

Hands-on Machine Learning with Scikit-Learn and PyTorch

▼ Landscape

▼ Types of Machine Learning Systems

Types of Machine Learning Systems

- How they are **guided during training** (supervised, unsupervised, semi-supervised, self-supervised, and others)
- **Whether or not they can learn incrementally on the fly** (online versus batch learning)
- Whether they work by simply **comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model**, much like scientists do (instance-based versus model-based learning)

Unsupervised learning:

- **novelty detection:** it aims to detect new instances that look different from all instances in the training set. This requires having a very "clean" training set, devoid of any instance that you would like the algorithm to detect.
- **anomaly detection:** The system is shown mostly normal instances during training, so it learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly
- **association rule learning:** Dig into large amounts of data and discover interesting

relations between attributes

- clustering

Semi-supervised learning:

- Normal combinations of unsupervised and supervised algorithms. For example, a clustering algorithm may be used to group similar instances together, and then every unlabeled instance can be labeled with the most common label in its cluster. Once the whole dataset is labeled, it is possible to use any supervised learning algorithm.

▼ Semi-supervised learning:

Self-supervised learning:

- Generating a fully labeled dataset from a fully unlabeled one, once the whole dataset is labeled, any supervised learning algorithm can be used
- a large dataset of unlabeled images, you can randomly mask a small part of each image and then train a model to recover the original image.
- uses (generated) labels during training
- focuses the same tasks as supervised learning : mainly classification and regression
- pet classification model:
 - dataset of unlabeled photos of pets, you can start by training an image-repairing model using self-supervised learning
 - distinguish different pet species—when it repairs an image of a cat whose face is masked, it must know not to add a dog's face
 - fine-tuning: neural network architectures do, it is then possible to tweak the model so that it predicts pet species instead of repairing images



model can learn the mapping between the species it already knows and the labels we expect from it

Transferring knowledge from one task to another is called **transfer learning** (important deep neural networks)

▼ Reinforcement learning

learning system, called an **agent** in this context, can observe the environment, select and perform actions, and **get rewards** in return (or **penalties**). It must then learn by itself what is the best **strategy**, called a **policy**, to get the most reward over time

Offline learning: learning was turned off (puede analizar datos o lo que este haciendo)

▼ Batch Vs Online Learning

Batch learning:

- trained using all the available data
- **offline learning** : First the system is trained, and then it is launched into production and runs without learning anymore
- **data drift (or model rot)**: model's performance tends to decay slowly over time, world continues to evolve, but the model is fixed.
- **regularly retrain the model on up-to-date data**: no solo por cambios de datos sino por cosas externas (clasificación de mascotas, con el tiempo mejoran las cámaras)
- To know about new data, need to **train** a new version of the system **from scratch** on the full dataset → mejor opción: algorithms that are capable of learning incrementally

Online learning

- train the system incrementally by feeding it data instances sequentially, individually or in small groups called **mini-batches**
- **out-of-core** : train models on huge datasets that cannot fit in one machine's memory, usually done offline

- learning rate: adapt to changing data
- catastrophic forgetting (or catastrophic interference): quickly forget the old data
- bad data is fed to the system: performance will decline. To reduce this risk, switch learning off (and possibly revert to a previously working state), react to abnormal data (anomaly algorithm)

Instance-Based Versus Model-Based Learning

categorize machine learning systems is by how they generalize, good performance measure on the training data is good, but insufficient; the true goal is to perform well on new instances.

instance-based learning: the system learns the examples by heart, then generalizes to new cases by using a similarity.

- shines with small datasets
- does not scale very well : deploying a whole copy of the training set to production (se fija el nuevo input sobre todo el training set y comprara los mas cercanos), bad work well with high-dimensional data

Model-based learning and a typical machine learning workflow:

build a model of these examples and then use that model to make predictions

model selection: you selected a linear model of life satisfaction with just one attribute, GDP per-capita

utility function (or fitness function): performance measure, performance measure

cost function: measures how bad it is

linear regression algorithm comes in: you feed it your training examples, and it finds the parameters that make the linear model fit best to your data. **training the model**

ensemble learning involves training multiple models and combining their individual predictions into improved predictions

federated learning is a decentralized approach where models are trained across multiple devices (e.g., smartphones) and adapted to each user without exchanging raw data

meta-learning is a learning-to-learn approach where models learn how to learn new tasks quickly with minimal data

▼ Main Challenges of Machine Learning

Insufficient Quantity of Training Data: se necesita muchos datos

Nonrepresentative Training Data: training data be representative of the new cases you want to generalize to.(nonrepresentative training set makes model inaccurate)

sampling bias: sample is too small, you will have sampling noise (i.e., nonrepresentative data as a result of chance), but even very large samples can be nonrepresentative if the sampling method is flawed. (**nonresponse bias another type**)

Poor-Quality Data: training data is full of errors, outliers, and noise(**simply discard, missing a few features so either** ignore these instances, fill in the missing values

Irrelevant Features : model need training data contains enough relevant features
and not too many irrelevant ones.

feature engineering :coming up with a good set of features to train on.

- Feature selection (selecting the most useful features to train on among existing features
- Feature extraction (combining existing features to produce a more useful one—as we saw earlier, dimensionality reduction algorithms help
- Creating new features by gathering new data

Overfitting the Training Data:

Overgeneralizing(overfitting), model performs well on the training data, but it does not generalize well.

Complex models such as deep neural networks can detect subtle patterns in the data, but if the **training set is noisy**, or if it is **too small**, which introduces **sampling noise**, then the model is likely to **detect patterns in the noise itself**.

Overfitting happens when the model is too complex relative to the amount and noisiness of the training data, so it starts to learn random patterns in the training data.

- Simplify the model by selecting one with fewer parameters (by reducing the number of attributes), or by constraining the model
- Gather more training data.
- Reduce the noise in the training data

error rate on new cases is called the **generalization error** se calcula evaluate model on test set (training error low y gen error alto → overfitting)



different machine learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation once they were given enough data

Regularización

La **regularización** consiste en aplicar restricciones al modelo para simplificarlo y reducir el riesgo de sobreajuste

- **Hiperparámetros:** Son parámetros del *algoritmo de aprendizaje* (no del modelo en sí) que se definen antes del entrenamiento y permanecen constantes durante el proceso. Por ejemplo, un hiperparámetro de regularización controla cuánto se debe simplificar el modelo; si es muy alto, el modelo será muy simple (incluso plano), lo que evita el sobreajuste pero puede llevar al subajuste

Underfitting

Es el opuesto al sobreajuste. Ocurre cuando el modelo es demasiado simple para aprender la estructura subyacente de los datos.

Soluciones: Seleccionar un modelo más potente, usar mejores características (*feature engineering*) o reducir las restricciones (disminuir el hiperparámetro de regularización).

Prueba y Validación (Testing and Validating)

La única forma de saber si un modelo generalizará bien es probarlo con datos nuevos.

Conjuntos de Entrenamiento y Prueba (Train and Test Sets)

Para evaluar el rendimiento, se divide el conjunto de datos total en dos:

Train Set (Conjunto de Entrenamiento): Se utiliza para entrenar el modelo.

Test Set (Conjunto de Prueba): Se utiliza para medir el **error de generalización** (o error fuera de muestra), que indica qué tan bien funcionará el modelo con instancias que nunca ha visto. Normalmente se reserva un 20% de los datos para este fin.

apply some regularization to avoid overfitting. The question is, how do you

choose the value of the regularization hyperparameter?

Ajuste de Hiperparámetros y el Dev Set (Validation Set)

Si usamos el conjunto de prueba repetidamente para ajustar hiperparámetros, corremos el riesgo de que el modelo se adapte específicamente a ese conjunto de prueba, y el error de generalización real sea mayor al esperado. Para evitar esto, se introduce un tercer conjunto:

- **Validation Set / Dev Set (Conjunto de Validación o Desarrollo):**
Se extrae una parte del conjunto de entrenamiento para comparar varios modelos candidatos y seleccionar el mejor. El proceso es: entrenar varios modelos en el conjunto de entrenamiento reducido, elegir el mejor basado en su rendimiento en el *dev set*, y finalmente evaluar ese modelo único en el conjunto de prueba.

Validación Cruzada (Cross-Validation)

Cuando el conjunto de validación es demasiado pequeño, las evaluaciones pueden ser imprecisas. La solución es la **validación cruzada**.

- **Cómo funciona:** Se divide el conjunto de entrenamiento en múltiples subconjuntos pequeños. El modelo se entrena varias veces, utilizando cada vez un subconjunto distinto para validación y el resto para entrenamiento. El rendimiento final es el promedio de todas las evaluaciones, lo que da una medida mucho más precisa pero a cambio de mayor tiempo de cómputo.

Desajuste de Datos (Data Mismatch) y el Train-Dev Set

A veces, los datos disponibles para entrenamiento no son perfectamente representativos de los que se encontrarán en producción. Por ejemplo, entrenar una app para flores con millones de fotos de la web, pero usarla con fotos tomadas por usuarios con celulares.

Andrew Ng propuso el uso del **Train-Dev Set** para identificar la causa de un mal rendimiento:

1. Se entrena el modelo en el conjunto de entrenamiento (fotos de la web).
2. Se evalúa en el **Train-Dev Set** (una parte de las fotos de la web no usada para entrenar).
 - a. Si falla aquí, el modelo está **sobreajustado** al conjunto de entrenamiento.
3. Si pasa la prueba anterior, se evalúa en el **Dev Set** (fotos reales de la app).
 - a. Si falla aquí, existe un **desajuste de datos** (*data mismatch*) entre los datos de la web y los de la app.

Este enfoque permite saber si el problema está en el modelo (sobreajuste) o en la representatividad de los datos (desajuste) antes de realizar la evaluación final en el **Test Set**.

▼ **End-to-End Machine Learning Project**

▼ **Gemini**

. El flujo de trabajo sigue los pasos de un proyecto real utilizando el dataset de precios de viviendas en California.

1. Mirar el panorama general (Look at the Big Picture)

- **Definición del Problema:** Se busca predecir el precio medio de la vivienda en distritos de California basándose en datos del censo².
- **Marco del Proyecto:**
 - **Aprendizaje Supervisado:** Se tienen etiquetas (precios)³.
 - **Regresión Múltiple:** Se usan múltiples características para predecir⁴.
 - **Regresión Univariada:** Se predice un solo valor por distrito⁵.
 - **Aprendizaje por Lotes (Batch Learning):** Los datos son lo suficientemente pequeños para procesarlos juntos⁶.



If the data were huge, you could either split your batch learning work across multiple servers (using the MapReduce technique) or use an online learning technique

- **Métrica de Rendimiento:**
 - **RMSE (Root Mean Squared Error):** Es la métrica estándar para regresión; penaliza más los errores grandes (Norma ℓ_2)⁷⁷⁷⁷.
 - **MAE (Mean Absolute Error):** Se prefiere si hay muchos valores atípicos (*outliers*) (Norma ℓ_1)⁸⁸⁸⁸.
- **Verificación de Suposiciones:** Confirmar con el equipo de "aguas abajo" (*downstream*) si necesitan el precio exacto o solo una categoría (en cuyo caso sería clasificación)⁹.

2. Obtener los datos (Get the Data)

- **Inspección Inicial (Pandas):**

- `head()` : Ver las primeras filas.
- `info()` : Revisar tipos de datos, valores no nulos y uso de memoria.
- `value_counts()` : Útil para columnas categóricas como `ocean_proximity`.
- `describe()` : Resumen estadístico (media, desviación estándar, cuartiles).
- `hist()` : Visualizar la distribución de los atributos numéricos.

3. Crear el conjunto de prueba (Test Set)

- **Data Snooping Bias:** El error de mirar los datos de prueba antes de tiempo y sesgar el modelo.
- **Muestreo Aleatorio:** Se usa `train_test_split()` de Scikit-Learn.
- **Muestreo Estratificado:** Crucial si un atributo (como el ingreso medio) es muy importante. Se asegura de que el conjunto de prueba sea representativo de toda la población.
 - Se usa `pd.cut()` para crear categorías de ingresos (`income_cat`).
 - Se usa `StratifiedShuffleSplit` o el argumento `stratify` en `train_test_split()`.

4. Explorar y visualizar los datos

- **Visualización Geográfica:** Usar `plot(kind="scatter")`. El parámetro `alpha=0.2` ayuda a ver la densidad de puntos; `s` (tamaño) para población y `c` (color) para precio (`cmap="jet"`).
- **Correlaciones:**
 - `corr()` : Calcula el coeficiente de Pearson para ver qué atributos se relacionan con el precio.
 - `scatter_matrix()` : Grafica cada atributo numérico contra los demás.
- **Combinación de Atributos:** Crear nuevas variables como `rooms_per_household` o `bedrooms_ratio` para encontrar mejores correlaciones.



When creating new combined features, make sure they are not too linearly correlated with existing features: collinearity can cause issues with some models, such as linear regression. In particular, avoid simple weighted sums of existing features

5. Preparar los datos (Preprocessing)

- **Limpieza:** `SimpleImputer(strategy="median")` para rellenar valores faltantes.
- **Atributos Categóricos:**
 - `OrdinalEncoder` : Para categorías con orden.
 - `OneHotEncoder` : Crea variables binarias (0 o 1) para categorías sin orden.
- **Escalamiento de Atributos (Feature Scaling):**
 - `MinMaxScaler` (Normalización): Escala de 0 a 1.
 - `StandardScaler` (Estandarización): Resta la media y divide por la desviación. Menos sensible a *outliers*.
- **Pipelines:**
 - `Pipeline` : Encadena transformaciones.
 - `ColumnTransformer` : Aplica diferentes transformaciones a columnas específicas (numéricas vs. categóricas) de una vez.
- **Transformadores Personalizados:** Crear clases que hereden de `BaseEstimator` y `TransformerMixin`.

6. Seleccionar y entrenar un modelo

- **Modelos evaluados:**
 - `LinearRegression` : Suele subajustar (*underfitting*).
 - `DecisionTreeRegressor` : Tiende a sobreajustar (*overfitting*) severamente (da error 0 en entrenamiento).
 - `RandomForestRegressor` : Un ensamble de árboles que funciona mucho mejor al promediar predicciones.

- **Validación Cruzada:** `cross_val_score()` (K-fold) permite evaluar el modelo en diferentes fragmentos de los datos de entrenamiento para obtener una medida de error robusta sin usar el conjunto de prueba.

7. Ajustar el modelo (Fine-Tune)

- **Grid Search:** `GridSearchCV` prueba todas las combinaciones posibles de hiperparámetros.
- **Randomized Search:** `RandomizedSearchCV` prueba combinaciones aleatorias, mejor para espacios de búsqueda grandes.
- **Análisis de Errores:** Ver la importancia de cada característica (`feature_importances_`) para eliminar atributos irrelevantes.

8. Evaluación final y Lanzamiento

- **Evaluación en el Test Set:** Estimar el error de generalización.
- **Intervalo de Confianza:** Usar `scipy.stats.bootstrap()` para calcular un intervalo de confianza del 95% para el RMSE.
- **Persistencia:** Guardar el modelo con `joblib.dump()` y cargarlo con `joblib.load()`.
- **Monitoreo:** Implementar alertas para detectar cuando el rendimiento caiga debido al **Data Drift** (cuando los datos nuevos cambian su naturaleza respecto a los de entrenamiento).
- **Despliegue:** El modelo puede servirse como un microservicio web con una **API REST** o en plataformas en la nube como **Vertex AI**.

▼ GPT

resumen-chuleta del **Cap. 2: "End-to-End Machine Learning Project"** del libro *Hands-On Machine Learning with Scikit-Learn and PyTorch* (Aurélien Géron, O'Reilly).

(Lo dejo en formato de repaso, nombrando **todas las ideas y "calls"/clases** que aparecen como eje del capítulo.)

1) La idea del capítulo: el “pipeline mental” de un proyecto ML real

- Un proyecto ML se organiza como una **secuencia de pasos** (checklist) para evitar olvidos, sesgos y trabajo irreproducible.
- **Pipeline** (concepto): secuencia de operaciones de procesamiento de datos (y luego el modelo) que querés poder **repetir idéntica** en entrenamiento y en producción.

Los **pasos principales** que usa el capítulo (de punta a punta) son:

1. *Look at the Big Picture* (encuadrar el problema, métrica, supuestos), 2) *Get the Data*, 3) *Discover/Visualize*, 4) *Prepare the Data*, 5) *Select & Train*, 6) *Fine-Tune*, 7) *Present*, 8) *Launch/Monitor/Maintain*.
-

2) Look at the Big Picture: encuadre, métrica y supuestos

2.1 Frame the Problem (qué estás resolviendo)

- En el ejemplo (housing), se predice **median_house_value** a partir de atributos del distrito → es **aprendizaje supervisado** (hay labels), y es **regresión**.
- Es **multiple regression** (muchas features) y **univariate regression** (una sola variable objetivo).
- Es **batch/offline learning** (entrenás “de vez en cuando” con un dataset; si el flujo fuese continuo y necesitás actualizar en línea, sería otro setting).

2.2 Select a Performance Measure (cómo medís “bueno/malo”)

- Se usa **RMSE** (raíz del MSE) como métrica típica en regresión; penaliza más los errores grandes.
- Se contrasta con **MAE** (error absoluto medio): $RMSE \approx \text{norma } \ell_2$, $MAE \approx \text{norma } \ell_1$; RMSE “castiga” más outliers.

2.3 Check the Assumptions (restricciones reales)

- Chequear supuestos/constraints del sistema antes de modelar (ej.: qué variables estarán disponibles en producción, riesgos de leakage, objetivos del negocio, etc.).
-

3) Get the Data: descarga, carga y “primer vistazo”

3.1 Workspace + descarga

- El capítulo arma funciones para automatizar la obtención del dataset:
 - `fetch_housing_data()` (con `pathlib.Path`, `urllib.request.urlretrieve`, `tarfile.open(...).extractall(...)`).
 - `load_housing_data()` (lee el CSV con `pd.read_csv(...)`).

3.2 Inspección rápida con pandas

- Métodos típicos: `head()`, `info()`, `value_counts()` (para categóricas), `describe()`.
 - Histograma para ver distribuciones: `housing_full.hist(bins=50, figsize=(12, 8))`.
 - El capítulo llama la atención sobre **escalas muy distintas** y **distribuciones sesgadas** (colas a la derecha), y sobre variables **capadas** (p. ej. valores máximos fijados).
-

4) Crear el test set bien (evitar “data snooping”)

4.1 Por qué primero se separa test

- Se insiste en separar test temprano para evitar **data snooping bias** (optimizar decisiones mirando info que después medís).

4.2 Splits “simples” y el problema de estabilidad

- Split aleatorio básico con `train_test_split(..., test_size=0.2, random_state=42)`.
- Se muestra una implementación manual: `shuffle_and_split_data(data, test_ratio)` usando `np.random.default_rng(seed=42)` + `permutation()` para barajar índices.
- Problema: si el dataset cambia con el tiempo, los splits aleatorios pueden “mover” filas entre train y test.

4.3 Split estable por hash (para datos que se actualizan)

- Solución: split determinístico por **hash** de un identificador (ej.: índice o combinaciones como lat/long), con `zlib.crc32(...)` y una función estilo `is_id_in_test_set(...)` / `split_data_with_id_hash(...)`.

4.4 Muestreo estratificado (cuando hay sesgo de representatividad)

- Se crea `income_cat` con `pd.cut(median_income, bins=[...], labels=[...])` para estratificar por nivel de ingreso (proxy importante).
- Se usa `StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)` y luego se elimina `income_cat` de los sets para no “contaminar” features.

5) Discover & Visualize: entender patrones, correlaciones y features útiles

5.1 Copias y exploración

- Trabajar sobre **copias** del train para explorar (sin tocar test). (Esta idea está reforzada por el checklist del capítulo).

5.2 Visualización geográfica

- Scatter geográfico típico: `plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)` para densidad/patrones espaciales.

5.3 Correlación y scatter matrix

- Matriz de correlación: `corr_matrix = housing.corr()` y ordenar correlaciones con el target para detectar predictores fuertes.
- `pandas.plotting.scatter_matrix(...)` para ver relaciones entre features y el target (y no quedarte solo con un número de correlación).

5.4 Feature engineering “clásico” del capítulo

- Crear combinaciones tipo ratios:
 - `rooms_per_house`, `bedrooms_per_room`, `population_per_house` (muy típico en housing).

6) Prepare the Data: limpieza + transformación reproducible

6.1 Separar X e y

- Se separa `housing_labels = strat_train_set["median_house_value"].copy()` y `housing = strat_train_set.drop("median_house_value", axis=1)`.

6.2 Missing values (opciones + scikit-learn)

- Opciones “a mano” con pandas: `dropna()`, `drop(...)`, `fillna(...)`.
- Opción estándar: `SimpleImputer(strategy="median")`, `fit()` / `transform()`; el imputer guarda estadísticas (`statistics_`).
- Se mencionan imputers más avanzados: `KNNImputer`, `IterativeImputer`.

6.3 Categóricas: Ordinal vs One-Hot

- `OrdinalEncoder` asigna enteros (útil solo si el orden tiene sentido).
- `OneHotEncoder` produce una matriz (a menudo **sparse**) con una columna por categoría.

6.4 Feature scaling & transformaciones (colas largas)

- Se discuten escalados típicos: `MinMaxScaler`, `StandardScaler`, `RobustScaler`, y que hay que “fitear” en train y aplicar en valid/test.

- Para colas largas: `FunctionTransformer(np.log, inverse_func=np.exp)` y nota de `inverse_func` (útil si pensás usar `TransformedTargetRegressor`).

6.5 RBF similarities (ingeniería de features más “potente”)

- `FunctionTransformer(rbf_kernel, kw_args=dict(Y=[...], gamma=0.1))` para crear features de **similitud Gaussiana (RBF)**; ejemplo con coordenadas (similitud a San Francisco).
- Importante: `rbf_kernel()` mide distancia en el espacio de features (si pasás 2 features, usa distancia 2D).

6.6 Custom transformers (cuando querés `fit()` + `transform()`)

- `FunctionTransformer` es rápido, pero si necesitás aprender parámetros en `fit()`, hacés una clase. Scikit-Learn usa **duck typing**: basta con `fit`, `transform`, `fit_transform`; con `TransformerMixin` y `BaseEstimator` ganás implementaciones y `get_params/set_params` (clave para tuning).
- Ejemplo didáctico: `StandardScalerClone(BaseEstimator, TransformerMixin)` mostrando `check_array`, `check_is_fitted`, `n_features_in_`, etc.
- Transformer más “pro”: `ClusterSimilarity` (usa `KMeans` + `rbf_kernel`) para crear features de similitud a centros de clusters.

6.7 Pipelines y ColumnTransformer (la pieza central)

- `Pipeline` / `make_pipeline(...)` para encadenar pasos; idea: **un solo objeto** que hace todo el preprocesamiento + modelo sin fugas.
- `ColumnTransformer` para aplicar transformaciones distintas por tipo de columna, con selectores como `make_column_selector(dtype_include=object)` y opción `remainder="drop"` / `"passthrough"`.
- `get_feature_names_out()` para recuperar nombres finales de features (importantísimo cuando hay one-hot y transforms).

7) Select & Train: modelos base + evaluación correcta

7.1 Árbol y el “engaño” del training error

- Se entrena `DecisionTreeRegressor` y da RMSE 0 en training → señal típica de **overfitting**.

7.2 Cross-validation (la evaluación “de verdad” en train)

- `cross_val_score(..., scoring="neg_root_mean_squared_error", cv=10)` devuelve RMSE negativos porque scikit-learn espera una **utility** (mayor=mejor), así que hay que cambiar el signo.
- Con CV obtenés **media** y **desviación estándar** (precisión de la estimación).

7.3 Random Forest y ensembles

- `RandomForestRegressor` (ensemble de árboles) mejora mucho; se evalúa igual con CV y RMSE.
- Ensembles reducen overfitting promediando errores poco correlacionados entre modelos.
- Aun así, puede haber gap train vs valid (se menciona que el train RMSE puede ser mucho menor → todavía hay overfitting).

8) Fine-Tune: búsqueda de hiperparámetros y análisis del mejor modelo

8.1 GridSearchCV

- `GridSearchCV` explora combinaciones dadas en `param_grid` y usa CV para evaluarlas.
- Ejemplo del capítulo: pipeline con pasos nombrados (e.g. `("preprocessing", ...), ("random_forest", RandomForestRegressor(...))`) y parámetros como `preprocessing_geo_n_clusters`, `random_forest_max_features`, etc.
- Regla clave: **doble guión bajo** `__` para referenciar hiperparámetros dentro de pipelines/transformers anidados.

8.2 RandomizedSearchCV y Successive Halving

- `RandomizedSearchCV` permite muestrear combinaciones cuando la grilla es enorme.
- `HalvingRandomSearchCV` / "successive halving": asigna más recursos a los modelos prometedores y descarta el resto.

8.3 Entender el modelo: importancias y selección de features

- Con random forest: `feature_importances_` + nombres de features del preprocesamiento para ver qué aporta más.
- Se muestra selección por importancia (ej.: ordenar importancias, elegir top-k).

8.4 "Análisis de errores" y fairness

- Recomendación: mirar rendimiento en **subgrupos** (p. ej. distritos ricos/pobres, rural/urbano), y no solo una métrica global.

9) Test final + intervalo de confianza

- Solo al final tocás el test set: `X_test = strat_test_set.drop("median_house_value", axis=1)` y `y_test = strat_test_set["median_house_value"].copy()`, luego `predict()` y RMSE.
- Se agregan **intervalos de confianza** (ej.: bootstrap) para reportar incertidumbre del RMSE, no solo un número.

10) Present, Launch, Monitor, Maintain (producción real)

10.1 Guardar/cargar el modelo

- Guardado con `joblib.dump(final_model, "my_california_housing_model.pkl")`.
- Para cargar: `joblib.load(...)` **importando antes** cualquier clase/función custom que el pipeline use (p. ej. `ClusterSimilarity`, funciones de ratio,

etc.).

10.2 Formas de desplegar

- Llamar `predict()` dentro de una app web; o desplegar como **servicio REST** para desacoplar la app del modelo y facilitar upgrades/escala.
- También se menciona despliegue en cloud (ej. servicios gestionados).

10.3 Monitoreo y data drift (lo que más se olvida)

- El rendimiento puede caer rápido (infra rota) o lento (data drift) → hay que **monitorear performance en vivo**.
- A veces se monitorea con métricas downstream (ventas, CTR, etc.); otras, con **human raters** (muestrear casos para etiquetado/QA).
- Automatización recomendada: recolectar/etiquetar data fresca, entrenar/tunear periódicamente, evaluar vs modelo anterior y desplegar si no empeora; además, monitorear **calidad de inputs** (faltantes, drift en media/std, nuevas categorías).

Si querés, en el próximo mensaje puedo convertir esto en un **"checklist de examen"** (tipo: *pregunta* → *respuesta corta* + *línea de código asociada*), manteniendo exactamente los mismos conceptos del capítulo.

▼ Classification

▼ Gemini

El capítulo 3 de *Hands-On Machine Learning* es la "Biblia" de la **Clasificación**. A diferencia del capítulo 2 (Regresión), aquí no predecimos un número, sino una **categoría**.

Aquí tienes la hoja de ruta completa, desde los datos hasta las métricas más avanzadas.

1. El Dataset: MNIST

El capítulo usa el dataset MNIST: 70,000 imágenes pequeñas de dígitos escritos a mano por estudiantes y empleados.

- **El objetivo:** Clasificar qué número es (0 al 9).
 - **Preprocesamiento:** Las imágenes de 28×28 píxeles se "aplanan" en un vector de 784 características (cada píxel es un valor de 0 a 255).
-

2. Clasificación Binaria (El "Detector de 5")

Antes de ir a los 10 dígitos, el libro enseña a identificar solo uno. ¿Es un 5 o no lo es?

Python

```
from sklearn.linear_model import SGDClassifier

# Creamos etiquetas booleanas: True para los 5, False para el resto
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Concepto clave: El `SGDClassifier` (Descenso de Gradiente Estocástico) es eficiente para datasets grandes porque maneja las instancias de una en una.

3. Medidas de Rendimiento (Donde la precisión engaña)

A. ¿Por qué la exactitud (Accuracy) no sirve?

Si el 10% de tus datos son "5" y el 90% "no son 5", un modelo que siempre diga "No es 5" tendrá un **90% de exactitud**. ¡Pero el modelo es inútil! Por eso usamos la **Matriz de Confusión**.

B. Matriz de Confusión

Python

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
cm = confusion_matrix(y_train_5, y_train_pred)
```

- **TN (Verdaderos Negativos):** Eran "no-5" y predijo "no-5".
- **FP (Falsos Positivos):** Eran "no-5" pero predijo "5" (Error Tipo I).
- **FN (Falsos Negativos):** Eran "5" pero predijo "no-5" (Error Tipo II).
- **TP (Verdaderos Positivos):** Eran "5" y predijo "5".

C. Precisión, Recall y F1-Score

Aquí es donde aplicamos las fórmulas matemáticas:

1. Precisión: ¿Qué tan exacto es el modelo cuando dice que algo es positivo?

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

2. Recall (Exhaustividad): ¿Qué porcentaje de los positivos reales fue capaz de encontrar el modelo?

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

3. F1-Score: La media armónica entre ambas. Penaliza mucho los valores bajos.

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

4. El Umbral de Decisión (Threshold)

El modelo no decide "sí o no" directamente. Calcula un **score** (puntaje).

- Si el score > **Umbral**, predice Positivo.
- Si el score < **Umbral**, predice Negativo.

El Trade-off (Compromiso):

- Si **subes** el umbral: La Precisión aumenta (eres más exigente), pero el Recall baja (se te escapan más casos).
- Si **bajas** el umbral: El Recall aumenta (atrapas a todos), pero la Precisión baja (cometes más errores).

5. Curva ROC y AUC

La curva **ROC** (Receiver Operating Characteristic) grafica la tasa de verdaderos positivos (Recall) frente a la tasa de falsos positivos (FPR).

- **AUC (Área bajo la curva):** Un clasificador perfecto tiene un AUC de **1.0**. Uno puramente aleatorio tiene **0.5**.
-

6. Clasificación Multiclase

¿Cómo clasificamos del 0 al 9 si los modelos suelen ser binarios?

1. **OvR (One-vs-Rest):** Entrenas 10 clasificadores (un detector de 0, un detector de 1...). El que tenga el score más alto gana. (Es el estándar para la mayoría).
 2. **OvO (One-vs-One):** Entrenas un clasificador por cada par (0 vs 1, 0 vs 2...). Para N clases, necesitas $N \times (N-1) / 2$ clasificadores. Útil para modelos que no escalan bien con el tamaño del dataset (como SVM).
-

7. Análisis de Errores

Para mejorar el modelo, graficamos la matriz de confusión como una imagen (heatmap).

Python

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred, normalize="true",
values_format=".0%")
plt.show()
```

Tip del libro: Si normalizas la matriz y pones a cero la diagonal principal (los aciertos), verás claramente dónde el modelo se confunde. Por ejemplo, si el modelo confunde mucho los **3** con los **5**.

8. Clasificación Multietiqueta y Multisaliada

- **Multietiqueta:** Una instancia puede pertenecer a varias clases a la vez (ej. una foto con [Alicia, Bob]).

- **Multisaliada (Multioutput):** Cada etiqueta puede tener múltiples valores (ej. un sistema que elimina ruido de una imagen: cada píxel es una etiqueta y cada valor de intensidad es el resultado).
-

¿Cuál es el siguiente paso lógico?

Ya vimos cómo medir y comparar. ¿Te gustaría que te ayude a escribir el código para **graficar la curva Precision-Recall** y encontrar el punto exacto donde la precisión llega al 90%? Sería un gran ejercicio de práctica.

▼ GPT

Voy a ubicar en el PDF el **Capítulo 3 (Classification)** y, sección por sección, te lo explico con intuición + fórmulas clave + el código típico en Scikit-Learn (y qué hace cada línea).

Ya encontré la parte de **métricas (matriz de confusión, precision/recall, umbral, PR/ROC)**. Ahora busco en el PDF las secciones restantes del capítulo 3 (multiclase, análisis de errores, multilabel y multioutput) para explicarte todo sin saltos.

Voy a ubicar el **Capítulo 3 (Classification)** dentro del PDF, extraer sus secciones (conceptos + código) y luego darte una explicación paso a paso, incluyendo métricas, umbrales y ejemplos en Scikit-Learn.

A continuación tenés una explicación **sección por sección** del **Capítulo 3 (Classification)** del libro, con **intuición + fórmulas + el código típico en Scikit-Learn explicado línea por línea**. (No copio el texto del libro: lo explico y reconstruyo el flujo y el código.)

1) Qué problema resuelve el capítulo y dataset (MNIST)

El capítulo usa **MNIST** (dígitos escritos a mano) como ejemplo guía: son imágenes pequeñas de dígitos, aplanadas como vectores de **784** píxeles (28×28). Se descarga con `fetch_openml`, y el set viene ordenado (primero entrenamiento y luego test), por eso se **mezcla (shuffle)** antes de entrenar, para evitar que el modelo vea datos "ordenados" por clase o por estilo de escritura.

Código base (cargar, separar y mirar un dígito)

```
from sklearn.datasets import fetch_openml
import numpy as np

mnist = fetch_openml("mnist_784", as_frame=False)
X, y = mnist.data, mnist.target # y viene como strings
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]

# Mezclar entrenamiento
shuffle_idx = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_idx], y_train[shuffle_idx]
```

Qué hace cada línea:

- `fetch_openml("mnist_784")`: baja el dataset.
- `X`: matriz (n_samples, 784). `y`: etiquetas (como strings, "0"... "9").
- El split "60k/10k" es el split clásico del dataset en el libro.
- `permutation` reordena índices para mezclar entrenamiento.

2) Primer clasificador: binario ("¿es un 5 o no?")

Se arranca con un problema binario: detectar el dígito 5. Se crean etiquetas booleanas:

- `True` si es "5"
- `False` si es cualquier otro dígito

Y se entrena un clasificador lineal con descenso de gradiente estocástico (`SGDClassifier`).

Código

```
from sklearn.linear_model import SGDClassifier

y_train_5 = (y_train == "5")
y_test_5 = (y_test == "5")

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Ideas clave:

- `SGDClassifier` puede entrenar modelos lineales eficientes en datasets grandes.
- Acá aprende una frontera que separa "5" vs "no-5".

3) Medir performance: por qué accuracy puede engañar (cross-validation)

3.1 Accuracy con cross-validation

Se usa `cross_val_score` para estimar desempeño con CV (por defecto, folds estratificados para clasificación).

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(sgd_clf, X_train, y_train_5, cv
                        =3, scoring="accuracy")
scores
```

3.2 El problema: clases desbalanceadas

En MNIST hay muchos más "no-5" que "5". Entonces un clasificador "tonto" que **siempre diga no-5** puede dar ~90% de accuracy (¡pero no sirve!). El capítulo lo muestra con un `Never5Classifier`.

Lección para finales:

Cuando la clase positiva es rara, accuracy no alcanza. Necesitás métricas que miren falsos positivos/negativos.

4) Matriz de confusión (Confusion Matrix)

La **confusion matrix** resume conteos de aciertos/errores. Para binario, el formato típico es:

- **TN**: no-5 predicho como no-5
- **FP**: no-5 predicho como 5
- **FN**: 5 predicho como no-5
- **TP**: 5 predicho como 5

Para obtenerla "honestamente" en entrenamiento se usa `cross_val_predict` (predicciones out-of-fold).

Código

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
cm = confusion_matrix(y_train_5, y_train_pred)
cm
```

Interpretación rápida:

- Si ves muchos **FN**, el modelo "se pierde 5s" (bajo recall).
- Si ves muchos **FP**, el modelo "se confunde y grita 5 de más" (baja precision).

5) Precision, Recall, F1 (lo que te preguntan SIEMPRE)

5.1 Precision

“De todos los que predije como positivos, ¿cuántos eran realmente positivos?”

$$\text{Precision} = \frac{TP}{TP + FP}$$

5.2 Recall (TPR, Sensibilidad)

“De todos los positivos reales, ¿cuántos detecté?”

$$\text{Recall} = \frac{TP}{TP + FN}$$

5.3 F1 score

Promedio armónico: castiga fuerte cuando una de las dos (precision/recall) es baja.

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Código (Scikit-Learn)

```
from sklearn.metrics import precision_score, recall_score, f1_score

precision = precision_score(y_train_5, y_train_pred)
recall    = recall_score(y_train_5, y_train_pred)
f1        = f1_score(y_train_5, y_train_pred)
precision, recall, f1
```

6) El corazón del capítulo: Precision/Recall trade-off y el umbral (threshold)

Muchos clasificadores no “deciden” directamente, sino que producen un **score**. En `SGDClassifier` lo obtenés con `decision_function`. Un umbral más alto hace que el modelo sea **más exigente** para decir “positivo”.

6.1 Obtener scores out-of-fold

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5,
                             cv=3, method="decision_function")
```

6.2 Convertir scores a predicción con un threshold

```
threshold = 0
y_pred_thr = (y_scores > threshold)
```

Efecto: al subir el threshold, normalmente **sube precision** y **baja recall** (porque marcás menos positivos).

6.3 Curva Precision-Recall y elegir un threshold (ej: 90% precision)

El capítulo usa `precision_recall_curve` para obtener `precisions`, `recalls`, `thresholds`.

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(
    y_train_5, y_scores)

# ejemplo: buscar el threshold que te da ~90% precision
idx = np.argmax(precisions >= 0.90)
threshold_90 = thresholds[idx]
```

La idea de "threshold_for_90_precision" aparece explícitamente en el flujo del capítulo.

Punto importante: cuando decís "quiero 90% precision", la pregunta natural es:

| "ok, ¿y qué recall obtenés con eso?"

7) (Nuevo en sklearn) Wrappers para umbral:

FixedThresholdClassifier y
TunedThresholdClassifierCV

El capítulo menciona que desde **Scikit-Learn 1.5** hay dos clases para manejar umbrales más cómodo:

- **FixedThresholdClassifier:** fijás el umbral manualmente.
- **TunedThresholdClassifierCV:** busca automáticamente el umbral óptimo con cross-validation, maximizando una métrica (por defecto **balanced accuracy**).

Cuándo usar cuál:

- Tenés un objetivo operativo (ej. "máximo 1% FP") → fijás threshold.
- Querés el mejor equilibrio para una métrica (balanced accuracy, F1, etc.) → tuneado CV.

| Nota práctica: si tu modelo tiene predict_proba, el umbral suele ser probabilidad (0–1); si no, suele ser un score de decision_function.

8) ROC curve y ROC AUC

La **ROC curve** grafica:

- **TPR** (que es recall)
- vs **FPR** (tasa de falsos positivos)

La métrica **AUC** (área bajo la curva) resume la curva: cuanto más cerca de 1, mejor.

Código típico

```
from sklearn.metrics import roc_curve, roc_auc_score

fpr, tpr, roc_thresholds = roc_curve(y_train_5, y_scores)
auc = roc_auc_score(y_train_5, y_scores)
```

PR vs ROC: regla de oro del capítulo

- Si la clase positiva es **rara** (desbalance fuerte), suele ser más informativa la **PR curve** que ROC.

9) Comparar clasificadores:

RandomForestClassifier vs SGD y probabilidades

Para modelos como Random Forest, se pueden usar **probabilidades** con `predict_proba`. El capítulo muestra cómo obtener scores con `cross_val_predict(..., method="predict_proba")` y comparar curvas/métricas.

Advertencia clave: esas "probabilidades" pueden no estar calibradas; a veces conviene calibrarlas con `CalibratedClassifierCV`.

10) Multiclass classification (0–9): OvR vs OvO

Ahora el problema real: 10 clases. Estrategias:

- **OvR (One-vs-Rest / One-vs-All):** entrenás 10 clasificadores binarios (cada clase vs resto).
- **OvO (One-vs-One):** entrenás un clasificador por cada par de clases → para 10 clases son **45** modelos.

SVM (`svc`) usa OvO por defecto (mejor performance, aunque entrena más modelos).

Scores por clase: `decision_function` devuelve 10 valores

Para un modelo multiclase, `decision_function` suele devolver un vector con un score por clase, y se elige el máximo.

Scaling ayuda (muy típico para SGD/SVM)

El capítulo escala los inputs con `StandardScaler` (porque muchos modelos lineales mejoran mucho con features escaladas) y muestra mejora de accuracy vía CV.

11) Error analysis (analizar dónde falla)

Acá el objetivo no es “otra métrica”, sino **entender errores**.

11.1 Matriz de confusión normalizada

Para multiclase, se usa `ConfusionMatrixDisplay.from_predictions(..., normalize="true")` para ver proporciones por clase real.

11.2 Resaltar errores (quitando diagonal)

El capítulo sugiere “apagar” la diagonal (aciertos) para que resalten confusiones entre clases; también usa pesos como `np.log` para destacar errores.

11.3 Mirar ejemplos mal clasificados

Se construyen cuadrículas de imágenes para ver patrones: por ejemplo, muchos errores vienen de dígitos escritos de forma ambigua o desplazados.

Qué conclusiones típicas salen (muy de examen):

- El modelo puede confundir por **similitud visual** o por **desplazamiento**.
 - Solución clásica: **data augmentation** (copias desplazadas/rotadas) para robustez.
-

12) Multilabel classification (múltiples “tags” por instancia)

Multilabel = una imagen puede tener varios labels booleanos a la vez (ejemplo conceptual: una foto con Alice y Charlie → `[True, False, True]`).

Ejemplo con MNIST: dos etiquetas por dígito:

- "es grande" (7/8/9)
- "es impar"

Código (crear labels y entrenar KNN)

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= "7")
y_train_odd    = (y_train.astype("int8") % 2 == 1)
y_multilabel   = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

Métricas multilabel: promedios macro/weighted con F1

- `average="macro"` : promedia labels por igual.
- `average="weighted"` : pondera por soporte (útil si algunos labels aparecen mucho más).

Dependencias entre labels: `ClassifierChain`

Si los labels están correlacionados, una cadena de clasificadores puede capturar dependencias. El capítulo explica el porqué y muestra

`ClassifierChain` .

13) Multioutput classification (salida "tipo vector" con muchas componentes)

Multioutput = generalización de multilabel donde **cada label puede ser multiclase**.

Ejemplo: **denoising** de imágenes:

- **Input:** imagen con ruido (784 features).
- **Output:** imagen limpia (784 salidas), donde cada "label/píxel" puede tomar valores 0–255.

Código (crear datos con ruido + entrenar KNN)

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

rng = np.random.default_rng(seed=42)
noise_train = rng.integers(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise_train

noise_test = rng.integers(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise_test

y_train_mod = X_train
y_test_mod = X_test

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_mod, y_train_mod)

clean_digit = knn_clf.predict([X_test_mod[0]])
```

14) Ejercicios del capítulo (qué te están entrenando)

Sin copiar el enunciado literal, el bloque final te hace practicar:

1. Lograr >97% en MNIST (pista: KNN + búsqueda de hiperparámetros).
2. Implementar "shift" y hacer data augmentation (mejora robustez).
3. Resolver Titanic (pipeline de features + clasificación).

4. Spam classifier (pipeline de texto, extracción de features, evaluación).

Si querés, lo hacemos “como para rendir final”

Decime si tu profe evalúa más:

- **métricas y umbrales** (precision/recall/ROC),
- o **estrategias multiclase y análisis de errores**,

y te armo un **resumen tipo machete** + preguntas típicas + mini-ejercicios con solución.

▼ GPT tips

Perfecto. Te dejo una **chuleta completa y comentada** de **todas las funciones de Scikit-Learn + tips prácticos** que aparecen en el **Capítulo 2 y 3**, explicadas **una por una**, con **para qué sirven, cuándo usarlas y trampas típicas**.

Esto está pensado **para estudiar y rendir**, y también para **trabajo real**.

CAPÍTULO 2 – *End-to-End Machine Learning Project*

👉 En este capítulo **NO importa el modelo**, importa **el proceso correcto**.

Carga y división de datos

train_test_split

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

◆ Qué hace

Divide los datos en **train** y **test** de forma aleatoria.

◆ Tips de examen / industria

- `random_state`: garantiza **reproducibilidad**
- **Nunca** toques el test hasta el final
- El test **simula datos futuros**

2 Stratified Sampling (clase importante)

StratifiedShuffleSplit

