CENTER FOR DEVELOPMENT OF
ADVANCED COMPUTING

Report on Personal Financial Advisor

PG-DBDA Feb 2025

Submitted by: Project Team 3

Shubham Talele

Prathamesh Gavali

Jai Juneja

Pankaj Chauhan

Muskan Chauhan

Shubham Aphashinkar

Mrunali

Project Guide:

1. INTRODUCTION

A Personal Financial Advisor system is a web-based tool that turns a person's raw income and expense data into clear, actionable guidance. First, it cleans large bank-statement files with PySpark to remove duplicates, fill in missing values, and convert every rupee spent into easy-to-compare percentages of income. Next, two machine-learning models go to work: a Random-Forest model estimates the user's true monthly savings rate, while a LightGBM classifier checks if any spending patterns indicate overspending. The results appear instantly on a colourful dashboard—bar and pie charts show where the money goes, a gauge highlights the gap between desired and actual savings, and a traffic-light message tells whether the user is within healthy limits. A built-in Gemini-powered chatbot then answers follow-up questions such as "How can I cut transport costs?" Finally, the whole application is containerised and hosted on Microsoft Azure, so anyone can open a single link on phone or laptop and receive personalised, data-driven financial advice in seconds.

Models used in this project :

1. Random Forest Regressor
2. LightGBM Classifier

2. Problem Statement

Individuals set savings goals but rarely know their *actual* monthly savings rate or which expense categories silently push them over budget. Raw bank/expense data is messy and often too large for spreadsheets, making manual tracking unreliable. There is no simple, real-time tool that can (1) clean and aggregate this data at scale, (2) predict a user's true savings percentage, (3) detect overspending patterns, and (4) present clear, actionable advice through an easy web interface. Our project addresses this gap by building a PySpark-based data pipeline, machine-learning models (for savings rate prediction and overspending classification), and a cloud-hosted Flask application with interactive visuals and a chatbot to deliver instant, personalized financial guidance.

3. LITERATURE SURVEY

3.1 Introduction

"What other people think" has always influenced personal financial decisions—whether it is a friend's advice on budgeting apps or an expert's blog on savings rules. With the rise of the Internet, research papers, fintech blogs, open-source notebooks and community forums now make it easy to learn from strangers' experiences. At the same time, many users openly publish their expense trackers and ML code. This section reviews prior work and existing approaches that inspired our Personal Financial Advisor system.

3.2 Existing Methods

We group the literature and industry practices into six streams: rule-based budgeting tools, ML savings prediction, overspend detection, big-data ETL, financial dashboards, and conversational assistants.

3.2.1 Rule-Based Budgeting & Tracking Apps

Mint, YNAB, Walnut, ET Money and similar apps categorise spends using merchant rules and alert users on limit breaches. Limitations:

- Mostly heuristic (50-30-20 rule) rather than predictive.
- Feedback is post-fact; no forecasting of savings rate.
- CSV imports are limited; large raw dumps are hard to process.

Relevance: Motivated us to add **predictive ML** and scalable preprocessing (PySpark), not just static tracking.

3.2.2 ML for Savings-Rate Prediction

Academic works on household economics often use **linear regression or tree ensembles** to model savings. Findings show tree-based models capture non-linear interactions between category ratios and income better than linear baselines.

Our takeaway: Adopt **Random Forest Regression** for robust, explainable savings-rate prediction.

3.2.3 Overspending / Anomaly Detection

Expense anomalies are treated as classification problems. Studies comparing Logistic Regression, SVM, and Gradient Boosting show **LightGBM/XGBoost** achieve higher F1 on imbalanced financial datasets.

Choice: LightGBM—faster training, built-in class weighting, high accuracy.

3.2.4 Big-Data Cleaning with PySpark

Case studies recommend Spark for finance logs due to:

- Distributed handling of millions of rows.
- Easy feature creation (withColumn) and imputation at scale.
- Delta/Parquet for versioned, query-efficient storage.

Impact: We built a **PySpark ETL** to future-proof data volume growth.

3.2.5 Visual Analytics for Personal Finance

Interactive visuals (gauges, treemaps) increase comprehension of budget gaps. Plotly enables rich charts embedded directly in Flask without external BI servers.

Decision: Use **Plotly** for bar/pie/treemap/gauge charts injected into HTML templates.

