ADR-003-002-linear-weighted-average

ADR-003-002: Linear Weighted Average vs Other Ensemble Methods

Status: Accepted **Date**: 2025-11-16 **Deciders**: Research Team **Related**: SPEC-PPP-003 (Interaction Scoring & Weighted Consensus)

Context

Once we have technical and interaction scores for each agent, we need a method to combine them into a final score for agent selection. Multiple ensemble combination methods exist in machine learning literature.

Question: Which method should we use to combine technical + interaction scores?

Options considered: 1. **Linear weighted average** - Arithmetic combination with fixed weights 2. **Geometric mean** - Multiplicative combination 3. **Stacking** - Meta-learner combines scores 4. **Rank-based** - Convert scores to ranks, combine ranks 5. **Non-linear** - Custom function (e.g., sigmoid, polynomial)

Decision

We will use **linear weighted average** with the formula:

```
final_score = w_technical × technical_score + w_interaction ×
interaction_score

where:
  w_technical + w_interaction = 1.0
  Default: w_technical = 0.7, w_interaction = 0.3
```

Rationale

1. Simplicity & Interpretability

Linear weighted average: - Easy to explain to users - Transparent: Can see exactly how scores combine - Predictable: Small changes in input → proportional changes in output

Example:

```
Agent A:
  technical = 0.80
  interaction = 0.10
  final = 0.7 × 0.80 + 0.3 × 0.10 = 0.56 + 0.03 = 0.59
```

User understanding: “70% from code quality, 30% from UX” (immediately clear)

---

2. ML Ensemble Literature Standard

Ensemble averaging (Machine Learning literature): > “The simplest and most common ensemble method is weighted averaging, where predictions from each model are combined using fixed weights that sum to 1.”

Evidence: - Random Forest: Averages decision trees - Gradient Boosting: Weighted sum of weak learners - Neural Network Ensembles: Average predictions

Why linear works: - Models (agents) have uncorrelated errors → averaging reduces variance - Weights allow prioritizing better models (70% technical > 30% interaction)

Source: “Ensemble Methods in Machine Learning” (Dietterich, 2000)

---

3. Comparison to Alternatives

Method	Formula	Pros	Cons	Ver
Linear	$w_1 x_1 + w_2 x_2$	Simple, interpretable	Assumes independence	✅ **ACC**
Geometric Mean	$(x_1{}^{w_1} \cdot x_2{}^{w_2})^{1/(w_1+w_2)}$	Penalizes low scores	Hard to interpret	✗ R
Stacking	$f(x_1, x_2)$ (learned)	Optimal (in theory)	Needs training data	✗ R
Rank-Based	Combine ranks	Robust to outliers	Loses magnitude info	✗ R
Non-Linear	$\sigma(w_1 x_1 + w_2 x_2)$	Flexible	Arbitrary, complex	✗ R

---

4. Why NOT Geometric Mean

Formula:

```
final_score = (technical^0.7 × interaction^0.3)^(1/1.0)
            = technical^0.7 × interaction^0.3
```

Example:

```
Agent A:
  technical = 0.80
  interaction = 0.10
```

```
final = (0.80^0.7) × (0.10^0.3)
      = 0.833 × 0.501
      = 0.417
```

Problems:

1. **Hard to interpret**: Users don't understand exponentiation
2. **Penalizes low scores heavily**: interaction=0.10 → reduces final to 0.417 (vs 0.59 with linear)
3. **Interaction negative scores**: Geometric mean undefined for negative numbers!
   - interaction can be negative (e.g., -0.45 from high-effort question)
   - Geometric mean: 0.80^0.7 × (-0.45)^0.3 = undefined (complex number)

Fatal flaw: Cannot handle negative interaction scores.

