Solution for

State Bank of India's

FINNOVATION

Hack II

SBI

Technical Board
IIT Guwahati

Welcome!

Thank you for joining us today

We are team **KMeans**, presenting our solution on the given problem statement of ***Detecting Loan Defaulters and tracking them in digital ecosystem***.

Over the course of the presentation we'll deep dive into our approach and the results we have obtained on our approach.

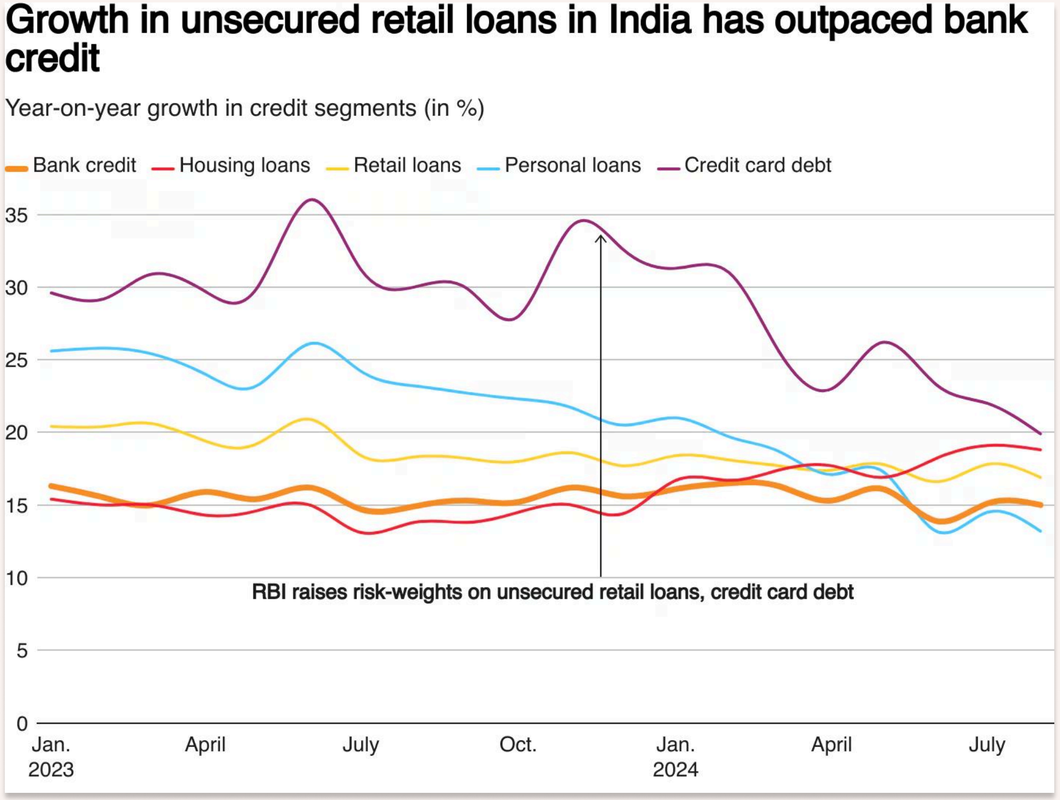
Flow

1. Understanding Problem Statement
2. Approach for Task 1
3. Approach for Task 2
4. Wrap-Up and Q&A

Understanding

Problem Statement

Growth in unsecured retail loans in India has outpaced bank credit



Source: Reserve Bank of India

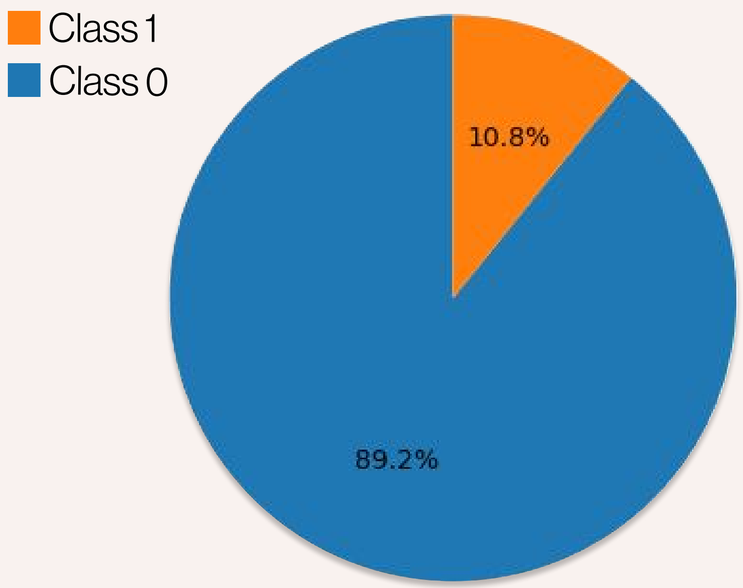
This challenge calls for intelligent systems capable of two core tasks: **defaulter detection**, which involves identifying potential defaulters from behavioral patterns, and **defaulter localization**, which aims to estimate a user's last known position using indirect digital traces and tower logs even when explicit location data is missing.

