Cicerone

Come insegnare a un computer che film guardare stasera

ngw

Di cosa parliamo oggi

1. **Un po' di machine learning** (senza formule spaventose, promesso)
   - Che problema risolve un sistema di raccomandazione
   - Come "impara" i gusti guardando cosa fanno gli utenti
   - Come si misura se funziona
2. **Cos'è Cicerone**: cosa gli diamo in pasto e cosa ci restituisce
3. **Deploy completo**, un passo alla volta
   - Ogni file di configurazione, riga per riga
   - Per ogni opzione: *cosa succede davvero* se la cambiate

> ⓘ Note
>
> Tutto si riferisce a Cicerone **0.8.5**. Il catalogo di esempio è fatto di film **rigorosamente inventati**: ogni somiglianza con film reali è puramente voluta.

Parte 1 · Machine learning in pillole

Il problema: cosa guardo stasera?

Un catalogo ha migliaia di film, ma una persona ne guarda uno a sera. Nessuno ha voglia di sfogliarli tutti (di solito dopo 40 minuti di scroll si riguarda The Office).

Un **sistema di raccomandazione** sceglie al posto nostro: per ogni persona prepara una **lista corta e ordinata** dei film che con più probabilità le piaceranno.

Due parole che useremo sempre:

- **utente**: chi riceve i consigli
- **item**: la cosa consigliata. Oggi un film, ma può essere qualunque cosa

Servizio	Utente	Item
Netflix	spettatore	film/serie
Spotify	ascoltatore	brano
Amazon	cliente	prodotto

La lista dei primi K item per un utente si chiama **top-K** (es. top-10).

Da dove impara: il feedback

Il sistema non sa niente dei gusti di nessuno. Li ricava da quello che gli utenti **fanno** sul sito. Ogni cosa che fanno è un **evento**.

Feedback esplicito

L'utente ci **dice** cosa pensa: stelle, pollice in su, recensione.

Chiarissimo, ma **raro**: quante recensioni avete scritto questo mese?

Feedback implicito

Lo **deduciamo** da cosa fa: ha guardato il film, ha visto il trailer, l'ha messo in watchlist.

Abbondante, ma **ambiguo**: aprire una scheda vuol dire "forse", non "mi piace".

Quasi tutti i dati che abbiamo sono **impliciti**. Per questo a ogni tipo di evento diamo un **peso**: quanto ci fidiamo di quel segnale. Guardare un film fino in fondo pesa molto, aprire la scheda pesa poco.

Tutti i dati in una tabella

Mettiamo i pesi in una tabella: una **riga per utente**, una **colonna per film**.

	Squalo	Robot	Frigo	Lunedì	Riunione
Anna	5	.	4	.	.
Bruno	.	4	.	5	.
Carla	4	.	?	.	?
Dario	.	5	.	4	2

Le colonne sono film del nostro catalogo (inventato): *Lo Squalo del Lago di Como*, *Il Robot che Voleva le Ferie*, *Non Aprite quel Frigo*, *Ritorno al Lunedì*, *Riunione che Poteva Essere una Mail*.

- Il numero dice **quanto** quell'utente ha interagito con quel film
- Il punto `.` vuol dire **nessuna interazione**: non sappiamo niente. In un catalogo vero oltre il 99% delle caselle è vuoto
- Raccomandare vuol dire **stimare i numeri mancanti** e proporre i più alti

Guardiamo Carla: ha gli stessi gusti di Anna (tutte e due hanno visto lo Squalo). Ad Anna è piaciuto il Frigo, quindi è un buon consiglio anche per Carla. Questa idea si chiama **filtraggio collaborativo**: gli utenti si consigliano a vicenda, senza saperlo.

Strategia 1 (LightFM): i gusti come numeri

Il modello dà a ogni utente e a ogni film una **lista di numeri** (embedding). Immaginiamo che siano solo due, "paura" e "risate":

```
Anna     = [paura 0.9, risate 0.1]
Squalo   = [paura 0.8, risate 0.3]   0.9×0.8 + 0.1×0.3 = 0.75
Riunione = [paura 0.0, risate 0.9]   0.9×0.0 + 0.1×0.9 = 0.09
```

Anna ama la paura. Lo Squalo fa paura; la *Riunione che Poteva Essere una Mail* fa solo ridere (nel catalogo è un horror, ma solo per chi c'era).

