**Chapter 1 - Data Collection**

In our **Manning-style data extraction step**

1. **Identified the data source**

* + Located the **Student Performance** dataset on the UCI Machine Learning Repository.
  + Found that UCI stores it as a **ZIP file** containing two CSVs:
    - student-mat.csv (Math course)
    - student-por.csv (Portuguese course)

1. **Chose programmatic access**
   * Instead of manually downloading the ZIP, we used **httpx** for asynchronous HTTP requests, following Manning’s reproducibility mindset.
2. **Downloaded the raw file**
   * Sent a GET request to the ZIP URL.
   * Verified the response with response.raise\_for\_status() to catch errors (e.g., 404).
3. **Processed the data in-memory**
   * Used **BytesIO** to treat the downloaded bytes like a file.
   * Used **ZipFile** to read the CSVs directly from memory without saving to disk first.
4. **Loaded into Pandas DataFrames**
   * Read student-mat.csv and student-por.csv into df\_math and df\_port.
   * Kept the original delimiter (;) from UCI’s format.
5. **Confirmed ingestion success**
   * Printed dataset shapes:
     + Math → 395 rows × 33 columns
     + Portuguese → 649 rows × 33 columns
6. **Ensured reusability**
   * Wrote the process inside an async main() function so it can be reused as part of a **larger ingestion pipeline**.
   * Can run any time to get fresh copies of the data — no manual steps.

📌 **Why this follows Manning’s technique**

* **Automated**: No manual downloads.
* **Reproducible**: Script can be re-run any time for consistent results.
* **Self-contained**: Handles download, extraction, and loading in one place.
* **Validates the source**: Checks for HTTP errors before continuing.
* **Extensible**: Can be expanded with logging, caching, and version control later.

**Chapter 2 - Data Ingestion**

**Why "Data Ingestion"**

In ML system design, **data ingestion** refers to:

* Moving data **from a source** (files, APIs, streams)
* Into a **storage system** (DB, data lake, warehouse)
* Often with **light transformations** (like adding timestamps, cleaning field names, or adding unique IDs).

Putting your dataset into a database is not just an academic exercise, it’s a **practical ML system design decision** that solves several problems you’d run into if you only kept raw CSVs.

Here’s why:

## **1. Persistence and Reliability**

* A CSV is just a file. If it gets deleted, moved, or corrupted, you’re stuck.
* A database ensures the data is stored in a **managed, queryable, fault-tolerant** environment.
* You can recover it anytime without going back to the UCI website.

## **2. Querying and Filtering Without Loading All Data**

* With CSVs, you typically pd.read\_csv() the **entire file** into memory.
* In MongoDB (or any DB), you can query **only what you need**:

python

CopyEdit

db.students.find({"school": "GP", "G3": {"$gte": 15}})

This means less memory use and faster prototyping.

## **3. Enrichment with Metadata**

* In ingestion, you can add **timestamps**, **source tags**, or **processing status** columns.
* For example:
  + ingested\_at: "2025-08-12T10:23:00Z"
  + source: "UCI student-mat.csv"
* This makes your data **traceable** for debugging or auditing.

## **4. Multiple Pipeline Access**

* If you keep data in CSVs, **only one process at a time** can realistically read/write without conflicts.
* In a DB, **multiple systems** can access it:
  + Model training job
  + Data validation service
  + API for external queries

## **5. Easier Integration with Real-Time ML**

* Many production ML systems are **not batch-only**.
* If you later want a **real-time dashboard** of student performance or an **API** where someone can query their predicted grade, the model’s feature pipeline can pull directly from the DB.

## **6. Scalability Beyond One File**

* Your current student dataset is small (395 + 649 rows), but imagine if you collected **years of student performance data** across multiple schools.
* Databases scale better for storage and indexing than keeping dozens of separate CSVs.

**Data Ingestion into Database**

* **Set up MongoDB Atlas** (cloud database) instead of a local install.
* Created a **connection string** for secure access.
* Wrote a Python ingestion script to:
  + Load CSVs into pandas DataFrames.
  + Tag each record with dataset ("math" or "portuguese").
  + Add ingested\_at timestamp.
  + Insert all records into MongoDB.
* Verified insertion count: **1044 documents** stored.
* Viewed a **sample record** with \_id, original features, dataset tag, and timestamp.

