2025-04-10 13:46

# tags :

- Machine Learning
- Hands on ML - book

# Chapter 3 - Hands on

- **MNIST**
    - The `sklearn.datasets` package offers three main types of functions:
        - `fetch_*` **functions** (e.g., `fetch_openml()`): Download real-world datasets.
        - `load_*` **functions**: Load built-in small toy datasets (no internet needed).
        - `make_*` **functions**: Generate synthetic datasets for testing.
    - Generated datasets usually come as `(X, y)` NumPy arrays, while others are returned as `Bunch` objects (dictionary-like), typically containing:
        - `"DESCR"` : Dataset description
        - `"data"` : Input features (2D array)
        - `"target"` : Labels (1D array)
    - The `fetch_openml()` function typically returns inputs as a Pandas DataFrame and labels as a Series. However, for image datasets like MNIST, it's better to set `as_frame=False` to get NumPy arrays, which are more suitable for image data.

      ```
      from sklearn.datasets import fetch_openml
      mnist = fetch_openml('mnist_784', as_frame=False)
      X, y = mnist.data, mnist.target # X.shape->(70000, 784) y.shape->(70000,)
      ```

    - Each of the 70,000 MNIST images has 784 features (28×28 pixels), where each feature represents a pixel's intensity from 0 (white) to 255 (black). To visualize an image, reshape its feature vector to 28×28 and display it using `matplotlib.pyplot.imshow()` with `cmap="binary"` for grayscale. Here's a sample function:

      ```
      def plot_digit(image_data):
            image = image_data.reshape(28, 28)
            plt.imshow(image, cmap="binary")
            plt.axis("off")

        some_digit = X[0]
      ```

```
plot_digit(some_digit)
plt.show()
```

# Training a Binary Classifier

- To build a simple digit classifier, we focus on detecting the number **5**, turning it into a **binary classification** problem (5 vs. not 5). We create target labels using:
- Then, we use Scikit-Learn’s `SGDClassifier`, which is efficient for large datasets and supports online learning:

```
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)

sgd_clf.predict([some_digit])  # Output: array([True])
```

# Performance Measures

## Measuring Accuracy Using Cross-Validation

- To evaluate the **SGDClassifier**, we use **3-fold cross-validation** with `cross_val_score`, which gives over **95% accuracy**—seemingly impressive

```
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
# Output: array([0.95035, 0.96035, 0.9604 ])
```

- However, a **DummyClassifier** that always predicts "not 5" still achieves **over 90% accuracy**, because only ~10% of the data are actually 5s:

```
cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
# Output: array([0.90965, 0.90965, 0.90965])
```

- **Conclusion**: Accuracy can be misleading on **imbalanced datasets**. A better alternative for evaluation is the **confusion matrix**, which gives more insight into model performance.
- **IMPLEMENTING CROSS-VALIDATION**
    - To manually implement **cross-validation** with more control, you can use `StratifiedKFold`, which ensures each fold has a representative class distribution Like CROSS-VALIDATION. Here's the process:
        1. Use `StratifiedKFold(n_splits=3)` to split the data into 3 balanced folds.
        2. For each fold:
           - Clone the classifier.

- Train on the training folds.
- Predict on the test fold.
- Calculate and print the accuracy.

This replicates what `cross_val_score()` does, but allows customization. Example output:
`0.95035`, `0.96035`, `0.9604` — matching the automated method.

```python
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone
skfolds = StratifiedKFold(n_splits=3) # add shuffle=True if the dataset is not already shuffled
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]
    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # prints 0.95035, 0.96035, and 0.9604
```

## Confusion Matrices

- A **confusion matrix** shows how often instances of class **A** are predicted as class **B**.
- You use it to evaluate classification performance in more detail than accuracy.
- **Code to build it:**

```python
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3) #it is 1d array contain 1 value for each instance, we pass the label to train the model

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_train_5, y_train_pred)

[[53892   687]   # Non-5s: 53892 correct (TN), 687 mistaken as 5s (FP)
 [1891   3530]]  # 5s: 1891 missed (FN), 3530 correct (TP)
```

- **Interpretation**:
  - Rows = actual class
  - Columns = predicted class
  - Diagonal = correct predictions

    - Off-diagonal = errors (false positives/negatives)
- A **perfect classifier** would have only true positives and true negatives
- While the **confusion matrix** is detailed, you might want simpler metrics:
    - **Precision** (a.k.a. positive prediction accuracy):
        -
        - Measures how **many predicted positives are actually correct**.
        - A model can get **100% precision** by making **very few positive predictions**—but it may miss many actual positives.
    - **Recall** (a.k.a. sensitivity or true positive rate):
        -
        - Measures how **many actual positives the model correctly detects**.
        - High recall means **fewer false negatives**.

