Andrés Almécija

Table of Contents

Práctica No. 3. Algoritmos de Machine Learning.

Esta práctica constituye la tercera del módulo **Fundamentos de Machine Learning: datos y algortimos** dentro del **Programa Executive en Artificial Intelligence** de **ThreePoints** dedicada a la aplicación de algortimos de Machine Learning de cuerdo con lo especificado en la Unidad 3 del módulo.

Finalidad de la práctica

Se investigarán algortimos de aprendizaje supervisado y no supervisado sobre datasets conocidos. Aprenderemos la metodología de uso de los principales `estimadores` de `Scikit-Learn API` , entre los que se incluyen los siguientes pasos:

1. Elección una clase de modelo importando la clase de estimador adecuada desde `Scikit-Learn` .
2. Elección de hiperparámetros creando una instancia de esta clase con los valores deseados.
3. Organización los datos en una matriz de características y un vector objetivo siguiendo la discusión anterior.
4. Ajuste del modelo a sus datos llamando al método `fit ()` de la instancia del modelo.
5. Aplicación el modelo a los nuevos datos:

    - Para el aprendizaje supervisado, a menudo predecimos etiquetas para datos desconocidos utilizando el método predict ().
    - Para el aprendizaje no supervisado, a menudo transformamos o inferimos las propiedades de los datos utilizando el método transform () o predict ().

Ahora veremos varios ejemplos simples de aplicación de métodos de aprendizaje supervisados y no supervisados.

Introduciendo Random Forests y Decision Trees

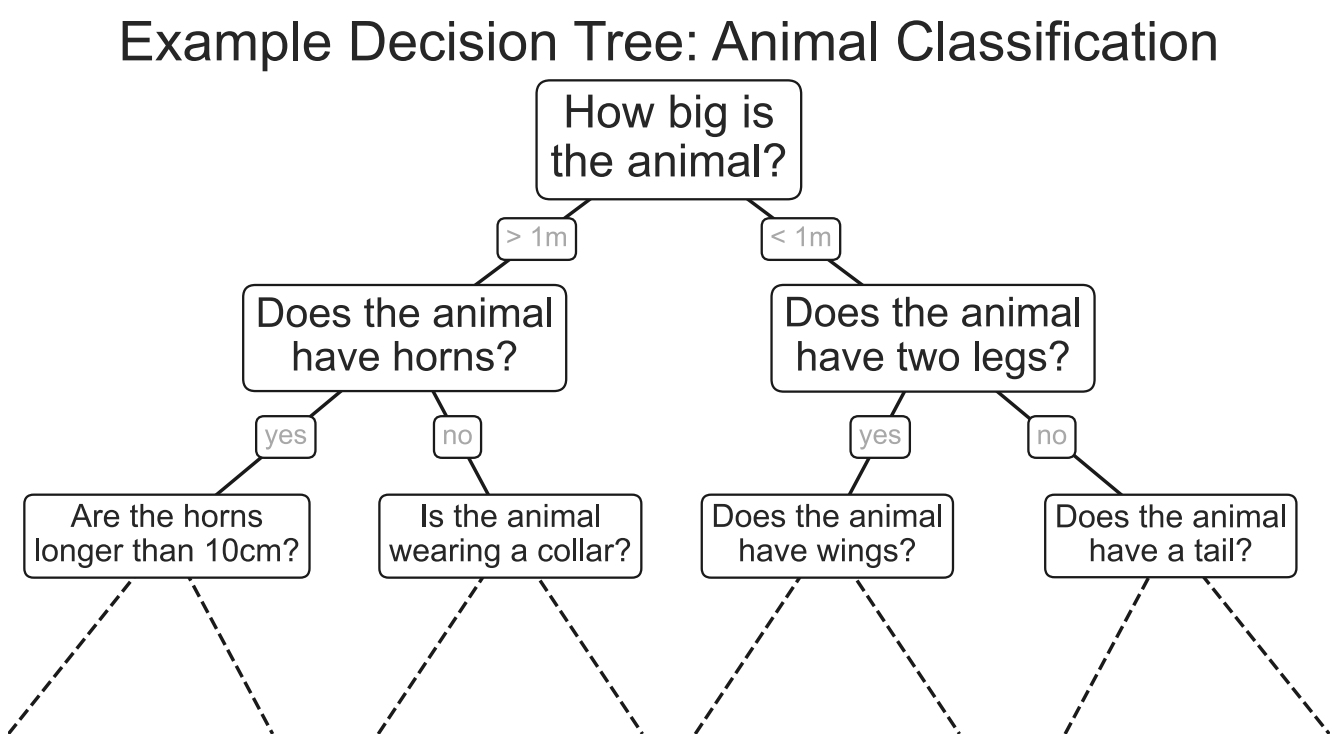
Los **Random Forests** son un ejemplo de un *ensemble learner* construido sobre árboles de decisión. Por esta razón, comenzaremos discutiendo los **árboles de decisión** o **Decision Trees**.

