ADSA Data Analytics Project — Final Report

Report index (Table of contents)

---

1. Project overview and objective

A retail company “ABC Private Limited” wants to understand the customer purchase behaviour (specifically, purchase amount) against various products of different categories. They have shared purchase summary of various customers for selected high volume products from last month.\ The data set also contains customer demographics (age, gender, marital status, city_type, stay_in_current_city), product details (product_id and product category) and Total purchase_amount from last month.

Now, they want to build a model to predict the purchase amount of customer against various products which will help them to create personalized offer for customers against different products.

Success criteria. e.g., target metric (RMSE, MAE, R2, Accuracy), baseline performance, and production readiness requirements (unit tests, logging, packaging).

---

2. Step 1 — GitHub & code setup (expanded)

2.1 Environment creation (conda + venv)

Use an isolated environment to ensure reproducibility. Create a conda environment in the project folder and activate it.

Typical commands to run:

- `conda create -p ./venv python==3.8 -y`
- `conda activate ./venv`
- Install dependencies from `requirements.txt` after creating it: `pip install -r requirements.txt`

Note: Use explicit pinned versions in `requirements.txt` for reproducibility.

2.2 Git initialization & remote

Initialize a repository, create README and .gitignore, add files, commit, set the main branch and add a remote.

Typical git commands to run:

- `git init`
- `git add README.md .gitignore`
- `git commit -m "first commit"`
- `git branch -M main`
- `git remote add origin https://github.com/Manoj-Sh-AI/ADSA_Data_Analytics-1_ml_pipeline.git`
- `git push -u origin main`

2.3 Project packaging (setup.py)

If you plan to install the project as a package during development, create a `src/` package and a minimal `setup.py`. Use `pip install -e .` to install locally during development.

2.4 Requirements

Keep `requirements.txt` updated and pin versions. Example packages to include: pandas, numpy, scikit-learn, xgboost, catboost, joblib, pyyaml, pytest.

---

3. Step 2 — Error handling, logging & project structure

3.1 Project structure (high-level)

(See Appendix A for full skeletal tree.)

Key folders and responsibilities:

- `src/components/`: modular units (data_ingestion, data_transformation, model_trainer)
- `src/pipeline/`: orchestrates end-to-end flow (train_pipeline, predict_pipeline)
- `src/config/`: YAML/JSON with paths, hyperparameters, and constants
- `src/logs/`: runtime logs
- `tests/`: unit & integration tests

3.2 Logging

Use Python `logging` configured to write to both console and a file in a `logs/` folder. Provide a helper (e.g., `get_logger`) that sets up formatters, stream handler and file handler. Call the logger at the top of each module to standardize messages and timestamps.

3.3 Exception handling

Create a custom exception class to attach context and preserve tracebacks. Wrap high-level pipeline functions with try/except blocks, log the exception with stack trace (using logger.exception), and re-raise or handle with meaningful messages.

---

4. Step 3 — Data Ingestion

4.1 Goals

- Read raw data from CSV / BigQuery / local storage
- Validate schema (expected columns)
- Save raw snapshot (versioned) and a cleaned copy for transformations

4.2 Data ingestion API (recommended behaviour)

Design data ingestion to:

- Accept configuration (paths for raw, train and test)
- Read the raw dataset

- Perform a deterministic train/test split (e.g., 80/20, with a fixed seed)
- Persist the train and test sets to `data/processed/`
- Log operations and raise meaningful exceptions on failure

4.3 Validation checks

- Column presence
- Missing value proportions per column
- Duplicate rows
- Data types and ranges (e.g., dates, numeric ranges)

---

5. Step 4 — Exploratory Data Analysis (EDA)

5.1 Objectives

- Understand distributions, missingness, correlation with target
- Detect outliers and data quality issues
- Identify candidate feature transformations

5.2 Recommended EDA checklist

- Summary statistics (describe and info)
- Missing value heatmap and percent-missing table
- Univariate plots for numeric features (histograms / boxplots)
- Countplots for categorical variables
- Correlation matrix / heatmap for numeric relationships
- Target vs feature plots (scatter, violin, bar) for top predictors
- Time-series checks if data is temporal

5.3 Tools

Pandas and Matplotlib (or Seaborn for convenience) are recommended. Conduct EDA interactively in Jupyter notebooks and save relevant visuals to `reports/figures/`.

