Decoding the Dating Game
A Data-Driven Comparison of Matchmaking Algorithms

A data-driven comparison of Tinder, Bumble, OkCupid, and Hinge algorithms - their mechanics, strengths, weaknesses, and how they could be improved.

Introduction:

Dating apps are now one of the most common ways people meet, yet they're also surrounded by curiosity and complaints. Each app has its own reputation - Tinder for swiping at scale, Bumble for women making the first move, OkCupid for its compatibility percentages, and Hinge for "most compatible" matches. I wanted to dig deeper and see how these reputations line up with the actual algorithms behind them.

Beyond curiosity, this was also a way to practice applying data science: reverse-engineering how matches are generated, comparing different approaches, and asking whether one algorithm actually produces better or fairer results. In this project, I recreated the core matching logic of Tinder, Bumble, OkCupid, and Hinge, then tested them side by side. The goal was to evaluate their strengths and weaknesses - from overreliance on text similarity to issues like stale profiles or lack of diversity - and explore how tweaks could make the process better.

Dating apps influence millions of connections every day, so understanding their mechanics isn't just academic; it highlights where product design shapes user experience, and where new ideas could lead to fairer, more transparent, and more engaging matches.

Vector Embeddings (SBERT)

To compare profiles by the meaning of their bios, we use SBERT (Sentence-BERT), a transformer model trained for sentence similarity.

Model: all-MiniLM-L6-v2
all → trained on many data sources (general-purpose).
MiniLM → small, efficient transformer.
L6 → 6 transformer layers (compact + fast).
v2 → second version of this setup.

Trade-off: produces 384-dimensional vectors, accurate enough for similarity while being lightweight.

How it works:

1. Tokenize the text → subwords (WordPiece/BPE).
2. Transformer layers encode context for each token.
3. Pooling turns all token vectors into one sentence vector.
4. Normalize (set normalize_embeddings=True) → scale the vector to unit length so cosine similarity is just a dot product.

Example Walkthrough:
Sample Data:

index	user_id	age	sex	orientation	location	bio_text (short)
0	100	26	f	straight	NYC	"i like **hiking** and **music**"
1	101	27	m	straight	NYC	"i love **trails** and **concerts**"
2	102	24	m	gay	SF	"i like **hiking** and **coding**"
3	103	29	f	bi	LA	"i enjoy **swimming** and **coding**"

After SBERT encoding (and normalization), suppose the model gives these **3-D embeddings**

- E[0] (user 100): **A** = [0.7071, 0.7071, 0.0000]
- E[1] (user 101): **B** = [0.6000, 0.8000, 0.0000]
- E[2] (user 102): **C** = [0.7071, 0.0000, 0.7071]
- E[3] (user 103): **D** = [0.0000, 0.0000, 1.0000]

(All four are length 1, i.e., already normalized.)

When the top_k_semantic_neighbors(user_id=100, k=2) is run, the question is *"Who are the top-2 semantic neighbors of user 100?"*

1) Find row index i
For user_id == 100, i = 0 (first row)

2) Compute similarities sims = E @ E[i]
Because embeddings are normalized, dot product = cosine similarity.
Compute dot products with A:
A·A = 0.7071*0.7071 + 0.7071*0.7071 + 0*0
= 0.5000 + 0.5000 + 0 = 1.0000

B·A = 0.6000*0.7071 + 0.8000*0.7071 + 0*0
= 0.4243 + 0.5657 + 0 = 0.9900 (≈ 0.9899)

C·A = 0.7071*0.7071 + 0*0.7071 + 0.7071*0
= 0.5000 + 0 + 0 = 0.5000

D·A = 0*0.7071 + 0*0.7071 + 1*0
= 0 + 0 + 0 = 0.0000

So sims = [1.0000, 0.9900, 0.5000, 0.0000]

3) Exclude self

sims[i] = -1

Now sims = [-1.0000, 0.9900, 0.5000, 0.0000].

4) Grab the top-k indices (fast)

idx = np.argpartition(-sims, k)[:k]

We negate sims so “largest similarity” becomes “smallest negative number.”
For k=2, this returns the indices of the two highest sims. Top two indices are 1 (0.99) and 2 (0.50).

5) Sort those k by true similarity

idx = idx[np.argsort(-sims[idx])]

This sorts the selected indices in descending similarity. idx = [1, 2] (since 0.99 > 0.50).

6) Build the output table

user_id	age	sex	orientation	location	score (cosine)
101	27	m	straight	NYC	0.9900
102	24	m	gay	SF	0.5000

Top-2 semantic neighbors for user 100 based on bio_text meaning alone.

