

14

Introducción al aprendizaje automático con scikit-learn

Llegamos al final del libro con los primeros pasos en **aprendizaje automático**. En este capítulo introduciremos los conceptos fundamentales del *aprendizaje automático* (*machine learning*, ML) y los aplicaremos a algunos problemas concretos, varios de ellos tomados del ámbito de la seguridad: clasificar correos como *spam* o legítimos, distinguir tráfico benigno de malicioso, detectar conexiones anómalas en una red. Trabajaremos con **scikit-learn**, la biblioteca de referencia para ML clásico en Python: ofrece decenas de algoritmos con una API uniforme, herramientas de preprocesado, validación y evaluación, y una documentación excelente. Al construirse sobre **numpy** y **pandas**, todo lo aprendido en los dos capítulos anteriores es directamente reutilizable.

No es nuestro objetivo cubrir el ML en profundidad: el lector interesado encontrará tratados completos en libros y cursos dedicados. Lo que queremos es dar una primera vista **operativa** del flujo de trabajo (carga, preprocesado, entrenamiento, evaluación, predicción) y mostrar cómo se manifiesta este flujo en escenarios reales, varios de ellos del ámbito de la ciberseguridad.

14.1. Conceptos fundamentales

14.1.1. Aprendizaje supervisado y no supervisado

Los problemas de ML se agrupan en dos grandes familias:

- **Aprendizaje supervisado:** disponemos de pares (*entrada, salida correcta*). Un

algoritmo busca una función que prediga la salida a partir de la entrada. Ejemplos: clasificar un correo como *spam* (clase $\in \{0, 1\}$) o predecir el número de visitas de mañana (regresión, salida continua).

- **Aprendizaje no supervisado:** solo disponemos de entradas, sin etiquetas. El algoritmo busca estructura: agrupar observaciones similares (*clustering*), detectar anomalías o reducir dimensionalidad. Es muy frecuente en seguridad, donde rara vez tenemos etiquetas masivas y bien curadas.

Entre ambos extremos hay variantes (*semi-supervisado*, *auto-supervisado*, *aprendizaje por refuerzo*) que no abordaremos aquí.

14.1.2. Variables, features y vectores de datos

En ML se trabaja típicamente con dos estructuras:

- **X:** matriz de *features* (entradas) de forma (**n_muestras**, **n_features**). Cada fila es una observación, cada columna una variable.
- **y:** vector de etiquetas (salidas) de longitud **n_muestras**.

En *scikit-learn*, **X** e **y** son *ndarray* de *numpy* o *DataFrame*/*Series* de *pandas* – ambos funcionan.

14.1.3. Entrenamiento, validación y prueba

El error que un modelo comete *sobre los datos con los que se entrenó* es siempre demasiado optimista: el modelo puede haber memorizado peculiaridades del conjunto de entrenamiento (*sobreajuste* o *overfitting*). Para estimar el rendimiento real, se reserva una parte de los datos como conjunto de **prueba** (*test*) que el modelo nunca ve durante el entrenamiento. Una división típica es 70 %-80 % para entrenamiento y 20 %-30 % para prueba.

Cuando se quiere ajustar *hiperparámetros* (parámetros que no aprende el algoritmo, sino que debemos elegir nosotros), se reserva además un conjunto de **validación** o se utiliza *validación cruzada* sobre el entrenamiento.

14.1.4. Métricas de evaluación

Para clasificación binaria, las métricas más utilizadas son:

- **Exactitud** (*accuracy*): proporción de aciertos. Engaña en problemas desbalanceados (si el 99 % del tráfico es benigno, predecir siempre “benigno” acierta el 99 %).
- **Precisión** (*precision*): de las predicciones positivas, cuántas son correctas. Importa cuando un falso positivo es costoso (cerrar una cuenta legítima por sospecha).

- **Sensibilidad** (*recall*): de los casos positivos reales, cuántos detectamos. Importa cuando un falso negativo es peligroso (no detectar un *malware*).
- **F1**: media armónica de precisión y sensibilidad. Balancea ambas.
- **Matriz de confusión**: tabla 2×2 con verdaderos/falsos positivos/negativos.
- **Curva ROC y AUC**: relación entre tasa de verdaderos positivos y de falsos positivos al variar el umbral de decisión. Útil para comparar modelos independientemente del umbral.

