1) Fix data errors & harden the schema (prevent silent mistakes)

Issues:

- `expected_categorya` (typo) will silently break learning.
- `expected_mood`/`expected_energy` are free-text prone to drift.
- `Date` in `UserFeedback.timestamp` is unsafe to (de)serialize consistently.

What to do

- Lock types with strict unions, add an `id`, `source`, and `version`.
- Validate every example at load time (fail fast).
- Store timestamps as ISO strings.

```
// types.ts
export const Categories =
['Growth','Challenge','Achievement','Planning','Learning','Research'] as
const;
export type Category = typeof Categories[number];

export const Energies = ['high','medium','low'] as const;
export type Energy = typeof Energies[number];

export interface TrainingExample {
  id: string;                        // e.g., "GROWTH_001"
  version: number;                   // bump when edited
  text: string;
  expected_category: Category;
  expected_mood: string;             // see §4 for normalization
  expected_energy: Energy;
  confidence_range: [number, number]; // 0-100
  business_context: string;
  source?: 'handwritten' | 'synthetic' | 'user_correction';
}

export interface UserFeedback {
  entry_id: string;
  original_category: Category;
  corrected_category: Category;
  original_mood: string;
  corrected_mood: string;
  text_content: string;
  user_id: string;
  timestamp_iso: string;             // ISO8601
  feedback_type: 'category_correction' | 'mood_correction' | 'both';
}
```

Runtime validation (fail fast)

```
export function validateDataset(ds: TrainingExample[]): void {
  const seen = new Set<string>();
  for (const ex of ds) {
    if (!ex.id) throw new Error(`Missing id for: ${ex.text.slice(0,60)}`);
```

```
    if (seen.has(ex.id)) throw new Error(`Duplicate id: ${ex.id}`);
    seen.add(ex.id);

    if (!Categories.includes(ex.expected_category))
      throw new Error(`Bad category ${ex.expected_category} in ${ex.id}`);
    if (!Energies.includes(ex.expected_energy))
      throw new Error(`Bad energy ${ex.expected_energy} in ${ex.id}`);
    const [lo, hi] = ex.confidence_range;
    if (!(lo >= 0 && hi <= 100 && lo <= hi))
      throw new Error(`Bad confidence_range in ${ex.id}`);
  }
}
```

Fix your dataset

- Correct `expected_categorya` → `expected_category`.
- Give each example a unique `id` and add `version: 1`.
- You also have the long "What an incredible week!" **duplicated** in Growth (once as short list and once under "Extended Growth scenarios"). Keep one; give the other a different wording or delete to avoid overweighting that pattern.

2) Upgrade similarity: TF-IDF + bigrams + business keywords

Your current cosine over raw word counts underweights rare business terms and misses phrasal cues. Use a simple TF-IDF with unigrams + bigrams and a weighted domain keyword boost.

```
// similarity.ts
type Counts = Map<string, number>;

function tokenize(text: string): string[] {
  return text.toLowerCase()
    .replace(/[^\p{L}\p{N}\s]/gu, ' ')
    .split(/\s+/)
    .filter(w => w.length > 2 && !StopWords.has(w));
}

const StopWords = new Set([

'the','and','for','with','that','this','have','has','but','are','was','were
',
  'you','your','our','from','into','about','not','too','very','just','also'
]);

function ngrams(tokens: string[], n=2): string[] {
  const grams: string[] = [];
  for (let i=0;i<tokens.length;i++){
    grams.push(tokens[i]); // unigrams
    if (n >= 2 && i < tokens.length-1) grams.push(tokens[i]+'
'+tokens[i+1]); // bigrams
  }
  return grams;
}
```

```
const DOMAIN_TERMS = new Map<string, number>([
  ['cash flow', 1.5], ['revenue', 1.3], ['gross margin', 1.4],
  ['customer', 1.2], ['client', 1.2], ['churn', 1.5], ['mrr', 1.6],
  ['launch', 1.2], ['hiring', 1.3], ['funding', 1.4], ['kpi', 1.4],
  ['roadmap', 1.3], ['onboarding', 1.3], ['retention', 1.5],
  ['partnership', 1.3], ['competitor', 1.3], ['expansion', 1.2]
]);

export class TFIDF {
  private df = new Map<string, number>();
  private docs: string[][] = [];
  private N = 0;

  addDocument(text: string) {
    const terms = new Set(ngrams(tokenize(text)));
    this.docs.push([...terms]);
    for (const t of terms) this.df.set(t, (this.df.get(t) || 0) + 1);
    this.N++;
  }

  vectorize(text: string): Counts {
    const tokens = ngrams(tokenize(text));
    const tf = new Map<string, number>();
    for (const t of tokens) tf.set(t, (tf.get(t)||0) + 1);
    const vec = new Map<string, number>();
    for (const [t, f] of tf) {
      const idf = Math.log((this.N + 1) / ((this.df.get(t) || 0) + 1)) + 1;
      const domainBoost = DOMAIN_TERMS.get(t) || 1;
      vec.set(t, f * idf * domainBoost);
    }
    return vec;
  }

  static cosine(a: Counts, b: Counts): number {
    let dot=0, na=0, nb=0;
    const keys = new Set([...a.keys(), ...b.keys()]);
    for (const k of keys) {
      const va = a.get(k) || 0, vb = b.get(k) || 0;
      dot += va*vb; na += va*va; nb += vb*vb;
    }
    return (na===0 || nb===0) ? 0 : dot / (Math.sqrt(na)*Math.sqrt(nb));
  }
}
```