In today's digital financial landscape, managing loan defaulters has become more critical than ever. With rising unsecured credit and remote borrower activity, identifying at-risk accounts and tracing unresponsive users is increasingly urgent.

Defaulters often disengage from traditional channels, making timely recovery difficult. Meanwhile, loan defaults in India have crossed ₹3.5 lakh crore, and conventional tracing methods relying on static contact details are no longer effective.

Task 1

Data Overview



Pie-chart depicting class imbalance in train-set

1. **Around 327,000 rows and 138 features**

2. **Highly Imbalanced Target: 292,000(0) vs 35,000 (1)**

3. **High Overall fill rates: Over 85% of Bureau score exists**

4. **Dataset includes the following category of features :**
   - Account & loan metadata
   - Credit/debt activity
   - Account utilization
   - Flags & indicators
   - Bureau scores

Data Preprocessing

Data preprocessing involves converting object columns to numerical types, applying one-hot encoding to binary flags, and using ordinal mapping to transform multi-label categorical columns into numeric form.

Outlier handling was performed using the Isolation Forest algorithm, which effectively identifies and isolates anomalies in the dataset based on patterns in feature distributions.

Missing values were treated by filling NPA-related columns with 0s where appropriate. KMeans clustering was used to impute columns with less than 15% missing data, and the model was applied to the test set.

Removed one of each pair of features with perfect correlation (Pearson = 1) to eliminate redundancy and improve model stability.

Isolation Forest

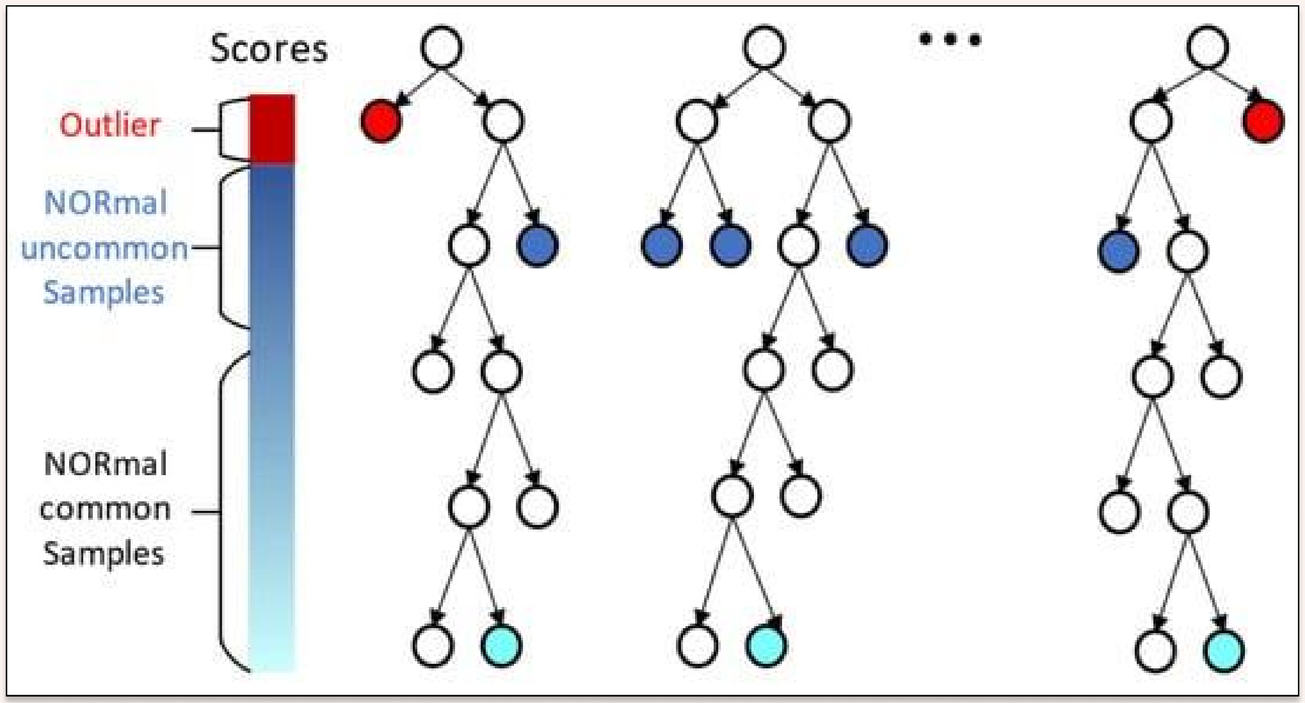
An unsupervised anomaly detection algorithm.

How It Works ?

