

MNIST Classification

Metrics & Concepts Deep-Dive Guide

Binary · Multiclass · Multilabel · Multioutput

Extra-depth focus: Precision, Recall, F1, k-fold CV, Decision Function

Section	Covers
1. Binary Classification	SGD vs Random Forest vs Dummy baseline — detecting digit 8
2. Cross-Validation	cross_val_score, cross_val_predict, StratifiedKFold
3. Core Metrics	Confusion matrix, precision, recall, F1 — explained number by number
4. Decision Function	How a raw score becomes a yes/no decision
5. PR & ROC Curves	Visualizing the precision/recall and TPR/FPR tradeoffs
6. Multiclass	One-vs-One / One-vs-Rest with SVC, 10-class confusion matrix
7. Multilabel	KNN predicting two independent labels at once
8. Multioutput	KNN image denoising — predicting a full image

01 Binary Classification Setup

The task

MNIST has 10 digit classes. To start simple, the original problem is collapsed into a yes/no question: **is this image an 8, or not?** This is done with `y_train8 = (y_train == "8")` — a boolean array, True where the label is 8.

Three models compared

Model	Type	Role here
DummyClassifier	Baseline, no learning	Exposes whether accuracy alone is meaningful
SGDClassifier	Linear model	Fast, simple linear decision boundary
RandomForestClassifier	Tree ensemble	Non-linear, generally stronger on pixel data

Why a Dummy baseline matters

About 90% of MNIST digits are *not* an 8. A classifier that always guesses "not 8" scores ~90% accuracy while learning nothing. If your real model only matches the dummy's score, it has not actually learned the pattern — this is the single most common beginner trap with imbalanced classes.

02 Cross-Validation — In Depth

Why not just train once and check?

A model evaluated on the data it trained on will look artificially good — it has already seen the answers. Cross-validation (CV) repeatedly trains on part of the data and tests on a different part it has never seen, giving an honest estimate.

StratifiedKFold — the manual version

k-fold CV splits the training data into **k** equal chunks ("folds"). For each of the k rounds: train on k-1 folds, test on the 1 remaining fold. Every sample gets tested exactly once, across the k rounds combined.

Stratified means each fold keeps the same ratio of 8s to non-8s as the full dataset. Without this, a fold might randomly end up with very few real 8s, making that fold's score unreliable just from bad luck in the split — not from the model actually being worse.

```
skf = StratifiedKFold(n_splits=5)
for train_i, test_i in skf.split(X_train, y_train8):
    cloned_model = clone(sgd) # fresh, untrained copy
    cloned_model.fit(X_train[train_i], y_train8[train_i])
    y_pred = cloned_model.predict(X_train[test_i])
    accuracy = (y_pred == y_train8[test_i]).mean()
```

cross_val_score vs. cross_val_predict

Function	Returns	Use it when...
<code>cross_val_score</code>	One score per fold (e.g. 5 accuracy numbers)	You just want to know how good the model is

cross_val_predict	One prediction per training sample	You need the actual predictions, e.g. to build a confusion matrix
-------------------	------------------------------------	---

Critically, cross_val_predict never lets a sample's prediction come from a model that trained on that same sample — it always comes from the fold where that sample was held out. This is what makes it safe to compute metrics like precision and recall directly from its output.

03 The Confusion Matrix

Every metric in this guide is built from four numbers. For the binary "is it an 8" problem, the confusion matrix looks like this:

	Predicted: Not 8	Predicted: 8
Actual: Not 8	True Negative (TN)	False Positive (FP)
Actual: 8	False Negative (FN)	True Positive (TP)

Term	Plain meaning
TP (True Positive)	Said "8", and it really was an 8
TN (True Negative)	Said "not 8", and it really wasn't
FP (False Positive)	Said "8", but it wasn't — a false alarm
FN (False Negative)	Said "not 8", but it actually was — a miss

04 Precision, Recall & F1 — The Deep Dive

Precision — "When I say 8, am I right?"

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

Precision only looks at the predictions that were **positive** (said "8") and asks what fraction of those were actually correct.

■ DEEP DIVE — What a precision number actually means

If precision = 0.85, picture it like this: **out of every 100 times the model says "this is an 8", 85 of those really are 8s, and 15 are mistakes (false alarms)**. The higher the number, the more you can trust a positive prediction. Precision says nothing about digits the model never flagged at all — it could be missing tons of real 8s and still have perfect precision, as long as it's never wrong about the ones it does flag.

Recall — "Of all the real 8s, how many did I catch?"

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

Recall only looks at the actual **positives** (real 8s) and asks what fraction the model managed to find.

■ DEEP DIVE — What a recall number actually means

If recall = 0.75, picture it like this: **out of every 100 real 8s in the data, the model correctly identified 75 of them, and missed the other 25.** Recall says nothing about false alarms — a model could flag almost everything as an 8 (catching nearly all real 8s, recall near 1.0) while being wrong constantly, and recall alone wouldn't penalise that.