3.2.6 Conversational Finance Bots

LLM-based chatbots (Gemini, GPT) can answer budgeting questions but must be guarded against hallucinations and privacy leakage.

Implementation: A lightweight Gemini REST integration; we send only sanitized prompts and catch exceptions.

5. **LIBRARIES / TOOLS USED IN OUR PROJECT**

1. PySpark

PySpark is the Python API for Apache Spark. We used it to **clean and preprocess large expense datasets** efficiently.

What it lets us do:

- Read huge CSV/Parquet files (spark.read.csv, spark.read.parquet)
- Remove duplicates, handle NULLs, and outliers at scale (dropDuplicates, na.fill)
- Create new ratio features with withColumn (e.g., pct_rent = Rent/Income)
- Save cleaned data to Delta/Parquet for fast reuse

Example:

python

CopyEdit

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, lit

spark = SparkSession.builder.appName("finance_clean").getOrCreate()
df = spark.read.csv("expenses.csv", header=True, inferSchema=True)

df_clean = (df.dropDuplicates()
        .na.fill({"Rent":0, "Insurance":0})
        .withColumn("pct_rent", col("Rent")/col("Income")))
df_clean.write.mode("overwrite").parquet("clean_finance.parquet")
```

2. Pandas

Pandas is used for **tabular data manipulation** once the heavy lifting is done. Ideal for forming DataFrames that go into the model.

We used Pandas to:

- Convert form inputs into model-ready DataFrames
- Do quick exploratory data analysis (describe, correlations)
- Merge/reshape smaller datasets

Answers questions like:

- “What’s the mean savings rate?”
- “How many users have loans?”
- “What is the correlation between eating_out% and savings?”

3. NumPy

NumPy powers **fast numerical operations** under the hood.

Key benefits:

- Efficient array math for feature scaling and transformations
- Random number generation (train/validation splits)
- Basis for Pandas and scikit-learn operations

4. Scikit-learn (sklearn)

Scikit-learn is our main **machine learning toolkit** for classical models.

In our project it provides:

- **RandomForestRegressor** to predict savings rate
- Train/test split, cross-validation, GridSearchCV

- Metrics like RMSE, MAE, R^2

Why we like it:

- Simple, consistent API
- Robust and well-tested algorithms
- Great for explainability (feature_importances_)

5. LightGBM

LightGBM is a **fast, gradient-boosting framework** especially good for classification with large/imbalanced data.

Used for:

- Classifying **overspending vs not overspending**
- Handling skewed class distribution with class_weight='balanced'
