CHAPTER 3 CLASSIFICATION

MNIST:

The MNIST dataset consists of 70,000 small, labeled images of handwritten digits, widely recognized as the "Hello World" of Machine Learning due to its frequent use in testing classification algorithms. Scikit-Learn offers helper functions to download popular datasets, including MNIST.

Scikit-Learn datasets usually have a dictionary-like structure with keys for:

* **DESCR**: A description of the dataset.
* **data**: An array where each row is an instance, and each column is a feature.
* **target**: An array containing the labels.

For MNIST:

* **X, y = mnist["data"], mnist["target"]**
  + X.shape: (70000, 784) (70,000 images, each with 784 pixels for 28×28 images).
  + y.shape: (70000,) (labels for each image).
  + Each pixel’s intensity ranges from 0 (white) to 255 (black).

**Viewing an Image:**

To view a digit, take one row from X, reshape it into a 28×28 array, and display it using Matplotlib.

**Preparing the Data:**

Always split your data before inspecting it:

* Training set: First 60,000 images.
* Test set: Last 10,000 images.

CODE:

X\_train, X\_test, y\_train, y\_test = X[:60000], X[60000:], y[:60000], y[60000:]

Shuffling the training set is essential:

* Ensures cross-validation folds are similar.
* Prevents issues with learning algorithms sensitive to instance order.

TRAINING A BINARY CLASSIFIER:

To simplify the problem, we create a binary classifier to detect the digit **5** (5 or not-5):

* **Create target vectors for 5-detection:**

CODE:

y\_train\_5 = (y\_train == 5) # True for 5s, False for others.

y\_test\_5 = (y\_test == 5)

* **Choose a classifier:**  
  The **Stochastic Gradient Descent (SGD)** classifier is efficient for large datasets and supports online learning.
* **Train the SGDClassifier:**

CODE:

from sklearn.linear\_model import SGDClassifier

sgd\_clf = SGDClassifier(random\_state=42)

sgd\_clf.fit(X\_train, y\_train\_5)

* **Randomness in SGD:**  
  Training is stochastic (random), so set random\_state for reproducible results.
* **Make a prediction:**

CODE:

sgd\_clf.predict([some\_digit])

Returns True if the digit is **5**. For example, if the model predicts True, it thinks the image is a **5**.

PERFORMANCE MEASURE:

MEASURING ACCURACY USING CROSS VALIDATION:

**Measuring accuracy using cross-validation** means testing your model's performance on different parts of the dataset to ensure it works well overall, not just on the training data.

Here’s how it works:

1. **Split the training data:** The data is divided into smaller parts (folds). For example, in 3-fold cross-validation, the data is split into 3 parts.
2. **Train and test repeatedly:**
   * Use 2 parts for training and 1 part for testing.
   * Repeat this process, rotating which part is used for testing.
3. **Average the results:** The accuracy scores from all rounds are averaged to get a final score.

In Scikit-Learn, this can be done with the cross\_val\_score function.This gives the accuracy for each fold. Cross-validation ensures the model’s accuracy is reliable and not just specific to one data split.

MNIST, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is not a 5, you will be right about 90% of the time. Beats Nostradamus. This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with skewed datasets (i.e. when some classes are much more frequent than others)

CONFUSION MATRIX:

A **confusion matrix** is a table that helps evaluate how well your classification model is performing by comparing actual labels with predicted labels.

It looks like this for a binary classifier:

|  | **Predicted: Yes** | **Predicted: No** |
| --- | --- | --- |
| **Actual: Yes** | True Positive (TP) | False Negative (FN) |
| **Actual: No** | False Positive (FP) | True Negative (TN) |

**What the terms mean:**

* **True Positive (TP):** Model correctly predicted "Yes" (e.g., a 5 is detected as 5).
* **True Negative (TN):** Model correctly predicted "No" (e.g., not-5 detected as not-5).
* **False Positive (FP):** Model incorrectly predicted "Yes" (e.g., not-5 predicted as 5).
* **False Negative (FN):** Model incorrectly predicted "No" (e.g., 5 predicted as not-5).

**Example:**

If you have 100 test samples, a confusion matrix might look like this:

|  | **Predicted: Yes** | **Predicted: No** |
| --- | --- | --- |
| **Actual: Yes** | 40 (TP) | 10 (FN) |
| **Actual: No** | 5 (FP) | 45 (TN) |

This means:

* Out of 50 actual "Yes" cases, 40 were correctly predicted, and 10 were missed.
* Out of 50 actual "No" cases, 45 were correctly predicted, and 5 were wrongly classified.