En ciberseguridad, el desbalance es la norma

Los datasets de seguridad están casi siempre desbalanceados: hay muchísimas más observaciones benignas que maliciosas. La *accuracy* no es una métrica adecuada en estos contextos: prefiere *precision*, *recall*, F1 y curvas ROC/PR (*precision-recall*). Y cuando entrenes, considera técnicas como ponderar las clases (`class_weight="balanced"`) o submuestrear/sobremuestrear.

14.2. La API de *scikit-learn*

Una de las virtudes de *scikit-learn* es su API uniforme. Casi todos los algoritmos siguen el mismo patrón, con tres métodos principales:

- `modelo.fit(X, y)`: ajusta (entrena) el modelo a los datos.
- `modelo.predict(X)`: produce predicciones para nuevas entradas.
- `modelo.score(X, y)`: evalúa el rendimiento (depende del estimador).

Esto significa que cambiar de algoritmo es esencialmente cambiar la línea `modelo = ...`; el resto del código sigue igual.

14.2.1. Un ejemplo mínimo de clasificación

Vamos a clasificar muestras del clásico dataset Iris (no es de seguridad, pero sirve como “Hola, mundo” de ML):

```
1 from sklearn.datasets import load_iris
2 from sklearn.model_selection import train_test_split
3 from sklearn.linear_model import LogisticRegression
4 from sklearn.metrics import accuracy_score,
  ↪ classification_report
5
```

```
6 # 1. Cargar datos
7 iris = load_iris()
8 X, y = iris.data, iris.target
9
10 # 2. Dividir en entrenamiento / prueba
11 X_train, X_test, y_train, y_test = train_test_split(
12     X, y, test_size=0.3, random_state=42, stratify=y)
13
14 # 3. Entrenar
15 modelo = LogisticRegression(max_iter=1000)
16 modelo.fit(X_train, y_train)
17
18 # 4. Predecir y evaluar
19 y_pred = modelo.predict(X_test)
20 print("Accuracy:", accuracy_score(y_test, y_pred))
21 print(classification_report(y_test, y_pred,
22                             target_names=iris.target_names))
```

Algunos puntos clave:

- `train_test_split` con `stratify=y` preserva la proporción de clases entre los dos subconjuntos. Importante en problemas desbalanceados.
- `random_state=42` fija la semilla de la división para reproducibilidad.
- `classification_report` devuelve precisión, sensibilidad y F1 por clase, que es mucho más informativo que la *accuracy* sola.

14.2.2. Preprocesado: StandardScaler y LabelEncoder

Muchos algoritmos asumen que las variables están en escalas comparables (regresión logística, SVM, redes neuronales) o requieren entradas numéricas (no categóricas). Las dos herramientas más usadas son:

- **StandardScaler**: estandariza cada variable restando la media y dividiendo por la desviación típica, dejándola con $\mu = 0$ y $\sigma = 1$.
- **OneHotEncoder**: convierte una variable categórica de k valores en k columnas binarias.

```
↪ from sklearn.preprocessing import StandardScaler,
    OneHotEncoder

scaler = StandardScaler()
X_train_esc = scaler.fit_transform(X_train)
X_test_esc = scaler.transform(X_test)
```

Ajusta el preprocesado solo con *train*

Es un error muy común aplicar `fit_transform` sobre todos los datos antes de dividir. Esto introduce *data leakage*: el conjunto de prueba influye en la media y la desviación con que se escala el entrenamiento, y las métricas de prueba quedan optimistas. La regla es: `fit_transform` solo con `X_train` y `transform` (sin `fit`) con `X_test`.

14.2.3. Pipelines

Para no tener que recordar el orden de las transformaciones, *scikit-learn* ofrece la clase `Pipeline`, que encadena pasos de preprocesado y un estimador final en un único objeto. Es la forma recomendada de trabajar:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("clf", LogisticRegression(max_iter=1000)),
])
pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

`Pipeline` aplica `fit_transform` en cada paso al entrenar y `transform` al predecir, evitando el *leakage* y manteniendo el código limpio.