- Fast training without GPUs

Why choose LightGBM:

- Faster than XGBoost on CPU
- High F1-score on our data
- Leaf-wise growth captures complex interactions

6. Flask

Flask is our **web framework** for the backend.

We used Flask to:

- Expose /predict (POST) for form submissions
- Expose /chat (POST JSON) for the Gemini chatbot
- Render HTML templates (index.html, result.html) with Jinja2

- Serve interactive Plotly charts directly in the response

Core files: app.py, templates/, static/

7. Plotly

Plotly creates **interactive charts** that embed easily into HTML.

Charts we generate:

- **Bar chart** — amount per category
- **Pie chart** — percentage breakdown
- **Treemap** — hierarchical view of spending
- **Gauge** — compares actual savings vs target

All charts are inserted as HTML blocks via plotly.offline.plot(..., output_type='div').

8. Joblib

Joblib is used to **save and load ML models** efficiently.

In practice:

python

CopyEdit

import joblib

rf = joblib.load("savings_rate_model_rf_optimized.pkl")

lgb = joblib.load("lightgbm_overspending_model.pkl")

It keeps deployment fast (no retraining) and model files portable.

9. Requests (Gemini API)

requests is a simple HTTP library used to call the **Gemini REST API** for chatbot responses.

Usage:

python

CopyEdit

payload = {"contents":[{"parts":[{"text": user_msg}]}]}

res = requests.post(GEMINI_API_URL, headers=HEADERS, json=payload)

We parse the JSON reply and return it to the user in the chat window.

10. python-dotenv

python-dotenv loads secrets (API keys) from a .env file without hardcoding them.

Why:

- Keeps Gemini API key safe
- Easy to manage environment-specific config

Example:

python

CopyEdit

from dotenv import load_dotenv

load_dotenv()

API_KEY = os.getenv("GEMINI_KEY")

(Optional) 11. HTML / CSS / JavaScript

Though not Python libraries, they're crucial for the **frontend**:

- **HTML5**: Forms to collect user inputs
- **CSS (style.css)**: Modern, responsive styling
- **JavaScript**: Toggle chatbox, call /chat asynchronously, auto-scroll messages

5. ALTERNATE MODELS AND THEIR DISADVANTAGES

Overspending Classification – Model Comparison

Model	Best Params (main)	F1 (class 1)	Recall (class 1)	Precision (class 1)	Confusion Matrix (TN FP / FN TP)	Verdict
Logistic Regression	C = 0.2, liblinear, class_weight=balanced	**0.3347**	0.8538	0.21	35305 11235 / 506 2954	Baseline only; low precision
Random Forest	n_estimators=50, max_depth=15, min_samples_leaf=4, …	**0.5821**	**1.0000**	0.41	41571 4969 / 0 3460	High recall but too many FPs
