Tennis Match Outcome Prediction: Classic Formula vs. Machine Learning Approach

Introduction

This report explains a C++ program (tennis_predictor.cpp) that predicts the outcome of a tennis match between two players based on their performance statistics. The program implements **two different prediction methods** side by side: a **classic formula-based approach** derived from domain expertise, and a **machine learning approach** using a simple linear model (logistic regression). We will break down how the program reads input data, computes predictions with the formula, trains the linear model, and produces output. Both technical details (like the use of a sigmoid function and gradient descent) and high-level concepts are covered, so that readers of varying expertise can understand the underlying algorithm and code.

Overview of Input and Output

The program is executed from the command line and expects **two CSV files** as input, each containing historical performance stats for one player. It aggregates those stats for each player (using median by default) to get a representative profile. The key steps and I/O can be summarized as follows:

- **Input:** Two CSV files, one for *Player A* and one for *Player B*. Each file contains multiple records of the player's stats (with a header row). By default the program will use the **median** of each stat column as the player's overall value (an option allows using the average instead).
- **Player Stats in CSV:** Each player's stats file should have the following columns in order:
    - **DR:** *Dominance Ratio* – a measure of overall dominance (ratio of points won to points lost; >1 means the player typically wins more points than the opponent).
    - **A% (Ace %):** Percentage of serves that are aces (serves winning a point outright).

- **DF% (Double Fault %):** Percentage of serves that are double faults (serves losing a point without opponent input).
- **FirstIn%:** First serve in percentage (how often the first serve lands in bounds).
- **FirstServeWin%:** First serve win percentage (the percentage of points won when the first serve goes in).
- **SecondServeWin%:** Second serve win percentage (the percentage of points won when a second serve is used).

Command-line Options: The program supports flags:

- --avg to use average instead of median for aggregating stats.
- --classic to use only the classic formula prediction.
- --ml to use only the machine learning prediction.
- --both (default) to output both predictions for comparison.

Output: For each player, the program prints a **Stats Summary** (the aggregated values of each stat). Then, depending on the mode, it prints:

- **Classic Formula Prediction:** The computed score for each player using the formula (called **Wsp** in the code), the difference between these scores (CSA), and the implied win probability for each player.
- **Machine Learning Prediction:** The predicted win probability for each player from the trained linear model, followed by a list of **feature importance** values indicating which stats differences had the most influence in the model.

In summary, the program reads two players' performance data, calculates a predicted win chance for Player A vs Player B using two methods, and displays those predictions along with a breakdown of the stats.

Data Loading and Preprocessing

Before making any predictions, the program **loads and preprocesses the data** from the CSV files. Each CSV file may contain multiple entries of a player's performance stats (for example, stats from many past matches). The code aggregates these into a single representative profile for each player:

- **Parsing CSV Files:** The function load_all_player_stats() reads a CSV file line by line (skipping the header) and converts each line into a PlayerStats structure. It uses C++ file I/O and string parsing:

Each PlayerStats holds the numeric values for DR, Ace%, DF%, First serve in%, First serve win%, and Second serve win%.

Median vs Average Aggregation: Once all records are loaded, the function load_player_stats() computes a single aggregated PlayerStats for that player. By default it uses the **median** of each column of values

Using the median is a robust choice to avoid outliers (e.g. one unusual match with extremely high aces or double faults won't skew the result too much). The user can opt for average with --avg if desired. After this step, we have one PlayerStats struct for each player (Player A and Player B) representing their typical performance.

At this stage, the input data has been transformed into a convenient form for the prediction algorithms: we have two sets of aggregated stats (one per player) that will feed into both the classic formula and the ML model.

Classic Formula Prediction Method

The first prediction method uses a **hand-crafted formula** based on tennis domain knowledge to evaluate each player's serving performance. This formula produces a score called **Wsp** (we can think of it as a *Weighted Service Performance* score) for each player, and the difference in these scores is then converted to a win probability using a sigmoid function.

