Predicción lanzamiento SpaceX

Laboratorio práctico: Completa el laboratorio de predicción con Machine Learning

SpaceX anuncia lanzamientos de cohetes Falcon 9 en su sitio web con un costo de 62 millones de dólares; otros proveedores tienen costos superiores a 165 millones de dólares cada uno. Gran parte del ahorro se debe a que SpaceX puede reutilizar la primera etapa del cohete. Por lo tanto, si podemos determinar si la primera etapa aterrizará con éxito, podemos estimar el costo de un lanzamiento. Esta información puede ser utilizada si una compañía alterna quiere competir con SpaceX para un lanzamiento de cohetes.

En este laboratorio, crearás un pipeline de aprendizaje automático para predecir si la primera etapa aterrizará, utilizando los datos de los laboratorios previos.

Objetivos:

1. Realizar un análisis exploratorio de datos y determinar las etiquetas de entrenamiento.
2. Crear una columna para la clase (éxito/fallo).
3. Estandarizar los datos.
4. Dividir los datos en conjuntos de entrenamiento y prueba.
5. Encontrar el mejor hiperparámetro para:
    - Máquinas de soporte vectorial (SVM),
    - Árboles de clasificación,
    - Regresión logística.
6. Determinar cuál método funciona mejor utilizando los datos de prueba.

Índice:

Importar librerías

Instalación

```
import piplite
await piplite.install(['numpy'])
await piplite.install(['pandas'])
await piplite.install(['seaborn'])
```

Importación

```
# Pandas is a software library written for the Python programm
import pandas as pd
# NumPy is a library for the Python programming language, add
import numpy as np
# Matplotlib is a plotting library for python and pyplot give
import matplotlib.pyplot as plt
```

```
#Seaborn is a Python data visualization library based on matp
import seaborn as sns
# Preprocessing allows us to standarsize our data
from sklearn import preprocessing
# Allows us to split our data into training and testing data
from sklearn.model_selection import train_test_split
# Allows us to test parameters of classification algorithms a
from sklearn.model_selection import GridSearchCV
# Logistic Regression classification algorithm
from sklearn.linear_model import LogisticRegression
# Support Vector Machine classification algorithm
from sklearn.svm import SVC
# Decision Tree classification algorithm
from sklearn.tree import DecisionTreeClassifier
# K Nearest Neighbors classification algorithm
from sklearn.neighbors import KNeighborsClassifier
```

Funciones auxiliares

```
def plot_confusion_matrix(y,y_predict):
    "this function plots the confusion matrix"
    from sklearn.metrics import confusion_matrix

    cm = confusion_matrix(y, y_predict)
    ax= plt.subplot()
    sns.heatmap(cm, annot=True, ax = ax); #annot=True to anno
    ax.set_xlabel('Predicted labels')
    ax.set_ylabel('True labels')
    ax.set_title('Confusion Matrix');
    ax.xaxis.set_ticklabels(['did not land', 'land']); ax.yax
    plt.show()
```

Carga de archivos

```
from js import fetch
import io
```

```
URL1 = "https://xxxxx
resp1 = await fetch(URL1)
text1 = io.BytesIO((await resp1.arrayBuffer()).to_py())
data = pd.read_csv(text1)
```

En este caso se cargan los archivos de fuentes externas, si se descarga al entorno, se tendría que modificar usamos el archivo en local.

Segundo conjunto de datos

```
URL2 = 'https://xxxxxx
resp2 = await fetch(URL2)
text2 = io.BytesIO((await resp2.arrayBuffer()).to_py())
X = pd.read_csv(text2)
```

En este punto se revisa los dos archivos, ¿qué es la información que tenemos? usando para cada caso `data.head()` y `X.head()` respectivamente, podemos ver la cantidad de filas y columnas de cada uno asi como la información de valor que se puede utilizar. Los archivos al ser distintas estructuras hay que ir enlazando lo que se va a utilizar de cada uno de ellos en nuevas variables para generar conjuntos de datos sobre los que se pueda trabajar.

Trabajo con los datos

Dado que el objetivo es saber las probabilidades de éxito o fallo de los aterrizaje de la primera fase de los cohetes, hay que buscar la columna que corresponde con esa información en una de las dos tablas (de ahora en adelante estas tablas las llamaré data frame ya que es el término correcto utilizado en data, o df).

La lógica simplificada aplicada a esta tarea es, según las estadísticas de lanzamientos pasados hay variables que determinan el éxito o fracaso, una de las primeras cuestiones es determinar que variables pesan mas sobre este resultado. A partir de ahí, usando esas variables puedes predecir los próximos lanzamientos como serían, según esa influencia de cada uno de los distintos factores.

Hipótesis: término utilizado para asumir algo estadístico, donde H_0 es lo primero que asumes, basado en algo de lógica pero sin respaldo. *Ejemplo*:

- Si el lanzamiento es desde X lugar, va a fallar.
- Si el tiempo en llegar a X órbita es mayor que Y_t va a fallar.