LightGBM (**Chosen**)	n_estimators=200, max_depth=−1, lr=0.1, class_weight=balanced	**0.5916**	0.9948	0.42	41805 4735 / 18 3442	Best F1 + near-perfect recall + fastest

Why LightGBM?

- **Highest F1 (0.5916)** ⇒ best balance of precision & recall on imbalanced data.
- **Recall ≈ 0.995** ⇒ hardly misses true overspenders.
- Faster & easier to tune than XGBoost/CatBoost on CPU.
- Built-in handling of class imbalance (class_weight='balanced').

One-line report sentence:

"Among Logistic Regression, Random Forest and LightGBM, the LightGBM classifier achieved the best F1-score (0.5916) with a recall of 0.9948 on the overspending class; hence, we selected LightGBM for the production overspending model."

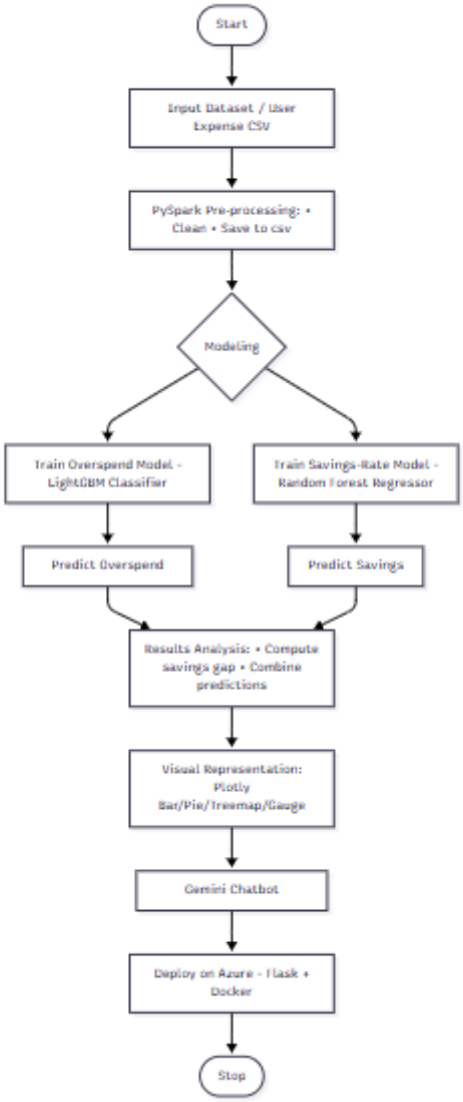
Savings-Rate Regression – Model Comparison (for completeness)

Model	RMSE	MAE	R^2	Verdict
GradientBoostingRegressor	0.0358	0.0277	0.8653	Good, but worse than RF
LightGBM Regressor	0.0314	0.0245	0.8961	Strong, still behind RF RMSE
Random Forest Regressor (Chosen)	0.0283	0.0218	0.9155	Best RMSE/R^2, robust & explainable

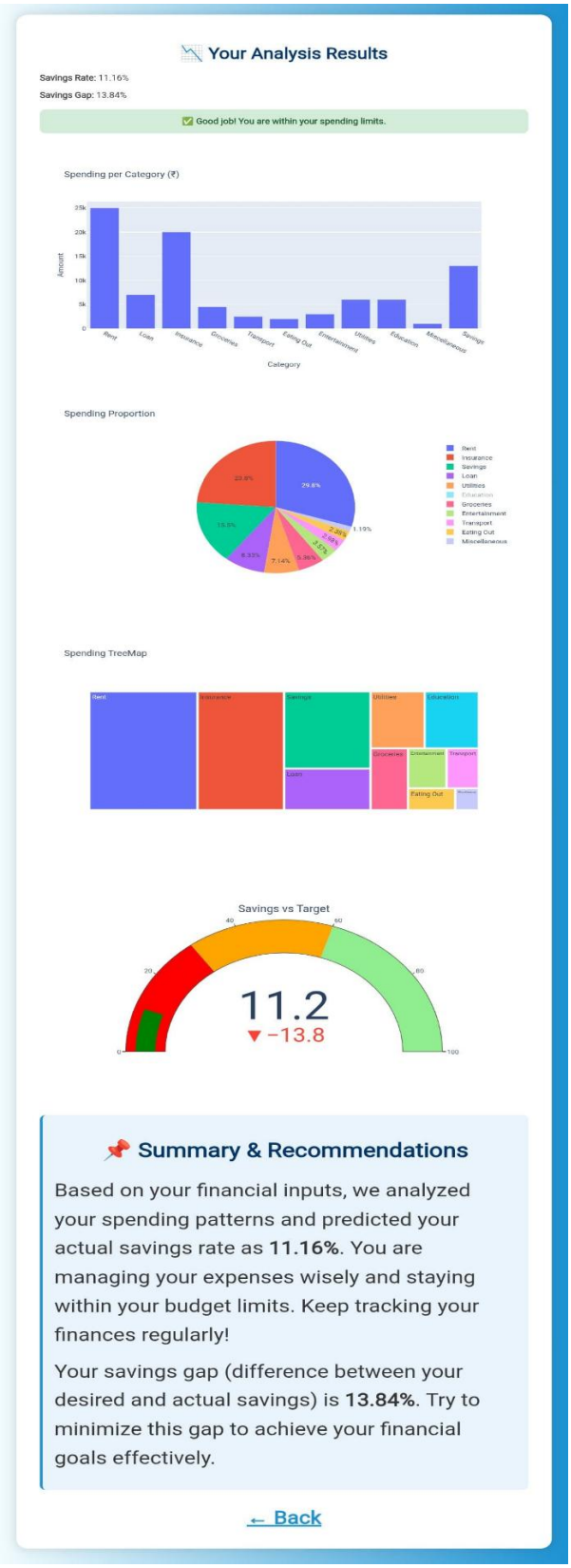
One-line report sentence:

"Random Forest Regressor produced the lowest RMSE (0.0283) and highest R^2 (0.9155), therefore it was adopted for predicting the savings percentage."

6. FLOWCHART


Start
Input Dataset / User Expense CSV
PySpark Pre-processing: • Clean • Save to csv
Modeling
Train Overspend Model - LightGBM Classifier
Train Savings-Rate Model - Random Forest Regressor
Predict Overspend
Predict Savings
Results Analysis: • Compute savings gap • Combine predictions
Visual Representation: Plotly Bar/Pie/Treemap/Gauge
Gemini Chatbot
Deploy on Azure - Flask + Docker
Stop

6. VISUALIZATIONS


Your Analysis Results
Savings Rate: 11.16%
Savings Gap: 13.84%
Good job! You are within your spending limits.
Spending per Category (₹)
Amount
Category
Rent
Loan
Insurance
Groceries
Transport
Eating Out
Entertainment
Utilities
Education
Miscellaneous
Savings
Spending Proportion
Spending TreeMap
Savings vs Target
11.2
▼ −13.8
Summary & Recommendations
Based on your financial inputs, we analyzed your spending patterns and predicted your actual savings rate as 11.16%. You are managing your expenses wisely and staying within your budget limits. Keep tracking your finances regularly!
Your savings gap (difference between your desired and actual savings) is 13.84%. Try to minimize this gap to achieve your financial goals effectively.
← Back

7. MODELING

