1. Métodos de *clustering*:

Este ejercicio trata de explorar distintas técnicas de agrupamiento ajustándolas a distintos conjuntos de datos.

El objetivo es doble: entender la influencia de los parámetros en su comportamiento, y conocer sus limitaciones en la búsqueda de estructuras de datos.

In [1]:
```
import warnings

# Ignorar todos los warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
import random

import tqdm
import umap
import numpy as np
import pandas as pd
import sklearn
from sklearn import cluster        # Algoritmos de clustering.
from sklearn import datasets       # Crear datasets.
from sklearn import decomposition  # Algoritmos de reduccion de dimensionalidad.

# Visualizacion.
import matplotlib
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import plotly.express as px

%matplotlib inline
```

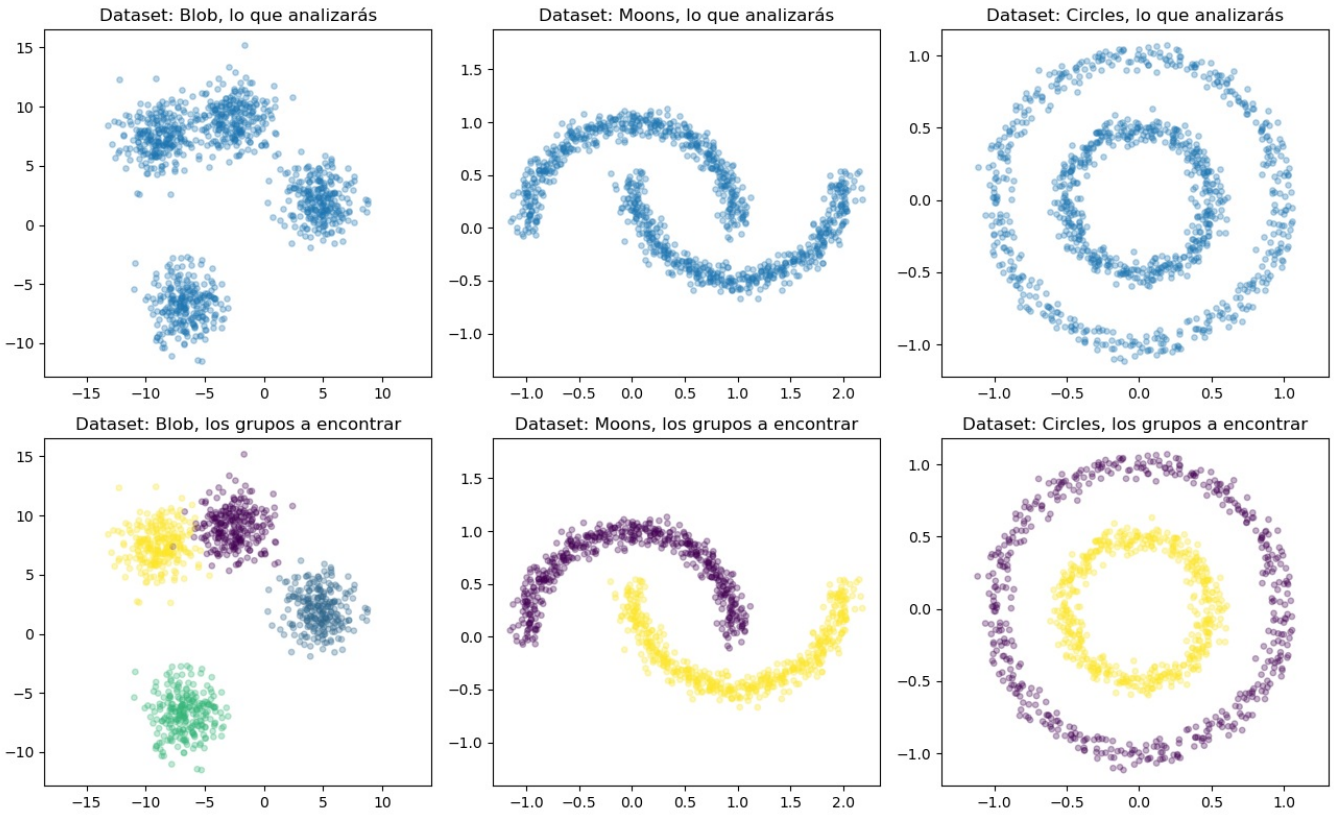
Generación de los conjuntos de datos

In [4]:
```
X_blobs, y_blobs = datasets.make_blobs(n_samples=1000, n_features=2, centers=4, cluster_std=1.6, random_state=4
X_moons, y_moons = datasets.make_moons(n_samples=1000, noise=.07, random_state=42)
X_circles, y_circles = datasets.make_circles(n_samples=1000, factor=.5, noise=.05, random_state=42)
```

Cada dataset tiene 2 variables: una variable *X* que contiene 2 features (columnas) y tantas filas como muestras. Y una variable *y* que alberga las etiquetas que identifican cada cluster.

A lo largo del ejercicio no se usará la variable *y* (sólo con el objetivo de visualizar). El objetivo es a través de los distintos modelos de *clustering* conseguir encontrar las estructuras descritas por las variables *y*.

In [5]:
```
fig, axis = plt.subplots(2, 3, figsize=(13, 8))
for i, (X, y, ax, name) in enumerate(zip([X_blobs, X_moons, X_circles] * 2,
                                         [None] * 3 + [y_blobs, y_moons, y_circles],
                                         axis.reshape(-1),
                                         ['Blob', 'Moons', 'Circles'] * 2)):
    ax.set_title('Dataset: {}, '.format(name) + ('lo que analizarás' if i < 3 else 'los grupos a encontrar'))
    ax.scatter(X[:,0], X[:,1], s=15, c=y, alpha=.3)
    ax.axis('equal')
plt.tight_layout()
```



1 a. K-means

En este apartado vamos a probar el algoritmo *k-means* sobre los tres datasets presentados anteriormente ajustando con los parámetros adecuados y analizar sus resultados.

In [6]:
```
X, y = X_blobs, y_blobs
```

Para estimar el número de clusters a detectar por *k-means*. Una técnica para estimar k es la que popularmente se conocer como *regla del codo*.

Primero es necesario calcular la suma de los errores cuadráticos (*SSE*) que consiste en la suma de todos los errores (distancia de cada punto a su centroide asignado) al cuadrado.

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} euclidean(x, c_i)^2$$

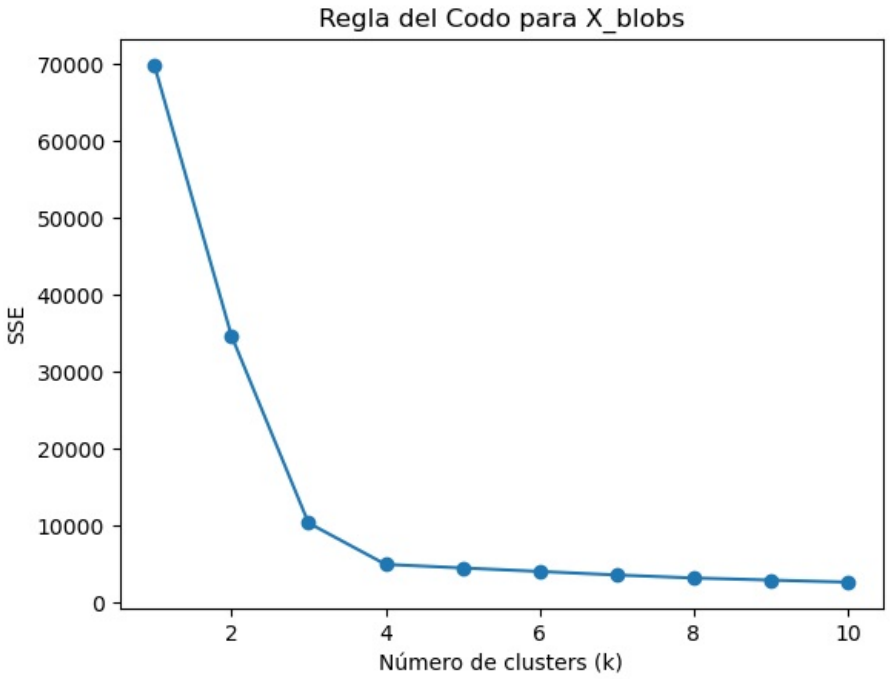
Donde K es el número de clusters a buscar por *k-means*, $x \in C_i$ son los puntos que pertenecen a i-ésimo cluster, c_i es el centroide del cluster C_i (al que pertenece el punto x), y $euclidean$ es la distancia euclídea.

Este procedimiento realizado para cada posible valor k, resulta en una función monótona decreciente, donde el eje x representa los distintos valores de k, y el eje y el SSE. Intuitivamente se podrá observar un significativo descenso del error, que indicará el valor idóneo de k.

Vamos realizar la representación gráfica de la regla del codo junto a su interpretación, utilizando la librería `matplotlib` y la implementación en scikit-learn de *k-means*.

In [7]:
```
# Lista donde alojaremos los distintos resultados para el error cuadrático (SSE).
SSE = []


# Calcula los SSE para diferentes valores de k.
for k in range(1, 11):
    # Definimos el método kmeans para el valor de k correspondiente en cada iteración.
    kmeans = cluster.KMeans(k)
    # Aplicamos el método definido para ese valor k a nuestros datos.
    kmeans.fit(X)
    # Calculamos el SSE (inertia) del resultado y guardamos el resultado.
    SSE.append(kmeans.inertia_)


# Visualizamos la regla del codo.
plt.plot(range(1, 11), SSE, marker='o')
plt.title('Regla del Codo para X_blobs')
plt.xlabel('Número de clusters (k)')
plt.ylabel('SSE')
```

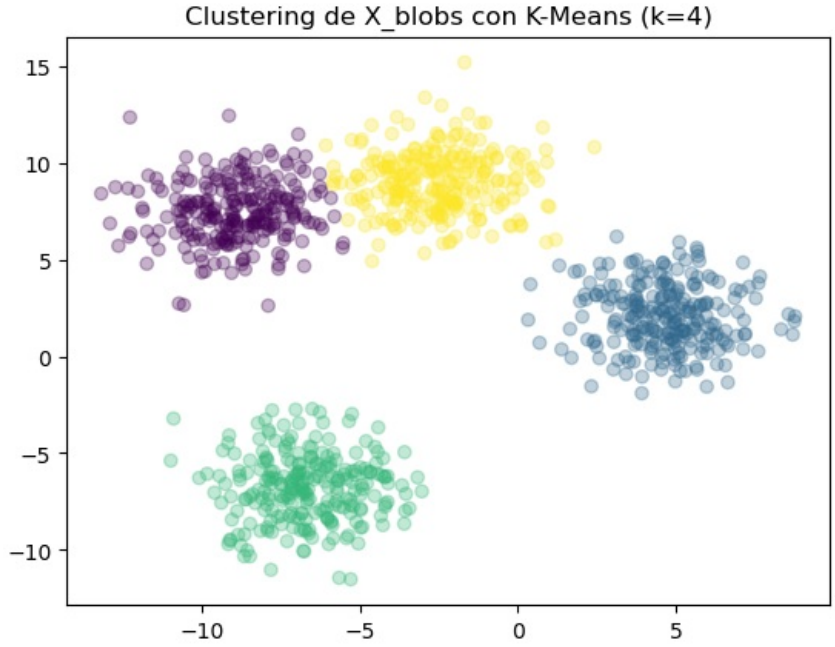
```
plt.show()
```



Claramente el valor más adecuado para k es $k = 4$ ya que a partir de ese punto las variaciones de la suma de los errores cuadráticos (*SSE*) no varían de forma notoria. En este caso no es necesario mejorar la elección de k.