(Esta sección me la invento, no son hipótesis reales de este caso, es para darle contexto y explicación simple al proceso de análisis)

Viendo los df creados y por experiencia de trabajar previamente con ellos, se sabe que la columna `'Class'` del df `data` es la columna que tiene los valores de éxito/fracaso en formato de 0 y 1

Variables: para hacer una predicción, cálculo, ecuación etc, siempre hay una variable "independiente" que conocemos y una variable "dependiente" que es la que queremos determinar o calcular. Las variables pueden ser numéricas (1,2... etc) o categóricas(blanco, negro, si, no... etc). En este caso y como se ha dicho antes, queremos saber si tiene éxito/fracaso el lanzamiento, y depende de los factores externos o propios del lanzamiento, como el lugar u orbita. Sabemos entonces que `'Class'` es la variable dependiente, con 0 o 1, nos dice si fue bien o mal, por lo que:

`'Class'` = Y

Todo lo demás, podría ser X (es lo que conocemos y recuerda, estamos determinando que depende de quién, quién influye y en qué medida para saber si es real o no esa H_0)

(Nuevamente, esta sección es un poco aclaratoria para que se entienda que se está haciendo, puede tener errores)

Variable dependiente

```
Y = data['Class'].to_numpy() #definiendo quien es Y
```

Del análisis previo en los df ya sabemos que la columna `'Class'` del df `data` es la columna que tiene los valores de éxito/fracaso, mientras que en otro df que ya hemos llamado X, tiene todos los datos del lanzamiento que se han medido o cuantificado durante estos.

Por supuesto las dos tablas están relacionadas, usualmente por un valor “llave” o código, indicador, etc que conecta que fila corresponde con cada lanzamiento. (Gestión de bases de datos, SQL, etc)

En este punto, hay un valor Y que depende de varios valores X, esos varios valores pueden ser de todo tipo, diversos, en término cotidiano: es una cesta de productos alimenticios, no todos se pueden mezclar, pero todos se pueden comer. Hay que estandarizarlos y ponerlos en su lugar y comerse cuando corresponda, no comes helado con arroz. Lo que haces es “**estandarizar**” que productos vas a mezclar y en qué momento lo vas a comer. (Quizás no sea el mejor ejemplo para explicar esto, pero espero que se entienda el punto)

Estandarización de variables independientes

```
transform = preprocessing.StandardScaler()
X = transform.fit_transform(X)
```

En este punto, tenemos una variable Y y un conjunto de variables X para comparar. Si comparamos todas las variables entre sí, no tenemos como comprobar o validar esas relaciones que vamos a tener. Vamos, que si usamos todos los datos para un mismo propósito, no tendremos datos para comparar si lo que hicimos o determinamos está bien o mal.

En matemática simple esto sería como tener una gráfica de una linea usando el típico sistema de coordenadas X;Y. Ponemos dos pares de (X;Y) y hacemos una linea.

Donde esa linea corresponde con la típica ecuación de:

$$Y = b + mX$$

Si usando cualquier valor X determinamos Y, siempre obtenemos el mismo valor, lo que es estadísticamente falso, porque no se está probando con nuevos valores, solo se está repitiendo el mismo cálculo existente, y tanto b como m son unas constantes ficticias para este conjunto de datos.

Para poder tener como comprobar esas constantes o como influyen las variables en el resultado, se divide todos los datos que tenemos. Una parte se usa para "estimar" y la otra parte para "comprobar" (de lo contrario hay que esperar mas lanzamientos para nuevos datos, produce pérdidas).

Entrenamiento y test del conjunto de datos

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, tes
```

Explicación:

1. `train_test_split(X, Y)`:
   - Divide los datos de entrada (`X`) y las etiquetas (`Y`) en dos subconjuntos: uno para entrenamiento y otro para prueba.
2. **Parámetros:**
   - `test_size=0.2`: El 20% de los datos se destina al conjunto de prueba.
   - `random_state=2`: Asegura que la división sea reproducible cada vez que se ejecuta el código.
3. **Etiquetas de los conjuntos:**
   - `X_train`: Datos de entrada para entrenamiento.
   - `X_test`: Datos de entrada para prueba.
   - `Y_train`: Etiquetas correspondientes al conjunto de entrenamiento.
   - `Y_test`: Etiquetas correspondientes al conjunto de prueba.

Resultado esperado:

Si tienes 100 filas en tus datos originales:

- **80 filas** estarán en los conjuntos de entrenamiento (`X_train` y `Y_train`).
- **20 filas** estarán en los conjuntos de prueba (`X_test` y `Y_test`).

De esta forma ya se tiene con qué se "entrena" y con qué se "comprueba"

```
Y_test.shape # ver el tamaño de las muestras
```

Regresión logística

```
# Definir el modelo de regresión logística
lr = LogisticRegression()