Si moltiplicano i numeri a coppie e si sommano: più è alto, più il film è adatto. Ad Anna consigliamo lo Squalo.

Allenare (training) vuol dire trovare quei numeri: si parte **a caso** e si correggono un po' alla volta, finché i film che gli utenti hanno davvero visto prendono punteggi alti. In realtà i numeri sono 64 e nessuno sa cosa rappresentino: "paura" e "risate" li abbiamo inventati noi per capirci.

Il punto forte di LightFM: dà dei numeri anche alle *caratteristiche* (genere, paese...). Così sa cosa consigliare anche a chi si è appena iscritto e ha solo detto "mi piace l'horror".

Strategia 2: "chi ha visto questo ha visto anche..."

Si chiama **item-based KNN** (i K vicini più simili). Funziona in due passi.

Passo 1, una volta per tutti: per ogni coppia di film si misura quanto sono simili. Due film sono **simili se li hanno visti le stesse persone**. Qui "paura" e "risate" non c'entrano: KNN non sa niente del film, guarda solo **chi** l'ha visto.

```
"Lo Squalo del Lago di Como" è stato visto da 30 persone.
Di queste:
  20 hanno visto anche "Non Aprite quel Frigo"  → molto simile
   2 hanno visto anche "Riunione che Poteva
     Essere una Mail"                           → poco simile
```

Passo 2, per ogni utente: si prendono i film che ha già visto e si propongono i loro vicini più simili, che non ha ancora visto.

- **Pro:** facile da spiegare all'utente: *"perché hai visto lo Squalo"*
- **Contro:** serve una storia. A chi non ha visto niente non sa cosa dire

Il problema della partenza a freddo (cold start)

Le strategie viste finora imparano dalla storia. Ma a volte non c'è:

- **utente nuovo:** non ha visto niente, non sappiamo se è da horror o da commedia
- **film nuovo:** non l'ha visto nessuno, quindi nessuno lo consiglia, quindi nessuno lo vede... e così via

Per ogni situazione c'è una strategia adatta, tutte presenti in Cicerone:

- **Utente nuovo, non sappiamo niente** → `popular` (i più visti di sempre) o `latest` (i più visti degli ultimi 14 giorni)
- **Utente nuovo che ci ha detto i generi preferiti** → `collaborative`: LightFM usa solo le sue caratteristiche
- **Film nuovo, mai visto da nessuno** → `content_fallback`: lo propone a chi ha visto film con lo stesso genere, regista, decennio...

Un sistema **ibrido** usa più strategie insieme: personalizza quando ha i dati, ripiega sul popolare quando non li ha.

Quando si calcolano i consigli?

"Live", a ogni richiesta

Quando l'utente apre la home, il modello calcola i consigli in quel momento.

- **Pro:** sempre aggiornati
- **Contro:** ogni richiesta è lenta e costosa, e il modello deve stare in memoria sul server web

"Batch", in anticipo

Una volta per notte si calcola la top-K di **tutti** e la si salva. Quando l'utente apre la home, la si legge e basta.

- **Pro:** risposte velocissime, server leggero; se il training si rompe restano i consigli di ieri
- **Contro:** se stasera guardate un horror, i consigli cambiano domattina

---

La via di mezzo: batch + aggiornamenti in giornata

Cicerone fa **tutte e due le cose**: di notte un giro completo in batch; durante il giorno gli eventi nuovi aggiornano, entro un minuto, i consigli di chi li ha fatti, anche **allenando ancora il modello** (online training). Chi chiede i consigli continua a leggere una lista pronta (passo 12).

Come si misura se funziona?

Offline: con i dati che abbiamo già

Facciamo finta di essere nel passato e vediamo se il modello avrebbe indovinato.

1. Alleniamo il modello solo sugli eventi **fino al 1° ottobre**
2. Gli facciamo preparare la top-10 per ogni utente
3. La confrontiamo con quello che gli utenti hanno **davvero** visto dopo

```
Anna dal 1° al 14 ottobre ha visto 4 film.
2 di quei 4 erano nella sua top-10.  →  Recall@10 = 2/4 = 50%
```

MAP@K e **NDCG@K** fanno lo stesso conto, ma un film indovinato al 1° posto vale più di uno al 10° (nessuno scrolla fino in fondo). Vanno da 0 a 1: più alto è meglio. [1]

[1] **@K**: si guardano solo i primi K consigli. **MAP** (Mean Average Precision): ogni film indovinato vale "indovinati fin lì ÷ posizione" (1° = 1/1, 3° = 2/3), poi media su tutti gli utenti. **NDCG** (Normalized Discounted Cumulative Gain): ogni film indovinato vale $1/\log_2(\text{posizione}+1)$, diviso per il punteggio della lista perfetta.