14.2.4. Validación cruzada

Para estimar el rendimiento de forma más robusta que con una única división, se usa *k-fold cross-validation*: se parte el conjunto de entrenamiento en *k* pliegues, se entrena *k* veces dejando uno fuera y se promedian las métricas:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(pipe, X_train, y_train, cv=5,
    ↪      scoring="f1_macro")
print(f"F1 medio: {scores.mean():.3f} +- {scores.std():.3f}")
```

14.2.5. Búsqueda de hiperparámetros: GridSearchCV

`GridSearchCV` explora combinaciones de hiperparámetros con validación cruzada y devuelve la mejor:

```
from sklearn.model_selection import GridSearchCV

grid = {
    "clf__C": [0.1, 1.0, 10.0],
    "clf__penalty": ["l2"],
}
busqueda = GridSearchCV(pipe, grid, cv=5, scoring="f1_macro",
    ↪ n_jobs=-1)
busqueda.fit(X_train, y_train)
print("Mejores parámetros:", busqueda.best_params_)
print("Mejor F1:", busqueda.best_score_)
```

La notación `clf__C` accede al hiperparámetro `C` del paso `clf` del pipeline.

14.3. Algoritmos clásicos

scikit-learn implementa decenas de algoritmos. Recogemos los más utilizados, agrupados por familia.

14.3.1. Clasificación

- `LogisticRegression`: regresión logística. Lineal, interpretable, base de comparación natural.
- `KNeighborsClassifier`: k vecinos más cercanos. No paramétrico; sensible a la escala (estandariza siempre).
- `DecisionTreeClassifier`: árbol de decisión. Interpretable, propenso al sobreajuste si no se poda.
- `RandomForestClassifier`: bosque aleatorio. Robusto, buen punto de partida.
- `GradientBoostingClassifier` y `HistGradientBoostingClassifier`: *boosting*. Suelen dar el mejor rendimiento en problemas tabulares (a menudo `xgboost` / `lightgbm` si se necesita aún más rendimiento).
↪
- `SVC`: máquinas de vector de soporte. Potentes pero costosas en datasets grandes.
- `MLPClassifier`: perceptrón multicapa. Para redes serias, salta a `pytorch` o `tensorflow`.

14.3.2. Regresión

Para predecir valores continuos: `LinearRegression`, `Ridge`, `Lasso`, `RandomForestRegressor`, `GradientBoostingRegressor`, `SVR`.

14.3.3. Clustering

Para agrupar sin etiquetas: KMeans, DBSCAN, AgglomerativeClustering, GaussianMixture.

14.3.4. Detección de anomalías

Específicamente útil en seguridad: IsolationForest, OneClassSVM, LocalOutlierFactor. Aprenden la “normalidad” y marcan como anómalo lo que se aparta.

14.3.5. Reducción de dimensionalidad

Para visualizar o comprimir: PCA (lineal, rápido), TSNE (no lineal, costoso, ideal para 2D/3D), UMAP (similar a t-SNE pero más rápido; en biblioteca aparte).

14.4. Aplicación: detección de spam

Como primer caso práctico, construimos un clasificador de *spam* a partir de texto. Es un problema clásico de seguridad y de procesamiento de lenguaje natural.

```
1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.feature_extraction.text import TfidfVectorizer
4 from sklearn.linear_model import LogisticRegression
5 from sklearn.pipeline import Pipeline
6 from sklearn.metrics import classification_report,
↪     confusion_matrix
7
8 # Dataset: cada fila tiene 'texto' y 'etiqueta' (0=ham, 1=
↪     spam)
9 df = pd.read_csv("correos.csv")
10 X = df["texto"]
11 y = df["etiqueta"]
12
13 X_train, X_test, y_train, y_test = train_test_split(
14     X, y, test_size=0.25, random_state=42, stratify=y)
15
```

```
16 pipe = Pipeline([
17     ("tfidf", TfidfVectorizer(min_df=2, ngram_range=(1, 2),
18                             stop_words="english")),
19     ("clf", LogisticRegression(max_iter=1000, class_weight=
    ↪ "balanced")),
20 ])
21 pipe.fit(X_train, y_train)
22 y_pred = pipe.predict(X_test)
23
24 print(classification_report(y_test, y_pred, target_names=["
    ↪ ham", "spam"]))
25 print(confusion_matrix(y_test, y_pred))
```

Qué está pasando.

- `TfidfVectorizer` convierte cada texto en un vector numérico ponderando los términos por frecuencia inversa de documento (TF-IDF). `ngram_range=(1, 2)` usa palabras y bigramas; `min_df=2` ignora términos que aparecen solo en un correo.
- `class_weight="balanced"` pondera las clases inversamente a su frecuencia, mitigando el desbalance.
- El pipeline aplica TF-IDF sobre el entrenamiento y la *misma* transformación al test.
- Mostramos `classification_report` y la matriz de confusión, donde se distinguen los falsos positivos (ham marcados como spam) y falsos negativos (spam que se nos cuela).

14.5. Aplicación: clasificación de tráfico de red

A partir de *features* numéricas extraídas de capturas de tráfico (duración de conexión, bytes enviados/recibidos, número de paquetes, flags TCP...), entrenamos un clasificador para distinguir tráfico benigno de ataques. Usamos `RandomForestClassifier`, que tolera bien las variables sin escalar y suele ser un punto de partida competitivo:

