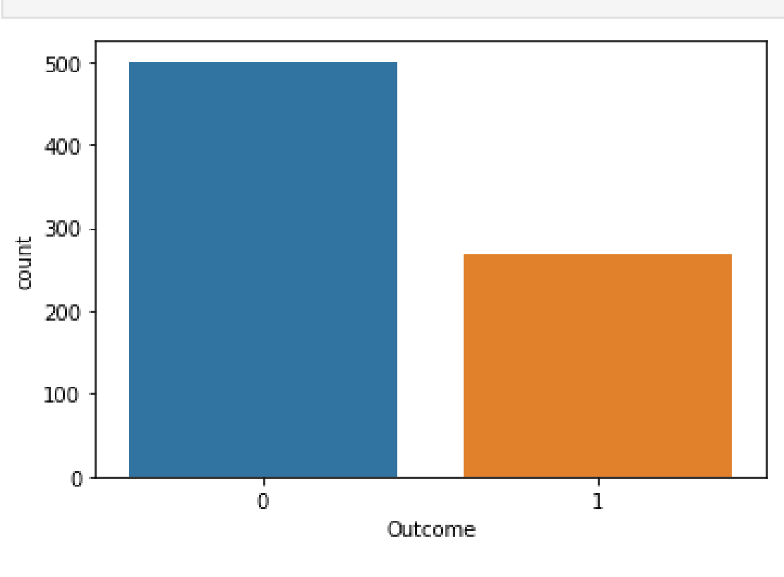
In [2]:
```
df = pd.read_csv("diabetes.csv")
df.head()
```

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
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

In this data-set the label is the 'Outome' feature. This contains '1' if the person developed diabetes and '0' otherwise. There are a smaller number of people who developed diabetes than people who are well (as shown by the following chart). But still, there are sufficient number of positive cases to build a reasonably predictive model.

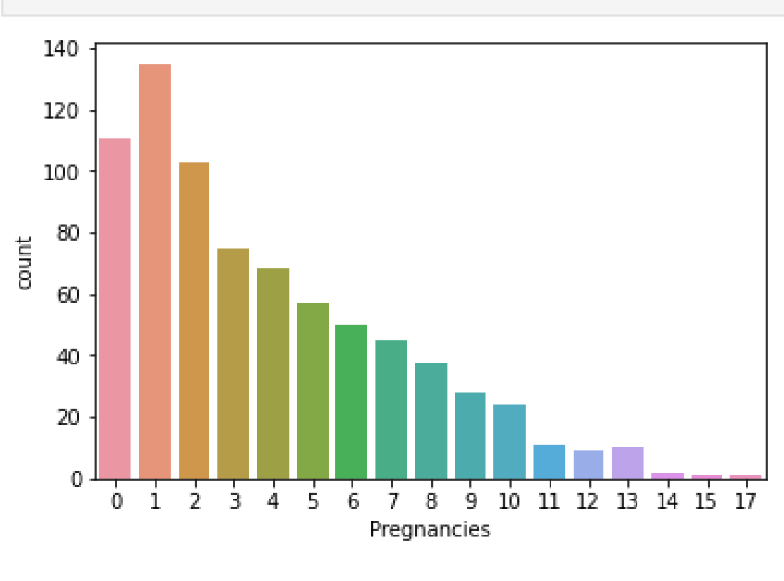
In [3]:
```
sns.countplot(x = "Outcome", data = df);
```



We can review other features in the same way.

Student task : Substitute various features from the data-set into the following function. That is, replace 'Pregnancies' with 'Age' etc. Think about which of these you find useful charts and which you do not.

In [4]:
```
sns.countplot(x = "Pregnancies", data = df);
```



As in previous workshops we can obtain an overall 'statistical' view of the data using the Pandas 'describe' function:

In [5]:
```
df.describe()
```

Out[5]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

On reviewing this data we can see that certain values seem to be missing. For example the lower 'Blood Pressure' measurement is zero! There are also zero values for 'SkinThickness', 'Insulin' and 'BMI' - none of which can be correct.