Online: con utenti veri

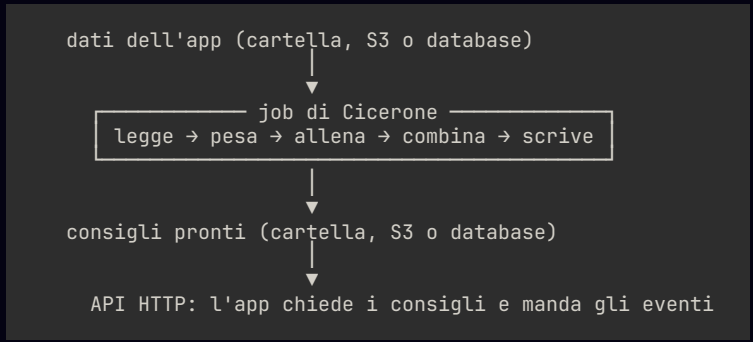
- **CTR**: su 100 consigli mostrati, quanti vengono cliccati
- **A/B test**: metà utenti vede la versione A, metà la B (passo 9b)

Parte 2 · Cicerone

Cos'è Cicerone

Un sistema di raccomandazione pronto da installare. In quattro parole:

- **batch + online**: ricalcola tutto di notte, aggiorna al volo di giorno
- **ibrido**: usa più strategie insieme (quelle viste finora e altre)
- **self-hosted**: gira sui vostri server, i dati non escono da casa vostra
- **generico**: non sa cosa sia un film. Bastano un id utente e un id item



Non si scrive codice: **tutto si configura in file TOML.** Il nome? Il cicerone è la guida che ti dice cosa vale la pena vedere.

Cosa gli diamo in pasto

`events` (obbligatorio): chi ha fatto cosa, e quando

colonna	significato	esempio
`user_id`	chi	`u042`
`item_id`	su quale film	`shrek-all-ikea`
`event_type`	cosa ha fatto	`watched`, `trailer`
`quantity`	quante volte (facoltativo)	`7` (sì, 7 volte)
`occurred_at`	quando	`2026-10-01T21:30:00Z`

`users` e `items` (facoltativi): le caratteristiche

- **utenti**: generi preferiti, paese
- **film**: genere, regista, decennio, durata, se è pubblicato, in quali paesi abbiamo i diritti

Servono a tre cose: consigliare a chi è appena arrivato, filtrare per genere nell'API e applicare le regole di business (passo 8).

Cosa fa il job, passo per passo

1. **Legge** gli eventi, gli utenti e i film
2. **Pesa** gli eventi: per ogni coppia (utente, film) ricava un solo numero, "quanto gli interessa" → *passi 4 e 5*
3. **Allena** le strategie che abbiamo scelto → *passo 6*
4. **Esclude** i film che un utente non può vedere → *passo 8*
5. **Combina** le liste delle strategie in una top-K per utente → *passo 7*
6. **Dà una spinta** ai film che vogliamo promuovere → *passo 8*
7. **Scrive** i consigli, più un riassunto del giro (`manifest.json`)

Ogni consiglio viene salvato con il suo **perché**, così l'app può mostrarlo all'utente:

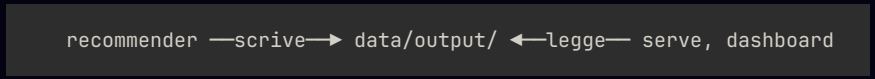
```
"Ti consigliamo Ritorno al Lunedì perché hai visto
 Il Robot che Voleva le Ferie, ed è fantascienza
 come piace a te"
```

Parte 3 · Deploy completo

Tre servizi, una cartella in comune

Cicerone si avvia in tre modi, ognuno con il suo file di configurazione:

Servizio	Cosa fa	Porta
`recommender`	il job: allena e scrive i consigli	8080
`serve`	l'API che l'app interroga (e a cui può mandare gli eventi)	8000
`dashboard`	pagina web per controllare il tutto	8090



Perché separarli? Il job è **pesante** e gira una volta per notte; l'API è **leggera** e deve rispondere sempre, anche mentre il job lavora. Con gli aggiornamenti in giornata (passo 12) anche `serve` scrive in `data/output`.

