**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 & Evaluation**

1. **Data Preparation**
   * Loaded the student dataset with all features.
   * Handled missing values (if any) and ensured data consistency.
   * Split features (X) and target (y) (passed column).
2. **Feature Engineering**
   * Encoded categorical features (school, sex, Mjob, Fjob, etc.).
   * Scaled or normalized numerical features if necessary.
3. **Model Training**
   * Built a **pipeline** with preprocessing + model (e.g., Logistic Regression).
   * Performed **train-test split** for evaluation.
   * Optionally used **cross-validation** for robust model selection.
4. **Model Evaluation**
   * Evaluated model using metrics like **accuracy**, **precision**, **recall**, etc.
   * Verified predictions are sensible with sample inputs.
5. **Model Saving**
   * Saved the **fitted pipeline** using joblib.dump() for deployment.

**Outcome:** A validated, production-ready ML model (performance\_pipeline.pkl) that can predict whether a student passes.

**Chapter 6 – Flask app for model Serving**

**Objective:** Build an API to serve the ML model for real-time predictions.

**Steps Completed:**

1. **Setup Flask App**
   * Initialized a Flask application (app.py).
   * Created endpoints:
     + GET / – basic health check.
     + POST /predict – accepts student data JSON and returns prediction.
2. **Load Model**
   * Loaded the saved pipeline using joblib.load().
3. **Input Handling**
   * Converted incoming JSON into a **Pandas DataFrame**.
   * Ensured column names and types match the model pipeline.
4. **Prediction**
   * Called model.predict(df) on input data.
   * Returned prediction as JSON: {"passed": 1}.
5. **Testing**
   * Verified endpoint returns **HTTP 200**.
   * Checked correct predictions are returned for sample input.

**Outcome:** A **fully functional Flask API** capable of serving predictions for your student dataset.

**Chapter 7 – MongoDB integration with Flask**

**Objective**

This chapter focused on connecting a Flask web application to a MongoDB Atlas cluster, creating API endpoints for student records, and verifying the database connection.

**1. MongoDB Connection Setup**

* Installed necessary Python packages for Flask and MongoDB integration.
* Connected the Flask application to MongoDB Atlas using a connection string.
* Implemented a helper function to verify that the connection to MongoDB was successful.

**2. Flask Application Structure**

* Initialized the Flask app to serve as the backend API.
* Created a home route that indicates the API is running and displays the MongoDB connection status.

**3. API Endpoints**

* **Get all students**: Fetch all student records from MongoDB.
* **Add a new student**: Insert a new student record into the database.
* **Get student by ID**: Fetch a single student record using its unique identifier.
* Included connection checks in all endpoints to ensure database availability.

**4. MongoDB Connection Testing**

* Verified the database connection both through the Flask app and with a standalone Python script.
* Ensured that the connection to MongoDB Atlas is active and functional.

**5. Key Learnings**

* Connecting Flask to a remote MongoDB cluster.
* Structuring a Flask API with multiple endpoints for CRUD operations.
* Importance of checking database connectivity before performing operations.
* Returning structured JSON responses for API endpoints.

**Chapter 8 – Building and Testing Prediction API**

**Objective**

* Create a Flask-based API that connects to MongoDB and serves predictions from a trained ML model (student dropout/pass prediction).
* Handle both single and batch student data with input validation.

**Steps Completed**

1. **MongoDB Integration**
   * Connected Flask to a MongoDB cluster using a connection string.
   * Implemented a helper function to **check MongoDB connectivity** (ping command).
   * Created API routes to:
     + List all student records.
     + Add new student records.
     + Retrieve a student by ID.
   * Verified MongoDB connection successfully through the API homepage.
2. **ML Model Loading**
   * Loaded a pre-trained logistic regression model (dropout/pass prediction).
   * Ensured the model was ready to accept structured input for prediction.
3. **Prediction Endpoint**
   * Created a /predict POST endpoint.
   * Implemented:
     + **Single student prediction**.
     + **Batch prediction** (multiple students at once).
     + **Input validation** to check for missing features.
   * Designed the endpoint to return predictions in **JSON format**.
4. **Input Validation**
   * The API checks that all required features (columns) are present before sending data to the model.
   * Returns an informative error if any features are missing.
5. **Testing the API**
   * Tested **single-student prediction** using:
     + **PowerShell** with Invoke-RestMethod.
     + **Postman** as an alternative for GUI-based testing.
   * Learned that **sending all required features** is necessary for the model to work.
   * Successfully received a prediction from the API, confirming end-to-end functionality.
6. **Challenges and Solutions**
   * **PowerShell curl issues**: Single vs. double quotes, backslashes, and escaping caused JSON parsing errors.
   * **Method Not Allowed errors**: Resolved by using POST requests instead of GET when accessing /predict.
   * **Missing columns errors**: Solved by sending all expected features in the request payload.

**Outcome**

* A fully functional **Flask API** that connects to MongoDB, validates inputs, and returns ML predictions.

**Chapter 9 – Logging into Mongodb Flask file**

## Why Logging Matters

Logging serves multiple purposes in ML systems:

1. **Real-Time Visibility**: Know what the API is doing right now.
2. **Error Tracking**: Quickly detect and troubleshoot issues, like failed database connections or invalid requests.
3. **Audit & Debug**: Maintain a trail of predictions, inputs, and outputs to reproduce or debug problems.
4. **Compliance & Governance**: Some industries require detailed records of ML predictions and system behavior.

In Manning’s framework, logging sits alongside monitoring and feedback loops. It ensures the system is transparent, auditable, and reliable.

## What We Implemented

Our API handles student records, offering endpoints to:

* Retrieve all students
* Add a new student
* Retrieve a student by ID

For each of these endpoints, we implemented logging to capture:

* **Request inputs**: What data was sent to the API
* **Responses**: What the API returned
* **Errors and exceptions**: Invalid IDs, MongoDB connection failures, or unexpected server issues

We used Python’s built-in logging module for simplicity and extensibility. All logs are printed to the console for now, but this can be easily extended to external services like **MLflow**, **Sentry**, or the **ELK stack**.

**NEXT PROJECT**

In your **next project**, you can focus on the “industrial-scale” features:

* Automated feature pipeline
* Persistent Feature Store
* Model registry with versioning
* Scheduled batch predictions
* Automation layer