Los árboles de decisión son formas muy intuitivas de clasificar o etiquetar objetos: simplemente hace una serie de preguntas diseñadas para enfocarse en la clasificación:

Importamos las librerías necesarias

In [142]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from IPython.core.display import HTML
import numpy as np
import pandas as pd
sns.set()
```

In [143]:
```
import fig_code
fig_code.plot_example_decision_tree()
```

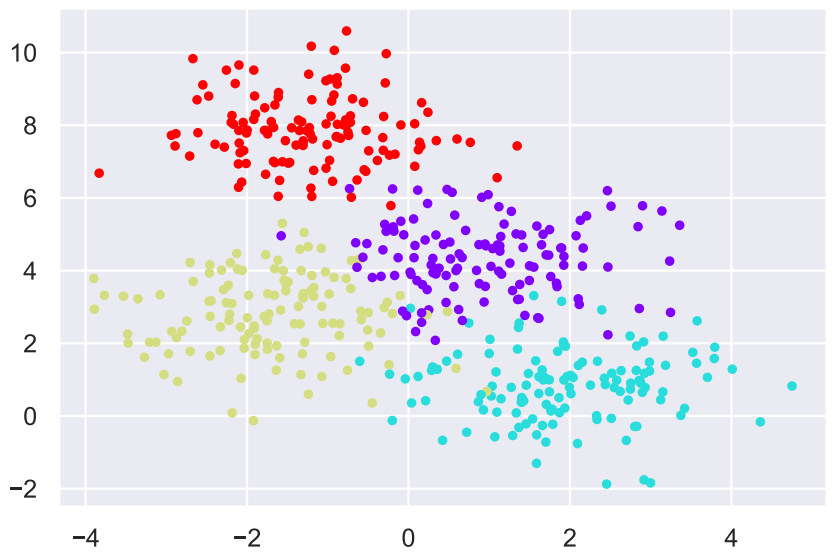


La división binaria hace que esto sea extremadamente eficiente. Como siempre, sin embargo, el truco es *hacer las preguntas correctas* . Aquí es donde entra el proceso algorítmico: en el entrenamiento de un clasificador de árbol de decisión, el algoritmo examina las características y decide qué preguntas (o "divisiones") contienen la mayor información.

Creando un Decision Tree

Aquí hay un ejemplo de un clasificador de árbol de decisión en `scikit-learn` . Comenzaremos por definir algunos datos etiquetados bidimensionales:

In [144]:
```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=500, centers=4,
                  random_state=0, cluster_std=1.0)
