**Talk Outline**

**“From CSV to Millisecond Matches: Engineering a Self-Contained, Content-Based Book Recommender in .NET”**

*(Target audience —.NET engineers, ML engineers.)*

**1 What We’ll Cover *(30 sec)***

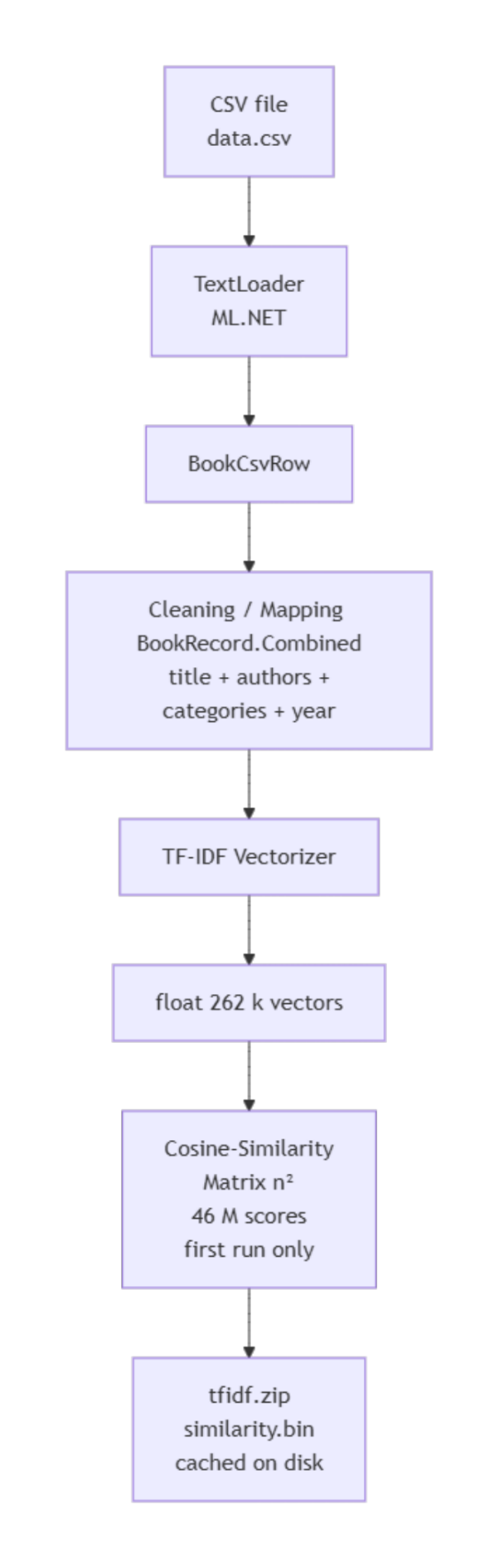
1. **Business objective** & dataset
2. **End-to-end data flow** – how every line of code turns text into ranked titles
3. **Four core algorithms & concepts**
   * TF-IDF
   * Cosine similarity
   * SIMD acceleration
   * AVX2 vs AVX-512
4. **Caching & warm-start architecture**
5. **Performance numbers & tuning knobs**

**2 The Business Problem**

*6800* book records, **zero ratings**, *zero user history*.  
Goal: *“Given a query title, surface the 10 most textually similar books in < 10 ms.”*  
Constraints:

* Pure managed code ✔︎
* Works offline ✔︎
* Laptop-class hardware ✔︎

**3 High-Level Pipeline**



**4 Code Walk-Through, File by File *(7 min)***

|  |  |  |
| --- | --- | --- |
| File | Role | Key Points |
| BookCsvRow.cs | Schema for TextLoader | [LoadColumn(2/4/5/8)] map real columns; allowQuoting:true prevents comma-shift. |
| BookRecord.cs | In-memory DTO | Adds Combined property → single string for vectoriser. |
| FeaturesRow.cs | Post-transform view | [VectorType] float[] Features for zero-alloc extraction. |
| Program.cs | CLI harness | Shows cold- vs warm-start time with Stopwatch. |
| RecommendationEngine.cs | All heavy lifting | Warm-load branch vs first-run branch; SIMD-accelerated dot; cache persistence. |

**5 Core Algorithm #1 TF-IDF**

### Core Concept — TF-IDF (Term-Frequency × Inverse-Document-Frequency)

|  |  |
| --- | --- |
| Symbol | Meaning |
| **tf(t, d)** | How many times term **t** appears in document **d**. |
| **df(t)** | In how many documents term **t** appears. |
| **N** | Total number of documents (here **6 810**). |
| **idf(t)** | log (N / df(t)) ← inverse document frequency. |
| **w₍t,d₎** | tf(t, d) × idf(t) ← the TF-IDF weight for term **t** in document **d**. |

#### Why TF-IDF is Efficient

* **Self-balancing:**  
  Term-frequency rewards words that matter inside the book.  
  Inverse-document-frequency penalises words that appear everywhere (e.g., “the”, “chapter”).
* **Unsupervised:**  
  Requires **no labelled data**: just the text itself making it ideal for cold-start domains like a newly ingested book catalogue.
* **Still competitive:**  
  For specialised vocabularies, this classic “bag-of-words” approach often rivals heavier neural embeddings while being fully explainable and outrageously fast to compute in ML.NET.