C'è un quarto file, `features.toml`, con pesi, caratteristiche e regole. Lo legge solo il job.

> Important
>
> serve e dashboard devono leggere **lo stesso posto** in cui scrive il job. Se sbagliate, mostrano dati vecchi senza nessun errore.

Passo 1 · I segreti

Le API sono protette da un **token**: una specie di password che l'app manda a ogni richiesta. I token **non** vanno nei file TOML (che magari finiscono su git), ma nel file `.env`:

```
SERVE_AUTH_TOKEN=cambiami-serve       # per leggere i consigli
TRIGGER_AUTH_TOKEN=cambiami-trigger   # per lanciare un retrain
```

Nel TOML si scrive solo il **nome** della variabile. All'avvio Cicerone lo sostituisce con il valore vero:

```
auth_token = "${SERVE_AUTH_TOKEN}"
```

Se la variabile non esiste, Cicerone **si rifiuta di partire**: meglio un errore subito che un'API aperta a tutti.

```
cp .env.example .env     # poi cambiate i due token
mkdir -p data            # la cartella per i dati
docker compose build     # costruisce l'immagine di Cicerone 0.8.5
```

Passo 2 · Da dove leggo, dove scrivo

In `config/cicerone.toml` si dice dove sono i dati e dove mettere i risultati:

```toml
[input]
kind = "dataset"               # file su disco o su S3

[input.options]
storage_backend = "local"      # "local" o "s3"
path = "data/input"            # events, users, items

[output]
kind = "dataset"

[output.options]
storage_backend = "local"
path = "data/output"           # qui finiscono i consigli
```

- `dataset` vuol dire file **parquet**: tabelle come un CSV, ma compresse e più veloci da leggere
- `storage_backend = "s3"`: gli stessi file, ma su un bucket. **Solo parquet**, con nomi fissi (`events.parquet`...): un CSV **non** viene letto, va convertito prima (es. `pd.read_csv(...).to_parquet(...)`)
- `kind = "db"`: si legge (o si scrive) su un database, con una query SQL
- Input e output sono indipendenti: si può leggere da Postgres e scrivere su S3

Passo 3 · Quando e quanto lavora il job

```toml
[job]
mode = "batch"
top_k = 10
cron_schedule = "0 3 * * *"
```

- `mode`: `batch` vuol dire "questo processo allena". Con `serve` invece non allena e fa solo da API (lo vediamo al passo 10)
- `top_k`: quanti consigli salvare per ogni utente. Più alto = più scelta quando l'app filtra per genere, ma file più grande
- `cron_schedule`: quando ripartire. Si legge "minuto 0, ora 3, ogni giorno", in ora **UTC**. Con `"0 */6 * * *"` gira ogni 6 ore: consigli più freschi, server più impegnato

Tre modi di avviare il job:

- `cicerone start`: fa subito un giro, poi segue il cron (lo usa Docker)
- `cicerone job`: fa un giro solo e termina
- `cicerone scheduler`: segue il cron, senza fare subito il primo giro

Passo 4 · Dagli eventi a un numero

Per ogni coppia (utente, film) il job **somma i pesi** di tutti i suoi eventi.

```toml
# config/features.toml
quantity_scaled_events = [
  "watched",
]

[event_weights]
watched = 4.0
review_positive = 5.0
review_negative = -3.0
watchlist = 2.0
trailer = 1.0
view = 0.3

[event_caps]
view = 5
```

```
Anna e lo Squalo
  3 visioni    4 × ln(1+3) =  5.5
  trailer                     1.0
  recensione +                5.0
  totale                     11.5
```