plt.scatter(X[:, 0], X[:, 1], c=y, s=10, cmap='rainbow');
```



Hacemos llamada a algunas funciones de ayuda:

In [145]:
```
from fig_code import visualize_tree, plot_tree_interactive
```

Usamos IPython's `interact` (Disponible en IPython 2.0+, requiere un live kernel) para ver cómo el árbol parte los datos progresivamente:

In [146]:
```
plot_tree_interactive(X, y);
```

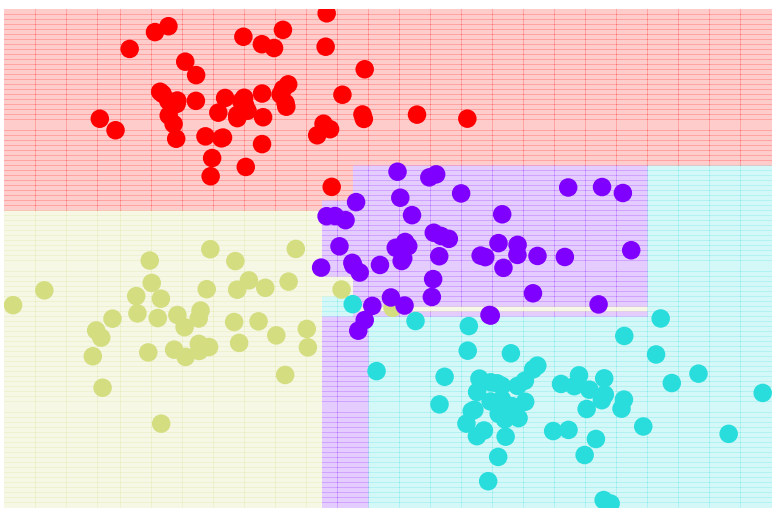
Tenga en cuenta que a cada aumento de profundidad, cada nodo se divide en dos, **excepto los nodos que contienen una sola clase**. El resultado es una clasificación **no paramétrica** muy rápida y puede ser extremadamente útil en la práctica.

Decision Trees y Over-fitting

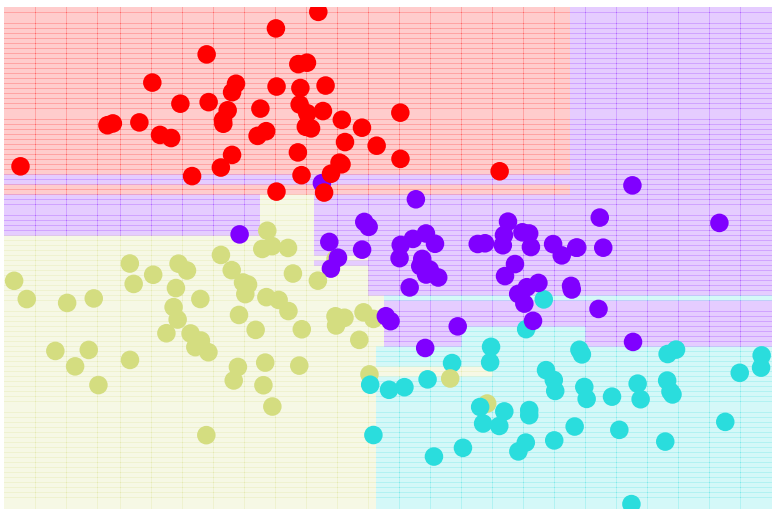
Un problema con los árboles de decisión es que es muy fácil crear árboles que **se ajustan en exceso** a los datos. Nos encontramos con **overfitting**. Es decir, ¡son lo suficientemente flexibles como para que puedan aprender la estructura del ruido en los datos en lugar de la señal! Por ejemplo, observe dos árboles construidos en dos subconjuntos de este conjunto de datos:

In [147]:
```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier()

