Week4: Metrics and Naive Bayes

Table of contents

1. Why Evaluation Deserves Its Own Discussion

So far in this module, you have trained some models and understood their behavior.

In doing so, we often report **accuracy** — not because it is the best metric, but because it is simple and easy to interpret at first glance.

However, this raises an important question:

> *How do we decide whether a model is actually "good"?*

Answering this requires more than training a model.
In addition to other factors, it requires **choosing the right way to evaluate it**.

1.1 Models Do Not Speak for Themselves

A trained model does not tell us:

- whether it is reliable,
- whether it generalises,
- or whether it is useful in practice.

All of these judgements come from **evaluation metrics**.

Crucially:

> **Different metrics answer different questions.**

Using the wrong metric can:

- make a weak model look strong,
- hide important failure modes,
- or encourage the wrong modelling decisions.

---

2. Classification Evaluation — Confusion Matrix & Core Metrics

In a **classification task**, a model predicts a *class label* (e.g. `yes` / `no`, `positive` / `negative`).

To evaluate such predictions meaningfully, we must first understand **how predictions can be right or wrong**.

2.1 The Confusion Matrix: The Foundation

At the heart of classification evaluation lies the **confusion matrix**.

It compares: - what the model *predicted*, - against what actually *happened*.

For a binary classification problem, there are four possible outcomes:

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

Every classification metric is derived from these four quantities.

2.2 Interpreting the Outcomes

Understanding these outcomes is more important than memorising formulas.

- **True Positive (TP)**
  The model predicted *yes*, and the outcome was *yes*.
- **False Positive (FP)**
  The model predicted *yes*, but the outcome was *no*.
  (A false alarm.)
- **False Negative (FN)**
  The model predicted *no*, but the outcome was *yes*.
  (A missed case.)
- **True Negative (TN)**
  The model predicted *no*, and the outcome was *no*.

Different applications care about these errors **very differently**.

2.3 Accuracy: The Simplest Metric

Accuracy measures how often the model is correct overall. Conceptually:

> *Out of all predictions, how many were correct?*

Accuracy is intuitive — but also dangerous.

Why Accuracy Can Be Misleading

Accuracy treats all errors equally. This becomes problematic when:

- the dataset is **imbalanced**,
- one class dominates the data,
- some mistakes are more costly than others.

A model that predicts the majority class every time may achieve high accuracy while being practically useless.

2.4 Precision, Recall, and F1 Score: A Loan Approval Example

Consider a loan default prediction model used by a bank.

- **Positive ("yes")** = the applicant will **default**
- **Negative ("no")** = the applicant will **repay**

This framing lets us interpret each metric in business terms.

Precision: Are Default Warnings Trustworthy?

Precision answers:

> *When the model predicts that an applicant will default, how often is it correct?*

Suppose the model flags **100 applicants as likely to default**:

- 80 actually default
- 20 would have repaid

Precision = 80%

High precision means:

- few **false positives**,
- fewer safe customers wrongly rejected,
- risk flags can be trusted.

Low precision means the bank loses good customers by rejecting loans unnecessarily.

Recall: How Many Risky Loans Did We Catch?

Recall answers:

> *Of all applicants who will actually default, how many did the model identify?*

Suppose **200 applicants would eventually default**:

- the model flags 80 of them,
- 120 risky applicants are missed.

Recall = 40%

High recall means:

- few **false negatives**,
- fewer bad loans slip through,
- lower financial losses.

Low recall means the bank approves many loans that later default.

The Precision–Recall Trade-off

Precision and recall often conflict. Which is preferable depends on context:

- Risk-averse institutions prioritize **recall**
- Growth-focused institutions prioritize **precision**

This choice reflects business values, not algorithmic correctness.

F1 Score: A Balanced Summary

The **F1 score** combines precision and recall into a single metric. It is useful when:

- defaults are rare (class imbalance),
- both rejecting good customers and approving bad ones are costly,
- accuracy alone is misleading.

However, using F1 implicitly assumes that **false positives and false negatives matter equally**—a judgement that may or may not align with real-world priorities.

---

3. Naive Bayes Demo — Classification Evaluation in Practice

We now apply the ideas from the previous section to a **concrete example** using Naive Bayes. Our goal here is **not** to build a high-performing model. It is to see how **evaluation metrics behave** in practice.

3.1 Dataset and Feature Choice

For this demonstration, we use the same **UCI Bank Marketing dataset**.

To keep the demo interpretable, we restrict ourselves to a **small set of categorical features**:

- `job`
- `education`
- `housing`
- `loan`

- `contact`

This is a **modelling choice**, not a requirement.

