In [1]:

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

Introducción a los árboles de decisión (Parte II)

Ensambles de Árboles de Decisión

Los ensambles son métodos que combinan múltiples modelos de aprendizaje automático para crear modelos más poderosos. Hay muchos modelos en la literatura de aprendizaje automático que pertenecen a esta categoría, pero en esta notebook usaremos dos enfoques que han demostrado ser efectivos en una amplia gama de conjuntos de datos para clasificación y regresión, los cuales utilizan árboles de decisión como sus componentes básicos: *Random Forest* y *Gradient Boosted Regression Trees (Gradient Boosting Machines)*.

El ensamble de varios clasificadores elementales es una técnica de larga data y que ha permitido en general evitar problemas como el sobreajuste. La idea es que estos clasificadores elementales son obtenidos utilizando el mismo método de aprendizaje pero variando levemente las colecciones con que son entrenados o los atributos (features) que se usan en la generacíon del modelo. Un enfoque bastante usado en este caso es el de "bagging", que en Scikit Learn es provito mediante el meta-estimator `BaggingClassifier`, que para el caso de obtener un ensamble de árboles de decisión podría ser:

In [2]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()


tree = DecisionTreeClassifier()
bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                        random_state=0)
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)

bag.fit(X_train, y_train)

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

```
Accuracy on training set: 0.998
Accuracy on test set: 0.951
```

Es importante notar que bagging tiene parámetros para controlar la aletoriedad en los sub-sets que se generan para entrenar cada clasificador como así también en los atributos que serán seleccionados para el modelo.

Random forests (bosques aleatorios)

Si bien en el caso anterior hicimos un ensamble "manual" de varios árboles de decisión (podria haberse utilizado otro estimador elemental) con un método general como bagging, también existen métodos específicos pensados para el caso de árboles de decisión como lo es *Random forests*

Como vimos previamente, un inconveniente principal de los árboles de decisión es que tienden a sobreajustar los datos de entrenamiento. Random forest es una forma de abordar este problema. Un random forest es esencialmente una colección de árboles de decisión, donde cada árbol es ligeramente diferente de los demás. La idea detrás de los bosques aleatorios es que cada árbol podría hacer una buen trabajo de predicción, pero es probable que se adapte a parte de los datos. Si construimos muchos árboles, todos los cuales funcionan bien y se ajustan (aprenden) de diferentes maneras, podemos reducir la cantidad de sobreajuste al promediar sus resultados. Esta reducción en el sobreajuste, mientras se mantiene el poder predictivo de los árboles, puede ser demostrado matemáticamente.

Para implementar esta estrategia, necesitamos construir muchos árboles de decisión. Cada arbol debe hacer un trabajo aceptable de predecir el objetivo, y también debe ser diferente de los otros arboles Los bosques aleatorios obtienen su nombre de inyectar aleatoriedad en la construcción del árbol para asegurarse de que cada árbol es diferente. Hay dos formas de introducir aleatoriedad en los árboles de un bosque: seleccionando los puntos de datos utilizados para construir un árbol y seleccionando las características en cada test que se realiza en la generacion de un nodo del árbol.

En este sentido Random Forest realiza un muestreo "bootstrap" sobre el conjunto de entrenamiento que lleva a que cada árbol de decisión en el bosque aleatorio que se está construyendo sea entrenado en un conjunto de datos ligeramente diferente. También introduce aleatoriedad en la selección de características en cada nodo, por lo que cada división en cada árbol opera sobre un subconjunto diferente de características. Juntos, estos dos mecanismos aseguran que todos los árboles en el bosque aleatorio sean diferentes.

En este contexto, un parámetro crítico en este proceso es `max_features`. Si configuramos `max_features` = al número de características, significa que cada división puede ver todas las características en el conjunto de datos y ninguna aleatoriedad se inyectará en la selección de características (aunque se sigue manteniendo la de bootstrapping). Si configuramos `max_features` en 1, eso significa que sólo una característica será considerada como alternativa. Por lo tanto, un alto `max_features` significa que los árboles en el bosque aleatorio serán bastante similares, y serán capaces de ajustar los datos fácilmente, utilizando las características más distintivas. A `max_features` más bajos significa que los árboles en el bosque aleatorio serán bastante diferentes, y que cada árbol es posible que tenga que ser muy profundo para ajustar bien los datos.