## Precision and Recall

- After evaluating the 5-detector using accuracy, it's time to dig deeper using **precision**, **recall**, and the **F1 score**.
- **Metrics:**
    - **Precision**:
        -

          → Of all predicted 5s, **83.7% were correct**.
    - **Recall**:
        -

          → The model found **65.1% of actual 5s**.
- **F1 Score** (harmonic mean of precision and recall):
    - Balances both precision and recall.
    - Favors models with **similar** values for both.
- You **can't maximize both** at the same time.
    - Choose based on context:
        - **High precision, low recall**: Good for **kid-safe content filters**.
        - **High recall, low precision**: Okay for **shoplifter detection**, where missing one is worse than a false alarm.

## The Precision/Recall Trade-off

- **SGDClassifier** computes a **decision score** for each input.
- If the score is **above a threshold**, it predicts **positive** (e.g., “5”); otherwise, **negative**.
- **Raising the threshold**:
    - ✅ Increases **precision** (fewer false positives)

    - ❌ Decreases **recall** (more false negatives)
- **Lowering the threshold**:
    - ✅ Increases **recall**
    - ❌ Decreases **precision**
- **Example:**
    - If threshold is at default `0`: prediction is **True** (high recall).
    - Raising it to `3000`: prediction becomes **False** (misses a correct 5 → lower recall).
- **How to Tune the Threshold:**
    1. **Get decision scores** (instead of predictions):

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

    2. **Compute precision and recall for all thresholds**:

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

    3. **Visualize** precision & recall against thresholds:

        ```
        plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
        plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
        # note here we exclude the last value in the precisions and recalls
        because recalls nad precisions have length N+1 but thresholds have
        length = N
        plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")
        # plt.vlines(x, ymin, ymax, color, linestyle, label)
        ```

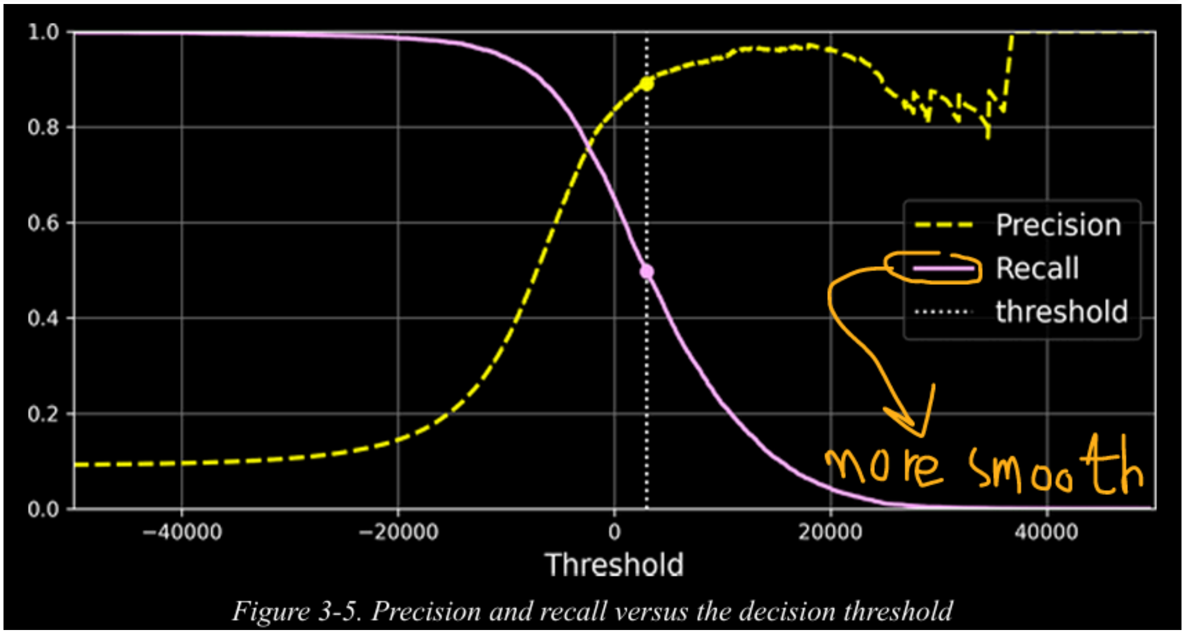
- 

*Figure 3-5. Precision and recall versus the decision threshold*

4. **Or plot precision directly vs. recall** to spot the sweet spot before a sharp drop in precision.

```
plt.plot(recalls, precisions, linewidth=2, label="Precision/Recall
curve")
#[...] beautify the figure: add labels, grid, legend, arrow, and text
plt.show()
```