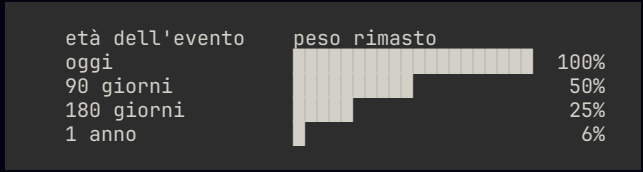
- `event_weights`: quanto vale ogni evento. Un tipo non elencato viene **ignorato**
- **Peso negativo**: toglie. Se il totale è ≤ 0 la coppia viene scartata (da noi 40 coppie)
- `quantity_scaled_events`: conta anche *quante volte*, ma con il logaritmo: 1 volta = 2.8, 3 = 5.5, 7 = 8.3. La settima visione non vale come la prima
- `event_caps`: contano solo gli ultimi 5 `view` per utente e film. Chi apre 200 volte la scheda non "gonfia" niente

Passo 5 · Il tempo conta: half_life_days

I gusti cambiano: un film visto un anno fa dice meno di uno visto ieri.

```
[job]
half_life_days = 90
```

Ogni evento **perde metà del suo peso** ogni 90 giorni:



Esempio: Anna un anno fa guardava solo horror (peso 4 → **0.24**), la settimana scorsa una commedia (peso 4 → **3.8**). Oggi conta la commedia.

Se lo cambio:

- **più basso** (es. 14): segue le mode e dimentica in fretta
- **più alto** (es. 365): gusti stabili, ma reagisce piano ai cambiamenti

Passo 6 · Quali strategie usare

```
[job]
models = ["collaborative", "item_based", "popular", "latest"]
```

Ogni strategia prepara la sua top-10 per ogni utente. Di default si arriva a una sola lista **per priorità**: si mettono in fila le liste nell'ordine di `models`, si saltano i doppioni e ci si ferma a 10.

```
Anna (12 film visti)          Ugo (appena iscritto)
collaborative → 10 film       collaborative → 10 film
item_based    → non serve     item_based    → non serve
```

Quindi la prima strategia riempie quasi sempre tutto e le altre sono una **riserva**, per quando la prima non ha abbastanza consigli. Sul nostro dataset tutti i 2000 consigli vengono da `collaborative`.

- **Se cambio l'ordine:** con `popular` per primo tutti vedono gli stessi blockbuster. Errore classico
- **Se aggiungo strategie:** più riserve, ma il job ci mette di più

Per far *collaborare* davvero le strategie servono i prossimi due passi.

Passo 7a · Combinare a voti: RRF

La priorità non si accorge se più strategie concordano. **RRF** fa **votare** tutte le strategie; si sommano i voti e vince chi ne ha di più:

```
voto = peso / (rrf_k + posizione)
```

```toml
[job]
rrf_k = 60

[job.model_weights]
collaborative = 1.0
item_based = 0.6
popular = 0.3
```

Film A: 1° per collaborative, 5° per popular → 1.0/61 + 0.3/65 = **0.021**

Film B: 1° solo per popular → 0.3/61 = **0.005**

Vince A: lo votano due strategie, e una è quella di cui ci fidiamo di più.

- Basta scrivere `[job.model_weights]` per passare da priorità a RRF. `rrf_k` va **sopra**: in TOML una chiave appartiene alla tabella sopra di lei
- `rrf_k` **basso** (1): il 1° posto vale molto più del 10° (1/2 contro 1/11)
- `rrf_k` **alto** (60): 1/61 contro 1/70, quasi uguali. Conta il **consenso**

Passo 7b · Combinare in base all'utente: blending

Chi ha visto un film e chi ne ha visti cento non dovrebbero avere la stessa ricetta. Il **blending** mescola *personalizzati*, *più visti* e *ultime uscite* in proporzioni diverse **per ogni utente**.

```toml
# in config/features.toml
[blending]
enabled = true
curve = "sigmoid"
midpoint = 5.0
steepness = 1.0
popular_share = 0.7
```

film visti	person.	più visti	ultime
0	0%	70%	30%
1	2%	69%	29%
5	50%	35%	15%
10	99%	<1%	<1%

- `curve`: la forma del passaggio. `sigmoid` = a S (lento, veloce intorno al midpoint, poi piatto); `linear` = costante fino a `saturate_at`
- `midpoint`: con quanti film si arriva a metà personalizzati
- `steepness`: quanto è ripida la S. 0.5 = graduale (1 film 12%, 10 film 92%); 3 = quasi un interruttore (4 film 5%, 6 film 95%)
- `popular_share`: come si divide il resto (70% più visti, 30% ultime)
- Salva anche una lista per chi **non ha storia** (la usa l'API). Se attivo, sostituisce priorità e RRF