We will fix this as a two stage process .. first replacing all of the zero values with 'NaN' values:

In [6]:
```
df[['Glucose','BloodPressure','SkinThickness','Insulin','BMI','DiabetesPedigreeFunction','Age']] = df[['Glucose
df.head()
```

Out[6]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148.0	72.0	35.0	NaN	33.6	0.627	50	1
1	1	85.0	66.0	29.0	NaN	26.6	0.351	31	0
2	8	183.0	64.0	NaN	NaN	23.3	0.672	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.167	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.288	33	1

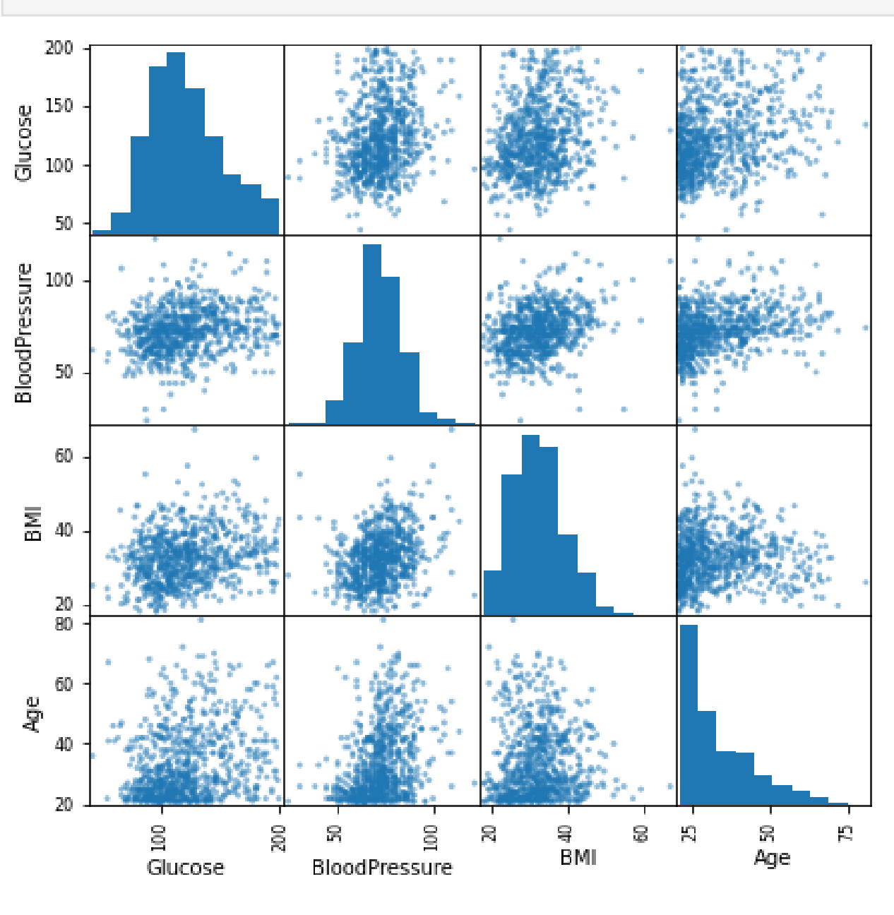
This means that we can use the easy, built-in Pandas function to replace NaN values with imputed Mean values:

In [7]:
```
df.fillna(df.mean(), inplace = True) #Filled Mising values with Mean
df.isnull().sum()
```

