In [1]:

```
%matplotlib inline
from preamble import *
```

Introducción a los árboles de decisión

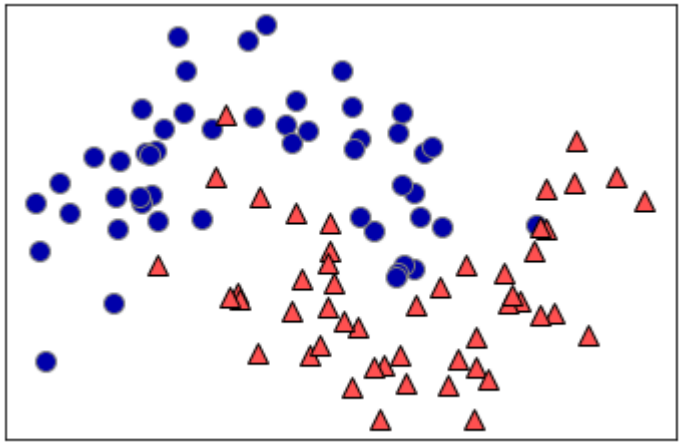
Construcción incremental de un árbol de Decisión

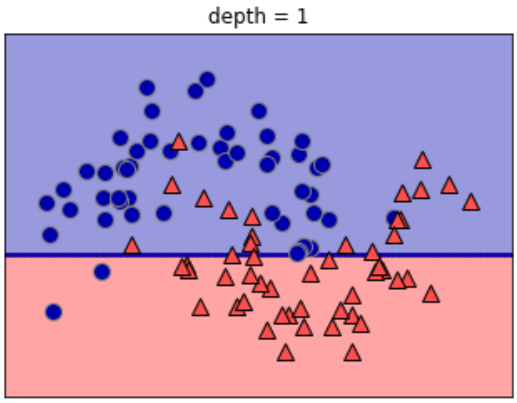
`

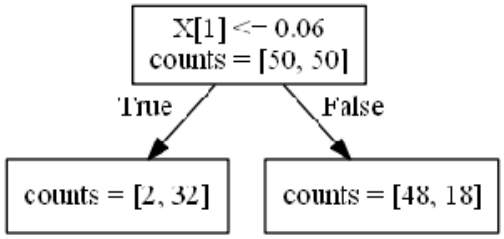
Analicemos en primer lugar, cómo se construye incrementalmente un árbol de decisión, con un conjunto de datos para clasificación en 2D que referiremos comotwo_moon`. Esta colección consiste de dos formas de medialunas, con cada clase consistiendo en 50 puntos

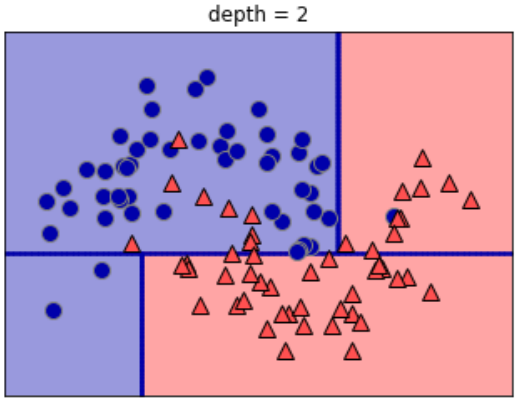
In [2]:

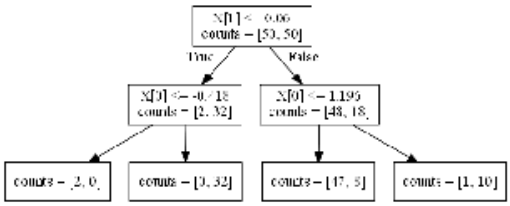
```
mglearn.plots.plot_tree_progressive()
```