# Definir los hiperparámetros a optimizar
parameters = {'C': [0.01, 0.1, 1], 'penalty': ['l2'], 'solv
er': ['lbfgs']}

# Crear el objeto GridSearchCV con validación cruzada (cv=1
0)
logreg_cv = GridSearchCV(estimator=lr, param_grid=parameter
s, cv=10)

# Ajustar GridSearchCV a los datos de entrenamiento
logreg_cv.fit(X_train, Y_train)
```

Explicación:

1. `LogisticRegression()`:
   - Crea el modelo base de regresión logística.
2. `parameters`:
   - Define los hiperparámetros a explorar:
     - `C`: Parámetro de regularización.
     - `penalty`: Tipo de penalización (L2 para Ridge).
     - `solver`: Método de optimización (en este caso, `'lbfgs'`).
3. `GridSearchCV`:
   - `estimator=lr`: Modelo base que se optimizará.
   - `param_grid=parameters`: Conjunto de parámetros para probar.
   - `cv=10`: Validación cruzada con 10 particiones.
4. `logreg_cv.fit(X_train, Y_train)`:
   - Ajusta `GridSearchCV` con los datos de entrenamiento y busca la combinación óptima de parámetros.

Con esto, estamos mezclando la caja de alimentos que teníamos de antes, y la estamos organizando en distintas gavetas y armarios aleatoriamente y probando donde queda mejor acomodado, se repite varias veces hasta tener claro, donde va cada producto y cual puedo mezclar con cual, luego he probado a cocinar y comer, si esa distribución era cómoda y eficiente hasta determinar la distribución final.

Después de ejecutar el código, puedes acceder a los mejores parámetros encontrados y la puntuación correspondiente:

```
# Obtener los mejores parámetros
print("Mejores parámetros:", logreg_cv.best_params_)

# Obtener la mejor puntuación
print("Mejor puntuación:", logreg_cv.best_score_)
```

Esto ofrece unos valores estadísticos cuantificables que te da a entender si está bien o mal la organización.

En este punto, volviendo a los cohetes. Tenemos un modelo inicial creado para comenzar a predecir. Este modelo puede tener errores o no, además que no es la única metodología que se puede usar, por lo que se requiere saber, si con los dos conjuntos (prueba y entrenamiento) ¿Qué cantidad de aciertos o fallos ha cometido? determinando la calidad del modelo.

Precisión en los datos de prueba

```
# Calcular la precisión en los datos de prueba
test_accuracy = logreg_cv.score(X_test, Y_test)

# Mostrar el resultado
print("Precisión en los datos de prueba:", test_accuracy)
# Resultado 0.83333333334
```

Explicación:

1. `logreg_cv.score(X_test, Y_test)`:
    - Calcula la precisión del modelo optimizado en los datos de prueba.

- Usa el mejor modelo ajustado por `GridSearchCV`.

2. `test_accuracy`:
    - Almacena la precisión calculada.

Una precisión de **0.8333 (83.33%)** generalmente se considera **buena**, especialmente si:

1. **El conjunto de datos no está desequilibrado** (es decir, si las clases de éxito y fallo están representadas de manera relativamente equilibrada).
2. **El problema es complejo**, como predecir el aterrizaje exitoso de una etapa de cohete basada en múltiples factores.

Sin embargo, si el contexto requiere un alto nivel de precisión (por ejemplo, misiones espaciales críticas), un modelo con 83.33% puede no ser suficiente. Siempre es útil comparar esta precisión con:

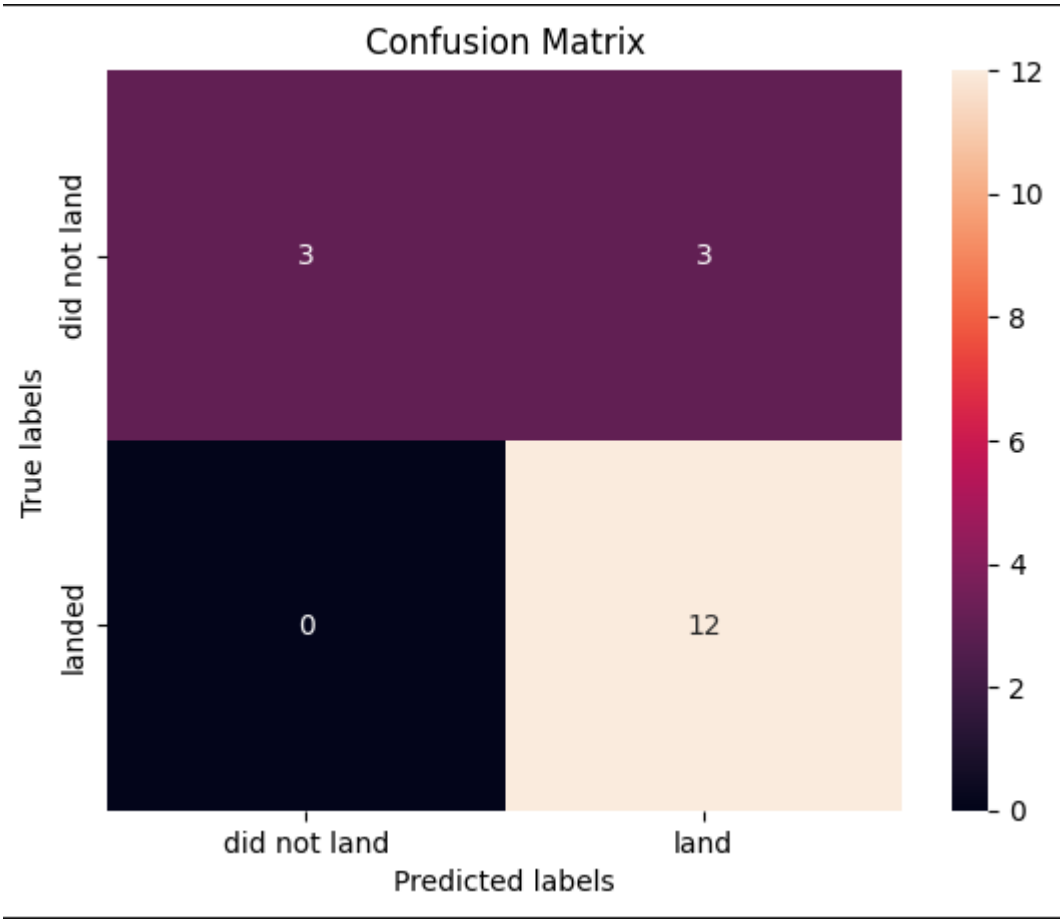
- Un modelo base (como un modelo que predice aleatoriamente o siempre la clase más común).
- Los resultados esperados o aceptables para el problema específico.

En resumen, **es un buen punto de partida**, pero se podría intentar mejorar con ajustes de hiperparámetros o probando otros modelos.

En este punto, lo mejor es ver de forma gráfica los resultados hasta este punto y asi evaluar con mayor facilidad para tomar decisiones reales.

Matriz de confusión

