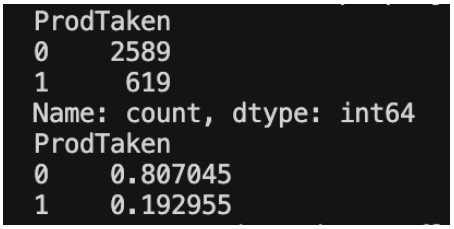
Data Preparation (how you have prepared the data, how you split the data etc)

I used class5data.csv which contains 3208 rows and 19 columns. The target label is ProdTaken (0/1). I checked data quality and found 0 missing values and 0 duplicate rows. The target distribution is imbalanced: 0 = 2589 (80.7%) and 1 = 619 (19.3%). I separated features (X) and target (y). There are 6 categorical columns, so I applied one-hot encoding, increasing the feature count from 18 to 29. I then split the data into 75% training and 25% testing using stratified splitting to keep the class ratio consistent (train = 2406 rows, test = 802 rows).

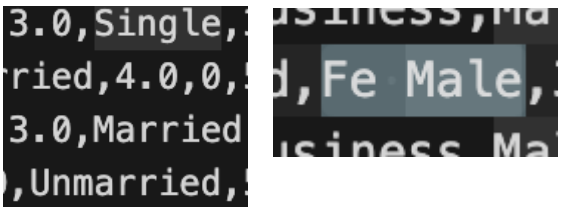
Analysis (What you observe in the data)

1. The target label ProdTaken is not balanced. Class 0 is much larger than class This means the model can get high accuracy by predicting mostly 0, so I must also report recall/F1 and not only accuracy.

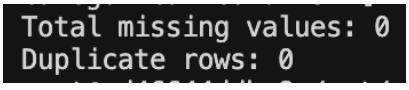


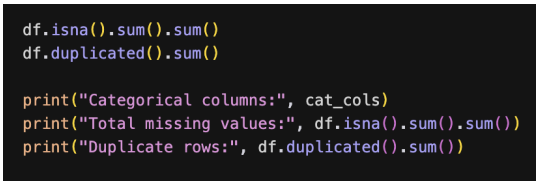
2. Some columns are numeric (e.g., Age, MonthlyIncome, DurationOfPitch), and some columns are text categories (e.g., Gender, Occupation, ProductPitched). A neural network cannot use text directly, so categorical columns need encoding (one-hot encoding).

3. Some text categories are not consistent. For example, single and unmarried mean the same thing, so they should be merged into one value to avoid confusing the model. Also, there are typos like fe male vs female, which should be fixed. If we do not fix these, one-hot encoding will create extra fake categories and the model may learn wrong patterns.



4. I checked the data quality. There are no missing values and no duplicate rows (or show the number if it exists). This means I can focus on encoding and modeling.

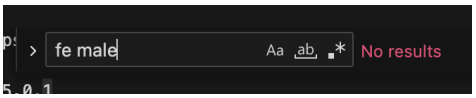


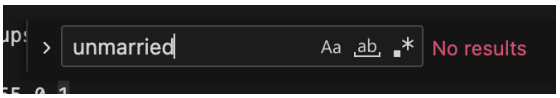


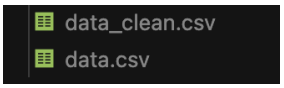
Feature Extraction and Engineering

1. Data Cleaning

   I cleaned inconsistent categorical values to prevent duplicated categories during one-hot encoding. Specifically, I fixed Gender: 'Fe Male' → 'Female' and merged MaritalStatus: 'Unmarried' → 'Single'. I also checked data quality and confirmed there were 0 missing values and 0 duplicate rows







2. Extract Features

   - Demographic: Age, Gender, MaritalStatus, MonthlyIncome, Designation, Occupation
   - Travel / background behavior: NumberOfTrips, Passport, OwnCar, CityTier
   - Interaction / sales process: TypeofContact, ProductPitched, PitchSatisfactionScore, NumberOfPersonVisiting, NumberOfFollowups, DurationOfPitch, PreferredPropertyStar

   Target (y): ProdTaken (0 = No, 1 = Yes)
   Features (X): all other columns (customer info + travel behavior + sales interaction)

   I used customer demographics, travel behavior, and sales interaction features to predict whether the customer takes the product (ProdTaken).

3. Splitting the dataset

   I split the dataset **before building the model** to make sure the evaluation is fair and to avoid data leakage.

   1. **Test split (25%)**
      First, I separated the data into:
      - **Train+Validation = 75%**
      - **Test = 25%**

   I used **stratified splitting** (stratify=y) because the target ProdTaken is imbalanced. Stratified split keeps the **same class ratio** (0/1) in both training and testing sets, so the test result is more reliable.

```
X_trainval, X_test, y_trainval, y_test = train_test_split(
    X, y_encoded, test_size=0.25, random_state=42, stratify=y_encoded
)
```

2. **Validation split (from the 75%)**
   After that, I split the 75% Train+Validation again into:

   - **Training set (80% of the 75%)**
   - **Validation set (20% of the 75%)**

This validation set is used for:

- monitoring val_loss during training (EarlyStopping)
- finding the **best probability threshold** that gives the best **F1 score** on validation (instead of using the test set)

This is important because I should not “tune” the model using the test set. The test set must be used only once for the final performance report.

```
# Split trainval into train/val (so we can tune threshold)
X_train, X_val, y_train, y_val = train_test_split(
    X_trainval, y_trainval, test_size=0.2, random_state=42, stratify=y_trainval
)
print(f"Split sizes -> Train: {len(X_train)}, Val: {len(X_val)}, Test: {len(X_test)}")
```

Building Model (How you build the model: hidden layers, optimizers, loss functions, etc.)