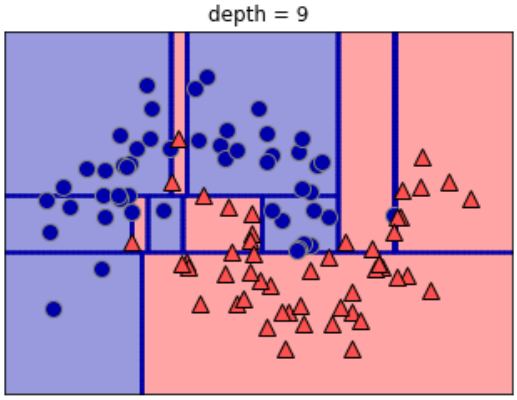


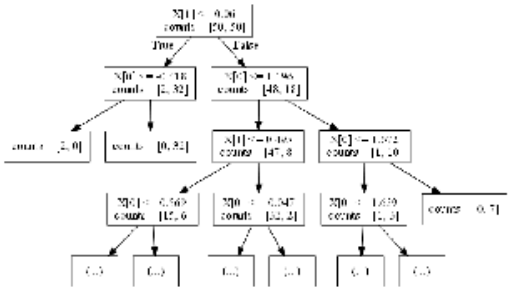

depth = 1


X[1] <= 0.06
counts = [50, 50]
True
False
counts = [2, 32]
counts = [48, 18]


depth = 2







Habiendo analizado intuitivamente la generación incremental de un árbol de decisión, procederemos ahora a presentar al data set con el cual se ejemplificarán los restantes conceptos de este tema, el *Breast Cancer Wisconsin (Diagnostic) Data Set*.

Breast Cancer Wisconsin (Diagnostic) Data Set

Descripción de la colección

El *"Breast Cancer Wisconsin (Diagnostic) Data Set"* es uno de los datasets más populares en Minería de Datos, y está disponible en el repositorio de Machine Learning de la UCI (https://archive.ics.uci.edu/ml/datasets), específicamente en https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic (https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic))

Esta colección tiene información de 569 muestras de tejido orgánico de mamas en las que, a partir de imágenes digitalizadas, se derivan una serie de 10 características medidas en los núcleos de las células: textura, perímetro, área, simetría, etc. Para cada muestra se ha registrado su valor medio en las células de dicha muestra, su desviación standard y el caso más desfavorable, lo cual da un total de 30 características (features) numéricas disponibles para la predicción. Cada ejemplo, contiene además una clase que indica el carácter maligno o benigno del tumor y un atributo identificador del ejemplo, lo que da un total de 32 atributos por ejemplo (incluida la clase y el atributo identificador).

Para trabajar con este dataset, en primer lugar seguiremos un enfoque similar al utilizado en la segunda notebook utilizada para el data ser **Iris** (**clase-ap-auto-ejemplos-iris-v2.ipynb**) donde se usaba la utilidad de scikit_learn `load_iris()` que retornaba un objeto `Bunch`. En este caso, scikit-learn provee otra función de carga específica para cargar el data set de cáncer de mama, que se denomina `load_breast_cancer()`

In [3]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): {}".format(cancer.keys()))
```

```
cancer.keys(): dict_keys(['data', 'target', 'target_names', 'DESCR', 'feat
ure_names'])
```

In [4]:

```
print("Shape of cancer data: {}".format(cancer.data.shape))
```

```
Shape of cancer data: (569, 30)
```

In [5]:

```
print("Sample counts per class:\n{}".format(
      {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))
```

```
Sample counts per class:
{'malignant': 212, 'benign': 357}
```

In [6]:

```
print("Feature names:\n{}".format(cancer.feature_names))
```

```
Feature names:
['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']
```

Generando Arboles de Decisión con los datos de breast-cancer

Lo primero que haremos es importar el tipo de modelo apropiado para este caso (`DecisionTreeClassifier`), instanciarlo, entrenarlo con los datos de `breast-cancer` y evaluarlo.

In [7]:

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

Es claro en este caso que el modelo obtenido trabaja muy bien sobre los datos de entrenamiento, pero no tan bien sobre el conjunto de test (sobreajuste). Como se vió en teoría, una forma de aliviar este problema es detener la generación de nuevos niveles del árbol (pre-poda) utilizando algún criterio como limitar *la profundidad máxima del árbol*, el *número máximo de hojas* o requerir un *mínimo número de puntos* en el nodo para seguir dividiéndolo.

Usemos la primera opción de pre-poda, limitando la profundidad máxima del árbol seteando el hiperparámetro `max_depth`

In [8]:

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

Como vemos, esto lleva a una menor accuracy en el conjunto de entrenamiento pero una mejora en el test set.

Analizando qué sucede con las distintas profundidades, podemos tener una idea más clara de los fenómenos de "subajuste" y "sobreajuste"

In [9]:

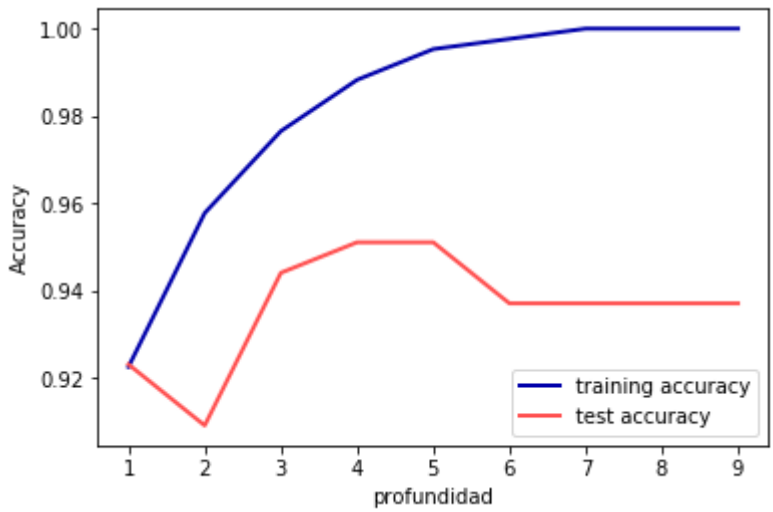
```
training_accuracy = []
test_accuracy = []
# probar profundidades de 1 a 10
profundidades_settings = range(1, 10)

for n_prof in profundidades_settings:
    # build the model
    clf = DecisionTreeClassifier(max_depth=n_prof, random_state=0)
    clf.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(clf.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(profundidades_settings, training_accuracy, label="training accuracy")
plt.plot(profundidades_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("profundidad")
plt.legend()
```

Out[9]:

```
<matplotlib.legend.Legend at 0x14041e1d6d8>
```



In [10]:

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

Visualizando y analizando árboles de decisión

Es interesante visualizar el árbol de decisión generado, exportándolo como un archivo ".dot" (formato de archivo de textos para almacenar grafos) y visualizandolo con `graphviz`

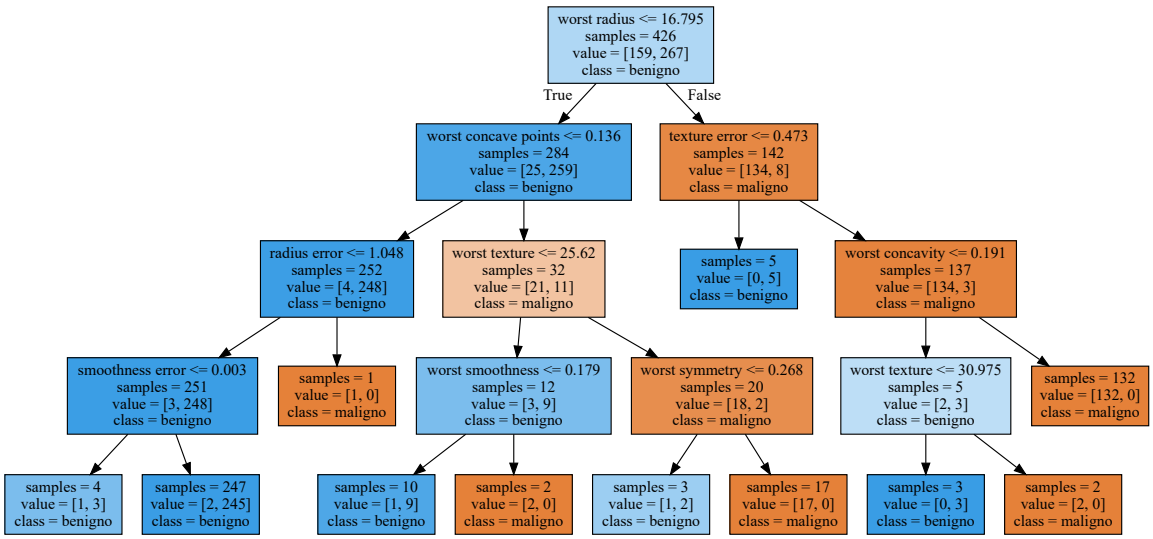
In [11]:

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["maligno", "benigno"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

In [12]:

```
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
display(graphviz.Source(dot_graph))
```



Importancia de las características en los árboles

Otra forma de resumir la información aprendida en un árbol de decisión, es usar una medida de la "importancia de las características" (*feature importance*) la cual evalúa cuan importante es una característica en él arbol de decisión en que participa. Es un numero entre 0 y 1 para cada feature donde 0 significa "no es considerado en absoluto" y 1 significa "perfectamente predice la clase". Las importancias de todas las características siempre deben sumar 1:

In [13]:

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

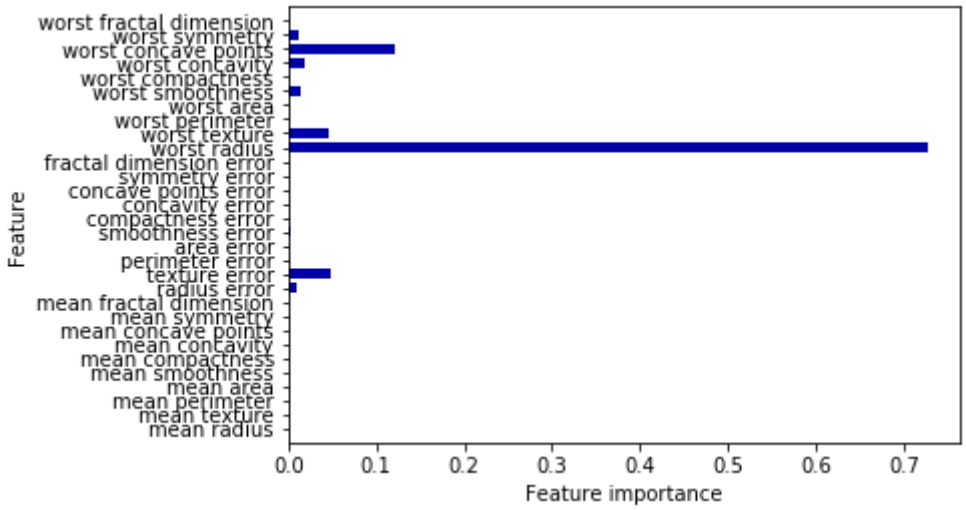
```
Feature importances:
[0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.01  0.048
 0.    0.    0.002 0.    0.    0.    0.    0.    0.727 0.046 0.    0.
 0.014 0.    0.018 0.122 0.012 0.   ]
```

También se puede visualizar la importancia de las características de una manera más gráfica

In [14]:

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
    plt.ylim(-1, n_features)

plot_feature_importances_cancer(tree)
```



Aqui se puede observar que la característica usada en el nodo del tope ("worst radius") es, por lejos, la característica más importante y confirma la intuición de que este control en la raíz, ya separaba bastante bien las dos clases.

Es importante notar que la importancia de las características siempre será positiva, pero no indica de cual clase esta característica es indicativa . Para este ejemplo, significa que "worst radius" es importante, pero que un radio alto no es indicativo de que una muestra sea maligna o benigna. Para concluir con este tema, si bien la ganancia de información es la principal candidata para evaluar la importancia de una característica, de acuerdo al manual de scikit-learn este valor se calcula como la reducción total (normalizada) del criterio evaluado para esta característica (importancia de Gini).