Why they trade off against each other

Precision and recall pull in opposite directions because of how a model decides "yes" vs "no":

- Make the model more **cautious** (only say "8" when very confident) → fewer false alarms (precision goes up), but more real 8s slip through uncaught (recall goes down).
- Make the model more **liberal** (say "8" more easily) → catches more real 8s (recall goes up), but more false alarms slip in (precision goes down).

There is no threshold that maximises both at once — improving one costs the other. Which one matters more depends entirely on the problem:

Scenario	Which matters more	Why
Cancer screening test	Recall	Missing a real case (FN) is far worse than a false alarm (FP)
Spam email filter	Precision	Blocking a real email (FP) is worse than letting one spam through (FN)
This MNIST "is it an 8" task	Depends on the goal	No inherent real-world cost here — just a learning exercise

F1 Score — one number that balances both

$$F1 = 2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$$

F1 is the **harmonic mean** of precision and recall — not the plain average. The harmonic mean has a specific property: it stays low if *either* input is low, even if the other is high.

■ DEEP DIVE — Why harmonic mean, not regular average?

Example: precision = 1.0, recall = 0.0. A plain average would give 0.5 — sounds decent. But this model is useless (it never actually catches any real 8s). The harmonic mean instead gives $F1 = 0.0$, correctly reflecting that the model is broken. F1 only looks good when **both** precision and recall are reasonably good — it can't be "propped up" by one strong number hiding a weak one.

Use F1 when you want one summary number and don't have a strong reason to favor precision or recall specifically. Use precision or recall directly when you do.

05 The Decision Function — Deep Dive

How a model actually decides yes or no

A model like SGD doesn't internally think in "yes" or "no" — it computes a single **raw numeric score** for each input, and a yes/no answer is derived from that score afterward. This raw score is what `decision_function()` returns.

```
sgd.decision_function([X_train[0]])  
# e.g. returns something like [-3.42]
```

■ DEEP DIVE — Reading a decision function number

The sign and size of the number both matter:

- A large **positive** number (e.g. +8.1) → the model is very confident this is an 8
- A small **positive** number (e.g. +0.3) → the model leans "8" but isn't sure
- A small **negative** number (e.g. -0.3) → the model leans "not 8" but isn't sure
- A large **negative** number (e.g. -8.1) → the model is very confident this is NOT an 8

`.predict()` is just this score compared against a threshold (default 0): $\text{score} > 0 \rightarrow \text{predict "8"}; \text{score} \leq 0 \rightarrow \text{predict "not 8"}$.

Why this threshold is the lever behind precision/recall

Because 0 is just the *default* cutoff, you can move it yourself:

- Raise the threshold (e.g. require score > 3 instead of > 0) → the model only says "8" when very confident → precision goes up, recall goes down
- Lower the threshold (e.g. score > -2) → the model says "8" more easily → recall goes up, precision goes down

The Precision-Recall curve (next section) is generated by sweeping this threshold across every possible value and recording precision/recall at each point.

Random Forest's equivalent — `predict_proba`

Tree ensembles like Random Forest don't compute a single linear score, so they have no `decision_function`. Instead, `predict_proba()` returns a probability for each class, summing to 1:

```
y_probas = rf.predict_proba(X) # shape (n_samples, 2)
# column 0 = P(not 8), column 1 = P(is 8)
y_scores = y_probas[:, 1] # use this exactly like decision_function output
```

The same threshold-sweeping logic applies — just on probability (0 to 1) instead of an unbounded raw score.

06 PR Curve, ROC Curve & AUC

Precision-Recall (PR) Curve

`precision_recall_curve()` computes precision and recall at every possible threshold value in one call, instead of you picking one threshold manually. Plotting recall (x-axis) against precision (y-axis) traces the full tradeoff curve.

■ DEEP DIVE — Reading a PR curve

A curve that stays close to the top-right corner (high precision AND high recall together) means a strong model. A curve that sags toward the bottom-left early means the model has to sacrifice a lot of one to gain a little of the other — a weaker model.

ROC Curve

ROC plots **True Positive Rate** (= recall) against **False Positive Rate**:

$$\text{FPR} = \text{FP} / (\text{FP} + \text{TN})$$

FPR asks: of all the actual negatives (non-8s), what fraction got wrongly flagged as positive? It's recall's mirror image, computed on the negative class instead.

■ DEEP DIVE — Reading a ROC curve

The dashed diagonal line (FPR = TPR) represents a model making pure random guesses — no better than a coin flip. The further the actual curve bulges toward the top-left corner (high TPR, low FPR achieved simultaneously), the better the model is at separating the two classes.

ROC AUC — the curve as one number

AUC = Area Under the (ROC) Curve. It condenses the entire curve into a single score between 0.5 and 1.0:

AUC value	What it means
0.5	No better than random guessing
0.7 – 0.8	Acceptable separation between classes
0.8 – 0.9	Good separation
0.9 +	Excellent — common on MNIST since digits are visually distinct

PR curve vs. ROC curve — which to trust?