- Randomly select a feature
- Randomly select a split value within that feature
- Repeat splits to build isolation trees
- Anomalies get isolated faster (shorter paths)

Isolation Forest V/S Simple Methods

- Captures multivariate outliers — detects anomalies based on combinations of features, not just individual values.
- No distribution assumptions — works well with non-normal and skewed data.
- Scalable and automated — handles large, high-dimensional datasets without manual threshold tuning.



K Means for Imputation

K-Means groups similar rows using complete features, allowing context-aware imputations. While traditional imputation (mean/median) ignores feature relationships, this helps preserve internal structure of the data.

How It Works ?

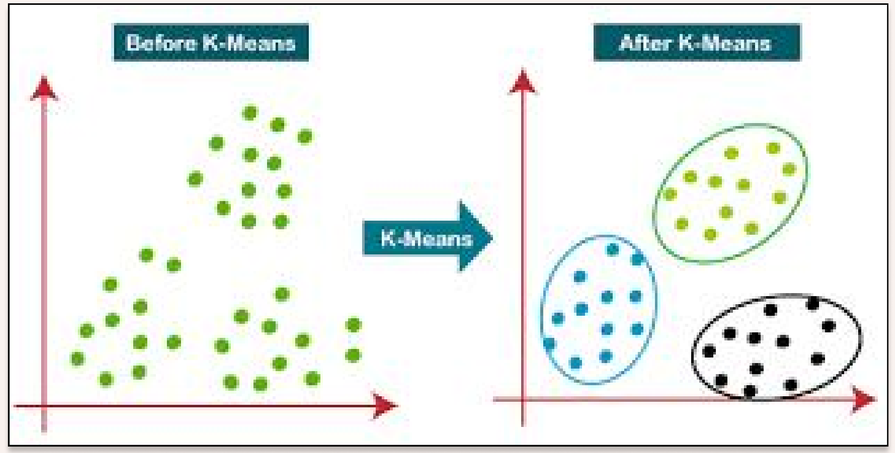
Select rows with no missing values.
Class-wise clustering:

- Data split into defaulters (class 1) and non-defaulters (class 0).
- K-Means trained separately on each class to avoid majority class bias.

For missing rows:

- Assigned to nearest cluster (within their class) based on available values.
- Imputed using cluster-level mean/median.

Final trained K-Means models were used to impute test data.



Modelling Approaches

1 Random Forest

Type: Ensemble (Bagging)
✓ Robust and easy to interpret
✗ Slower and less accurate than boosting
✓ Good as a baseline model

2 Gradient Boost

Type: Ensemble (Boosting)
✓ Fast, highly optimized
✓ Handles missing values natively
✗ Complex to tune, high memory usage

3 LightGBM

Type: Boosting (leaf-wise)
✓ Very fast on large datasets
✓ Handles categorical features well
✗ Can overfit on small or noisy data

4 TabNET

Type: Neural Network with Attention
✓ Built-in interpretability.
✗ Requires GPU for efficiency.
✗ Underperformed on our structured data.

5 CatBoost

Type: Ordered Gradient Boosting
✓ Highest F1 score in our evaluation
✓ Handles categorical & missing data natively
✓ Minimal preprocessing required
✗ Slightly slower than LightGBM but more stable
✓ Best tradeoff between performance and interpretable

Final Model

Why CatBoost?

- Boosting algorithm optimized for tabular data
- Handles:
  - Categorical features natively
  - Missing values automatically
- Requires minimal preprocessing

How We Used It

- Class imbalance tackled using class_weights='balanced'
  → Ensures defaulters (minority class) are not ignored
- Optuna used for optimal threshold tuning
  → Maximized F1 score instead of relying on default 0.5
- Hyperparameter tuning via *GridSearchCV*

```
Threshold  |  F1-score
------------------------
Threshold: 0.3000 | F1-score: 0.5362
Threshold: 0.3500 | F1-score: 0.5606
Threshold: 0.4000 | F1-score: 0.5825
Threshold: 0.4500 | F1-score: 0.6031
Threshold: 0.5000 | F1-score: 0.6244
Threshold: 0.5500 | F1-score: 0.6425
Threshold: 0.6000 | F1-score: 0.6570
Threshold: 0.6500 | F1-score: 0.6674
Threshold: 0.7000 | F1-score: 0.6738
Threshold: 0.7500 | F1-score: 0.6756
Threshold: 0.8000 | F1-score: 0.6681
Threshold: 0.8500 | F1-score: 0.6433
Threshold: 0.9000 | F1-score: 0.5786

Best threshold: 0.7500 with F1-score: 0.6756
Accuracy (CatBoost, best threshold): 0.9260
```

Fig1 : Results after training

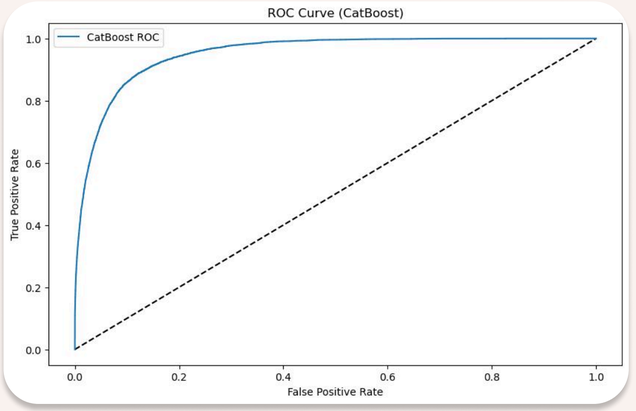


Fig3 :ROC-AUC Score: 0.9517

Tackling Imbalance Challenge:

- Accuracy is misleading due to class skew

Used Metrics:

- **F1 Score:** Balances precision & recall
- AUC-ROC Curve: Evaluates classifier performance across thresholds

```
Classification Report (CatBoost, best threshold):

              precision    recall  f1-score   support
-------------------------------------------------------
           0       0.97      0.95      0.96     55605
           1       0.64      0.71      0.68      6666

    accuracy                           0.93     62271
   macro avg       0.80      0.83      0.82     62271
weighted avg       0.93      0.93      0.93     62271
```

Fig2 : Classification report

Interpretability with SHAP

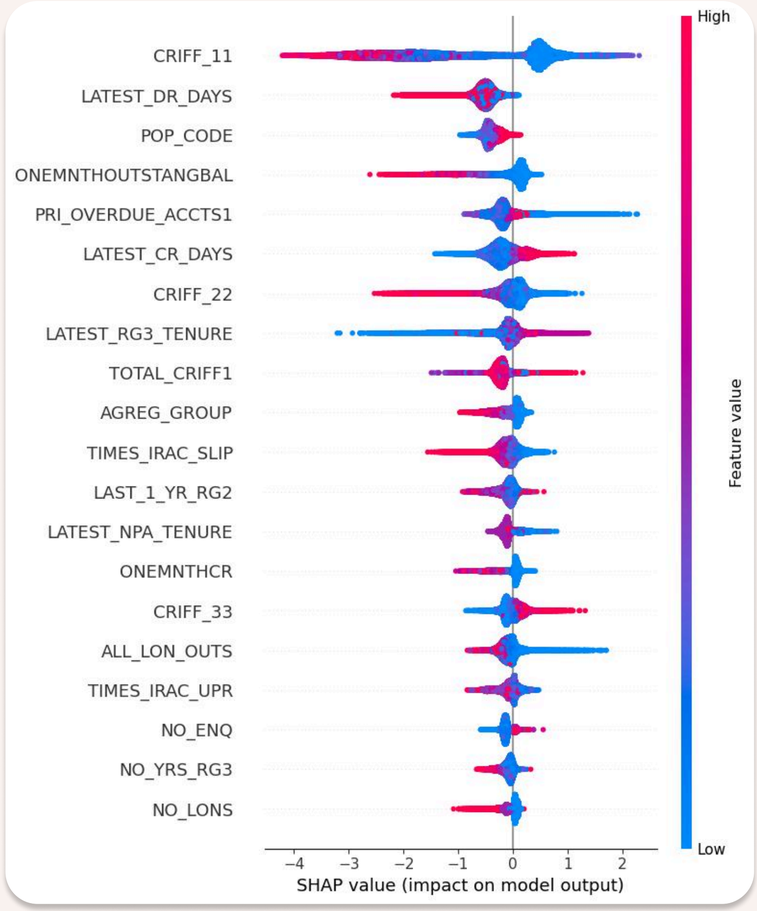
SHAP explains a model's prediction by showing how much each feature contributed to pushing the output higher or lower. It provides both local (per-sample) and global (overall) feature importance using game theory-based Shapley values.

Why Interpretability Matters

- In financial systems, especially credit default prediction, transparency is critical.
- Decisions like declining or flagging a cardholder must be explainable and justifiable.

Insights from SHAP in Our Model

- Top contributing feature: CRIFF_11 (Credit Bureau score in the last 1 month)
- This aligns with domain knowledge, as it's a core indicator of creditworthiness.
- Recommendation : To have high fill rate for Credit Bureau based features



Evaluation Metric

K-Means groups similar rows using complete features, allowing context-aware imputations. While traditional imputation (mean/median) ignores feature relationships, this helps preserve internal structure of the data.

How It Works ?

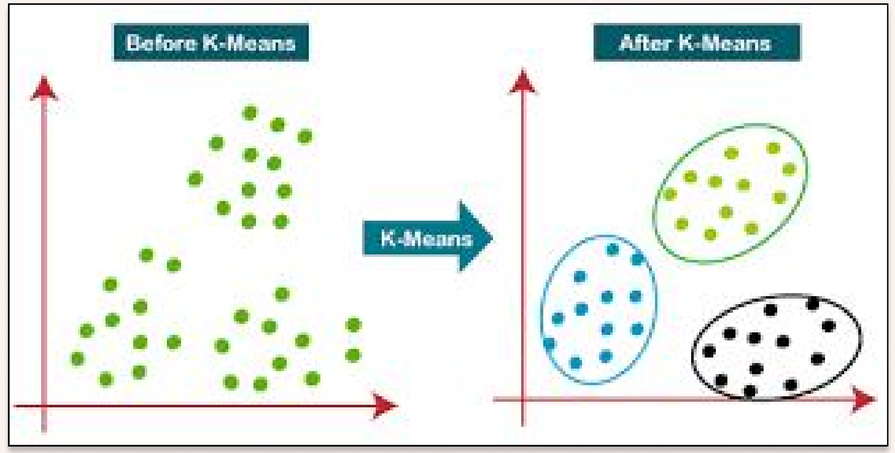
Select rows with no missing values.
Class-wise clustering:

- Data split into defaulters (class 1) and non-defaulters (class 0).
- K-Means trained separately on each class to avoid majority class bias.

For missing rows:

- Assigned to nearest cluster (within their class) based on available values.
- Imputed using cluster-level mean/median.

Final trained K-Means models were used to impute test data.



Task 2

Problem Overview

Given a defaulter's history of activity through digital means—including mobile connectivity and merchant transactions—the goal is to estimate their most probable **last known location** based on available proximity and behavioral data.

1

Approach 1 : This method improves location accuracy by inserting virtual towers based on merchant transactions, which serve as reliable location indicators. It filters out noisy transitions between cell towers and infers missing coordinates, resulting in a cleaner, more complete user mobility trace by combining tower data with contextual location signals.

2

Approach 2: User events are spatially clustered, and each event is weighted using time decay (favoring recent events) and night-time boosts (indicating likely home locations). Weighted centroids are calculated for each cluster, and the one with the highest average weight is selected as the most significant or representative user location.

The proposed solution utilizes a combination of tower connection logs and merchant transaction data to estimate the most probable last known location of a defaulter. Each cell tower and merchant point is modeled as a location node, and user activity across these nodes is chronologically arranged to form a behavioral timeline.

Input Data

Tower-Connection Logs:
device_id, timestamp, *tower_id*, optional latitude/ longitude

Merchant Logs:
Similar structure, marked with a merchant flag

Output Data

Returns the *top 5 tower locations,* based on probability with the highest scores within a *one-hour* window from the current time

Technicalities

Nodejs environment is used with multiple API's to infer last known defaulter location

Common Issues encountered:

1. Incomplete tower coordinate data.
2. Logs with gaps and sudden unrealistic transitions.
3. Merchant logs not tied to towers.

Given , the common issues we encountered we devised another approach in case, this is not feasible to logistical and technical constraints. We used the inference of Part 1 to add risk flags in the software system.

filterNoisyTowerLogs(logs, coords)

Removes unrealistic tower transitions where the distance is >50 km in under 5 minutes.
Improves data quality by filtering out location anomalies caused by tower hopping.

insertMerchantVirtualTowers(logs)

Fills time gaps between tower logs using merchant transactions as virtual tower entries.
Ensures better location continuity by capturing possible activity without tower signals.

inferMissingTowerCoords (towerCoords, logs)

Estimates missing location coordinates for towers based on neighboring known towers.
Uses an average sum approach to infer locations with reasonable accuracy.

inferLocation(logs, towerCoords)

Processes tower and merchant logs to compute a score for each location node.
Returns top 5 towers, the most probable current tower, and next likely tower.

The logic filters logs based on **time-of-day proximity**, identifying entries within a **one hour window** relative to the current time, irrespective of the actual date. The node with the **highest aggregate score** is identified as the most probable last known location of the defaulter.

1 Total connection time

Longer durations of connectivity at a specific location reflect a stronger likelihood of the defaulter being present there.
This metric captures habitual behavior, indicating locations where the individual tends to stay for extended periods.

2 Visit Frequency

A higher number of connections to a particular node suggests regular visits or repeated proximity.
This frequency reinforces spatial relevance, highlighting areas integral to the defaulter's movement pattern.

3 Night-Time Connectivity

Connections logged during night hours often correlate with residential locations.
Such temporal behavior helps distinguish between daytime transit points and potential home bases.

4 Merchant Interaction Proximity

If a merchant transaction occurs near a tower or location node, it strengthens the likelihood of physical presence.
This is especially useful for validating connection data with actual commercial activity.

5 Recency of Last Activity

More recent interactions carry higher predictive value for current presence estimation.
This temporal proximity increases the confidence in identifying the last known location.

1 Time spent

Higher score if the user stayed longer at a location continuously.

2 Visit Frequency

Incremented each time the node appears in the logs.

3 Night-Time Connectivity

Nodes active during night hours (8 PM–6 AM) are scored higher as possible home locations.