Out[7]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

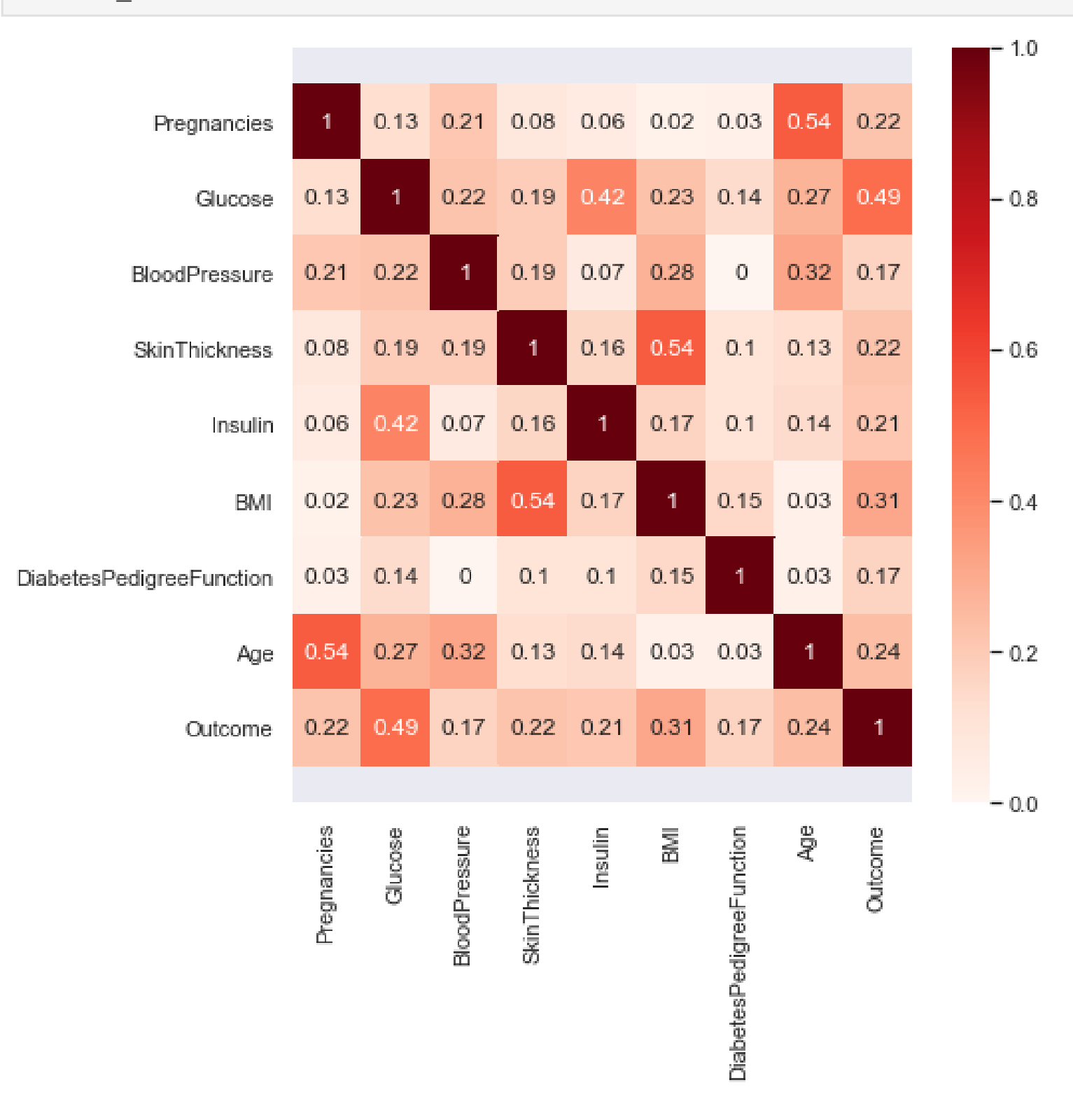
We have introduced various other ways of visually reviewing data in order to identify important features, including 'scatter_matrix':

In [8]:
```
scatter_matrix(df[['Glucose', 'BloodPressure','BMI', 'Age']], figsize=(7, 7));
```



.. and correlation matricies:

In [9]:
```
correlation_matrix = np.absolute(df.corr().round(2))
sns.set(rc={'figure.figsize':(7,7)})
ax = sns.heatmap(correlation_matrix, annot=True, cmap='Reds')
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5);
```



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
Pregnancies	1	0.13	0.21	0.08	0.06	0.02	0.03	0.54	0.22
Glucose	0.13	1	0.22	0.19	0.42	0.23	0.14	0.27	0.49
BloodPressure	0.21	0.22	1	0.19	0.07	0.28	0	0.32	0.17
SkinThickness	0.08	0.19	0.19	1	0.16	0.54	0.1	0.13	0.22
Insulin	0.06	0.42	0.07	0.16	1	0.17	0.1	0.14	0.21
BMI	0.02	0.23	0.28	0.54	0.17	1	0.15	0.03	0.31
DiabetesPedigreeFunction	0.03	0.14	0	0.1	0.1	0.15	1	0.03	0.17
Age	0.54	0.27	0.32	0.13	0.14	0.03	0.03	1	0.24
Outcome	0.22	0.49	0.17	0.22	0.21	0.31	0.17	0.24	1

2 Building the Model

2.1 Decision Tree

The first model we will create is a basic 'Decision Tree' using the sklearn library.

Import the required libraries:

In [10]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
```

Split the data into inputs (x's) and label (y):

In [11]:
```
x = df.loc[:,'Pregnancies':'Age']
y = df['Outcome']
```

Create the model object:

In [12]:
```
# Step 1
model = DecisionTreeClassifier(random_state=0, max_depth = 3)
```

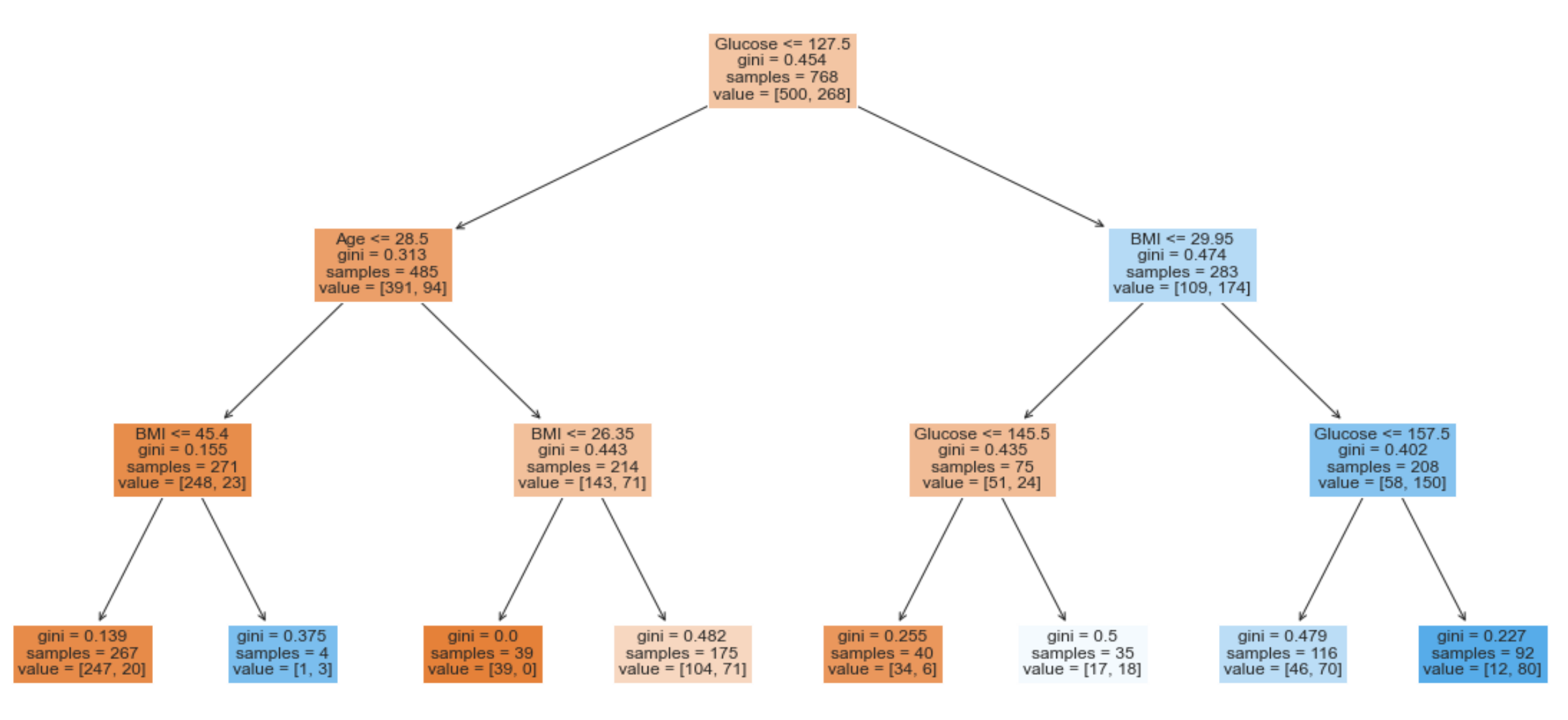
Fit the data using the model:

In [13]:
```
model.fit(x, y)
```

Out[13]:
```
DecisionTreeClassifier(max_depth=3, random_state=0)
```

Displaying the Tree as a Graphic

In [14]:
```
plt.figure(figsize=(20, 10))  # Adjust the figure size if needed
plot_tree(model, filled=True, fontsize=12, feature_names = df.columns)  # Adjust the fontsize as desired
plt.show()
```



Student task : Experiment by changing the 'max_depth = 3' to some other values.

Note: If the max depth gets too large then the graphic may too large to display. Also, it may take a long time to generate the graphic.

2.2 Pruning decision trees

The 'max_depth' parameter provides one method for preventing over-fitting in Decision Trees. It simply prevents the tree from growing below a certain depth. It divides the data set to a certain degree, choosing the most effective decision at each point .. but when the tree gets to a defined depth it simply stops.