OkCupid

OkCupid's system is grounded in the idea that relationships work better when people agree on the things that matter most. Instead of relying on a single personality test, users can answer as many or as few multiple-choice questions as they like, covering lifestyle, values, politics, and more.

How the app works:
For each question, a user provides their own answer, chooses which answers they would accept from a partner, and marks how important the question is to them. Some questions might be "a little important," while others can be marked "mandatory."

Scoring:
Scoring happens in two directions. OkCupid calculates how satisfied *you* would be with another person's answers, and how satisfied *they* would be with yours. These two scores are blended into a single Match %, which reflects the compatibility of both perspectives. The stronger the agreement on highly important questions, the higher the Match %.

Certain factors act as dealbreakers regardless of answers. Orientation, age, and distance are enforced as gates: if those don't align, the match won't be surfaced at all.

Our implementation captures the spirit of the match % by using semantic text similarity and structured heuristics in place of OkCupid's large Q&A system.

How it's built (my Ok-cupid style algorithm):
Inputs:

- User attributes: user_id, bio_text, age, sex, orientation, diet/drinks/smokes/drugs, lat/lon, location.
- SBERT embeddings for bios (unit-norm).
- Configurable parameters: k, pool_k, max_km, lifestyle strict flags, allow_missing, blend weights.

Candidate generation (hard gates):

- Pre-pool with top semantic neighbors (pool_k).
- Filter by mutual orientation, distance (using max_km), age restrictions ("half candidates age + 7").
- Optional lifestyle hard filters (diet, drinks, smokes, drugs).

Component scoring (soft signals):

- Age score: 1 if age rule holds, else 0.
- Lifestyle score: average of diet/drinks/smokes/drugs matches, strictness controlled by flags; small neutral bump if missing.
- Location score: linear decay with distance (1 − distance/max_km).

Blend & finalization:

- Weighted sum of scores (defaults: text 0.70, age 0.15, lifestyle 0.10, location 0.05).
- Clip to [0,1], convert to a percentage and rank candidates by percentage; keep top-K.
- Output includes reasons string (e.g., "text=0.8, age=1, life=0.6, loc=0.9, ~12km").

Feature	Actual OkCupid Algorithm	OkCupid-style Interpretation
Core data	Users answer thousands of multiple-choice questions.	User bios (SBERT embeddings) + structured profile fields (age, lifestyle, location).
Importance weighting	Each question can be marked as Irrelevant → Very Important → Mandatory; weights influence scoring.	Weights are fixed or tunable across components (text, age, lifestyle, location).
Scoring mechanism	Calculates *your satisfaction with them* and *their satisfaction with you*, then blends both into Match %.	Weighted blend of component scores (text similarity, age, lifestyle, location) → compatibility %.
Dealbreakers / hard gates	Orientation, age, and distance set as dealbreakers; no match if they don't align.	Same gates enforced: mutual orientation, mutual age rule, max distance, lifestyle optional strict filters.
Personalization	Highly personalized — two users may value different traits/questions differently.	Less personalized — all users scored using the same global weights unless manually changed.
Output	Match % (0–100) shown to both users.	Compat % (0–100) with detailed breakdown.

	Actual OkCupid Algorithm	My Algorithm (OkCupid-style)
Pros	Deep personalization: thousands of optional questions, highly granular. Importance weighting lets users say what really matters. Symmetric scoring: considers both how much *you* like them and how much *they* like you.	Transparent and easy to explain (weighted blend). Flexible: weights and filters can be tuned easily. Handles structured attributes (age, lifestyle, location) in a clean way. Outputs interpretable "reasons" string for matches.
Cons	Requires lots of user effort (answering many questions). Cold start problem: poor quality matches if few/no questions are answered. Algorithm is opaque to users; final Match % isn't directly interpretable. Heavy reliance on self-reported Q&A, which may be incomplete or inconsistent.	Simpler than real OkCupid; misses Q&A + importance weight logic. Linear weighting may underfit complex interactions. Relies heavily on text embedding quality and completeness of bios. Less personalization than true OkCupid (everyone scored by same formula).

Hinge

Hinge brands itself as the “dating app designed to be deleted,” focusing less on swipes and more on thoughtful matches. Instead of quick yes/no gestures, users build detailed profiles with prompts and photos that encourage conversation.

The app’s **matching concept** has two layers:

1. **Daily Feed** – Candidates shown based on preference filters (age, distance, religion, etc.), profile activity, and engagement signals.

2. **Most Compatible** – Hinge’s signature feature, powered by the Gale–Shapley stable matching algorithm. Each day, Hinge suggests one person it believes is the best reciprocal fit for you, balancing who you like with who is likely to like you back.