```
from sklearn.model_selection import StratifiedShuffleSplit  
split = StratifiedShuffleSplit(n_splits=1, test_size=0.  
2, random_state=42)
```

◆ Qué hace

Asegura que **la proporción de clases importantes** se mantenga igual en train y test.

◆ Tip clave

👉 **Siempre usar estratificación cuando haya clases desbalanceadas**

3 Pipelines (CLAVE en ML real)

Pipeline

```
from sklearn.pipeline import Pipeline  
  
pipeline = Pipeline([  
    ("imputer", SimpleImputer(strategy="median")),  
    ("scaler", StandardScaler()),
```

```
    ("model", LinearRegression())  
])
```

◆ Qué hace

Encadena pasos:

1. Limpieza
2. Escalado
3. Modelo

◆ Tips IMPORTANTES

- Evita **data leakage**
- Todo se entrena SOLO con train
- `fit()` → entrena todo
- `predict()` → aplica todo automáticamente

👉 Pipeline = estándar profesional

4 Columnas numéricas y categóricas

ColumnTransformer

```
from sklearn.compose import ColumnTransformer
```

Permite aplicar **distintos pipelines** a **distintas columnas**.

◆ Tip de oro

Nunca mezcles:

- escalado numérico
- one-hot categórico

5 Imputación de valores faltantes

SimpleImputer

```
from sklearn.impute import SimpleImputer
SimpleImputer(strategy="median")
```

♦ Estrategias

- "mean"
- "median" ✓ más robusto
- "most_frequent"

⚠ Error típico

✗ Imputar antes del split

✓ Imputar **dentro del pipeline**

6 Escalado de features

StandardScaler

```
from sklearn.preprocessing import StandardScaler
```

♦ Qué hace

Convierte:

[
 $x \rightarrow \frac{x - \mu}{\sigma}$
]

♦ Cuándo usar

- SGD
- SVM
- KNN

- Regresión logística

✗ No necesario

- Árboles
 - Random Forest
-

7 Evaluación

cross_val_score

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X_train, y_train, cv=5)
```

◆ Qué hace

Evalúa con **cross-validation**.

◆ Tip de examen

- CV \neq test
 - CV se usa para **ajuste**
 - Test solo al final
-

8 Grid Search

GridSearchCV

```
from sklearn.model_selection import GridSearchCV
```

Busca la mejor combinación de hiperparámetros.

◆ Tip profesional

Siempre usar GridSearch **con Pipeline**.

CAPÍTULO 3 – *Classification*

👉 Acá entran **métricas, umbrales y errores reales**.

9 Clasificador base

SGDClassifier

```
from sklearn.linear_model import SGDClassifier
```

♦ Qué es

Modelo lineal entrenado con **descenso de gradiente estocástico**.

♦ Tip

- Rápido
 - Escala bien
 - Sensible al escalado
-

10 Validación cruzada con predicciones

cross_val_predict

```
from sklearn.model_selection import cross_val_predict  
y_pred = cross_val_predict(model, X, y, cv=3)
```

♦ Diferencia con **cross_val_score**

Función	Devuelve
cross_val_score	métricas
cross_val_predict	predicciones

👉 Necesario para matriz de confusión

1 1 Matriz de confusión

`confusion_matrix`

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_true, y_pred)
```

♦ Te da

- TN, FP
- FN, TP

♦ Tip de final

Si el dataset está desbalanceado → **accuracy no sirve**

1 2 Métricas fundamentales

Precision

```
precision_score(y_true, y_pred)
```

| “De los positivos predichos, cuántos son correctos”

Recall

```
recall_score(y_true, y_pred)
```

| “De los positivos reales, cuántos detecté”

F1

```
f1_score(y_true, y_pred)
```

| Balance entre precision y recall

1 3 Scores en lugar de predicciones

`decision_function`

```
scores = model.decision_function(X)
```

♦ Qué devuelve

Un **score continuo**, no una clase.

♦ Por qué importa

Permite:

- Ajustar umbral
- PR curve
- ROC curve

1 4 Precision–Recall Curve

`precision_recall_curve`

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(
    y_true, scores)
```

♦ Tip clave

- Mejor que ROC cuando hay **desbalance**
- El umbral controla el trade-off

1 5 ROC Curve y AUC

`roc_curve`

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_true, scores)
```

roc_auc_score

```
roc_auc_score(y_true, scores)
```

◆ Tip

ROC puede mentir si la clase positiva es rara.

1 6 Clasificación multiclase

OvR vs OvO

- **One-vs-Rest:** 1 clase vs resto
- **One-vs-One:** pares de clases (SVM)

decision_function multiclase

Devuelve un score **por clase** → se elige el máximo.

1 7 Análisis de errores

ConfusionMatrixDisplay

```
from sklearn.metrics import ConfusionMatrixDisplay
```

◆ Tip clave

- Normalizar la matriz
- Apagar la diagonal para ver errores reales

1 8 Multilabel classification

KNeighborsClassifier

```
from sklearn.neighbors import KNeighborsClassifier
```

Predice **varias etiquetas por muestra**.

♦ Métricas multilabel

```
f1_score(y_true, y_pred, average="macro")
```

19 Multioutput (denoising)

👉 El target es una imagen entera

KNN:

- Busca imágenes ruidosas parecidas
- Devuelve las imágenes limpias asociadas

🧠 RESUMEN DE ORO (para examen)

! Reglas que siempre suman puntos

- Nunca evaluar en test antes del final
- Pipelines evitan data leakage
- Accuracy no sirve en datasets desbalanceados
- Precision vs Recall depende del problema
- PR curve > ROC en clases raras
- KNN no "aprende", memoriza

Si querés, el próximo paso puede ser:

- 📌 Resumen ultra corto tipo machete
- 📌 Preguntas típicas de parcial/final

-  **Ejercicios resueltos solo con sklearn**

Decime cómo lo necesitás.