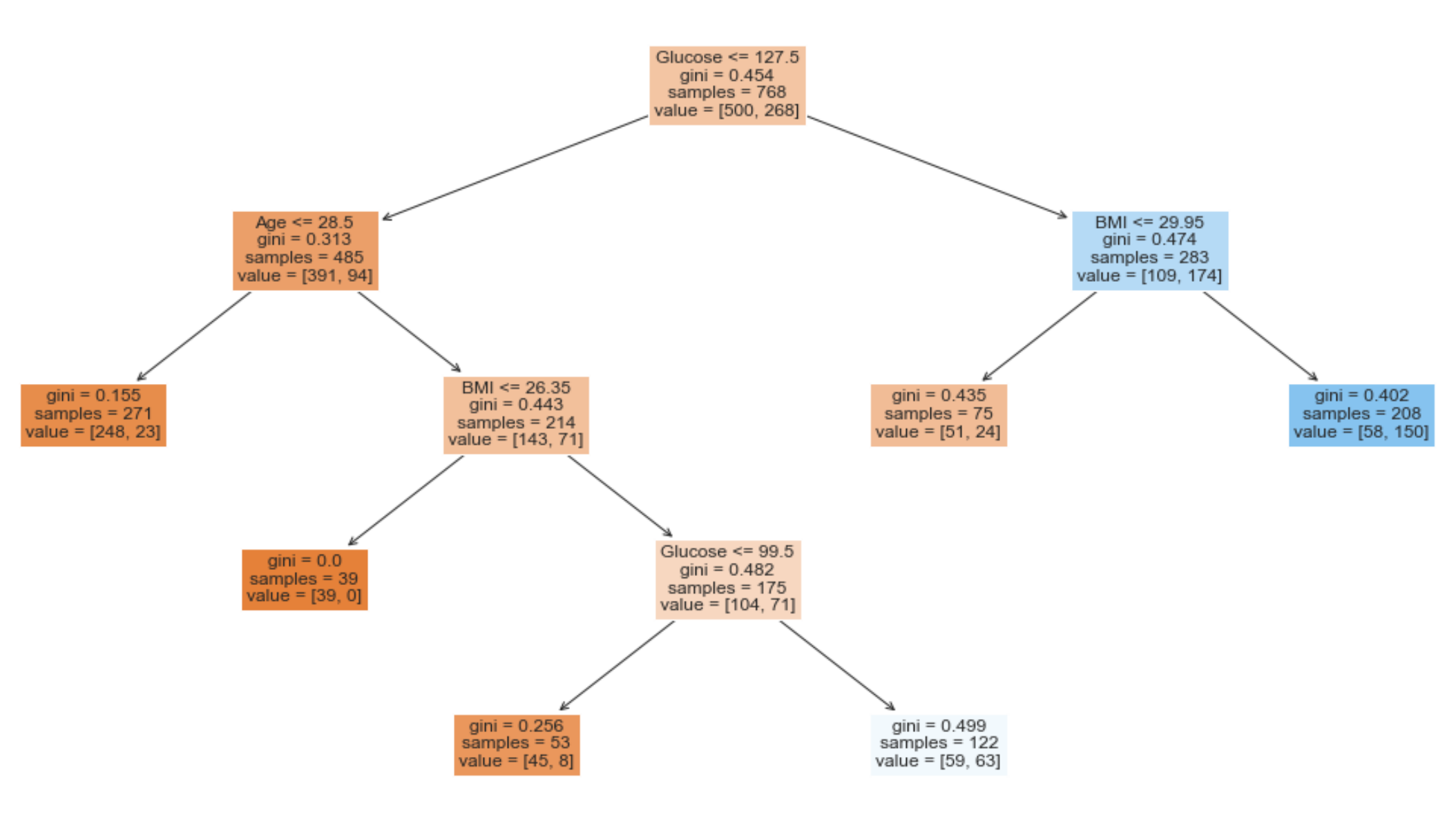
A more sophisticated way of preventing over-fitting is to 'prune' the tree. This means that nodes (branches) on tree are successively removed after the tree is built. The nodes are removed which causes the smallest loss of information and decision making power.

Describing the pruning algorithm in detail is beyond the scope of this course. However, if you are interested in this, after the workshop you may wish to review: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html#sphx-glr-auto-examples-tree-plot-cost-complexity-pruning-py

I the following block, the 'ccp_alpha' parameter controls the amount of pruning. Most of the code below is copied from code blocks above. The only significant difference in the inclusion of the 'ccp_alpha' parameter to DecisionTreeClassifier' and the larger value that max_depth has been set to.

In [24]:
```
#step 2
model = DecisionTreeClassifier(random_state=0, max_depth = 6, ccp_alpha = 0.01)
model.fit(x, y)
export_graphviz(model, out_file='diabetes_tree.dot',
                feature_names = x.columns,
                class_names = ['Diabetes','NOT diabetes'],
                rounded = True, proportion = False,
                precision = 2, filled = True)