So right now:

* You have the **raw data safely stored** in a database (versioned by ingestion time).
* You can **query it any time** to feed into preprocessing or model training.
* This is exactly what Manning recommends in *Data Collection & Feature Extraction* for robust ML system design.

**Chapter 3 - Query and sampling**

**Query & Sampling**.

We’ll:

1. **Connect** to the MongoDB database.
2. **Query** for all records or specific subsets.
3. **Load into pandas** for analysis.
4. **Prepare for preprocessing** in the next stage.

Here’s a summary of what we accomplished in the **Query & Sampling** chapter:

1. **Queried the database** — We pulled data back from MongoDB collections using queries.
   * For example, retrieving all records or filtered subsets.
2. **Loaded the data into pandas** — Converted the MongoDB cursor (query result) into a pandas DataFrame for easy manipulation and analysis.
3. **Explored the dataset shape and contents** — Checked how many records and columns we have, and previewed sample rows to confirm data integrity.
4. **Performed sampling** — Retrieved a smaller, random subset of data (e.g., 100 rows) from the database to use for quick experiments and prototyping without loading the entire dataset.
5. **Prepared for ML pipeline** — By having data in pandas, the next steps (cleaning, feature engineering, model training) become straightforward.

**Chapter 4 - Data Preprocessing & Feature Engineering**

### Data Preprocessing & Feature Engineering

1. **Loaded full dataset from MongoDB into pandas DataFrame**  
   We queried and sampled the data, bringing it locally for processing.
2. **Handled missing data and cleaned columns**  
   Ensured data consistency, fixed types, and handled any missing or inconsistent entries.
3. **Converted categorical variables to numeric form**
   * Used one-hot encoding (dummy variables) for categorical features like reason, school, Mjob, etc.
   * Converted boolean labels (passed) from True/False to 1/0 for modeling.
4. **Engineered new features**
   * Binned age into meaningful groups such as teenage (15-17), young adult (18-20).
   * Created binary flags (like high\_absences) for domain-specific features.
5. **Processed timestamp column**
   * Kept ingested\_at as metadata but didn't include it in modeling features.
6. **Prepared the dataset for modeling**
   * Ensured all columns were numeric or encoded properly.
   * Verified shape and data distribution.

**Chapter 5 – Model Building**

**Goal:** Build a predictive model using your cleaned and preprocessed data.

**What Manning emphasizes:**

* Start with simple, interpretable models (e.g., Naive Bayes, Logistic Regression, Decision Trees).
* Use boolean or numeric features transformed thoughtfully (e.g., binarize, encode categorical).
* Build pipelines that encapsulate preprocessing, feature extraction, and modeling steps.
* Train on a training set, ensuring no data leakage.
* Focus on understanding the problem, feature relevance, and the model’s assumptions.

**What we did:**

* Encoded categorical variables (one-hot or dummy encoding).
* Binarized target variable (passed column).
* Selected features, including engineered features (e.g., binned age, absence flags).
* Built a classification model (e.g., Logistic Regression or Random Forest).
* Trained the model on training data.

**Chapter 6 – Model Evaluation**

**Goal:** Assess the trained model’s performance fairly and thoroughly.

**What Manning emphasizes:**

* Split data into training and test (or validation) sets to measure generalization.
* Use multiple metrics to understand performance beyond accuracy:
  + Precision, Recall, F1 score — especially for imbalanced classes.
  + Confusion matrix for insight into types of errors.
  + Precision-Recall curve and AUC for threshold tuning.
* Analyze errors and decide on next steps (e.g., feature engineering, model tuning).
* Avoid overfitting by careful validation.

**What we did:**

* Evaluated the model on test data.
* Calculated metrics: Accuracy, Precision, Recall, F1-score.
* Generated and interpreted confusion matrix.
* Plotted Precision-Recall curve.
* Observed high recall and precision, indicating a strong model.
* Discussed next steps such as model tuning.
* 

We performed hyperparameter tuning (e.g., adjusting regularization strength) to improve model performance.

* The final tuned model achieved perfect precision, recall, and F1-score on the test set.

**Model Saving:**

* We saved the trained logistic regression model (logreg) for later reuse, ensuring reproducibility and easy deployment
* We saved the dummy and scaler as pickle