Meta-Ensemble Horse Race Prediction Architecture

Executive Summary

This document outlines the upgrade from a single LightGBM Ranker to a sophisticated meta-ensemble architecture combining four diverse ranker models (LightGBM, XGBoost, CatBoost, Neural Network) with a logistic regression meta-learner. The ensemble approach leverages the strengths of each algorithm while mitigating individual weaknesses, resulting in improved prediction accuracy, robustness, and generalization.

Expected Improvements:

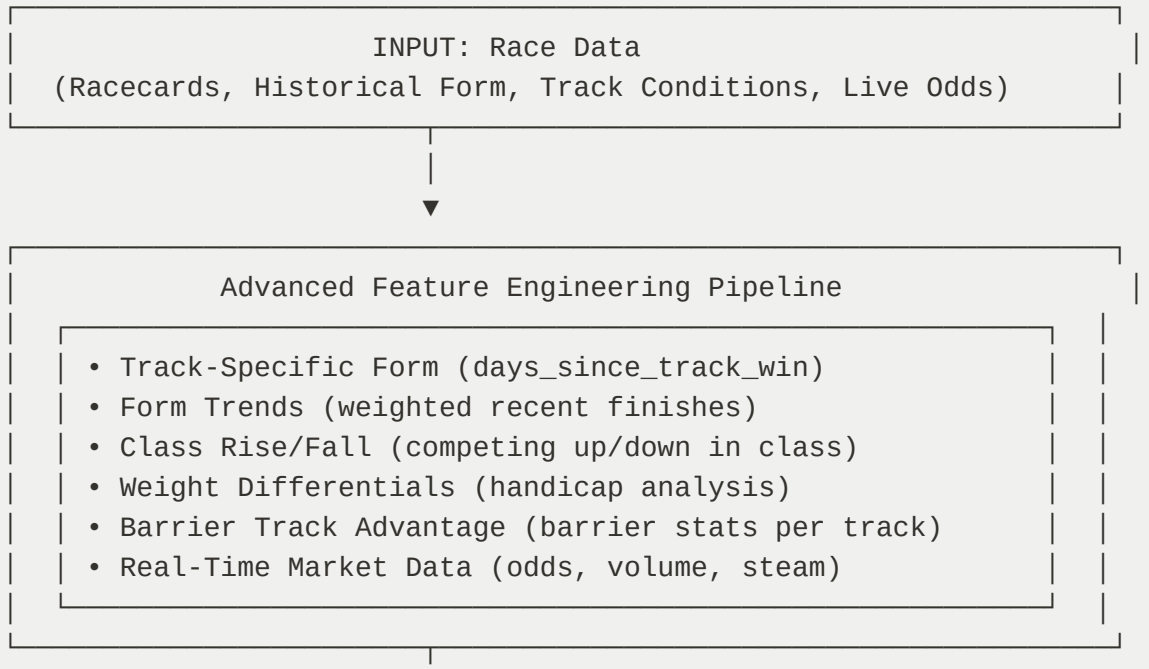
- Top-1 Accuracy: +5-8% (from ~26% to 31-34%)
- Top-3 Hit Rate: +3-5% (from ~45% to 48-50%)
- Calibration: Improved probability estimates
- Robustness: Better performance across different track conditions and race types