Aplicamos el método K-means con 4 *clusters* y visualizamos los grupos en el dataset *Blobs*.

In [8]:
```
# Definimos el método k-means para k=4 y lo aplicamos sobre los datos.
kmeans = cluster.KMeans(4)
kmeans.fit(X)

# Obtiene las etiquetas de los clusters.
etiquetas = kmeans.labels_

# Visualiza los datos coloreados por cluster
plt.scatter(X[:, 0], X[:, 1], c=etiquetas, cmap='viridis', alpha=0.3)
plt.title('Clustering de X_blobs con K-Means (k=4)')
plt.show()
```



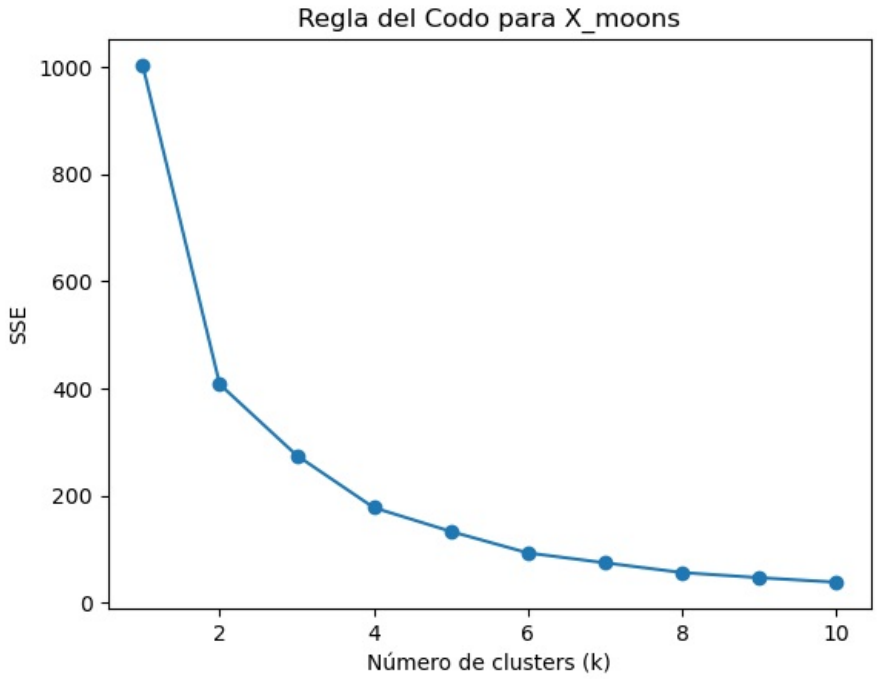
Dataset *moons*.

In [9]:
```
X, y = X_moons, y_moons
```

Implementación: cálculo y visualización de la regla del codo en el dataset Moons.

In [10]:
```
# Lista donde alojaremos los distintos resultados para el error cuadrático (SSE).
```

```
SSE = []


# Calcula los SSE para diferentes valores de k.
for k in range(1, 11):
    # Definimos el método kmeans para el valor de k correspondiente en cada iteración.
    kmeans = cluster.KMeans(k)
    # Aplicamos el método definido para ese valor k a nuestros datos.
    kmeans.fit(X)
    # Calculamos el SSE (inertia) del resultado y guardamos el resultado.
    SSE.append(kmeans.inertia_)


# Visualizamos la regla del codo.
plt.plot(range(1, 11), SSE, marker='o')
plt.title('Regla del Codo para X_moons')
plt.xlabel('Número de clusters (k)')
plt.ylabel('SSE')
plt.show()
```



Siguiendo el criterio de la regla del codo estaríamos tentados a determinar que el valor más adeucado para k es nuevamente $k = 4$ o incluso $k = 6$. Sabemos el valor correcto sería $k = 2$ pero en una situación real es muy probable que no contemos con esa información de antemano.

Nuestra alternativa sería evaluar el valor óptimo de k con otro método. Por ejemplo, con el método del *Shilhouette Score* que cuantifica como de bien están separados los distintos *clusters* entre sí con la siguiente fórmula.

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}$$

Donde:

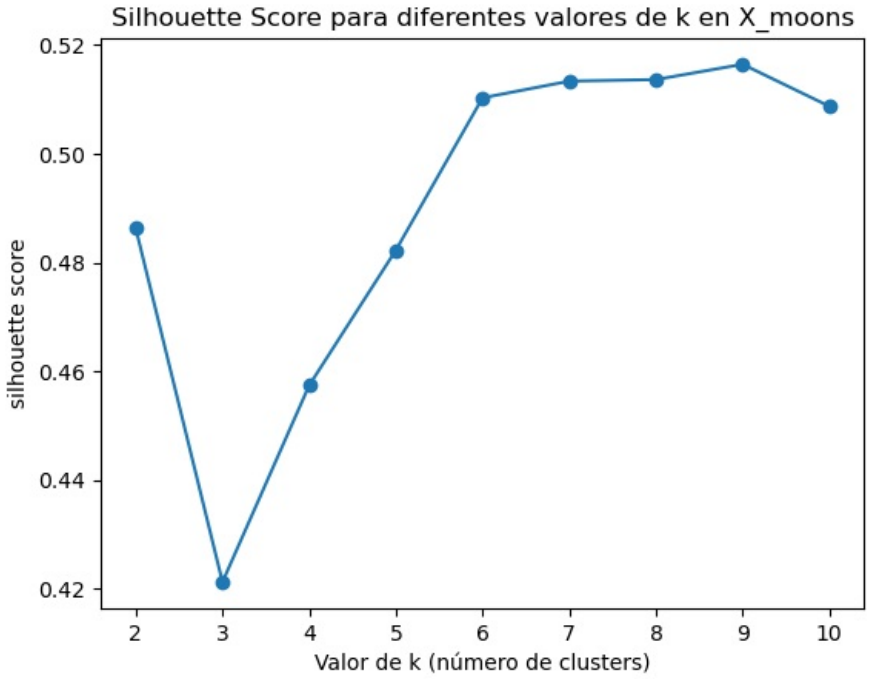
- $a(i)$: Es la distancia media del punto i a los demás puntos del mismo *cluster*. Mide la cercanía del punto con respecto a los demás dentro de su propio *cluster*.
- $b(i)$: Es la distancia media del punto i a los puntos del *cluster* más cercano al que i no pertenece. Mide cuán separado está el punto del *cluster* vecino más cercano.

El valor $s(i)$ varía entre -1 y 1. Un valor cercano a 1 nos dice que el punto está bien ajustado a su propio *cluster* pero mal ajustado a *clusters* vecinos. Un valor cercano a -1 nos indica que el punto podría estar en el *cluster* incorrecto. El promedio de $s(i)$ para todos los puntos de un conjunto de datos, proporciona una medida general de la calidad del agrupamiento.

Un *silhouette score* promedio alto indica una buena separación entre *clusters* mientras que un valor promedio bajo indica que estos pueden estar superpuestos entre sí. Por ello, la estrategia a seguir con *k-means* se basa en buscar el valor k que maximiza el *shilhouette score* promedio ya que esto nos indica una mejor partición de los datos en *clusters* mejor definidos para ese valor k.

In [11]:
```
# Importamos la función silhouette_score.
from sklearn.metrics import silhouette_score


# Rango de valores de k a probar.
k_values = range(2, 11)
```

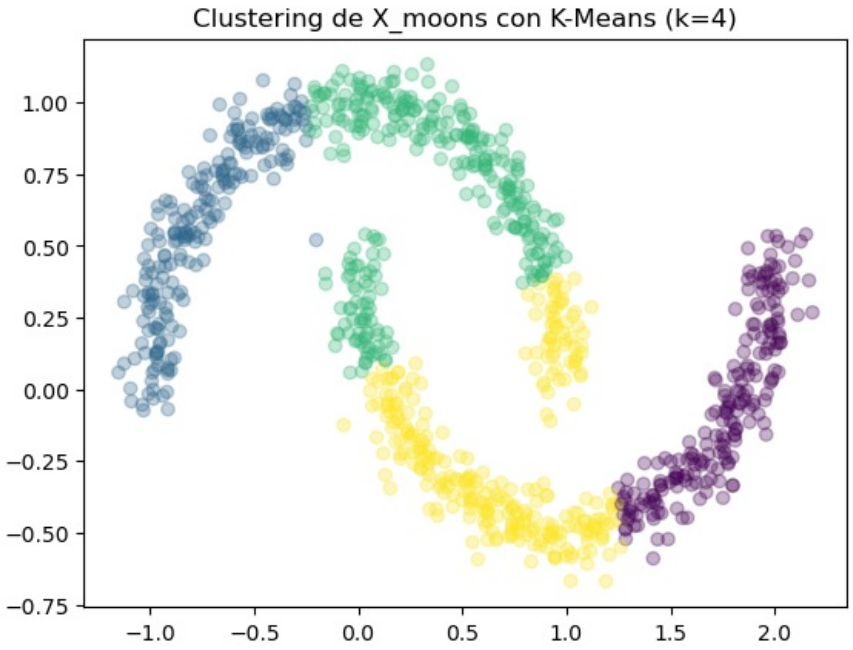
```
# Lista para almacenar los valores de silhouette score.
silhouette_scores = []


# Calcula el promedio de silhouette score para distintos valores de k.
for k in k_values:
    # Aplicamos k-means.
    kmeans = cluster.KMeans(n_clusters=k)
    # Buscamos las etiquetas de los clusters con la función fit_predict.
    etiquetas = kmeans.fit_predict(X)
    # Calculamos el silhouette score promedio.
    silhouette_avg = silhouette_score(X, etiquetas)
    # Guardamos el resultado obtenido en esta iteración.
    silhouette_scores.append(silhouette_avg)


# Buscamos el valor de k que maximiza el silhouette score.
best_k = k_values[np.argmax(silhouette_scores)]
best_score = silhouette_scores[np.argmax(silhouette_scores)]

# Mostramos el resultado del silhouette score para diferentes valores de k.
plt.plot(k_values, silhouette_scores, marker='o')
plt.title('Silhouette Score para diferentes valores de k en X_moons')
plt.xlabel('Valor de k (número de clusters)')
plt.ylabel('silhouette score')
plt.show()

# Indicamos el valor de k con el que obtuvimos el silhouette score más alto (el mejor candidato de k).
print(f"El mejor valor para K es {best_k} con un silhouette score de {best_score:.4f}")
```




```
El mejor valor para K es 9 con un silhouette score de 0.5164
```