---

6. Step 5 — Data Transformation & Feature Engineering

6.1 Goals

- Convert raw features into model-ready features
- Handle missing data, encode categoricals, scale numeric features

- Persist transformers (scalers, encoders) with joblib for inference

6.2 Typical steps

1. **Imputation**: numeric (mean/median/KNN), categorical (mode or new category "Missing")
2. **Categorical encoding**: One-Hot for low-cardinality, Target/Ordinal or Count encoding for high-cardinality
3. **Scaling**: Standard scaling for models sensitive to scale, MinMax for neural networks
4. **Feature creation**: interaction terms, datetime decompositions, aggregated features
5. **Dimensionality reduction** (if needed): PCA, feature selection via tree-based importance

6.3 Implementation notes

- Build a preprocessing pipeline object that encapsulates imputation, encoding and scaling and fit it on the training data only.
- Persist the fitted preprocessing object to `artifacts/` so it can be reused by the prediction pipeline.

---

7. Step 6 — Model Training (Why I chose these models and how I achieved the accuracy)

7.1 Modeling strategy

- Try multiple models: Linear Regression (baseline), RandomForest, XGBoost, CatBoost, and a simple neural net if needed.
- Use cross-validation (KFold, or TimeSeriesSplit if temporal) and compare with metrics (RMSE, MAE, R2). Keep a validation set for final selection.

7.2 Why these models?

- **Linear Regression**: interpretable baseline, fast.
- **RandomForest**: strong baseline for tabular data, robust to outliers and feature scaling.
- **XGBoost/CatBoost**: state-of-the-art for many tabular problems; often give best performance with moderate tuning.

7.3 Hyperparameter tuning

Use randomized or grid search for hyperparameter tuning; for faster, use Bayesian optimizers (Optuna). Keep `n_jobs=-1` where possible and use sensible search spaces.

7.4 Evaluation

Report train/validation/test metrics and show learning curves. Record metrics in a `reports/` file and compare several candidate models in a table.

7.5 Persisting the model

Persist the trained model artifact to `artifacts/` (e.g., `model.joblib`) for later inference.

---

8. Step 7 — Prediction / Inference pipeline

8.1 Design

- Provide a single entry point (predict pipeline) that accepts raw input (CSV or JSON), applies the saved preprocessing object, loads the model artifact, and outputs predictions (CSV/JSON).
- Validate input schema and produce outputs with predictable column names (e.g., `prediction`).

8.2 Inference behaviour

- Load preprocessor and model artifacts from `artifacts/`.
- Transform the raw input using the preprocessor and call `model.predict()`.
- Return or persist predictions alongside input identifiers.

8.3 Deployment considerations

- For a demo: expose via a lightweight API (Flask/FastAPI).
- For production: containerize, add health checks, secure secrets, and add monitoring.

---

9. Reproducibility & How to run the pipeline

9.1 Quick start (development)

Typical steps to run locally:

1. Clone the repo.
2. Create the environment and install requirements.
3. `conda create -p ./venv python==3.8 -y`
4. `conda activate ./venv`
5. `pip install -r requirements.txt`
6. Run the training pipeline (script entry point) and the prediction script with input and output paths. Ensure `config/` contains correct paths and seeds.

---

10. Results summary & discussion

Include here a short table with final metrics for the chosen model and the baseline. Also include: feature importances, top 5 features, and saved visuals (learning curves, residual plots) placed in `reports/figures/`.

Example columns to report: Model, Dataset, RMSE, MAE, R2.

---

11. Conclusion, limitations & future work

- **Conclusion:** Which model you selected and why, summary of results.
- **Limitations:** data quantity/quality, label noise, potential leakage, or limited features.
- **Future work:** more feature engineering, ensembling, better cross-validation, model explainability (SHAP), productionizing with CI/CD.

---

12. File Structure

Appendix A — Skeletal file structure

```
ADSA_Data_Analytics-1_ml_pipeline/
├─ .gitignore
├─ README.md
├─ requirements.txt
├─ setup.py
├─ config/
│  └─ config.yaml
├─ data/
│  ├─ raw/
│  ├─ processed/
│  └─ external/
├─ src/
│  ├─ __init__.py
│  ├─ logger.py
│  ├─ exception.py
│  ├─ utils.py
│  ├─ components/
│  │  ├─ __init__.py
│  │  ├─ data_ingestion.py
│  │  ├─ data_transformation.py
```

```
│   │   └─ model_trainer.py
│   ├─ pipeline/
│   │   ├─ __init__.py
│   │   ├─ train_pipeline.py
│   │   └─ predict_pipeline.py
│   └─ config/
│       └─ params.yaml
├─ artifacts/
│   ├─ preprocessor.joblib
│   └─ model.joblib
├─ reports/
│   ├─ figures/
│   └─ report.pdf
└─ tests/
    └─ test_data_ingestion.py
```

Appendix B — Sample scripts & snippets

- `train_pipeline` should: load config → run data_ingestion → run data_transformation → run model_trainer → save artifacts and metrics.
- `predict_pipeline` should: parse args → validate input → load artifacts → run predict → save results.