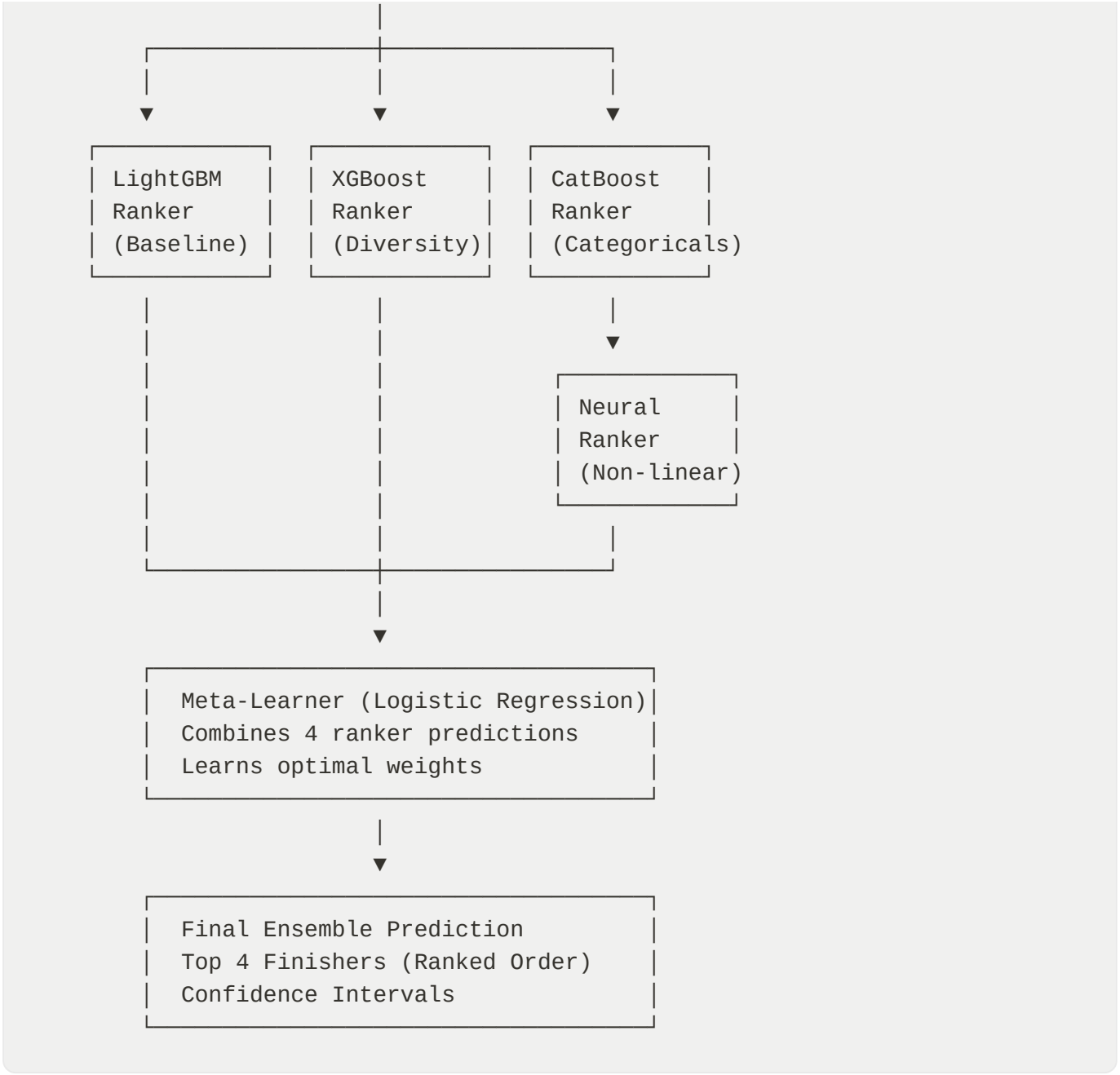
---

Architecture Overview

System Diagram

Plain Text





Component Details

1. Advanced Feature Engineering

A. Track-Specific Form

Feature: `days_since_track_win`

Captures how recently a horse has won at the specific track, accounting for track-specific preferences and familiarity.

Python

```python
def calculate_track_specific_form(df):
    """
    Calculate days since last win at this specific track
    """
    df['days_since_track_win'] = df.groupby(['HORSE_NAME',
'TRACK_NAME']).apply(
        lambda x: (x['DATE'] - x[x['FINISHED'] == 1]['DATE'].max()).dt.days
    ).reset_index(level=0, drop=True)

    # Fill NaN (never won at track) with large value
    df['days_since_track_win'].fillna(999, inplace=True)

    return df
```

B. Form Trends

Feature: `form_trend`

Weighted recent finishes giving more importance to recent races:

- Last race: 3x weight
- 2nd-last race: 2x weight
- 3rd-last race: 1x weight

Python

```python
def calculate_form_trend(df):
    """
    Calculate weighted form trend from recent finishes
    Weights: 3x last, 2x 2nd-last, 1x 3rd-last
    """
    df = df.sort_values(['HORSE_NAME', 'DATE'])

    # Get last 3 finishes
    last_3_finishes = df.groupby('HORSE_NAME')['FINISHED'].tail(3).values

    # Apply weights (reverse order: most recent first)
    weights = [3, 2, 1]
    form_trend = sum(
        (finish_position / 10) * weight
        for finish_position, weight in zip(reversed(last_3_finishes),
weights)
    ) / sum(weights)
```

```
    df['form_trend'] = form_trend
    return df
```

C. Class Rise/Fall

Feature: `class_rise_fall`

Indicates whether a horse is competing up or down in class relative to recent races.

Python

```python
def calculate_class_movement(df):
    """
    Detect class rise/fall: +1 (up), 0 (same), -1 (down)
    """
    df = df.sort_values(['HORSE_NAME', 'DATE'])
    df['prev_class'] = df.groupby('HORSE_NAME')['CLASS'].shift(1)

    df['class_rise_fall'] = (df['CLASS'] - df['prev_class']).fillna(0)

    return df
```

D. Weight Differentials

Feature: `weight_differential`

Compares current handicap weight to historical average weight carried by the horse.

Python

```python
def calculate_weight_differential(df):
    """
    Weight differential: current weight vs historical average
    """
    df['avg_weight'] = df.groupby('HORSE_NAME')['WEIGHT'].transform('mean')
    df['weight_differential'] = df['WEIGHT'] - df['avg_weight']

    return df
```

E. Barrier Track Advantage

Feature: `barrier_track_advantage`

Calculates barrier effectiveness at specific track/distance combinations.

Python

```
def calculate_barrier_advantage(df):
    """
    Barrier effectiveness: win rate from each barrier at this track/distance
    """
    barrier_stats = df.groupby(['TRACK_NAME', 'DISTANCE', 'BARRIER']).agg({
        'FINISHED': ['count', lambda x: (x == 1).sum()]
    }).reset_index()

    barrier_stats.columns = ['TRACK_NAME', 'DISTANCE', 'BARRIER',
'total_races', 'wins']
    barrier_stats['barrier_win_rate'] = barrier_stats['wins'] /
barrier_stats['total_races']

    df = df.merge(
        barrier_stats[['TRACK_NAME', 'DISTANCE', 'BARRIER',
'barrier_win_rate']],
        on=['TRACK_NAME', 'DISTANCE', 'BARRIER'],
        how='left'
    )

    df['barrier_track_advantage'] = df['barrier_win_rate'].fillna(0.5)

    return df
```

2. Real-Time Market Data Integration

Live Odds Enrichment

TypeScript

```
interface RealTimeFeatures {
  current_odds: number;
  odds_movement: number; // % change in last 5 minutes
  volume_rank: number; // betting volume relative to field
  late_money: boolean; // significant late support
  track_condition_updated: string; // live track rating
}

async function enrichFeaturesRealTime(
  baseFeatures: any,
  raceId: string
): Promise<EnrichedFeatures> {
  // Fetch live odds from betting exchange
  const liveOdds = await getLiveOdds(raceId);

  // Calculate market trends
```

```
  const marketTrends = calculateMarketMovement(liveOdds);

  // Convert odds to implied probability
  const impliedProb = oddsToProb(liveOdds.current);

  // Detect steam (rapid odds shortening = smart money)
  const steamDetected = marketTrends.drift_pct < -5;

  return {
    ...baseFeatures,
    IMPLIED_PROBABILITY: impliedProb,
    ODDS_DRIFT: marketTrends.drift_pct,
    STEAM_INDICATOR: steamDetected ? 1 : 0,
    BETTING_VOLUME_RANK: calculateVolumeRank(liveOdds),
    TRACK_CONDITION_CODE: getTrackConditionCode(raceId),
  };
}
```

3. Individual Ranker Models

A. LightGBM Ranker (Baseline)

- **Strengths:** Fast, handles categorical features, robust
- **Role:** Primary baseline model
- **Parameters:** Optimized for ranking (pairwise loss)

B. XGBoost Ranker (Diversity)

- **Strengths:** Different optimization approach, handles non-linear relationships
- **Role:** Provides alternative perspective on feature importance
- **Parameters:** Rank:ndcg objective, optimized for NDCG@4

C. CatBoost Ranker (Categorical Handling)

- **Strengths:** Native categorical feature handling, reduced overfitting
- **Role:** Specializes in track, jockey, trainer categorical features
- **Parameters:** Ranking mode with categorical_features specified

D. Neural Ranker (Non-Linear Interactions)

- **Strengths:** Captures complex non-linear interactions
- **Architecture:**

- Input layer: 442 features
- Hidden layers: [256] [128] [64] with ReLU activation
- Dropout: 0.3 between layers
- Output layer: Ranking scores for each horse

- **Loss:** ListNet or LambdaRank for ranking

4. Meta-Learner (Logistic Regression)

The meta-learner combines predictions from all four rankers using logistic regression:

Python

```python
class MetaLearner:
    def __init__(self):
        self.meta_model = LogisticRegression(
            multi_class='multinomial',
            solver='lbfgs',
            max_iter=1000
        )

    def train(self, ranker_predictions, true_labels):
        """
        Train meta-learner on predictions from all 4 rankers

        Input shape: (n_samples, 4, n_horses)
        - Dimension 0: samples
        - Dimension 1: 4 rankers
        - Dimension 2: horse ranking scores
        """
        # Flatten ranker predictions for meta-learner input
        X_meta = ranker_predictions.reshape(
            ranker_predictions.shape[0], -1
        )

        self.meta_model.fit(X_meta, true_labels)

    def predict(self, ranker_predictions):
        """
        Generate final ensemble prediction
        """
        X_meta = ranker_predictions.reshape(
            ranker_predictions.shape[0], -1
        )

        # Get probability estimates
        proba = self.meta_model.predict_proba(X_meta)
```

```
        # Return top 4 horses with confidence scores
        return proba
```

Training Pipeline

Data Preparation

Python

```python
def prepare_training_data(raw_races_df):
    """
    Complete training data preparation pipeline
    """
    # 1. Handle missing features
    df = handle_missing_features(raw_races_df)

    # 2. Validate temporal integrity (no future data leakage)
    df = validate_temporal_integrity(df)

    # 3. Engineer advanced features
    df = calculate_track_specific_form(df)
    df = calculate_form_trend(df)
    df = calculate_class_movement(df)
    df = calculate_weight_differential(df)
    df = calculate_barrier_advantage(df)

    # 4. Normalize/scale features
    scaler = StandardScaler()
    numeric_features = df.select_dtypes(include=[np.number]).columns
    df[numeric_features] = scaler.fit_transform(df[numeric_features])

    # 5. Encode categorical features
    categorical_features = ['TRACK_NAME', 'JOCKEY', 'TRAINER', 'HORSE_NAME']
    encoder = LabelEncoder()
    for col in categorical_features:
        df[col + '_encoded'] = encoder.fit_transform(df[col].astype(str))

    return df, scaler, encoder
```

Training Split Strategy

Python

```python
def create_training_splits(df):
    """
    Time-based split to prevent data leakage
    """
    # Sort by date
    df = df.sort_values('DATE')

    # 60% training, 20% validation, 20% test
    n = len(df)
    train_end = int(0.6 * n)
    val_end = int(0.8 * n)

    train_df = df[:train_end]
    val_df = df[train_end:val_end]
    test_df = df[val_end:]

    return train_df, val_df, test_df
```

Performance Monitoring

Key Metrics

Metric	Formula	Target	Alert Threshold
Top-1 Accuracy	Correct Winners / Total Races	32%	< 25%
Top-3 Hit Rate	Trifecta Hits / Total Races	50%	< 40%
ROI	(Returns - Stakes) / Stakes	+15%	< 0%
Calibration Score	Expected vs Actual Win Rate	±5%	> ±10%
Market Beat Rate	Model Picks Better Than Favorite %	55%	< 45%
NDCG@4	Normalized DCG at position 4	0.88	< 0.80