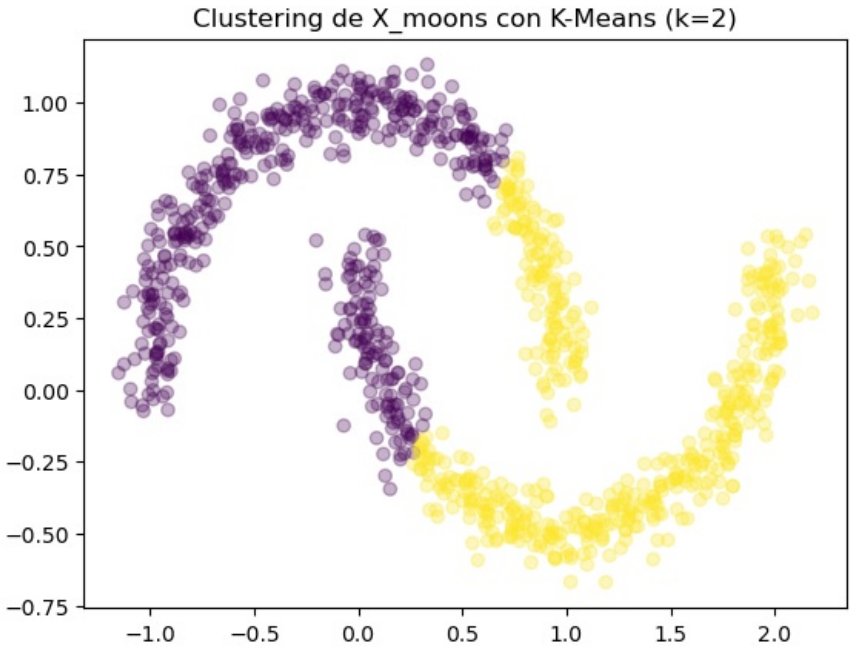
En este caso el resultado es incluso peor que mediante la "regla del codo".

In [12]:
```
# Definimos el método k-means para k=4 como nos sugería la regla del codo.
kmeans = cluster.KMeans(4)
kmeans.fit(X)

# Obtiene las etiquetas de los clusters.
etiquetas = kmeans.labels_

# Visualiza los datos coloreados por cluster
plt.scatter(X[:, 0], X[:, 1], c=etiquetas, cmap='viridis', alpha=0.3)
plt.title('Clustering de X_moons con K-Means (k=4)')
plt.show()
```



Vamos a probar con $k = 2$ para intentar replicar el resultado original.

In [13]:
```
# Definimos el método k-means para k=2 y lo aplicamos sobre los datos.
kmeans = cluster.KMeans(2)
kmeans.fit(X)

# Obtiene las etiquetas de los clusters.
etiquetas = kmeans.labels_

# Visualiza los datos coloreados por cluster
plt.scatter(X[:, 0], X[:, 1], c=etiquetas, cmap='viridis', alpha=0.3)
plt.title('Clustering de X_moons con K-Means (k=2)')
plt.show()
```



Para el caso de $k = 4$ como nos sugería el resultado observado para el cálculo del *SSE* y aplicando la "regla del codo", como era de esperar, no hemos obtenido una clasificación nada parecida a la real.

Luego, dado que conocemos el valor de $k = 2$ ya de antemano, hemos probado con dicho valor por curiosidad y comprobar si hubiésemos obtenido la distribución esperada, pero el resultado no ha sido el correcto. Parece ser que el etiquetado de nuestro algoritmo no respeta la distribución geométrica de los puntos y simplemente ha agrupado aquellos más cercanos entre sí (en términos de distancia euclídea). Al final no tenemos una etiqueta para cada "media luna" por separado (como en el enunciado), sino que tenemos una división global en la que cada etiqueta recoge puntos de ambas "medias lunas".

Podríamos intentar buscar o definir algún método que agrupe los datos teniendo en cuenta un criterio de equidistancia respecto los centroides y que estos a su vez se definan cerca del centro de cada "media luna", pero sería un esfuerzo absurdo ya que en una situación real no conocemos de antemano el valor de k ni tampoco sabemos como "deberían" agruparse los puntos. La motivación de usar *k-means* radica en eso mismo, en hacer agrupamiento de una muestra de la que *a priori* desconocemos el número de *clusters* (k) o

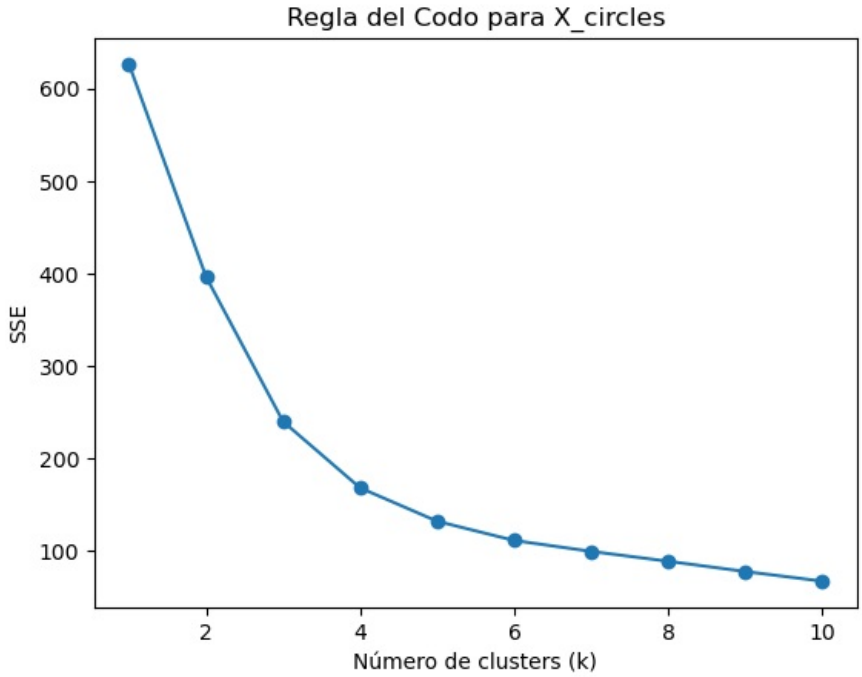
la distribución de las mismas (forma).

Definitivamente *k-means* no es el método óptimo para conjuntos de datos cuyos puntos sigan cierta geometría ordenada, en su lugar deberíamos emplear otros métodos como el *DBSCAN* cuyo fundamento no es la distancia entre puntos, sino en la densidad de puntos repartidos en el espacio.

Dataset *circles*:

In [14]:
```
X, y = X_circles, y_circles
```

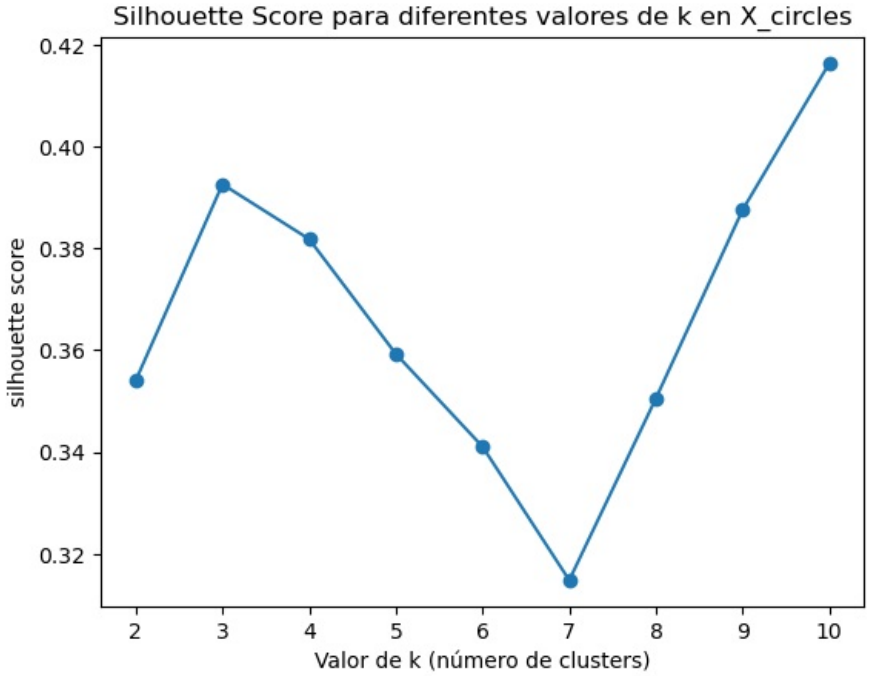
In [15]:
```
# Lista donde alojaremos los distintos resultados para el error cuadrático (SSE).
SSE = []


# Calcula los SSE para diferentes valores de k.
for k in range(1, 11):
    # Definimos el método kmeans para el valor de k correspondiente en cada iteración.
    kmeans = cluster.KMeans(k)
    # Aplicamos el método definido para ese valor k a nuestros datos.
    kmeans.fit(X)
    # Calculamos el SSE (inertia) del resultado y guardamos el resultado.
    SSE.append(kmeans.inertia_)


# Visualizamos la regla del codo.
plt.plot(range(1, 11), SSE, marker='o')
plt.title('Regla del Codo para X_circles')
plt.xlabel('Número de clusters (k)')
plt.ylabel('SSE')
plt.show()
```



Este caso es incluso peor que el anterior, sabemos que el valor de k debería ser $k = 2$ pero aquí vemos que según la "regla del codo" podríamos empezar a considerar como bueno un valor $k = 6$. Nuevamente, podríamos mejorar esto mediante el uso del método de *Shilhouette Score*.