plt.figure()
visualize_tree(clf, X[:200], y[:200], boundaries=False)
plt.figure()
visualize_tree(clf, X[-200:], y[-200:], boundaries=False)
```

```
<Figure size 432x288 with 0 Axes>
```



```
<Figure size 432x288 with 0 Axes>
```



¡Los detalles de las clasificaciones son completamente diferentes! Eso es una indicación de **ajuste excesivo**: cuando predice el valor para un nuevo punto, el resultado refleja más el ruido en el modelo que la señal.

Ensembles of Estimators: Random Forests

Una posible forma de abordar el **overfitting** es usar un **ensembles method**: este es un meta-estimador que esencialmente promedia los resultados de muchos estimadores individuales que sobre-ajustan los datos. Sorprendentemente, las estimaciones resultantes son mucho más sólidas y precisas que las estimaciones individuales que las componen.

Uno de los métodos de conjunto más comunes es el **Random Forest**, en el que el conjunto está formado por muchos árboles de decisión que de alguna manera están perturbados.

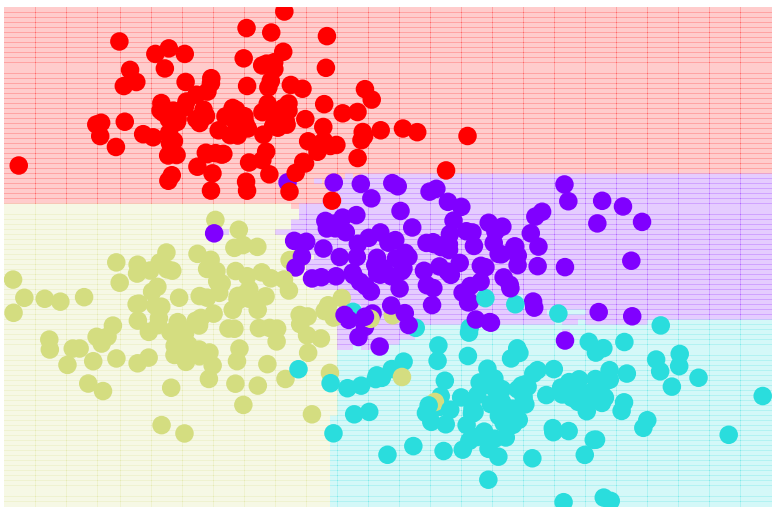
Hay volúmenes de teoría y precedentes sobre cómo aleatorizar estos árboles, pero como ejemplo, imaginemos que un conjunto de estimadores se ajusta a los subconjuntos de los datos. Podemos tener una idea de cómo se verían estos de la siguiente manera:

In [148]:
```
def fit_randomized_tree(random_state=0):
    X, y = make_blobs(n_samples=300, centers=4,
                      random_state=0, cluster_std=2.0)
    clf = DecisionTreeClassifier(max_depth=15)

    rng = np.random.RandomState(random_state)
    i = np.arange(len(y))
    rng.shuffle(i)
    visualize_tree(clf, X[i[:250]], y[i[:250]], boundaries=False,
                   xlim=(X[:, 0].min(), X[:, 0].max()),
                   ylim=(X[:, 1].min(), X[:, 1].max()))

from ipywidgets import interact
interact(fit_randomized_tree, random_state=(0, 100));
```

Vea cómo cambian los detalles del modelo en función de la muestra, mientras que las características más grandes siguen siendo las mismas.

El clasificador de bosque aleatorio hará algo similar a esto, pero usa una versión combinada de todos estos árboles para llegar a una respuesta final:

In [149]:
```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=100, random_state=0)
visualize_tree(clf, X, y, boundaries=False);
```



Al promediar **100 modelos perturbados al azar**, terminamos con un modelo general que se ajusta mucho mejor a nuestros datos.

(Nota: anteriormente, aleatorizamos el modelo mediante submuestreo ... Los bosques aleatorios utilizan medios de aleatorización más sofisticados, sobre los cuales puede leer, por ejemplo, la documentación de scikit-learn (http://scikit-learn.org/stable/modules/ensemble.html#forest))

Decision Trees y Random Forest para Classificación de Dígitos

Tomando el dataset de **hand-written digits** vamos a probar la eficacia de un clasificador por Decision Tree.

Cargamos el dataset en memoria.

In [150]:
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.keys()
```

Out[150]:
```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images', 'DESCR'])
```

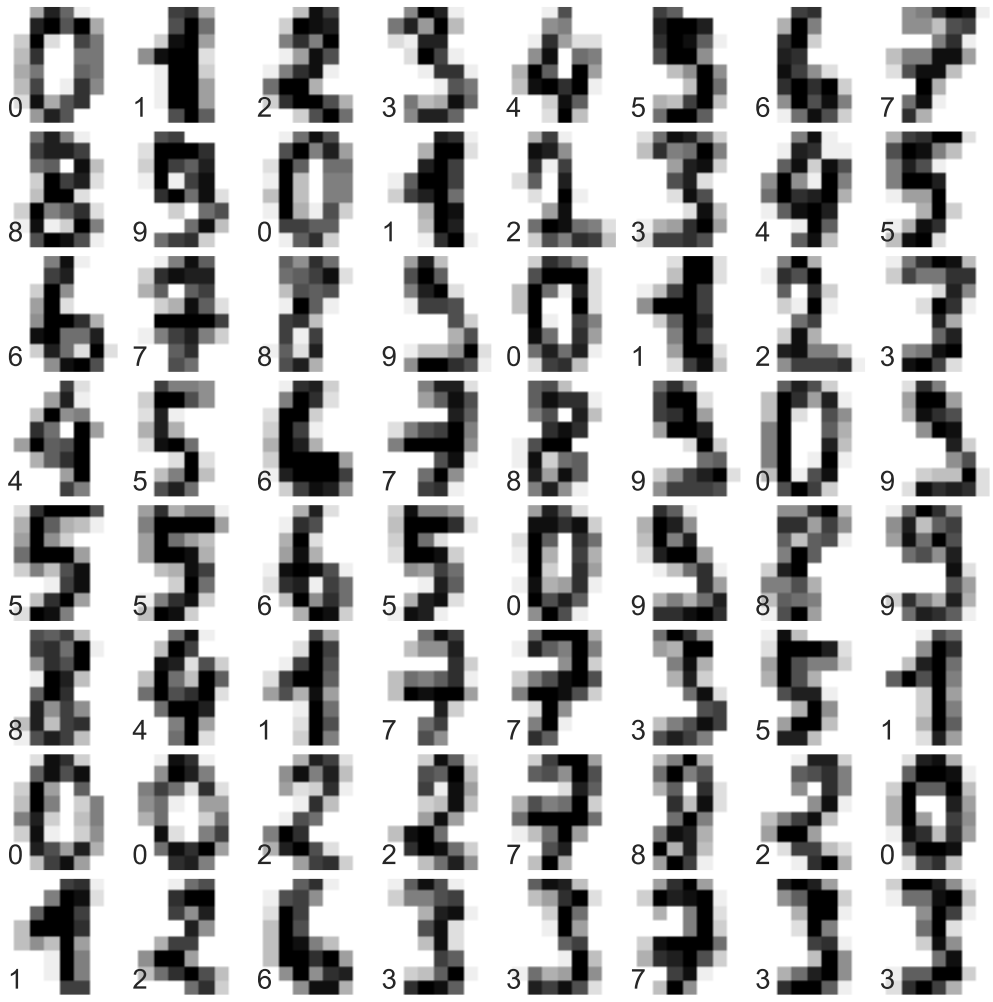
In [151]:
```
import collections
X = digits.data
y = digits.target
print(X.shape)
print(y.shape)
collections.Counter(digits.target)
```

```
(1797, 64)
(1797,)
```

Out[151]:
```
Counter({0: 178,
         1: 182,
         2: 177,
         3: 183,
         4: 181,
         5: 182,
         6: 181,
         7: 179,
         8: 174,
         9: 180})
```

Para recordarnos lo que estamos viendo, visualizaremos los primeros puntos de datos:

In [152]:
```
# 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]))
```



Podemos clasificar rápidamente los dígitos utilizando un árbol de decisión de la siguiente manera:

Hacemos una partición de los datos en set de entrenamiento y set de testeo con la función `train_test_split` :

In [153]:
```
from sklearn.model_selection import train_test_split
from sklearn import metrics

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
```

Definimos el `clasificador` y lo ajustamos a nuestros datos de entrenamiento.

In [154]:
```
clf = DecisionTreeClassifier(max_depth=5)
clf.fit(Xtrain, ytrain)
```

Out[154]:
```
DecisionTreeClassifier(max_depth=5)
```

Predecimos las etiquetas del test dataset, para posteriormente poder comprobar su validez contra las reales:

In [155]:
```
ypred = clf.predict(Xtest)
```

Podemos comprobar la exactitud de este clasificador:

In [156]:
```
print('Precisión sobre el test dataset: ', metrics.accuracy_score(ypred, ytest)*100, '%')
```

```
Precisión sobre el test dataset:  66.0 %
```