On heavily imbalanced data (like "is it an 8" — only ~10% positive), the PR curve is usually more informative. ROC's FPR is calculated against the large negative class, so it can look deceptively good even when the model performs poorly on the rare positive class. PR curve, built entirely from positive-class performance, exposes that weakness more clearly.

07 Multiclass Classification

From yes/no to 10 classes

Multiclass means choosing one answer out of more than two options — here, which digit (0–9) an image shows. Some algorithms (like SVC) are fundamentally binary under the hood, so scikit-learn automatically combines many binary classifiers:

Strategy	How it works	Classifiers needed (10 classes)
One-vs-One (OvO)	Train one classifier per pair of classes; they vote	$10 \times 9 / 2 = 45$
One-vs-Rest (OvR)	Train one classifier per class: "is it this class, or anything else?"	10

SVC defaults to OvO internally. Wrapping it in `OneVsRestClassifier` forces the OvR strategy instead — useful to compare, or when a specific algorithm only supports one strategy natively.

Multiclass `decision_function`

For multiclass, `decision_function()` returns one score **per class** instead of a single number — 10 scores for 10 digits. The predicted class is simply whichever score is highest.

The 10×10 Confusion Matrix

With 10 classes the confusion matrix grows to a 10×10 grid. The diagonal holds correct predictions; everything off-diagonal shows exactly which digit got confused for which other digit (e.g. a 4 mistaken for a 9).

Normalizing by row (**`normalize="true"`**) converts raw counts into a percentage of each digit's total occurrences, making error rates comparable across digits that appear different numbers of times. Weighting by errors only (**`sample_weight=(y_pred != y_true)`**) further highlights misclassification patterns by suppressing the (usually large) correct-prediction diagonal visually.

08 Multilabel Classification

Multiple independent yes/no answers at once

Multilabel is different from multiclass: instead of picking *one* answer from several options, each sample gets *several independent* yes/no labels simultaneously. A digit can be both "greater than 5" AND "even" at the same time — these labels don't compete with each other.

```
y_above_5 = (y_train > '5')
y_even = (y_train.astype(int) % 2 == 0)
y_train_ml = np.c_[y_above_5, y_even] # shape (n_samples, 2)
```

F1 with `average='macro'`

With two labels, F1 must be computed per-label first, then combined into a single number. **`average="macro"`** computes F1 separately for each label and takes the plain average — every label counts equally regardless of how common it is.

The alternative, **`average="weighted"`**, would instead weight each label's F1 by how many samples have that label — useful when labels are imbalanced and you want common labels to count more.

ClassifierChain — when labels are related

KNN above treats both labels as completely independent — predicting "is it even?" with zero knowledge of the "above 5?" prediction. **ClassifierChain** instead predicts labels one at a time, feeding each completed prediction as an extra input feature to the next classifier in the chain. This helps when labels are correlated — e.g. knowing a digit is "above 5" already narrows down whether it's even, since 6 and 8 satisfy both conditions.

09 Multioutput Classification

Predicting a whole structured output

Multioutput pushes the idea further still: instead of predicting a handful of labels, the model predicts an entire array — in this case, 784 pixel values (an entire image), not a category at all.

The denoising task

Random noise is added to clean digit images. KNN is then trained where the **input** is the noisy image and the **"label"** is the original clean image — turning denoising into a (very large) prediction problem.

```
image_noise = np.random.randint(0, 100, X_train.shape)
X_train_noisy = X_train + image_noise

knn.fit(X_train_noisy, X_train) # label = the clean image itself
denoised = knn.predict([X_test_noisy[0]])
```

KNN works here because it predicts by finding the most similar training examples and combining their labels — and since labels are pixel arrays, it effectively blends together the clean images of the most visually similar noisy neighbours.

10 Quick-Reference Summary

Concept	One-line takeaway
Accuracy	Misleading on imbalanced data — always sanity-check against a Dummy baseline
Precision	$TP/(TP+FP)$ — of predicted positives, how many were right
Recall	$TP/(TP+FN)$ — of actual positives, how many were caught
F1	Harmonic mean of precision & recall — low if either one is weak
Decision threshold	The cutoff turning a raw score into yes/no — moving it trades precision for recall
decision_function	Raw signed score from a linear model; sign+magnitude = direction+confidence
predict_proba	Probability-based equivalent for non-linear models like Random Forest
PR curve	Precision vs recall swept across every threshold
ROC curve / AUC	TPR vs FPR; AUC condenses the curve to one number (0.5=random, 1.0=perfect)
k-fold CV	Rotates train/test splits so every sample is evaluated fairly without leakage
Multiclass (OvO/OvR)	Splitting an N-class problem into many binary sub-problems
Multilabel	Several independent yes/no labels per sample
Multioutput	Predicting a full structured output (e.g. an image), not a category

You've now covered the full classification toolkit.

Precision/recall/F1 and decision thresholds come up in almost every classification problem you'll meet from here on.