In [16]:
```
# Importamos la función silhouette_score.
from sklearn.metrics import silhouette_score


# Rango de valores de k a probar.
k_values = range(2, 11)

# Lista para almacenar los valores de silhouette score.
silhouette_scores = []


# Calcula el promedio de silhouette score para distintos valores de k.
for k in k_values:
    # Aplicamos k-means.
    kmeans = cluster.KMeans(n_clusters=k)
    # Buscamos las etiquetas de los clusters con la función fit_predict.
    etiquetas = kmeans.fit_predict(X)
```

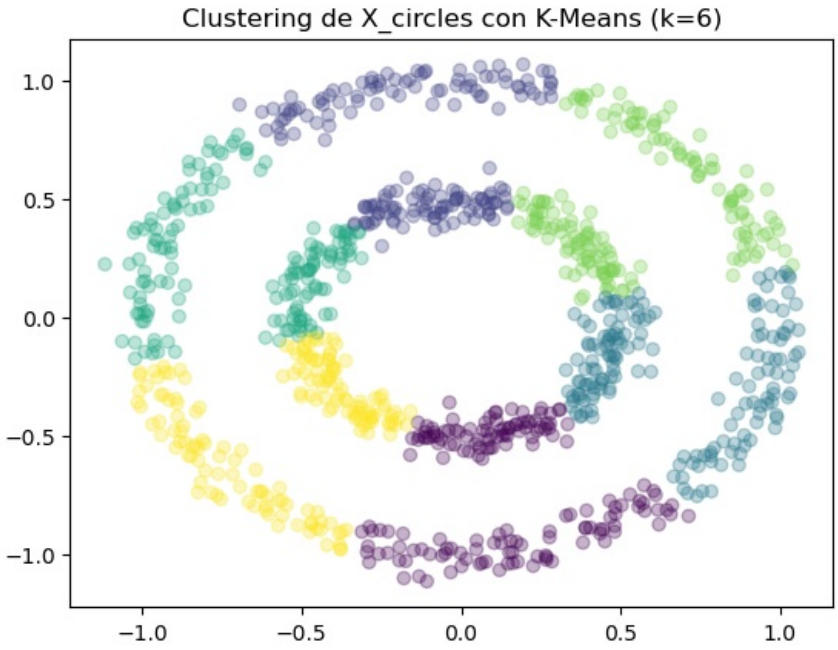
```
    # Calculamos el silhouette score promedio.
    silhouette_avg = silhouette_score(X, etiquetas)
    # Guardamos el resultado obtenido en esta iteración.
    silhouette_scores.append(silhouette_avg)


# Buscamos el valor de k que maximiza el silhouette score.
best_k = k_values[np.argmax(silhouette_scores)]
best_score = silhouette_scores[np.argmax(silhouette_scores)]

# Mostramos el resultado del silhouette score para diferentes valores de k.
plt.plot(k_values, silhouette_scores, marker='o')
plt.title('Silhouette Score para diferentes valores de k en X_circles')
plt.xlabel('Valor de k (número de clusters)')
plt.ylabel('silhouette score')
plt.show()

# Indicamos el valor de k con el que obtuvimos el silhouette score más alto (el mejor candidato de k).
print(f"El mejor valor para K es {best_k} con un silhouette score de {best_score:.4f}")
```




```
El mejor valor para K es 10 con un silhouette score de 0.4164
```

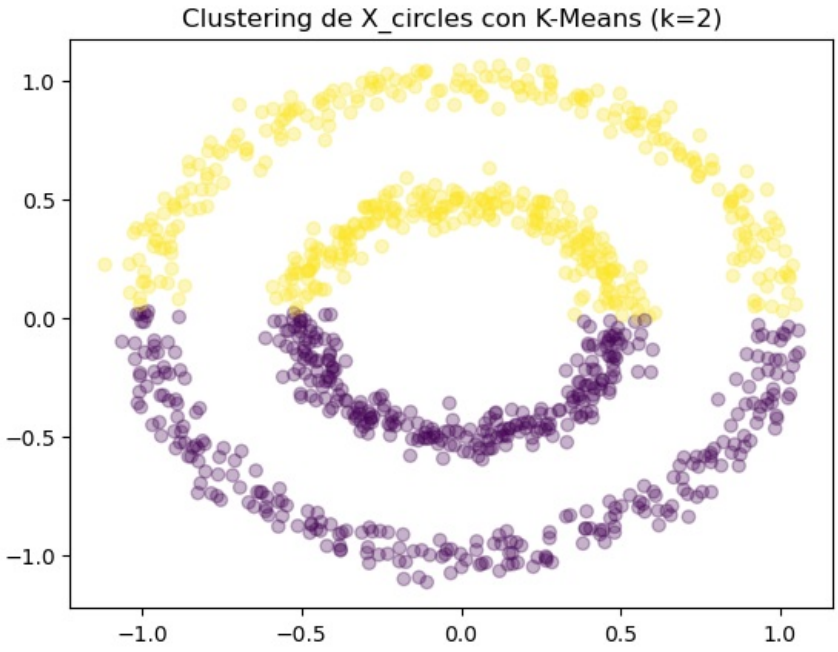
El resultado para k con el método *Silhouette Score* es incluso peor, ya que nos sugiere $k = 10$ pero se supone que es un método más óptimo para evaluar k.

In [17]:
```
# Definimos el método k-means para k=6 como nos sugería la regla del codo.
kmeans = cluster.KMeans(6)
kmeans.fit(X)

# Obtiene las etiquetas de los clusters.
etiquetas = kmeans.labels_

# Visualiza los datos coloreados por cluster
plt.scatter(X[:, 0], X[:, 1], c=etiquetas, cmap='viridis', alpha=0.3)
plt.title('Clustering de X_circles con K-Means (k=6)')
plt.show()
```



Ya que conocemos *a priori* el valor real de $k = 2$ comprobamos si el modelo agrupa correctamente los datos en comparación al resultado esperado.

In [20]:
```
# Definimos el método k-means para k=2 y lo aplicamos sobre los datos.
kmeans = cluster.KMeans(2)
kmeans.fit(X)

# Obtiene las etiquetas de los clusters.
etiquetas = kmeans.labels_

# Visualiza los datos coloreados por cluster
plt.scatter(X[:, 0], X[:, 1], c=etiquetas, cmap='viridis', alpha=0.3)
plt.title('Clustering de X_circles con K-Means (k=2)')
plt.show()
```



Ha sucedido exactamente lo mismo que en el caso de *X_moons*, en lugar de obtener dos *clusters* en forma de anillos concéntricos como tenemos en el enunciado (resultado esperado), hemos obtenido dos *clusters* en términos de distancia y ambos *clusters* contienen datos de ambos anillos a la vez.

El uso de *DBSCAN* podría solventar este problema al ser un método basado en la densidad de puntos en el espacio y no en la distancia de ellos entre sí.

1 b. Algoritmos basados en densidad: DBSCAN

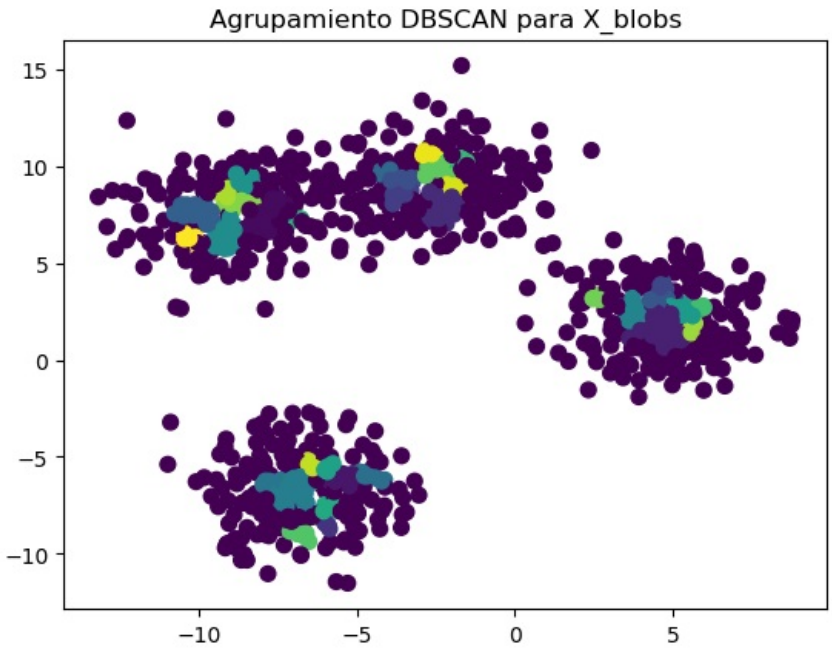
En este apartado aplicamos clustering por densidad como DBSCAN a los datasets anteriores para detectar los dos grupos subyacentes.

Ésta es una visualización intuitiva de su funcionamiento: https://www.youtube.com/watch?v=RDZUdRSDOok

Dataset *blobs*:

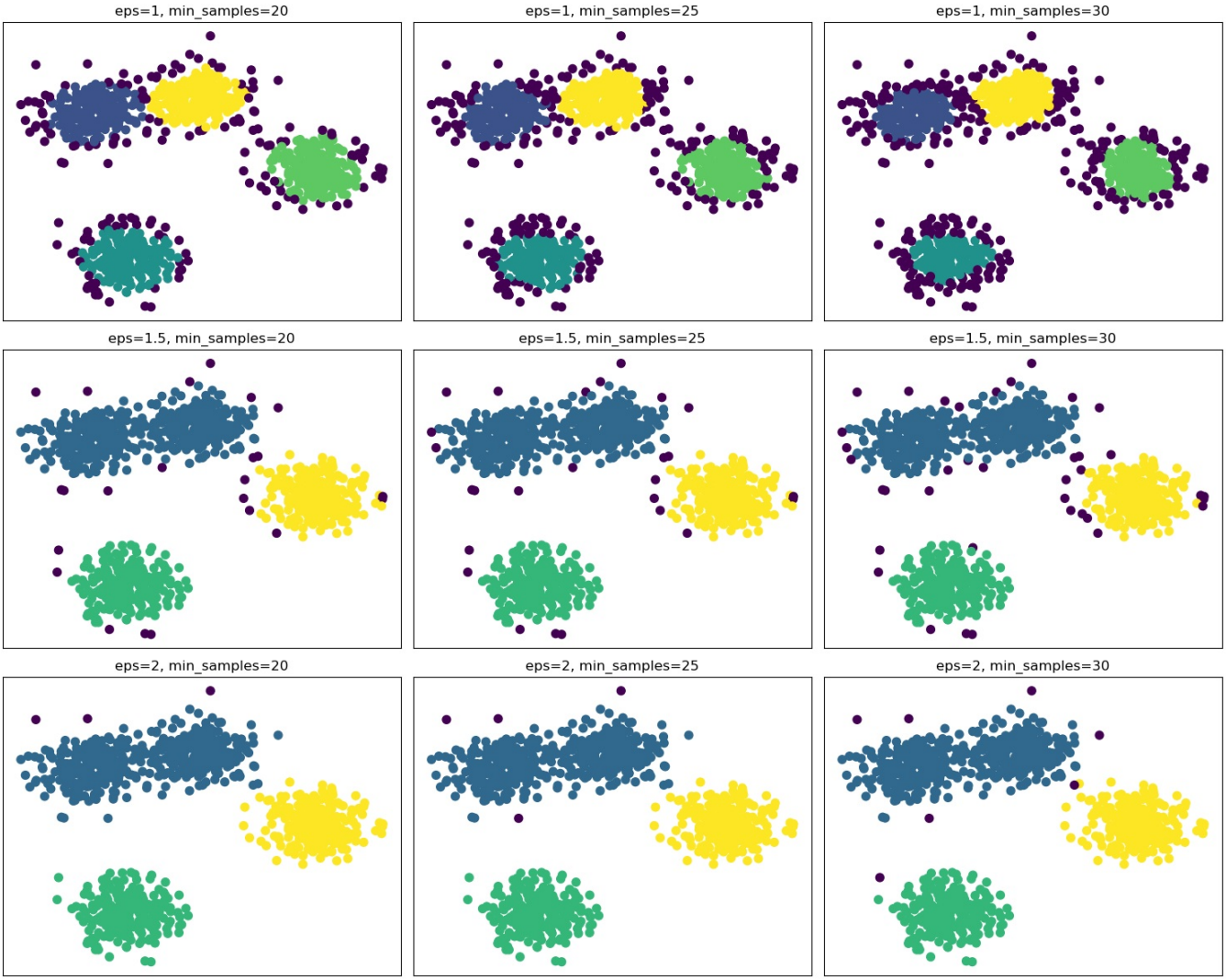
In [21]:
```
X, y = X_blobs, y_blobs
```

In [22]:
```
# Creamos el modelo DBSCAN.
dbscan = cluster.DBSCAN(eps=0.3, min_samples=5)

# Aplicamos el modelo para predecir los clusters.
y_pred = dbscan.fit_predict(X)

# Visualizamos el resultados
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', marker='o', s=50)
plt.title('Agrupamiento DBSCAN para X_blobs')
plt.show()
```



Vamos a probar con distintas parejas de valores *eps* y *min_samples*.