Dealbreakers like age, distance, and orientation are enforced strictly. Beyond that, the ranking emphasizes shared interests, prompt engagement, and reciprocal intent, so the goal is not just to maximize matches, but to surface people who are *mutually* most likely to connect.

My implementation mirrors this spirit by combining profile similarity and probabilistic “mutual like” estimates, but in a simplified form adapted to our dataset.

How it’s built (my Hinge-style algorithm)

Inputs:
- user_id, profile attributes (age, sex, orientation, location, bio_text).
- SBERT embeddings for bios (unit-norm).
- prob_mutual_like column (probability of reciprocal interest) from your model.

Candidate generation (hard gates):
- Pre-pool with top semantic neighbors (pool_k)
- Mutual orientation filter, Mutual age rule and Distance ≤ max_km.

Component scoring (soft signals):
- Mutual-like probability (prob_mutual_like) - primary factor, already bounded [0,1].
- Text similarity - cosine similarity of bio embeddings.
- Location score - linear decay: 1 − distance/max_km, clipped [0,1].

Blend & ranking:
- Weighted sum: final = w_like*prob_mutual_like + w_text*text_sim + w_loc*loc_score.
- Sort candidates by final score; keep top-K.
- Output includes [user_id, prob_mutual_like, text_sim, loc_score, distance_km, age, sex, orientation].

Differences from real Hinge:
- Real Hinge’s “Most Compatible” uses Gale-Shapley stable matching at the network level.
- This build is a user-centric ranked list, not a global stable pairing.
- Reciprocity is approximated with prob_mutual_like instead of exact two-sided matching.

Feature	Actual Hinge Algorithm	Hinge-style Interpretation
Core data	Profile prompts, likes, replies, activity history.	Bio embeddings + structured profile data + prob_mutual_like.
Matching model	Gale–Shapley stable matching → “Most Compatible.”	Weighted blend of prob_mutual_like, text similarity, location.
Dealbreakers / gates	Age, distance, orientation strictly enforced.	Same: orientation, age rule, max distance.
Personalization	Daily feed + one “Most Compatible” suggestion, based on network dynamics.	Ranked list per user, personalized by similarity and mutual-like probability.
Output	Ranked feed + single “Most Compatible.”	Ranked candidate list with blended scores.

	Actual Hinge Algorithm	My Hinge-style Algorithm
Pros	Stable matching ensures reciprocity across network. Uses full engagement graph (likes, replies, prompts). “Most Compatible” adds novelty and focus.	Transparent and tunable (weights adjustable). Incorporates prob_mutual_like as a clear reciprocity proxy. Simple, efficient to compute per user. Combines semantic bios + location + probability in one score.
Cons	Opaque to users; few details disclosed. Needs large active user base for stable matching to work. One “Most Compatible” per day can feel limiting.	No true stable matching; ignores network-wide pair balancing. Accuracy tied to prob_mutual_like estimates. Less personalization depth vs real Hinge prompts & engagement features.

Gale–Shapley Stable Matching (a.k.a. the "Stable Marriage Problem")
How it works:

- You have two sets of participants (say Set A and Set B).
- Each participant ranks the others in order of preference.
- The algorithm finds pairings so that no two people would rather be with each other than their current partners.
- That's what makes the result *stable*: nobody has an incentive to ditch their match for someone else.

How it works (classic version):

1. Everyone in Set A proposes to their top choice in Set B.
2. Each person in Set B temporarily accepts the "best" proposal they've received so far and rejects the rest.
3. Rejected people from Set A propose to their next choice.
4. Repeat until everyone is matched (or no valid matches remain).
5. The result is a stable set of pairings.

No pair exists where A prefers B over their assigned partner *and* B prefers A over their assigned partner. That's why it's called "stable."

Why my implementation didn't (and realistically couldn't) use Gale–Shapley "network-wide" stable matching:

1. Scope of your project
    - Goal was to compare algorithms side by side (OkCupid, Hinge, Tinder, etc.) using one dataset.
    - Running a global matching algorithm requires designing the whole system around pair formation across all users.
    - My setup is user-centric: "Given user X, rank their best matches." That's easier to compare across apps.
2. Data availability
    - Gale–Shapley needs preference lists from both sides. In dating apps, that means: not just how you rank candidates, but also how they rank you.
    - By making a ML model, I made proxies like prob_mutual_like, but not full ranked lists of everyone's preferences.
    - Without two-sided ranked data, a true stable matching can't be constructed.
3. Practicality for apps
    - Hinge can do stable matching because it has millions of active users and continuous engagement data to estimate preferences.