```
yhat=logreg_cv.predict(X_test)
plot_confusion_matrix(Y_test,yhat)
```



Examinando la matriz de confusión, vemos que la regresión logística puede distinguir entre las diferentes clases. Observamos que el problema son los falsos positivos.

Resumen:

- **Verdadero positivo (True Positive):** 12 (La etiqueta real es "aterrizado" y la etiqueta predicha también es "aterrizado").
- **Falso positivo (False Positive):** 3 (La etiqueta real es "no aterrizado", pero la etiqueta predicha es "aterrizado").

De aqui en adelante se repite el proceso con distintos modelos y parámetros para determinar cual es mejor para la predicción según los indicadores de salida y resultados obtenidos, algo quizás tedioso pero muy importante para valorar la metodología más útil para cada caso.

Máquina de soporte vectorial (SVM)

```
# Definir el modelo de SVM
svm = SVC()

# Definir los hiperparámetros a optimizar
parameters = {
    'kernel': ('linear', 'rbf', 'poly', 'sigmoid'),
    'C': np.logspace(-3, 3, 5),
    'gamma': np.logspace(-3, 3, 5)
}

# Crear el objeto GridSearchCV con validación cruzada (cv=1
0)
svm_cv = GridSearchCV(estimator=svm, param_grid=parameters,
cv=10)

# Ajustar GridSearchCV a los datos de entrenamiento
svm_cv.fit(X_train, Y_train)

# Mostrar los mejores parámetros y la mejor puntuación
print("Mejores parámetros:", svm_cv.best_params_)
print("Mejor puntuación en entrenamiento:", svm_cv.best_sco
re_)
```

Explicación:

1. `SVC()`:
   - Define el modelo base de máquina de soporte vectorial.
2. `parameters`:
   - `kernel`: Tipo de kernel a probar (`linear`, `rbf`, `poly`, `sigmoid`).
   - `C`: Parámetro de regularización, con valores entre `10^-3` y `10^3`.
   - `gamma`: Parámetro para el kernel no lineal, con valores entre `10^-3` y `10^3`.
3. `GridSearchCV`:
   - Busca la mejor combinación de hiperparámetros en el espacio definido por `parameters`.

- Usa validación cruzada con 10 particiones (`cv=10`).

4. `svm_cv.fit(X_train, Y_train)`:
   - Ajusta el modelo con los datos de entrenamiento.
5. `best_params_` y `best_score_`:
   - `best_params_`: Muestra la mejor combinación de parámetros.
   - `best_score_`: Muestra la mejor puntuación promedio obtenida durante la validación cruzada.

Resultados esperados:

Después de ejecutar el código, verás algo como:

```
Mejores parámetros: {'C': 1.0, 'gamma': 0.1, 'kernel': 'rb
f'}
Mejor puntuación en entrenamiento: 0.87
```

Esto indica que el modelo SVM está optimizado para predecir correctamente con los mejores parámetros encontrados. Volvemos a comprobar como se comportan las predicciones con este nuevo modelo como se hizo antes.

Precisión en los datos de prueba