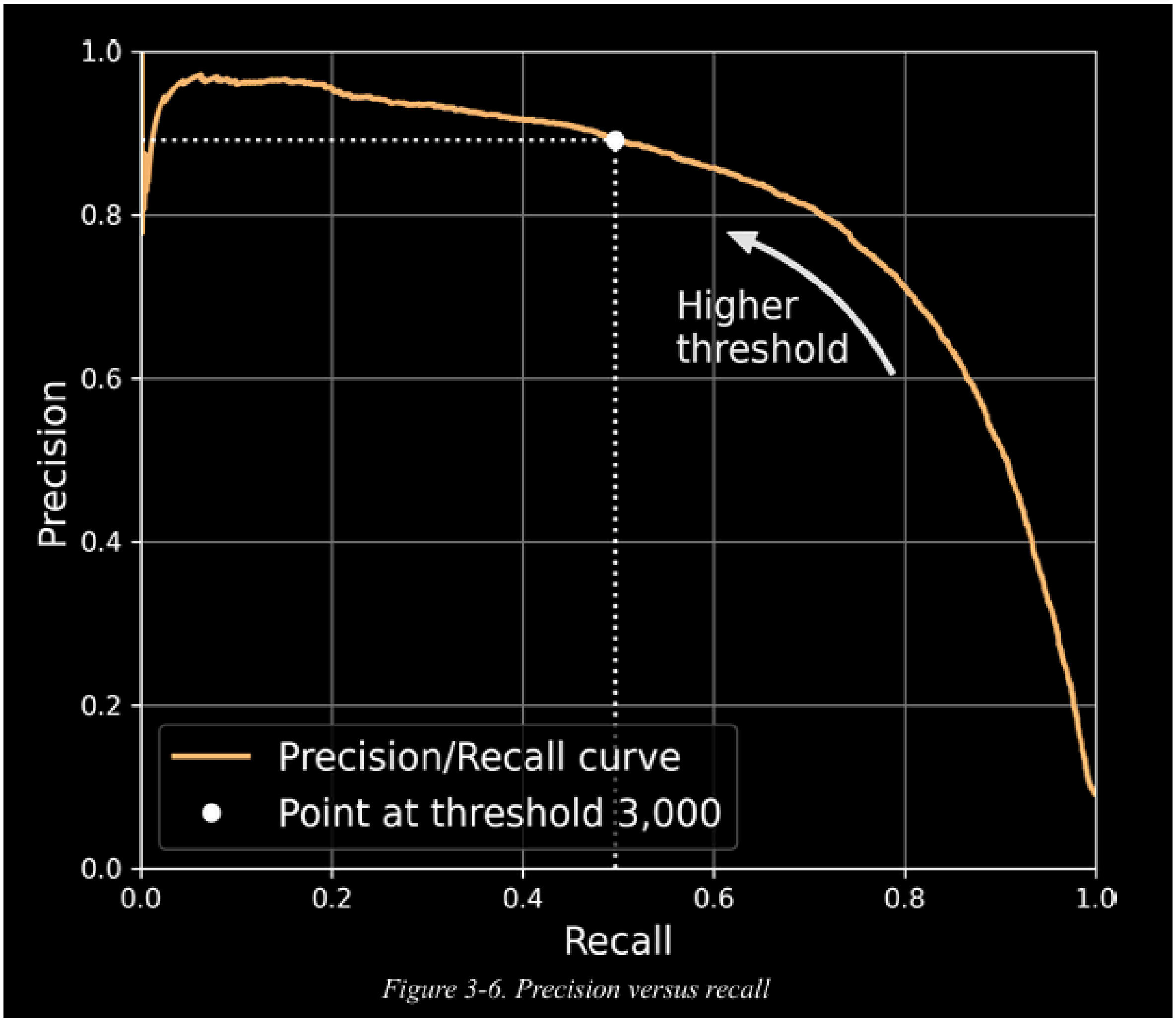


*Figure 3-6. Precision versus recall*

- Key Insight:
  - There's **no universal best threshold**.
  - Choose it based on **what matters more**:
    - **Precision**: when **false positives are costly** (e.g., kid-safe content).
    - **Recall**: when **missing positives is worse** (e.g., medical diagnosis or shoplifter detection).
- You can **tune your classifier** to reach **90% precision** by:

```
idx = (precisions >= 0.90).argmax()
threshold = thresholds[idx]
y_pred_90 = (y_scores >= threshold)
# hecking performance Precision: 90% and Recall: Only ~48%
```

