For our project, we were interested in analyzing trends in professional basketball and the NBA. We focused on predicting an NBA player's next game score. We explored methods to transform each play-by-play data into lower-dimensional numerical values that could be used for statistical analysis and machine learning interfaces.

We focused on total points scored (PTS), rebounds in a game (REB), and number of assists (AST). Over a sample size of 10 games, we standardized the data by taking the z-score of each dataset using the formula

$$z = \frac{x - \mu}{\sigma},$$

where x represents the player's observed statistic for a given game, μ is the mean of that statistic across the 10-game window, and σ is the standard deviation over that same period. This standardization ensures that all variables (points, rebounds, and assists) have comparable scales, preventing any single variable with larger numeric values, such as points, from disproportionately influencing the statistical analysis.

This standardization is implemented in our code through the `get_pca_vector()` function, which applies `StandardScaler()` from `scikit-learn` before applying PCA. Normalization is particularly important for methods sensitive to scale, like covariance analysis. After z-scoring, each statistic has mean 0 and standard deviation 1. Since

$$\mathrm{Corr}(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y},$$

after taking the z-score, each $\sigma_X, \sigma_Y = 1$, and thus

$$\mathrm{Corr}(X, Y) = \mathrm{Cov}(X, Y).$$

We use these standardized values in a maximum-likelihood estimator.

We next perform Principal Component Analysis (PCA), which takes our score data — which is correlated — and turns it into an orthogonal basis, highlighting the directions of largest variance. Since these directions hold the most variance, they most likely contribute more towards a player's average performance. Mathematically, PCA solves an eigenvalue problem involving the covariance matrix of the data.

PCA is implemented specifically in our code within the `get_pca_vector()` function, using `PCA()` from `scikit-learn`, where we average the PCA-transformed coordinates across recent games. This compresses the large matrix of data without losing significant information. The result is computationally efficient.

Each principal component can be interpreted as a blend of skills — for example, "high-usage scoring," "good rebounding," or "facilitation/playmaking." By representing a player's recent ten-game history as the average coordinates along these eigenvectors, we get a compact form that feeds cleanly into a similarity search and predictive modeling, while maintaining a proper connection to on-court behavior.

Each English summary produced after the PCA stage is converted to a high-dimensional vector, allowing similar games to be identified with vector arithmetic. An `all-MiniLM-L6-v2` model tokenizes the sentences — changing words into numbered tokens — and averages the resulting token vectors. This sentence encoding step is handled by the `build_vector_db()` function. All such sentence vectors are stored in a FAISS structure called `IndexFlatL2`, essentially a flat matrix in memory. This steers the language model toward outcomes consistent with nearby examples while allowing natural generation. In short, the model maps text to points, FAISS finds the nearest points, and the language model uses that neighborhood to base its forecast.

This embedding and retrieval process forms the basis of our Retrieval-Augmented Generation (RAG) pipeline. Traditional statistical models often rely solely on broad averages, ignoring the context. Our RAG pipeline addresses this by first retrieving contextually relevant past performances stored in FAISS. For each prediction, the most relevant historical games, identified through the similarity search, are retrieved. The RAG pipeline significantly enhances the model's accuracy by ensuring predictions consider the relevant historical contexts.

Prior to generating a forecast from a player's recent averages, we incorporate two context-specific adjustments reflecting the future opponent. First, we quantify the opponent's shot-prevention ability by computing its season-to-date defensive field-goal percentage $\hat{p}$, calculated by

$$\hat{p} = \frac{\text{Field Goals Made}}{\text{Field Goals Attempted}}.$$

This is retrieved through the `get_opponent_defensive_stats()` function from the `LeagueDashPtTeamDefend` endpoint using `nba_api`. If $\hat{p}$ lies below the league mean $\bar{p}$, the projected scoring output is proportionally reduced; if $\hat{p}$ exceeds $\bar{p}$, the projection is increased.

Second, we correct counting statistics for expected floor time. Each raw projection S is rescaled linearly by multiplying by

$$\frac{\text{minutes that the coach uses a certain player}}{\text{average minutes played by this player across ten games}}.$$

These modifiers help reduce bias due to opponent defensive strength and variations in player minutes.

Once these matchup numbers are attached, we bundle everything into one prompt:

- the 3-number PCA vector (an orthogonal summary of points, rebounds, assists),
- the opponent's defensive FG%,
- the player's minutes trend,

and input this into a large language model (LLM) trained on memories of prior games.

This structured information is input into a large language model, specifically OpenAI's GPT-4o, trained on extensive data including prior NBA game summaries. This prompt is generated in our code through the `generate_llm_prompt()` function, which integrates PCA vectors, defensive statistics, recent historical performances, and minute adjustments before passing them to the GPT-4o API.

The LLM outputs predictions for points, rebounds, and assists, providing explanations grounded in the statistical and historical context supplied in the prompt.

In the future, our application can be extended beyond player-level performance forecasting to broader analytics use-cases, such as real-time game scenario modeling, optimizing lineups based on predictive analytics, or assisting coaching decisions by evaluating optimal player matchups. We can incorporate additional metrics like shot charts, defensive impact metrics, or player fatigue data to enhance predictions. Moreover, the structure of our application could be adapted to sports beyond basketball or used in sports media for AI-driven predictions.

Overall, we take ordinary multivariate statistics, clean and compress the datat using PCA, refer to relevant past examples, and use tokenized representations of in-game context, feeding them into a large language model that structures the information into a predictor for future games.

References

[1] *nba_api: An API Client Library for NBA Data*, Python Package Index, 2019. Available: https://pypi.org/project/nba-api/.

[2] Facebook AI Research, *FAISS: Facebook AI Similarity Search*, 2017. Available: https://faiss.ai.

[3] OpenAI, *GPT-4o Technical Report*, 2024. Available: https://openai.com/research/gpt-4o.