ChatGPT

CohortIQ Technical Documentation

Overview

CohortIQ is a student-built churn prediction and user segmentation dashboard designed for SaaS products. It combines a **MySQL** relational database for data storage, a **Python** backend for data processing and machine learning, and a **Streamlit** frontend for an interactive dashboard interface. The goal of CohortIQ is to help product and growth teams identify users at risk of churn (i.e. those likely to stop using the service) and understand user behavior patterns through segmentation and retention analysis.

At a high level, the system ingests user and session data from the SQL database and engineers a variety of features capturing user activity. A binary **churn label** is created for each user (e.g. flagging users with no activity in the last 30 days as churned). A **neural network model** (built with TensorFlow/Keras) is trained to predict churn based on these features. Additionally, the project performs **cohort analysis** (retention over time by signup week), **user segmentation** using clustering (Gaussian Mixture Model with PCA visualization), and **survival analysis** (Kaplan-Meier curves to analyze user retention). The Streamlit dashboard presents key performance indicators (KPIs), charts (such as churn rates by segment, cohort retention heatmaps, etc.), and allows filtering (e.g. by signup platform) for interactive exploration of the data.

Technology Stack Summary

The table below summarizes the major technologies, libraries, and languages used in CohortIQ and their roles in the project:

Technology/Library	Role in the Project
Python (3.x)	Primary programming language for the backend logic and ML code.
MySQL (Relational DB)	Stores application data (user profiles and session logs).
SQL	Used for defining the schema and querying data from MySQL.
SQLAlchemy	Python ORM/DB toolkit to connect to MySQL and execute queries (uses MySQL Connector under the hood) [1].
MySQL Connector/ Python	Driver used by SQLAlchemy to communicate with the MySQL database [1].
Pandas	Data analysis library to load SQL query results into DataFrames and perform data manipulation (feature engineering, merging tables, etc.).
NumPy	Provides efficient numeric arrays and mathematical functions, used for feature arrays and feeding data into the ML model [2].

Technology/Library	Role in the Project
Matplotlib & **Seaborn**	Visualization libraries used to create charts (e.g. bar charts, heatmaps, scatter plots) for data analysis and for display in the dashboard.
Scikit-learn	Machine learning toolkit used for data preparation (train/test splitting [3], feature scaling) and algorithms like clustering (Gaussian Mixture Model), PCA for visualization, and utilities (e.g. metrics).
TensorFlow (Keras API)	Deep learning framework used to build and train the neural network for churn prediction [4].
LIME (Local Interpretable Model-agnostic Explanations)	Explainability library to interpret the neural network's predictions by highlighting the influence of features on churn risk [5].
lifelines (Python package)	Survival analysis library used for Kaplan-Meier analysis of user retention (churn over time) [6].
Streamlit	Python web framework for building the interactive dashboard UI. It renders the metrics, filters, and charts in a web app format.
ipywidgets (Jupyter Widgets)	Used during development in Jupyter notebooks to create interactive controls for exploring model explanations (not used in the deployed dashboard, but helpful for prototype) [7] [8].

System Architecture

CohortIQ follows a layered architecture with three main components: the data storage layer, the backend analysis layer, and the frontend presentation layer. Below is an overview of how these components interact:

- **Data Layer (MySQL Database):** All raw data is stored in a MySQL database. This includes a **users** table (with user profiles and signup info) and a **sessions** table (with usage session logs). The database is the single source of truth for the dashboard's input data.

- **Backend Analysis (Python):** The Streamlit app (and accompanying Python modules/notebooks) serves as the backend logic. On startup, the app connects to the MySQL database via SQLAlchemy and loads the data into Pandas DataFrames [1]. The backend then performs all necessary computations: feature engineering on the data, calculating churn labels, training the churn prediction model, running clustering algorithms, and computing survival curves. This layer uses TensorFlow for modeling and various analysis libraries (Pandas, scikit-learn, etc.) to produce insights. The heavy computation is done in Python, and the results (e.g., predictions, cluster assignments, survival statistics) are prepared for visualization.

- **Frontend Presentation (Streamlit):** The Streamlit layer is what the end user interacts with. Streamlit renders a web dashboard directly from the Python code. It displays summary metrics, interactive charts, and controls (filters) for users to slice the data. The Streamlit app invokes the backend computations (since Streamlit scripts are Python, they run the data queries and ML computations on the fly or use precomputed results) and then uses Streamlit's UI functions to

display results to the user. Essentially, Streamlit ties the data and analysis together into an easy-to-use interface.

Data Flow: A typical flow in the system starts with the **Streamlit app** loading, which triggers a **database query** to fetch the latest data. The data is loaded into Pandas and **features are engineered** (e.g., computing each user's activity metrics). The **churn prediction model** can be trained or loaded (in development, training is done on the fly for demonstration). When the user interacts with the dashboard (for example, selecting a filter), the backend code updates the data subset and recomputes relevant metrics or charts, which Streamlit then re-renders. In addition to on-demand computation, some analyses like clustering and survival curves might be precomputed when the app starts (or offline in a Jupyter notebook) and then simply displayed, given their computational cost.

(Optionally, a system architecture diagram could illustrate the above, showing MySQL as the data source, feeding into Python for processing, then into Streamlit for display.)

Backend (Python)

The backend of CohortIQ is a Python codebase that handles data ingestion, feature engineering, machine learning, and analysis. Below we detail each key component of the backend:

Data Ingestion with SQLAlchemy

Data ingestion is performed via **SQLAlchemy** using MySQL Connector. The application creates a connection engine to the MySQL database using credentials and database URL. For example, the code uses a connection string like `mysql+mysqlconnector://<user>:<password>@<host>:3306/cohortiq` to connect to the **cohortiq** database [1]. Using this engine, Pandas can directly read tables into DataFrames. The app reads the **users** table and **sessions** table with commands such as:

```
users_df    = pd.read_sql("SELECT * FROM users", engine)
sessions_df = pd.read_sql("SELECT * FROM sessions", engine)
```

This brings the data into memory as Pandas DataFrames for further processing [1]. The users table contains one row per user, and the sessions table contains one row per user session (linked by a user_id foreign key). By loading both, the backend can combine user attributes with their activity logs.

Feature Engineering

Once data is loaded, the backend computes a number of **features** from the raw data to characterize user behavior. These features are engineered from the sessions log and user info, and will serve both for descriptive analytics and as inputs to the churn model. Key feature engineering steps include:

- **Session Aggregations:** Using Pandas group-by operations, the code computes summary stats per user. For each user_id, it calculates the total number of sessions, the average session duration (in minutes), the timestamp of the last session, and the timestamp of the first session [9]. These give a

sense of overall engagement (how many sessions, how long on average) and recency (when last seen).

- **Recency (Days Since Last Session):** The difference in days between a reference date (e.g. today's date) and the user's last session is computed for each user [9]. This yields a **days_since_last_session** feature. A larger value indicates the user has been inactive for a longer period.

- **Signup Lag to First Session:** By merging the users table with session data, the code computes the **time_to_first_session** for each user – the number of days between the user's signup date and their first-ever session [10]. This feature captures how quickly a user became active after registering.

- **First Week Activity:** The number of sessions a user had in their first week after signup is calculated. The code filters sessions that occurred within 7 days of signup and counts them per user (resulting in a **first_week_sessions** feature) [11]. This feature can indicate initial engagement – users who heavily use the product in the first week might be more likely to stick around.