1. Calculating the Wsp Score:
For each player, the program computes Wsp = Wa + Wdf + W1st + W2nd based on their stats:

- **Ace Contribution (Wa):** equal to the Ace percentage (as a fraction). Aces directly contribute to winning points, so higher ace rate increases Wsp. For example, if Ace% = 12%, then Wa = 0.12.
- **Double Fault Contribution (Wdf):** equal to **-3** times the double fault rate (as a fraction). Double faults are severely penalized because they lose points; the factor of -3 gives triple weight to the negative impact of double faults. For example, DF% = 5% gives Wdf = -3 * 0.05 = -0.15.
- **First Serve Contribution (W1st):** accounts for points won on **non-ace first serves**. It is calculated as 2 * (FirstIn - AceFraction) * FirstServeWinFraction. Here:
    - (FirstIn - AceFraction) is the fraction of serves that were first serves **in** play but **not aces** (we subtract aces since they were counted in Wa).
    - Multiply by the first-serve win percentage to get the fraction of total points won via a successful first serve (excluding aces).
    - This product is doubled (×2) to weight these first-serve points fairly strongly in the Wsp score.
- **Second Serve Contribution (W2nd):** accounts for points won on second serves. It is 4 * (SecondServeInFraction) * SecondServeWinFraction. Here:
    - SecondServeInFraction can be inferred as 1 - FirstIn - DF (the fraction of points where the first serve missed but a second serve was successfully made, i.e., not a double fault).
    - Multiply by second-serve win percentage to get the fraction of points won on second serve, and then weight it by ×4, emphasizing the importance of second-serve points (which are critical in tight situations).

Putting it together more formally, if we denote the stats (as fractions) for a player as: *A = Ace%/100, DF = DF%/100, FI = FirstIn%/100, Fw = FirstServeWin%/100, Sw = SecondServeWin%/100*, then:

$$\text{Wsp} = A + (-3 \times DF) + 2 \times (FI - A) \times Fw + 4 \times (1 - FI - DF) \times Sw.$$

This single number is a composite score reflecting the player's serving success. Higher Wsp means better expected performance. For example, using the formula:

- If Player A has very strong stats (high aces, low double faults, high first and second serve win rates), Wsp_A will be relatively high.
- If Player B is weaker in those areas, Wsp_B will be lower.

2. Computing Win Probability (Sigmoid Function):
After calculating Wsp for Player A and Player B, the program finds the **difference**: CSA = WspA - WspB. We can call this difference **Composite Score Advantage (CSA)** for Player A. A positive CSA means Player A's serve performance is better than Player B's, implying an advantage for Player A. A negative CSA would favor Player B.

To translate this score difference into a **win probability**, the program uses the logistic sigmoid function. In code, this is done as:

double CSA = WspA - WspB;

double winrateA = 1.0 / (1.0 + exp(-CSA));

double winrateB = 1.0 - winrateA;

The sigmoid function σ(x) = 1/(1 + e^{-x}) converts any real value into a range between 0 and 1, making it suitable for probabilities. When CSA = 0 (meaning both players have equal Wsp), exp(-0) = 1 and thus winrateA = 0.5 (50% – a toss-up match). If CSA is positive (Player A has better stats), exp(-CSA) is less than 1, making winrateA > 0.5. The larger the CSA, the closer winrateA gets to 1 (though with diminishing returns due to the S-shaped curve of sigmoid). Similarly, a negative CSA yields winrateA < 0.5 in a smooth, continuous way. This logistic formula is commonly used to map a scoring difference to win probability in Elo ratings and other sports models.

Player B's win rate would be 46.25%. A small advantage in Wsp translates to a somewhat above-50% chance of winning for Player A.

Machine Learning Model Design and Training

The second approach in the program uses a **machine learning model**, specifically a simple **linear logistic regression model**, to predict the match outcome. Instead of relying on fixed formula weights (like 2x or 4x in Wsp), this model will **learn the weights** from data. The code encapsulates this in a class LinearModel which implements training and prediction for a logistic regression.

1. Model Structure (Linear Model):
The linear model considers the **difference in stats between Player A and Player B** as input features. There are 6 features (as there are 6 stats), and the model will output a probability of Player A winning. Internally:

- It has a weight coefficient for each feature and an additional bias term. These weights and bias are initially set to small random values.
- To predict a match outcome, the model computes a weighted sum of the features plus the bias, then applies the sigmoid function to produce a probability between 0 and 1

2. Feature Engineering for the Model:
Before training or using the model, we need to provide it with input features that capture the matchup between the two players. The program uses a simple and effective feature engineering strategy: for each stat, take the **difference between Player A's value and Player B's value**. This is implemented by the function generate_features(playerA, playerB).