**Why it’s useful:**

The confusion matrix gives more detailed insights than accuracy alone, showing where the model is making mistakes.

PRECISION AND RECALL:

**Precision**

Precision tells us:  
**Out of all the times the model said "Yes" (e.g., detected 5), how many were correct?**

Precision=True Positives (Correct Yes)True Positives + False Positives (Wrong Yes)\text{Precision} = \frac{\text{True Positives (Correct Yes)}}{\text{True Positives + False Positives (Wrong Yes)}}Precision=True Positives + False Positives (Wrong Yes)True Positives (Correct Yes)​

**Example:**

* The model predicted "Yes" 50 times.
* It was right 40 times (True Positives) and wrong 10 times (False Positives).
* Precision = 4040+10=0.8\frac{40}{40+10} = 0.840+1040​=0.8 (80%).

**Recall**

Recall tells us:  
**Out of all the actual "Yes" cases (e.g., actual 5s), how many did the model correctly predict?**

Recall=True Positives (Correct Yes)True Positives + False Negatives (Missed Yes)\text{Recall} = \frac{\text{True Positives (Correct Yes)}}{\text{True Positives + False Negatives (Missed Yes)}}Recall=True Positives + False Negatives (Missed Yes)True Positives (Correct Yes)​

**Example:**

* There are 50 actual "Yes" cases.
* The model correctly predicted 40 (True Positives) but missed 10 (False Negatives).
* Recall = 4040+10=0.8\frac{40}{40+10} = 0.840+1040​=0.8 (80%).

PRECISION/ RECALL TRADEOFF:

The **Precision/Recall tradeoff** refers to the balance between precision and recall in a classification model. Improving one often reduces the other, so you need to decide which is more important for your specific problem.

**Why There’s a Tradeoff**

* **High Precision:** The model predicts "Yes" only when it’s very confident. This reduces false positives but may miss some actual positives, lowering recall.
* **High Recall:** The model tries to catch all actual "Yes" cases, even if it means predicting "Yes" more often. This increases true positives but may lead to more false positives, lowering precision.

**Example**

Imagine detecting spam emails:

* **High Precision:** The model only marks an email as spam if it’s almost certain.
  + Few false alarms (important emails don’t go to spam).
  + Some spam emails may slip through (low recall).
* **High Recall:** The model aggressively marks emails as spam to catch all of them.
  + No spam slips through (high recall).
  + Many important emails may also be marked as spam (low precision).

**How to Adjust the Tradeoff**

Most classifiers allow you to adjust the **decision threshold**, which determines when the model predicts "Yes":

* **Higher threshold:** Increases precision, lowers recall.
* **Lower threshold:** Increases recall, lowers precision.

**When to Prioritize Precision or Recall**

* **Precision is key:** When false positives are costly (e.g., diagnosing a serious disease incorrectly).
* **Recall is key:** When missing actual positives is costly (e.g., identifying fraudulent transactions).

Adjusting the threshold allows you to find the right balance for your problem!

ROC CURVE:

The **ROC curve** (Receiver Operating Characteristic curve) is a graph that shows how well your model distinguishes between classes by plotting two things:

1. **True Positive Rate (TPR) = Recall**: How many actual positives (e.g., 5s) the model correctly predicts.
2. **False Positive Rate (FPR)**: How many negatives (e.g., not-5s) the model incorrectly predicts as positives.

**How it Works**

* The model predicts probabilities (not just "Yes" or "No") for each instance.
* You can set a **threshold** to decide when the model predicts "Yes":
  + **Low threshold**: More "Yes" predictions → Higher recall but more false positives.
  + **High threshold**: Fewer "Yes" predictions → Lower recall but fewer false positives.
* The ROC curve shows how TPR and FPR change as you adjust the threshold.

**Interpreting the ROC Curve**

* **X-axis**: False Positive Rate (FPR).
* **Y-axis**: True Positive Rate (TPR).
* A perfect model has a curve that goes straight up and then across the top (FPR = 0, TPR = 1).
* A random guess gives a diagonal line (no skill).

**ROC AUC (Area Under the Curve)**

The **AUC** (Area Under the Curve) measures how good the model is:

* **1.0**: Perfect model.
* **0.5**: Random guessing.
* Closer to 1.0 is better.