Use it in your validator

```
export class AITrainingValidator {
  private static tfidf: TFIDF | null = null;
  private static all: TrainingExample[] | null = null;

  private static getAll(): TrainingExample[] {
    if (!this.all) {
      const base = [...BUSINESS_JOURNAL_TRAINING_DATA,
...ENHANCED_BUSINESS_TRAINING_DATA];
      validateDataset(base);
      this.all = base;
      this.tfidf = new TFIDF();
      for (const ex of base) this.tfidf!.addDocument(ex.text);
    }
    return this.all!;
```

```
  }

  static getBestTrainingMatch(text: string): TrainingExample | null {
    const all = this.getAll();
    const qv = this.tfidf!.vectorize(text);
    let best: TrainingExample | null = null;
    let bestScore = 0;
    for (const ex of all) {
      const dv = this.tfidf!.vectorize(ex.text);
      const s = TFIDF.cosine(qv, dv);
      if (s > bestScore) { bestScore = s; best = ex; }
    }
    return (bestScore >= 0.22) ? best : null; // tuned threshold
  }

  static validateCategoryAccuracy(text: string, predictedCategory: string):
number {
    const all = this.getAll();
    const qv = this.tfidf!.vectorize(text);
    const sims = all
      .map(ex => ({ ex, s: TFIDF.cosine(qv, this.tfidf!.vectorize(ex.text))
}))
      .filter(x => x.s >= 0.22)
      .sort((a,b) => b.s - a.s)
      .slice(0, 10); // top-k neighborhood
    if (sims.length === 0) return 0.5;
    const correct = sims.filter(x => x.ex.expected_category.toLowerCase()
=== predictedCategory.toLowerCase()).length;
    return correct / sims.length;
  }
}
```

3) Add negation & contrastive cue handling (huge accuracy win)

Phrases like “not happy”, “but”, “however” flip or dilute sentiment and category cues.

```
const NEGATORS = ['not','no','never','hardly','barely','without'];
const CONTRAST = ['but','however','though','yet','although'];

export function negationAwareScore(tokens: string[], sentimentLex:
Map<string, number>) {
  let score = 0;
  let negateWindow = 0; // negate next 3 tokens
  for (let i=0;i<tokens.length;i++){
    const t = tokens[i];
    if (NEGATORS.includes(t)) { negateWindow = 3; continue; }
    let w = sentimentLex.get(t) || 0;
    if (negateWindow > 0) { w = -w * 0.9; negateWindow--; }
    score += w;
  }
  return score;
}

export function contrastPenalty(text: string): number {
  const lower = text.toLowerCase();
  let hits = 0;
```

```
  for (const c of CONTRAST) if (lower.includes(` ${c} `)) hits++;
  return Math.min(hits * 0.05, 0.15); // reduce confidence up to 15%
}
```

Use the contrast penalty to **shrink** confidence and avoid over-certainty when the user mixes good/bad in one entry.