```
1 import pandas as pd
2 from sklearn.model_selection import train_test_split,
    ↪ cross_val_score
3 from sklearn.ensemble import RandomForestClassifier
4 from sklearn.metrics import classification_report,
    ↪ roc_auc_score
5
```

```

6 df = pd.read_csv("trafico.csv")      # con columna 'etiqueta'
↪   (0=benigno, 1=ataque)
7 X = df.drop(columns=["etiqueta"])
8 y = df["etiqueta"]
9
10 X_train, X_test, y_train, y_test = train_test_split(
11     X, y, test_size=0.25, random_state=42, stratify=y)
12
13 clf = RandomForestClassifier(
14     n_estimators=300, max_depth=None,
15     class_weight="balanced", random_state=42, n_jobs=-1)
16 clf.fit(X_train, y_train)
17
18 y_pred = clf.predict(X_test)
19 y_proba = clf.predict_proba(X_test)[: , 1]
20
21 print(classification_report(y_test, y_pred, target_names=["
↪     benigno", "ataque"]))
22 print(f"AUC ROC: {roc_auc_score(y_test, y_proba):.3f}")
23
24 # Importancia de las variables
25 importancia = pd.Series(clf.feature_importances_, index=X.
↪     columns)
26 print(importancia.sort_values(ascending=False).head(10))

```

Notas.

- `predict_proba` devuelve la probabilidad estimada de cada clase. La AUC ROC se calcula con la probabilidad de la clase positiva.
- `feature_importances_` permite identificar qué variables influyen más en la decisión, lo que ayuda a entender el modelo y a detectar *features* potencialmente fugadas (*data leakage*: variables que codifican accidentalmente la etiqueta).

14.6. Aplicación: detección de anomalías sin etiquetas

A menudo no tenemos etiquetas: solo sabemos que la mayoría del tráfico es “normal” y queremos descubrir lo distinto. `IsolationForest` es un detector de anomalías eficaz y rápido que asigna a cada muestra una puntuación; las muestras con puntuación más baja son las más anómalas:

```

1 import pandas as pd
2 from sklearn.ensemble import IsolationForest
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.pipeline import Pipeline
5
6 df = pd.read_csv("conexiones.csv")
7 X = df[["duracion", "bytes_enviados", "bytes_recibidos", "
↪     paquetes"]]
8
9 modelo = Pipeline([
10     ("scaler", StandardScaler()),
11     ("if", IsolationForest(contamination=0.01, random_state
↪     =42)),
12 ])
13 modelo.fit(X)
14
15 # +1 = normal, -1 = anomalia
16 df["anomalia"] = modelo.predict(X) == -1
17 df["score"]     = -modelo["if"].score_samples(modelo["scaler"
↪     ].transform(X))
18
19 # Top 10 mas anomalas
20 print(df.sort_values("score", ascending=False).head(10))

```

Contamination y validación

contamination es la fracción esperada de anomalías. Si lo desconoces, prueba con varios valores y revisa manualmente los resultados; en producción, conviene comparar puntuaciones contra un umbral fijo en lugar de la decisión binaria `predict`. Sin etiquetas, validar un detector de anomalías es difícil: lo habitual es combinar inspección manual con reglas conocidas.

14.7. Cierre del libro

↪ Hemos recorrido un camino largo: del primer `print("Hola, mundo desde Python")` al entrenamiento de un clasificador con un bosque aleatorio, aplicado por ejemplo a la detección de tráfico malicioso. Por el camino hemos visto los tipos básicos del lenguaje, las estructuras compuestas, la entrada/salida, las dos bibliotecas que dan acceso al ecosistema de datos (`numpy` y `pandas`), las herramientas de visualización (`matplotlib` y `seaborn`) y, finalmente, los conceptos básicos del aprendizaje automático con `scikit-learn`.

↪ Por supuesto, esto es solo un punto de partida. El aprendizaje automático y, en general, la ciencia de datos forman un campo en evolución muy rápida, donde conviven

técnicas clásicas como las que hemos visto con redes neuronales profundas, modelos de lenguaje y agentes autónomos. La ciberseguridad es uno de los dominios donde estas técnicas encuentran aplicación. Algunas direcciones naturales para seguir aprendiendo:

- **Redes neuronales profundas:** `pytorch` o `tensorflow` son los dos *frameworks* dominantes. Para texto, los modelos de tipo *transformer* (a través de `transformers` de Hugging Face) son hoy el estándar.
- **Procesado de lenguaje natural:** análisis de logs, descripciones de vulnerabilidades, correos de *phishing*.
- **Modelos generativos:** detección de contenido sintético, simulación de adversarios.
- **Aprendizaje por refuerzo:** agentes que aprenden a explorar o defender entornos.
- **Adversarial machine learning:** cómo atacar y defender modelos.

Esperamos que el libro te haya servido como base sólida. El resto depende de tu curiosidad y de los problemas concretos en los que decidas embarcarte. ¡Buena suerte!

14.8. Ejercicios

EJERCICIO 14.1

Carga el dataset Iris de `sklearn.datasets` y compara cuatro clasificadores: `LogisticRegression`, `KNeighborsClassifier` (con $k = 5$), `DecisionTreeClassifier` y `RandomForestClassifier`. Para cada uno, usa un `Pipeline` con `StandardScaler` y reporta la *accuracy* media y desviación típica obtenidas con validación cruzada de 10 pliegues. ¿Cuál gana? ¿Justifica el coste extra?

Solución (girar para leer)

```

1 from sklearn.datasets import load_iris
2 from sklearn.model_selection import cross_val_score
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.pipeline import Pipeline
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.neighbors import KNeighborsClassifier
7 from sklearn.tree import DecisionTreeClassifier
8 from sklearn.ensemble import RandomForestClassifier
9 X, y = load_iris(return_X_y=True)
10
11 # Cuatro clasificadores a comparar
12 modelos = {
13     "LogisticRegression": LogisticRegression(max_iter
14         =1000, random_state=0),
15     "KNN(k=5)": KNeighborsClassifier(
16         n_neighbors=5),
17     "DecisionTree": DecisionTreeClassifier(
18         random_state=0),
19     "RandomForest": RandomForestClassifier(
20         n_estimators=100, random_state=0),
21 }
22
23 # Validacion cruzada de 10 pliegues con escalado dentro
24 del pipeline
25 for nombre, clf in modelos.items():
26     pipe = Pipeline([("scaler", StandardScaler()), ("
27         clf", clf)])
28     scores = cross_val_score(pipe, X, y, cv=10, scoring
29         = "accuracy")
30     print(f"{nombre:20s} accuracy = {scores.mean():.3f}
31         +/- {scores.std():.3f}")

```

Código 14.1: Solución al Ejercicio 14.1.

EJERCICIO 14.2

Implementa el clasificador de *spam* de la sección 14.4 y experimenta con tres variaciones: (I) cambia `TfidfVectorizer` por `CountVectorizer`; (II) prueba un `RandomForestClassifier` en lugar de la regresión logística; (III) ajusta `ngram_range` a (1, 3). Compara las matrices de confusión y los F1 obtenidos.

Como dataset puedes utilizar el clásico *SMS Spam Collection* disponible públicamente (también puedes generar uno sintético si prefieres).

Solución (girar para leer)