```
# Calcular la precisión del modelo SVM en los datos de prue
ba
svm_test_accuracy = svm_cv.score(X_test, Y_test)

# Mostrar la precisión
print("Precisión en los datos de prueba:", svm_test_accurac
y)
#resultado 0.8333333333333334
```

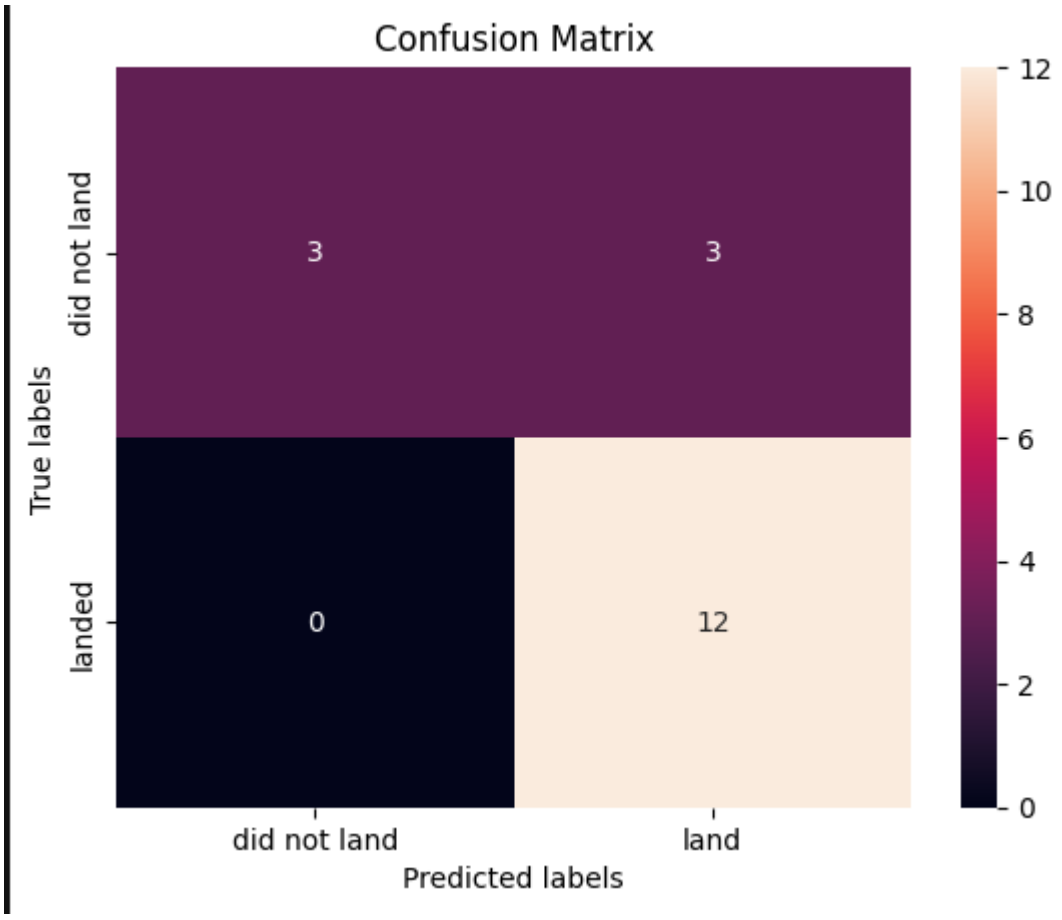
Explicación:

1. `svm_cv.score(X_test, Y_test)`:
   - Calcula la precisión del mejor modelo encontrado por `GridSearchCV` en el conjunto de prueba.
2. **Resultado:**

- La precisión se almacena en `svm_test_accuracy` y representa el porcentaje de predicciones correctas en el conjunto de prueba.

Matriz de confusión

```
yhat=svm_cv.predict(X_test)
plot_confusion_matrix(Y_test,yhat)
```



Se han obtenido los mismos resultados, por lo que se sigue buscando algún modelo que de otros resultados y se pueda comparar el más preciso para el caso de estudio.

Árboles de decisión

```
# Crear el modelo base de árbol de decisión
tree = DecisionTreeClassifier()

# Definir el espacio de búsqueda de hiperparámetros
```

```
parameters = {
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random'],
    'max_depth': [2 * n for n in range(1, 10)],
    'max_features': ['auto', 'sqrt'],  #auto da una adverte
ncia, no es util para arboles de decision
    'min_samples_leaf': [1, 2, 4],      # variante 'max_fea
tures': ['sqrt', 'log2'],
    'min_samples_split': [2, 5, 10]
}

# Crear el objeto GridSearchCV con validación cruzada (cv=1
0)
tree_cv = GridSearchCV(estimator=tree, param_grid=parameter
s, cv=10)