plt.figure(figsize=(20, 10))  # Adjust the figure size if needed
plot_tree(model, filled=True, fontsize=12, feature_names = df.columns)  # Adjust the fontsize as desired
plt.show()
```



You may see that the tree is no longer a simple binary tree. The tree is deeper on one side as compared with the other. This is simply because 'weak' nodes have been removed in the pruning.

Student task : In the code block labelled "#Step 2" above, experiment with different values for 'max_depth' and 'ccp_alpha'.

3 Random Forrest Classifier

One way of extending the basic Decision Tree model is to build a 'forrest'. That is, build a collection of Decision Trees, each based on a slightly different sub-set of the data.

In this section, we will build just such a 'Random Forrest Classifier' and then compares its performance with the basic Decision Tree.

In order to do this, we need to split our data-set into a training and a test set:

In [16]:
```
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.2)
```

3.1 Random Forrest Classifier Performance

Then we can build and score our Random Forrest Classifier

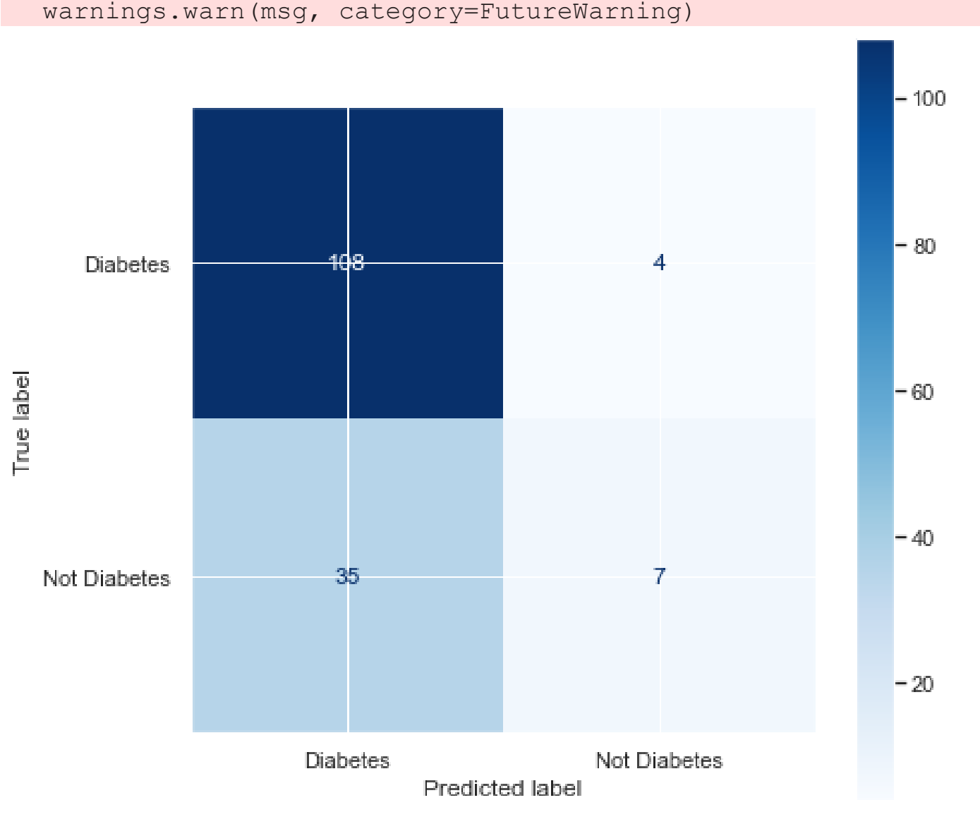
In [17]:
```
rfc = RandomForestClassifier()
rfc.fit(x_train,y_train)
rfc.score(x_test,y_test)
```

Out[17]:
```
0.7727272727272727
```

Or for a better view of performance, plot the whole Confusion Matrix:

In [26]:
```
plot_confusion_matrix(rfc, x_test, y_test,display_labels=['Diabetes','Not Diabetes'], cmap=plt.cm.Blues,)
plt.show()
```

