LISBON DATA SCIENCE ACADEMY

Retail Pricing Strategy

Campaign Effectiveness and Predictive Modeling for Competitor Behavior

Prepared for:
Retailz

Prepared by:
José Miguel Mendes
Data Scientist

03/06/2025

Table Of Contents

1. Summary

The primary business requirement from Retailz was to anticipate competitor pricing behavior to enable more informed and proactive pricing decisions. By forecasting the final selling price (PVP) of products, Retailz aimed to remain competitive without compromising margins. This predictive capability supports the pricing intelligence team in adjusting retail prices in alignment with market movements and seasonal demand.

After data exploration and experimentation with various modeling approaches, we concluded that a tree-based regression model offered the best balance between accuracy, interpretability, and operational simplicity. It captured complex interactions between historical price differences, seasonal effects, and the base chain price without requiring extensive parameter tuning.

The final model was implemented as a reusable pipeline, integrating preprocessing and prediction into a single artifact. This pipeline was deployed via a fully functional API that is now running in a production environment. The API receives a SKU and a target date, dynamically constructs the necessary input features using historical data, and returns the predicted final price in real time. This deployment is scalable, robust, and actively supports Retailz's internal decision-making processes.

2. Modelling

2.1 Model specifications

To address Retailz's business goal of forecasting competitor pricing, we framed the task as a supervised regression problem, predicting the final competitor price (pvp_final) for a given product (SKU) and date. Identical models were created for Competitor A and Competitor B, with each version using a dedicated feature set reflecting that competitor's pricing dynamics.

- Feature Engineering: We engineered features based on:

- **Lag-based pricing features** (e.g., lag_diff, lag_diff_sl4, diff_std_sku) designed to quantify historical price discrepancies and their temporal dynamics. These features were generated by shifting and aggregating historical price difference data across multiple grouping levels, such as SKU and product category hierarchies (e.g., structure_level_4). The calculation leverages a strictly causal window by using only past observations up to the current inference timestamp, ensuring no future leakage. For example, lag_diff corresponds to the previous period's price gap, while lag_diff_sl4 captures lagged differences at a higher category aggregation. The diff_std_sku feature applies a standardized z-score transformation over historical price differences within each SKU group, robustifying the model to volatility in pricing behavior.

  This process assumes the availability of historical pricing data (including competitor and chain prices) at inference time, which is maintained through a rolling data pipeline that updates these variables before prediction generation.

- **Chain price**: Retailz's current product price, a key predictor.
- **Temporal features**: Extracted using a custom TimeFeaturesExtractor, which derived month, day, weekday, and seasonal event flags including:
    - Christmas season
    - New Year window
    - Summer months
    - Back-to-school period
    - Black Friday (using date logic)

These features allow the model to learn from both past competitive pricing behavior and known market seasonality.

- Model Design: The final model was built using a RandomForestRegressor, integrated into a pipeline consisting of:

1. **TimeFeaturesExtractor**: a custom class to add date-based features
2. **SimpleImputer**: to fill missing values in numerical fields using the mean
3. **Random Forest**: for prediction, chosen for its ability to model non-linear patterns and resist overfitting

- Hyperparameter Tuning: We tuned hyperparameters using RandomizedSearchCV with 3-fold cross-validation and the following search space:

- n_estimators: 50–200
- max_depth: 5–30
- min_samples_split: 2–10
- min_samples_leaf: 1–10

The full pipeline and feature engineering logic were serialized into a .pkl file for reuse, along with grouped historical SKU-level data for real-time inference.

- Handling Lag Features in Real-Time Prediction: Since lag-based features are calculated from historical data and are not provided as inputs via the API in real-time, a strategy was devised to fill these features using pre-aggregated historical SKU-level data. For a given SKU and target date, the system searches the historical data for the closest previous date with matching day and month to populate lag features such as lag_diff, lag_diff_sl4, and diff_std_sku. If an exact date match is unavailable, the most recent prior date is used as fallback. This ensures the model receives consistent feature inputs during inference, maintaining prediction accuracy despite the absence of direct lag feature inputs in the API request.

-Metric Justification: MAE (Mean Absolute Error) was chosen as the primary metric because it directly measures the average magnitude of prediction errors in the same units as the target price, making it intuitive and interpretable for business stakeholders. It penalizes errors proportionally, aligning well with the business goal of minimizing absolute price deviations.

sMAPE (Symmetric Mean Absolute Percentage Error) complements MAE by providing a scale-independent percentage error, which is useful for understanding relative prediction accuracy across products with varying price levels. This is critical in pricing contexts where both absolute and relative errors matter.

Together, these metrics provide a balanced and comprehensive assessment of model performance, ensuring that predictions are both numerically precise and proportionally accurate—key factors for effective pricing strategy and operational deployment.

As requested by the client, we have included an evaluation of model performance across different product categories (structure_level_2) using MAE and sMAPE metrics. The results indicate that sMAPE values are consistently below 1% for all structure_level_2 groups, meaning the model's predictions deviate on average less than 1% from the actual prices. This demonstrates a high level of accuracy and robustness across categories.

Therefore, the model's performance at the structure_level_2 granularity is considered very satisfactory, confirming its suitability for practical deployment.

2.2 Model performance

- Evaluation Metrics:
The model's performance was evaluated by comparing the predicted prices with the actual competitor prices stored in the database for the same SKU.

MAE shows the average difference between the predicted and actual prices, indicating the model's average error.

sMAPE is a percentage error that balances the actual and predicted prices, helping to understand the model's relative accuracy.

Competitor	Model Version	MAE (Mean Absolute Error)	sMAPE (Symmetric Mean Absolute Percentage Error)
Competitor A	Original	4.35	9.70%
Competitor A	Retrained	3.90	10.67%
Competitor B	Original	4.67	10.53%
Competitor B	Retrained	4.47	10.75%

-Retraining Context and Feature Engineering Strategy:

The retrained model was developed following the first round of predictions, which exposed performance limitations in specific product structures—most notably in segments such as structure levels 104 and 105. These results revealed the need to improve absolute prediction accuracy and ensure more robust performance across segments.

Retraining was performed using an updated dataset that preserved the original model's feature structure while incorporating approximately 498 new records gathered after the initial prediction cycle. These new data points provided more recent and representative pricing behavior, enabling the model to learn from previous shortcomings. The objective was to fine-tune performance in underperforming segments and improve overall prediction reliability.

In parallel, a targeted feature engineering and imputation strategy was implemented to prepare the new SKUs and their predicted pvp_final values for future forecasts. Since several time-sensitive features may be missing for new or recently introduced SKUs—such as lag_diffA, lag_diffA_sl4, diffA_std_sku, and chain_price—a robust imputation method was applied:

- **Historical Aggregation:** Using a pre-processed dictionary of historical dataframes (one per SKU), we calculated monthly averages for each feature per SKU, capturing seasonal and cyclical behaviors.
- **SKU-Month Matching:** For each missing value in the future prediction dataset, the corresponding SKU and month were identified. When available, the missing feature was imputed using the historical average for that specific SKU-month pair.
- **Fallback Handling:** If no match was found for a given SKU/month combination, the value remained as NaN. This allows for future fallback imputation strategies or relies on the model's tolerance for missing values.

This combined approach—retraining with enriched data and targeted feature imputation—ensured that predictions remained consistent, accurate, and reflective of real-world pricing dynamics, even for SKUs with limited historical data.

- Performance Analysis:

For Competitor A, retraining improved the MAE from 4.35 to 3.90, indicating that the average absolute prediction error decreased, which means the model's point predictions became more accurate in terms of price units. However, the sMAPE slightly increased from 9.70% to 10.67%, suggesting a marginally higher relative percentage error after retraining.

For Competitor B, retraining also yielded improvements in MAE (from 4.67 to 4.47), showing better absolute accuracy. The sMAPE increased slightly from 10.53% to 10.75%, a small trade-off in relative error.

Overall, the retrained models reduced absolute errors but incurred minor increases in relative error percentage. This behavior may be due to the relatively small dataset size used for retraining—approximately 498 entries—which can limit the model's ability to generalize perfectly across all price ranges and impact the stability of percentage-based error metrics like sMAPE.

-Performance Across Product Structures (structure_level_2):

To ensure the model meets business expectations at a more granular level, we analyzed MAE and sMAPE for each structure_level_2 category:

Key observations for Competitor A:

- Significant improvements were seen in structures 104 and 105:
    - Structure 104*:* MAE dropped from 4.79 to 2.15, sMAPE from 10.85% to 5.88%.
    - Structure 105*:* MAE decreased from 6.99 to 3.63, sMAPE from 20.38% to 9.38%.
- Consistent accuracy was maintained in structure 103, where both metrics improved slightly.
- Degradations were observed in structure 101, where sMAPE increased from 12.73% to 13.25%.