Bumble

Bumble's central idea is about flipping the traditional dynamic: in heterosexual matches, only women can send the first message. Every new match also comes with a **24-hour timer**, which encourages users to start conversations quickly. These product choices are as central to Bumble's "algorithm" as they shape which matches turn into conversations.

Unlike OkCupid or Hinge, Bumble does not disclose a detailed compatibility formula. Publicly, it is known that preferences (age, distance, orientation) act as filters, and profile visibility is influenced by activity and engagement. But the key differentiator is not the math of scoring profiles - it is the **conversation-first mechanics** (women initiate, 24-hour countdown, extensions).

How it's built (my Bumble-style algorithm)

Inputs

- df with: user_id, bio_text, age, sex, orientation, lat/lon, location, lifestyle (diet_c, drinks_c, smokes_c, drugs_c)
- SBERT embeddings for bio_text (unit-norm)
- Config: k, pool_k, max_km, allow_missing, strict flags for lifestyle, Bumble weights
- Imported utilities (from base notebook): get_matches, compute_component_scores, apply_weights

Candidate generation (hard gates)

- get_matches(user_id, pool_k, max_km, ...) →
    - Pre-pool with top semantic neighbors (pool_k)
    - Mutual orientation check
    - Mutual age rule ("half your age + 7")
    - Distance cap: distance_km ≤ max_km
    - Optional lifestyle hard filters (per strict flags)

Component scoring (soft signals)

- compute_component_scores(...) adds:
    - age_score ∈ {0,1}
    - lifestyle_score ∈ [0,1] (avg of diet/drinks/smokes/drugs; neutral bump if missing when allow_missing=True)
    - loc_score ∈ [0,1] = clip(1 − distance_km/max_km, 0, 1)

Blend & ranking

- apply_weights(..., weights={"text":0.75,"age":0.05,"life":0.15,"loc":0.05}) → bumble_score ∈ [0,1]
- Convert to % if needed, sort desc by bumble_score, keep top-k
- Output Columns: user_id, bumble_score, text_sim, age_score, lifestyle_score, loc_score, distance_km, location, age, sex, orientation
- Optional compact "reasons" string (text/age/life/loc breakdown, ~km)

API sections (stubs; not part of ranking)

- **Bumble "women-first openers"**: OpenAI API key placeholder; helper to generate pickup lines for top-K

Feature	Actual Bumble Algorithm	My Bumble-style Interpretation
Core dynamic	Women make the first move in heterosexual matches; 24-hour timer to start chatting; options to Extend/Rematch.	Ranking only; no built-in timers or gender-specific rules. (Notebook has API stubs where “make the first move” logic could be simulated, but not in scoring.)
Dealbreakers / gates	Age, orientation, and distance enforced strictly.	Same: mutual orientation, age rule, max distance; optional lifestyle hard filters.
Scoring signals	Internal ranking not disclosed; feed believed to be based on filters + engagement.	Weighted blend of: text similarity (0.75), age score (0.05), lifestyle score (0.15), location score (0.05).
Output	Swipe feed with time-limited chat initiation; product logic dominates UX.	Ranked top-K list with bumble_score, plus optional “reasons” string.

	Actual Bumble Algorithm	My Bumble-style Algorithm
Pros	Empowers women to initiate conversations. 24-hour urgency encourages engagement. Clear, distinctive UX compared to other apps.	Transparent, simple, and reproducible ranking. Tunable weights; emphasizes text similarity. Incorporates structured signals (age, lifestyle, location). Clean outputs with reasons. API stubs allow extension into Bumble-like UX.
Cons	Internals of ranking are opaque; hard to know why profiles appear. 24-hour limit may cause good matches to expire.	No implementation of women-first or 24-hour rules in ranking. Linear weighted blend may miss complex interactions. Depends heavily on bio text embeddings for quality. Less reflective of Bumble’s conversation-first identity.

Tinder

Tinder popularized the swipe-based interface, where users quickly accept or reject potential matches by swiping right or left. The app's core idea is about creating a fast, low-friction way to browse through profiles, with matches forming only when both people swipe right. This design makes Tinder highly engaging and addictive, emphasizing volume and immediacy over deep compatibility.

Behind the scenes, Tinder historically used an ELO-style rating system (similar to chess rankings) to prioritize which profiles appeared in your feed. This meant that if people others swiped right on you often, your score went up, and you'd be shown to more desirable users.

In 2019 Tinder announced it no longer relies directly on ELO, instead shifting toward a blend of activity, recency, and preference signals. The specifics remain undisclosed, but the platform is known to emphasize showing active users and maintaining balance in the match pool.