Matriz de confusión

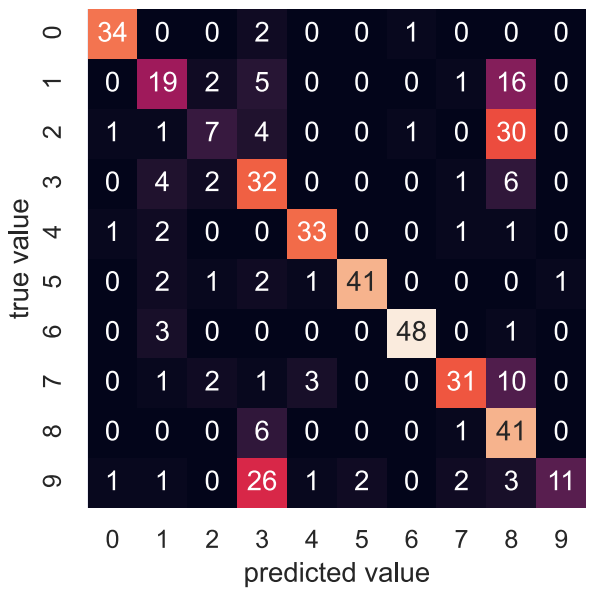
Visualicemos el comportamiento de nuestro árbol clasificador con la matriz de confusión.

Podéis encontrar detalles sobre esta matriz en [este artículo (https://data-speaks.luca-d3.com/2018/01/ML-a-tu-alcance-matriz-confusion.html)](https://data-speaks.luca-d3.com/2018/01/ML-a-tu-alcance-matriz-confusion.html).

In [157]:
```
from sklearn.metrics import classification_report, confusion_matrix
cm = confusion_matrix(ytest, ypred)
print(classification_report(ytest, ypred))
print(cm)
```

```
precision    recall  f1-score   support

           0       0.92      0.92      0.92        37
           1       0.58      0.44      0.50        43
           2       0.50      0.16      0.24        44
           3       0.41      0.71      0.52        45
           4       0.87      0.87      0.87        38
           5       0.95      0.85      0.90        48
           6       0.96      0.92      0.94        52
           7       0.84      0.65      0.73        48
           8       0.38      0.85      0.53        48
           9       0.92      0.23      0.37        47

    accuracy                           0.66       450
   macro avg       0.73      0.66      0.65       450
weighted avg       0.73      0.66      0.65       450

[[34  0  0  2  0  0  1  0  0  0]
 [ 0 19  2  5  0  0  0  1 16  0]
 [ 1  1  7  4  0  0  1  0 30  0]
 [ 0  4  2 32  0  0  0  1  6  0]
 [ 1  2  0  0 33  0  0  1  1  0]
 [ 0  2  1  2  1 41  0  0  0  1]
 [ 0  3  0  0  0  0 48  0  1  0]
 [ 0  1  2  1  3  0  0 31 10  0]
 [ 0  0  0  6  0  0  0  1 41  0]
 [ 1  1  0 26  1  2  0  2  3 11]]
```

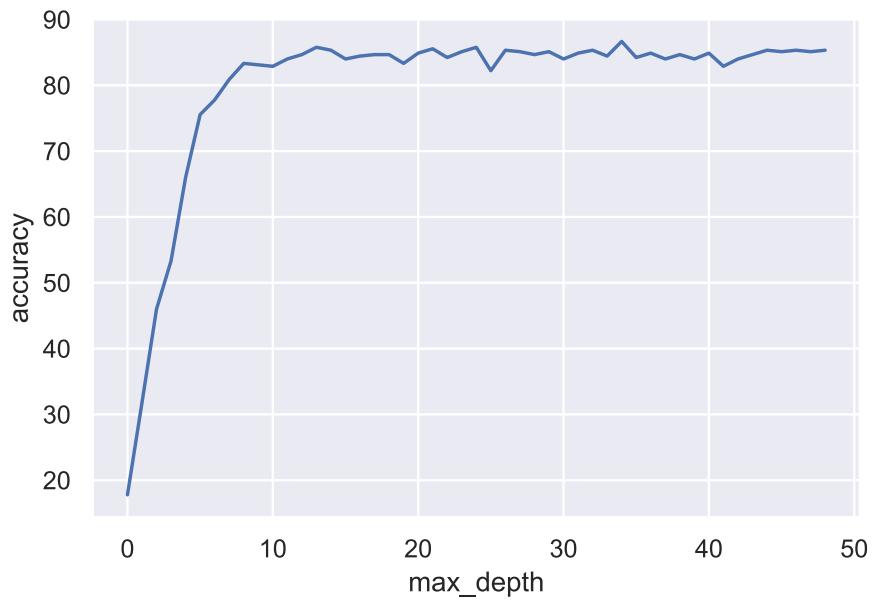
In [158]:
```
sns.heatmap(cm, square=True, annot=True, cbar=False)
plt.xlabel('predicted value')
plt.ylabel('true value');
```