- **Consistency of Activity (Weekly):** Another feature is **weekly_activity_consistency**, which counts the number of distinct weeks in which the user had at least one session [11]. This is computed by converting session dates into week periods and counting unique weeks of activity per user. A higher count means the user returned in many different weeks, indicating sustained engagement.

After computing these, all features are merged into a single **features DataFrame** (one row per user). Missing values (for users with no sessions in the first week, etc.) are filled with 0 as appropriate [12].

Churn Label Creation

CohortIQ defines a user as **"churned"** if they have been inactive for a prolonged period. In this project, the threshold is set to **30 days** since last session. In other words, if today's date minus the user's last session date is greater than 30 days, the user is labeled as churned [13] [14]. This rule is implemented as a boolean column and then converted to an integer (1 for churned, 0 for active) for modeling:

```
features_df['churned'] = features_df['days_since_last_session'] > 30
features_df['churned'] = features_df['churned'].astype(int)
```

This churn label serves as the target variable for the prediction model. It's a simplistic definition (a proxy for true churn), but it aligns with common SaaS metrics (e.g., considering a user churned if they haven't returned in a month). The threshold could be adjusted as needed for different contexts.

Neural Network Model (TensorFlow)

For predicting churn, CohortIQ uses a **neural network** built with TensorFlow's Keras API. The model takes the engineered features as input and outputs a probability of churn for each user. Here are the details of the model:

- **Input Features:** The features used as inputs include total_sessions, avg_session_duration, days_since_last_session, weekly_activity_consistency, first_week_sessions, and time_to_first_session [15]. These correspond to the user's engagement metrics described above. Before modeling, the feature matrix is typically scaled (standardized) using scikit-learn's `StandardScaler` to ensure all features are on comparable scales [16].

- **Training Data Split:** The data is split into a training set and test set (for example, 80% train, 20% test) using scikit-learn's `train_test_split` [3]. This allows evaluation of the model's performance on unseen data.

- **Architecture:** The neural network is a simple feed-forward **Sequential** model with one hidden layer (or two hidden layers, depending on configuration). In the current implementation, the architecture consists of an input layer, one or two hidden layers, and an output layer:

- Input layer expects 6 features (the input dimension is 6).
- Hidden layer(s): For example, one configuration uses a first dense layer with 32 neurons and ReLU activation, and a second dense layer with 16 neurons and ReLU activation [4]. These layers learn intermediate representations of the user data.
- Output layer: a single neuron with sigmoid activation to predict the probability of churn (outputting a value between 0 and 1) [4].

This architecture is relatively small (suitable for a limited dataset) and can be trained quickly. It's essentially a logistic regression extended with non-linear hidden layers to potentially capture complex patterns.

- **Training Procedure:** The model is compiled with a binary crossentropy loss (appropriate for binary classification) and an optimizer like Adam, and it tracks accuracy as a metric [17]. The model is then trained on the training data for a number of epochs (e.g., 100 epochs) with a small batch size [18]. Given the small scale of the sample data, training is fast. During training, a portion of data is used for validation (e.g., 20% test set as held-out) to monitor performance.

- **Model Evaluation:** After training, the model is evaluated on the test set. In the prototype run, the model achieved a perfect 100% accuracy on the test set [19]. (This was likely due to the very small dataset in this example or potential overfitting – a perfect score on test data is not typical in real scenarios.) In practice, one would look at additional metrics like **ROC AUC**, precision/recall, and the **confusion matrix** to get a better sense of model performance beyond accuracy. The code has placeholders to compute some of these metrics (e.g., it imported `roc_auc_score` and `classification_report` from scikit-learn [20]), indicating the intention to evaluate the model more thoroughly.

  Note: In addition to the neural network, the team also experimented with simpler models for comparison. For instance, a logistic regression model was trained on the same features as a

baseline [21]. This kind of baseline helps validate that the neural network provides added value over a straightforward approach.

Model Evaluation and Interpretation

Beyond raw performance metrics, understanding *why* the model makes certain predictions is important for trust and insight. CohortIQ integrates **model interpretability** through LIME after training the model:

- After the model is trained, the code uses the test set to see how well the model generalizes. The test accuracy (and potentially other metrics) are computed as described above. The small dataset in this project means these results are more illustrative than definitive. One would normally perform cross-validation or use a larger hold-out set to better estimate real performance.

- To get a sense of feature importance, one might look at the weights in a simpler model or use techniques like permutation importance. However, with a neural network (which is not easily interpretable by inspecting weights), **LIME** is used to explain individual predictions, as described next.

Explainability with LIME

LIME (Local Interpretable Model-Agnostic Explanations) is used in CohortIQ to provide insights into the model's predictions. LIME treats the model as a black box and tries to approximate it locally with an interpretable model to explain how input features influence the prediction for a given instance. Here's how LIME is integrated:

- **Setup:** The code instantiates a `LimeTabularExplainer` with the training data and feature names, specifying the mode as classification and the class names as `['Not Churned', 'Churned']` [5]. This sets up LIME to know the context of the prediction problem (binary classification of churn).

- **Predict Function:** Because the explainer needs to call the model, a wrapper predict function (`keras_predict_fn`) is defined that takes an input `x` (numpy array of feature values) and returns the model's prediction in the format LIME expects (for binary classification, an array with probabilities for each class) [2]. In the code, `keras_predict_fn` uses the trained Keras model to produce both class probabilities (it returns `[P(not_churn), P(churn)]` for each input).

- **Explaining an Instance:** To explain a particular user's churn prediction, the code calls `explainer.explain_instance()` with that user's feature vector, the predict function, and a number of features to show in the explanation (e.g., top 6 features) [22]. LIME then perturbs that instance's data, observes the model predictions, and fits a simple model (like a linear model) to approximate the local decision boundary.

- **Output Interpretation:** The result from LIME is an explanation object that can be converted to a human-readable list of feature impacts. For example, LIME might output a list of feature-value conditions with weights, such as:

- `('0.00 < total_sessions <= 1.00', -0.034)`,

- `('first_week_sessions > -0.11', -0.029)`, etc. [23].

Each line can be read as a condition on a feature and a weight (positive weight would push the prediction towards "Churned" and negative towards "Not Churned"). In the above sample, it suggests that having very few sessions (0 to 1 total sessions) contributed negatively to the churn prediction (i.e., it was evidence for churn in this context). LIME essentially identifies which features most influenced the model's output for that specific user.

- **Usage:** In a Jupyter notebook environment, the code uses `exp.show_in_notebook(show_table=True)` to display the explanation visually and even sets up interactive widgets to select different test users and see their explanations [24]. While these interactive widgets are not directly part of the Streamlit app, they demonstrate how a user (or developer) can probe the model's reasoning. In a future iteration, such explanations could be integrated into the Streamlit dashboard (e.g., allowing a dashboard user to pick a user and see an explanation of their churn risk).

By using LIME, CohortIQ ensures the churn model isn't a black box – it provides insight into which factors are driving predictions, which can be invaluable for stakeholders. For instance, if LIME consistently shows that *"low first_week_sessions"* is a strong indicator for churn, product teams might focus on improving user onboarding to encourage more activity in the first week.