Verdict: ✖ Reject - Incompatible with PPP formula (allows negative R_Proact, R_Pers)

---

5. Why NOT Stacking (Meta-Learner)

Approach: Train a model to combine technical + interaction scores

Example:

```
# Train meta-learner
X = [[tech1, interact1], [tech2, interact2], ...]  # Features
y = [is_best1, is_best2, ...]  # Labels (which agent was best)

meta_model = LogisticRegression()
meta_model.fit(X, y)

# Predict best agent
final_score = meta_model.predict_proba([technical, interaction])[1]
```

Pros: - Theoretically optimal (learns best combination from data) - Can capture non-linear interactions

Cons: - Requires labeled training data (100+ examples of "which agent was best") - Black box (users can't understand why agent selected) - Overfitting risk (may not generalize) - Complex implementation (scikit-learn dependency)

Decision: Defer to Phase 3 research project (not Phase 1 default)

Verdict: ✖ Reject for Phase 1 - Too complex, needs data

---

6. Why NOT Rank-Based

Approach: Convert scores to ranks, combine ranks

Example:

```
3 agents:
  Agent A: technical=0.95 (rank 1), interaction=0.05 (rank 2)
  Agent B: technical=0.85 (rank 2), interaction=0.10 (rank 1)
  Agent C: technical=0.75 (rank 3), interaction=-0.20 (rank 3)
```

```
Combine ranks:
  Agent A: 0.7 × 1 + 0.3 × 2 = 1.3 ← Winner (lowest rank)
  Agent B: 0.7 × 2 + 0.3 × 1 = 1.7
  Agent C: 0.7 × 3 + 0.3 × 3 = 3.0
```

Pros: - Robust to outliers (extreme scores don't dominate) - Works with any score range

Cons: - Loses magnitude information (0.95 vs 0.85 both become "ranks") - Ties problematic (what if 2 agents have same technical score?) - Less intuitive than raw scores

Example problem:

```
Agent A: technical=0.95, interaction=0.10
Agent B: technical=0.94, interaction=0.10

Linear: A wins (0.695 vs 0.688)
Rank:   Tie (both rank 1 technical, rank 1 interaction)
```

Verdict: ✖ Reject - Loses important information (magnitude of difference)

---

7. Why NOT Non-Linear Functions

Examples: - Sigmoid: $\sigma(w_1 x_1 + w_2 x_2) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2)}}$ - Polynomial: $(w_1 x_1)^2 + (w_2 x_2)^2$ - Min/Max: $\min(x_1, x_2)$ or $\max(x_1, x_2)$

Problems: 1. **Arbitrary**: Why sigmoid vs polynomial vs…? No principled choice 2. **Hard to tune**: More hyperparameters (e.g., sigmoid temperature) 3. **Non-interpretable**: Users can't predict output from inputs 4. **Overkill**: Linear works well, no evidence non-linear needed

Verdict: ✖ Reject - Unnecessary complexity without clear benefit

---

Consequences

Positive

1. ✅ **Standard ML technique**: Weighted averaging proven effective
2. ✅ **Interpretable**: Users understand how scores combine
3. ✅ **Simple**: Single line of code
4. ✅ **Fast**: Arithmetic operations (<1μs)
5. ✅ **Handles negatives**: Works with negative interaction scores
6. ✅ **No training needed**: No labeled data required

Negative

1. ⚠ **Assumes independence**: Treats technical and interaction as uncorrelated
   - Reality: Might be correlated (e.g., agents that ask fewer questions might write worse code)

    - Impact: Low - Empirical validation can measure correlation
    - Mitigation: If high correlation found, Phase 3 can explore non-linear
2. ⚠ **Sub-optimal**: Meta-learner (stacking) theoretically better
    - Reality: Needs 100+ labeled examples to train
    - Impact: Low - Linear is 90% as good with zero training
    - Mitigation: Phase 3 can add stacking as advanced option

Neutral

1. 📊 **Convex combination**: Weights sum to 1.0 (constraint)
    - Good: Ensures final_score in reasonable range
    - Limitation: Can't amplify both dimensions simultaneously

---

Validation

Mathematical Properties

Property 1: Boundedness

If $x_1, x_2 \in [a, b]$ and $w_1 + w_2 = 1$, then:

```
w_1 x_1 + w_2 x_2 ∈ [a, b]
```

Application to PPP: - technical ∈ [0, 1] - interaction ∈ [-0.5, 0.1] (approximate range) - final_score ∈ [-0.15, 0.73] (0.7 × 0 + 0.3 × (-0.5), 0.7 × 1 + 0.3 × 0.1)

Implication: Final scores are bounded, predictable.

Property 2: Monotonicity

If x_1 increases and x_2 constant, then final_score increases:

```
∂(w_1 x_1 + w_2 x_2) / ∂x_1 = w_1 > 0
```

Implication: Better technical score → higher final score (as expected).