## The ROC Curve

- It plots **True Positive Rate (TPR) = Recall** vs **False Positive Rate (FPR)**.
- FPR = 1 - Specificity = fraction of **negatives misclassified as positives**.
- How is it constructed? (code)

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)

#plotting
idx_for_threshold_at_90 = (thresholds <=
threshold_for_90_precision).argmax()
tpr_90, fpr_90 = tpr[idx_for_threshold_at_90], fpr[idx_for_threshold_at_90]
plt.plot(fpr, tpr, linewidth=2, label="ROC curve") plt.plot([0, 1], [0, 1],
'k:', label="Random classifier's ROC curve")
plt.plot([fpr_90], [tpr_90], "ko", label="Threshold for 90% precision")
# [...] beautify the figure: add labels, grid, legend, arrow, and text
plt.show()
```

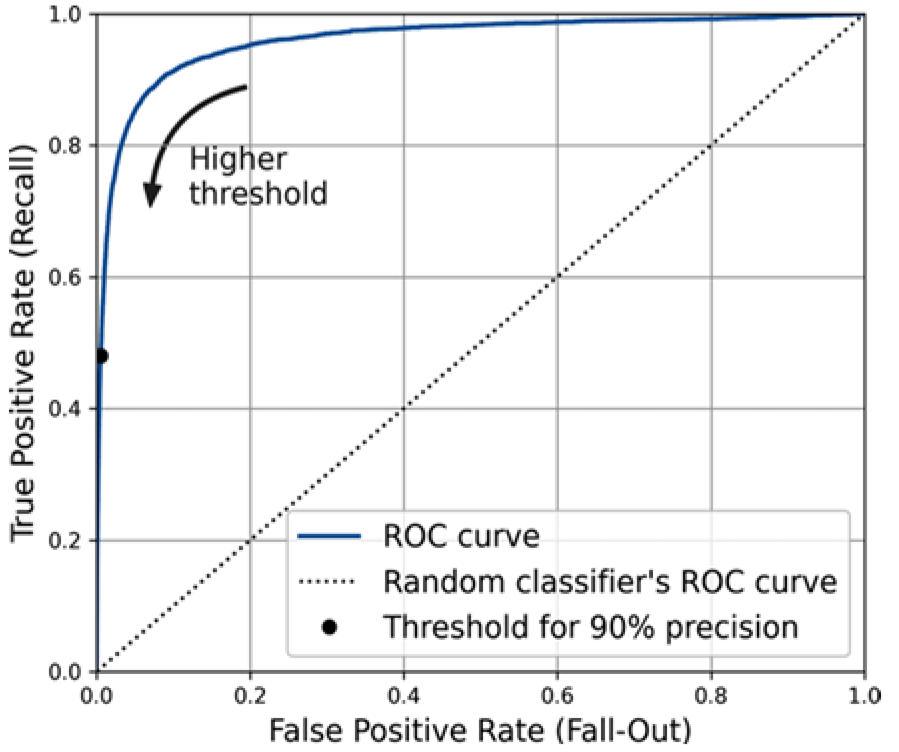


*Figure 3-7. A ROC curve plotting the false positive rate against the true positive rate for all possible thresholds; the black circle highlights the chosen ratio (at 90% precision and 48% recall)*

  -
  - A **great classifier** hugs the **top-left corner** (high recall, low FPR).
  - The **farther above the diagonal** it is, the better.
- AUC (Area Under the Curve):

```
from sklearn.metrics import roc_auc_score
```

```
roc_auc_score(y_train_5, y_scores)
```

- Value close to **1.0** = excellent classifier.
- Value **0.5** = random guessing.

- **Choosing Between ROC and PR Curves**
  - Use **PR curve** when:
    - The **positive class is rare**.
    - You care **more about false positives** than false negatives.
  - Use **ROC curve** otherwise.
  - ROC AUC can be misleading if positives are rare—it might look good even if the model isn't.
  - PR curve gives a more honest view in such cases.
- **Random Forest vs SGD Classifier (on PR Curve)**
  1. **RandomForestClassifier**:
     - Doesn’t use `decision_function()` like SGD.
     - Use `predict_proba()` to get probability scores instead.
     - Use the **positive class probability** as the score for PR/ROC analysis.

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=42)

y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
method="predict_proba")

y_probas_forest[:2]
# array([[0.11, 0.89],
#        [0.99, 0.01]])  First image: 89% likely to be positive, Second
image: 99% likely to be negative.
```