In [23]:
```
# Listamos los valores de los parámetros eps y min_samples que queremos probar.
eps_values = [1, 1.5, 2]
min_samples_values = [20, 25, 30]


# Configuramos la disposición de los gráficos dentro de una misma figura.
fig, axes = plt.subplots(nrows=len(eps_values), ncols=len(min_samples_values), figsize=(15, 12))


# Realizamos el clustering para cada pareja de valores de los valores eps y min_samples.
for i, eps in enumerate(eps_values):
    for j, min_samples in enumerate(min_samples_values):
        # Creamos el modelo DBSCAN para la pareja de parámetros eps y min_samples de esta iteración.
        dbscan = cluster.DBSCAN(eps=eps, min_samples=min_samples)
        # Aplicamos el modelo sobre el conjunto de datos.
        labels = dbscan.fit_predict(X)

        # Creamos el gráfico con los resultados de cada iteración
        axes[i, j].scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50)
        axes[i, j].set_title(f'eps={eps}, min_samples={min_samples}')
        axes[i, j].set_xticks([])
        axes[i, j].set_yticks([])

# Mostramos los resultados.
plt.tight_layout()
plt.show()
```



Haciendo alusión al video recomendado sobre el funcionamiento de *DBSCAN* vemos que:

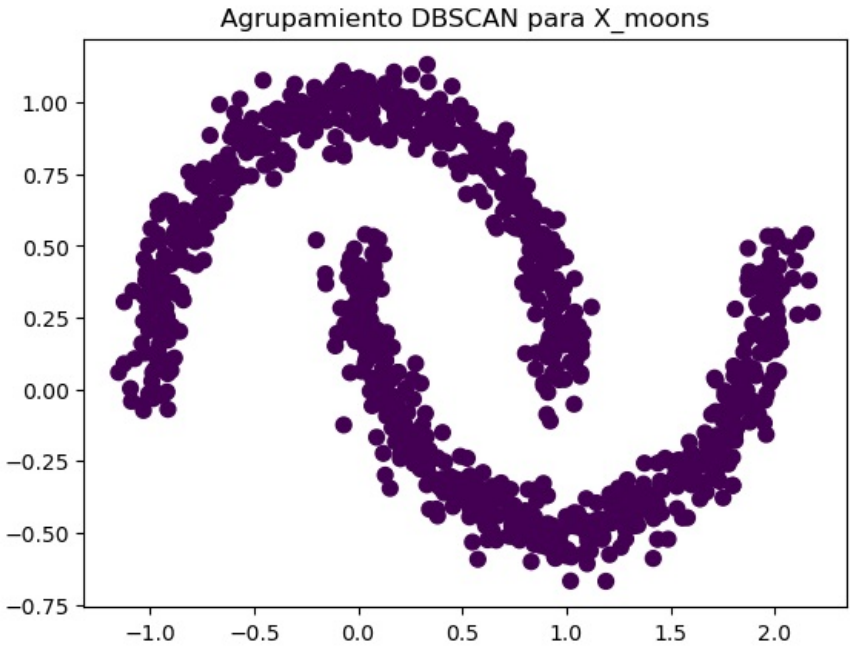
- El parámetro *eps* es el radio vecindad, es decir, situados sobre un punto, la distancia máxima dentro de la cual consideraríamos como "vecinos" al resto de puntos. Cuanto más alto sea este valor, el modelo tenderá a agrupar puntos más cada vez más distantes entre sí.
- El parámetro *min_samples* es, situados sobre un punto dado, el número mínimo de puntos que deben caer dentro del rango definido *eps* para considerar como núcleo (*core*) dicho punto. Cuanto más alto sea este valor, el modelo tenderá a definir *clusters* más densamente poblados, esto pa su vez logrará que los *clusters* no incluyan puntos muy dispersos.

En este caso, vemos que unos valores óptimos en cuanto a dejar el menor número de puntos sin clasificar serían $eps = 2$ y $min\_values = 25$. En contraparte, esta pareja de valores identifica como un único *cluster* los dos situados en la parte superior. Lo opuesto sería cualquiera de los casos con $eps = 1$ ya que en este caso si que identifica 4 *clusters* distintos pero deja mucho ruido sin clasificar. Podríamos intentar afinar más probando valores para eps entre 0.75 y 1.25 e ir viendo como afecta el uso de valores min_values menores que 10 o mayores a 30.

Dataset *moons*:

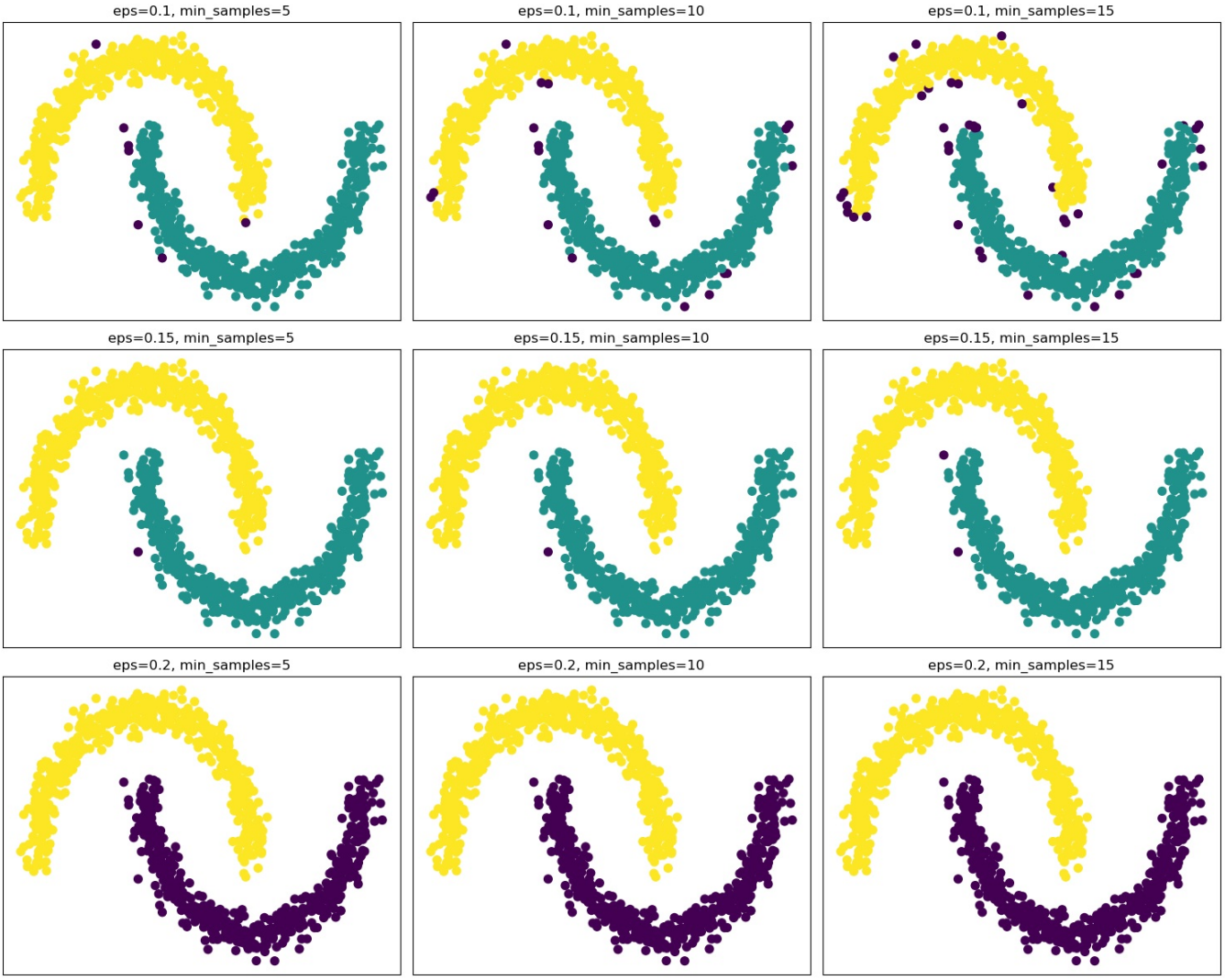
In [24]:
```
X, y = X_moons, y_moons
```

In [25]:
```
# Creamos el modelo DBSCAN.
dbscan = cluster.DBSCAN(eps=0.3, min_samples=5)

# Aplicamos el modelo para predecir los clusters.
y_pred = dbscan.fit_predict(X)

# Visualizamos el resultados
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', marker='o', s=50)
plt.title('Agrupamiento DBSCAN para X_moons')
plt.show()
```



Vamos a probar con distintas parejas de valores *eps* y *min_samples*.

In [26]:
```
# Listamos los valores de los parámetros eps y min_samples que queremos probar.
eps_values = [0.10, 0.15, 0.20]
min_samples_values = [5, 10, 15]


# Configuramos la disposición de los gráficos dentro de una misma figura.
fig, axes = plt.subplots(nrows=len(eps_values), ncols=len(min_samples_values), figsize=(15, 12))


# Realizamos el clustering para cada pareja de valores de los valores eps y min_samples.
for i, eps in enumerate(eps_values):
    for j, min_samples in enumerate(min_samples_values):
        # Creamos el modelo DBSCAN para la pareja de parámetros eps y min_samples de esta iteración.
        dbscan = cluster.DBSCAN(eps=eps, min_samples=min_samples)
        # Aplicamos el modelo sobre el conjunto de datos.
        labels = dbscan.fit_predict(X)

        # Creamos el gráfico con los resultados de cada iteración
        axes[i, j].scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50)
        axes[i, j].set_title(f'eps={eps}, min_samples={min_samples}')
        axes[i, j].set_xticks([])
        axes[i, j].set_yticks([])

# Mostramos los resultados.
plt.tight_layout()
plt.show()
```



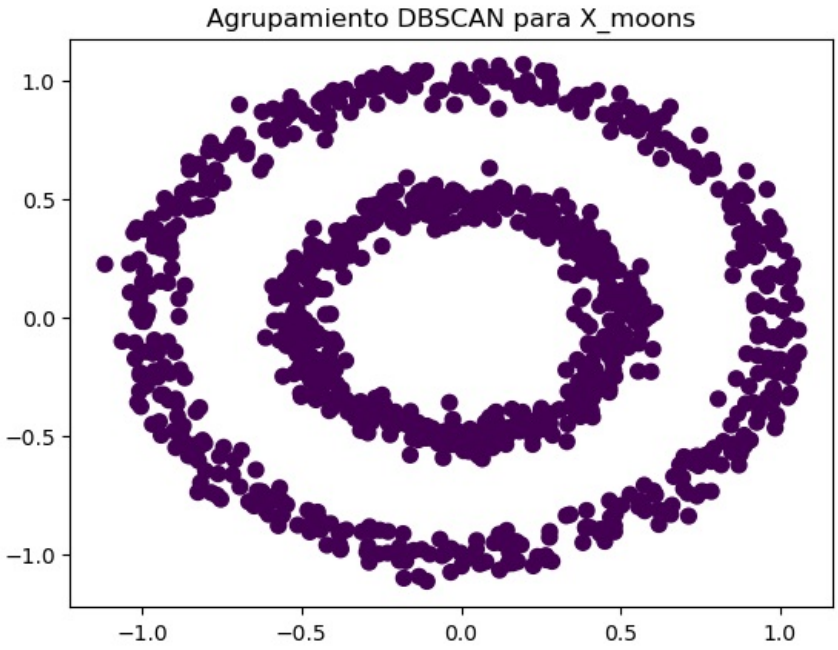
En este caso vemos que el factor determinante parece ser (*a priori*) el valor de eps ya que para valores mayores a $eps = 1.5$ el modelo sólo identifica un *cluster*. Quizá deberíamos probar con valores para eps entre 0.05 y 1.5 para poder analizar mejor como afecta el parámetro min_values.