Para hacer una predicción usando el bosque aleatorio, el algoritmo primero hace una predicción por cada árbol en el bosque. Para la regresión, podemos promediar estos resultados para obtener nuestra predicción final. Para la clasificación, se utiliza una estrategia de "votación suave". Esto significa que cada algoritmo hace una predicción "suave", proporcionando una probabilidad para cada salida (clase) posible. Luego, las probabilidades predichas por todos los árboles se promedian, y la clase con la probabilidad más alta es la elegida.

Analicemos cómo se comporta un bosque aleatorio con 5 estimadores sobre el data set `two_moons` que utilizamos previamente visualizando los límites de decisión aprendidos por cada árbol y también por el bosque.

In [3]:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

Out[3]:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gin
i',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=5, n_jobs=1,
            oob_score=False, random_state=2, verbose=0, warm_start=False)
```

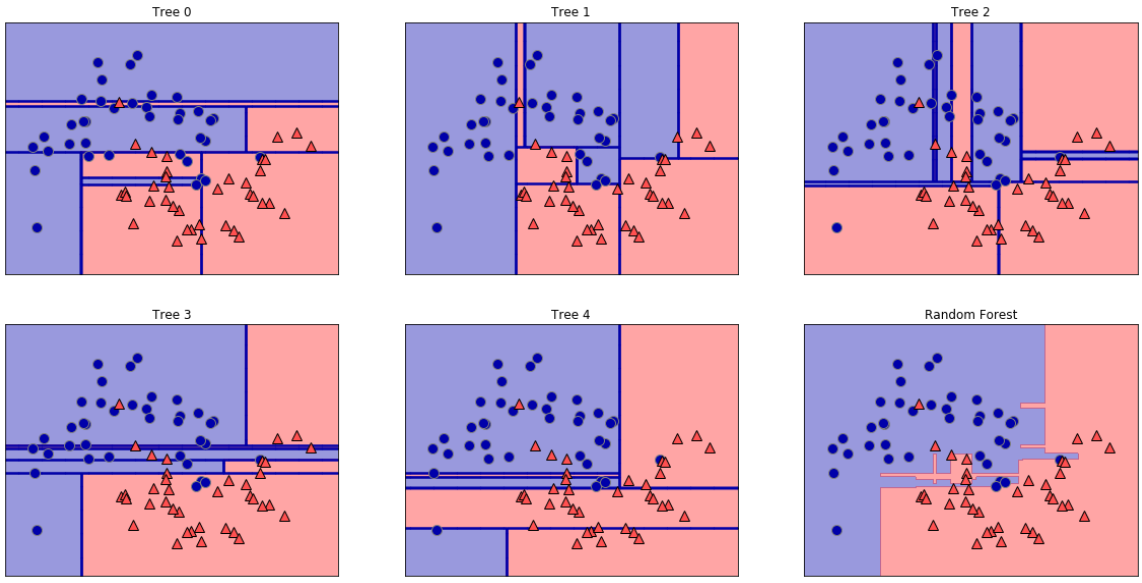
In [4]:

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
                                alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x175b1f556a0>,
 <matplotlib.lines.Line2D at 0x175b1f55b70>]
```



Se puede ver claramente que los límites de decisión aprendidos por los cinco árboles son bastante diferentes y cada uno de ellos comete algunos errores. El bosque aleatorio sobreajusta menos que cualquiera de los árboles individualmente, y proporciona un límite de decisión mucho más intuitivo. En cualquier aplicación real, usaríamos muchos más árboles (a menudo cientos o miles), lo que lleva a límites aún más suaves.

Como otro ejemplo, apliquemos un random forest con 100 árboles al data set de Breast Cancer:

In [5]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()