```

1 import numpy as np
2 from sklearn.model_selection import train_test_split
3 from sklearn.feature_extraction.text import
4 TfidfVectorizer, CountVectorizer, LogisticRegression
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.ensemble import RandomForestClassifier
7 from sklearn.pipeline import Pipeline
8 from sklearn.metrics import f1_score, confusion_matrix
9 np.random.seed(0)
10
11 # Generar correos sintéticos: ham y spam con
12 vocabulario parcialmente solapado
13 ham = "reunion equipo informe proyecto calendario".
14 split() + comun
15 spam = "gratis premio dinero oferta urgente gana clic".
16 split() + comun
17 def gen(pal, n):
18     return [" ".join(np.random.choice(pal, 8)) for _ in
19             range(n)]
20 X = gen(ham, 300) + gen(spam, 300)
21 y = np.array([0] * 300 + [1] * 300)
22 Xtr, Xte, ytr, yte = train_test_split(X, y, test_size
23                                     = 0.25, random_state=0, stratify=y)
24 # Tres variaciones del clasificador de spam
25 {
26     "Count + LogReg": Pipeline([("v",
27                                 CountVectorizer()), ("c", LogisticRegression(
28                                     max_iter=1000))]),
29     "Tfidf + RandomForest": Pipeline([("v",
30                                       TfidfVectorizer()), ("c", RandomForestClassifier(
31                                           n_estimators=100, random_state=0))]),
32     "Tfidf(1,3) + LogReg": Pipeline([("v",
33                                       TfidfVectorizer(ngram_range=(1, 3))), ("c",
34                                       LogisticRegression(
35                                           max_iter=1000))])],
36     }
37 for nombre, pipe in variantes.items():
38     pipe.fit(Xtr, ytr)
39     pred = pipe.predict(Xte)
40     print(f"== {nombre} == f1 = {f1_score(yte, pred)}")
41     print(confusion_matrix(yte, pred))

```

EJERCICIO 14.3

A partir del clasificador de tráfico de la sección 14.5, usa `GridSearchCV` para buscar los mejores valores de `n_estimators` ($\in \{100, 300, 500\}$), `max_depth` ($\in \{5, 10, \text{None}\}$) y `min_samples_split` ($\in \{2, 5, 10\}$) optimizando `f1_macro`. Reporta los mejores hiperparámetros y compara las métricas con el modelo por defecto.

Solución (girar para leer)

```

1 from sklearn.datasets import make_classification
2 from sklearn.model_selection import train_test_split,
3 GridSearchCV
4 from sklearn.ensemble import RandomForestClassifier
5 from sklearn.metrics import f1_score
6 # Dataset de tráfico sintético (benigno vs ataque)
7 X, y = make_classification(n_samples=1000, n_features
8                             =12, n_informative=6,
9                             weights=[0.7, 0.3],
10                             random_state=0)
11 # Modelo por defecto como referencia
12 base = RandomForestClassifier(random_state=0)
13 base.fit(Xtr, ytr)
14 print(f"F1 macro por defecto: {f1_score(yte, base.
15       predict(Xte), average='macro'):.3f}")
16 # Búsqueda de hiperparámetros optimizando f1_macro
17 rejilla = {
18     "n_estimators": [100, 300, 500],
19     "max_depth": [5, 10, None],
20     "min_samples_split": [2, 5, 10],
21 }
22 gs = GridSearchCV(RandomForestClassifier(random_state
23     =0), rejilla,
24     scoring="f1_macro", cv=5, n_jobs=-1)
25 gs.fit(Xtr, ytr)
26 print(f"Mejores hiperparámetros: ", gs.best_params_)
27 print(f"F1 macro ajustado: {f1_score(yte, gs.predict
28     (Xte), average='macro'):.3f}")

```

Código 14.3: Solución al Ejercicio 14.3.

EJERCICIO 14.4

Genera un dataset sintético con `numpy`: 1000 muestras de tres dimensiones procedentes de una normal multivariante con $\mu = (0,0,0)$ y matriz de covarianza identidad, y mezcla 20 anomalías cuyas coordenadas son uniformes en $[-5,5]$. Entrena un `IsolationForest` con `contamination=0.02` y reporta cuántas de las anomalías inyectadas detecta correctamente y cuántos falsos positivos produce.

Solución (girar para leer)