Each feature represents a *competitive edge* of Player A over Player B in that statistic:

- A positive value means Player A is better in that metric; negative means Player B is better.
- For instance, a positive **Ace % difference** means Player A hits a higher percentage of aces than Player B, which should increase A's chances.
- A positive **DF % difference** (Player A's double fault % minus Player B's) actually means Player A double-faults more than B, which is bad for A. We would expect the model to learn a **negative weight** for this feature (since higher double faults for A decreases A's win probability).

Using differences as features is intuitive and also ensures symmetry – if we swapped the players, the features would just invert in sign and the model would predict the opposite

outcome. This is a common approach in sports modeling: model the match outcome as a function of the difference in competitor attributes.

3. Training Data Generation:

Machine learning models require training data (examples of matches with known outcomes) to learn from. In this program, rather than reading an external dataset, the code **generates synthetic training data** to simulate past match outcomes. This approach is taken likely because a readily available dataset wasn't provided, so the author created a plausible dataset on the fly:

- The function generate_training_data(int num_samples) creates a specified number of hypothetical matches. For each sample:
- It randomly generates stats for a fake Player A and Player B. The random values are drawn from distributions that mimic real tennis stats ranges. For example, aces are typically between 0% and 20% of serves, so it uses a normal distribution with mean ~10%, std ~4% (bounded between 0 and 25%). Similar realistic ranges are used for DR, DF%, first serve %, etc.
- Using those generated stats, it computes WspA and WspB using the same formula as above, then computes a win probability for A (sigmoid of CSA).
- It then randomly decides a winner according to that probability (i.e. it flips a biased coin). This means if A's win probability was 70%, in 70% of similar samples A will be marked as the winner (result=1), otherwise B wins (result=0).
- It records the feature differences (A's stats minus B's stats) and the outcome (1 or 0) as one training example.
- This yields a **training dataset** of MatchData entries, where each has features (a vector of 6 differences) and a result (1 if A won, 0 if B won). The dataset is **balanced** by the nature of random draws and probabilistic outcome assignment (there's no systematic bias towards A or B winning except that driven by the stat differences and formula).

This synthetic data approach effectively uses the classic formula model to create examples, so that the ML model can learn to approximate the same kind of decision-making. While not as valuable as real match data, it provides a teaching signal to the linear model.

4. Gradient Descent Training:

With a training dataset in hand, the program trains the linear model using **batch gradient descent**. The training routine (LinearModel::train) iteratively adjusts the weights and bias to better fit the data:

- It runs for a fixed number of iterations (1000 by default in the code).
- In each iteration (epoch), it goes through every training example, computes the model's prediction, and measures the error. The **error** is defined as prediction - actual_result for each example (where actual_result is 1 or 0).
- It accumulates the **gradient** of the loss with respect to each weight. The loss function here is the mean squared error (MSE) for simplicity (note: logistic regression typically uses log-loss/cross-entropy, but MSE works acceptably for training a logistic model in this context). For each feature weight w_i:
    - The gradient is proportional to the error times the feature value x_i. Intuitively, if the model over-predicted (error is positive) and the feature value is positive, it will reduce the weight (and vice versa).
- After summing over all training samples, the code updates each weight by a small step in the opposite direction of the gradient (hence **gradient descent**):
- Here N is the number of samples, so they are using the average gradient. The learning rate is a parameter (0.1 in this program) that controls how big each update step is. A moderate learning rate and 1000 iterations were chosen to ensure convergence.
- The program prints out the loss periodically (at iteration 0 and every 100 iterations) to show training progress. The loss typically decreases and stabilizes as the model learns.

By the end of training, the model's weights capture the relationships between feature differences and winning. Because the training data was generated using the Wsp formula logic, the learned weights should, in theory, reflect that formula's influence: for example, we expect a negative weight on double fault difference (because more double faults by A makes winning less likely), and a positive weight on first-serve win percentage difference, etc. The bias term will adjust the base probability.

5. Trained Model and Weights:

After training, the program prints the final learned weights for interpretation. For instance, one run might output:

These weights (just an example) mean the model found **first serve win percentage difference** and **double fault percentage difference** to be the most influential features (0.12 and -0.08 respectively), while other features had smaller contributions (some near 0).

