Research Report: E-commerce Session Purchase Prediction

Universität St. Gallen, Big Data Analytics

Erica Trofimov Luana Borma Brugger Matteo Marello
Yardh Vilgot Göran Berglund

15 December 2025

1. Introduction

1.1 Context

The e-commerce industry has witnessed exponential growth over the past decade, transforming how consumers shop and interact with brands. With more people engaging in online shopping across a variety of platforms, businesses are faced with an overwhelming volume of data generated from customer interactions. According to a 2020 McKinsey report, the pandemic accelerated digital adoption, with online shopping becoming the primary channel for many consumers, as the crisis compressed years of e-commerce growth into a matter of months (McKinsey, 2021).
As a result, understanding consumer behaviour in the online environment has never been more critical. The ability to analyse customer behaviour not only helps companies optimize the shopping experience but also drives strategic decisions in areas such as marketing, customer service, and product offerings. The huge amount of behavioral data generated during online shopping sessions contain valuable insights into consumer intentions, preferences, and purchasing patterns. Every click, view, add-to-cart, and time spent on a website contributes to a detailed picture of how customers interact with a brand's digital interface. Analysing this session-level data can reveal patterns that drive conversions, allowing businesses to improve their targeting strategies, streamline the customer journey, and ultimately increase sales.
According to Dave Chaffey and Fiona Ellis-Chadwich in their book Digital Marketing: Strategy, Implementation and Practice, a well-structured understanding of consumer behaviour in e-commerce can significantly enhance a company's competitive advantage by tailoring marketing and product strategies to meet customers' needs more effectively (Chaffey & Ellis-Chadwich, 2000).

This project explores customer behaviour within the context of an online multi-category store, using session data to predict whether a session will result in a purchase. By examining various metrics such as time spent on the site, product views, and cart interactions, the study seeks to find out key behavioural patterns that influence purchasing decisions. Insights from such analyses can be used to improve not only the website's user interface but also the marketing campaign aimed at retaining customers and encouraging repeat purchases. The importance of this research is in its ability to contribute to the growing field of data-driven decision-making within e-commerce. In a time where customer preferences are quickly changing and there is high competition, understanding how different customer segments engage with online stores is valuable. As businesses shift from traditional marketing strategies to more personalized, data-driven approaches, the role of behavioral analytics becomes even more clear. By leveraging behavioral data to predict purchasing outcomes, businesses can improve their strategies, targeting customers more effectively and optimizing the overall shopping experience.

In conclusion, this project is timely and crucial as e-commerce continues to evolve. By uncovering key patterns, it seeks to enhance e-commerce strategies, optimize user experiences, and contribute to more effective, data-driven decision-making in an increasingly competitive online marketplace.

1.2. Research Questions

The central objective of this project is to understand and model customer behavior within an e-commerce environment using session-level data. To guide this investigations, the main research question is: **Can we predict whether an online session will end in a purchase based on behavioral features observed during that session?** To address this, the data is aggregated by *user_session* and associated with metrics such as number of events, product views, cart additions, and time spent. These variables form the basis of a predictive model designed to identify patterns associated with successful conversions.

1.3 Data Source

The analysis in this project is based on the Ecommerce Behavior Data from Multi-Category Store dataset, publicly available on Kaggle. The dataset contains seven months of behavioral logs collected through the Open CDP project from a large multi-category online retailer, the analysis will be conducted only in the month of November, 2019. It is a large-scale dataset of approximately 9GB, reflecting the high volume of interactions typical of modern e-commerce platforms. Each row represents a user event, capturing interactions such as product views, cart additions, and purchases.

2. Data Collection and Data Storage

The dataset used in this project was approximately 10 GB and was originally stored on Kaggle. Working with it in that format introduced two key challenges. First, processing a CSV file of that size is both slow and inefficient, particularly when analyses need to be repeated. Second, data stored on Kaggle cannot be accessed directly from R or other external systems without first downloading it locally, an approach that runs counter to the principles of big data workflows. To address these limitations, we implemented a two-step strategy. We began by converting the dataset into Parquet format directly on Kaggle, significantly improving its efficiency and usability. We then uploaded the resulting Parquet files to a Google Cloud Storage (GCS) bucket, enabling fast and scalable access through tools such as Arrow, BigQuery, and various distributed computing environments.

2.1 Why Parquet and Cloud Storage?

Benefits of Parquet over CSV

Parquet is a columnar data format, in contrast to CSV, which is row-based. This structural difference results in several important advantages. Parquet files are typically five to ten times smaller than their CSV counterparts, making storage far more efficient. Their columnar layout also enables much faster read operations, particularly for analytical tasks that require only a subset of columns. In addition, Parquet offers superior built-in compression, further reducing the overall memory footprint. Because of these features, modern analytical tools such as Arrow, Spark, and DuckDB are optimized to work natively with Parquet, allowing workflows to run more efficiently and at scale.

Benefits of Google Cloud Storage (GCS)

Moving the dataset to GCS offers several key benefits. Because the data can be accessed directly in the cloud, there is no need to download the files locally, eliminating disk space limitations and simplifying workflow management. GCS also provides flexible access controls, making it easy to share the dataset securely with collaborators. In addition, it integrates natively with tools such as Arrow, BigQuery, and R, enabling efficient, cloud-based data processing. Altogether, these advantages make GCS a scalable and sustainable long-term solution for managing large analytical datasets.

Step 1: Data Transformation on Kaggle. Because Kaggle notebooks can read the dataset directly from Kaggle's internal storage, we were able to perform the entire conversion process within the Kaggle environment, avoiding the need to download the raw 10 GB CSV file. To handle the data efficiently, we used a Python script (for more info see code/file_conversion_uploading.py) that loaded the CSV in manageable chunks, converted each chunk into Parquet format using pyarrow, and then saved the results as .parquet files. Kaggle's built-in support for writing Parquet files from notebooks further streamlined this process, making the overall workflow both straightforward and scalable.

Step 2: Uploading the Parquet File to Google Cloud Storage Because Kaggle cannot authenticate directly with a Google Cloud account, we created a dedicated service account in GCP to manage access. This service account was assigned the Storage Object Admin role, which provides the necessary permissions to upload files to Google Cloud Storage. After setting it up, we generated a JSON key associated with the service account and uploaded it into the Kaggle notebook environment, enabling secure programmatic access to the GCS bucket.

2.3 Outcome

By converting the dataset to Parquet and storing it in Google Cloud Storage, we achieved several important improvements. The dataset size was drastically reduced, making storage and transfer far more efficient. Accessing the data also became significantly faster, enabling smooth loading and analysis in R, Python, Arrow, Spark, and BigQuery. Because the data now resides entirely in the cloud, we avoided local disk constraints and established a centralized, scalable storage solution. In addition, managing permissions at the bucket level made it easy to share the dataset securely with collaborators.

In conclusion, this procedure effectively resolved the performance and accessibility challenges associated with handling a 10 GB dataset stored on Kaggle. Converting the data to Parquet greatly improved processing efficiency, and uploading it to Google Cloud Storage provided a scalable, cloud-based environment that supports collaboration and future analytical workflows.

3. Data Cleaning and Preparation

The complete code for the data analysis is available in the `Code/Cleaning.R` script.

Our raw data, which is around 10 GB and >67 million rows, is stored as a Parquet file in GCS, which solves local storage constraints. Still, the size poses challenges for cleaning and aggregation because it cannot be loaded into memory as a standard data frame.

3.1 Out-of-Memory Calculations

The first bottleneck is related to memory usage prior to the cleaning. This constraints forced us to keep most of the operations outside of R's memory and only collect a compact dataset with the absolute necessities for our prediction model. First, we load the Parquet file from GCS into an Arrow dataset which allows us to run futher data processing without fully loading the file into the memory. Second, since purchase events are so rare relative other events such as view, we construct a deduplicated "purchase" table as this reduces the amount of data that needs to be processed during our heavy operations. We later join this small table into the session-level features.

The cleaning pipeline was built using the arrow package which is an out-of-memory framework that can open and query Parquet data lazily, meaning the system does not fetch the data until a specific trigger is activated. Instead of reading the entire dataset into memory, this allows us to select, filter, group and summarize only using the necessary columns and rows. Heavy operations such as counting views, items put in the cart, average prices and deriving session start and end times are performed outside of the R memory.

3.2 In-Memory Functions

A further challenge arises as some of the time-based transformations to calculate session duration and our dichotomous variable of is_weekend and is_beginning_of_month are not supported by the lazy backend. As a result, these operations cannot be executed in the out-of-memory environment, and we first must collect the data into R.

To address this issue, we deliberately push the collection as late as possible and drop all columns that are no longer needed. After the collection, we apply the remaining operations for our time-related features and immediately remove the intermediate variables we no longer need.

Structuring our cleaning pipeline this way allows us to perform the heaviest aggregations out-of-memory and only perform the absolute necessary operations once collected. The final cleaned session_features dataset used in our prediction model is substantially smaller and uses only about 500 MB of memory, a substantially smaller size than the orgininal dataset.

3.3 Summary statistics

Inspect the object

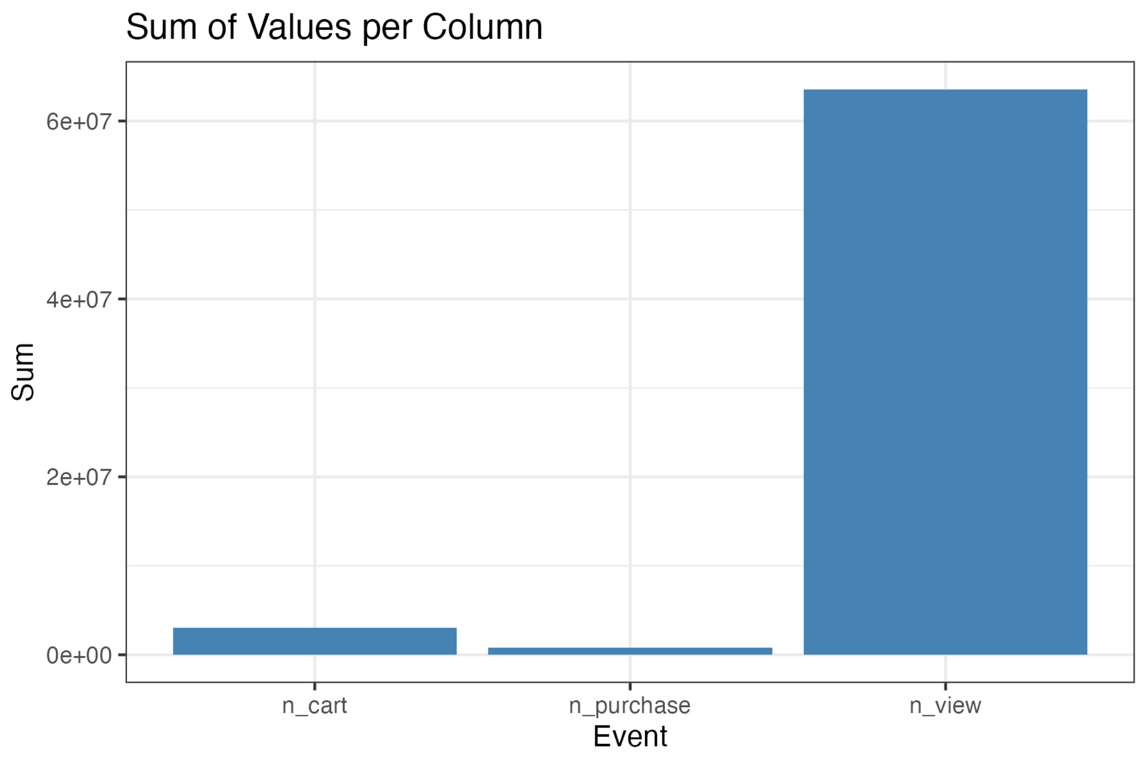

Figure 1: ***Figure 3.1***

The number of events is highly skewed towards number of views with only a small number of items put in cart and purchases.

Unique categories

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
  1.000   1.000   1.000   1.356   1.000  99.000
```

The number of unique categories is very skewed toward 1.

Avg price

```
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00   81.05  186.28  311.35  396.14 2574.07
```

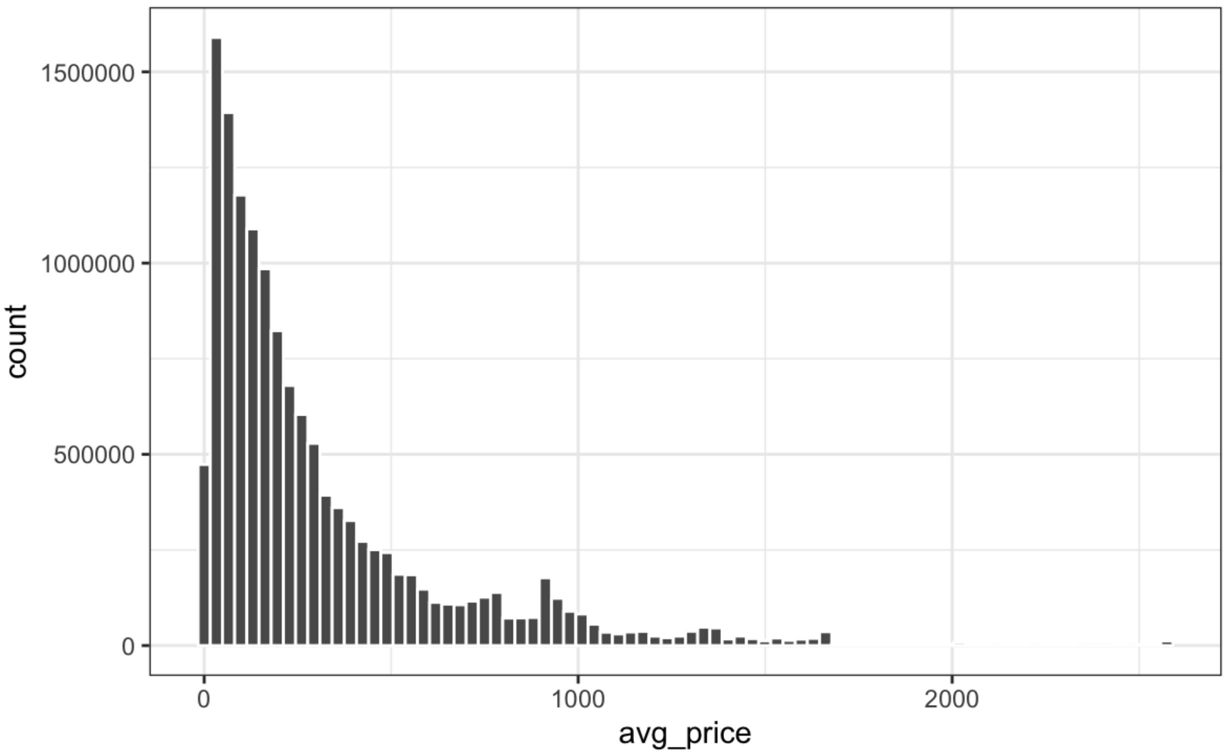

Figure 2: ***Figure 3.2***

Most observations of our average price variable fall at the low end of the price range. As prices increase the frequence drops producing a right-skewed tail. Overall, most of the observations lie within the 80-400 range.

Session Duration

```
   Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
    0.00     0.00     1.00    15.52     4.75 42303.73
```

The session duration feature is very skewed. A lot of observations are 0, while we also have outliers with extremly high values. For this reason we plot the session_duration + 1 and use a log scale to still be able to visualize this in a matter that makes sense.

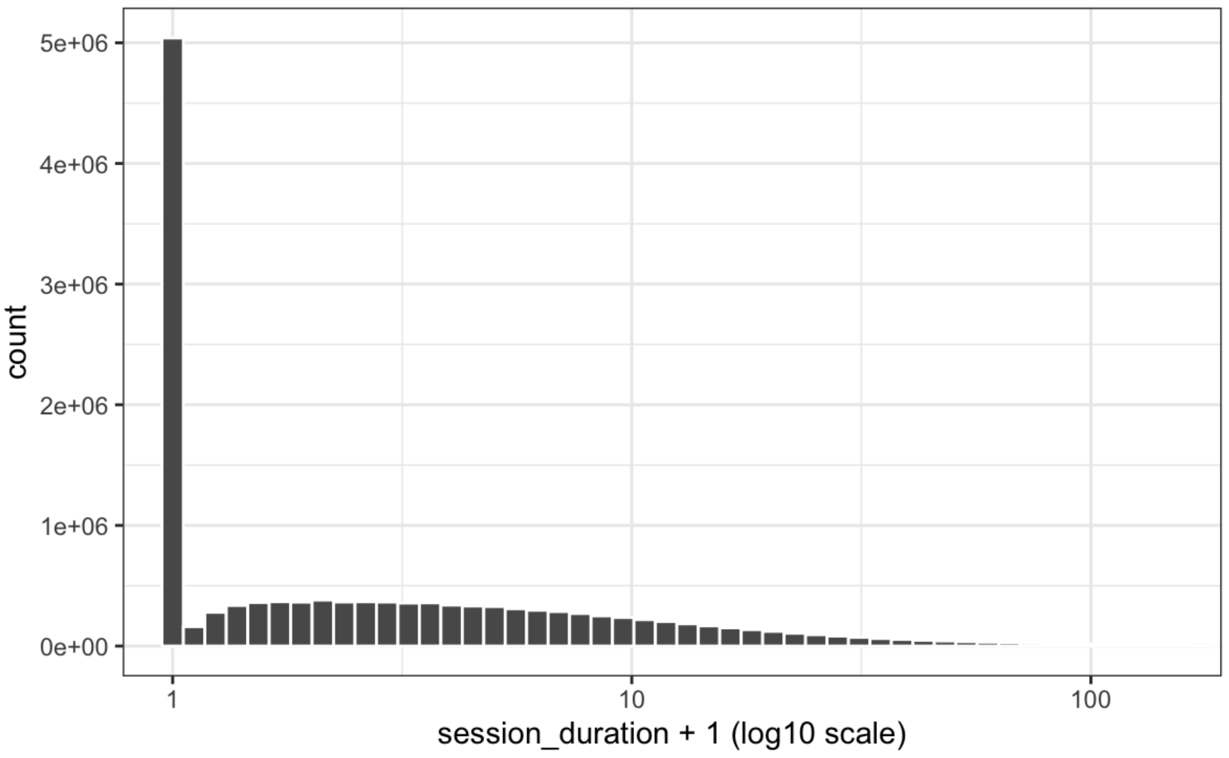

Figure 3: ***Figure 3.3***

4. Data Analysis and Data Visualization

4.1 Explorative Data Analysis

The complete code for the data analysis is available in the `Code/Analysis.R` *script.*

We begin by examining our session feature dataframe before proceeding to model construction. The reduced target dataset we will use occupies 525.5 MB of memory, which is manageable within R. With the command `summary`, we also get useful information and statistics about each variable.

```
                     | Min| 1st Qu.| Median|   Mean| 3rd Qu.|      Max|
|:---------------------|---:|-------:|------:|------:|-------:|--------:|
|n_view                |   0|    1.00|   2.00|   4.61|    5.00|  4128.00|
|n_cart                |   0|    0.00|   0.00|   0.22|    0.00|   709.00|
|avg_price             |   0|   81.05| 186.28| 311.35|  396.14|  2574.07|
|n_unique_categories   |   1|    1.00|   1.00|   1.36|    1.00|    99.00|
|n_purchase            |   0|    0.00|   0.00|   0.06|    0.00|     1.00|
|session_duration      |   0|    0.00|   1.00|  15.52|    4.75| 42303.73|
|is_weekend            |   0|    0.00|   0.00|   0.33|    1.00|     1.00|
|is_beginning_of_month |   0|    0.00|   0.00|   0.07|    0.00|     1.00|
```

Our dataset exhibits a ‘Big N’ complexity: the number of observations (sessions) is very large relative to the number of features of the data.
Proceeding with the analysis, we first compute a correlation matrix to assess relationships between features and identify any multicollinearity. This analysis informs our model selection and feature engineering decisions.

Next, we explore the dataset to determine whether it is imbalanced. The target variable, `n_purchase`, is binary, and understanding the degree of class imbalance is crucial for building an accurate model and avoiding biased predictions or overfitting. Hence, we also compute the proportion between the number of success of purchase (`n_purchase` = 1) and the number of unsuccesfull purchases, and visualize this using a barplot. We obtain that the dataset is quite imbalanced, with only about 5% of sessions resulting in a purchase. We then remove the computed object from the global environment in order to free up memory space.

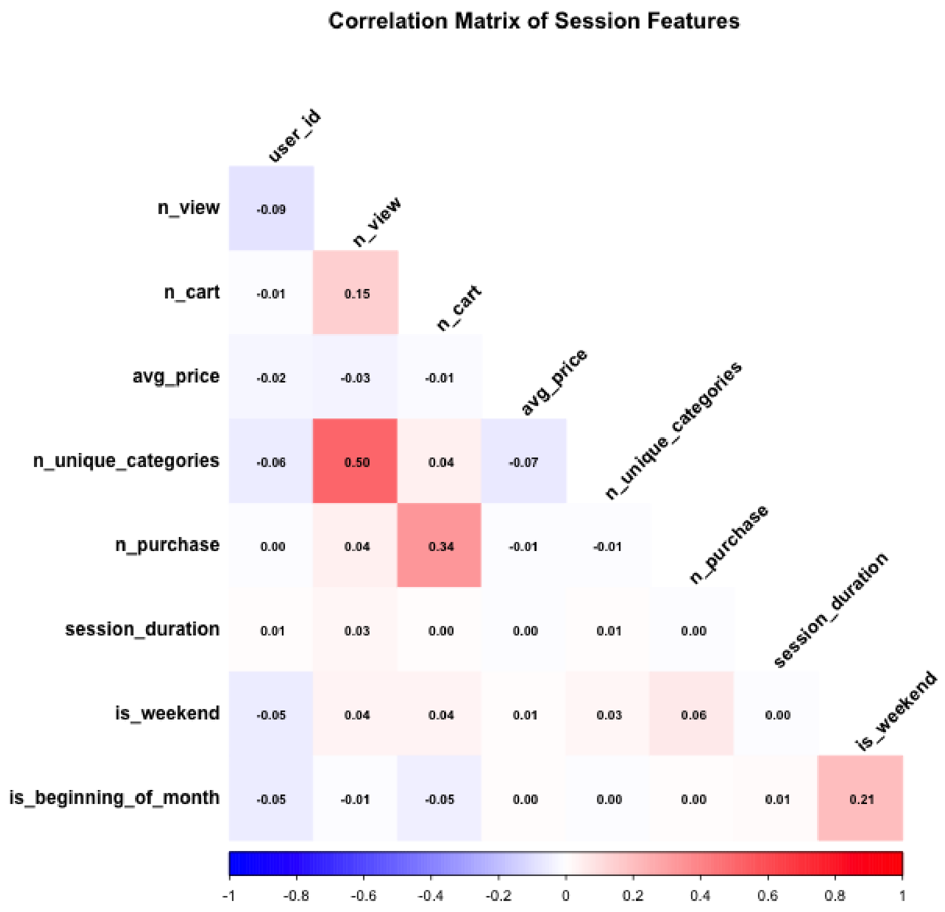

Figure 4: ***Figure 4.1*** *Correlation Matrix between all the features of interest. Blue blocks outline negative relationships, red blocks outline positive correlation, while white and light blocks outline quasi-uncorrrelated features. Most relationships are weak, with only moderate correlation observed between n_view and n_unique_categories, and between n_cart and n_purchase. Overall, the feature set does not exhibit strong multicollinearity.*

Although data exploration is usually less memory-intensive than modelling, we still emphasize memory efficiency by removing unnecessary intermediate objects (rm()) and storing visual outputs as PNG files rather than keeping large objects in memory.

4.2 Modelling

We can now proceed to build a predictive model that identifies which variables have the greatest impact on purchases and, ideally, predicts the probability of a purchase for unseen data.

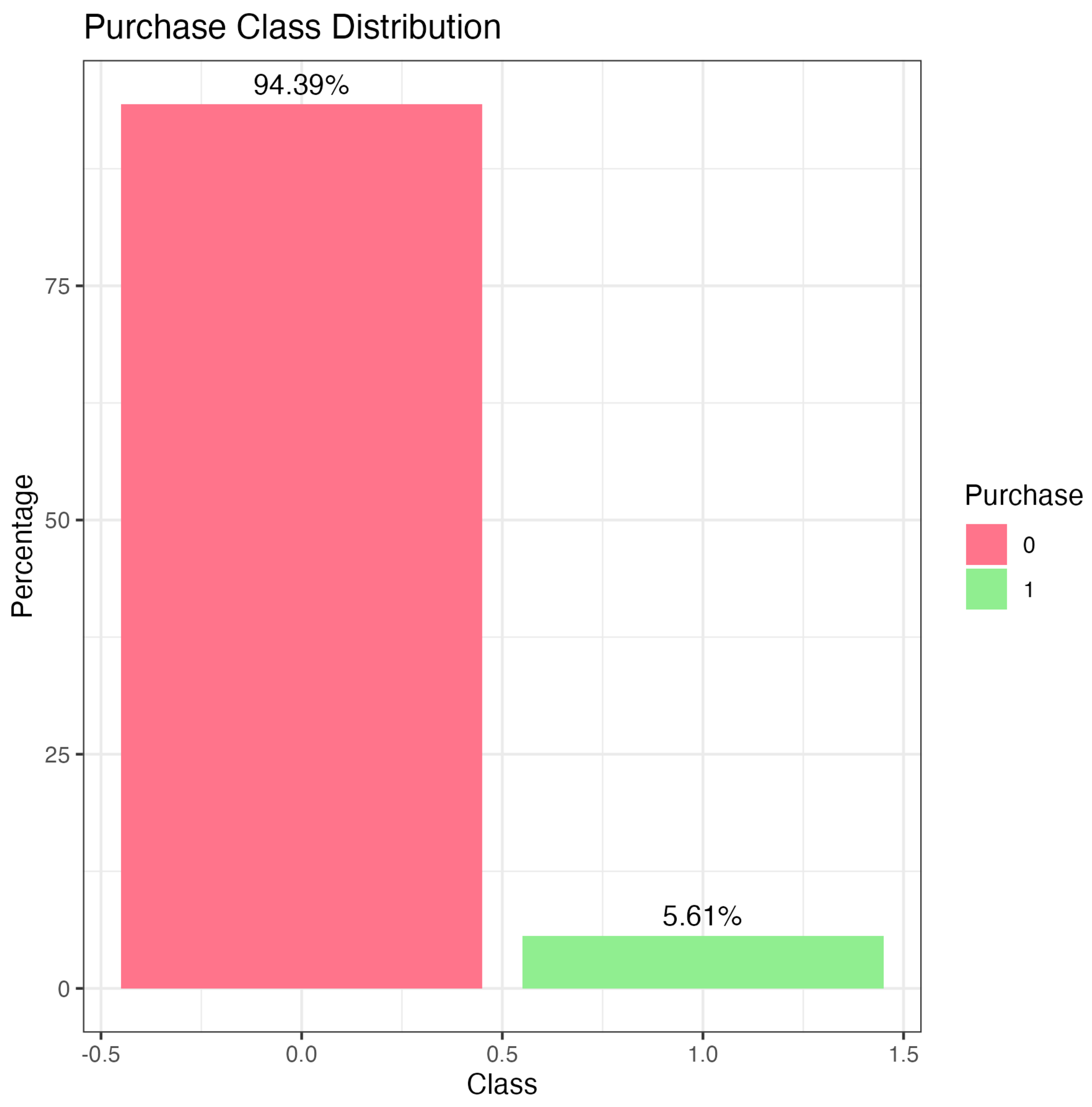

Figure 5: ***Figure 4.2*** *Barplot of the proportion of sessions with unsuccessful purchases (pink) versus successful purchases (green). The plotted percentages reveal a clear class imbalance, with non-purchasing sessions representing the majority of the dataset.*

4.3 Ridge Regression

As a first model, we choose to employ a **Ridge logistic regression.** This choice is reasonable for several reasons:

Firstly, Our dataset is highly imbalanced, with only about 5% of observations corresponding to successful purchases. A simple logistic regression could achieve high overall accuracy by predicting mostly “no purchase” events, but it would likely perform poorly on predicting actual purchases, resulting in a misleading assessment of model performance. A Ridge regression addresses this imbalance by assigning higher importance to the minority class, improving predictive accuracy for purchase events.
Another concern is multicollinearity. Ridge regression can mitigate this issue by shrinking correlated coefficients, preventing instability in the estimates.
Ridge regression also helps reduce model complexity by penalizing less significant features. In this case, as we are dealing with a Big N problem, Ridge is not probably life changing in terms of reducing complexity, but it helps prevent overfitting and improves generalization. Additionally, focusing on the most relevant variables can improve memory efficiency and computational speed, which is particularly beneficial for subsequent modeling with more complex machine learning or deep learning algorithms.

We implement the Ridge regression model using the `biglasso` R package [Zeng and Breheny, 2021]. As the author of this package empirically demonstrates, `biglasso` is optimized for large datasets and can handle data that exceeds available memory by utilizing memory-mapped files. Among other algorithms, this is the most memory efficient one.
Using `caret`’s `createDataPartition`, we perform a stratified random split to create training and testing sets, which is helpful with model performance for our highly imbalanced datasets.

An important step to assess memory usage and computational time regards the matrix preparation. `Biglasso` requires the input data to be in a big.matrix format, which is optimized for memory efficiency and speed. Moreover, employing `filebacked.big.matrix`, we can store the matrix on disk rather than in memory, further reducing memory consumption during model training.

Next, we set up the cross-validation procedure to tune the penalty parameter, λ, which controls the amount of regularization applied to the model, in order to find the parameter that minimizes prediction error. We denote that, while parallel computing could speed up the process, it may also increase memory usage, which can be a limitation for very large datasets, hence we avoid it. However, `biglasso` also supports parallel processing if needed. We also specifiy that this package already has a built in standardization of the features, hence we do not need to standardize them manually. Finally, we plot the cross-validated deviance to visualize the performance across different lambda values and extract the coefficients corresponding to the optimal lambda.

Once we fitted the model we can test the Ridge regression on unseen data to evaluate its predictive performance. Using accuracy as evaluation metric would be misleading, given the class imbalance of the dataset. Hence, we assess the model using ROC-AUC, which measures its ability to distinguish between purchases and non-purchases. Later, we will also analyse the F1-score, which balances precision and recall and is appropriate for our imbalanced dataset.

The visual results and evaluation metrics are discussed further in the Results chapter.

The most challenging aspect of implementing the Ridge regression model was managing memory usage during training and cross-validation on our local machine. At first, we employed the `glmnet` package, which is widely used for regularized regression. However, we encountered significant memory constraints due to the size of our dataset.
To overcome this, we switched to the `biglasso` package, but switching to a `binomial` family for logistic regression in `biglasso` presented additional challenges. Hence, we decided to decrease the number of folds in cross-validation from 5 to 4, which reduced memory consumption during model fitting, and the number of lambda values tested from 100 to 40.
This adjustment allowed us to successfully train the model without exceeding memory limits, albeit at the cost of slightly less robust hyperparameter tuning.

4.4 XGBoost

As a second predictive model we select an XGBoost, a Machine Learning algorithm based on decision trees, which can handle non-linear relationships and interactions between features. We employ the `xgboost` package, whose algorithm has been implemented in C++. Hence, it provides speed efficiency and is heavily used in large-scale machine learning.
To begin, we introduced a critical step to address class imbalance by calculating the `scale_pos_weight` parameter, which adjusts the weight of positive class instances during training. As a second step, we prepare the data in the DMatrix format, which is optimized for XGBoost. Then, in setting our model parameters, we specified the `hist` tree method, which is designed for large datasets. An important aspect of our training process is the inclusion of early stopping based on AUC performance on a validation set.
This way, we prevent overfitting and reduce unnecessary computation time, making the process stop when no improvement in the AUC is observed over 20 rounds. Finally, for evaluation, we again use ROC-AUC and, later on, the F1-score, to assess the model's predictive performance on unseen data.

5. Results and Discussion

5.1 XGBoost Feature Importance

To better understand which behavioral variables drive purchase predictions, we computed feature importance for the XGBoost model using `xgb.importance` and plotted the top predictors ranked by Gain, i.e., the improvement in model accuracy attributable to splits in each variable. ***Figure 5.1*** displays the 10 most important predictors according to this metric.

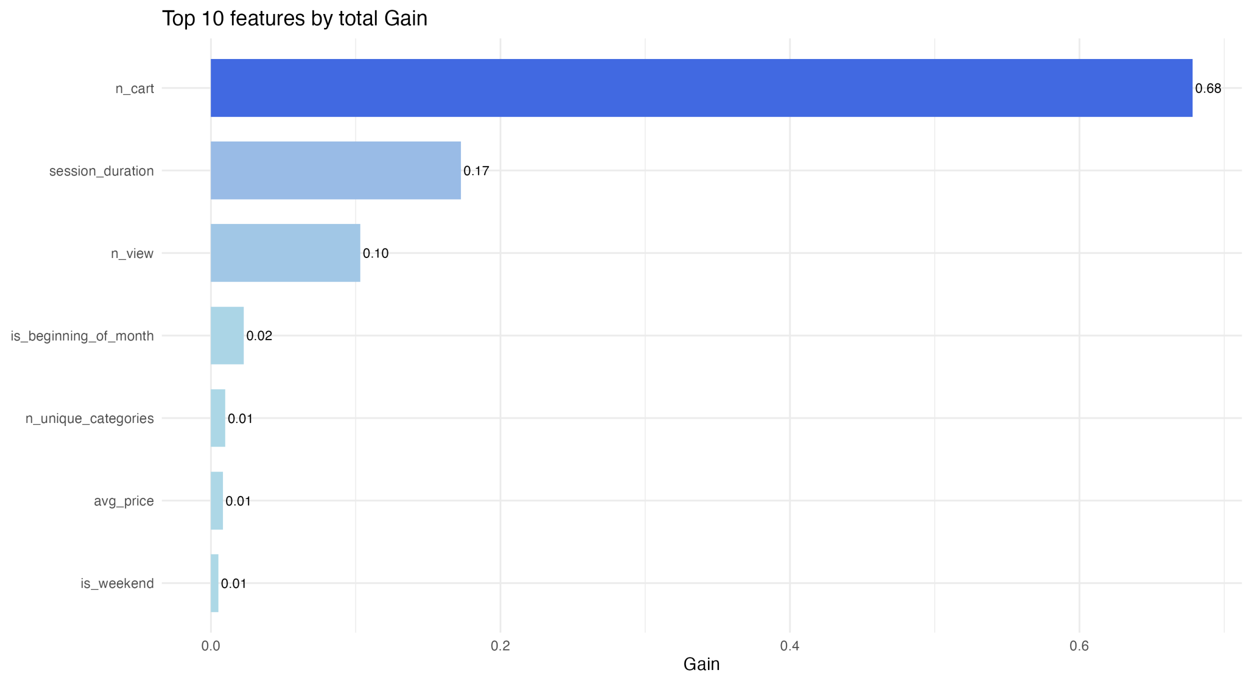

Figure 6: ***Figure 5.1*** *The figure plots the relative contribution (in %) of each feature to the XGBoost model's predictive performance, measured by Gain.* `n_cart` *(number of cart events) is by far the most influential feature (~69% of total Gain), followed by* `session_duration` *(~16%) and* `n_view` *(~10%). The remaining variables (*`is_beginning_of_month`*,* `n_unique_categories`*,* `avg_price`*,* `is_weekend`*) have comparatively smaller, but non-zero, contributions.*

The pattern in Figure 5.1 shows that in-session intent signals are central for predicting purchases. Sessions with more cart events, longer duration, and more product views are those the model associates with higher purchase likelihood. Temporal variables and price-related information add some incremental predictive power but are clearly secondary.

5.2 Partial Dependence Analysis (PDP)

To further interpret how cart activity relates to the predicted purchase probability, we compute a partial dependence profile for `n_cart` using the DALEX package. Due to computational constraints, this profile is estimated on a random subset of 5000 sessions, which is sufficient to obtain a smooth curve while reducing memory and plotting time. The resulting partial dependence plot is shown in **Figure 5.2.**

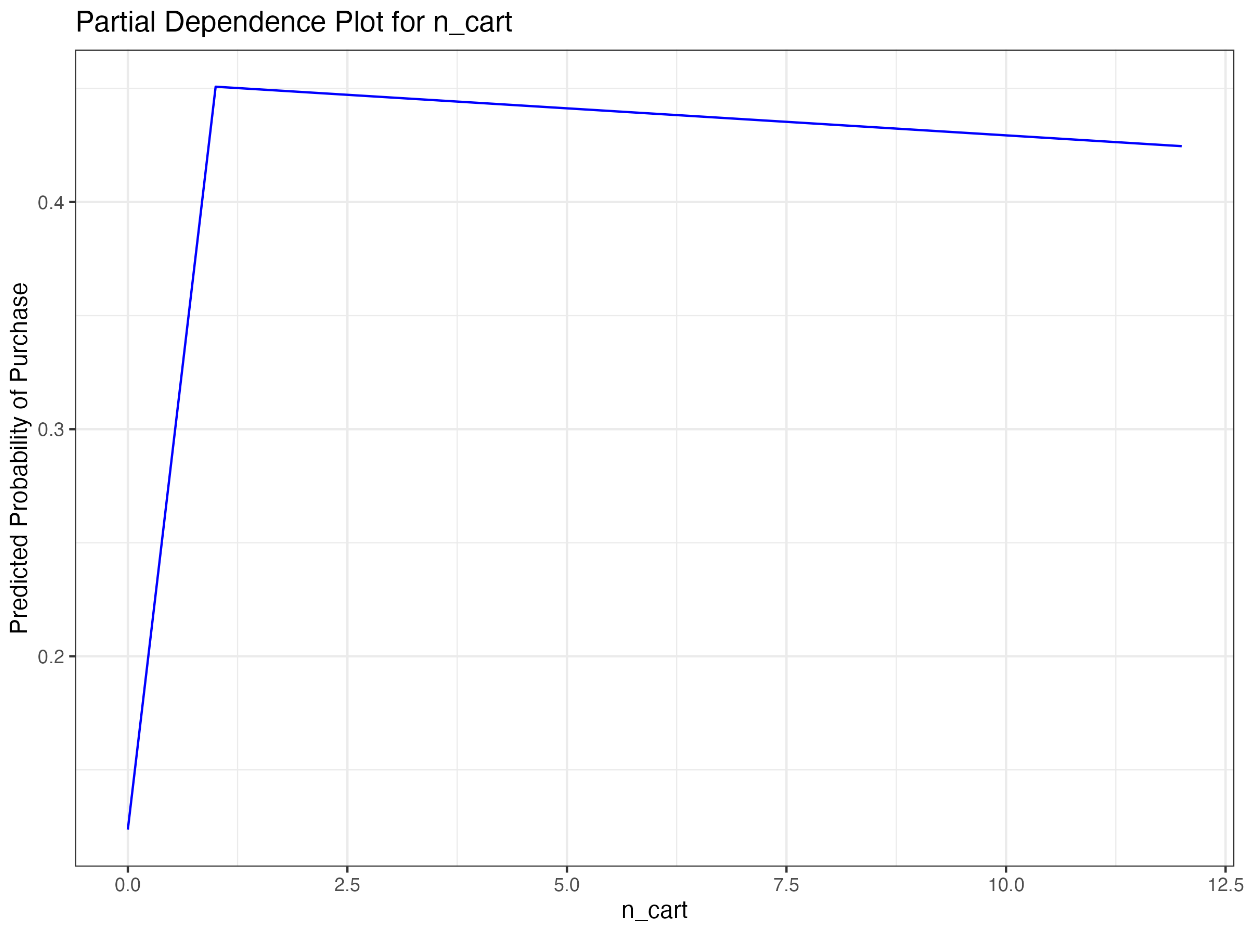

Figure 7: ***Figure 5.2.*** *The curve shows the average predicted probability of purchase as* `n_cart` *varies, holding other features at their observed values (on a 5,000-session subsample). The probability rises sharply from 0 to 1 cart event, (peaking slightly above 0.4) and then declines gradually as* `n_cart` *increases.*

Figure 5.2 indicates that the model associates the highest purchase probability with sessions that contain exactly one cart event, while additional cart actions are linked to a gradual decrease in predicted purchase likelihood.
This suggests that a single, decisive cart action differentiates many converting sessions from non-converting ones, whereas sessions with many cart events are more likely to involve indecision or cart abandonment. This behavioral pattern refines our answer to the research question by showing that not only the presence of cart activity matters, but also its intensity.

5.3 Model Comparison

Because the purchase outcome is highly imbalanced, accuracy alone would be misleading. We therefore compare Ridge and XGBoost using ROC-AUC, F1-score, and the confusion matrices for the purchase class. We also track training time and peak memory usage, which are summarized in a comparison table together with the performance metrics. Overall, XGBoost achieves better discriminative performance, while Ridge remains more lightweight in terms of computation.

```
=== PERFORMANCE DIFFERENCES ===
AUC Difference (XGB - Ridge): 0.071
Training Time Difference (XGB - Ridge): 5.05 minutes
```

ROC-AUC Analysis

To assess how well each model separates purchase from non-purchase sessions, we compute ROC curves on a 30,000-session test subsample, due to computational constraints, and plot both curves in **Figure 5.3.**

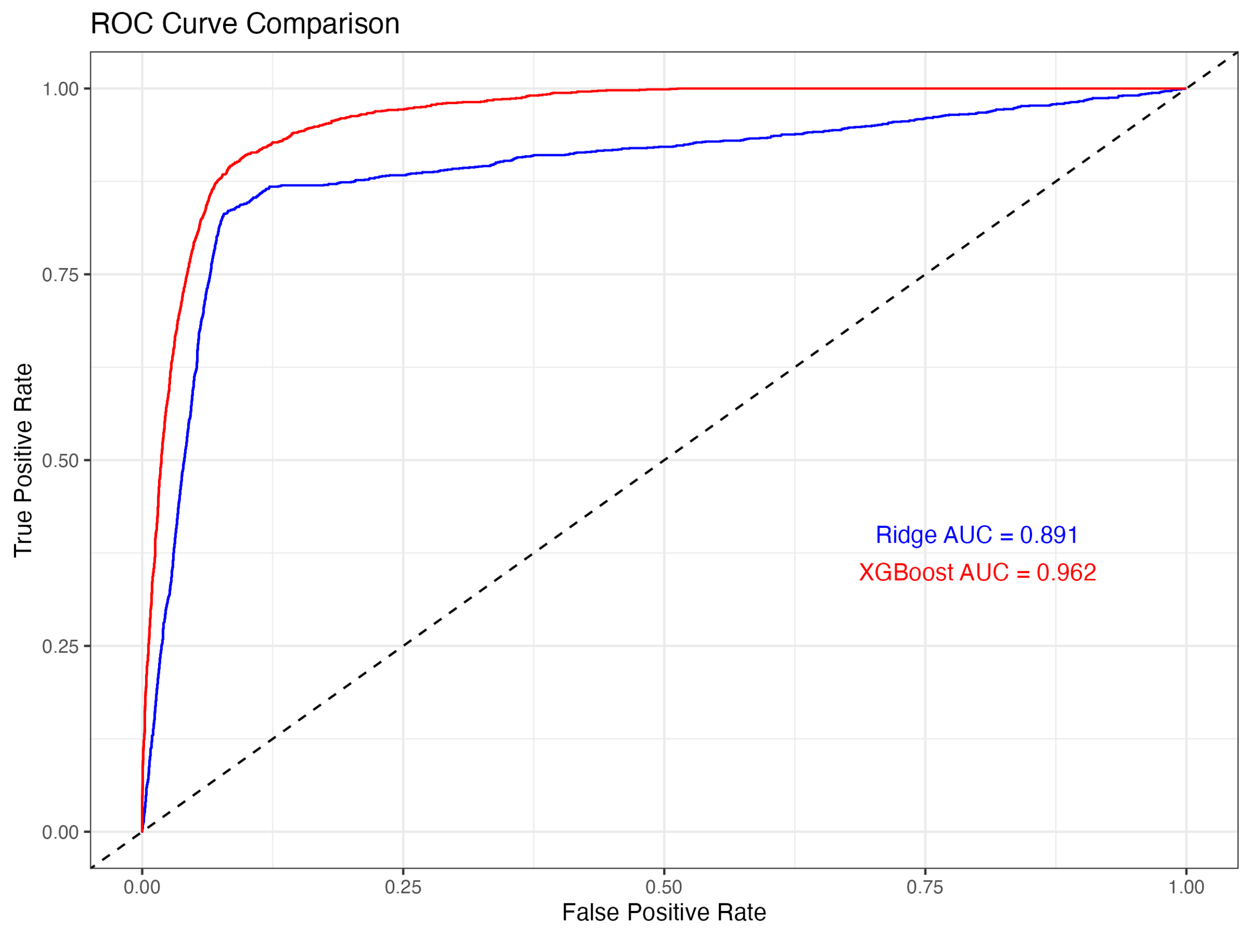

Figure 5.3. *The figure shows ROC curves evaluated on a 30,000-session test subsample. The red curve (XGBoost, $AUC \approx$ 0.96) lies consistently above the blue curve (Ridge, $AUC \approx$ 0.89), indicating better discrimination between purchase and non-purchase sessions across almost all false positive rates. The dashed diagonal represents random guessing (AUC = 0.5)*

Both models perform well in distinguishing sessions that end in a purchase from those that do not, but XGBoost clearly dominates Ridge in terms of ROC-AUC. This supports the use of a non-linear, tree-based model to capture more complex interactions between behavioral features when answering our research question.

Confusion Matrices

To examine classification errors in more detail, we compute confusion matrices for both models on the full test set. From that, we also compute the f1-score as a second metric to assess and compare the two predictive model's performance, and visualize the final comparison table.

```
Table: Ridge Regression Confusion Matrix

|   |       0|      1|
|:--|-------:|------:|
|0  | 5161789| 279007|
|1  |   38915|  30948|

Ridge True Positive Rate: 0.1
Ridge True Negative Rate: 0.993

Table: XGBoost Confusion Matrix

|   |       0|      1|
|:--|-------:|------:|
|0  | 4714989|  27931|
|1  |  485716| 282024|

XGBoost True Positive Rate: 0.91
XGBoost True Negative Rate: 0.907
```

For each model, the table reports true negatives (predicted 0/ actual 0), false negatives (predicted 0, actual 1), false positives (predicted 1/ actual 0), and true positives (predicted 1/ actual 1). Ridge produces relatively few false positives but also detects only a small fraction of purchase sessions. XGBoost, in contrast, identifies a much larger number of true purchases, at the cost of more false positives.

These results highlight a key trade-off. Ridge is conservative: it predicts "no purchase" for most sessions, which yields high overall accuracy but a low recall and F1-score. XGBoost, instead, captures many more purchase events, improving recall and F1-score at the expense of misclassifying more non-purchase sessions as purchases. In an e-commerce setting where the goal is to identify likely buyers (for targeting promotions or interventions), this higher recall is typically more valuable than marginal gains in overall accuracy.