4 Merchant Interaction Proximity

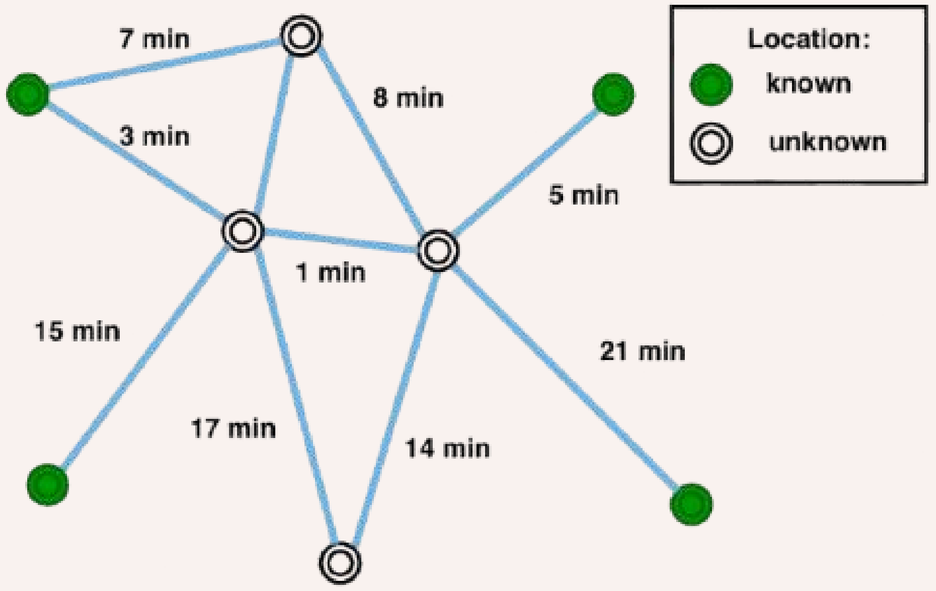
Merchant logs boost score of nearby towers, reinforcing visited areas with real-world transactions.

5 Recency of Last Activity

Boosts score based on how recently the node was visited. More recent = higher weight.

Final Scoring Algorithm

```
score =
      0.35 * Math.log(s.totalDuration + 1) +
      0.25 * s.count +
      0.15 * s.nightCount +
      0.15 * s.merchantHits +
      0.1 * Math.exp(-recency / 3600);
```



Frontend:

Framework & Styling: React.js, Tailwind CSS
Mapping & UI: React Leaflet (interactive maps)

Backend:

Runtime & Framework: Node.js, Express.js
API: REST API for scoring and location prediction logic

Deployment:

Frontend: Render / Vercel
Backend: Railway / Heroku

Mapping & Visualization:

Library: Leaflet.js
Base Maps: OpenStreetMap
Use Case: Visualizing tower and merchant locations

Additional Tools & Data Sources:

NetMonster App: Extracts real-time tower data (Cell ID, LAC, signal strength)
OpenSignal App: Cross-verifies live tower connections and signal metrics

The proposed solution infers the most probable last known location of a defaulter using a chronological sequence of events from the SBI digital ecosystem, including SBI Online logins, UPI transactions, and SBI App opens. Each event is tagged with a latitude, longitude, and timestamp, and contributing to a behavioral timeline for the user.

Input Data

- **SBI Login Events:** *device_id, timestamp, latitude, longitude*
- **UPI Transactions:** Similar structure, marked with a upi flag
- **SBI App Opens:** Similar structure, marked with a sbi app flag

Output Data

Returns a single most probable location cluster per user, including:

Latitude, Longitude, Confidence score, Number of events in the cluster, Cluster ID

Technicalities

Django environment with custom DBSCAN and APIs used to infer last known defaulter location.

Predictions are weighted based on event type, time decay over the ***last 72 hours***, and a ***night-time boost***. The final prediction selects the cluster with the highest weighted confidence.

Multiple location clusters can exist within a short timeframe, making it challenging to identify the most relevant one.

process_kalman_cluster_fusion (event_data)

Orchestrates the full processing pipeline by clustering coordinates, grouping events by user, merging outputs, and generating summary statistics, cluster metadata, and per-user predictions.

simpleDBSCAN (eps = 0.01°, min_samples = 2)

- Lightweight Python implementation of DBSCAN
- Groups near-by points into clusters, labels noise explicitly

process_user_data (user_events, event_weights)

Applies time decay over a 72 h window: weight = base_weight × (1 – Δ/72), with a night boost (×1.2) for events between 22:00–06:00.

Aggregates weights per cluster and computes weighted centroids

Selects the cluster with highest average weight as primary prediction

Records per-user summary (counts, clusters involved, time range)

These functions handle **event fusion** and **temporal weighting** to capture recent and behaviorally strong signals.

They compute weighted centroids from clustered events, ensuring consistent user-level predictions aligned with spatiotemporal patterns.