Passo 8 · Regole di business

Il modello pensa solo a cosa vi piacerà. Ma ci sono film che **non possiamo** e film che **vogliamo** consigliare.

```toml
# in config/features.toml
item_availability_filters = [
  "published", "licensed",
]

[[eligibility]]
name = "rights_in_country"
op = "user_in_item_list"
user_column = "country"
item_column = "available_countries"

[[boost]]
name = "originals"
kind = "boolean"
item_column = "is_original"
factor = 1.5
```

Disponibilità: licenza scaduta → il film non esce **a nessuno**

Eligibility (filtro duro): il paese dell'utente dev'essere tra i paesi del film. Niente diritti in Svizzera → non esce **agli svizzeri**

Boost (spinta morbida): punteggio × 1.5 agli "original" tra i 30 migliori candidati (3 × top_k), poi si riordina. Dal 20° posto può entrare nei primi 10, dal 200° no

Sul nostro dataset:

- **0** film consigliati senza diritti
- gli original passano da **0 a 185** consigli su 2000; primi per 107 utenti su 200
- `factor` alto = salgono troppo: tutti vedono solo quelli

Passo 9 · Far scegliere alla macchina: AutoML

Invece di scegliere noi strategie e pesi, il job fa la **verifica sul passato** (slide 11) a ogni giro e tiene la combinazione migliore.

```
[job.automl]
enabled = true
n_splits = 2
test_days = 14
primary_metric = "MAP"
```

- `test_days`: quanti giorni nasconde
- `n_splits`: quante volte ripete la prova. Più alto = scelta più affidabile, job più lungo
- `models` e `model_weights` scritti a mano **non contano più**

Sul nostro dataset:

candidato	MAP@10	
RRF di tutte	0.248	←
collaborative	0.221	
item_based	0.214	
popular	0.098	
random (a caso)	0.069	
latest	0.042	

- Insieme le strategie si correggono
- `random` è il **termine di paragone**: `latest` ha perso contro il caso!

Passo 9b · A/B test: decidono gli utenti veri

AutoML guarda il passato; un **A/B test** lo chiede agli utenti veri.

```toml
# cicerone.toml e cicerone.serve.toml
[experiment]
enabled = true
id = "lightfm-vs-knn"
primary_metric = "watched"

[[experiment.variants]]
name = "control"   # eredita [job]
traffic = 0.5

[[experiment.variants]]
name = "treatment"
traffic = 0.5
models = ["item_based",
  "collaborative", "popular"]
```

- Il job scrive **una lista per variante** per ogni utente. Da noi: `control` tutta LightFM, `treatment` 1870 consigli su 2000 da KNN
- `serve` sceglie con un **hash dell'id**: u001 è sempre `control`, anche con più server. 200 utenti → 100 e 100. La risposta dice `variant`
- Non cambiate `traffic` a test in corso: sposta utenti tra le varianti
- Durante il test l'online training è spento

- Dashboard, pagina **Experiments**: confronta i `watched`. **Promote** dà il 100% al vincitore solo se la differenza è chiara e i controlli passano
- Per i click (CTR) serve anche `[track]`: l'app li manda a `POST /track`

Passo 10 · L'API

`config/cicerone.serve.toml` avvia Cicerone come API, **senza training**:

```toml
[job]
mode = "serve"

[serve]
auth_token = "${SERVE_AUTH_TOKEN}"
port = 8000
default_k = 10
refresh_interval_seconds = 60
category_column = "genre"
```

- `default_k`: quanti consigli se l'app non lo dice. Più di `top_k` è inutile: il job ne ha salvati 10
- `refresh_interval_seconds`: ogni quanto rilegge i risultati del job
- `category_column`: la colonna usata da `?category=`. Con `"decade"` si filtra per decennio

```
curl -H "Authorization: Bearer $SERVE_AUTH_TOKEN" \
  "http://127.0.0.1:8000/recommendations/u001?limit=3"
```

Appena avviata, se il job non ha ancora finito il primo giro, risponde **404** fino al ricaricamento successivo (al massimo un minuto).