```
Table: Final Model Performance Comparison

|Model            |Training_Time_Seconds | Memory_Usage_MB|       AUC| F1_Score|
|:----------------|:---------------------|---------------:|---------:|--------:|
|Ridge Regression |3.490 mins            |     126.0700302| 0.8908006|    0.970|
|XGBoost          |8.543 mins            |       0.8130722| 0.9617946|    0.948|
```

5.4 Dealing with memory constraints

Consistent with the big-data constraints of this project, we keep the evaluation phase feasible by:

- computing ROC curves on a 30,000-session subsample rather than the entire test set, and
- removing heavy intermediate objects (e.g., explainers, PDP objects, importance matrices) from the R environment after use.

This allows us to complete model comparison on millions of sessions while staying within the memory limits of a standard local machine.

5.5 Limitations and future work

Some limitations emerged during the analysis. First regarding data quality and scope, the dataset comes from a single multi-category retailer and covers only November 2019. Behavior is therefore retail-specific and season-specific, which may bias the models toward patterns that do not generalise to other periods or platforms. We also only observe session-level behavioural logs, so we cannot distinguish new from returning customers, identify bots, or control for promotions and marketing campaigns, even though these factors may affect purchase probabilities.

Second, we faced computational limitations linked to the dataset's size. To fit models on a standard local machine, we relied on Parquet storage, Arrow and memory-mapped structures, and we frequently worked with subsamples (for example, 5,000 sessions for PDPs and 30,000 for ROC curves). While this keeps the analysis tractable, it introduces approximation error and may smooth over rare but informative patterns.

Third, the highly imbalanced outcome may still bias the models despite the use of class weights and appropriate metrics. These limitations suggest several possibilities for future work, including sequence-based models, richer covariates, alternative imbalance strategies and robustness checks across different months, retailers and subsamples.

6. Scaling and Cloud Deployment

As the dataset grows, several bottlenecks may hinder the scalability and efficiency of our workflow. Below we outline the three main issues and describe how cloud-based solutions can address them.

6.1 Problems

1. Data Transfer From Kaggle to Google Cloud Storage (GCS)

The first bottleneck lies in converting the raw data into Parquet format and transferring it from Kaggle to our Google Cloud Storage bucket. This process becomes increasingly challenging as the dataset grows (e.g., tens or hundreds of gigabytes).

The data must be converted and uploaded through an intermediary compute environment. As the volume scales, transfer time, network throughput, temporary disk limits, and conversion time (CSV $\rightarrow$ Parquet) all become potential blockers.

2. Computing session_features Efficiently

Currently, session-level features are computed directly from the raw events using the Arrow framework, which enables out-of-memory queries and selective materialization of only required data. This works well for moderately sized datasets (up to ~50–100 GB), but beyond that threshold the approach becomes inefficient or even infeasible: group_by() + summarise() operations require shuffles that strain a single machine. Eventually, even collecting a small subset of features into memory becomes too heavy.

At large scale, distributed query engines (e.g., BigQuery) perform dramatically better on operations requiring global grouping, deduplication, and joins because they parallelize execution across many workers and store intermediate results in a distributed storage layer.

3. Fitting Models on Large Datasets

Model training is currently performed locally using CPU-bound R packages (glmnet, biglasso, xgboost). This is fine for datasets up to a few million rows, but it becomes a hard limitation as data grows since the design matrix may not fit into memory, training time increases linearly or super-linearly with data size and cross-validation multiplies the computational cost. Eventually even large local machines eventually reach their limits.

To continue scaling, computation must move to the cloud, where resources can be expanded elastically.

6.2 Solutions

Solution to Problem 1: (mainly data source dependent)

Because Kaggle data cannot be accessed directly from R or GCS, some form of data transfer is unavoidable. However, the efficiency depends heavily on the infrastructure. If the dataset remains on Kaggle, our current approach, using a Kaggle notebook to convert the dataset into parquet format and then upload it to GSC, is already close to optimal. If future data sources are already stored in cloud-accessible environments (e.g., AWS S3, GCS buckets, public Google datasets), this first step can be eliminated entirely. The point here is that this bottleneck is mostly data-source dependent, not architectural.

Solution to Problem 2: Move Feature Engineering to BigQuery (or Spark)

To scale the computation of session-level features, the entire feature-engineering step should be migrated from local R to a cloud-native distributed compute engine. BigQuery performs distributed SQL processing across many workers, it can scan terabytes of Parquet files directly from GCS and GROUP BY user_session is executed in a fully parallel manner. Results can remain stored in the cloud as a derived table and no data needs to move to the local machine. Additional transformations (time differences, categorical indicators, joins) can also be expressed directly in SQL. This completely removes the need to collect() large results into R. The full session-level dataset stays in the cloud, where it can be used by cloud-native ML systems. Alternatively Spark Cluster can be used to compute session_features distributing the workload on different nodes of a cluster of machines.

Solution to Problem 3: Cloud-Based Model Training (Scale Up or Scale Out)

As datasets grow, local hardware becomes the hard limit. Cloud computing offers two scalability strategies. The first is scaling up, which consists of renting a more powerful machine—with more RAM (64–512 GB), more vCPUs (16–96 cores), and optional GPUs for deep learning. This makes it possible to fit larger models or datasets into memory.

The second strategy is scaling out, where computation is distributed across many machines. This can be done using Spark MLlib for distributed logistic regression or gradient-boosted trees, BigQuery ML, which automatically shards data and parallelizes computation, or Vertex AI for distributed XGBoost or TensorFlow/PyTorch training. With this approach, models can be trained on billions of rows without ever moving the full dataset to a single machine.

When the solutions from steps 2 and 3 are combined, the workflow becomes even more efficient: feature generation happens directly in BigQuery, model training is performed in BigQuery ML, Spark, or Vertex AI, and R only receives the outputs (metrics, summaries, model coefficients, or predictions on small samples). This architecture fully decouples the analytical workflow from local machine limits.

References

Bai, J., Fu, F., Guan, R., Hoffman, S., Karnani, P., Mysore, M., & Touse, S. (2021, December 30). Solving the paradox of growth and profitability in e-commerce. McKinsey & Company. https://www.mckinsey.com/industries/retail/our-insights/solving-the-paradox-of-growth-and-profitability-

in-e-commerce
Chaffey, D., & Ellis-Chadwick, F. (2000). Digital marketing: Strategy, implementation, and practice (6th ed.). Pearson. https://digilib.stiestekom.ac.id/assets/dokumen/ebook/feb_27aff686c21a3ec16bdc9e2e8d785bf6b8d8e4e8_1655821975.pdf
Zeng Y, Breheny P (2021). "The biglasso Package: A Memory- and Computation-Efficient Solver for Lasso Model Fitting with Big Data in R." R Journal, 12(2), 6–19. doi:10.32614/RJ-2021-001, 1701.05936, https://doi.org/10.32614/RJ-2021-001.