Clustering for User Segmentation (GMM & PCA)

Apart from predicting churn, CohortIQ segments users into distinct groups based on behavior. The idea is to identify clusters of users that share similar usage patterns, which can inform targeted marketing or product strategies. The steps involved in user segmentation are:

- **Feature Set for Clustering:** The project uses a similar set of features for clustering as for the churn model (engagement metrics like total sessions, average session duration, recency, consistency, etc.) [25]. These features (possibly scaled via `StandardScaler` again) form the feature matrix for clustering.

- **Gaussian Mixture Model (GMM):** A **Gaussian Mixture Model** from scikit-learn is applied to cluster the users. GMM is a probabilistic clustering method that assumes data is generated from a mixture of Gaussian distributions. In the code, they specify 3 components (clusters) for the GMM and fit it to the user feature data [26]. After fitting, each user is assigned a cluster label (0, 1, or 2) via `gmm.predict` [26]. Three clusters were chosen in this implementation (perhaps based on prior knowledge or by testing different counts).

- **Cluster Labeling:** To make the clusters more interpretable, the team assigned descriptive names to each cluster. In code, a mapping like `{0: 'One-Night Stands', 1: 'Power Users', 2: 'Window Shoppers'}` is used [27]. These colorful labels convey the essence of each segment:

- **Power Users** – likely the cluster of users with high engagement (many sessions, long durations, consistent usage). These users stick around and use the product frequently.
- **One-Night Stands** – likely users who tried the product briefly (perhaps 1 session or very short usage) and did not return (churned quickly).

- **Window Shoppers** – perhaps users who have moderate engagement: they explored the product (maybe multiple sessions or longer first session) but did not convert to heavy usage or eventually churned, or users who intermittently use the product without deep engagement. *(The exact interpretation would depend on the data, but the name suggests they looked around but didn't fully commit.)*

- **Segment Analysis:** Once each user has a segment label, this information can be used to calculate segment-specific metrics (e.g., churn rates per segment, average revenue per segment if data existed, etc.) or simply to filter the dashboard by segment. It provides a more qualitative understanding of the user base beyond just churn vs. not churn.

- **PCA Visualization:** To visualize the clusters, the project uses **Principal Component Analysis (PCA)** to reduce the multi-dimensional feature space down to 2 dimensions. The code applies PCA (2 components) on the scaled feature set and then creates a scatter plot of the users in this 2D space [28]. Each point represents a user, and points are colored/labeled by the segment (cluster) they belong to. This scatter plot (displayed with matplotlib) helps verify that the clusters are reasonably well-separated and gives an intuition of their distribution [29]. For example, one might see one cluster tightly grouped (e.g., power users) and another more dispersed. In the CohortIQ notebook, a legend is added to identify the cluster names on the plot [30].

This clustering and visualization step adds an unsupervised learning perspective to the project: instead of pre-defining segments, the data itself suggests groupings. These user segments can be very actionable – e.g., focusing retention efforts on "Window Shoppers" by converting them into power users, or understanding what drives "One-Night Stands" to leave so quickly.

Survival Analysis (Lifelines)

CohortIQ also incorporates **survival analysis** to understand user retention over time. Survival analysis (often used in medical studies for patient survival, hence the name) is applicable here to model "time until churn" for users. The **lifelines** library's Kaplan-Meier estimator is used to compute the survival curves. Here's how it's done:

- **Preparing Survival Data:** From the features DataFrame, a **survival dataframe** is created that contains for each user:
- **duration** – the number of days the user was active before churning or until the end of observation. In the code, this was computed as the difference (in days) between the last session date and the signup date [31]. Essentially, it's how long the user "survived" from signup until their last activity.
- **event_observed** – the churn event flag (1 if churned, 0 if the user is still active at observation end) [32]. In our case, `event_observed` is basically the same as the churn label: if the user's last activity was more than 30 days ago (churned), we consider the churn event to have occurred; if not, the user is treated as right-censored (they haven't churned as of the observation end date).

Additionally, this survival_df can include user attributes like signup platform, so we can subgroup the analysis (e.g., compare platforms) [33].

- **Overall Survival Curve:** A **Kaplan-Meier Fitter** (from lifelines) is instantiated and fitted on the entire dataset's duration and event columns [6]. The Kaplan-Meier estimator computes the probability of a

user *not having churned* by each point in time since signup. Plotting this **survival function** yields a curve that typically starts at 1 (100% of users active at time 0, i.e. signup) and decreases over time as users churn. In CohortIQ's example, with a small dataset, the curve would show the fraction of users still active after X days since signup. The chart is labeled "User Survival Curve" and has axes for days since signup vs. probability still active [34]. This gives a visual sense of retention – for example, one might interpret "after 30 days, only 40% of users are still active" from such a curve (hypothetically).

- **Survival by Segment or Platform:** The code also demonstrates stratifying the survival analysis by categories. In the example, it loops through unique **signup platforms** (e.g., web, iOS, Android) and fits separate Kaplan-Meier curves for each subset of users [35]. Each platform's curve is plotted on the same chart for comparison [35] [36]. This "Survival Curve by Platform" helps identify if users from certain platforms tend to churn faster or slower than others. For instance, perhaps users who signed up via the web have higher retention than those on a mobile platform, or vice versa. The plotted curves and a legend allow comparison at a glance.

- **Interpreting the Results:** From survival analysis, one can derive insights such as median survival time (e.g., the median active duration of a user before churn) or the retention rate at specific time milestones. These are valuable for business: e.g., if you know that 50% of users churn by day 20, you might intervene in that early window with re-engagement efforts.

In summary, the survival analysis in CohortIQ provides a statistical view of churn, complementing the predictive model and segmentation. It treats churn in a time-to-event framework rather than a binary classification, which is useful for understanding *when* churn happens, not just *if* it happens.

Frontend (Streamlit)

The frontend of CohortIQ is an interactive **Streamlit** dashboard that allows users to explore the churn and segmentation insights in a friendly web interface. Streamlit runs the Python code and displays UI components (numbers, charts, filters) in a browser. Key aspects of the Streamlit UI implementation include:

- **Dashboard Structure:** The app is structured as a single-page dashboard. When the Streamlit app is launched, it runs the Python script which loads data and computes metrics, then renders the interface. The title (e.g., "CohortIQ: Churn Dashboard") is displayed at the top [37], and a sidebar is used for interactive filters.

- **Filters (Sidebar):** A sidebar is provided for filtering the data. In this project, an example filter is **Signup Platform** – a dropdown that lets the user select one of the signup platforms (e.g., "All", "web", "iOS", "Android") [38]. By default, "All" is selected to show overall metrics. If a specific platform is chosen, the backend filters the DataFrame to include only users who signed up on that platform [39]. This allows the user to drill down into metrics for that segment. Additional filters could easily be added (such as date ranges, referral sources, etc.) given the flexible Streamlit sidebar.

- **Key Metrics (KPIs):** At the top of the main page, the dashboard displays high-level metrics using `st.metric` components. For example:

- **Total Users:** The count of users in the current filtered dataset.

- **Churn Rate:** The percentage of those users who are labeled as churned. This is calculated as the mean of the `churned` column (since it's 1 for churned, 0 for not) and formatted as a percentage [40].

These give a quick snapshot of the situation (e.g., "We have 1000 users, and 20% of them have churned under the 30-day definition"). The metrics update automatically when a filter is changed (since the underlying DataFrame is filtered).

- **Charts and Visualizations:** The dashboard includes visual charts to make the data insights more digestible:
- **Churn by Referral Source:** As an example, the code groups the filtered user DataFrame by `referral_source` and calculates the churn rate for each group, then displays a bar chart of churn rate by referral source [41]. This can highlight if users coming through certain channels are more likely to churn. Streamlit's `st.bar_chart` is used for a quick bar plot. In a fully developed dashboard, one could include multiple charts: for instance, a retention curve (using `st.pyplot` to display the matplotlib survival plot), a PCA scatter plot of user segments, or a cohort retention heatmap. These would involve running the corresponding analysis code and then plotting within the Streamlit app. (In the current code, some of these analyses were done in a notebook, but they could be integrated into the app with a bit of refactoring.)
- **Retention/Cohort Analysis:** Although not explicitly shown in the initial Streamlit code snippet, a natural addition to the dashboard is a cohort retention heatmap (given the project name *CohortIQ*). This could be implemented by computing the cohort matrix (as done in the backend with Pandas and Seaborn) and then using `st.pyplot` to display the Seaborn heatmap of retention by week cohort. This would let users visually inspect how each signup cohort retains over weeks (the code for this exists in the project's analysis notebook [42] and could be ported to Streamlit).
- **User Segments & Clusters:** Another possible section is a visualization of the user segments. For example, the PCA scatter plot showing clusters could be rendered in Streamlit. Streamlit supports Matplotlib figures via `st.pyplot(fig)`. By coloring points by segment and perhaps allowing a legend or tooltips, users could see the distribution of segments. Additionally, summary stats per segment (like average churn rate, average sessions, etc.) could be shown in a table or bar chart.

- **Survival Curves:** The survival analysis results can also be presented. For instance, using Streamlit's plotting or by converting the Kaplan-Meier plot to a figure, the dashboard could show an overall retention curve and perhaps allow toggling different segments (e.g. a radio button to select "Overall vs By Platform"). This would provide a visual of churn over time right in the app.

- **Dynamic Alerts/Insights:** In the code, there's an example of using `st.error` to flag a concerning situation. If the overall churn rate exceeds 50%, the app displays an alert message ("High churn rate detected! Send marketing a passive-aggressive Slack.") [43]. This is a lighthearted way to draw attention to a high churn scenario. In a real setting, one could imagine more nuanced alerts or recommendations popping up based on the data (for example, highlighting if a particular segment's churn spiked, etc.).

- **Interactivity:** Streamlit automatically reruns the script when a widget (like the platform selectbox) is changed. This means all metrics and charts are updated for the new filter selection. The app, therefore, behaves interactively without the developer needing to manually manage state beyond reading the widget values and filtering the DataFrame.

- **Layout and Additional Pages:** Currently, the content described appears on one page. Streamlit does allow multiple pages/tabs (via `st.sidebar.radio` or the newer multipage app feature). For instance, one page could be "Churn Overview" and another "User Segmentation" or "Retention Analysis," separating different analyses. The project could evolve to use this for better organization if the single page becomes too crowded.

In summary, the Streamlit frontend ties everything together by providing an accessible interface. A user (such as a product manager or an analyst) can choose filters, view key metrics at a glance, and see various charts that the backend analysis produces – all without needing to run code manually. The combination of backend power with a front-end UI makes CohortIQ a self-service analytics tool for churn and retention insights.

SQL Schema

The **CohortIQ** database (MySQL) consists of at least two main tables: **users** and **sessions**. Understanding the schema is important for anyone looking to extend or modify the data model. Below is an overview of these tables, their columns, and relationships:

- **Users Table (`users`):** Each row represents a unique user of the SaaS product.
- **user_id** – Integer, primary key. A unique identifier for each user.
- **username** – String, the user's chosen username or handle.
- **full_name** – String, the full name of the user (if provided).
- **user_email** – String, the email address of the user.
- **signup_date** – DateTime, the timestamp or date when the user signed up/registered.
- **signup_platform** – String, the platform or source of signup (e.g., "web", "iOS", "Android"). This indicates where the user registered.
- **referral_source** – String, how the user found the product (e.g., "ads", "invite", "organic"). In the example dataset, values like "ads", "invite", "organic" appear [44].

Example: A sample user row might be: user_id=1, username="astroTom", full_name="Tom Nielsen", user_email="tom@spacemail.com", signup_date="2025-06-01", signup_platform="web", referral_source="ads" [45] [44].

- **Sessions Table (`sessions`):** Each row represents a single session or visit in which a user engaged with the product.
- **session_id** – Integer, primary key for each session (e.g., 101, 102, 103, ...).
- **user_id** – Integer, foreign key referencing `users.user_id`. This links the session to the user who initiated it.
- **session_start** – DateTime, the start timestamp of the session.
- **session_end** – DateTime, the end timestamp of the session.
- **device_type** – String, the type of device used ("desktop", "mobile", etc.).
- **session_type** – String, category of session activity (for example, in the data we see values like "tutorial", "usage", "support" indicating what kind of session it was) [46] [47].
- **pages_viewed** – Integer, how many pages or screens the user viewed during the session.
- **is_conversion** – Boolean/Integer (e.g., 0/1), indicating if the session resulted in a conversion (this could mean different things depending on context – perhaps a purchase or subscription upgrade). In the sample data, we see 1 or 0 for conversion within sessions [46] [47].

Example: A session row might look like: session_id=101, user_id=1, session_start="2025-06-01 10:00:00", session_end="2025-06-01 10:30:00", device_type="desktop", session_type="tutorial", pages_viewed=5, is_conversion=1 [46] [47]. This would correspond to user_id 1 (Tom) having a 30-minute desktop session doing a tutorial, viewing 5 pages and converting on that session.

- **Relationships:** There is a one-to-many relationship from **users** to **sessions**. Each user can have multiple sessions, but each session is tied to exactly one user. The `user_id` in sessions is the foreign key that enforces this link. This relationship means we can join users with their sessions to aggregate data per user.

- **Schema Usage in Code:** In the Python backend, after loading the tables, these relationships are leveraged (via Pandas merge or groupby) to compute features. For instance, grouping the sessions by user_id uses the relationship to derive user-level aggregates, and merging session data with user signup dates relies on the foreign key to align sessions with user attributes [9] [10].

- **Additional Tables:** (Not explicitly mentioned in the prompt, but for completeness) If the project were extended, one might expect other tables such as a **subscriptions** table or **transactions** table (if financial data or plan info is relevant), but in the given scope, users and sessions suffice to derive churn-related metrics.

Anyone looking to extend the schema should maintain this normalized design: user info separate from session logs. If new user attributes are to be added (e.g., plan type, or marketing cohort), those can typically go into the users table. If new event types or logs are recorded, they might form new tables or extend the sessions table depending on their granularity.

ML Model Details

This section provides more details on the machine learning model used for churn prediction, including its inputs, architecture, training parameters, and evaluation approach:

- **Input Features:** The model takes as input a vector of features that summarize each user's behavior and status. These are the features engineered in the backend:
- `total_sessions` – total number of sessions the user has had.
- `avg_session_duration` – average session length (in minutes) for the user.
- `days_since_last_session` – how many days since the user's last session (recency).
- `weekly_activity_consistency` – number of distinct weeks the user was active (a measure of consistent engagement).
- `first_week_sessions` – number of sessions in the user's first week after signup (early engagement metric).
- `time_to_first_session` – days from signup until the user's first session (onboarding lag).

These features are all numeric. Before feeding into the model, they are scaled to a standard range (mean 0, std 1) using `StandardScaler` [16]. Scaling is important for neural networks so that one feature doesn't dominate just because of scale differences.