4) Normalize mood + map intensity → energy

Right now “Determined”, “Confident”, “Proud” etc. can vary. Normalize to a **controlled mood set** but retain the original text. Also infer energy from intensifiers.

```
const MoodMap = new Map<string, 'Positive'|'Negative'|'Neutral'>([
  ['excited','Positive'], ['confident','Positive'], ['proud','Positive'],
  ['optimistic','Positive'], ['grateful','Positive'],
['relieved','Positive'],
  ['stressed','Negative'], ['worried','Negative'],
['overwhelmed','Negative'],
  ['frustrated','Negative'], ['uncertain','Negative'],
['guilty','Negative'],
  ['reflective','Neutral'], ['analytical','Neutral'],
['thoughtful','Neutral'],
  ['determined','Neutral'], ['contemplative','Neutral']
]);

const Intensifiers = new
Set(['very','extremely','incredibly','so','really','totally','absolutely','
highly']);
const Dampeners   = new Set(['slightly','somewhat','a
bit','kinda','fairly','moderately']);

export function normalizeMood(raw: string): {norm: string, polarity:
'Positive'|'Negative'|'Neutral'} {
  const key = raw.trim().toLowerCase();
  const polarity = MoodMap.get(key) || 'Neutral';
  // Keep original label but use polarity downstream
  return { norm: raw.trim(), polarity };
}

export function inferEnergy(text: string): 'high'|'medium'|'low' {
  const t = text.toLowerCase();
  const exclam = (t.match(/!/g) || []).length;
  const ints = [...Intensifiers].reduce((a,k)=>a+(t.includes(k)?1:0),0);
  const dams = [...Dampeners].reduce((a,k)=>a+(t.includes(k)?1:0),0);
  const score = exclam*0.6 + ints*0.5 - dams*0.4;
  return score >= 0.8 ? 'high' : score <= -0.2 ? 'low' : 'medium';
}
```

Training data tip: add `normalized_mood_polarity` to each example (derived at build time) instead of relying on ad-hoc mood words at inference.

5) Make the UserLearningSystem production-safe

- Don’t write to `localStorage` in non-browser contexts.
- Cap history to prevent unbounded growth.
- Prefer most-recent **and** most-similar correction (not just last).

```
export class UserLearningSystem {
  private static userCorrections: UserFeedback[] = [];
  private static MAX = 1000;

  static recordUserFeedback(feedback: UserFeedback): void {
    this.userCorrections.push(feedback);
    if (this.userCorrections.length > this.MAX)
this.userCorrections.shift();
    try {
      if (typeof window !== 'undefined' && window.localStorage) {
        localStorage.setItem('ai_user_corrections',
JSON.stringify(this.userCorrections));
      }
    } catch { /* ignore */ }
  }

  static loadUserFeedback(): void {
    try {
      if (typeof window !== 'undefined' && window.localStorage) {
        const stored = localStorage.getItem('ai_user_corrections');
        if (stored) this.userCorrections = JSON.parse(stored);
      }
    } catch { /* ignore */ }
  }

  static getUserPatterns(userId: string): UserFeedback[] {
    return this.userCorrections.filter(f => f.user_id === userId);
  }

  static adjustPredictionBasedOnHistory(
    text: string,
    predicted: { primary_mood: string; business_category: string;
confidence: number; },
    userId: string
  ) {
    const patterns = this.getUserPatterns(userId);
    if (!patterns.length) return predicted;

    // most similar among the most recent N
    const recent = patterns.slice(-200);
    let best: { fb: UserFeedback; sim: number } | null = null;
    for (const fb of recent) {
      const sim =
AITrainingValidator.calculateSimilarity(text.toLowerCase(),
fb.text_content.toLowerCase());
      if (!best || sim > best.sim) best = { fb, sim };
    }
    if (!best || best.sim < 0.40) return predicted;

    const adj = { ...predicted };
```

```
    if (best.fb.feedback_type !== 'mood_correction') {
      adj.business_category = best.fb.corrected_category.toLowerCase();
    }
    if (best.fb.feedback_type !== 'category_correction' &&
best.fb.corrected_mood) {
      adj.primary_mood = best.fb.corrected_mood;
    }

    // Confidence shaping: similarity + contrast penalty
    const penalty = contrastPenalty(text);
    adj.confidence = Math.min(95, Math.max(40, predicted.confidence +
Math.round(best.sim*20) - Math.round(penalty*100)));
    (adj as any).user_learned = true;
    return adj;
  }
}
```

6) Add a lightweight rule layer (cheap, big lift)

Before MLish similarity kicks in, capture **high-precision** patterns (e.g., cash-flow pain → Challenge/low energy; “hired/signed/launched” → Growth/Achievement/high).

```
type Rule = { test: (t:string)=>boolean; category: Category; energy?:
Energy; moodPolarity?: 'Positive'|'Negative'|'Neutral'; };
const Rules: Rule[] = [
  { test: t => /\bcash\s*flow\b/.test(t) || /\bpayroll\b/.test(t),
category: 'Challenge', energy: 'low', moodPolarity: 'Negative' },
  { test: t => /\b(churn|downtime|outage|bug|incident)\b/.test(t),
category: 'Challenge' },
  { test: t => /\b(launched?|release(d)?)\b/.test(t), category:
'Achievement', energy: 'high', moodPolarity: 'Positive' },
  { test: t => /\b(hired?|recruit(ing)?|offer accepted)\b/.test(t),
category: 'Growth', energy: 'high' },
  { test: t => /\bplan(ning)?\b|\broadmap\b|\bbudget(s|ing)?\b/.test(t),
category: 'Planning' },
  { test: t =>
/\b(user|customer)\s+(interview|feedback|research|study)\b/.test(t),
category: 'Research' },
];

export function ruleFirstPass(text: string): Partial<TrainingExample> |
null {
  const t = text.toLowerCase();
  for (const r of Rules) {
    if (r.test(t)) {
      return {
        expected_category: r.category,
        expected_energy: r.energy || inferEnergy(text),
        expected_mood: r.moodPolarity ||
normalizeMood('Analytical').polarity
      } as any;
    }
  }
  return null;
}
```

Use this as a quick, precise pre-classifier; fall back to TF-IDF neighborhood if no rule hits.