Key observations for Competitor B:

- Large gains were achieved in structures 103, 104, and 105:

    - Structure 104*:* MAE improved from 11.83 to 1.97, sMAPE dropped sharply from 24.83% to 5.25%.
    - Structure 103*:* sMAPE reduced from 7.64% to 3.74%, nearly halving the relative error.
- Notable degradation occurred in structure 101 (MAE increased from 4.17 to 6.74, sMAPE from 8.20% to 13.49%) and 303 (sMAPE rose from 8.49% to 11.55%).
- Most other structures showed stable or modestly improved performance.

-Conclusion:
The evaluation results demonstrate that the models provide a reliable and accurate foundation for deployment in a production environment. With improved absolute prediction accuracy and generally consistent performance across key product structures, the models are well-suited to support Retailz's pricing strategy. Nonetheless, structure-level variations highlight opportunities for further refinement to maximize predictive performance in all segments. Overall, the models are fit for purpose and offer valuable insights for data-driven pricing decisions.

2.3 Alternatives considered

Alternative models evaluated included:

- **Linear Regression**: discarded due to underfitting and inability to capture non-linear price dynamics.
- **Gradient Boosting**: considered but not selected due to higher complexity and marginal gains compared to Random Forests in initial trials.
- **ARIMA/SARIMA:** not used because the available historical data spanned only 22 months, which was insufficient for reliable time series modeling with these methods.

Random Forests offered a good balance of interpretability, performance, and ease of deployment.

3. Model deployment

3.1 Deployment specifications

The application is implemented as a REST API in Python, using Flask for endpoint handling and Peewee ORM for SQLite database interactions, which stores forecast records and actual prices. The predictive models are dynamically loaded at startup.

Due to size limitations on GitHub, the model files are hosted externally on Hugging Face Hub. A Python script, download_models.py, is responsible for downloading these model files at container startup. This script makes HTTP requests to specific Hugging Face URLs, verifies that the content is valid (not an HTML error page), and saves the files locally under the models directory.

The Docker container is based on the official python:3.11-slim image and automates environment setup. The Dockerfile copies the source code and installs dependencies listed in requirements.txt. Upon container start, it runs the download_models.py script to fetch the model files, then launches the Flask API server.

The API exposes two main endpoints:

- **POST /forecast_prices/**: Accepts a JSON payload with sku (string) and time_key (integer in YYYYMMDD format). The input payload is validated to ensure sku is a non-empty string and time_key is a valid date in the required format. It also checks for duplicates in the database before generating predictions using the loaded models for two competitors. The forecast is saved to the database and returned in the response.
- **POST /actual_prices/**: Accepts a JSON payload with sku, time_key, and actual prices pvp_is_competitorA_actual and pvp_is_competitorB_actual (floats). Validation confirms presence and type correctness of these fields, ensuring the prices are numeric and the date format is valid. The API updates the corresponding database record with these actual prices.

To replicate the deployment:

1. Clone the repository containing the source code and configuration files.
2. Build the Docker image with:

   ```
   bash
   docker build -t forecast-app .
   ```

3. Run the container exposing port 5001:

   ```
   bash
   docker run -p 5001:5001 forecast-app
   ```

4. On startup, the container automatically downloads the model files from Hugging Face and starts the Flask server.
5. Interact with the API via HTTP requests to the defined endpoints for forecasts and actual price updates.

This deployment architecture ensures a reproducible, isolated environment that simplifies distribution and execution in any Docker-supported platform while guaranteeing model availability despite size constraints on source code hosting.

3.2 Deployment performance

During the testing period, the API was subjected to two rounds of requests, distributed across the forecast_prices and actual_prices endpoints, both accessed via **POST**.