X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

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

```
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```

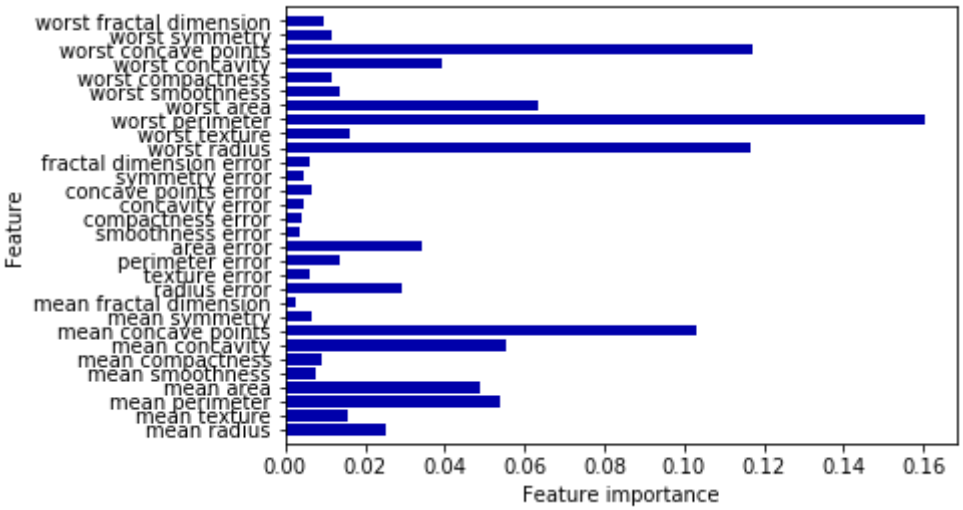
El bosque aleatorio nos da una precisión del 97%, mejor que un árbol de decisión único, sin ajustar ningún parámetro. Podríamos ajustar el `max_features` o aplicar una poda previa como lo hicimos para el árbol de decisión único. Sin embargo, a menudo los parámetros por defecto del bosque aleatorio ya funcionan bastante bien.

Al igual que para árboles de decisión el bosque aleatorio permite ver la importancia de las características, que se calculan agregando las importancias de las características de los árboles en el bosque. Típicamente, la importancia de las características proporcionadas por el bosque aleatorio es más confiable que la proporcionada por un solo árbol. Esta información se muestra a continuación:

In [6]:

```
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(forest)
```



Gradient Boosted Regression Trees (Gradient Boosting Machines)

Los árboles de regresión reforzados con el gradiente es otro método de ensamble que combina múltiples árboles de decisión para crear un modelo más poderoso. A pesar de la "regresión" en el nombre, estos modelos se pueden utilizar para la regresión y la clasificación. En contraste con el enfoque del bosque aleatorio, este método funciona mediante la construcción de árboles de manera serial, donde cada árbol intenta corregir los errores del anterior. Por defecto, no hay aleatorización en los árboles de regresión potenciados por gradiente; en su lugar, se utiliza una fuerte poda previa. Este enfoque a menudo utiliza árboles muy poco profundos, de una profundidad de uno a cinco, lo que hace que el modelo sea más pequeño en términos de memoria y haga predicciones más rápidas.

La idea principal detrás de este método es combinar muchos modelos simples (en este contexto, conocidos como "aprendices débiles"), como lo son los árboles de poca profundidad. Cada árbol solo puede proporcionar buenas predicciones sobre parte de los datos, por lo que se agregan cada vez más árboles para mejorar el rendimiento de forma iterativa.

Este enfoque es uno de los enfoques más ganadores en las competiciones de aprendizaje automático, y es ampliamente utilizado en la industria. En general, son un poco más sensibles a la configuración de parámetros que los bosques aleatorios, pero pueden proporcionar una mejor precisión si los parámetros se configuran correctamente.

Aparte de la poda previa y el número de árboles en el conjunto, otro importante parámetro de este algoritmo es la `learning_rate` (tasa de aprendizaje), que controla cuan fuertemente cada árbol intenta corregir los errores de los árboles anteriores. Una mayor tasa de aprendizaje significa que cada árbol puede hacer correcciones más fuertes, lo que permite modelos más complejos. Agregando más árboles al conjunto (mediante el aumento de `n_estimators`), también aumenta la complejidad del modelo, ya que el modelo tiene más posibilidades de corregir errores en el conjunto de entrenamiento.

A continuación se da un ejemplo del uso de `GradientBoostingClassifier` en la colección de cáncer de mama.Por defecto, se usan 100 árboles de profundidad máxima 3 y una tasa de aprendizaje de 0.1

In [7]:

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

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

```
Accuracy on training set: 1.000
Accuracy on test set: 0.958
```

Parcería que hay algo de sobreajuste y podemos tratar de evitarlo ya sea con una pre-poda más fuerte o bajando la taza de aprendizaje

In [8]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

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

```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

In [9]:

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)