Property 3: Linearity

Doubling both weights has same effect as doubling final_score:

```
2(w_1 x_1 + w_2 x_2) = (2w_1) x_1 + (2w_2) x_2
```

Implication: Scaling weights proportionally doesn't change relative ranking.

Empirical Validation

Test Case 1: Linear combination matches manual calculation

```
let technical = 0.85;
let interaction = 0.10;
let final_score = 0.7 * technical + 0.3 * interaction;
```

```
assert_eq!(final_score, 0.625);  // 0.595 + 0.030
```

Test Case 2: Negative interaction handled correctly

```
let technical = 0.95;
let interaction = -0.45;
let final_score = 0.7 * technical + 0.3 * interaction;

assert_eq!(final_score, 0.530);  // 0.665 + (-0.135)
```

Test Case 3: Ranking preserved

```
// If A > B on both dimensions, A wins
let score_a = 0.7 * 0.95 + 0.3 * 0.10;  // 0.695
let score_b = 0.7 * 0.85 + 0.3 * 0.05;  // 0.610

assert!(score_a > score_b);
```

Alternative Considered: Pareto Optimization

Approach: Multi-objective optimization (no single score)

Concept: - Don't combine scores at all - Report Pareto frontier: Agents where improving one dimension requires sacrificing the other

Example:

```
Agent A: technical=0.95, interaction=-0.20
Agent B: technical=0.85, interaction=0.10
Agent C: technical=0.75, interaction=0.05

Pareto frontier: {A, B}
- A dominates C (better technical, similar interaction)
- B dominates C (similar technical, better interaction)
- A vs B: Trade-off (A better technical, B better interaction)
```

User choice: Present both A and B, let user pick.

Pros: - No arbitrary weighting - Shows trade-offs explicitly

Cons: - Doesn't select single "best" agent (defeats purpose of consensus) - Requires user input every time (slow) - Complex UI (show Pareto frontier in TUI?)

Verdict: ❌ Reject - PPP requires automatic selection, not manual choice.

Future Enhancements (Phase 3)

Non-Linear Exploration

If empirical data shows linear is sub-optimal:

Option 1: Learned weights

```rust
// Instead of fixed 0.7/0.3, learn per-user
let w_tech = learn_weight_from_history(user_id);
let w_interact = 1.0 - w_tech;
```

Option 2: Context-dependent weights

```rust
// Different weights based on task properties
let weights = if task_is_critical {
    (0.8, 0.2)  // Favor correctness
} else {
    (0.6, 0.4)  // Balance UX
};
```

Option 3: Meta-learner

```rust
// Train on historical consensus decisions
let meta_model = train_stacking_model(historical_data);
let final_score = meta_model.predict([technical, interaction]);
```

Implementation

Rust Code:

```rust
pub fn calculate_final_score(
    technical: f32,
    interaction: f32,
    weights: (f32, f32),
) -> f32 {
    let (w_tech, w_interact) = weights;

    // Validate weights sum to 1.0
    debug_assert!((w_tech + w_interact - 1.0).abs() < 0.001);

    // Linear weighted average
    (w_tech * technical) + (w_interact * interaction)
}
```

Performance: ~1ns (2 multiplications + 1 addition)

References

1. Dietterich, T. (2000). “Ensemble Methods in Machine Learning” - Weighted averaging standard technique
2. “Ensemble averaging” (Wikipedia) - Mathematical properties of weighted average
3. “Optimizing Ensemble Weights” (arXiv:1908.05287) - Linear weighting effective in practice
4. Pareto optimization (Multi-objective optimization literature) - Alternative to single score

---

Decision Matrix

Method	Interpretability	Performance	Handles Negatives	Training Data

Linear	✅ **Excellent**	✅ **Fast**	✅ **Yes**	✅ **None needed**
Geometric	❌ Poor	✅ Fast	❌ **No**	✅ None needed
Stacking	❌ Black box	⚠️ Medium	✅ Yes	❌ **Needs 100+ labels**
Rank-Based	⚠️ Medium	✅ Fast	✅ Yes	✅ None needed
Non-Linear	❌ Poor	✅ Fast	⚠️ Depends	⚠️ Maybe

Winner: Linear - Best across all criteria for Phase 1.