- First round:

- **POST /forecast_prices**
  - Total requests: 1000
  - Successful: 999
  - Errors: 1 (0.1%) — due to a SKU with no available historical data
- **POST /actual_prices**
  - Total requests: 500
  - Successful: 500 (100%)
  - Errors: 0

- Second round:

- **POST /forecast_prices**
  - Total requests: 500
  - Successful: 457
  - Errors: 43 (8.6%) — mostly due to invalid SKUs or lack of sufficient historical data
- **POST /actual_prices**
  - Total requests: 500
  - Successful: 497
  - Errors: 3 (0.6%) — caused by malformed requests

- Overall summary:

- Total requests: **2500**
  - Successful: **2453** (98.1%)
  - Errors: **47** (1.9%)

The API showed consistent and reliable performance, with a high overall success rate. Most errors were due to issues with input data (such as invalid SKUs), rather than internal failures of the API.

3.3 Unexpected problems

During deployment, one of the main challenges was managing the large size of the model and historical data files. Due to their substantial size, these files could not be stored directly in the source code repository or on platforms like GitHub. This required using an external hosting solution—in this case, Hugging Face Hub—to reliably store and retrieve these assets at container startup. Implementing a robust download script (download_models.py) to handle this process was essential to ensure smooth initialization.

Another unexpected issue encountered was related to API endpoint formatting: initially, the client requests omitted the trailing slash at the endpoint URL, which caused the server to respond with redirects or errors. Clarifying that the endpoints require a trailing slash (e.g., /forecast_prices/) was necessary to avoid unexpected HTTP errors.

Overall, the deployment was stable and did not crash, but these subtle issues around data management and API usage required troubleshooting to achieve reliable production behavior.

3.4 Known issues and risks

Despite thorough development, the current application and deployed models carry inherent risks and known issues that could impact reliability and accuracy in production.

- Data Integrity and Input Validation:
The API depends on correctly formatted input data, especially the sku identifier and the time_key date in YYYYMMDD format. Malformed or missing fields can cause the application to reject requests or produce errors. Although input validation is implemented, unexpected or malicious payloads could still lead to failures or security concerns.

- Model Generalization and Drift:
The predictive models are trained on historical data frozen at a specific point in time. Over time, changes in market conditions, competitor behavior, or product assortment could reduce model accuracy, causing forecasts to degrade. Without continuous retraining and monitoring, predictions may become unreliable.

- External Dependencies and Model Hosting:
Model files are hosted externally (Hugging Face). Network issues or changes in the hosting URLs could disrupt model downloads during container startup, preventing the application from initializing properly. This dependency introduces a potential single point of failure outside the application control.

- Scalability and Performance:
The current deployment uses a single Flask process with no built-in scaling. Under high load or large-scale usage, performance bottlenecks may emerge, leading to slow responses or timeouts. Proper load balancing and scaling strategies are not yet implemented.

- Logging and Error Handling:
While basic error handling exists, more comprehensive logging and alerting mechanisms are needed to detect anomalies, failures, or degraded model performance proactively.

In summary, the system requires ongoing monitoring, robust validation, retraining pipelines, and infrastructure enhancements to mitigate these risks and ensure reliable production use. There is no risk-free deployment; understanding and addressing these limitations is crucial.

4. Learnings and future improvements

To enhance the current forecasting application and better meet the client's needs, several clear improvements and exploratory initiatives can be implemented.

1. Model Retraining and Monitoring:
Currently, the model has been retrained manually using actual price data received through the "actual prices" endpoint. However, this process is not automated. Implementing an automated retraining pipeline would allow the model to continuously update with the latest data, improving accuracy and adaptability to market changes.

2. Input Validation and Data Quality Control:
Strengthening input validation mechanisms can reduce errors caused by malformed or unexpected data, increasing system robustness. Additionally, incorporating data quality checks and alerts will help identify inconsistencies or anomalies early, ensuring more reliable forecasts.

3. Scalable and Resilient Infrastructure:
The system already uses PostgreSQL hosted on Railway, providing a scalable and reliable database. The next step could focus on application orchestration via containers (Docker), ensuring better horizontal scalability, fault tolerance, and easier maintenance.

4. User Experience Improvements and API Flexibility:
Enhancing the API to support batch requests, partial updates, or asynchronous processing could improve efficiency and user satisfaction. Adding comprehensive API documentation and client SDKs would facilitate easier integration by end users.

5. Data Enrichment and Feature Expansion:
Incorporating additional data sources — such as promotional calendars, seasonal indicators, or competitor pricing strategies — can improve forecast quality. Exploring advanced feature engineering or alternative modeling techniques (e.g., time series models) may yield more accurate results.