Pregunta 1 (3 puntos):

Modifique el parámetro max_depth en `clf = DecisionTreeClassifier(max_depth = ...)` ¿Qué ocurre con la precisión sobre el test dataset cuando lo disminuímos? ¿Y cuando lo aumentamos?

In [159]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics

digits = load_digits()
X = digits.data
y = digits.target

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)

accuracies = []
for i in range(1, 50):
    clf = DecisionTreeClassifier(max_depth=i)
    clf.fit(Xtrain, ytrain)
    ypred = clf.predict(Xtest)
    accuracy = metrics.accuracy_score(ypred, ytest)*100
    accuracies.append(accuracy)

plt.plot(accuracies)
plt.ylabel('accuracy')
plt.xlabel('max_depth')
plt.show()
```



Al evaluar el comportamiento de `accuracy` al modificar el parámetro `max_depth` entre 1 y 50 se ve que:

- el valor crece rápidamente hasta `max_depth` = 8
- una vez que se alcanza este punto, `accuracy` oscila alrededor de ese valor

Entonces, por debajo de `max_depth = 5` , `accuracy` cae fuertemente. Por arriba de `max_depth = 5` el valor de `accuracy` crece, pero solo hasta cierto punto.

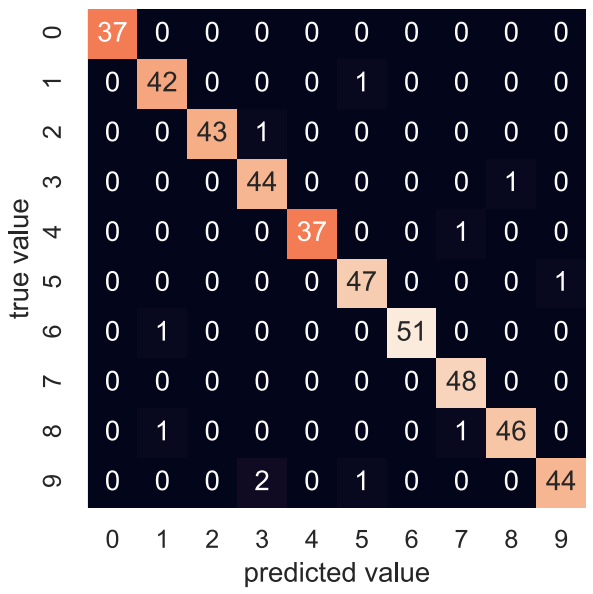
Pregunta 2 (6 puntos):

Repita esta clasificación con `sklearn.ensemble.RandomForestClassifier` ¿Mejoran los resultados de precisión en el test dataset? Represente la matriz de confusión.

Pista: Hay que ejecutar los mismos comandos que antes pero con una única variación:

In [166]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

digits = load_digits()
X = digits.data
y = digits.target

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)

clf = RandomForestClassifier(random_state=0)
clf.fit(Xtrain, ytrain)
ypred = clf.predict(Xtest)
accuracy = metrics.accuracy_score(ypred, ytest)*100
print(accuracy)

print(classification_report(ytest, ypred))
cm = confusion_matrix(ytest, ypred)

sns.heatmap(cm, square=True, annot=True, cbar=False)
plt.xlabel('predicted value')
plt.ylabel('true value');
```

```
97.55555555555556
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        37
           1       0.95      0.98      0.97        43
           2       1.00      0.98      0.99        44
           3       0.94      0.98      0.96        45
           4       1.00      0.97      0.99        38
           5       0.96      0.98      0.97        48
           6       1.00      0.98      0.99        52
           7       0.96      1.00      0.98        48
           8       0.98      0.96      0.97        48
           9       0.98      0.94      0.96        47

    accuracy                           0.98       450
   macro avg       0.98      0.98      0.98       450
weighted avg       0.98      0.98      0.98       450
```



from sklearn.ensemble import RandomForestClassifier clf = RandomForestClassifier()

Al utilizar `Random Forest` para este dataset, los resultados mejoran notablemente.