```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from lightgbm import LGBMClassifier
from sklearn.metrics import classification_report, confusion_matrix, f1_score,
recall_score

# ✅ Define features & target
X = df[selected_features]
y = df["Overspend_Flag"]

# 🔀 Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# ✅ Define models and hyperparameters
models = {
  "LogisticRegression": {
    "pipeline": Pipeline([
      ("scaler", StandardScaler()),
```

```
        ("model", LogisticRegression(solver='liblinear', class_weight='balanced'))
    ]),
    "params": {
        "model__C": [0.001, 0.01, 0.05, 0.1, 0.2, 0.5]
    }
},
"RandomForest": {
    "pipeline": Pipeline([
        ("model", RandomForestClassifier(random_state=42, class_weight='balanced'))
    ]),
    "params": {
        "model__n_estimators": [50, 100, 150],
        "model__max_depth": [5, 10, 15],
        "model__min_samples_split": [5, 10],
        "model__min_samples_leaf": [4, 6],
        "model__max_features": ["sqrt"]
    }
},
"LightGBM": {
    "pipeline": Pipeline([
        ("model", LGBMClassifier(random_state=42, class_weight='balanced'))
    ]),
    "params": {
```

```
        "model__n_estimators": [100, 150, 200],
        "model__max_depth": [5, 10, -1],
        "model__learning_rate": [0.05, 0.1]
      }
    }
}

# 🧪 Train and Evaluate all models
best_f1 = 0
best_model = None

for name, cfg in models.items():
    print(f"\n🔍 Training {name}...")
    grid = GridSearchCV(cfg["pipeline"], cfg["params"], cv=3, scoring="f1", verbose=1,
n_jobs=-1)
    grid.fit(X_train, y_train)
    y_pred = grid.predict(X_test)

    f1 = f1_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)

    print(f"✅ {name} F1 Score: {f1:.4f} | Recall: {recall:.4f}")
    print("📋 Best Params:", grid.best_params_)
```

```
    print("📊 Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
    print("📄 Classification Report:\n", classification_report(y_test, y_pred))

    if f1 > best_f1:
        best_f1 = f1
        best_model = (name, grid.best_estimator_)

# ✅ Final selected model
print(f"\n🏆 Best Model: {best_model[0]} with F1 Score: {best_f1:.4f}")
```

2ND : spendings rate

```
from lightgbm import LGBMRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# 🔧 LightGBM pipeline (no scaling needed, but keeping for comparison)
pipeline_lgbm = Pipeline([
    ("model", LGBMRegressor(
        n_estimators=200,
        max_depth=10,
        learning_rate=0.1,
        num_leaves=31,
```

```
        min_child_samples=20,
        subsample=0.8,
        colsample_bytree=0.8,
        random_state=42
    ))
])

pipeline_lgbm.fit(X_train, y_train)
y_pred = pipeline_lgbm.predict(X_test)

print("📌 LightGBM →",
    "RMSE:", mean_squared_error(y_test, y_pred, squared=False),
    "| MAE:", mean_absolute_error(y_test, y_pred),
    "| R²:", r2_score (y_test, y_pred))
```

8. APP CREATION (FLASK)

What is Flask (for our app)?

Flask is a lightweight, open-source Python web framework that lets you turn ML code into a real web application with only a few files. It gives routing (/predict, /chat), templating (HTML with Jinja2), and easy API creation—perfect when you need full control but don't want a heavyweight framework.

Why did we use Flask instead of Streamlit?

- **Full front-end control** (custom HTML/CSS/JS, Plotly embeds, floating chatbot button).
- **Easy to integrate multiple models & REST endpoints** (savings %, overspend flag, Gemini chat).
- **Production friendly** (works well with Gunicorn/Docker/Azure App Service).
- **Lightweight & fast**—no hidden UI layer, we decide exactly what's served.

Key Benefits for Data/ML Projects

- No need to learn big frameworks like Django.
- Simple to expose a POST /predict that accepts form/JSON and returns results.
- Works smoothly with Pandas, scikit-learn, LightGBM, Plotly, PySpark outputs.
- Jinja2 templating lets you drop Plotly chart s right into HTML.

1. Create & activate venv

```
python3 -m venv venv

source venv/bin/activate   # Windows: venv\Scripts\activate


# 2. Install deps

pip install -r requirements.txt    # contains Flask, plotly, scikit-learn, lightgbm, joblib,
requests, python-dotenv