3.2 Loading Data

We reuse the same train/validation split introduced earlier.

R: Loading Data and Creating Splits

```r
library(tidyverse)
library(dplyr)

bank <- read.csv("data/raw/bank-additional.csv", sep = ";", stringsAsFactors = FALSE)

model_data <- bank %>%
  select(-duration) %>%
  mutate(y = as.factor(y))

set.seed(42)

n <- nrow(model_data)
idx <- sample(n)

train_idx <- idx[1:floor(0.6 * n)]
val_idx   <- idx[(floor(0.6 * n) + 1):floor(0.8 * n)]
test_idx  <- idx[(floor(0.8 * n) + 1):n]

train_data <- model_data[train_idx, ]
val_data   <- model_data[val_idx, ]
test_data  <- model_data[test_idx, ]
```

Python: Loading Data and Creating Splits

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline

bank = pd.read_csv("data/raw/bank-additional.csv", sep=";")
```

```python
model_data = bank.drop(columns=["duration"])
model_data["y"] = model_data["y"].astype("category")

# Split features / target
X = model_data.drop(columns=["y"])
y = model_data["y"]

# Identify column types
cat_cols = X.select_dtypes(include=["object", "category"]).columns
num_cols = X.select_dtypes(exclude=["object", "category"]).columns

# Preprocessing: encode categoricals, pass through numerics
preprocess = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", OneHotEncoder(handle_unknown="ignore"), cat_cols),
    ]
)

X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.4, random_state=42, stratify=y
)

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp
)
```

3.3 Preparing the Data

R: Select Features and Split Data

```r
library(e1071)

features <- c("job", "education", "housing", "loan", "contact")

nb_train <- train_data[, c(features, "y")]
nb_val   <- val_data[, c(features, "y")]
```

What this code is doing

Selects a small subset of categorical predictors.
Uses the existing training and validation splits.
Keeps the target variable unchanged.
Why this matters:
Fewer features make probability estimates easier to interpret.
This avoids accidental leakage or overfitting.

Python: Select Features and Split Data

```python
from sklearn.naive_bayes import CategoricalNB
from sklearn.preprocessing import OrdinalEncoder

features = ["job", "education", "housing", "loan", "contact"]

X_train_nb = X_train[features]
X_val_nb   = X_val[features]
```

What this code is doing

Restricts the feature set to categorical variables.
Prepares separate training and validation inputs.

3.4 Training a Naive Bayes Classifier

R: Training the Model

```r
nb_model <- naiveBayes(
  y ~ .,
  data = nb_train
)
```

What this code is doing

Fits a Naive Bayes classifier.
Estimates class priors and conditional probabilities.
Assumes conditional independence between features.

Python: Training the Model

```
nb_model = Pipeline(
    steps=[
        ("enc", OrdinalEncoder()),
        ("nb", CategoricalNB())
    ]
)

nb_model.fit(X_train_nb, y_train)
```

```
Pipeline(steps=[('enc', OrdinalEncoder()), ('nb', CategoricalNB())])
```

> **What this code is doing**
>
> Encodes categorical variables numerically because scikit-learn requires numeric inputs.
> Trains a categorical Naive Bayes classifier.
> Keeps preprocessing and modelling in one pipeline.

3.5 From Probabilities to Predictions

Naive Bayes produces probabilities, not hard decisions. To obtain class predictions, we apply a threshold.

R: Predicted Probabilities and Classes

```
nb_probs <- predict(nb_model, nb_val, type = "raw")
nb_preds <- predict(nb_model, nb_val, type = "class")
```

Python: Predicted Probabilities and Classes

```
nb_probs = nb_model.predict_proba(X_val_nb)
nb_preds = nb_model.predict(X_val_nb)
```

> **Why this step matters**
>
> Probabilities express uncertainty.
> Class labels depend on a threshold.

Changing the threshold changes precision and recall.
This directly links model output to evaluation metrics.

---

4. Computing Classification Metrics (R & Python)

We now evaluate the Naive Bayes model using the metrics introduced earlier. The key idea in this section is simple:

> **The same predictions can look "good" or "bad" depending on the metric we use.**

4.1 Confusion Matrix

We begin with the confusion matrix, which shows **how predictions break down**.

R: Confusion Matrix

```r
table(
  Predicted = nb_preds,
  Actual    = nb_val$y
)
```

```
         Actual
Predicted  no yes
      no  736  88
      yes   0   0
```

i What this shows

- How many predictions fall into each TP, FP, FN, TN category.
- The raw material from which all classification metrics are computed.

Why this matters:

- Metrics summarise behaviour.
- The confusion matrix *reveals* behaviour.

Python: Confusion Matrix

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_val, nb_preds)
```

```
array([[733,   1],
       [ 90,   0]])
```

i Why start with the confusion matrix

Looking only at a single metric hides important failure modes. The confusion matrix forces us to ask:

- *What kinds of mistakes is the model making?*

4.2 Accuracy

Accuracy measures how often the model is correct overall.