7) Confidence calibration & tie-breaking

- Use neighborhood agreement to set confidence.
- Penalize contrastive texts and short texts.
- If top-2 categories within 0.03 similarity, lower confidence and surface both.

```
export function calibratedPrediction(text: string) {
  const rule = ruleFirstPass(text);
  const best = AITrainingValidator.getBestTrainingMatch(text);
  if (!best && !rule) return { business_category: 'Learning', confidence:
45, rationale: 'fallback' };

  const candidates: Array<{cat: Category; score: number}> = [];
  if (best) candidates.push({ cat: best.expected_category, score: 1.0 });
  if (rule) candidates.push({ cat: rule.expected_category as Category,
score: 0.92 });

  // Aggregate (simple voting with weights)
  const byCat = new Map<Category, number>();
  for (const c of candidates) byCat.set(c.cat, (byCat.get(c.cat)||0) +
c.score);
  const sorted = [...byCat.entries()].sort((a,b)=>b[1]-a[1]);

  let confidence = 60 + Math.min(30, Math.round((sorted[0][1]-
(sorted[1]?.[1]||0))*25));
  confidence -= Math.round(contrastPenalty(text)*100);
  if (text.length < 60) confidence -= 8;

  return {
    business_category: sorted[0][0],
    confidence: Math.max(40, Math.min(95, confidence)),
    alternatives: sorted.slice(1,3).map(([cat,score])=>({cat, score:
Math.round(score*100)/100}))
  };
}
```

8) Balance the dataset & cover edge cases

- Ensure each category has **comparable counts**. You're heavier on "Challenge" and "Growth" (and repeated patterns like "cash flow" and "signed clients"). Add 10–15 more **Planning** and **Research** examples (e.g., budgeting, roadmaps, interviews, cohort analysis, pricing tests).
- Add **ambiguous/mixed** entries (e.g., "numbers up but churn rising") to train the contrast penalty and blended predictions.
- Add **short entries** ("Tired." "Massive win.") to calibrate low-confidence behavior.

9) Evaluation harness (know it's better, not just different)

Write a tiny test runner to measure accuracy and confusion:

```
type Pred = { category: Category; };
function predictCategory(text: string): Pred {
  const p = calibratedPrediction(text);
  return { category: p.business_category as Category };
}

export function evaluate(ds: TrainingExample[]) {
  let correct = 0;
  const cm = new Map<string, number>(); // "true->pred" -> count
  for (const ex of ds) {
    const pred = predictCategory(ex.text);
    if (pred.category === ex.expected_category) correct++;
    const key = `${ex.expected_category}->${pred.category}`;
    cm.set(key, (cm.get(key)||0) + 1);
  }
  const acc = correct / ds.length;
  return { accuracy: acc, confusion: cm };
}
```

Run this before/after the TF-IDF+rules+negation changes. Track that **Planning**/**Research** misclassifications drop (they usually get confused with Growth/Challenge without the rules).

10) Practical guardrails & hygiene

- **Environment-safe storage**: Wrap all `localStorage` calls.
- **Determinism**: If you later add randomness (e.g., sampling neighbors), seed it for repeatability.
- **Version your data** and bump `version` when any example text changes; keep an audit trail.

TL;DR—What changes drive the biggest accuracy gains?

1. **TF-IDF with bigrams + domain term boost** (replaces plain counts)
2. **Rule-first pass** for high-precision business cues (cash flow, launch, churn, hiring, planning, research)
3. **Negation & contrast handling** to avoid confident but wrong calls on mixed entries
4. **Mood normalization + energy inference** to standardize targets and reduce label drift
5. **Strict schema + runtime validation** to catch typos/duplication (like `expected_categorya` and repeated long Growth sample)
6. **Calibration** using neighborhood agreement & penalties for short/contrastive texts
7. **Balanced dataset + evaluation harness** so you can prove improvements quantitatively

If you want, I can bundle these into a single drop-in `analysis/` module and a script that runs the evaluation and prints before/after metrics.