```

1 import numpy as np
2 from sklearn.ensemble import IsolationForest
3
4 np.random.seed(0)
5
6 # 1000 muestras normales en 3D con media cero y
7   covarianza identidad
8   normales = np.random.multivariate_normal([0, 0, 0], np.
9     eye(3), size=1000)
10 # 20 anomalías uniformes en [-5, 5]
11 anomalías = np.random.uniform(-5, 5, size=(20, 3))
12 X = np.vstack([normales, anomalías])
13 # Verdad: 1 = normal, -1 = anomalía (convención de
14   IsolationForest)
15 y-real = np.array([1] * 1000 + [-1] * 20)
16 clf = IsolationForest(contamination=0.02, random-state
17   =0)
18 pred = clf.fit_predict(X)
19 # Anomalías inyectadas detectadas y falsos positivos
20 detectadas = np.sum((y-real == -1) & (pred == -1))
21 falsos-pos = np.sum((y-real == 1) & (pred == -1))
22 print(f"Anomalías inyectadas: {np.sum(y-real ==
23   -1)}")
24 print(f"Anomalías detectadas: {detectadas}")
25 print(f"Falsos positivos (normales): {falsos-pos}")

```

Código 14.4: Solución al Ejercicio 14.4.

EJERCICIO 14.5

Dado un fichero CSV con *features* numéricas extraídas de eventos de log (sin etiquetas), aplica `KMeans` con k entre 2 y 10 y dibuja la curva del “codo” (*inertia* vs k) para elegir un buen número de clústeres. Una vez elegido, visualiza los clústeres en 2D proyectando con `PCA` y comenta si los grupos parecen razonables (uniformes, bien separados, etc.).

Solución (girar para leer)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.datasets import make_blobs
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.cluster import KMeans
6 from sklearn.decomposition import PCA
7
8 np.random.seed(0)
9
10 # Features numéricas sintéticas de eventos de log (sin
    etiquetas)
11 X, _ = make_blobs(n_samples=600, centers=4, n_features
    =6,
12 cluster_std=1.5, random_state=0)
13 X = StandardScaler().fit_transform(X)
14
15 # Curva del código: inercia frente a k
16 ks = range(2, 11)
17 inercias = [KMeans(n_clusters=k, n_init=10,
    random_state=0).fit(X).inertia_ for k in ks]
18
19 # Numero de clusters elegido tras inspeccionar el código
20 k_elegido = 4
21 etiquetas = KMeans(n_clusters=k_elegido, n_init=10,
    random_state=0).fit_predict(X)
22 Xp = PCA(n_components=2, random_state=0).fit_transform(
    X)
23
24 fig, (a1, a2) = plt.subplots(1, 2, figsize=(11, 4))
25 a1.plot(list(ks), inercias, "o-"); a1.set_xlabel("k");
    a1.set_ylabel("Inercia")
26 a1.set_title("Curva del código")
27 a2.scatter(Xp[:, 0], Xp[:, 1], c=etiquetas, cmap="tab10",
    s=12)
28 a2.set_title(f"Clusters (k={k_elegido}) en PCA 2D")
29 fig.tight_layout()
30 fig.savefig("kmeans-logs.png", dpi=120)
31 print(f"El código = {k_elegido}; figura guardada en
    kmeans-logs.png")

```

Código 14.5: Solución al Ejercicio 14.5.

EJERCICIO 14.6

Construye un pipeline que combine: (I) `ColumnTransformer` para aplicar `StandardScaler` a las columnas numéricas y `OneHotEncoder` a las categóricas; y (II) un `RandomForestClassifier` como clasificador final. Aplícalo a un dataset con columnas de ambos tipos (puedes usar el *Adult* o cualquier CSV con *features* mixtas) y entrena/valida con `cross_val_score`.

Solución (girar para leer)

```

1 import numpy as np
2 import pandas as pd
3 from sklearn.compose import ColumnTransformer
4 from sklearn.preprocessing import StandardScaler,
5 OneHotEncoder
6 from sklearn.ensemble import RandomForestClassifier
7 from sklearn.pipeline import Pipeline
8 from sklearn.model_selection import cross_val_score
9 np.random.seed(0)
10 n = 600
11
12 # Dataset con features numericas y categoricas
13 df = pd.DataFrame({
14     "edad": np.random.randint(18, 70, n),
15     "ingresos": np.random.normal(30000, 9000, n),
16     "pais": np.random.choice(["ES", "FR", "DE"], n),
17     "rol": np.random.choice(["admin", "user", "
18     "guest"], n),
19 })
20 # Etitqueta dependiente de variables numericas y
21 categoricas
22 y = (df["ingresos"] > 30000) ~ (df["rol"] == "admin")
23     .astype(int)
24 num = ["edad", "ingresos"]
25 cat = ["pais", "rol"]
26 pre = ColumnTransformer([
27     ("num", StandardScaler(), num),
28     ("cat", OneHotEncoder(handle_unknown="ignore"), cat
29     ),
30 ])
31 pipe = Pipeline([
32     ("pre", pre),
33     ("clf", RandomForestClassifier(n_estimators=100,
34     random_state=0)),
35 ])
36 scores = cross_val_score(pipe, df, y, cv=5, scoring="
37 accuracy")
38 print(f"Accuracy CV: {scores.mean():.3f} +/- {scores.
39 std():.3f}")

```

Código 14.6: Solución al Ejercicio 14.6.