6. Robust Security and Compliance Measures:
As the system scales, addressing data privacy, secure access, and compliance with relevant regulations will become increasingly important, ensuring client trust and legal adherence.

Throughout the entire process, several important learnings emerged. Managing large model and data files required hosting them externally on Hugging Face instead of GitHub, which introduced added complexity around downloading and version control. Strict payload validation proved essential to prevent processing errors and maintain data integrity throughout the application. The retraining of the model with actual price data was performed manually, underscoring the need to automate retraining for continuous improvement and adaptability. Small technical details, such as including trailing slashes in API endpoints, affected client requests and highlighted the importance of thorough and consistent API documentation.

5. Annexes

These charts show the relationship between the actual observed prices and the prices predicted by the model**.** The reference line (red dashed) represents the ideal case where the predicted price exactly matches the actual price (y = x). Points close to this line indicate accurate predictions, while deviations represent errors.

Fig1 - Relationship between the actual observed prices and the prices predicted by the model for Competitor A before retraining.

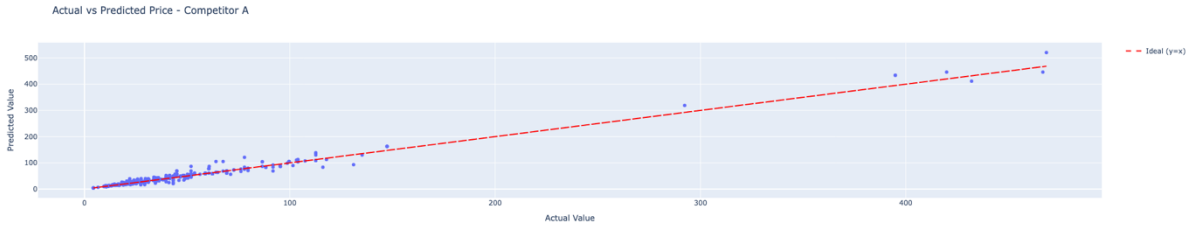

Fig2 - Relationship between the actual observed prices and the prices predicted by the model for Competitor A after retraining.

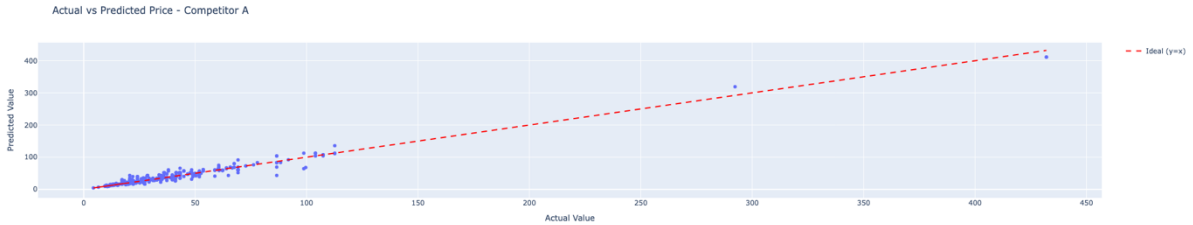

Fig 3 - Relationship between the actual observed prices and the prices predicted by the model for Competitor B before retraining.

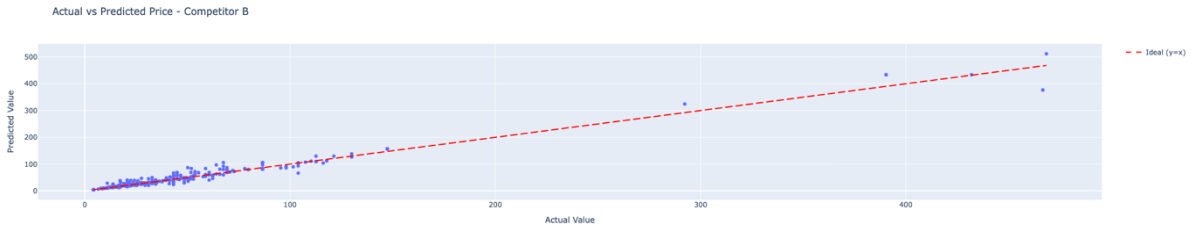

Fig 4 - Relationship between the actual observed prices and the prices predicted by the model for Competitor B after retraining.