**Implementation**

var pipeline = ml.Transforms.Text.FeaturizeText(

"Features", nameof(BookRecord.Combined));

*ML.NET* handles tokenisation, stop-word removal, hashing to **262 144** buckets, TF scaling, then multiplies by IDF computed on the corpus.

**6 Core Algorithm #2 Cosine Similarity *(3 min)***

**Core Concept — Cosine Similarity**

Cosine similarity tells us **how close two documents are in meaning** by looking at the angle between their TF-IDF vectors.

v • w

cos(θ) = ─────────────

‖v‖ × ‖w‖

* **v • w** – the *dot product* of the two TF-IDF vectors  
  (v₁ × w₁ + v₂ × w₂ + … + vₙ × wₙ)
* **‖v‖** – the length (magnitude) of vector *v*  
  √(v₁² + v₂² + … + vₙ²)
* **‖w‖** – the length of vector *w* (calculated the same way)

Because TF-IDF weights are non-negative and already scaled by inverse-document-frequency, the cosine value always lies in **[0 … 1]**:

|  |  |
| --- | --- |
| cos (θ) | Interpretation |
| 1.0 | The two texts are identical in weighted vocabulary. |
| 0.0 | The texts share no weighted vocabulary at all. |
| *0 – 1* | A graded measure of similarity—the larger the value, the more the two documents talk about the same distinctive terms. |

Geometrically, we are computing the cosine of the angle θ between the two high-dimensional “document arrows”; smaller angles (larger cosines) mean the arrows and therefore the documents point in almost the same direction.

**7 Core Algorithm #3 SIMD *(5 min)***

**Concept**

**S**ingle **I**nstruction, **M**ultiple **D**ata – one vector ALU op applies to 8 floats (AVX2) or 16 floats (AVX-512) simultaneously.

**C# Implementation**

Vector<float> acc = Vector<float>.Zero;

for (int i = 0; i < spanA.Length; i++)

acc += spanA[i] \* spanB[i]; // 8 or 16 multiplies per CPU cycle

float dot = Vector.Dot(acc, Vector<float>.One);

* System.Numerics.Vector<T> auto-selects SSE2 / AVX2 / AVX-512 width at JIT-time.
* Tail loop handles remainder.