- **Warning About Probabilities**
  - These are **estimated probabilities**, not guaranteed.
  - For example, samples predicted with 50–60% confidence were actually positive **94% of the time** → model underestimated.
  - Some models can also be **overconfident**.
  - Use `sklearn.calibration` to fix this and make probabilities more accurate.
- **Evaluating with PR Curve**
  - Use `precision_recall_curve()` on `y_scores_forest = y_probas_forest[:, 1]`.
  - Plot both **Random Forest** and **SGD** PR curves for comparison.

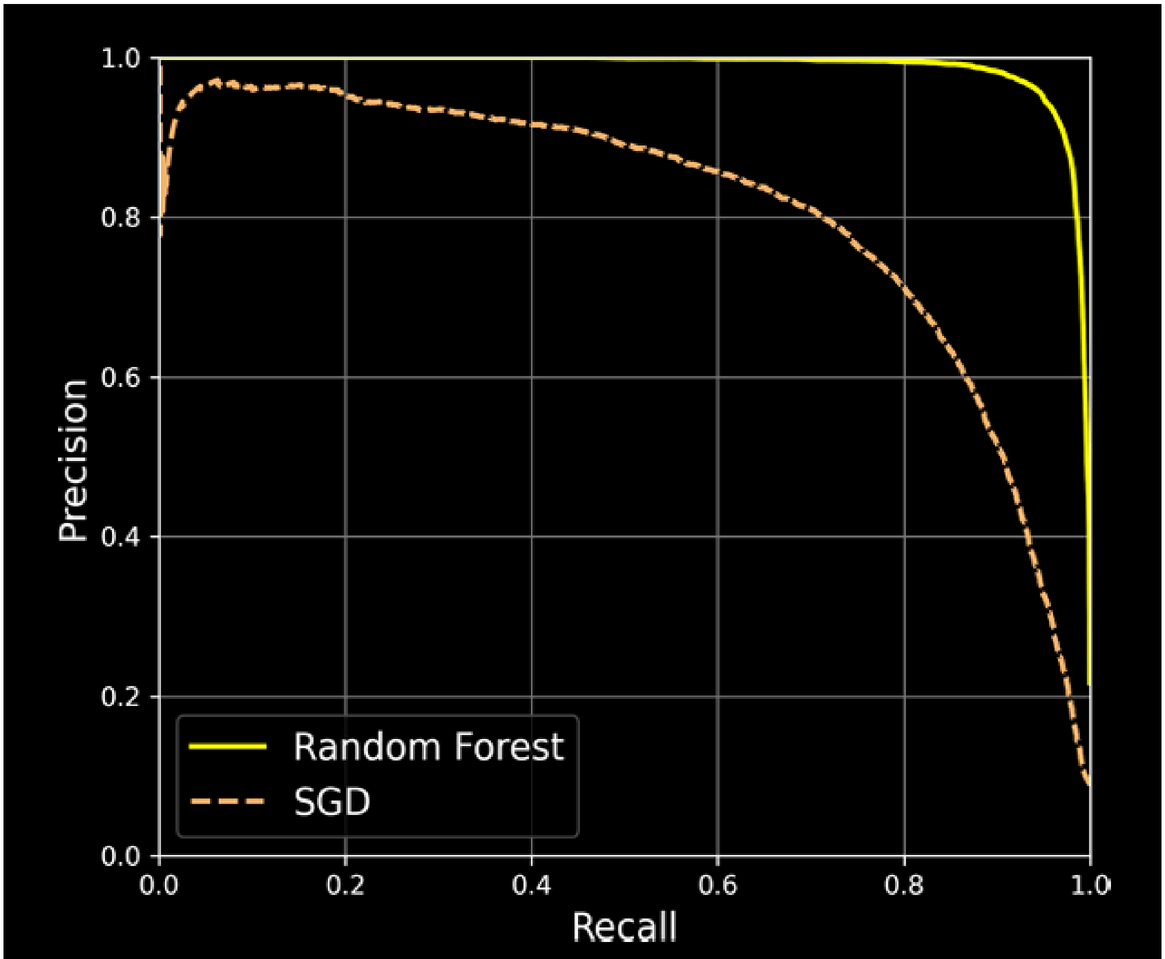
    - 

      *Figure 3-8. Comparing PR curves: the random forest classifier is superior to the SGD classifier because its PR curve is much closer to the top-right corner, and it has a greater AUC*