1 Event Weight Components

Base weights by event type
(UPI = 1.0, app_open = 0.8, login = 0.6)

Time decay: linear decrease over 72h from event timestamp

Night boost: ×1.2 for hours ∈ [22, 6]

2 Final Event Weight

$$w_{\text{decay}} = \max\left(0,\ 1 - \frac{\Delta t_{\text{hours}}}{72}\right)$$

$$w_{\text{night}} = \begin{cases} 1.2, & 22:00 \le t \le 06:00, \\ 1.0, & \text{otherwise.} \end{cases}$$

$$w_i = w_{\text{base}} \times w_{\text{decay}} \times w_{\text{night}}$$

3 Cluster confidence

$$C_{\text{cluster}} = \frac{1}{n} \sum_{i=1}^{n} w_{\text{base,i}}$$

$$C_{\text{overall}} = \frac{1}{U} \sum_{u=1}^{U} C_{\text{primary}}^{(u)}$$

4 Accuracy Metric

$$d_i = \sqrt{(lat_i - \hat{lat})^2 + (lon_i - \hat{lon})^2}$$

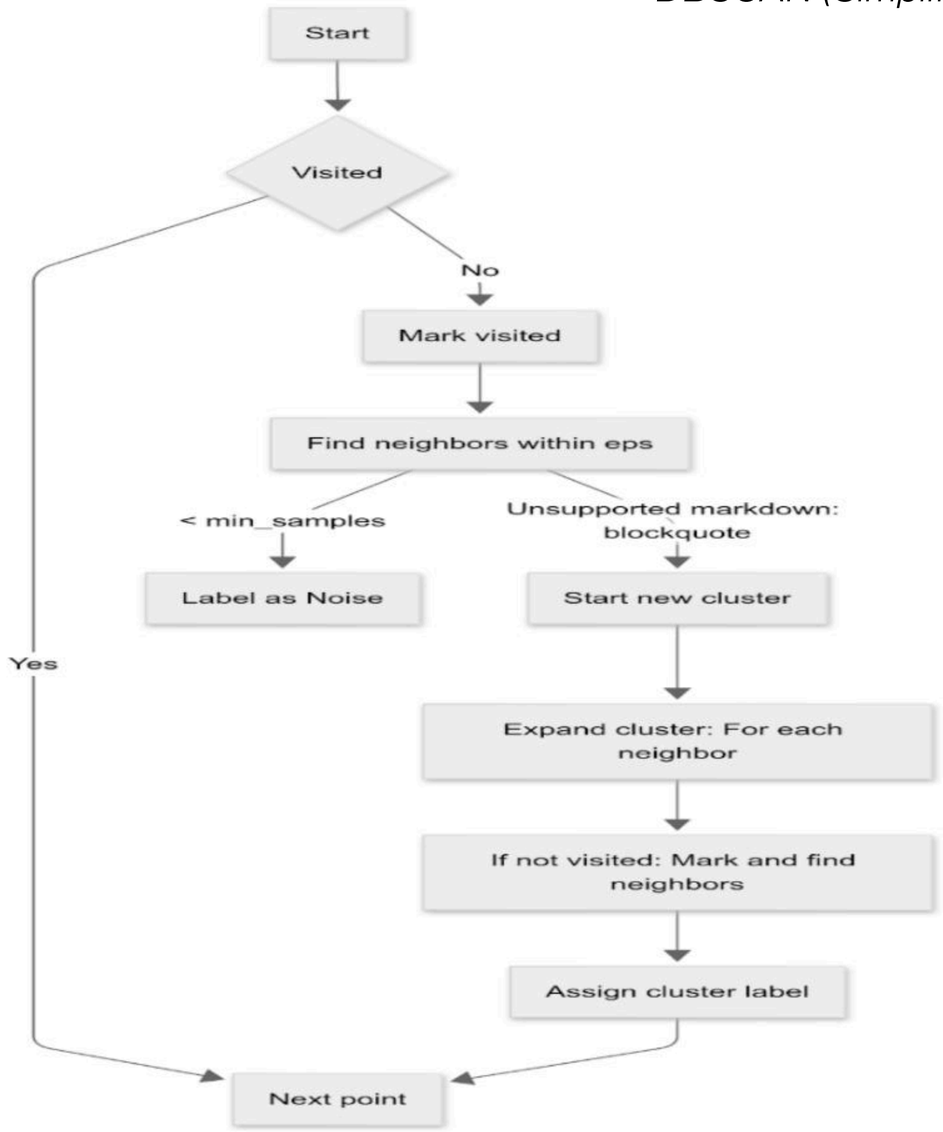
$$\bar{d} = \frac{1}{N} \sum_i d_i \qquad \text{accuracy} = \max\left(0,\ 1 - \frac{\bar{d}}{0.1}\right)$$

Our custom DBSCAN + weighted centroid offers a transparent, dependency-free, and interpretable pipeline.