EJERCICIO 14.7

Entrena dos modelos sobre un mismo problema de clasificación binaria (e.g., `LogisticRegression` y `RandomForestClassifier`). Calcula la curva ROC de cada uno con `sklearn.metrics.roc_curve` y dibújalas en una sola figura con `matplotlib`, mostrando también la diagonal de azar. Anota en la leyenda el AUC de cada modelo.

Solución (girar para leer)

```

1 import matplotlib.pyplot as plt
2 from sklearn.datasets import make_classification
3 from sklearn.model_selection import train_test_split
4 from sklearn.linear_model import LogisticRegression
5 from sklearn.ensemble import RandomForestClassifier
6 from sklearn.metrics import roc_curve, roc_auc_score
7
8 # Problema de clasificación binaria
9 X, y = make_classification(n_samples=1000, n_features
10                             =10, n_informative=5,
11                             weights=[0.6, 0.4],
12                             random_state=0)
13 Xtr, Xte, ytr, yte = train_test_split(X, y, test_size
14                                       =0.3, random_state=0, stratify=y)
15 modelos = {
16     "LogisticRegression": LogisticRegression(max_iter
17                                             =1000, random_state=0),
18     "RandomForest": RandomForestClassifier(
19         n_estimators=100, random_state=0),
20 }
21 plt.figure(figsize=(6, 5))
22 for nombre, clf in modelos.items():
23     clf.fit(Xtr, ytr)
24     proba = clf.predict_proba(Xte)[:, 1]
25     fpr, tpr, _ = roc_curve(yte, proba)
26     auc = roc_auc_score(yte, proba)
27     plt.plot(fpr, tpr, label=f"{nombre} (AUC = {auc:.3f})")
28 }
29 plt.legend(); plt.tight_layout()
30 plt.savefig("roc-comparativa.png", dpi=120)
31 print("Figura guardada en roc-comparativa.png")

```

EJERCICIO 14.8

Combina lo aprendido en los caps. 12, 13 y este capítulo: a partir de un fichero CSV con tráfico de red, detecta anomalías con `IsolationForest`, exporta las muestras anómalas a otro CSV y dibuja un *scatterplot* en 2D (proyección con PCA) coloreando puntos normales vs anómalos. El programa debe ser un *script* ejecutable con un argumento de línea de comandos para el fichero de entrada.

Solución (girar para leer)

```

1 import sys
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from sklearn.ensemble import IsolationForest
6 from sklearn.decomposition import PCA
7
8 np.random.seed(0)
9
10 # Si no se pasa CSV, se genera uno sintético de ejemplo
11 if len(sys.argv) > 1:
12     df = pd.read_csv(sys.argv[1])
13 else:
14     normal = np.random.normal(0, 1, (480, 4))
15     anom = np.random.uniform(-6, 6, (20, 4))
16     df = pd.DataFrame(np.vstack([normal, anom]),
17                        columns=["duracion", "bytes-env",
18                               "bytes-rec", "paquetes"])
19 X = df.select_dtypes("number")
20 clf = IsolationForest(contamination=0.04, random_state=0)
21 pred = clf.fit_predict(X)
22 # -1 = anormal,
23     I = normal
24     df["anormal"] = pred == -1
25 # Exportar nuestras anomalías a otro CSV
26 df[df["anormal"]].to_csv("anomalías.csv", index=False)
27 print(f"Anomalías detectadas: {df['anormal'].sum()}")
28 # Scatterplot 2D con proyección PCA, coloreando normal
29     us anormalo
30 Xp = PCA(n_components=2, random_state=0).fit_transform(X)
31 plt.scatter(Xp[df["anormal"], 0], Xp[df["anormal"], 1],
32             s=30, c="red", label="anormalo")
33 plt.legend(); plt.title("Anomalías de tráfico (PCA 2D)")
34 plt.tight_layout()
35 plt.savefig("anomalías-tráfico.png", dpi=120)

```

Código 14.8: Solución al Ejercicio 14.8.