Por ejemplo, utilizando `decision tree` habían problemas al predecir el número 8 (muchos 2 eran clasificados como 8)

Por ser un dataset balanceado, se puede utilizar `accuracy` para comparar los resultados:

- Accuracy decision tree: 66%
- Accuracy random forest: 98%

Donde se ve una gran mejora.

Pregunta 3 (1 punto): De acuerdo a lo indicado en el siguiente enlace (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html). ¿Podría justificar los hiperparámetros elegidos dadas las características de nuestro dataset?

Los hiperparámetros elegidos son los propuestos por default.

Algunos de ellos:

- n_estimators: el número de árboles utilizados "en el bosque", por default son 100
- max_depth y min_samples_split: esta combinación indica el numero mínimo de muestras para dividir un nodo. Como max_depth=none y min_samples_split=2, estas divisiones serán cuando existan al menos 2 muestras.
- bootstrap: en cada árbol se utilizan algunos o todos los datos del dataset. Por default, se utilizan solo algunos datos.
- max_features: el número de caracterísitica a considerar cuando se hace una nueva división. Para nuestro ejemplo, al estar parametrizado en auto se considera max_features=sqrt(n_features), es decir 8

Pregunta 4 ¡OPCIONAL! (3 puntos adicionales):

Este artículo (https://chrisalbon.com/machine_learning/model_evaluation/cross_validation_parameter_tuning_grid_search/) es muy interesante a la hora de averiguar cómo realizar cross validation y hyperparameter tunning de un modelo.

¿Quién se atreve a reproducir los pasos indicados en este artículo para encontrar los parámetros óptimos en el `RandomForestClassifier()` ?

In [169]:
```
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

digits = load_digits()

X = digits.data
y = digits.target

#Conjuntos de entrenamiento y prueba
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)

#Parámetros con los que jugará GridSearchCV
parameter_candidates = [
  {'n_estimators': [100, 300, 500, 1000], 'criterion': ['gini', 'entropy']}
]

clf = GridSearchCV(estimator=RandomForestClassifier(random_state=0), param_grid=parameter_candidates, n_jobs=-1)

#Entrenar el clasificador con los datos de entrenamiento (con los distintos parámetros)
clf.fit(Xtrain, ytrain)


#Resultados
print('Mejor score para datos de entrenamiento: ', clf.best_score_)
print('Mejores parámetros: ')
print('- n_estimators:',clf.best_estimator_.n_estimators)
print('- criterion:',clf.best_estimator_.criterion)
```

```
Mejor score para datos de entrenamiento:  0.9769847170590665
Mejores parámetros:
- n_estimators: 300
- criterion: entropy
```

In [170]:
```
#Cross validation
#Ejecutar el clasificador entrenado con los datos de prueba
print("Score con el modelo obtenido por GridSearchCV:")
print(clf.score(Xtest, ytest))

#Entrenar un nuevo clasificador usando los parámetros encontrador por GridSearchCV
clf2 = RandomForestClassifier(n_estimators=clf.best_estimator_.n_estimators,criterion=clf.best_estimator_.criterion, random_state=
0)
clf2.fit(Xtrain, ytrain)

print("Score con un modelo entrenado con los parámetros encontrados por GridSearchCV:")
print(clf2.score(Xtest, ytest))
```

```
Score con el modelo obtenido por GridSearchCV:
0.9755555555555555
Score con un modelo entrenado con los parámetros encontrados por GridSearchCV:
0.9755555555555555
```

Como prueba de concepto, se utiliza GridSearchCV para que haga pruebas variando los hiperparámetros `n_estimators` y `criterion` .

El resultado es que se mejora el score original de 0.9755555555555556 por 0.9769847170590665 utilizando los parámetros n_estimators=300 y criterion=entropy

Por último se valida entrenando un nuevo modelo y se comparan los resultados obtenidos, los cuales coinciden.

¡Por favor, no olvide guardar el Jupyter Notebook antes de mandar la práctica!