```
C:\Anaconda3\envs\ML_Course_2023\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function plo
t_confusion_matrix is deprecated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be removed in
1.2. Use one of the class methods: ConfusionMatrixDisplay.from_predictions or ConfusionMatrixDisplay.from_estim
ator.
  warnings.warn(msg, category=FutureWarning)
```



True label	Diabetes	Not Diabetes
Diabetes	108	4
Not Diabetes	35	7

3.2 Decision Tree performance

We can compare that performance with the performance of the basic Decision Tree:

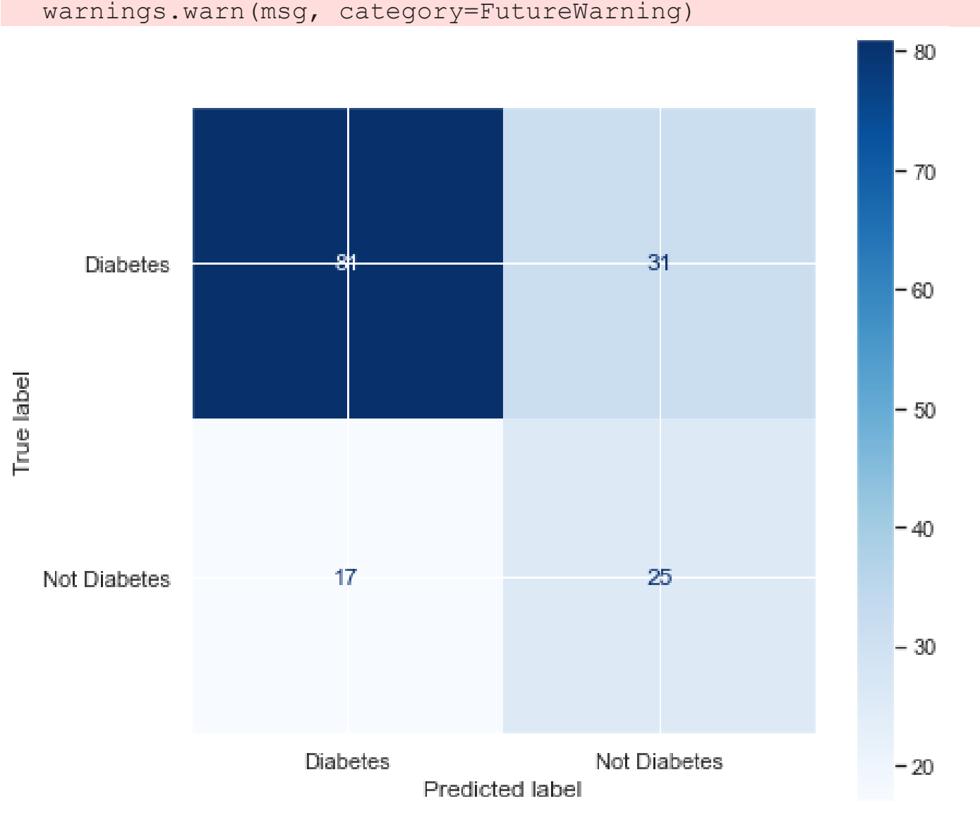
In [19]:
```
dtc  = DecisionTreeClassifier(random_state=0)
dtc.fit(x_train, y_train)
dtc.score(x_test, y_test)
```

Out[19]:
```
0.6883116883116883
```

And again, plot the confusion matrix.

In [20]:
```
plot_confusion_matrix(dtc, x_test, y_test,display_labels=['Diabetes','Not Diabetes'], cmap=plt.cm.Blues,)
plt.show()
```