  - Random Forest performs much better:
    - Closer PR curve to the top-right corner.
    - **F1 score**: `0.924`
    - **ROC AUC score**: `0.998`
    - **Precision** ≈ `99.1%`, **Recall** ≈ `86.6%`

## Multiclass Classification

- What is Multiclass Classification?
  - Unlike binary classification (2 classes), **multiclass classification** handles **more than two** classes (e.g., digits 0–9).
- Built-in Support
  - Some classifiers (like `LogisticRegression`, `RandomForestClassifier`) support multiclass natively.
  - Others (like `SGDClassifier`, `SVC`) only support binary classification—but **Scikit-Learn** handles multiclass automatically using two strategies:
- Strategies for Multiclass with Binary Classifiers
  1. **One-vs-Rest (OvR / OvA)**

    - Train **one binary classifier per class**.
    - Classify by picking the classifier with the **highest decision score**.
  2. **One-vs-One (OvO)**
    - Train a binary classifier for **every pair of classes**.
    - For 10 classes (MNIST): **45 classifiers** → `10 × 9 / 2`.
    - Each classifier “votes”; the class with **most wins** is chosen.
    - The main advantage of OvO is that each classifier only needs to be trained on the part of the training set containing the two classes that it must distinguish.

- Example: `SVC` with OvO
  - `SVC` scales poorly with large datasets, so **OvO is used by default**.

    ```
    svm_clf = SVC(random_state=42)
    svm_clf.fit(X_train[:2000], y_train[:2000])
    ```

    - It trains **45 classifiers** internally for 10 classes.
- When you predict, it:
  - Runs the sample through all 45 classifiers.
  - Collects **10 class scores** using `decision_function()` — one score per class (based on votes).
  - Picks the class with the **highest score**.

  ```
  some_digit_scores = svm_clf.decision_function([some_digit])
  class_id = some_digit_scores.argmax()
  svm_clf.classes_[class_id]  # gives predicted class
  ```

  - Even though OvO uses many classifiers, **you still get one score per class**.
- You can **manually choose** One-vs-One (OvO) or One-vs-Rest (OvR) using:
  - `OneVsOneClassifier`
  - `OneVsRestClassifier`

  ```
  from sklearn.multiclass import OneVsRestClassifier
  ovr_clf = OneVsRestClassifier(SVC(random_state=42))
  ovr_clf.fit(X_train[:2000], y_train[:2000]) #This trains **10 classifiers** (one for each class).
  ovr_clf.predict([some_digit])  # Output: '5'
  ```

- Using SGDClassifier for Multiclass

  ```
  sgd_clf = SGDClassifier(random_state=42)
  sgd_clf.fit(X_train, y_train)
  sgd_clf.predict([some_digit])  # Output: '3' (could be wrong)
  ```

```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
# → ~86% accuracy (better than random guessing, which is 10%)
```

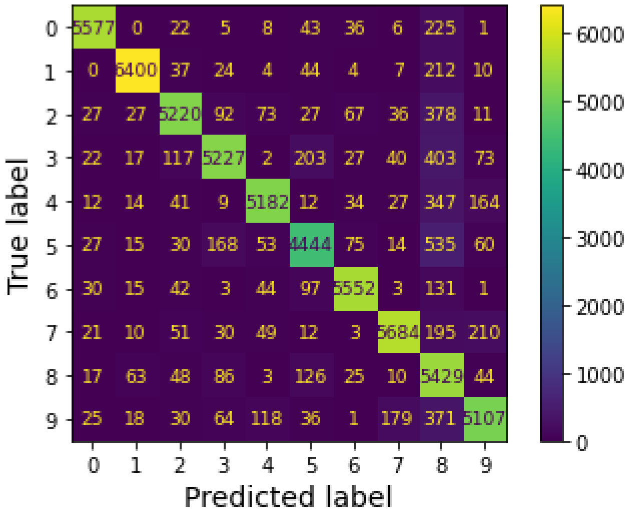
- `SGDClassifier` uses **OvR** by default.
- `decision_function([some_digit])` returns **10 confidence scores**, one per class.
- Boosting Accuracy with Scaling:

  ```
  from sklearn.preprocessing import StandardScaler
  scaler = StandardScaler()
  X_train_scaled = scaler.fit_transform(X_train.astype("float64"))
  cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3,
  scoring="accuracy")
  # → ~89–90% accuracy
  ```

# Error Analysis

- Once you've trained a promising model, analyze **what types of mistakes it makes** to improve it.
- Confusion Matrix
  - Use `cross_val_predict()` to get predictions.
  - Plot the confusion matrix using:

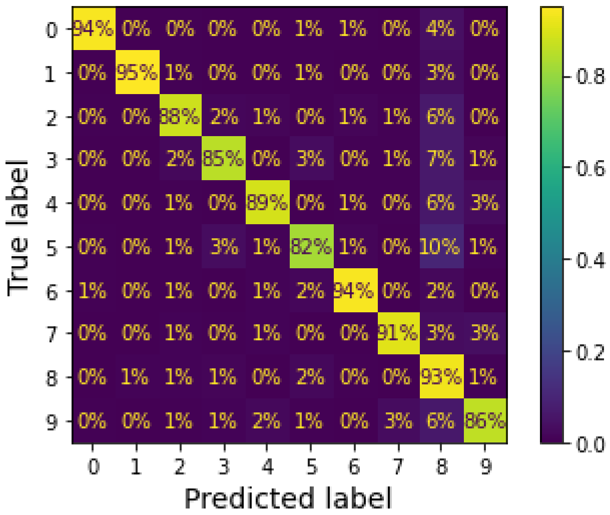
```
from sklearn.metrics import ConfusionMatrixDisplay
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
plt.show()
```



|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 5577 | 0 | 22 | 5 | 8 | 43 | 36 | 6 | 225 | 1 |
| 1 | 0 | 6400 | 37 | 24 | 4 | 44 | 4 | 7 | 212 | 10 |
| 2 | 27 | 27 | 5220 | 92 | 73 | 27 | 67 | 36 | 378 | 11 |
| 3 | 22 | 17 | 117 | 5227 | 2 | 203 | 27 | 40 | 403 | 73 |
| 4 | 12 | 14 | 41 | 9 | 5182 | 12 | 34 | 27 | 347 | 164 |
| 5 | 27 | 15 | 30 | 168 | 53 | 4444 | 75 | 14 | 535 | 60 |
| 6 | 30 | 15 | 42 | 3 | 44 | 97 | 5552 | 3 | 131 | 1 |
| 7 | 21 | 10 | 51 | 30 | 49 | 12 | 3 | 5684 | 195 | 210 |
| 8 | 17 | 63 | 48 | 86 | 3 | 126 | 25 | 10 | 5429 | 44 |
| 9 | 25 | 18 | 30 | 64 | 118 | 36 | 1 | 179 | 371 | 5107 |

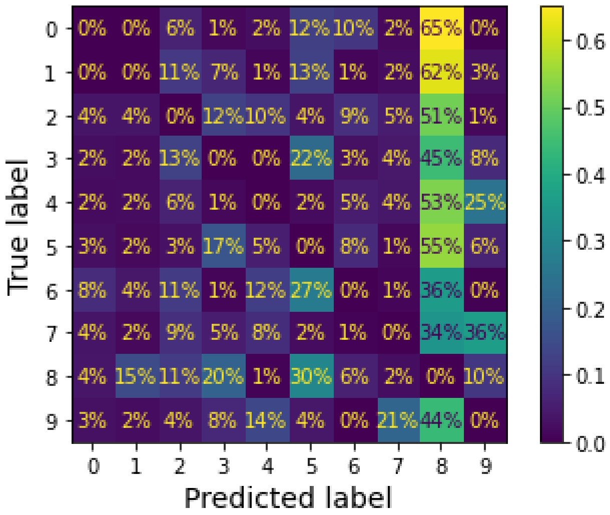
- Normalized Confusion Matrix

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



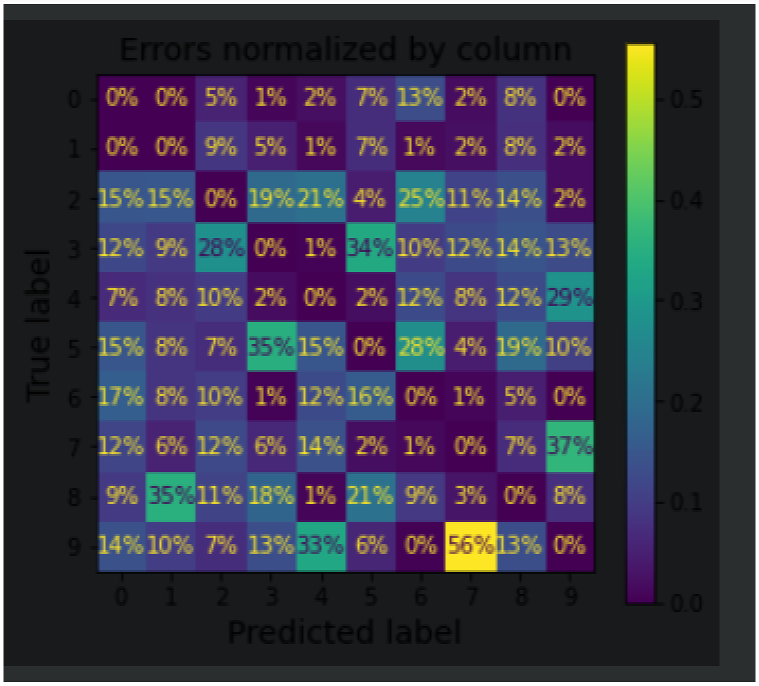
- **Example finding: Only 82% of digit 5s are correctly predicted.**
  - 10% of 5s were misclassified as 8s.
  - Only 2% of 8s were misclassified as 5s → **asymmetrical errors**.
- Highlight Only the Errors(To make misclassifications more visible)