How it's built (my Tinder-style algorithm)

Inputs

- df with: user_id, bio_text, age, sex, orientation, lat/lon, location, lifestyle (diet_c, drinks_c, smokes_c, drugs_c)
- SBERT embeddings (unit-norm) for bio_text
- Config params: k, pool_k, max_km, allow_missing, lifestyle strict flags, **Tinder weights**

Candidate generation (hard gates)

- Pre-pool with top semantic neighbors (pool_k)
- Mutual orientation, Mutual age rule, Distance cap
- Optional lifestyle hard filters per strict flags

Component scoring (soft signals)

- age_score ∈ {0,1}
- lifestyle_score ∈ [0,1] (avg of diet/drinks/smokes/drugs; neutral bump if missing when allow_missing=True)
- loc_score ∈ [0,1] = clip(1 − distance_km/max_km, 0, 1)
- recency_norm ∈ [0,1] (recent activity / "nudge active users") — included for Tinder

Blend & ranking

- Weighted sum (Tinder defaults in your file look like):
    - w_text≈0.55–0.60, w_age≈0.10, w_life≈0.10, w_loc≈0.10, w_rec≈0.10–0.15
- Clip to [0,1] → convert to % as tinder_score and rank by tinder_score

Outputs

- Columns: user_id, tinder_score, text_sim, age_score, lifestyle_score, loc_score, recency_norm, distance_km, location, age, sex, orientation
- Optional compact "reasons" string (text/age/life/loc/recency, ~km)

Feature	Actual Tinder Algorithm	My Tinder-style Interpretation
Dealbreakers / gates	Age, orientation, distance.	Same: mutual orientation, age rule, max distance; optional lifestyle gates.
Signals	Not disclosed in detail; believed to include activity, recency, swipe patterns, and engagement balance.	Explicit, tunable weights: text ~0.55–0.60, age 0.10, lifestyle 0.10, location 0.10, recency 0.10–0.15.
Output	Endless swipe feed, with profile ordering guided by ranking signals.	Ranked top-K candidates with tinder_score, plus breakdown (text_sim, age_score, life_score, loc_score, recency_norm).
Elo	Used early on for global desirability score; no longer central.	Elo simulation scaffold included (mutual match = win, one-sided right = draw), but commented out and not applied to final scores.

	Actual Tinder Algorithm	Our Tinder-style Algorithm
Pros	Extremely simple, engaging swipe UX. Early Elo created natural desirability clusters. Current system surfaces active users, reducing dead profiles.	Transparent, reproducible scoring formula. Combines semantic bios + structured profile factors + recency. Flexible weights make it easy to tune/experiment. Elo simulation (if enabled) provides a principled “mutual desirability” model.
Cons	Ranking internals opaque to users; prone to speculation. Elo criticized for reinforcing popularity biases. Swipe-first UX often leads to shallow interactions.	Simplified linear blend; may miss complex dynamics Tinder exploits. Recency modeled only as a normalized variable, not full engagement graph. Elo block scaffold not integrated, so mutual desirability modeling unused. Less reflective of Tinder’s large-scale, network-wide balancing.

Elo used by Tinder
What it is and how it works:

Elo is a rating system that updates a player's skill score after each head-to-head encounter. Higher rating ⇒ higher predicted win chance.

Expected score (probability you "win"):

$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$ — logistic on the rating difference.

Update rule (per encounter):

$R'_A = R_A + K \cdot (S_A - E_A)$

where S_A is the actual result (1=win, 0.5=draw, 0=loss) and **K** controls volatility.

In dating, each encounter is a head-to-head between two profiles. A mutual match can be seen as a win, no mutual match as a loss. Update both users' ratings after each outcome. Higher-rated users tend to be shown to other higher-rated users, clustering people of similar "desirability."

Elo in my algorithm:
Setup & graph:

- Everyone has starting Elo score **= 1200**.
- Build a **candidate graph** via `get_matches` for each user and then keep **mutual pairs** (A↔B).
- Run encounters for **N_ROUNDS**, giving each user up to **ENCOUNTERS_PER_U** sampled pairs per round.

Swipe simulation (how outcomes are generated):

- Each user gets a random hidden **true_attractiveness** value (e.g., A=0.8, B=0.1, C=0.5, D=−0.2).
- Find the probability both candidates will swipe on each other via:

Probability that i swipes right on j:

$p(i \to j) = \sigma(B_0 + B_1 \cdot (a_i - a_j) + \text{noise})$ (a sigmoid on attractiveness difference).

Each swipe is scored, so no mutual: S = 0, one sided match (swipe but failed): S = 0.5 and mutual match: S = 1.