Lo que sucede es que las "medias lunas" tienen una densidad de puntos muy grande en comparación al juego de datos de los *blobs* por lo que esta distribución de puntos parece ser más sensible al parámetro eps. Sin embargo, podríamos decir que los resultados para $eps = 1.5$ son perfectos, en especial aquellos con $min\_samples$ menor a 15 ya que sólo deja un único punto sin clasificar.

Dataset *circles*:

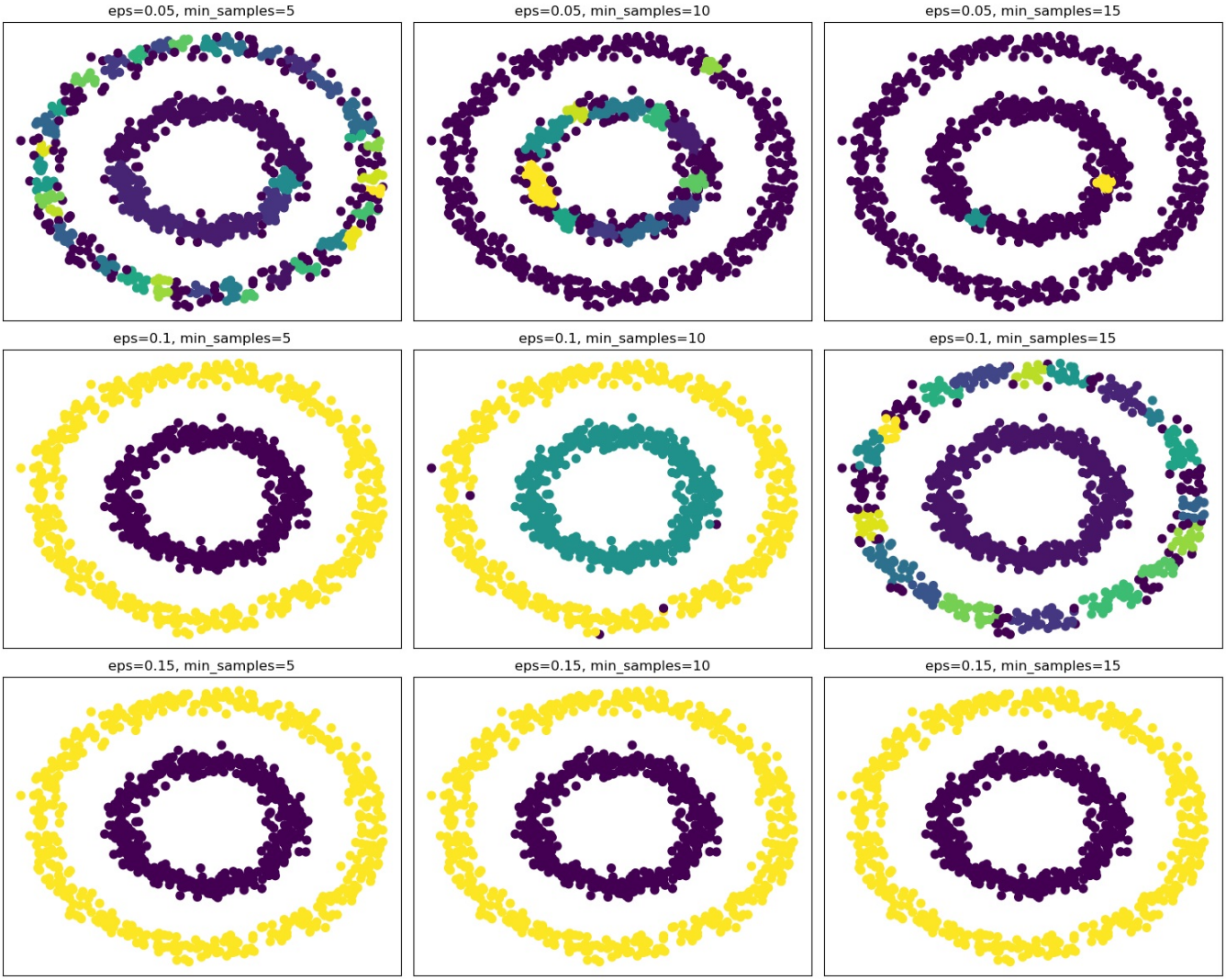
In [27]:
```
X, y = X_circles, y_circles
```

In [28]:
```
# Creamos el modelo DBSCAN.
dbscan = cluster.DBSCAN(eps=0.3, min_samples=5)

# Aplicamos el modelo para predecir los clusters.
y_pred = dbscan.fit_predict(X)

# Visualizamos el resultados
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', marker='o', s=50)
plt.title('Agrupamiento DBSCAN para X_moons')
plt.show()
```



Vamos a probar con distintas parejas de valores *eps* y *min_samples*.

In [29]:
```
# Listamos los valores de los parámetros eps y min_samples que queremos probar.
eps_values = [0.05, 0.10, 0.15]
min_samples_values = [5, 10, 15]


# Configuramos la disposición de los gráficos dentro de una misma figura.
fig, axes = plt.subplots(nrows=len(eps_values), ncols=len(min_samples_values), figsize=(15, 12))


# Realizamos el clustering para cada pareja de valores de los valores eps y min_samples.
for i, eps in enumerate(eps_values):
    for j, min_samples in enumerate(min_samples_values):
        # Creamos el modelo DBSCAN para la pareja de parámetros eps y min_samples de esta iteración.
        dbscan = cluster.DBSCAN(eps=eps, min_samples=min_samples)
        # Aplicamos el modelo sobre el conjunto de datos.
        labels = dbscan.fit_predict(X)

        # Creamos el gráfico con los resultados de cada iteración
        axes[i, j].scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50)
        axes[i, j].set_title(f'eps={eps}, min_samples={min_samples}')
        axes[i, j].set_xticks([])
        axes[i, j].set_yticks([])

# Mostramos los resultados.
plt.tight_layout()
plt.show()
```



En este caso parece que el conjunto de datos es bastante sensible tanto para el valor eps como para el valor de $min\_samples$. Podríamos decir que hemos encontrado una combinación ideal para $eps = 0.1$ y $min\_samples = 10$.

Cuando $min\_samples$ es menor a 10 hay un *cluster* sin reconocer por el modelo. Sin embargo cuando este valor es mayor a 10, el algoritmo detecta demasiados *clusters* distintos y sin sentido alguno bajo nuestro punto de vista.

Respecto al valor eps vemos como un valor inferior a 0.1 es muy errático y deja muchos datos sin agrupar como ruido. Para valores mayores a 0.1 siempre nos deja sin reconocer el *cluster* correspondiente al anillo interior.

2. Ejemplo práctico con aperturas de ajedrez: reducción de dimensionalidad (6 puntos)

En ajedrez existen multitud de aperturas y variantes. Te permiten planificar como posicionarás tus piezas, lo cual puede otorgar una gran ventaja durante el desarrollo de la partida. Hay tantas aperturas distintas (cada una de ellas con sus variantes) que puede ser dificil situarte, como puede apreciarse en el siguiente video del Maestro FIDE Luis Fernández.

Como muchas aperturas se parecen a otras, porque tienen planes similares, una buena forma para ubicarse es saber cuales se parecen entre sí. Y esa es la idea de este análisis.

Partiremos de un dataset de partidas de ajedrez en la plataforma lichess, que consta de los siguientes campos (se resaltan los útiles para el análisis):

- Game ID
- Rated (T/F)
- Start Time
- End Time
- **Number of Turns**
- Game Status
- Winner
- Time Increment
- White Player ID
- White Player Rating
- Black Player ID

- Black Player Rating
- **All Moves in Standard Chess Notation**
- Opening Eco (código estandar de aperturas)
- **Opening Name**
- **Opening Ply** (número de movimientos de la apertura de la partida)

Se carga el dataset en un dataframe de pandas:

In [3]:
```
df = pd.read_csv(r'C:\users\fjrc9\desktop\games.csv')
```

2 a. Preparación del dato

El primer paso al tratar con dato real es analizar el dato para comprender el dominio, y aplicar determinados filtrados en base a la lógica de tu tarea.