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

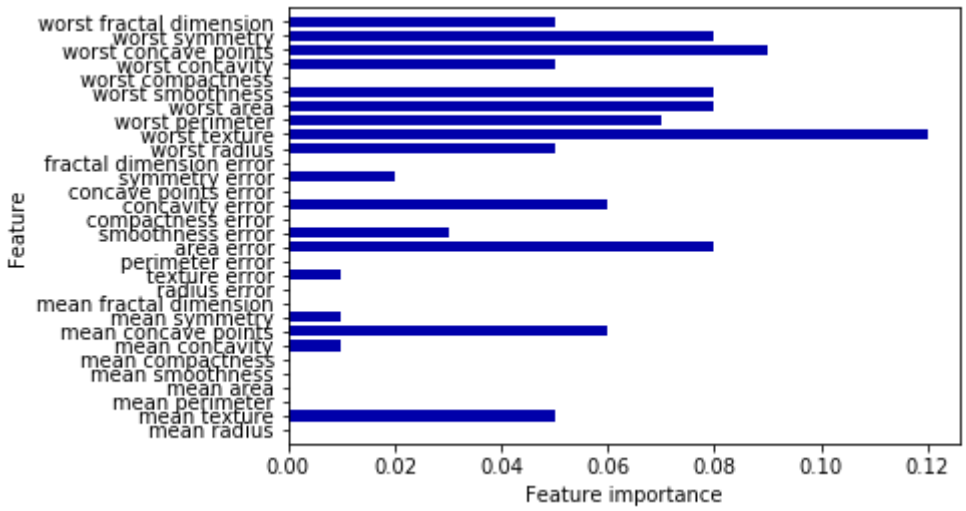
```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

Al igual que en los otros casos, también se puede visualizar la importancia de las características

In [10]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

plot_feature_importances_cancer(gbrt)
```



Práctica: Random Forest para clasificar dígitos

Utilizando los datos de dígitos manuscritos introducidos en las primeras teorías

In [11]:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.keys()
```

Out[11]:

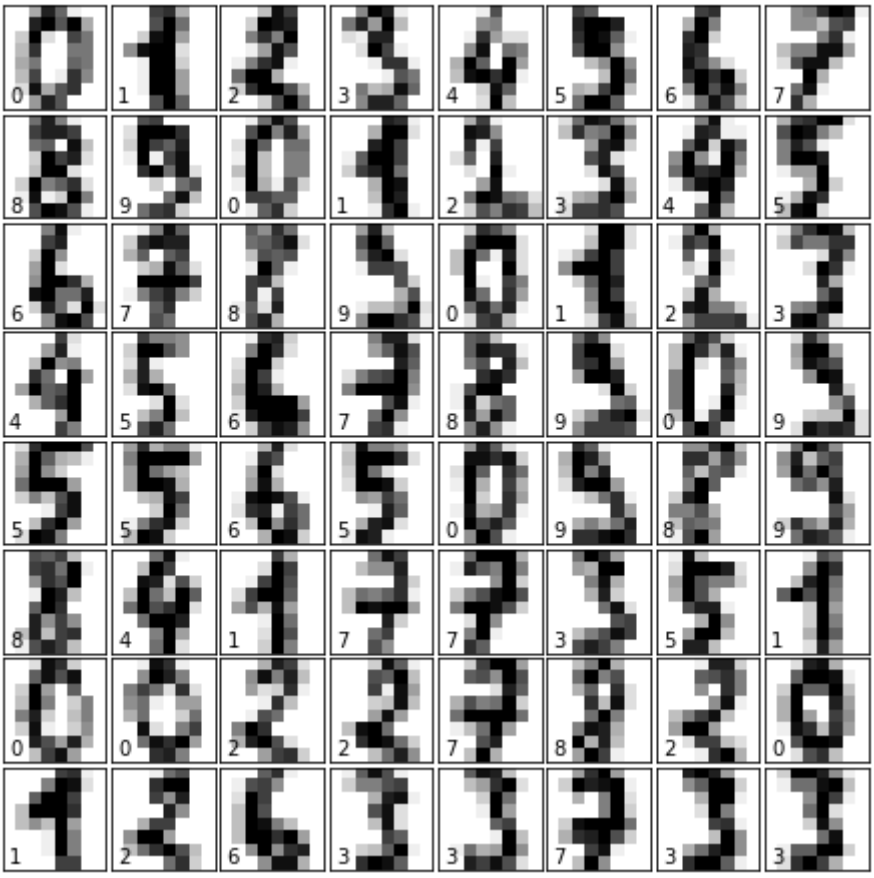
```
dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
```

In [12]:

```
# set up the figure
fig = plt.figure(figsize=(6, 6))  # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot the digits: each image is 8x8 pixels
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

    # label the image with the target value
    ax.text(0, 7, str(digits.target[i]))
```



Analice el desempeño que se obtiene en este caso con árboles de decisión comunes y con los métodos de ensamble antes vistos.