Note that weights are in logistic terms – they affect the log-odds of winning. A weight of 0.12 for first serve win% difference indicates that for each 1 percentage point increase in Player A's first-serve win rate advantage, the log-odds of A winning increase by 0.12. In simpler terms, a higher first-serve success rate for A significantly boosts A's win probability according to the model. The negative weight on DF% means if A's double fault rate is even a bit higher than B's, it drags down the win chances. (If the model were trained on real data, these weights would tell us which factors historically matter most for winning a match.)

It's worth noting that because the model is linear, it cannot capture complex interactions between features as the Wsp formula did (the formula had multiplications like FirstIn * FirstWin%). The linear model tries to approximate those effects with a weighted sum. In practice, with enough data, it might still align roughly with expert intuition, highlighting second serve performance or other key stats, but differences can occur. Nonetheless, the model provides a flexible, data-driven way to predict outcomes.

Feature Engineering

Feature engineering refers to how raw data is transformed into inputs for the model. In this program, the feature engineering is straightforward and is done in the generate_features function when comparing two players. The **six features** used by the machine learning model are:

1. **Dominance Ratio Difference (DR diff):** playerA.DR - playerB.DR – a positive value means Player A has a higher dominance ratio (indicating stronger overall performance).
2. **Ace % Difference:** playerA.A_percent - playerB.A_percent – positive if A serves more aces.
3. **Double Fault % Difference:** playerA.DF_percent - playerB.DF_percent – positive if A double-faults more (which actually is a disadvantage for A).
4. **First Serve In % Difference:** playerA.FirstIn - playerB.FirstIn – positive if A gets a higher percentage of first serves in.
5. **First Serve Win % Difference:** playerA.FirstPercent - playerB.FirstPercent – positive if A wins a higher percentage of points on their first serve.
6. **Second Serve Win % Difference:** playerA.SecondPercent - playerB.SecondPercent – positive if A wins more points on second serve.

By using differences, the model essentially focuses on **how much better or worse Player A is relative to Player B** in each aspect:

- If a weight associated with a feature is positive, it means the model has learned that *the larger that difference (in A's favor), the higher the chance of A winning*. If a weight is negative, a larger difference (with A ahead) actually lowers A's win probability – which would be the case for the double fault rate, for instance, since being "ahead" in double faults is bad.
- If a weight is near zero, the model found that particular stat difference not very useful in predicting the outcome, possibly because its effect is redundant with other stats or just not strongly correlated with winning in the training data.

This feature set captures many important facets of tennis matches (serve quality and consistency). One notable aspect is that **only serve-related stats are used** (since Wsp formula was also focused on serve performance). Other factors like return statistics are not included, but Dominance Ratio (DR) somewhat covers overall play including returns. In a more complex model, one might add more features, but the given six provide a reasonable basis for demonstration.

Prediction and Output Interpretation

After either computing the formula or training the model (or both), the program uses those methods to predict the outcome of the actual matchup between the given players and prints the results.

Classic Formula Prediction Output: As described earlier, the output of the classic method includes each player's Wsp score, the CSA, and the win percentages. This output is fairly **interpretable**:

- Wsp scores indicate the players' serving performance in a way that fans or analysts might discuss (higher is better).
- The difference (CSA) being positive or negative clearly states which player is favored according to the formula.
- The win percentages give a probabilistic prediction. For example, a 60% win rate means if these two players played 10 matches under similar conditions, we'd expect Player A to win about 6 out of 10.

One should note that the formula's accuracy depends on how good the Wsp model is – it's a hypothesis that certain weighted stats can predict match outcomes. It might not account for everything (like return game strength), but it provides a quick **estimate**.

Machine Learning Prediction Output: The ML section of the output provides:

- The predicted win probability for Player A and Player B from the model (as percentages). For instance, it might output *"PlayerA Win Rate: 63.22%"*. This can be interpreted similarly (if they played many matches, the model expects A to win ~63% of the time given these stats differences).
- **Feature Importance:** This is a list of the features with their importance values. In the code, importance is defined as the absolute value of the weight for each feature, sorted from highest to lowest.

This ranking tells us which stat differences most influenced the model's prediction. In this sample, **First serve win % difference** was the top factor (weight magnitude 0.12), and **Double Fault % difference** next (0.08). Ace % and others had negligible weight in this particular trained model. This resonates with an interpretation that winning points on the first serve and not double-faulting were crucial in those synthetic matches. Keep in mind, feature importance here is directly the learned weights' magnitudes – since it's a linear model, that is a reasonable indicator of influence.