# 3. Run locally

python app.py     # starts at http://0.0.0.0:8080


project/
│ app.py
│ requirements.txt
├─ models/
│   ├─ savings_rate_model_rf_optimized.pkl
│   └─ lightgbm_overspending_model.pkl
├─ templates/
│   ├─ index.html
│   └─ result.html
└─ static/
    └─ style.css
```

App.py :

```
@app.route("/")
def index():
    return render_template("index.html")


@app.route("/predict", methods=["POST"])
def predict():
    # 1) Read form data
    # 2) Build DataFrames
    # 3) rf_model.predict(), lgb_model.predict()
    # 4) Create Plotly charts -> div strings
    return render_template("result.html", ...)


@app.route("/chat", methods=["POST"])
def chat():
    # Call Gemini REST API with requests.post(...)
    return jsonify({"reply": reply_text})
```

Deployment (Azure)

1. **Dockerize** the app (Python 3.12 slim image, copy code, pip install).
2. Push image to Azure Container Registry / Docker Hub.
3. Create **Azure App Service (Linux Container)** → point to image.
4. Set environment variables (Gemini key) in Azure portal.
5. Enable HTTPS, scale up if needed.

10. CONCLUSION & FUTURE SCOPE

Conclusion

We built a Personal Financial Advisor system that cleans raw expense data with PySpark, predicts a user's actual savings rate using a Random Forest Regressor, and detects overspending patterns with a LightGBM Classifier. The results are delivered through a Flask + Plotly web interface, enriched by a Gemini-powered chatbot for instant guidance, and hosted securely on Microsoft Azure. Our models achieved strong performance (RF $R^2 \approx 0.91$; LightGBM F1 ≈ 0.59 with ~0.99 recall for overspenders), proving that classical ML with good feature engineering can provide accurate, interpretable financial insights. Overall, the project demonstrates that it is feasible to transform scattered personal finance data into actionable advice in real time.

Future Scope

- Live Bank/API Integration: Connect to Plaid/Yodlee to auto-ingest transactions instead of manual CSV uploads.
- Investment & Goal Planning: Recommend SIP amounts/asset allocation using MPT or risk-profiling models.
- Personalised Nudges: Reinforcement learning or rule-based alerts to reduce specific overspending categories.
- Mobile App & Voice Assistant: React Native/Flutter app and voice bot for hands-free budgeting help.
- Advanced Explainability: SHAP/LIME dashboards so users see *why* they were flagged as overspending.
- Multi-tenant SaaS & Auth: Azure AD B2C/Keycloak for secure multi-user deployment and billing.
- Data Privacy & Encryption: End-to-end encryption, differential privacy for aggregated analytics.

- Gamification & Community Benchmarks: Badges, streaks, and percentile comparisons to motivate saving habits.
- Streaming Pipeline: Kafka/Spark Streaming for real-time spend monitoring and instant alerts.
- Multilingual & Regionalization: Local language chatbot responses and city-specific cost benchmarks.

11.REFERENCES

● **Dataset / Data Source**

- User expense CSV + form inputs (cleaned with PySpark)
- (If you cite an open source base) National Sample Survey – Household Consumer Expenditure (India)
- Spark sample finance data (for ETL testing)

● **Models**

1. **Random Forest (Regression & Classification)**
   https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/
2. **LightGBM (Gradient Boosting for Overspend Classification)**
   https://lightgbm.readthedocs.io/
3. **Logistic Regression (Baseline Classifier)**
   https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
4. **Gradient Boosting Regressor (Tried for savings %)**
   https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting
5. **Decision Tree (Baseline for both tasks)**
   https://scikit-learn.org/stable/modules/tree.html

● **Big-Data / ETL**

- **PySpark**
  https://spark.apache.org/docs/latest/api/python/

● **Web & Visualization Stack**

- **Flask** (Backend web framework)
  https://flask.palletsprojects.com/
- **Plotly** (Interactive charts)
  https://plotly.com/python/

● **Model Persistence & Utilities**

- **joblib** (Model save/load)
  https://joblib.readthedocs.io/
- **requests** (Gemini REST calls)
  https://requests.readthedocs.io/

● **Chatbot / LLM**

- **Google Gemini API**
  https://ai.google.dev/gemini-api/docs