Example with Random 4 users: A, B, C, D:
Setup:
POOL_K_FOR_PAIRS = 2 → for each user, ask get_matches for up to 2 candidates.
N_ROUNDS = 2, ENCOUNTERS_PER_U = 2 → at most 2 encounters per user per round (≈4 total).
Everyone starts at Elo 1200.

Build the candidate graph (directed edges)
Pretend your get_matches returns: get_matches(A) → {B, C} get_matches(B) → {A, D}
get_matches(C) → {A, B} get_matches(D) → {B}

From this, we record directed edges (u → v): A→B, A→C B→A, B→D C→A, C→B D→B

Keeping only mutual pairs: (A,B) and (B,A) → mutual (A,C) and (C,A) → mutual
(B,D) and (D,B) → mutual

Make them undirected + unique (sort endpoints):{ {A,B}, {A,C}, {B,D} }
These are the pairs we'll sample encounters from during the rounds.

Set up users (state)
Everyone starts with: elo = 1200
true_attractiveness = random so A: 0.8, B: 0.1, C: 0.5, D: −0.2

Round 1 — sample encounters, simulate swipes, prepare Elo updates
Each user has a quota of ENCOUNTERS_PER_U = 2 this round. We shuffle the pair list {A–B, A–C, B–D} and walk it once, adding a pair only if both have quota left.

Now simulate swipes:
A vs B: p(A→B) = sigmoid(2*(0.8−0.1)) ≈ 0.80 → likely right
p(B→A) = sigmoid(2*(0.1−0.8)) ≈ 0.20 → likely left
Outcome (example): A right, B left ⇒ no match

A vs C: p(A→C) ≈ sigmoid(2*(0.8−0.5)) = sigmoid(0.6) ≈ 0.65 → likely right
p(C→A) ≈ sigmoid(2*(0.5−0.8)) = sigmoid(−0.6) ≈ 0.35 → could go either way
Outcome (example): both right ⇒ mutual match

B vs D: p(B→D) ≈ sigmoid(2*(0.1−(−0.2))) = sigmoid(0.6) ≈ 0.65 → likely right
p(D→B) ≈ sigmoid(−0.6) ≈ 0.35 → likely left
Outcome (example): B right, D left ⇒ no match

Rules for swiping: Right swipe = 0.5 Mutual match = 1 Left swipe = 0.0
The expected score is the Elo expectation vs the opponent:E(i) = 1 / (1 + 10^((elo_j - elo_i) / 400))
With both at 1200, E(i)=0.5.

The Elo update per user is: Δ = K * (actual − expected)
(K decays slightly as games increases; at the start it's ~24.)

Apply rules to Round 1:
A vs B (A right / B left; no match)
A: actual = 0.5, expected ≈ 0.5 → ΔA ≈ 0
B: actual = 0.0, expected ≈ 0.5 → ΔB ≈ −12 (with K≈24)

A vs C (mutual match)
A: actual = 1.0, expected ≈ 0.5 → ΔA ≈ +12
C: actual = 1.0, expected ≈ 0.5 → ΔC ≈ +12

B vs D (B right / D left; no match)
After B's first drop, B's Elo is ≈1188 vs D=1200, so E(B) ≈ 0.48:
B: actual = 0.5 → ΔB ≈ +0.4 (small recovery)
D: actual = 0.0, expected ≈ 0.52 → ΔD ≈ −12.5

End of Round 1 (approx):A ≈ 1212, B ≈ 1188.4, C ≈ 1212, D ≈ 1187.5

Round 2 — repeat sampling & swipes, more Elo updates
Quotas reset to 2 per user; we sample pairs again from {A–B, A–C, B–D} (subject to quotas).
New swipes done, new matches, and another set of Elo deltas.

Hence, after the rounds:
A: high Elo (did well, got at least one mutual vs decent opponents)
C: also high (mutual with A)
B: around/below 1200 (few/no mutuals, some right swipes unreciprocated)
D: below 1200 (few/no mutuals, faced stronger A/B/C)

Current Problems & Solutions (Data-Limited Context)