```
C:\Anaconda3\envs\ML_Course_2023\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function plo
t_confusion_matrix is deprecated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be removed in
1.2. Use one of the class methods: ConfusionMatrixDisplay.from_predictions or ConfusionMatrixDisplay.from_estim
ator.
  warnings.warn(msg, category=FutureWarning)
```



True label	Diabetes	Not Diabetes
Diabetes	81	31
Not Diabetes	17	25

You can see that in this instance the performance of the Random Forrest Classifier is rather better than that of the basic Decision Tree.

4 Hyper-parameter Tuning

Many Machine Learning model algorithms have a large set of 'Hyper-parameters'. That is, there are setting that change the way that the model is built. You saw an example of this in the description of the Decision Tree Classifier above. In that case the hyper-parameter 'ccp_alpha' controlled the amount of pruning.

An indicator of the number of hyper-parameters available for a model can be obtained using the 'get_params()' function which is available for some sklearn algorithms. For example, for our Random Forrest Classifier:

In [21]:
```
print(rfc.get_params())
```

```
{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_featur
es': 'auto', 'max_leaf_nodes': None, 'max_samples': None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1,
'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 100, 'n_jobs': None, 'oob_score': Fals
e, 'random_state': None, 'verbose': 0, 'warm_start': False}
```

4.1 Manually searching for the best hyper-parameters

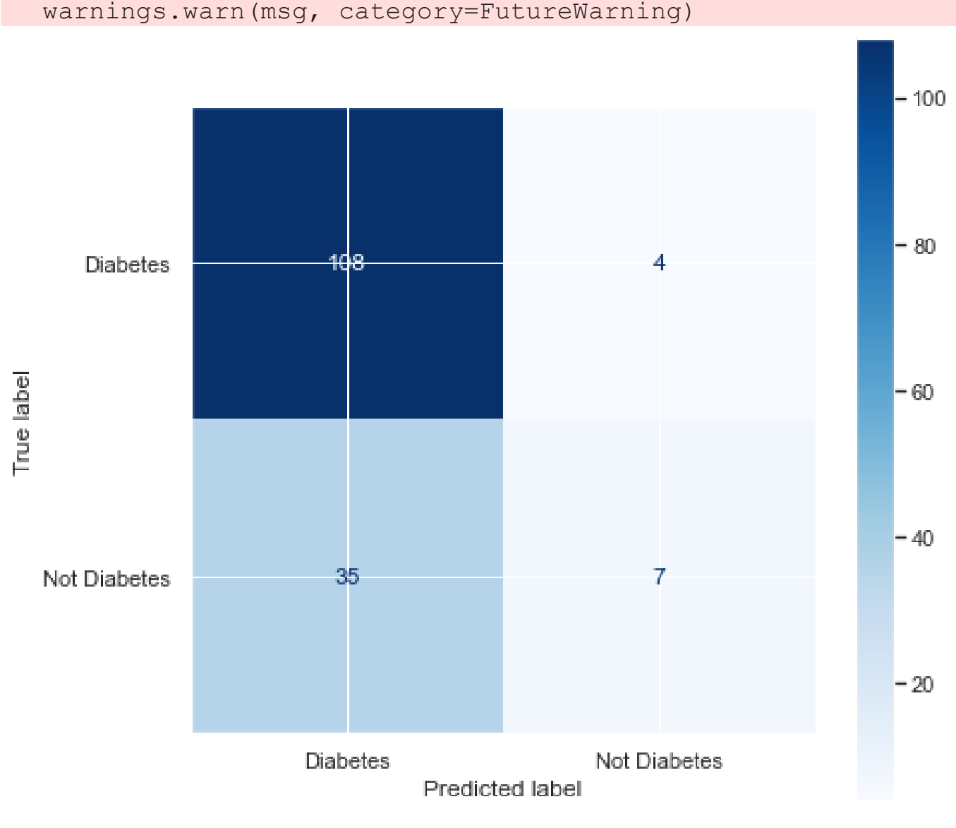
In the following block there another version of the RandomForrestClassifier, with some of the hyper-parameters set.

Student task : Experiment with various settings of the Hyper-parameters of 'RandomForrestClassifier' below to see if you can produce a better model. Typical values for the hyper-parameters are as follows:

- ccp_alpha : a number between 0 and 1
- criterion : 'entropy' or 'gini'
- max_depth : A number greater than 0
- max_leaf_nodes : A number greater than 1

In [22]:
```
rfc = RandomForestClassifier(ccp_alpha = 0.04, criterion = 'entropy', max_depth = 8, max_leaf_nodes = 2 )
rfc.fit(x_train,y_train)
plot_confusion_matrix(rfc, x_test, y_test,display_labels=['Diabetes','Not Diabetes'], cmap=plt.cm.Blues,)
plt.show()
print(rfc.score(x_test, y_test))
```

```
C:\Anaconda3\envs\ML_Course_2023\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function plo
t_confusion_matrix is deprecated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be removed in
1.2. Use one of the class methods: ConfusionMatrixDisplay.from_predictions or ConfusionMatrixDisplay.from_estim
ator.
  warnings.warn(msg, category=FutureWarning)
```



True label	Diabetes	Not Diabetes
Diabetes	108	4
Not Diabetes	35	7

```
0.7467532467532467
```

4.2 Automated search of optimum hyper-parameters

Experimenting with many parameters, each with many potential settings is time consuming. Additionally, there are further risks of over-fitting on the hyper-parameters since the same data is being used repeatedly for testing.

Alternatively, there is an automated method within sklearn for searching for optimum hyper-parameters.

First, a list is created that identifies the hyper-parameters and the range of values that should be applied to each:

In [23]:
```
param_grid = [
    {'ccp_alpha':[0.0, 0.005, 0.007, 0.01, 0.015, 0.02, 0.03, 0.05, 0.1],
    'criterion':['gini', 'entropy'],
    'max_depth': [2,3,4,5,6,7,8,9],
    'max_leaf_nodes' : [2,3,4,5,6,7,8,9]}
]
```

Then use the 'GridSearchCV' function to try each permutation of hyper-parameter until the optimum settings are discovered.

WARNING This operation may take a considerable amount of time, since the model has to be built for each combination of parameter settings. On my CORE i7-7700HQ gaming laptop with a GPU it takes about 20 minutes to run! So if you have a slower machine - you may be waiting a long time!

In [ ]:
```
clf = GridSearchCV(estimator=rfc,
           param_grid=param_grid)
best_model= clf.fit(x_train,y_train)
```

Finally, we can review the confusion matrix for the model with optimum settings:

In [ ]:
```
plot_confusion_matrix(best_model, x_test, y_test,display_labels=['Diabetes','Not Diabetes'], cmap=plt.cm.Blues,
plt.show()
```

In [ ]:
```
print(best_model.score(x_test, y_test))
```

In [ ]: