Take-Home Assignment: Optimising Wisdom of the Crowds for Australian Sports Betting

Question: How effectively can aggregated bookmaker odds, using the "wisdom of the crowds" model, identify value betting opportunities in Australian sports markets (AFL or NRL), and how would you implement a low-latency notification system to act on these opportunities?

Objective:
Leverage the "wisdom of dhe crowds" concept from football-data.co.uk to test if aggregated bookmaker odds from The Odds API can uncover value bets in AFL or NRL match winner markets. Design a performant, low-latency notification system in C++, Rust, or Go, and outline theoretical Betfair API automation. Try to complete this in ~2.5 hours, using AI assistance for code optimisation or analysis as needed.

Instructions:

1. **Setup and Data Collection (C++, Rust, or Go):**
   - Sign up for a free API key at The Odds API and implement a program in C++, Rust, or Go to fetch historical odds for the match winner market (home win, draw, away win) for at least 50 recent AFL or NRL matches. Include odds from 3-5 Australian bookmakers (e.g., Sportsbet, TAB, Ladbrokes, Neds, Betfair).
   - Use HTTP client libraries (e.g., `libcurl` for C++, `reqwest` for Rust, or `net/http` for Go) with proper error handling for network failures, rate limits, and invalid JSON responses.
   - Source match outcomes from aussportsbetting.com/data/ or Betfair-datascientists using a Python script (allowed for data scraping) to parse Excel or CSV files. Store outcomes in a structured format (e.g., SQLite or a flat file).
   - Optimise data fetching for performance, considering API rate limits and caching responses locally to avoid redundant calls.
2. **Data Analysis (C++, Rust, or Go):**
   - Write a program in C++, Rust, or Go to calculate average odds for each outcome (home win, draw, away win) across bookmakers for each match.
   - Convert average odds to implied probabilities (Implied Probability = 1 / Decimal Odds) and account for bookmaker margins (sum of probabilities > 100%). Normalise probabilities to sum to 100% for fair comparison.
   - Compare implied probabilities to actual outcome frequencies from

historical data (e.g., percentage of home wins, draws, away wins).

- Simulate a betting strategy: Bet on the outcome with the highest implied probability (favorite) for each match. Calculate theoretical profitability, factoring in bookmaker margins and a fixed stake (e.g., $10 per bet). Use efficient data structures (e.g., `std::vector` in C++, `Vec` in Rust, or slices in Go) for performance.

- Implement robust error handling for edge cases (e.g., missing odds, division by zero).

3. **Interpretation:**
   - Analyse whether aggregated odds better predict outcomes than individual bookmaker odds. For instance, do average odds consistently undervalue underdogs or overvalue favorites in AFL/NRL markets?

   - Identify potential value bets by comparing a single bookmaker's odds to the aggregated implied probability (e.g., a bookmaker offering 2.5 for an outcome with a 50% aggregated probability is a value bet).

   - Discuss limitations, such as bookmaker margin impact or sample size constraints.
4. **Notification System Design (C++, Rust, or Go):**
   - Design and implement a low-latency notification system in C++, Rust, or Go to alert users when a value bet is detected (e.g., when a bookmaker's odds exceed the aggregated implied probability by a threshold, say 5%).

   - Use asynchronous programming (e.g., C++'s `std::async`, Rust's `tokio`, or Go's goroutines) to poll The Odds API periodically (e.g., every 5 minutes) and compare live odds to historical averages.

   - Send notifications via email (using an SMTP library like `libcurl` for C++, `lettre` for Rust, or `net/smtp` for Go) or a webhook for SMS (e.g., via Twilio's REST API). Ensure thread-safety and minimise latency.

   - Optimise for performance: Use connection pooling for HTTP requests and in-memory caching (e.g., `std::unordered_map` in C++, `HashMap` in Rust, or maps in Go) to store recent odds.

   - Outline a theoretical Betfair API automation workflow: Fetch live odds, identify value bets, and place bets with parameters (e.g., stake, minimum odds). Discuss authentication, rate limits, and error handling without making actual API calls.
5. **Reporting:**
   - Write a concise report (400-600 words) summarising your method-

ology, analysis, findings, and notification system design. Include performance considerations (e.g., latency, memory usage) and theoretical Betfair API integration. Use AI tools to assist with drafting or code optimisation, but ensure all code is written in C++, Rust, or Go (except for Python-based scraping).

Deliverables:
- Source code in C++, Rust, or Go for data collection, analysis, and notification system, with comments explaining key logic. Include a Python script if used for scraping outcomes.
- A 400-600 word report answering the question, detailing findings, performance considerations, and the notification system/Betfair API design.
- A README explaining how to compile/run the code, including dependencies (e.g., `libcurl`, `reqwest`, `tokio`).

Constraints:
- Use C++, Rust, or Go for all core logic (data fetching, analysis, notification system). Python is allowed only for scraping match outcomes from external sources.
- Do not use the Betfair Exchange API for data collection; use only The Odds API. Betfair API discussion is theoretical.
- Handle edge cases (e.g., missing data, API errors) and optimise for performance (e.g., minimise API calls, use efficient data structures).
- No time constraints. - Use AI tools for assistance as needed.

Resources:
- The Odds API Documentation - Football-data.co.uk Wisdom of the Crowd
- Australian Sports Betting Data - Betfair API Documentation (for theoretical design)

Estimated Time: Approximately 2.5 hours.
Note: Focus on robust, performant code suitable for a mid-level developer. Prioritise error handling, concurrency, and optimisation.

Date: June 20, 2025

Example Code Snippet (Rust)

Below is a sample Rust snippet for fetching odds from The Odds API and calculating average odds, demonstrating error handling and async programming for a mid-level developer.

```rust
use reqwest::Client;
use serde::Deserialize;
use std::collections::HashMap;

#[derive(Deserialize)]
```

```rust
struct Match {
    id: String,
    sport_key: String,
    home_team: String,
    away_team: String,
    bookmakers: Vec<Bookmaker>,
}

#[derive(Deserialize)]
struct Bookmaker {
    key: String,
    markets: Vec<Market>,
}

#[derive(Deserialize)]
struct Market {
    key: String,
    outcomes: Vec<Outcome>,
}

#[derive(Deserialize)]
struct Outcome {
    name: String,
    price: f64,
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let api_key = "YOUR_API_KEY";
    let sport = "aussierules_afl"; // or "rugbyleague_nrl"
    let url = format!(
        "https://api.the-odds-api.com/v4/sports/{}/odds/?apiKey={}&regions=au&markets=h2h",
        sport, api_key
    );

    let client = Client::new();
    let res = client.get(&url).send().await?.error_for_status()?;
    let matches: Vec<Match> = res.json().await?;

    for m in matches {
        let mut odds_map: HashMap<String, Vec<f64>> = HashMap::new();
        for bookmaker in m.bookmakers {
            for market in bookmaker.markets {
                if market.key == "h2h" {
                    for outcome in market.outcomes {
                        odds_map.entry(outcome.name).or_insert(Vec::new()).push(outcome.pric
```

```
                    }
                }
            }
        }

        for (outcome, odds) in odds_map {
            let avg_odds = odds.iter().sum::<f64>() / odds.len() as f64;
            let implied_prob = 1.0 / avg_odds;
            println!("Match: {} vs {}, Outcome: {}, Avg Odds: {:.2}, Implied Prob: {:.2}%",
                m.home_team, m.away_team, outcome, avg_odds, implied_prob * 100.0);
        }
    }

    Ok(())
}
```

Notes for Developers:
- The snippet uses `reqwest` for HTTP requests and `serde` for JSON parsing, with proper error handling via `Result`.
- You can extend this by adding SQLite (You can use anything else too) for storing outcomes, async notifications with `lettre` or `tokio`, and caching with a `HashMap`.
- For C++, use `libcurl` and `nlohmann/json`; for Go, use `net/http` and `encoding/json`.
- Focus on thread-safety (e.g., mutexes in C++, `Arc<Mutex<>>` in Rust, or channels in Go) for the notification system. - We are not expecting the notification system to work, but the implementation must be sound - This is an open-ended question you are free to do anything you like as long as the core requirements are met.