Problem	Impact	Potential Solution
Overreliance on text similarity	Candidates with “hiking + coffee” dominate top-K in Tinder/OkCupid/Bumble.	Add a diversity penalty: when selecting top-K, penalize candidates with high keyword overlap in bio_text. Encourage varied essay keywords (e.g., hiking, books, music).
Ghost / stale profiles	Profiles offline for weeks still rank high if text & lifestyle are strong.	Use last_online → compute recency_norm = exp(−days_offline / τ). Add it as a small boost (e.g., +0.1 weight).
Cold-start buried	New users (short bios, missing lifestyle info) get low scores and don’t surface.	Detect profiles with short bios or many null fields → give a temporary exposure boost. Decay the boost after they’ve appeared a few times.
Popularity loop / exposure inequality		
Pool sensitivity (pool_k too small/large)	Small pool misses candidates; large pool slows runtime.	Implement adaptive pool size: start at 100, expand by +100 if score spread <0.05.
Hard filters brittle	get_matches with strict diet/smokes wipes out pools.	Treat lifestyle mismatches as soft penalties (subtract 0.2 from lifestyle score) instead of removing candidate.
Explainability gaps	Current “reasons” strings are too basic.	Enrich reasons using available fields: "Similar age (22 vs 23) · Both vegetarian · Active 2 days ago · Mention hiking in bio."
No serendipity / wildcards	All top-K are predictable by score.	Inject 1–2 wildcard picks: mid-tier candidates who share 1 lifestyle trait (diet/drinks) or a unique essay keyword.
No fairness guardrails	Minority orientations/sexes underrepresented in final slates.	Track distribution of sex/orientation in pools vs. slates. If a group is underrepresented, apply a small boost to bring exposure closer to pool share.

After comparing Tinder, Bumble, OkCupid, and Hinge, several recurring issues became clear — from overreliance on text similarity to stale profiles, lack of diversity, and limited explainability. While each app has its own strengths, none of the algorithms fully addressed these pain points.

To explore how these shortcomings could be solved, I designed a custom matching algorithm called CUPID. This model builds on the lessons from existing apps while introducing new mechanisms for freshness, fairness, diversity, and serendipity.

Custom Dating App: CUPID

CUPID is a custom-built dating algorithm designed to address many of the shortcomings observed in apps like Tinder, Bumble, OkCupid, and Hinge. Instead of relying almost entirely on text similarity, CUPID blends multiple factors - semantic bios, lifestyle compatibility, location, and freshness - while adding unique elements like exposure bonuses for new users, penalties for overexposed profiles, diversity-aware ranking, and guaranteed wildcard picks.

This makes CUPID different from other apps: it not only surfaces strong compatibility matches, but also ensures variety, fairness, and the chance for unexpected connections, creating a more balanced and transparent experience

CUPID = Compatibility, Uniqueness, Popularity, Interest, and Diversity

Compatibility → core weighted blend (text, age, lifestyle, location)
Uniqueness → keyword-based diversity penalty so bios don't all look the same
Popularity → exposure balancing via UCB bonus + popularity penalty
Interest → shared-interest keyword boost (with diminishing returns)
Diversity → slate-building step + serendipity picks to guarantee variety

How it's built (Custom Dating Model ("CUPID"))
Imports & config

- Core knobs: CUPID_TOP_K = 100, CUPID_POOL_K = 500, CUPID_MAX_KM = 45
- Base weights: CUPID_W = {"text":0.60,"age":0.10,"life":0.15,"loc":0.15}
- Freshness: CUPID_G_FRESH = 0.10
- Explore/Popularity: CUPID_ALPHA_UCB = 0.15, CUPID_BETA_POP = 0.20
- Keywords bonus: CUPID_DELTA_KW = 0.03

Build pool (hard gates only)

- Calls get_matches(...) with top semantic neighbors (pool_k), orientation/age/location enabled.
- Lifestyle gates turned off here (to treat lifestyle softly later).

Base compatibility (component scores → weighted blend)

- compute_component_scores(...) on pool to add:
  - age_score ∈ {0,1}
  - lifestyle_score ∈ [0,1] (soft, averaged)
  - loc_score ∈ [0,1] = clip(1 - distance/max_km, 0, 1)
- Blend to base_score = 0.60*text + 0.10*age + 0.15*life + 0.15*loc.

Freshness bump (recency)

- Data prep: Parse df["last_online"] → last_online_dt (UTC), normalize to recency_norm ∈ [0,1].
- Join recency_norm into base
- Compute cupid_score_fresh = base_score + CUPID_G_FRESH * recency_norm (with CUPID_G_FRESH = 0.10).

Output columns: base_score, recency_norm, cupid_score_fresh.

Explore vs. popularity (synthetic exposure signals)

- Every profile gets a little “exploration bonus” if they haven’t shown up often.
    - UCB bonus: ucb_bonus = CUPID_ALPHA_UCB / sqrt(1 + schedule_strength)
- Very popular profiles get a small penalty so they don’t dominate the list.
    - Popularity penalty: pop_penalty = CUPID_BETA_POP * right_swipe_rate
- The result is a base score that mixes compatibility + freshness + these fairness adjustments.