In [4]:
```
# Creamos un nuevo dataframe (df_filtrado) en el cual filtraremos el dataframe original (df).
df_filtrado = df[df['opening_ply'] >= 4]
```

In [5]:
```
# Vamos a comprobar que lo hemos hecho bien.
print('El dataframe original tiene una longitud de: {} muestras'.format(len(df)))
print('El dataframe filtrado tiene una longitud de: {} muestras'.format(len(df_filtrado)))
print('Tras filtrar los datos hemos reducido en {}'
      ' el número de muestras del dataframe original'.format(len(df)-len(df_filtrado)))
```

```
El dataframe original tiene una longitud de: 20058 muestras
El dataframe filtrado tiene una longitud de: 12536 muestras
Tras filtrar los datos hemos reducido en 7522 el número de muestras del dataframe original
```

In [6]:
```
# Comprobamos rápidamente los posibles valores para opening_ply en cada dataframe.
print('Los posibles valores de "opening_ply" para el dataframe original son:')
print(df['opening_ply'].unique(), '\n')
print('Los posibles valores de "opening_ply" para el dataframe filtrado son:')
print(df_filtrado['opening_ply'].unique())
```

```
Los posibles valores de "opening_ply" para el dataframe original son:
[ 5  4  3 10  6  1  9  2  8  7 17 11 12 13 18 19 15 16 14 28 20 22 24]

Los posibles valores de "opening_ply" para el dataframe filtrado son:
[ 5  4 10  6  9  8  7 17 11 12 13 18 19 15 16 14 28 20 22 24]
```

In [7]:
```
# Suponemos que nos piden seguir filtrando respecto la reducción de muestras anterior.
df_filtrado = df_filtrado[df_filtrado['turns'] >= 2*df_filtrado['opening_ply']]

# Comprobamos.
print('Longitud del data frame: {}\n'.format(len(df_filtrado)))
print('Posibles valores de opening_ply:'
      '\n{}\n'.format(df_filtrado['opening_ply'].unique()))
print('Posibles valores de turns:'
      '\n{}\n'.format(df_filtrado['turns'].unique()))
print('El valor mínimo de entre todos los posibles de turns'
     ' es: {}'.format(min(df_filtrado['turns'].unique())))
```

```
Longitud del data frame: 12272

Posibles valores de opening_ply:
[ 5  4 10  6  9  8  7 17 11 12 18 13 19 15 16 14 20 22 24 28]

Posibles valores de turns:
[ 13  16  95  33  66 119  38  31  43  52 101  14  36  69  54  53  64  21
  19  28  37  32  35  41  30  44  29  25 131  47  57  39  20  51  62 113
  75 135  81  46  91 137  18  40 111  90  49  50  59  26  23 118 120  24
  58  97  12  72  77  65  73 125  76  70  34  63 107  80 105 144 158 117
 110 124 106  15  42  60 159  74 112 104 150  88  55  48  87 103  22 177
  99  67  71 115  83  89 116  27  98 114 108  92  56  45  96 123 129  85
  17  94  79 195 122  11 121  84  61  86  93 148 133  78 100 102  82 178
 153  68 154 160 141 132 204 140   9 128   8  10 161 127 109 174 157 143
 170 171 145 179 149 173 208 130 136 134 169 155 147 126 139 146 138 190
 167 168 156 142 210 176 172 188 152 184 162 221 151 166 164 165 175 189
 349 182 198 218 207 216 200 181 180 222 209 185 259 212 163 183 187 201]

El valor mínimo de entre todos los posibles de turns es: 8
```

Si el nombre de la apertura (campo opening_name) contiene el caracter "|" ignoramos todo el texto que le sigue.

In [8]:
```
'''
Aplicamos el método split en una función lambda sobre
los atributos del campo "opening_name" del dataframe
de modo que nos quedamos con el texto previo al carácter
"|" en aquellos casos que lo contenga.
'''
```

```
# Hacemos el filtrado.
df_filtrado['opening_name'] = df_filtrado['opening_name'].apply(
            lambda x: x.split('|')[0].strip() if '|' in x else x)
```

Filtramos sólo las aperturas (por su nombre, campo opening_name) que hayan sido usadas en al menos 40 partidas. Para tener un mínimo de muestras de ese tipo, eliminando muchas aperturas apenas usadas.

In [9]:
```
# Contamos la frecuencia de cada valor de opening_name.
conteo_aperturas = df_filtrado['opening_name'].value_counts()

# Comprobamos previo al filtrado.
print('Los posibles valores para la frecuencia de los valores'
      ' de opening_name son:\n\n{}\n'.format(conteo_aperturas.unique()))
print('Y el valor mínimo de esas frecuencias es:'
      ' {}'.format(min(conteo_aperturas.unique())))
```

```
Los posibles valores para la frecuencia de los valores de opening_name son:

[268 254 236 195 193 184 180 179 178 169 163 156 154 136 135 126 123 121
 118 117 116 115 110 105 104 100  93  87  85  83  80  79  78  75  74  73
  71  69  63  60  59  58  56  53  52  50  49  48  47  46  45  44  43  42
  41  40  39  38  36  35  34  33  32  31  30  29  28  27  26  25  24  23
  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5
   4   3   2   1]

Y el valor mínimo de esas frecuencias es: 1
```

In [10]:
```
# Filtramos para quedarnos solo con los registros que se repiten al menos 40 veces
df_filtrado = df_filtrado[df_filtrado['opening_name'].isin(conteo_aperturas[conteo_aperturas >= 40].index)]
```

In [11]:
```
# Contamos la frecuencia de cada valor de opening_name.
conteo_aperturas = df_filtrado['opening_name'].value_counts()

# Comprobamos previo al filtrado.
print('Después de filtrar: Los posibles valores para la frecuencia de los valores'
      ' de opening_name son:\n\n{}\n'.format(conteo_aperturas.unique()))
print('Después de filtrar: El valor mínimo de esas frecuencias es:'
      ' {}'.format(min(conteo_aperturas.unique())))
```

```
Después de filtrar: Los posibles valores para la frecuencia de los valores de opening_name son:

[268 254 236 195 193 184 180 179 178 169 163 156 154 136 135 126 123 121
 118 117 116 115 110 105 104 100  93  87  85  83  80  79  78  75  74  73
  71  69  63  60  59  58  56  53  52  50  49  48  47  46  45  44  43  42
  41  40]

Después de filtrar: El valor mínimo de esas frecuencias es: 40
```

AVISO: ¡no es necesario saber interpretar correctamente la notación algebraica para el desarrollo de la práctica!

Los movimientos en ajedrez se pueden transcribir de distintas maneras, la más popular es la notación algebraica. En este caso, la notación algebraica empleada es la inglesa.

En concreto, el campo **moves** tiene los movimientos alternos tanto de las blancas como de las negras. Por tanto, la secuencia:

```
e4 c5 Nf3 Qa5...
```

Se interpretaría como:

- Las blancas mueven el peón de rey a la casilla e4.
- Las negras responden con el peón a c5.
- Las blancas mueven el caballo (N) a la casilla f3.
- Las negras mueven su dama a la casilla a5.

Y así alternan movimientos hasta el final de la partida (victoria, tablas, abandono o quedarse sin tiempo).

Es importante tener en cuenta que el campo **moves** es de tipo *string*, por lo que será necesario dividirlo por el separador (espacio) para tener cada movimiento por separado. Útil para los siguientes pasos.

Creamos una columna llamada "white_moves" que contenga una lista con sólo los movimientos de la apertura del participante blanco (los impares del campo moves y tantos como indique opening_ply).

In [12]:
```
# Usamos la función split para dividir los movimientos individuales como sugiere el enunciado.
# Los guardarmos en una nueva columna por separado y así no perdemos la columna original moves.
df_filtrado['moves_list'] = df_filtrado['moves'].apply(lambda x: x.split())

# Creamos la columna white_moves con los movimientos del jugador blanco.
# Además, le indicamos que sólo guarde los movimientos de apertura con el valor de opening_ply.
df_filtrado['white_moves'] = df_filtrado.apply(lambda row: row['moves_list'][0::2][:row['opening_ply']], axis=1
```

In [13]:
```
print(df_filtrado['white_moves'])
```

```
1                              [d4, e4, f4, dxe5]
4                           [e4, Nf3, d4, d5, a3]
8                     [e4, Bc4, Nf3, d3, Qxf3, h3]
9                           [e4, exd5, Nc3, Be2]
11       [e4, d4, e5, c3, Nf3, Be3, Nbd2, cxd4, Rb1]
                            ...
20026                          [e4, Nf3, Bc4, d4]
20034                [c4, Nc3, Nf3, d4, Nxd4, Qxd4]
20038                          [c4, Nc3, Nd5, Nf3]
20044                       [e4, d4, e5, Nf3, Bb5]
20048                      [e4, d4, exd5, Nc3, a3]
Name: white_moves, Length: 7044, dtype: object
```

Para comparar las aperturas entre sí usaremos los movimientos empleados en ella sólo por parte del jugador blanco (sólo hay nombre de la apertura del jugador blanco). Por simplicidad, podemos ignorar su orden, por lo que podemos usar la estrategia de bag of words. Generando, a partir del campo **white_moves** un nuevo dataset con tantas dimensiones como posibles movimientos con un 1 si se ha realizado durante la apertura y un 0 si no se ha realizado.

Podemos crear nuevas columnas a partir de valores con el método get_dummies() de pandas.

In [14]:
```
# Obtenemos un conjunto con la notación de cada movimiento registrado en el campo white_moves
# para generar este conjunto iteramos a nivel externo cada lista de movimientos en white_moves
# y para cada lista de movimientos de white_moves volvemos a iterar sobre cada movimiento individual.
# Luego con la función set creamos un conjunto cuya cualidad es eliminar duplicados y así tenemos
# una lista de movimientos únicos.
movimientos_unicos = set(move for moves_list in df_filtrado['white_moves'] for move in moves_list)

# Crear un nuevo DataFrame con las columnas binarias para cada movimiento usando get_dummies
df_blancas = pd.get_dummies(df_filtrado['white_moves'].apply(pd.Series).stack()).groupby(level=0).sum()

# Mostrar el nuevo DataFrame con las columnas binarias
print(df_blancas)
```

```
       Ba2  Ba3  Ba4  Bb1  Bb2  Bb3  Bb5  Bb5+  Bb6  Bc1  ...  gxf7+  gxh3  \
1        0    0    0    0    0    0    0     0    0    0  ...      0     0
4        0    0    0    0    0    0    0     0    0    0  ...      0     0
8        0    0    0    0    0    0    0     0    0    0  ...      0     0
9        0    0    0    0    0    0    0     0    0    0  ...      0     0
11       0    0    0    0    0    0    0     0    0    0  ...      0     0
...    ...  ...  ...  ...  ...  ...  ...   ...  ...  ...  ...    ...   ...
20026    0    0    0    0    0    0    0     0    0    0  ...      0     0
20034    0    0    0    0    0    0    0     0    0    0  ...      0     0
20038    0    0    0    0    0    0    0     0    0    0  ...      0     0
20044    0    0    0    0    0    0    1     0    0    0  ...      0     0
20048    0    0    0    0    0    0    0     0    0    0  ...      0     0

       gxh5  h3  h4  h5  hxg3  hxg4  hxg5  hxg6
1         0   0   0   0     0     0     0     0
4         0   0   0   0     0     0     0     0
8         0   1   0   0     0     0     0     0
9         0   0   0   0     0     0     0     0
11        0   0   0   0     0     0     0     0
...     ...  ..  ..  ..   ...   ...   ...   ...
20026     0   0   0   0     0     0     0     0
20034     0   0   0   0     0     0     0     0
20038     0   0   0   0     0     0     0     0
20044     0   0   0   0     0     0     0     0
20048     0   0   0   0     0     0     0     0

[7044 rows x 379 columns]
```

2 b. Reducción de dimensionalidad

En este punto tienes un dataset con tantas filas como partidas y tantas columnas como movimientos posibles efectuados en el conjunto de datos.

El problema es que ahora disponemos de muchas dimensiones, por lo que para visualizar los datos y comprobar si hay algún tipo de estructura es necesario reducir su dimensionalidad. Obteniendo un *embedding* (representación compacta) de las aperturas.