For a non-technical user, the feature importance essentially answers, *"Which stats made the biggest difference in this prediction?"* The higher the importance number, the more that stat swung the prediction. For technical readers, it's essentially the model's way of explaining itself in terms of the input features.

Comparing the Two Methods: In many cases (especially given the way training data is generated), the classic formula and the ML model may give similar predictions. However, they could differ. The classic formula is deterministic given the inputs, whereas the ML model's output can vary slightly due to random initialization and training (each run could produce slightly different weights unless a fixed random seed is used). Moreover, the linear model might not capture the non-linear intricacies of the formula exactly, leading to different probability estimates. If both are output (--both mode), the user can see both perspectives:

- If they agree (both say ~70% chance for Player A, for example), that reinforces confidence in the prediction.
- If they differ, it highlights that the approach to modeling can affect the result – perhaps an indication that the relationship of stats to outcome isn't perfectly linear or that certain factors weighed differently.

In any case, both outputs are meant to be **informative rather than definitive**. They provide insights based on past performance stats: the classic one rooted in tennis analytic heuristics, and the ML one rooted in data-driven learning.

Code Snippets and Annotations

To tie everything together, let's walk through some key parts of the code with annotations, which illustrate the program's logic:

- We load and aggregate stats for both players from their CSV files.
- We print the stats so the user can see what values are being used.
- If the classic method is enabled, we:
    - Calculate Wsp for each player using calculate_Wsp.
    - Compute the difference (CSA) and then apply the sigmoid to get win rates.
    - Print the win rates for Player1 and Player2.
- If the ML method is enabled, we:
    - Generate synthetic training data (here 5000 matches) via generate_training_data.
    - Initialize a linear model with 6 features, a learning rate of 0.1, and 1000 iterations, then train it on the data.
    - Compute the feature difference vector for the actual players and get a prediction (ml_winrateA).
    - Print the ML-predicted win rates for both players.
    - Retrieve and print the feature importance ranking from the model.

Throughout the code, careful use of functions and struct data types makes the logic clearer. For instance, the separation of concerns:

- calculate_Wsp encapsulates the formula details.
- LinearModel encapsulates all machine learning behaviors (predict, train, etc.).
- generate_features cleanly produces the feature vector.
- print_player_stats handles displaying stats nicely.

This modular design makes the code easier to read and explain. Each part we discussed corresponds to a specific function or block in the code, which we have annotated above. The code uses standard C++ libraries for random number generation, file I/O, etc., and prints to console for results.

Conclusion

In summary, the tennis_predictor.cpp program demonstrates two approaches to predicting tennis match outcomes using players' performance statistics:

- The **classic formula-based approach** uses a predefined weighted formula (Wsp) focusing on serve-related stats. It is straightforward and provides interpretable components (like how aces or double faults contribute). The result is a probability derived via a logistic curve, much like how one might convert a point difference to a win probability.
- The **machine learning approach** uses a simple linear model trained via gradient descent. It takes into account the same stats (in the form of differences) but lets the data determine the weights. This approach is adaptive – given real match data, it could learn the optimal influence of each factor. In this program, it learned from synthetic data that was aligned with the formula's logic, effectively validating the formula in a data-driven way. The ML model also provides a form of explanation through feature importance (the magnitude of learned weights).

Both methods ultimately leverage the logistic sigmoid to produce a winning probability, underlining that the problem is treated as a binary classification (win or lose) with an underlying score difference. Key concepts like the sigmoid function ensure the probabilities make sense (between 0 and 1), and gradient descent allows the model to improve its predictions iteratively.

For a technical audience, the code offers insight into implementing a simple machine learning algorithm from scratch in C++ and combining it with domain knowledge. For a non-technical audience, the report's explanations of what each statistic means and how they influence the prediction can help demystify how a computer program can forecast a match outcome.

In a real-world scenario, one might extend this program by incorporating actual historical match outcomes to train the ML model, possibly add more features (like return stats or head-to-head records), or use more advanced models. However, the given implementation provides a clear educational example of blending **sports analytics formulas** with **machine learning**, each validating and informing the other. The output of the program, with both methods side by side, allows users to see a classic analytic prediction and a learned prediction for the same match-up, making for an interesting comparison and deeper understanding of tennis statistics.