# Ajustar el modelo con los datos de entrenamiento
tree_cv.fit(X_train, Y_train)

# Mostrar los mejores parámetros y la mejor puntuación
print("Mejores parámetros:", tree_cv.best_params_)
print("Mejor puntuación en entrenamiento:", tree_cv.best_sc
ore_)
```

Explicación:

1. `DecisionTreeClassifier()`:
   - Define el modelo base de árbol de decisión.
2. `parameters`:
   - Contiene los posibles valores para optimizar los hiperparámetros:
     - `criterion`: Función para medir la calidad de una división (`gini` o `entropy`).
     - `splitter`: Método para dividir nodos (`best` o `random`).
     - `max_depth`: Máxima profundidad del árbol, calculada como 2 * n.
     - `max_features`: Número de características a considerar (`auto`, `sqrt`).

- `min_samples_leaf`: Número mínimo de muestras en una hoja.
- `min_samples_split`: Número mínimo de muestras necesarias para dividir un nodo.

3. `GridSearchCV`:
   - Busca la mejor combinación de hiperparámetros usando validación cruzada con 10 particiones (`cv=10`).
4. `tree_cv.fit(X_train, Y_train)`:
   - Ajusta el modelo de árbol de decisión utilizando los datos de entrenamiento.
5. `best_params_` y `best_score_`:
   - `best_params_`: Devuelve los mejores hiperparámetros encontrados.
   - `best_score_`: Devuelve la mejor puntuación promedio durante la validación cruzada.

Resultado esperado:

Después de ejecutar el código, verás algo como:

```
Mejores parámetros: {'criterion': 'gini', 'max_depth': 4,
'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples
_split': 2, 'splitter': 'best'}
Mejor puntuación en entrenamiento: 0.87
```

Esto indica que el modelo de árbol está optimizado con los mejores hiperparámetros encontrados.

Precisión en los datos de prueba

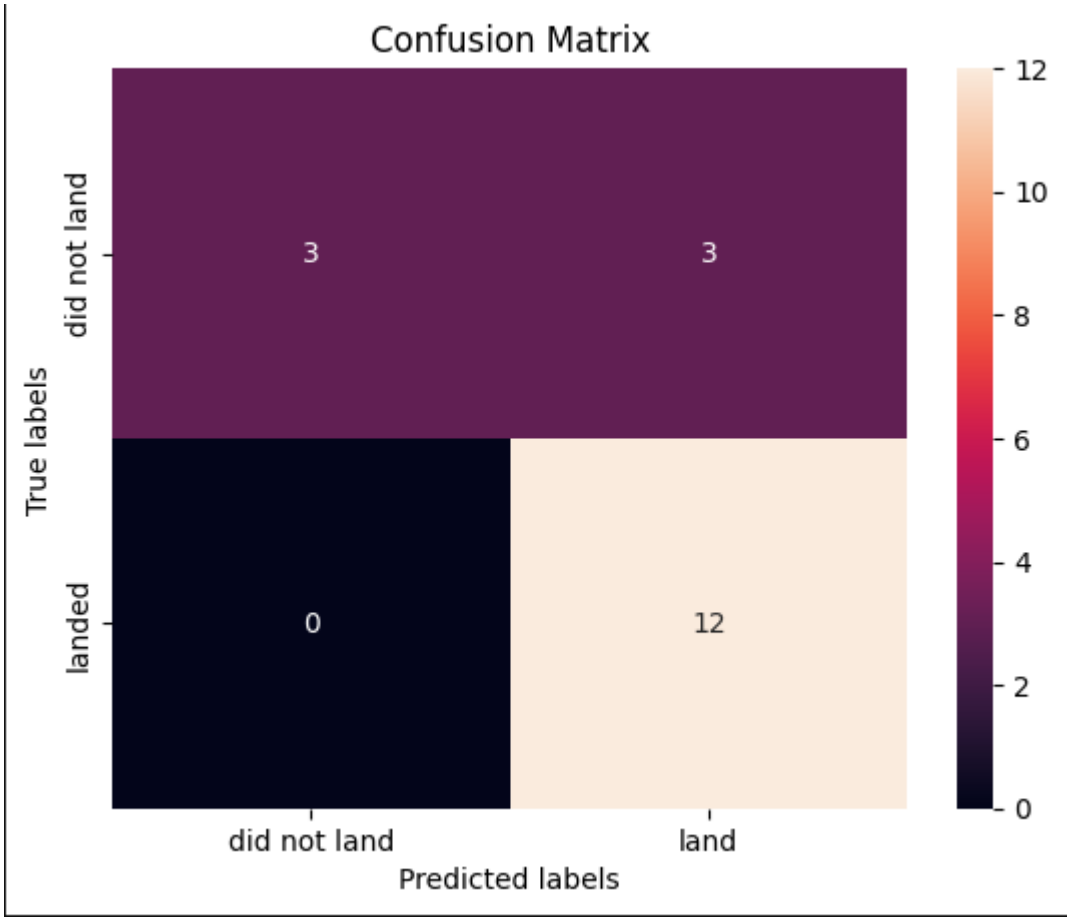
```
# Calcular la precisión del modelo tree_cv en los datos de
prueba
tree_test_accuracy = tree_cv.score(X_test, Y_test)