Passo 10 · Cosa risponde (e due sorprese)

```
{ "user_id": "u001", "fallback": false,
  "generated_at": "2026-10-19T10:20:30+00:00",
  "items": [{
    "item_id": "ritorno-al-lunedi",
    "source": "personalized",
    "reasons": {
      "similar_items": [
        {"item_id": "il-robot-che-voleva-le-ferie"}],
      "matched_attributes": [
        {"column": "genre", "value": "fantascienza"}]
  } }] }
```

- `generated_at`: quando ha girato il job · `source`: chi l'ha proposto (`personalized` = LightFM) · `reasons`: il perché, pronto per l'app

Sorpresa 1: `?category=horror` **non ricalcola**: filtra i 10 già salvati. Se nessuno è horror, la lista è vuota. Rimedio: `top_k` più alto.

Sorpresa 2: a un utente sconosciuto l'API dà, con `"fallback": true`, la lista generica del blending o, senza, quella di chi ha ricevuto i `popular`. Se non c'è nessuna delle due, **404**: da noi `collaborative` riempie tutto, quindi 404. L'app deve gestirlo.

Passo 11 · Dashboard e retrain su richiesta

Dashboard

Per chi gestisce il sistema: quando ha girato il job, i punteggi, cosa vede un certo utente.

```
[dashboard]
enabled = true
port = 8090
```

Utenti e password (cifrate) stanno nel file indicato da `users_path`. **Senza almeno un utente non parte:**

```
docker compose run --rm dashboard \
  users --config \
  config/cicerone.dashboard.toml \
  add alice
```

Poi: `127.0.0.1:8090/dashboard`

Trigger

Catalogo nuovo alle 15? Non serve aspettare le 3 di notte.

```
[job.trigger]
enabled = true
auth_token = "${TRIGGER_AUTH_TOKEN}"
port = 8080
debounce_seconds = 60
```

- `POST /trigger/retrain` con il token: il job riparte subito
- `debounce_seconds`: per 60 s dopo un avvio le altre richieste sono ignorate. 5 richieste in 10 s = un giro solo
- `poll_input_bucket` (S3): riparte da solo quando arrivano dati nuovi

Passo 12a · Aggiornare di giorno: gli eventi

Il batch di notte non basta: chi guarda un horror alle 21 vuole consigli diversi alle 21:05, non domattina.

```toml
# config/cicerone.serve.toml
[events]
enabled = true
kind = "webhook"

[events.incremental]
batch_size = 100
batch_window_seconds = 60
```

- L'app manda gli eventi a `serve` con `POST /events`, stesse colonne di `events`. Oppure l'API li legge da database, S3, Kafka, Redis o RabbitMQ (`kind`)
- Ogni **100 eventi o 60 secondi** l'API riscrive i consigli **solo** degli utenti coinvolti
- La lettura (`GET`) non cambia: resta una lista pronta, veloce come prima

- **202** vuol dire "ricevuto", **non** "già aggiornato": arriva al giro successivo
- Film e utenti **nuovi** aspettano il giro completo (o il trigger)
- Gli eventi **non** finiscono in `events.parquet`: se l'app non li salva anche lì, il giro di stanotte non li vede

> ⚠ Warning
>
> Con webhook gli eventi in attesa stanno **in memoria**: se l'API si riavvia prima dell'aggiornamento, vanno persi. Meglio db o kafka.

Passo 12b · Online training

Con gli eventi l'API rinfresca le liste. Con `[events.online]` fa di più: **continua ad allenare** LightFM.

```
# config/cicerone.toml (il job)
[job]
save_model_artifact = true

[output]
artifact_hmac_key = "${ARTIFACT_HMAC_KEY}"

# config/cicerone.serve.toml
[output]
artifact_hmac_key = "${ARTIFACT_HMAC_KEY}"

[events.online]
enabled = true
fit_partial_epochs = 1
fit_min_events = 100
```

- Il job salva il modello (`model.artifact`) **firmato** con la chiave del `.env`: l'API rifiuta un file manomesso
- L'API riparte dai numeri di stanotte e li corregge con i soli eventi nuovi (`fit_partial`), non da zero
- `fit_partial_epochs`: quante passate sugli eventi nuovi. 0 = non allena, aggiorna solo la storia
- `fit_min_events`: allena solo da 100 eventi in su
- Solo utenti e film che il job **conosce già**: gli altri aspettano il giro notturno