- **Model Architecture:** The churn prediction model is a neural network classifier. To recap the architecture in more technical terms:
- It's a fully-connected feedforward network (a Multilayer Perceptron).
- Layer 1 (Input layer): 6 neurons (one for each input feature, although in implementation Keras often just expects the input_dim rather than an explicit input layer neuron count).
- Layer 2 (Hidden layer): 32 neurons, ReLU activation [4].
- Layer 3 (Hidden layer): 16 neurons, ReLU activation [4]. (If only one hidden layer were used, it might be 32 -> 1 directly, but code indicates a second layer with 16).
- Layer 4 (Output layer): 1 neuron, Sigmoid activation [4].

There are no complicated structures like convolution or recurrence here – just dense layers. The choice of 32 and 16 units is somewhat arbitrary but provides the model with some representational capacity. The activations are ReLU for hidden layers (a good default for non-linear transformation) and Sigmoid for the output to squash the output into a [0,1] probability.

- **Training Hyperparameters:** The model is compiled with:
- **Loss function:** Binary Crossentropy – appropriate for binary classification, as it measures the divergence between the predicted probability and the actual 0/1 label.
- **Optimizer:** Adam – a popular gradient descent variant that adapts learning rates, good for quick convergence.
- **Metrics:** Accuracy (and implicitly loss). Accuracy is tracked to see what fraction of users are correctly classified as churn or not churn [17].

The model was trained for **100 epochs** with a **batch size of 4** in the example run [48]. 100 epochs on a small dataset is likely overkill and could lead to overfitting, but since this was a demonstration with only a handful of users, it's manageable. In a real scenario, one would use early stopping or cross-validation to decide the number of epochs to avoid overfitting. The batch size of 4 is quite small; with more data, one might use 32 or 64, but with very small data, a small batch can sometimes help the optimizer make progress (at the risk of noisy updates).

- **Model Training and Validation:** The dataset was split into training and testing (80/20). The training was done on the training set, and validation was likely done on the test set just for reporting purposes (though typically one would keep a separate validation if doing many epochs, or do a single train/test split for final evaluation). The history of training (loss, accuracy per epoch) was not explicitly visualized in the code, but one could inspect `history.history` if needed to see if it was converging. Given the simplicity of the task and small data, the model probably quickly fit the training data.

- **Evaluation Metrics:** After training, the code evaluates the model on the test set to get a **Test Accuracy** [19]. As noted, it printed 1.00 (100%). In practice, aside from accuracy, we would consider:

- **Confusion Matrix:** to see how many churners vs non-churners were correctly/incorrectly classified. This is important because with churn, often the class distribution might be imbalanced (many more non-churn than churn, or vice versa).

- **Precision/Recall:** especially if the business cares more about catching churners (recall of churn class) or about precision (not falsely labeling active users as churn).
- **ROC AUC:** a threshold-independent measure of classifier performance, which was imported in the code. AUC gives a sense of the model's ability to rank-order churn risk.

The documentation suggests technically savvy readers, so it's worth noting that with small data the model likely overfit (100% accuracy on test usually means the test set was very small or the model memorized it). If this were deployed on a larger scale, one would monitor these metrics and possibly perform regularization or gather more data to improve generalization.

- **Model Persistence:** The current project code does not explicitly show saving the trained model to disk. Each run re-fetches data and trains the model anew. For a production scenario, one would train the model offline (or during an update interval) and save it (using Keras's `model.save()` or similar), then load the trained model in the Streamlit app for prediction. This way, the dashboard can compute churn predictions on the fly for new data without retraining. The documentation doesn't show that step, so contributors might consider adding it.

- **Using the Model for Predictions:** With a trained model, one can feed new user feature data into `model.predict()` to get churn probability. In the Streamlit app context, if the model were integrated, we could display each user's churn probability or highlight top N users most likely to churn. The LIME explanations further augment this by explaining those probabilities for individual users, as discussed.

In conclusion, the ML model is a straightforward binary classifier tailored to the features available. It captures nonlinear interactions (if any) via hidden layers, and provides a probability of churn that can be used to rank users by risk. While simple, it's a solid starting point that could be improved with more data or more complex models if needed.

Explainability (LIME)

To ensure the churn model's predictions are interpretable, CohortIQ integrates the LIME library for explainability. Here we elaborate on how LIME is used and what insights it provides:

- **Why LIME:** Neural networks (and other complex models) can be "black boxes," meaning it's not obvious why they predict a user will churn. LIME addresses this by focusing on one prediction at a time and perturbing the input a bit to see how the prediction changes. It then produces an explanation that is essentially a weighted list of feature conditions that influenced the prediction.

- **Integration in Code:** After training the model, the code initializes `LimeTabularExplainer` with the training data and feature names [5]. By providing the raw training data, LIME knows the distribution of features (to sample perturbations) and uses that as a baseline. The class names `['Not Churned', 'Churned']` make the explanation more readable (so it will say "this pushes towards Churned" instead of "class 1").

- **Explaining a Prediction:** The typical usage shown is:

```
exp = explainer.explain_instance(X_test[i], keras_predict_fn,
num_features=6)
```

where `X_test[i]` is a sample user's features from the test set, and `keras_predict_fn` is the custom function that returns probabilities from the model [22]. The `num_features=6` tells LIME to show the influence of all 6 features (since we only have 6, it might as well show all; if there were more features, you might limit this to the top few).

- **Output of LIME:** The result `exp` can be output in various ways. In the notebook, `exp.show_in_notebook(show_table=True)` was used to display an HTML formatted explanation, and `exp.as_list()` returns a list of tuples (feature_condition, weight) [23]. For example, an explanation might read:

- `total_sessions <= 1.0 contributes -0.03 to the prediction of churn`
- `first_week_sessions > 0 contributes -0.02 to churn prediction`
- `time_to_first_session <= 0 days contributes +0.01 to churn prediction` (Note: sign interpretation: a negative weight might indicate it pushes towards Not Churned if the model output is on a Churn probability scale, whereas positive pushes towards Churn, depending on how LIME frames it. In a binary case, LIME typically explains the class of interest, e.g., Churned, so a negative weight means that condition is evidence against churn.)

Essentially, LIME produces a simple linear model approximation for that single prediction. The feature conditions (like `0.00 < total_sessions <= 1.00`) partition the feature's value range and indicate where this user's feature falls. In the example, if `total_sessions` for that user was 1, the condition "0 to 1 total sessions" had a negative weight, meaning having such few sessions made the model less likely to churn them (which is a bit counter-intuitive – one would expect few sessions to increase churn probability – but this could be an artifact of small data or how scaling is done; it's just an illustration of the output format).

- **Insights for Stakeholders:** By examining many such LIME explanations (for different users), patterns emerge. For instance, one might notice:
- Users with very high `days_since_last_session` consistently get a strong positive weight toward churn (as expected, long inactivity is evidence of churn).
- Users with higher `weekly_activity_consistency` (active in many weeks) might get negative weights (evidence against churn).
- A short `time_to_first_session` (user started using the product immediately after signup) could be evidence they are engaged (thus negative weight for churn), whereas a long delay might push towards churn risk.

These align with intuition and validate that the model is picking up reasonable signals. If something counter-intuitive is seen, it might indicate the model learned an odd pattern or there is data noise – which could spur further investigation.

- **Use in the Dashboard:** Currently, the Streamlit dashboard doesn't expose LIME explanations to the end-user. However, a developer or data scientist can run the notebook to generate explanations. For future development, one could integrate LIME into the app by, say, allowing a user to select a

particular user ID and then computing the LIME explanation for that user's churn prediction on the fly (the notebook's use of `ipywidgets` to select a user and display explanation demonstrates the concept [49] [24]). The explanation could be shown as a list or bar chart of feature contributions in the app.

- **Limitations:** It's important to note that LIME explanations are local and approximate – they explain a prediction for one user, not the model globally. They can sometimes be misleading if the model's behavior is highly non-linear in that region. Still, they are very useful for sanity-checking the model and providing actionable insights (e.g., "Users who do X are at risk of churn").

In summary, LIME brings transparency to CohortIQ's churn model. It helps translate model predictions into understandable reasons, bridging the gap between complex ML output and business understanding. This fosters trust in the model and helps target interventions (for example, if low first-week usage is a driver of churn, the team knows to focus on improving the onboarding experience).

User Segmentation

User segmentation in CohortIQ is achieved through clustering, specifically using a Gaussian Mixture Model on behavioral features, as described earlier. Here we compile the details and rationale of the segmentation process, and how the results can be interpreted:

- **Segmentation Goal:** Rather than treating all users as one homogeneous group, segmentation aims to identify distinct groups of users who behave similarly. Each segment can be analyzed and targeted separately (for marketing, support, etc.). In churn analysis, segmentation can reveal, for example, a group of users that churn very quickly vs. a loyal core group.

- **Clustering Method – Gaussian Mixture Model (GMM):** The project uses a GMM for clustering. GMM is a soft clustering approach (it gives probabilities of belonging to each cluster, though ultimately we assign the user to the cluster with highest probability) and can capture clusters that may have different shapes in feature space (unlike K-means which assumes spherical clusters). With GMM, we assume our feature data is generated from a mixture of 3 Gaussian distributions (since n_components=3 was chosen) [26]. The algorithm will output the parameters of these Gaussians and assign each user a likelihood for each cluster.

- **Chosen Features for Clustering:** The features used for clustering are the same or similar to those used in the churn model (since those features capture user behavior well). This includes overall usage (total sessions, avg duration), recency (days since last), consistency (weekly active weeks), early usage (first week sessions), and onboarding speed (time to first session) [25]. Using these, the clustering groups users by patterns like "low usage vs. high usage", "recently active vs. long gone", etc. Before running GMM, these features are scaled to ensure one feature doesn't dominate due to scale differences [50].

- **Number of Clusters:** 3 clusters were used in the implementation. The choice of 3 could have been based on an **elbow method** or simply a hypothesis of three distinct user types. If one wanted to be rigorous, they could vary the number of clusters and use metrics like Bayesian Information Criterion

(BIC) to find an optimal number for GMM. For simplicity, 3 was a reasonable guess (for example: high, medium, low engagement segments).

- **Cluster Labeling & Characteristics:** After running the GMM, each user got a cluster label 0, 1, or 2. To make these meaningful, the project mapped them to names:

- **Cluster 0: "One-Night Stands"** – likely users with one or very few sessions (they tried the app briefly and left). Characteristics might include: total_sessions very low (1 or 2), short average session, maybe a moderate time_to_first_session (they did log in quickly since they at least had one session), but then no consistent usage (weekly_activity_consistency = 1 maybe), and they definitely churned early (days_since_last_session high since they haven't come back in a long time).
- **Cluster 1: "Power Users"** – likely the opposite: users with many sessions, long session durations, very frequent usage over many weeks, started using immediately upon signup, etc. These users might still be active (days_since_last_session low). They represent the loyal customer base.
- **Cluster 2: "Window Shoppers"** – likely an intermediate group. Perhaps they have a few sessions (more than one-night stands, but not a power user's amount), or they spent some time browsing (maybe decent pages_viewed) but didn't convert (is_conversion maybe 0 mostly for them) or didn't become regulars. They might have moderate consistency (a few weeks of activity) but eventually churned or are on the fence. They "window shopped" the product without fully committing.

These interpretations are speculative; one would verify them by looking at the actual averages of features per cluster. For instance, one could compute the mean of each feature for each cluster label to confirm these assumptions (e.g., cluster 1 likely has the highest total_sessions, cluster 0 the lowest, etc.).

- **Use of Segments:** Once the segment labels are assigned to each user in `features_df` [26], this opens up many possibilities:
- We can calculate churn rate per segment. Perhaps "One-Night Stands" have a churn rate approaching 100% by definition (they all churn), "Power Users" have churn rate maybe 0% or very low, and "Window Shoppers" have something in between.
- We can filter the dashboard by segment (similar to platform filter) to see metrics for each segment.
- Business strategies can be tailored: e.g., focus retention efforts on the "Window Shoppers" with targeted campaigns to convert them into power users, and analyze why "One-Night Stands" didn't find value (maybe the product needs to deliver value faster).

- If new users come in, one could predict which segment they might fall into early on (though here it's unsupervised, you'd typically re-run clustering periodically or train a classification model to predict segment).

- **Visualization:** As mentioned, PCA was used to reduce features to 2D and scatter plot the users, colored by segment [28]. The resulting plot (e.g., "User Segments via GMM Clustering") would show how distinct or overlapping the clusters are. If the clusters are well-separated, the segments are quite distinct in behavior. If they overlap, some segments might be less clearly defined. The PCA components themselves are linear combinations of features, so one could even analyze what those components represent (though not strictly necessary for using the segments).

In essence, the segmentation adds a **descriptive analytics** layer to CohortIQ. Rather than just saying "who will churn", it asks "what *kinds* of users do we have?" and "which kind is each user?". This is valuable for storytelling and for targeting: for example, you might present to stakeholders that *"We have three main user*

segments. Segment A are power users who almost never churn – let's find ways to reward them. Segment B often leaves after one try – maybe there's an issue or mismatch in expectations. Segment C uses the product a bit but doesn't fully engage – perhaps with the right nudges, they could be retained." The combination of predictive modeling and clustering thus provides both **predictive** and **prescriptive** insights.

Survival Analysis

Survival analysis in CohortIQ provides a different perspective on churn by focusing on *when* users churn rather than simply if they churned. Using lifelines' Kaplan-Meier analysis, the project generates survival curves which can be informative to product managers and growth teams. Here's a closer look at how it's done and interpreted:

- **Kaplan-Meier Estimator Recap:** The Kaplan-Meier estimator calculates the probability of survival (in our case, "survival" means "remaining an active user") over time. It accounts for right-censoring (users who haven't churned by the end of observation are still "alive" in survival terms). The output is a stepwise function that drops at each observed churn event, reflecting the proportion of users still active.

- **Data Inputs:** As described, the key inputs are:

- `duration`: for each user, time from signup to churn (or to last observation if not churned).
- `event_observed`: a boolean flag if churn occurred (True) or if the data was censored (False, meaning the user was still active at last check).

In code, `duration` = `last_session_date - signup_date` in days [31], and `event_observed` = `churned` (1 or 0) [32]. One nuance: If a user is still active, their `last_session_date` might be quite recent relative to the observation cutoff (e.g., last_session could be yesterday). If the observation cutoff is July 1, 2025, and a user's last session was June 30, 2025 and they're not churned, their duration would be ~29 days and event_observed = 0 (censored). If another user last session was June 1, 2025 (and churn threshold is 30 days) then by July 1 they are churned with duration ~30 days, event_observed = 1.

- **Overall Survival Curve:** Fitting the KaplanMeierFitter on all users gives an overall retention curve [6]. For example, if we had 100 users:
- At time 0 (signup), survival probability = 1 (100% of users are "alive" i.e., active since none have churned at the moment of signup).
- The curve then might drop at the time when the first user churns. Suppose 10 users churn exactly at 7 days (meaning they had last sessions at day 7 and then no sessions after, so by day 37 they are declared churned – but careful: The times in KM are measured in the actual duration to churn. If we define churn as no activity for 30 days, the churn event time is effectively ~30 days after last session? This is a bit tricky: in this implementation, they took last_session - signup as duration, and churned = True/False at observation end. So a churn event is recorded at the full length of the user's usage. A user who churned after 10 days of usage will have last_session 10 days from signup; one who churned after 2 days will have 2, etc. So the survival curve is effectively showing "percentage of users who remain active at least X days after signup" with churn defined when they stop using for good.)
- The example output snippet said "fitted with 10 total observations, 10 right-censored" [51] which implies maybe none churned in that small sample by the observation end (so all were censored). If

that's the case, the survival curve would flatline at 1.0 (100% survival) for the period of observation, which isn't insightful. With a larger realistic dataset, you'd see a decline.

Interpretation: If the curve goes down to, say, 0.5 by day 30, that means 50% of users churned within the first 30 days post-signup. If it goes to 0.2 by day 90, it means only 20% of the original users are still active 3 months in. These are the kinds of insights KM curves provide.

- **Grouped Survival Curves (By Platform):** They did separate KM fits for each signup platform [35]. So imagine we have two platforms, Web and iOS:
- For Web signups, maybe the curve is higher (meaning web users tend to stay longer).
- For iOS, maybe it drops faster (maybe iOS users churn sooner).

The plotted chart "Survival Curve by Platform" would have (for example) a blue line for Web and orange line for iOS, etc., with a legend [36]. If one line is consistently above another, that segment has better retention. If they cross, their retention rates over time vary (maybe one group is better early on, but later they equalize or reverse). In the code, after plotting, they call `plt.legend()` to show labels, and add titles and grid for readability [52].

- **Findings and Usage:** In practice, such survival analysis can highlight differences in user behavior:
- Perhaps *mobile users churn faster* than *web users* – that might indicate issues in the mobile experience or a less engaging mobile onboarding.
- Or *referral invites* vs *ads* could be used as grouping to see which acquisition channel yields longer staying users.
- Or one could do survival by segment (Power Users vs others): likely "Power Users" have a very high survival curve (barely dropping, since they rarely churn in the observed window), whereas "One-Night Stands" drop to near 0 survival by day 7 or 14 (since they leave almost immediately).

This kind of analysis complements the static churn rate by adding a time dimension. Stakeholders might prefer to see "retention curves" (the complement of churn) to understand how quickly users drop off. Many SaaS companies track metrics like D30 retention (percentage still active on day 30). The KM curve is a more complete picture of that.

- **Technical Note:** The survival analysis as implemented treats the churn event time as the last session date difference, which is a proxy. A more precise method could have been: if churn = True, set event time = last_session - signup (as done); if churn = False (censored), set event time also = last_session - signup (their usage duration so far). This implies we assume if they haven't churned by July 1, 2025, that's the censoring date. Ideally, one might want to censor at a uniform date (e.g., all users observed until July 1). In this dataset it seems all signups were in June 2025 and observation ended July 1, 2025, so everyone had at most ~30 days of data. So indeed if none churned before July 1 by the 30-day rule, all were censored. If any last_session was more than 30 days before July 1, that user would count as churned and their event time would be the length of their usage. This approach is reasonable but could be refined if needed (just an implementation detail for those who want to extend it).

In summary, survival analysis provides an aggregate view of churn timing. It is particularly useful for presentations and reports, as one can say "Our 1-week retention is X%, 4-week retention is Y%" directly from the curve. CohortIQ's use of lifelines makes it easy to calculate these and even compare groups, which adds depth to the analysis beyond what a simple churn rate or even predictive model can offer.

Deployment Notes

Currently, CohortIQ appears to be set up for **local development and deployment**. Here we outline how one would run the project locally, and discuss considerations for deploying it to a cloud or production environment in the future:

- **Local Environment Setup:** To run CohortIQ, you need Python (with the required libraries) and access to the MySQL database. The dependencies include those listed in the tech stack (Pandas, SQLAlchemy, MySQL connector, scikit-learn, TensorFlow, Streamlit, etc.). It's convenient to use a virtual environment or `conda` environment with these installed. The MySQL database should be running (the code expects it at `localhost:3306` with a database named `cohortiq` and certain credentials [1]). One would need to have the schema and data loaded into MySQL as expected (the project likely comes with some sample data or scripts to generate it).

- **Running Locally:** To start the dashboard, you would run the Streamlit app script. For example, if the main file is `dashboard.py`, the command is:

  ```
  streamlit run dashboard.py
  ```

  This will launch a local web server (usually at `http://localhost:8501`) where the dashboard can be accessed. The app will connect to the local MySQL instance, query data, perform computations, and render the UI. Ensure that the database credentials in the code are correct for your local setup (the example uses user "root" with a password, which you might need to change to your own MySQL credentials or set up such a user).

- **Security of Credentials:** Currently, the database password is hardcoded in the code (even visible in the `create_engine` call) [53]. For deployment, it's recommended to **not hardcode secrets**. Streamlit provides a way to use a secrets file or environment variables. Future deployments should externalize the DB credentials (e.g., via `st.secrets["DB_PASSWORD"]` or an environment variable) to avoid exposing them in code repositories.

- **Streamlit Cloud Deployment:** One straightforward way to share the dashboard is using **Streamlit Cloud** (formerly Streamlit Sharing). This platform can run the app directly from a GitHub repo. To do this, one would have to:

- Push the code to a GitHub repository.
- Add a `requirements.txt` listing all the Python dependencies (Streamlit, SQLAlchemy, mysql-connector-python, pandas, scikit-learn, tensorflow, lime, lifelines, etc.).
- In Streamlit Cloud, set up the app to run `dashboard.py`.
- The tricky part is database access: Since Streamlit Cloud can’t directly connect to a local MySQL on your machine, you’d need a cloud-accessible database. Options include:
  - Deploy the MySQL database to a cloud service (e.g., AWS RDS, Google Cloud SQL) and allow the Streamlit app to connect to it. You’d supply the cloud DB host, user, password as secrets.
  - Or, use a lightweight database like SQLite for the demo; you could ship a `.sqlite` file with the data. That would remove the external dependency and might be easier for a demo.

Pandas can read SQLite via SQLAlchemy with a connection string like `sqlite:///cohortiq.db`.

- Ensure to configure secrets in Streamlit Cloud (they have a UI for adding secrets) so the credentials are not exposed in the repo.

- **Other Hosting Options:** Alternatively, the app could be containerized using Docker and hosted on a platform like Heroku, AWS, etc. A Docker setup would include:

- A container for the Streamlit app.
- Possibly a container for MySQL (with seeded data).

- Docker Compose could orchestrate running them together. This is more involved but gives more control (and is suitable if you expect higher traffic or need to run on an internal server).