R: Accuracy

```
mean(nb_preds == nb_val$y)
```

```
[1] 0.8932039
```

Python: Accuracy

```
from sklearn.metrics import accuracy_score

accuracy_score(y_val, nb_preds)
```

```
0.8895631067961165
```

> How to interpret accuracy here
>
> - Accuracy treats all mistakes equally.
> - With class imbalance, this can be misleading.
>
> Use accuracy as a **baseline**, not a final judgement.

4.3 Precision and Recall

Accuracy alone does not tell us **what kind of mistakes** the model is making.

Precision and recall help us look deeper.

R: Precision and Recall

```
#install.packages("recipes")
library(caret)

conf <- confusionMatrix(nb_preds, nb_val$y, positive = "yes")

conf$byClass["Precision"]
```

```
Precision
       NA
```

```r
conf$byClass["Recall"]
```

```
Recall
     0
```

Python: Precision and Recall

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

precision_score(y_val, nb_preds, pos_label="yes")
```

```
0.0
```

```python
recall_score(y_val, nb_preds, pos_label="yes")
```

```
0.0
```

> Interpreting precision vs recall
>
> - Precision focuses on **false positives**.
> - Recall focuses on **false negatives**.
>
> Which one matters more depends on the **context**, not the model.

4.4 F1 Score

The **F1 score** combines precision and recall into a single number.

It is useful when:

- classes are imbalanced,
- both types of errors matter.

R: F1 Score

```r
conf$byClass["F1"]
```

```
F1
NA
```

Python: F1 Score

```python
from sklearn.metrics import f1_score

f1_score(y_val, nb_preds, pos_label="yes")
```

```
0.0
```

4.5 A Note on Thresholds (Conceptual Only)

Remember that Naive Bayes produces **probabilities**. Changing the classification threshold would:

- change the confusion matrix,
- alter precision and recall,
- leave the underlying probabilities unchanged.

We do **not** tune thresholds here — but it is important to know that metrics depend on this choice.

---

5. Regression Metrics — What Changes When the Target Is Numeric

So far, we have focused on **classification**, where predictions are discrete labels (e.g. `yes` / `no`). In **regression**, the situation is different:

> There is no notion of "correct" or "incorrect" —
> only *how far off* a prediction is.

This requires a different way of evaluating models.

5.1 Why Accuracy Does Not Apply

In regression tasks: - predictions are numeric, - errors are continuous, - mistakes vary in magnitude.

Asking whether a prediction is "right" no longer makes sense.

Instead, we ask:

> *How large are the prediction errors?*

5.2 Absolute Error vs Squared Error

The core idea in regression evaluation is **measuring error size**.

We briefly introduce three commonly used metrics.

Mean Absolute Error (MAE)

MAE measures the average absolute difference between predictions and actual values.

Conceptually: - treats all errors equally, - easy to interpret in original units, - robust to occasional large errors.

MAE answers: > *On average, how far off are we?*

Mean Squared Error (MSE)

MSE measures the average of squared errors.

Conceptually: - penalises large errors more heavily, - sensitive to outliers, - useful when large mistakes are particularly costly.

MSE answers: > *How much do large errors matter to us?*

Root Mean Squared Error (RMSE)

RMSE is the square root of MSE.

Why it is used: - brings error back to the original scale, - still penalises large errors more than MAE.

RMSE balances: - interpretability, - sensitivity to large deviations.

5.3 Choosing Between MAE and RMSE

There is no universally correct choice.

A useful rule of thumb:

- Use **MAE** when:
  - interpretability matters,
  - all errors are equally costly.
- Use **RMSE** when:
  - large errors are particularly undesirable,
  - outliers should be penalised more strongly.

Metric choice reflects **modelling priorities**, not mathematical correctness.

5.4 Connecting Back to Your Project Dataset

In the **Online Retail dataset**, regression-style targets might include: - invoice value, - total customer spend, - basket size.

If you choose such a target: - MAE tells you average monetary error, - RMSE highlights whether large mistakes dominate.

As with classification, the metric must match the **question you are asking**.