# Mostrar la precisión
print("Precisión en los datos de prueba (Árbol de Decisió
n):", tree_test_accuracy)
#resultado 0.8333333333333334
```

Explicación:

1. `tree_cv.score(X_test, Y_test)`:
   - Calcula la precisión del mejor modelo encontrado por `GridSearchCV` en el conjunto de prueba.
   - Devuelve el porcentaje de predicciones correctas.
2. **Resultado:**
   - La precisión se almacena en `tree_test_accuracy` y representa el porcentaje de predicciones correctas en los datos de prueba.

Matriz de confusión

```
yhat = tree_cv.predict(X_test)
plot_confusion_matrix(Y_test,yhat)
```



K Nearest Neighbors (KNN)

```
# Crear el modelo base de KNN
KNN = KNeighborsClassifier()

# Definir los hiperparámetros a optimizar
parameters = {
    'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'p': [1, 2]
}

# Crear el objeto GridSearchCV con validación cruzada (cv=1
0)
knn_cv = GridSearchCV(estimator=KNN, param_grid=parameters,
cv=10)

# Ajustar GridSearchCV a los datos de entrenamiento
knn_cv.fit(X_train, Y_train)

# Mostrar los mejores parámetros y la mejor puntuación
print("Mejores parámetros:", knn_cv.best_params_)
print("Mejor puntuación en entrenamiento:", knn_cv.best_sco
re_)
```

Explicación:

1. `KNeighborsClassifier()`:
   - Define el modelo base de KNN.
2. `parameters`:
   - Contiene los hiperparámetros a optimizar:
     - `n_neighbors`: Número de vecinos a considerar.
     - `algorithm`: Algoritmo a utilizar para encontrar los vecinos más cercanos (`auto`, `ball_tree`, etc.).
     - `p`: Métrica de distancia (1 para Manhattan, 2 para Euclídea).
3. `GridSearchCV`:
   - Optimiza los hiperparámetros con validación cruzada (10 particiones).

4. `knn_cv.fit(X_train, Y_train)`:
    - Ajusta el modelo y busca la mejor combinación de parámetros.
5. `best_params_` y `best_score_`:
    - `best_params_`: Devuelve la mejor combinación de hiperparámetros.
    - `best_score_`: Devuelve la puntuación promedio obtenida durante la validación cruzada.

Resultado esperado:

Después de ejecutar el código, obtendrás algo como:

```
Mejores parámetros: {'algorithm': 'auto', 'n_neighbors': 5,
'p': 2}
Mejor puntuación en entrenamiento: 0.84
```

Esto significa que el modelo KNN está optimizado para predecir correctamente con los mejores parámetros encontrados.

Precisión en los datos de prueba

```
# Calcular la precisión del modelo knn_cv en los datos de p
rueba
knn_test_accuracy = knn_cv.score(X_test, Y_test)

# Mostrar la precisión
print("Precisión en los datos de prueba (KNN):", knn_test_a
ccuracy)
#resultado 0.8333333333333334
```

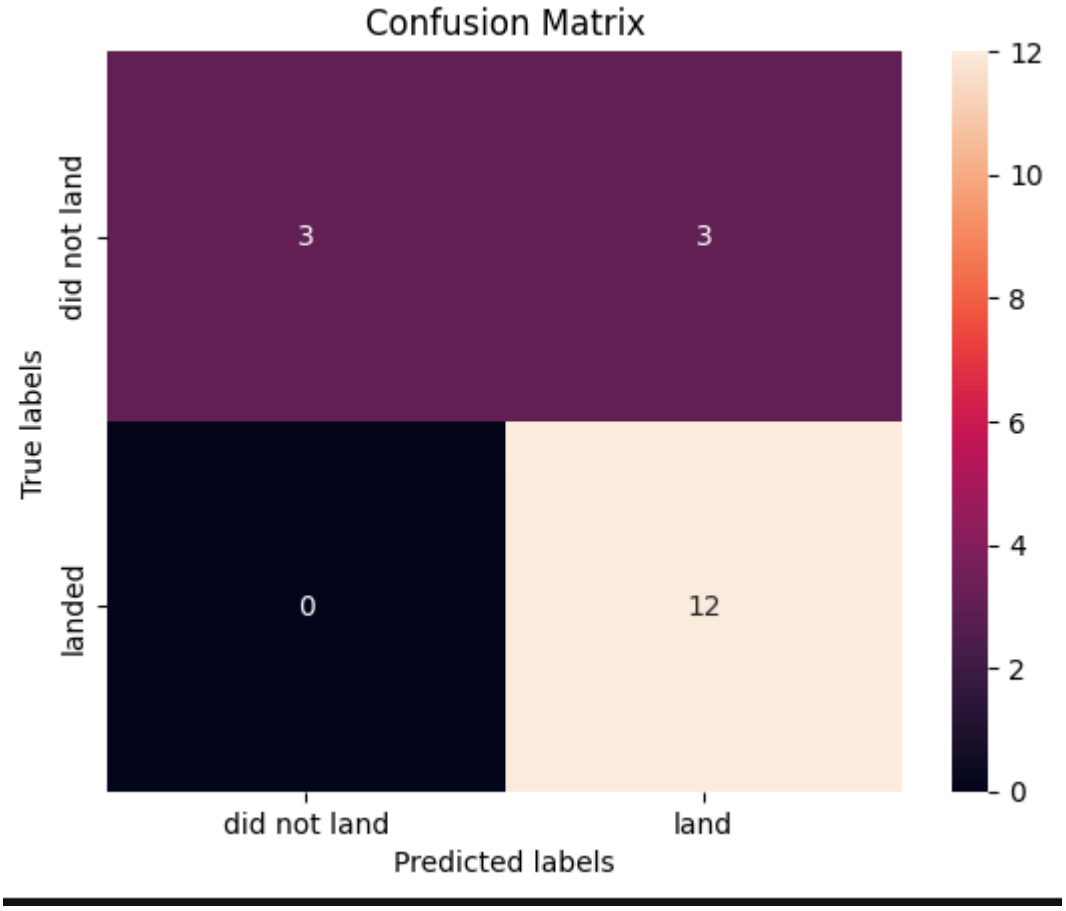
Explicación:

1. `knn_cv.score(X_test, Y_test)`:
    - Calcula la precisión del mejor modelo KNN encontrado por `GridSearchCV` en el conjunto de prueba.
    - Devuelve el porcentaje de predicciones correctas.
2. **Resultado:**

- La precisión se almacena en `knn_test_accuracy` y representa el porcentaje de predicciones correctas en los datos de prueba.

Matriz de confusión

