ML Report (Clean Dataset + Stacked Ensemble + Threshold Optimization)

Project Goal

Predict **ProdTaken (0 = No, 1 = Yes)** using customer and pitch-related features. The main challenge is that **“Yes” is the minority class**, so the model must prioritize **high recall for Yes** without sacrificing overall accuracy.

1) Data preparation

1.1 Raw data cleaning → `c5a_clean.csv`

I started from `data.csv` and applied consistency fixes so categorical columns don’t create noisy / duplicated categories during encoding.

Steps performed

A) Remove rare / abnormal category

- Dropped all rows where **Occupation = “Free Lancer”**
- Reason: this category was extremely rare and creates instability (models can overfit to tiny groups, and one-hot encoding would create sparse columns for it)

B) Fix typos and standardize text

- **Gender**
  - Fixed variations like `"fe male"`, `"female"` (case-insensitive) → `"Female"`
- **MaritalStatus**
  - Converted `"Unmarried"` (case-insensitive) → `"Single"`

C) No label encoding at this stage

- Although a `LabelEncoder` is imported, all label-encoding lines are commented out.
- That means the dataset remains “clean but unencoded” at this stage.
- Reason: label encoding can accidentally impose fake ordering on categories. Your pipeline later uses **one-hot encoding**, which is safer for nominal categories.

Output

- Saved as: `data/class5assignment/c5a_clean.csv`

1.2 Train-test split (90/10) → `train_data.csv` and `test_data.csv`

After cleaning, I split the dataset into training and test sets.

Split design

- Target: `ProdTaken`
- Features: all columns except `ProdTaken`
- Split ratio: **90% train / 10% test**
- Random seed: `random_state=42` (ensures reproducibility)

Files saved

- Train: `data/class5assignment/train_data.csv`
- Test: `data/class5assignment/test_data.csv`

Why this matters

- The test set remains unseen during training, so performance reflects real generalization (not memorization).

2) Analysis (what I observed)

2.1 Class imbalance (core problem)

In this dataset, **ProdTaken = 1 (Yes)** is much less frequent than **ProdTaken = 0 (No)**.

Effect if not handled

- A model can get “high accuracy” by predicting “No” for almost everything.
- But that would produce poor **recall for Yes**, which defeats the goal of identifying likely buyers.

So the whole pipeline is designed to:

- reduce bias toward majority class (No),
- boost minority detection (Yes),
- and evaluate recall explicitly.

2.2 Mixed feature types require careful handling

Your columns include:

- **Numerical** (Age, MonthlyIncome, DurationOfPitch, etc.)
- **Binary flags** (Passport, OwnCar, etc.)
- **Categorical strings** (Occupation, ProductPitched, Designation, etc.)

This combination creates two common issues:

1. **Categorical noise** (typos / inconsistent naming) → fixed in the cleaning step
2. **Nonlinear interactions** (income + designation + product pitched) → handled by feature engineering + ensemble models

2.3 Outliers in continuous variables can distort learning

Columns like:

- `MonthlyIncome`, `DurationOfPitch`, `NumberOfTrips`

can contain extreme values that dominate learning and reduce generalization.
That's why your pipeline applies **99th percentile capping** before modeling.

3) Feature extraction (why + what features were created)

Your feature engineering was built to convert real-world decision patterns into numeric signals that models can learn.

3.1 Outlier capping (99th percentile)

Applied to:

- `DurationOfPitch`
- `NumberOfTrips`
- `MonthlyIncome`

Why

- Prevents extreme values from pulling decision boundaries.
- Improves stability for both neural networks and tree-based models.

3.2 Group composition + affordability signals

Feature: `Adults`

- `Adults = NumberOfPersonVisiting - NumberOfChildrenVisiting`
  Why
- A group of 4 with 0 children is different from a group of 4 with 3 children (spending patterns differ).

Feature: `IncomePerPerson`

- `IncomePerPerson = MonthlyIncome / (Adults + 1)`
  Why
- A single monthly income supports different purchasing power depending on how many adults are in the group.

3.3 Age–income relationship features

Feature: `Income_to_Age_Ratio`

- `MonthlyIncome / Age`
  Why
- Captures income relative to life stage (higher ratio can indicate stronger discretionary spending).

Feature: `Income_Seniority`

- `MonthlyIncome * Age`
  Why
- Represents combined effect of seniority + earning power (often correlates with spending capacity).

3.4 Luxury alignment features (tier mapping)

You created an interpretable “luxury logic” by mapping **Designation** and **ProductPitched** into tiers.

Tier mapping

- `Designation_Tier`: Executive(1), Manager(2), Senior Manager(3), AVP(4), VP(5)
- `Product_Tier`: Basic(1), Deluxe(2), Standard(3), Super Deluxe(4), King(5)

Features built from tiers

A) LuxuryIndex

- `LuxuryIndex = Designation_Tier * Product_Tier * (MonthlyIncome / 1000)`
  Why
- People with higher seniority + higher pitched product + higher income are more likely to buy.

B) IncomePerTier

- `IncomePerTier = MonthlyIncome / (Product_Tier + 1)`
  Why
- Models whether the pitched product level is realistic relative to income.

3.5 Logical “readiness” interactions

These encode signals that become meaningful *when combined*:

- `Passport_Car_Interaction = Passport * OwnCar`
  - Travel readiness + lifestyle flexibility

- `Followup_Passport_Interaction = Passport * NumberOfFollowups`
    - Follow-up intensity matters more when the customer is travel-ready
- `PropDuration_Income = PreferredPropertyStar * MonthlyIncome`
    - Preference for higher star properties becomes more meaningful at higher income

3.6 Encoding strategy (categoricals → one-hot)

After feature creation:

- Categorical columns are cast to string
- One-hot encoding is applied using `pd.get_dummies(..., drop_first=True)`

Why

- Avoids fake ordering (unlike label encoding).
- Works well with ensembles and neural nets when combined with scaling.