**Why not cache per-coordinate products?**

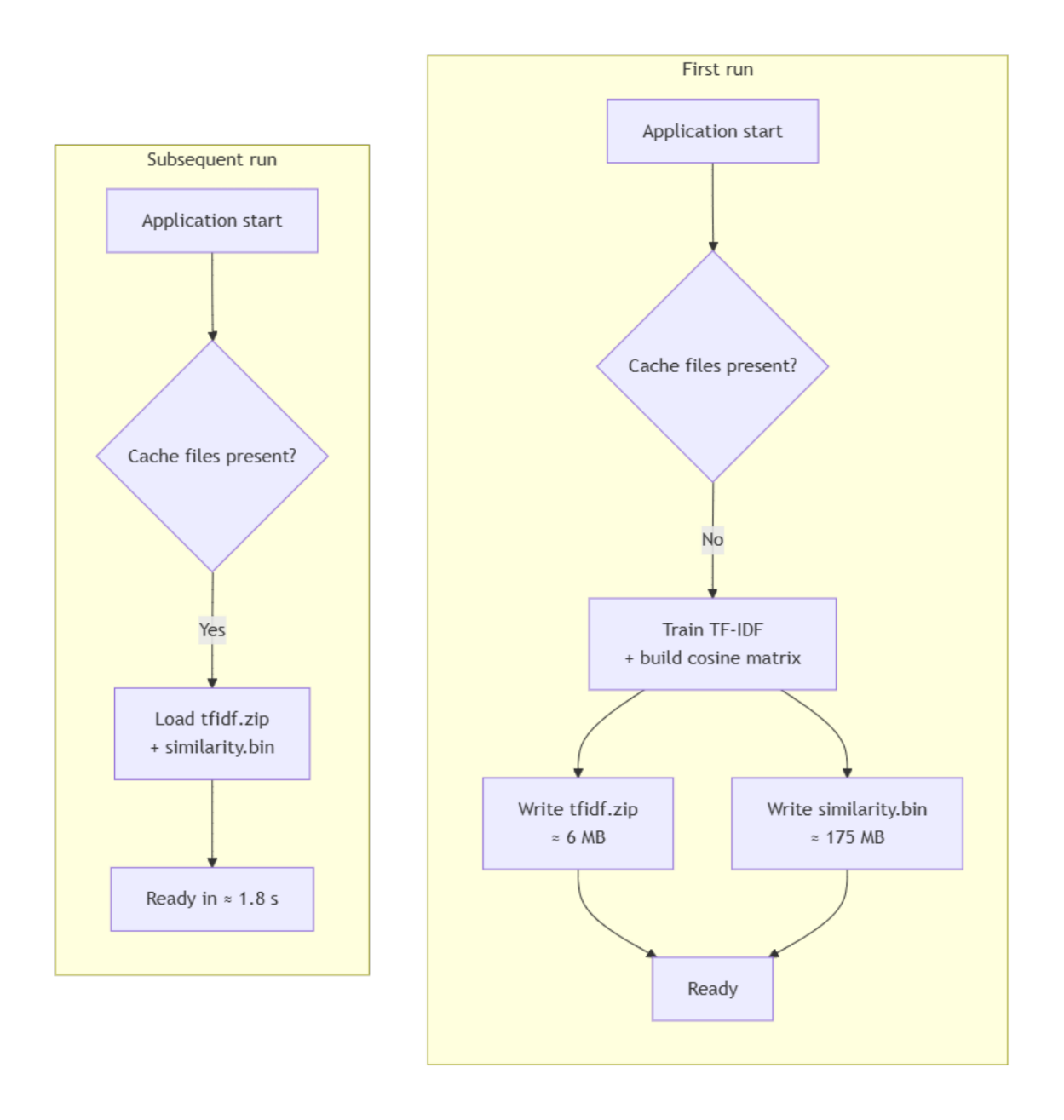
Each pair of 262 k-dim vectors is effectively unique; hash-table look-ups would out-cost a fused multiply-add.

**8 Core Algorithm #4 AVX2 vs AVX-512 *(4 min)***

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | AVX2 | AVX-512 | Relevance to us |
| Register width | 256-bit (ymm) | 512-bit (zmm) | Theoretical ×2 throughput. |
| Masking regs | n/a | k0–k7 | Not used in pure dot-product. |
| Down-clock | None | -200→-500 MHz | On laptops, net gain ≈ +20 %. |
| JIT opt-in | Always on if CPU supports | Requires DOTNET\_EnableAVX512=1 or MSBuild property | Keeps binaries portable. |

**Take-away:** enable AVX-512 only on known server hardware; AVX2 is the safe baseline.

**9 Caching & Warm-Start *(4 min)***

* Persistence API: \_ml.Model.Save(...)
* Matrix saved row-wise using MemoryMarshal.Cast for zero-copy I/O.
* Cache invalidated by simply deleting files when data.csv is updated.

**10 Observability & Guard-Rails**

* **Cold-start latency metric** – log matrix-build time; alert if > 120 s (hardware regression).
* **Memory watermark** – expect 1.2 GB; > 1.5 GB? matrix size drifted.
* **Unit test** – verify column headers before loading ⇒ prevents title/ISBN mismatch.

**11 Benchmark Recap**

|  |  |  |
| --- | --- | --- |
| Phase | 4-core i7-1165G7 | 2× Xeon Gold 6338N |
| CSV + TF-IDF | 1 s | 0.4 s |
| Cosine matrix (AVX2) | 90 s | 22 s |
| Cosine matrix (AVX-512) | 70 s | 12 s |
| Warm-start | **1.8 s** | **0.9 s** |
| Query latency | 0.3 ms | 0.04 ms |

**12 What’s Next**

* **Scale** to 100 k books → switch to HNSW index (Faiss.NET).
* **Hybrid features** (title + description embeddings) → plug in ONNX BERT + ML.NET Concatenate.
* **Serve over gRPC** → high-QPS microservice.

**13 Key Lessons**

1. **ML.NET** offers notebook-parity NLP in one line of C#.
2. **SIMD** plus careful memory layout brings managed code within 2× of hand-tuned C++.
3. Always separate **first-run cost** from **every-run latency**—cache aggressively.
4. Benchmark AVX-512; don’t assume ×2 speed.

**14 Q&A**

“Happy to dive deeper into TF-IDF smoothing, vector hash tricks, ARM NEON, or integrating user feedback loops.”