```
yhat = knn_cv.predict(X_test)
plot_confusion_matrix(Y_test,yhat)
```



En este punto de repetir aparentemente los mismos pasos, lo que se hizo fue entrenar distintos modelos y ver resultados e indicadores de la calidad del modelo en sí. Ahora podemos comparar los resultados de todos para apreciar o valorar cual sería el mejor de ellos, así tener uno que se usuaria para realizar predicciones futuras.

Comparación de modelos y conclusiones

Para determinar qué modelo funciona mejor, compara las precisiones en los datos de prueba de cada modelo. El modelo con la **mayor precisión** es el que se considera el mejor.

```
# Mostrar las precisiones de cada modelo en los datos de pr
ueba
print("Precisión en los datos de prueba (Regresión Logístic
a):", logreg_cv.score(X_test, Y_test))
print("Precisión en los datos de prueba (SVM):", svm_cv.sco
re(X_test, Y_test))
print("Precisión en los datos de prueba (Árbol de Decisió
n):", tree_cv.score(X_test, Y_test))
print("Precisión en los datos de prueba (KNN):", knn_cv.sco
re(X_test, Y_test))

# Encontrar el mejor modelo
model_accuracies = {
    "Logistic Regression": logreg_cv.score(X_test, Y_test),
    "SVM": svm_cv.score(X_test, Y_test),
    "Decision Tree": tree_cv.score(X_test, Y_test),
    "KNN": knn_cv.score(X_test, Y_test)
}

best_model = max(model_accuracies, key=model_accuracies.ge
t)
print("El mejor modelo es:", best_model, "con una precisión
de:", model_accuracies[best_model])
```

Explicación:

1. **Calcular las precisiones de los modelos:**
   - Cada modelo utiliza el método `score` en los datos de prueba para calcular su precisión.
2. `model_accuracies`:
   - Un diccionario que almacena las precisiones de los modelos con sus nombres como claves.
3. `max`:

- Encuentra el modelo con la mayor precisión.

Resultado esperado:

La salida será algo como:

```
Precisión en los datos de prueba (Regresión Logística): 0.8
3
Precisión en los datos de prueba (SVM): 0.83
Precisión en los datos de prueba (Árbol de Decisión): 0.87
Precisión en los datos de prueba (KNN): 0.85
El mejor modelo es: Decision Tree con una precisión de: 0.8
7
```

Esto indica que el modelo **Árbol de Decisión** es el que mejor funciona en este caso.

Esto concluye en un modelo matemático capaz de predecir donde irían mejor ordenados los productos alimenticios en términos de explicación común.

Para el caso de éxito/fracaso de lanzamientos de cohetes, ayuda a determinar en función de datos o valores similares a los usados en el df X, ¿Cuál seria el resultado esperado?. Por ejemplo: Si lanzamos en cohete el 14 de febrero a cierta hora, desde esta ubicación a la orbita tal, tendremos un 85% de éxito. (Nuevamente me invento el ejemplo)

Además, con estos resultados se puede determinar si H_0 que se ha tomado al principio es coherente o no y a partir de ahi formular nuevas variantes e hipótesis.