4) Building model (architecture, optimizers, loss, training logic)

4.1 Preprocessing before modeling

Steps applied in training:

1. Load `train_data.csv`
2. Apply `clean_data()` feature engineering
3. One-hot encode categorical columns
4. Scale features using **StandardScaler**
5. Balance classes using **manual oversampling**
6. Train stacked ensemble

4.2 Manual oversampling (class balancing)

After scaling, you build a temporary dataframe and oversample:

- Majority class: `ProdTaken = 0`
- Minority class: `ProdTaken = 1`
- Minority is upsampled with replacement until it matches majority count

Why

- Forces the model to learn patterns for Yes instead of treating Yes as noise.
- Helps recall for minority class.

4.3 Core model: Deep Stacked Ensemble

You trained a **StackingClassifier** with strong, diverse base models:

Base estimators

1. **RandomForestClassifier**
    - `n_estimators=1000, max_depth=25`
2. **ExtraTreesClassifier**
    - `n_estimators=1000, max_depth=25`
3. **GradientBoostingClassifier**
    - `n_estimators=500, learning_rate=0.03, max_depth=10`
4. **HistGradientBoostingClassifier**
    - `max_iter=500, learning_rate=0.03, max_depth=12`
5. **Neural Network (MLP)** via custom wrapper

Meta learner (final estimator)

- **RandomForestClassifier**
    - `n_estimators=500, max_depth=10`

Stacking configuration

- `cv=5` cross-validation inside stacking
- `n_jobs=-1` uses all CPU cores available

Why stacking worked well here

- Tree models capture nonlinear feature interactions well.
- Neural net captures smoother continuous patterns.
- The meta model learns how to combine them optimally.

4.4 MLP architecture (hidden layers)

The MLP component is:

- Input: number of encoded features
- Dense(512, ReLU) → BatchNorm → Dropout(0.4)
- Dense(256, ReLU) → BatchNorm → Dropout(0.3)
- Dense(128, ReLU) → BatchNorm
- Dense(1, Sigmoid)

Training setup:

- Epochs: **100**
- Batch size: **64**
- Validation split: **0.15** (inside training)
- Metric: accuracy

4.5 Loss function: Focal Loss

You used **focal loss** with:

- `alpha = 0.25`
- `gamma = 2.0`

Why focal loss

- In imbalanced problems, many negatives are “easy”.
- Focal loss down-weights easy examples and focuses on harder ones.
- This helps improve learning for minority class (Yes).

4.6 Optimizer

- **Adam optimizer**
- Learning rate: **0.001**

4.7 Saving the model bundle

You saved:

- the trained stacking model
- the scaler
- the expected one-hot column list

Output:

- `output/c5a_clean_model_bundle.joblib`

This is critical because inference must match training preprocessing exactly.

5) Evaluation results (accuracy + recall)

5.1 What inference does differently (important)

During inference you do **threshold optimization** instead of fixed 0.5:

1. Predict **probabilities** for Yes: `predict_proba[:, 1]`
2. Use **Precision–Recall Curve**
3. Choose threshold that maximizes F1
4. Apply the best threshold to convert probabilities → class predictions

Best threshold found

- **Optimal Threshold = 0.176**

This explains why recall can be very high: the model is allowed to classify “Yes” at a lower probability cutoff than 0.5.

5.2 Final confusion matrix (from your image)

Raw confusion matrix:

	Pred No	**Pred Yes**
Actual No	**264**	**1**
Actual Yes	**3**	**54**

So:

- TN = 264
- FP = 1
- FN = 3
- TP = 54
- Total test samples = 322

5.3 Final metrics (from your image + computed)

Overall

- **Accuracy = 0.9876** (318/322)
- **Weighted F1 = 0.9875**

Recall (most important for imbalanced target)

- **Recall (Yes) = TP / (TP + FN) = 54 / 57 = 0.9474 → 94.74%**
- **Recall (No) = TN / (TN + FP) = 264 / 265 = 0.9962 → 99.62%**

This matches the normalized confusion matrix:

- Actual No predicted correctly: **99.62%**
- Actual Yes predicted correctly: **94.74%**

Precision (useful interpretation)

- **Precision (Yes) = TP / (TP + FP) = 54 / 55 = 98.18%**
- This means when the model predicts “Yes”, it is correct almost all the time.

Errors summary

- Only **1 false positive** (predicted Yes but actually No)
- Only **3 false negatives** (missed Yes)

This is an excellent result for an imbalanced dataset because the model:

- keeps false alarms extremely low,
- while still catching most Yes cases.

5.4 Training curve interpretation (MLP plots)

From the MLP training plots:

- Accuracy steadily rises and stabilizes near ~0.98–0.99
- Loss drops sharply early and then flattens
- Validation curves track training curves closely → suggests **good convergence** and **limited overfitting** (at least for the MLP component)

Even though the final model is a stack, this plot confirms the MLP is learning useful patterns and not collapsing.

Final conclusion

This pipeline achieved **strong real-world performance** by combining:

1. **Data cleaning** (remove rare category + fix category typos)
2. **Feature engineering** (income, seniority, luxury alignment, and interactions)
3. **Balanced learning** (manual oversampling + focal loss for minority sensitivity)
4. **Powerful model design** (stacking multiple strong learners)
5. **Threshold optimization** (selecting the best cutoff instead of default 0.5)

Final test performance

- **Accuracy: 98.76%**
- **Recall (Yes): 94.74%**
- **Weighted F1: 98.75%**
- Threshold used: **0.176**