**In Simple Terms**

The ROC curve helps you visualize how well your model balances recall (catching positives) and false alarms (avoiding false positives) at different thresholds. The AUC score summarizes its performance—higher is better!

MULTICLASS CLASSIFICATION:

Multiclass classification is about distinguishing between **more than two classes** (e.g., digits 0–9). Here's how it works conceptually:

**Strategies for Multiclass Classification**

1. **One-vs-All (OvA):**
   * Train one binary classifier for each class.
   * Each classifier predicts if the input belongs to its class or not.
   * When making predictions, the class with the highest score is selected.
2. **One-vs-One (OvO):**
   * Train a binary classifier for every pair of classes (e.g., 0 vs. 1, 0 vs. 2, etc.).
   * Total classifiers = N×(N−1)/2\text{N} \times (\text{N}-1) / 2N×(N−1)/2. For 10 classes, that’s 45 classifiers.
   * When predicting, each classifier votes, and the class with the most votes wins.

**How Scikit-Learn Handles Multiclass Tasks**

* Some models (e.g., Random Forest) can natively handle multiclass classification.
* Other models (e.g., SGDClassifier) are binary classifiers but Scikit-Learn automatically applies OvA or OvO when needed.

ERROR ANALYSIS:

Error analysis helps you understand where your model is making mistakes, so you can improve it. Here’s how it works:

1. **Confusion Matrix**:
   * This matrix shows how well the model is performing, comparing predicted vs. actual results.
   * It helps to see how many predictions were correct and where the model went wrong (e.g., classifying a 3 as a 5).
2. **Visualizing Errors**:
   * By dividing the confusion matrix by the number of examples in each class, you can compare error rates (not just raw counts).
   * The errors that stand out are those where the model is confused between specific classes, like misclassifying 8s as 9s.
3. **Analyzing Misclassifications**:
   * To improve the model, look at examples where the model made mistakes.
   * You can focus on particular misclassifications, like 3s being confused with 5s, and try to find patterns to address (e.g., images that are poorly written or shifted).
4. **Improving the Model**:
   * Based on the error analysis, you might try things like:
     + Gathering more data for the problematic classes.
     + Preprocessing images (e.g., centering or rotating them) to make them clearer for the model to classify.

In short, error analysis helps you identify where your model struggles, so you can take targeted steps to improve it.

MULTILABEL CLASSIFICATION:

In multilabel classification, each instance can belong to more than one class at the same time. Unlike regular classification (where each instance belongs to just one class), multilabel classification assigns multiple labels to each instance.

**Example**:  
Imagine you want a classifier to recognize multiple people in a photo. If the classifier recognizes Alice and Charlie, it would output [1, 0, 1], where:

* 1 means "Alice is present"
* 0 means "Bob is not present"
* 1 means "Charlie is present"

**How it works:**

1. **Creating Multiple Labels**:  
   For each instance (like a digit image), you can create multiple labels. For example:
   * Is the digit large (7, 8, or 9)?
   * Is the digit odd?
2. **Training**:  
   The classifier is trained on the multiple labels (like whether the digit is large and odd).
3. **Prediction**:  
   When you ask the model to make a prediction, it will output multiple labels. For example, it might predict [False, True], meaning the digit is not large but is odd.
4. **Evaluation**:  
   You can evaluate multilabel classification using metrics like F1 score, either averaging over all labels or weighting them based on their importance.

In summary, multilabel classification allows you to predict multiple attributes for each instance, making it useful for cases where one instance can belong to several categories simultaneously.

MULTIOUTPUT CLASSIFICATION:

Multioutput classification is a type of machine learning task where the model predicts multiple labels for each instance, and each label can have more than two possible values (i.e., it can be multiclass).

**Example**:  
Imagine you want to build a system that removes noise from images. The model takes a noisy image as input and tries to output the clean version of the image.

* The input is a noisy image with pixel intensities (values between 0 and 255).
* The output is the clean image, where each pixel also has a value between 0 and 255.

**How it works:**

* **Training**: You add noise to the images (input) and use the original clean images as the target.
* **Prediction**: The model tries to clean the noisy input and predicts the clean image by adjusting each pixel's intensity.
* **Multioutput**: In this case, the model predicts the pixel values for all the pixels in the image, which is a multioutput task because each pixel has its own value.

In summary, **multioutput classification** is when the model predicts multiple values (labels) for each instance, and those labels can have more than two possible values.