- **Scaling Considerations:** Right now, the app likely loads all data into memory and recomputes features on the fly. For a small dataset, this is fine. If the user base grows (say millions of users, tens of millions of sessions), this approach would become slow or impossible to run in real-time. Deployment in that case might require:

- Using database-side aggregations (SQL queries) to pre-compute some features rather than pulling all raw data into Pandas.
- Caching results or loading a precomputed dataset rather than crunching everything on each run. Streamlit has caching mechanisms (`@st.cache_data` or similar) that could be used to avoid re-running expensive computations on every interaction.

- Possibly moving the heavy ML computations (training, clustering) offline. For instance, train the churn model separately and just load it for inference in the app; compute cluster assignments offline and store them in the database so the app just fetches them, etc.

- **Current Deployment Status:** The documentation implies that currently the setup is local (e.g., connecting to localhost MySQL). So any contributor trying to run it should replicate a similar environment. There might be a sample dataset or SQL dump provided (perhaps that PDF had the head of tables as demonstration).

- **Future Deployment Enhancements:**

- **Continuous Deployment:** If this is a student project, deployment might not be fully automated. Setting up CI/CD to automatically test and deploy the app upon new commits would be a nice improvement for a production scenario.
- **Monitoring:** In a deployed app, one would monitor usage, performance (Streamlit can show if scripts are slow), and possibly add logging for user interactions (with care for privacy).
- **Streamlit Features:** Streamlit offers features like authentication (not built-in, but can be done), or ability to password-protect an app if it's internal. For a production tool, we'd consider who is allowed to view this dashboard – if it contains sensitive data, it shouldn't be open to the internet without a login.

In short, deploying CohortIQ beyond local use requires addressing database access (using cloud-hosted data or embedding data), securing credentials, and potentially refactoring parts of the code for performance. For demonstration and development, running it locally with the provided data works well, and Streamlit Cloud is an accessible next step if the database aspect is solved.

Future Improvements

While CohortIQ provides a solid foundation for churn analysis and user segmentation, there are numerous opportunities to enhance and extend the project. Here are several suggestions for future improvements:

- **Increase Data Volume & Variety:** The current project likely uses a relatively small, simulated dataset. Obtaining more data (both in number of users and in additional attributes) would help improve the model. For example, incorporating user demographic info or product usage details (features used, support tickets, etc.) could enrich the feature set and boost prediction accuracy. With more data, one could also move beyond the simple 30-day rule for churn label – perhaps use a longer window or a tiered definition of churn.

- **Feature Engineering Enhancements:** Introduce more features that could influence churn:

- Cohort-based features, e.g., relative position compared to peers (did this user do fewer sessions than the average of their signup cohort?).
- Engagement scores combining multiple factors.
- Text analysis of user feedback or support tickets (if available) to see sentiment as a churn predictor.

- Recency/frequency/monetary (RFM) analysis if applicable (common in user segmentation).

- **Model Improvements:**

- Experiment with more advanced models like **XGBoost or LightGBM** for churn prediction, which often perform well on structured data. Or try a simpler model like **Random Forest** which can handle feature interactions and give importance scores (the code already imported RandomForestClassifier, hinting at potential use [20]).
- Use hyperparameter tuning (via GridSearchCV or randomized search) to find the optimal neural network architecture or settings (e.g., number of layers, neurons, learning rate).
- Implement cross-validation to better estimate performance and avoid overfitting. This is especially important if the dataset grows.

- If the user base is large and churn is relatively rare, consider techniques for imbalanced data (resampling, class weights in the model, etc.).

- **Real-Time Prediction and Alerts:** Currently, churn is analyzed periodically (with data up to a certain date). For a live SaaS, one could integrate real-time prediction:

- As new sessions come in (or as time passes without sessions), update the user's churn probability.
- Set up alerts: e.g., if a user's churn probability exceeds a threshold, trigger an email to account management or add the user to a re-engagement campaign list.

- Integrate with marketing automation: feed the segment or churn risk data into tools that can, say, send special offers to at-risk users.

- **Dashboard UX Enhancements:**

- Add more pages or tabs to organize content (e.g., a page for “Churn Prediction”, another for “User Segments”, another for “Retention Analysis”).
- Include interactive elements like tooltips on charts (Streamlit doesn’t natively support rich tooltips on Matplotlib, but Plotly could be used for interactive charts).
- Allow selecting a single user to see a “profile” of that user: their features, their churn risk, and possibly show their LIME explanation in the app. This could be done by adding a selectbox or text input for user_id and then displaying info for that user.
- Implement a cohort analysis section explicitly: show the retention heatmap for weekly cohorts that was computed in the analysis. Make it interactive (choose monthly cohorts vs weekly, etc., if data permits).

- Use Streamlit’s caching to improve performance so that expensive computations (like training model or clustering) are not redone unless data changes.

- **Survival Analysis Deep-Dive:** Expand the survival analysis:

- Use statistical tests to compare survival curves between groups (lifelines provides functions for that, e.g., log-rank test to see if platform A vs B have statistically different retention).
- Fit parametric survival models (e.g., Cox proportional hazards model) to identify which factors significantly affect churn time. This can complement the classification model by giving hazard ratios for features.

- Update the survival analysis as new data comes in (perhaps do it monthly to see if retention curves are improving after certain product changes, etc.).

- **Data Pipeline and Automation:**

- If this were to be a continuously updated dashboard, implement a pipeline to fetch new data regularly. For instance, a daily job that pulls the latest database entries, updates the features, retrains the model if necessary, and refreshes the dashboard (or the dashboard could be set to query fresh data on load).

- Use tools like Apache Airflow or cron jobs for scheduling data updates and model retraining.

- **Collaborative Features:**

- Integrate comments or annotations on the dashboard for team members to note observations (this is more of a web app feature than Streamlit’s typical scope, but interesting for knowledge sharing).

- Export functionality: allow downloading a CSV of users with their churn probability and segment, or downloading charts as images for reports.

- **Deployment and Scalability:**

- Containerize the application (Docker) and consider a microservices approach if it grows (e.g., a separate service for the ML model if needed).
- If using cloud deployment, ensure to use a managed database and possibly use secrets management for credentials (as noted earlier).
- Set up monitoring (like using Streamlit's logger or external monitoring) to track the app's usage and performance in production.
- **Data Privacy and Ethics:** If this were real user data, implement proper data protection. For instance, hashing or not displaying any personally identifiable information (emails, names) on the dashboard unless necessary. Ensure compliance with any regulations (GDPR etc., if applicable, when handling user data for churn analysis).
- **Extend to Other KPIs:** Since it's called CohortIQ and focuses on SaaS metrics, one could extend it to not just churn but also conversion rates, lifetime value predictions, etc., using a similar framework. For instance, predict which trial users will convert to paid, using a classification model, and present that on the dashboard.

Each of these improvements could be a project on its own. They would make CohortIQ more robust, scalable, and feature-rich. The current system provides a baseline that is already valuable: it demonstrates end-to-end how to go from raw data to insights (via charts and predictions). By iterating on it with the above enhancements, CohortIQ can evolve from a student project into a production-grade analytics tool for SaaS businesses.

---

1 2 3 4 5 6 7 8 9 10 11 12 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 42 44 45 46 47 48 49 50 51 52 cohortiq.pdf

file://file-5yXzz3kc34tDjUfpoY7vsF

13 37 38 39 40 41 43 53 dashboard.py

file://file-AEYALVQNxBsfN48UnGVdXA