Shared-interest keyword boost (log-scale)

- Cupid_add_keywords_log builds a small **dictionary of keywords** made from bios.
- For every keyword you share with the target user, a **bonus is added** to your score.
    - kw_bonus = CUPID_DELTA_KW * log1p(kw_shared) with CUPID_DELTA_KW = 0.03.
- The bonus is bigger if you share at least one keyword, but **diminishes** as you share many (so 10 common words isn’t 10x stronger than 1).
- This is add: cupid_score_kw = cupid_base + kw_bonus.

In-slate diversity (greedy with similarity penalty)

- When building the final top list, the algorithm doesn’t just take the highest scores blindly.
- It checks each new pick against what’s already been chosen.
- If the new candidate looks too similar (e.g., same keywords as many already in the list), their score is slightly reduced, letting someone different move up.
- This ensures the top list has **variety** rather than being full of near-duplicates

Add wildcards

- Wildcard picks: from the remainder, prefer candidates with:
    - recency_norm > 0.6 (fresh) and ≥ 1 shared keyword with the user.
- These are tagged as “suggested" and merged with the “ranked” list

Conversation starters (API only; not part of ranking)

- cupid_print_starters(user_id, k, model="gpt-4o-mini")
- What it prints per top-k match:
    - CUPID score & profile snippet + pickup line, playful question, date idea (LLM-generated).
    - Date ideas tailored to the key interests and location of the candidates

Example CUPID Output
User id: 120, NYC, 25F, straight

Rank	User id	Cupid final	Slot	Age	Sex	Location	Job	Shared Interests	Notes
1	305	0.871	matched	26	m	NYC	finance / consulting	hiking, concerts	Strong text similarity + lifestyle match + very fresh
2	412	0.823	matched	25	m	NYC	education /academia	books, travel	Balanced match, shared hobbies, similar age
3	509	0.791	matched	24	m	Brooklyn	tech / startup	hiking	Shared hobby, moderate freshness
4	611	0.682	suggested	27	m	NYC	arts / media	art, food	Wildcard: unique keywords, active recently
5	724	0.652	suggested	25	m	Jersey City	engineer	sports	Wildcard for variety, different interests but nearby

Problems Solved by CUPID

Problem	CUPID handling
Overreliance on text similarity	cupid_build_diverse_slate(...): greedy slate with keyword diversity penalty so “essay clones” don’t crowd top-K.
Ghost / stale profiles	freshness bump: add recency_norm into score (cupid_score_fresh = base + 0.10·recency_norm).
Cold-start buried	cupid_add_explore_pop_synth(...): UCB bonus $\alpha/\sqrt{(1+\text{schedule\_strength})}$ gives exposure to low-seen profiles. (Uses synthesized exposure since no logs.)
Popularity loop / exposure inequality	popularity penalty $\beta \cdot$right_swipe_rate (synthetic) subtracts from score to prevent runaway winners.
Hard filters brittle (diet/smokes, etc.)	lifestyle gates OFF in pool; later treated softly in lifestyle_score.
Explainability gaps	produce kw_shared, kw_bonus, slot (serendipity). Easy to surface as reasons (e.g., “3 shared interests • fresh this week • lifestyle match”).
No serendipity / wildcards	cupid_finalize_table_simple(..., serendipity_n=…): guaranteed serendipity picks.

Conclusion

Through this project, I explored how four of the most widely used dating apps: Tinder, Bumble, OkCupid, and Hinge shape the way people meet by ranking and surfacing potential matches. Each platform uses a different approach, from OkCupid's compatibility percentages to Hinge's "Most Compatible" feature, but they also share recurring weaknesses: overreliance on text similarity, stale or inactive profiles ranking highly, and a lack of transparency or diversity in their results.

To address these gaps, I developed a custom algorithm called **CUPID**, which combines compatibility scoring with freshness, fairness, diversity, and serendipity. By introducing exploration bonuses, popularity penalties, shared-interest boosts, and wildcard suggestions, CUPID creates a slate that is not only accurate but also more balanced and engaging. While some challenges remain such as modeling true conversational outcomes or ensuring fairness across all groups, this project demonstrates how data science can reveal the strengths and shortcomings of existing systems and inspire new approaches for more transparent, user-friendly matchmaking.

Future Work

- **Conversation outcomes:** Incorporate reply rate or message engagement (not just likes) to better optimize for meaningful matches.

- **Fairness guardrails:** Add group-level exposure checks (e.g., orientation, ethnicity) to ensure equitable visibility across smaller demographics.

- **Adaptive pools:** Dynamically adjust candidate pool size to balance runtime with diversity, ensuring users always see a varied set of matches.

- **Hybrid features:** Experiment with blending Elo-style mutual desirability signals with CUPID's freshness/diversity logic for stronger two-sided matching.