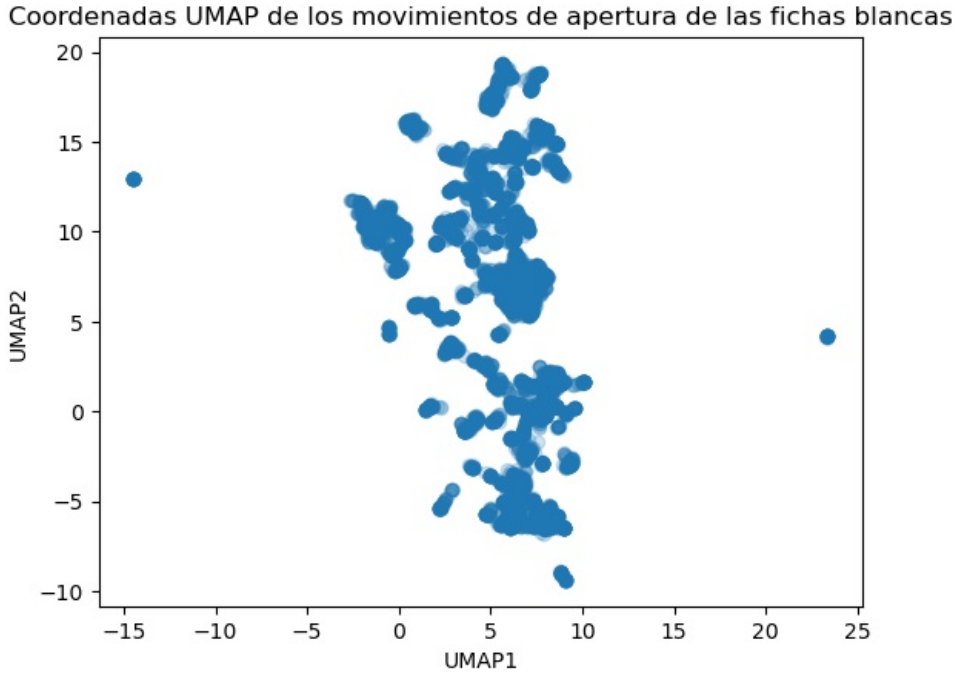
La reducción de dimensionalidad puede llevarse a cabo con métodos como PCA. Pero este método tiene la limitación de que sólo realiza proyecciones lineales. Por lo que otros métodos como t-SNE o UMAP pueden ofrecer mejores resultados.

In [15]:
```
# Creamos un modelo UMAP para reducir la dimensionalidad a dos dimensiones.
umap_model = umap.UMAP(n_components=2, random_state=42)
# Aplicamos el modelo al dataframe obtenido en el apartado anterior.
embedding = umap_model.fit_transform(df_blancas)

# Creamos un nuevo dataframe con las coordenadas obtenidas por el modelo UMAP.
df_umap = pd.DataFrame(embedding, columns=['UMAP1', 'UMAP2'])
```

```
# Mostramos el resultado de las coordenadas UMAP obtenidas.
print(df_umap)
```

```
         UMAP1      UMAP2
0     7.097513   1.256575
1     6.181747   0.260311
2     6.768152  14.665121
3     2.019637   9.343969
4     8.454629   1.865763
...        ...        ...
7039  2.604108   3.453873
7040  6.568399  -3.651820
7041  6.886249  -3.862518
7042  8.085086   1.286453
7043  9.231920  -2.925843

[7044 rows x 2 columns]
```

In [16]:
```
# Visualizamos las coordenadas obtenidas por el modelo UMAP.
plt.scatter(df_umap['UMAP1'], df_umap['UMAP2'], alpha=0.2)
plt.title('Coordenadas UMAP de los movimientos de apertura de las fichas blancas')
plt.xlabel('UMAP1')
plt.ylabel('UMAP2')
plt.show()
```



In [18]:
```
# Una vez hecha la representación, redefinimos el dataframe
# para volver al estado original del mismo.
df_umap = df_umap[['UMAP1', 'UMAP2']]
print(df_umap)
```

```
         UMAP1      UMAP2
0     7.097513   1.256575
1     6.181747   0.260311
2     6.768152  14.665121
3     2.019637   9.343969
4     8.454629   1.865763
...        ...        ...
7039  2.604108   3.453873
7040  6.568399  -3.651820
7041  6.886249  -3.862518
7042  8.085086   1.286453
7043  9.231920  -2.925843

[7044 rows x 2 columns]
```

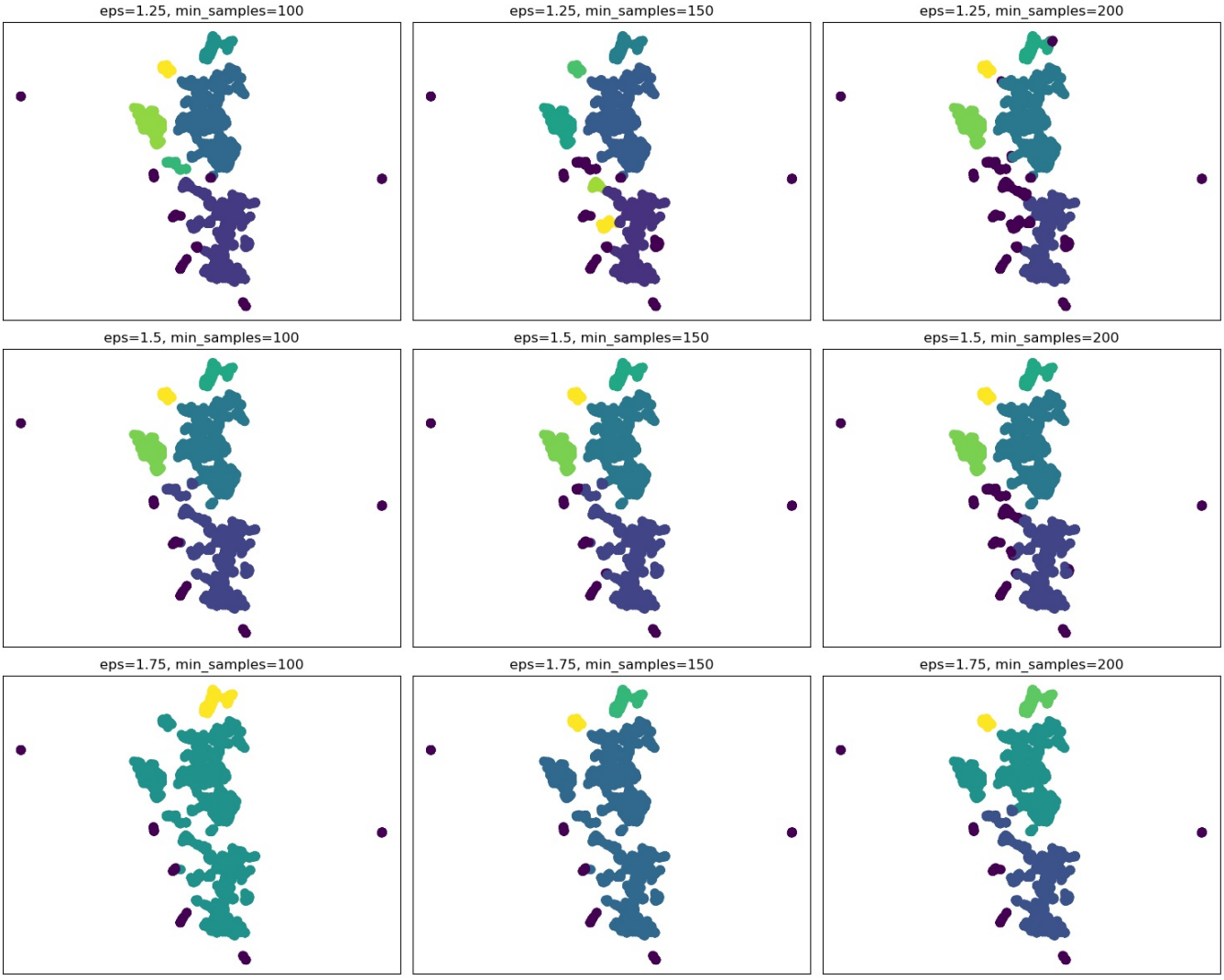
2 c. Clustering

Tras observar la estructura de las muestras en baja dimensionalidad.

Tenemos muchos puntos superpuestos en distintas zonas dando lugar regiones más densas que otras, además, estas regiones están próximas entre sí y con otras no tan densamente pobladas. Por este motivo lo ideal sería usar el método *__DBSCAN__* ya que se basa en la densidad de muestras y no en la distancia entre las mismas con respecto a ningún centro (como ocurre con *k-means*) ya que esto podría lugar a confundir la asignación de *clusters* a muestras cercanas de regiones muy densas.

Además, observamos ciertas estructuras geométricas, que aunque irregulares, si que presentan una tendencia como líneas entre los *blobs* generados en aquellas regiones más pobladas. Por ello, vamos a evitar *k-means* para que no nos ocurra lo mismo que cuando lo aplicamos a los conjuntos X_moons y X_circles del ejercicio anterior.

In [19]:
```
# Listamos los valores de los parámetros eps y min_samples que queremos probar.
eps_values = [1.25, 1.5, 1.75]
min_samples_values = [100, 150, 200]


# Configuramos la disposición de los gráficos dentro de una misma figura.
fig, axes = plt.subplots(nrows=len(eps_values), ncols=len(min_samples_values), figsize=(15, 12))


# Realizamos el clustering para cada pareja de valores de los valores eps y min_samples.
for i, eps in enumerate(eps_values):
    for j, min_samples in enumerate(min_samples_values):
        # Creamos el modelo DBSCAN para la pareja de parámetros eps y min_samples de esta iteración.
        dbscan = cluster.DBSCAN(eps=eps, min_samples=min_samples)
        # Aplicamos el modelo sobre el conjunto de datos.
        labels = dbscan.fit_predict(df_umap)

        # Creamos el gráfico con los resultados de cada iteración
        axes[i, j].scatter(df_umap['UMAP1'], df_umap['UMAP2'], c=labels, cmap='viridis', s=50)
        axes[i, j].set_title(f'eps={eps}, min_samples={min_samples}')
        axes[i, j].set_xticks([])
        axes[i, j].set_yticks([])

# Mostramos los resultados.
plt.tight_layout()
plt.show()
```



Vamos a quedarnos con el caso de $eps = 1.5$ y $min\_samples = 100$ a pesar de tener algo de ruido en la región central y dos *outliers* en los extremos izquierdo y derecho, vemos que agrupa y diferencia bastante bien 4 *clusters*.

In [20]:
```
# Creamos el modelo DBSCAN.
dbscan = cluster.DBSCAN(eps=1.5, min_samples=100)

# Aplicamos el modelo para predecir los clusters.
y_pred = dbscan.fit_predict(df_umap)
```

```
# Visualizamos el resultados.
plt.scatter(df_umap['UMAP1'], df_umap['UMAP2'], c=y_pred, cmap='viridis', marker='o', s=50)
plt.title('Agrupamiento DBSCAN para las coordenadas UMAP de los movimientos de apertura del'
          ' jugador blanco')
plt.text(-15, -5, 'eps=1.5\nmin_samples=100', bbox = {'facecolor': 'oldlace', 'alpha': 0.5,
                                                      'boxstyle': "square,pad=0.3", 'ec': 'red'})
plt.show()
```