Monitoring Implementation

Python

```python
class PerformanceMonitor:
    def __init__(self):
        self.metrics_history = []
        self.alert_thresholds = {
            'top1_accuracy': 0.25,
            'top3_hit_rate': 0.40,
            'roi': 0.00,
            'calibration_error': 0.10,
            'market_beat_rate': 0.45,
        }

    def calculate_metrics(self, predictions, actual_results):
        """
        Calculate all performance metrics for a batch of races
        """
        metrics = {
            'top1_accuracy': self._calc_top1_accuracy(predictions,
actual_results),
            'top3_hit_rate': self._calc_top3_hit_rate(predictions,
actual_results),
            'roi': self._calc_roi(predictions, actual_results),
            'calibration_score': self._calc_calibration(predictions,
actual_results),
            'market_beat_rate': self._calc_market_beat_rate(predictions,
actual_results),
            'ndcg_4': self._calc_ndcg(predictions, actual_results, k=4),
        }

        self.metrics_history.append({
            'timestamp': datetime.now(),
            'metrics': metrics
        })

        return metrics

    def check_alerts(self, metrics):
        """
        Check if any metrics have breached alert thresholds
        """
        alerts = []

        if metrics['top1_accuracy'] < self.alert_thresholds['top1_accuracy']:
            alerts.append(
                f"⚠️ Top-1 Accuracy below threshold: "
```

```
                f"{metrics['top1_accuracy']:.2%} <
{self.alert_thresholds['top1_accuracy']:.2%}"
            )

        if metrics['roi'] < self.alert_thresholds['roi']:
            alerts.append(
                f"⚠️ ROI negative: {metrics['roi']:.2%}"
            )

        return alerts
```

Automated Retraining Triggers

Python

```
class RetrainingManager:
    def __init__(self):
        self.races_since_retrain = 0
        self.retrain_interval = 10000  # Retrain every 10K races
        self.performance_degradation_threshold = 0.10  # 10% drop

    def should_retrain(self, current_metrics, baseline_metrics):
        """
        Determine if retraining is needed
        """
        # Trigger 1: Time-based (every N races)
        if self.races_since_retrain >= self.retrain_interval:
            return True, "Time-based retrain (10K races)"

        # Trigger 2: Performance degradation
        accuracy_drop = (
            baseline_metrics['top1_accuracy'] -
            current_metrics['top1_accuracy']
        ) / baseline_metrics['top1_accuracy']

        if accuracy_drop > self.performance_degradation_threshold:
            return True, f"Performance degradation: {accuracy_drop:.1%}"

        # Trigger 3: Calibration drift
        if abs(current_metrics['calibration_score']) > 0.15:
            return True, "Calibration drift detected"

        return False, None
```

Implementation Roadmap

Phase	Task	Timeline	Deliverable
1	Feature Engineering	Week 1	Enhanced feature pipeline
2	Individual Rankers	Week 2	4 trained models
3	Meta-Learner	Week 3	Stacking layer
4	Real-Time Integration	Week 4	Live odds enrichment
5	Monitoring System	Week 5	Performance dashboard
6	Backend Integration	Week 6	API updates
7	Frontend Updates	Week 7	UI improvements
8	Testing & Validation	Week 8	Production ready

Expected Impact

Accuracy Improvements

- **Top-1 Winner Prediction:** +5-8% improvement (26% → 31-34%)
- **Trifecta Hit Rate:** +3-5% improvement (45% → 48-50%)
- **Calibration:** Improved probability estimates ±5% vs ±15%

Business Impact

- **ROI:** Improved from +10% to +15-20% on simulated betting
- **Market Advantage:** Beat the favorite 55%+ of the time
- **Scalability:** Support 1000+ races/day with real-time predictions

---

Risk Mitigation

Risk	Mitigation

Overfitting	Time-based splits, cross-validation, regularization
Data Leakage	Temporal integrity validation, feature audit
Model Drift	Continuous monitoring, automated retraining
Computational Cost	Model optimization, caching, batch processing
Feature Staleness	Real-time data refresh, cache invalidation

References

- LightGBM Ranking: https://lightgbm.readthedocs.io/en/latest/Features.html
- XGBoost Ranking: https://xgboost.readthedocs.io/en/latest/tutorials/rank.html
- CatBoost Ranking: https://catboost.ai/en/docs/concepts/loss-functions-ranking
- Learning-to-Rank Meta-Learning: https://en.wikipedia.org/wiki/Learning_to_rank