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



- **Column-Normalized Confusion Matrix**
  - Use `normalize="pred"` to normalize **by column**:
    - Shows **what predicted class the errors are going to**.
    - Example: 56% of **predicted 7s** are actually **9s**.

```
ConfusionMatrixDisplay.from_predictions(
                              y_train, y_train_pred,
                              ax=axs[1],
                              sample_weight=sample_weight,
                              normalize="pred", # the change
                              values_format=".0%"
                              )
```

- 

- Error Analysis:
  - Use a **confusion matrix** to find common mistakes (e.g., many digits misclassified as 8).
  - Improve accuracy by:
    - Adding **more training data** for confusing digits.
    - **Engineering features** (e.g., count loops in digits).
    - **Preprocessing images** to highlight important patterns.
  - **Visualize errors** (like 3s vs 5s) to see where and why mistakes happen.
  - **SGDClassifier is linear**, so it's sensitive to small changes like shifts or rotations.
  - **Best fix**: Use **data augmentation**—add rotated/shifted versions of training images to improve model robustness.

# Multilabel Classification

- In **multilabel classification**, each input can be assigned **multiple binary labels** (not just one class).

```
y_train_large = (y_train >= '7')       # True if digit is 7, 8, or 9
y_train_odd = (y_train.astype('int8') % 2 == 1)  # True if digit is odd
y_multilabel = np.c_[y_train_large, y_train_odd]
```

  - This creates **two labels per digit**:
  - Then trains a `KNeighborsClassifier` with these multilabels.

  - Prediction Example

    ```
    knn_clf.predict([some_digit]) → [[False, True]]
    ```

    - → The digit is not large but it is odd (like 5).

- Evaluation
  - Use **F1 score** with `average="macro"` to give equal weight to each label:

  ```
  f1_score(y_multilabel, y_train_knn_pred, average="macro")
  ```

  - → `average="weighted"` gives more weight to labels with more examples.
- **When Classifier Doesn’t Support Multilabels**
  - Train one model per label (but they won’t communicate).
  - A better option: use `ClassifierChain`.
- ClassifierChain
  - Organizes models **in a chain** so later models use predictions from earlier ones.
  - Use `cv` to get clean predictions during training.

  ```
  chain_clf = ClassifierChain(SVC(), cv=3, random_state=42)
  chain_clf.fit(X_train[:2000], y_multilabel[:2000])
  chain_clf.predict([some_digit])  # → [[0., 1.]]
  ```

# Multioutput Classification

- **What is Multioutput Classification?**
  - A **generalization of multilabel classification**.
  - Each label can have **more than two values** (i.e., it's multiclass per output).
  - Example: A system that removes noise from images.
- **Noise Removal Example**
  - Input: **Noisy MNIST digit** image (added random noise to each pixel).
  - Output: **Clean digit** image (original pixel intensities).
  - Each pixel is a label, and its value ranges from 0–255 → **multioutput multiclass**.
- code

  ```
  noise = np.random.randint(0, 100, (len(X_train), 784))
  X_train_mod = X_train + noise       # Noisy inputs
  y_train_mod = X_train               # Clean targets
  ```

  - Trained a **KNeighborsClassifier** to predict clean images from noisy ones:

  ```
  knn_clf.fit(X_train_mod, y_train_mod)
  clean_digit = knn_clf.predict([X_test_mod[0]])
  ```

---

# Resources :

-

---

# Related notes :

-

---

# References :

- **Internal :**
    -
    -
    -
- **External :**
    - notebook of the chapter
        - hegab videos
    - the book
    -