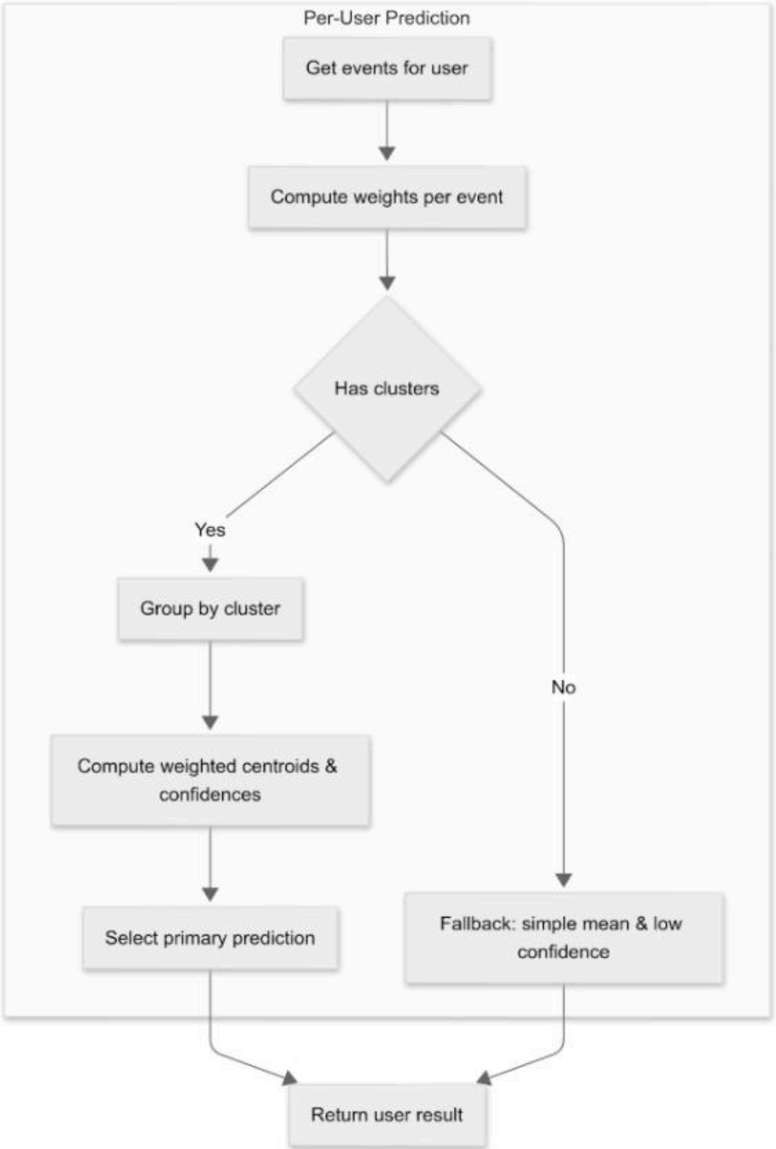
Location Inference Output View

APP_OPEN	Aug 02, 22:32	26.1927, 91.6988
UPI	Aug 02, 22:32	26.1927, 91.6988
UPI	Aug 02, 16:06	26.1927, 91.6988
APP_OPEN	Aug 02, 16:06	26.1927, 91.6988

DBSCAN (Simplified)



Per-User Weighting & Prediction



Backend:

Framework: Django 4.2.7 (Python 3.9+)
Database: SQLite3 (default; swappable to PostgreSQL/MySQL)
Clustering: Custom implementation of DBSCAN in pure Python (SimpleDBSCAN)
Math Operations: Pure Python (SimpleMath), no external libraries
Timezone Handling: Python zoneinfo, django.utils.timezone
Serialization: Custom JSON encoder (NumpyEncoder)

Deployment:

Platform: PythonAnywhere (optimized for free-tier limits)
Server: WSGI via Django

Frontend:

Tech: HTML5, CSS3, JavaScript (ES6+)
Styling & Templating: Bootstrap 5, Jinja2/Django Templates

Notable Constraints:

No heavy dependencies: Avoids numpy, pandas, scikit-learn, etc.

Wrap Up

The project focused on two key tasks using digital activity data from the SBI ecosystem.

Task 1: Defaulter Prediction

Developed a classification pipeline to identify defaulters in a highly imbalanced financial dataset. CatBoost demonstrated the best performance, with SHAP analysis highlighting CRIFF_11 as the most influential feature.

Missing values were handled using class-wise K-Means clustering, and model evaluation focused on F1 Score and AUC-ROC.

Task 2: Location Prediction for Defaulters

Designed two approaches to estimate the last known location of users from event-based activity data.

Approach 1 enriches tower logs by inserting virtual towers from merchant activity, filters noisy transitions, and infers missing coordinates.

Approach 2 clusters user events and applies time decay and night-time boosts to compute weighted centroids, selecting the cluster with the highest average weight.

*The final system achieved consistent predictions with an average confidence score of **0.85**.* The combined system is modular, interpretable, and lightweight, with no heavy dependencies.

Thank you for your patience!

We are open to Questions now!