Passo 12c · Proviamolo: u001 scopre l'horror

u001 ha visto solo thriller e fantascienza. Gli facciamo guardare 3 horror (nella prova `fit_min_events = 1` e finestra di 10 s, per non aspettare):

```
curl -X POST http://127.0.0.1:8000/events \
  -H "Authorization: Bearer $SERVE_AUTH_TOKEN" \
  -H "Content-Type: application/json" \
  -d '[{"user_id": "u001", "item_id": "it-la-bolletta",
       "event_type": "watched", "occurred_at": "..."}, ...]'
# → 202 {"accepted": 3, ...}
```

	prima (stanotte)	~20 s dopo
1°	Fuga dalla Riunione delle 18	Lo Squalo del Lago di Como ★
2°	Ultimo Tango a Pescara	Non Aprite quel Frigo ★
3°	Memento: Dove Ho Messo le Chiavi	It: la Bolletta ★
4°	Ritorno al Lunedì	la Riunione (era 7°)
...	4 film di fantascienza	fuori; entra la Lavatrice Assassina (horror)

★ = source "incremental"; dal 4° in giù: LightFM riallenato

Sorpresa 3: 3 posti su 10 vanno ai film **appena toccati**, anche se già visti. Per un trailer ha senso, per un film appena finito meno; non c'è un'opzione per spegnerlo.

Passo 13 · Si parte!

```
# crea 200 utenti, 48 film e circa 3200 eventi in data/input
docker compose run --rm --no-deps --entrypoint python \
  recommender scripts/make_sample_data.py data/input

docker compose up -d
```

Cosa succede, in ordine:

1. `recommender` fa subito un giro (qualche decina di secondi) e scrive in `data/output`. Poi aspetta le 3 di notte
2. `serve` all'inizio risponde 404; al primo ricaricamento trova i risultati e risponde su `127.0.0.1:8000`
3. `dashboard` è su `127.0.0.1:8090/dashboard`

> ⚠ Warning
>
> Di default i servizi ascoltano su 127.0.0.1: si raggiungono **solo dalla macchina stessa**. Il nostro compose usa network_mode: host proprio per tenerli così. Con host = "0.0.0.0" diventano raggiungibili **da tutta la rete**.

E senza Docker? Con pip

Le configurazioni sono **le stesse**. Serve **Python 3.11**:

```
uv venv --python 3.11 .venv
source .venv/bin/activate
pip install cicerone-recommender==0.8.5   # non "cicerone"!
set -a; source .env; set +a               # carica i token
```

Preparazione, una volta sola:

```
python scripts/make_sample_data.py data/input
cicerone job --config config/cicerone.toml
cicerone users --config config/cicerone.dashboard.toml add alice
```

Poi un terminale per ogni servizio:

```
cicerone start --config config/cicerone.toml
cicerone start --config config/cicerone.serve.toml
cicerone dashboard --config config/cicerone.dashboard.toml
```

I percorsi nelle config sono relativi: **lanciate tutto da questa cartella**.

Riassunto: cosa cambio → cosa succede

Se cambio...	Succede che...
`event_weights`	cambia quali comportamenti contano di più
`half_life_days`	il sistema ricorda più a lungo, o segue le mode
`models`	cambia quale strategia riempie la top-K
`model_weights`	salgono i film su cui più strategie concordano
`[blending]`	i nuovi vedono i più visti, gli altri consigli personali
`[[eligibility]]`	certi film spariscono per certi utenti
`[[boost]]`	certi film salgono in classifica
`[job.automl]`	la combinazione la sceglie il sistema, a ogni giro
`[experiment]`	metà utenti vede una ricetta, metà l'altra
`cron_schedule`	ogni quanto si rifà il giro completo
`[events]`	i consigli si aggiornano di giorno, non solo di notte
`[events.online]`	anche il modello impara di giorno

> ⓘ Note
>
> Una chiave scritta male (es. `top_K` invece di `top_k`) **non** blocca l'avvio: viene ignorata, con un **warning** nei log. Controllateli dopo ogni modifica!

Grazie! Domande?

Documentazione: **cicerone.dev** · Codice: **github.com/torbido-hq/cicerone**

Scrivetemi: **ngw@nofeed.org**

Stasera, nel dubbio: *Riunione che Poteva Essere una Mail*.

Ne avete appena vista una dal vivo.