1. 1st try

What I built (Model Architecture):

I built a binary classification neural network to predict ProdTaken (0/1).

- Input: all engineered features after preprocessing (numeric + one-hot encoded categorical)
- Hidden layers:
  - Dense layer (32 units, ReLU)
  - Dropout (0.3) to reduce overfitting
  - Dense layer (8 units, Tanh)
  - Dense layer (4 units, ReLU)
  - Dropout (0.3)
- Output layer:
  - Dense (1 unit, Sigmoid) → outputs a probability between 0 and 1

How I trained it (optimizer, loss, metrics):

- **Optimizer:** Adam
  - Reason: good default optimizer that converges fast for tabular neural networks.
- **Loss function:** binary_crossentropy (baseline)
  - Reason: standard loss for binary classification, measures how far predicted probability is from the true label.
- **Metrics tracked during training:**
  - accuracy (overall correct %)
  - AUC (how well the model separates classes)

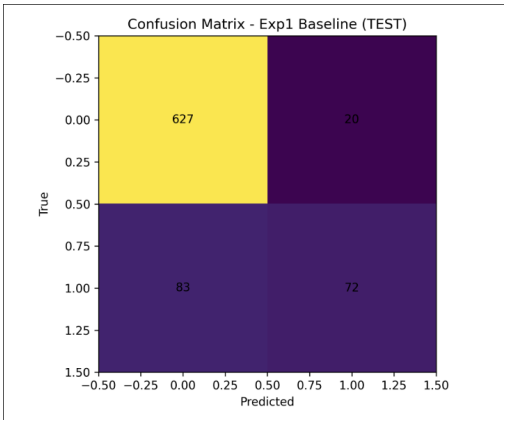
Training setup:

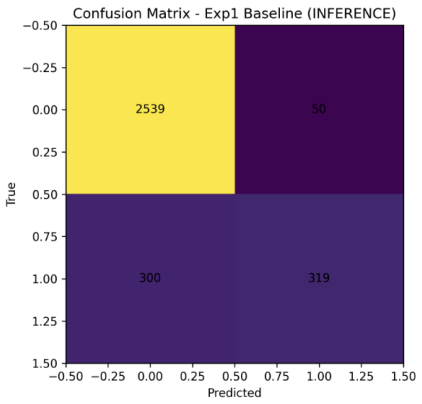
- **Train/Val/Test split:**
  - Train+Val = 75%, Test = 25% (stratified)
  - Then Train/Val split again (so validation can be used for tuning / early stopping)
- **Epochs:** 100
- **Batch size:** 32
- **Early stopping:** Early stopping: monitor **val_loss**, patience = **10**, restore best weights
  - Reason: stops training when validation stops improving, prevents overfitting.

Results :

```
============================================================
EXPERIMENT 1 (BASELINE) – TEST RESULTS
============================================================
Accuracy : 0.8716
Precision: 0.7826
Recall   : 0.4645
F1-score : 0.5830
AUC      : 0.8692

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.97      0.92       647
           1       0.78      0.46      0.58       155

    accuracy                           0.87       802
   macro avg       0.83      0.72      0.75       802
weighted avg       0.86      0.87      0.86       802


Confusion Matrix:
 [[627  20]
 [ 83  72]]
```

```
============================================================
INFERENCE (EXP1 BASELINE) – RESULTS
============================================================
Accuracy : 0.8909
Precision: 0.8645
Recall   : 0.5153
F1-score : 0.6457
AUC      : 0.9056

Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.98      0.94      2589
           1       0.86      0.52      0.65       619

    accuracy                           0.89      3208
   macro avg       0.88      0.75      0.79      3208
weighted avg       0.89      0.89      0.88      3208


Confusion Matrix:
 [[2539   50]
 [ 300  319]]
```





2. 2nd try

What I built (Model Architecture):

I built a binary classification neural network to predict **ProdTaken (0/1)**.

- **Input:** all processed features after preprocessing (numeric features + one-hot encoded categorical features, then scaled)
- **Hidden layers:**
  - Dense layer (**64 units, ReLU**)
  - Dropout (**0.3**) to reduce overfitting
  - Dense layer (**16 units, ReLU**)
  - Dropout (**0.2**) to reduce overfitting
- **Output layer:**
  - Dense (**1 unit, Sigmoid**) → outputs a probability between 0 and 1

How I trained it (optimizer, loss, metrics):

- **Optimizer:** Adam
  - Reason: good default optimizer, trains fast on tabular data

- **Loss function:** Binary Crossentropy (**binary_crossentropy**)
    - Reason: standard loss for binary classification (0/1)
- **Metrics tracked during training:**
    - **Accuracy** (overall correct %)
    - **AUC** (how well the model separates classes)

Training setup:

Train = **75%**, Test = **25%** (**stratified**)

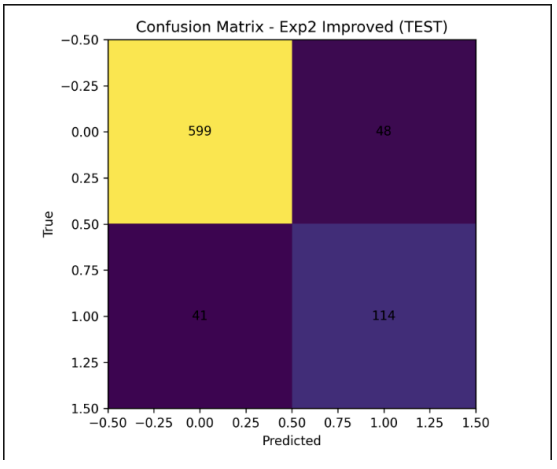
Reason: keep class ratio (0/1) similar in train and test

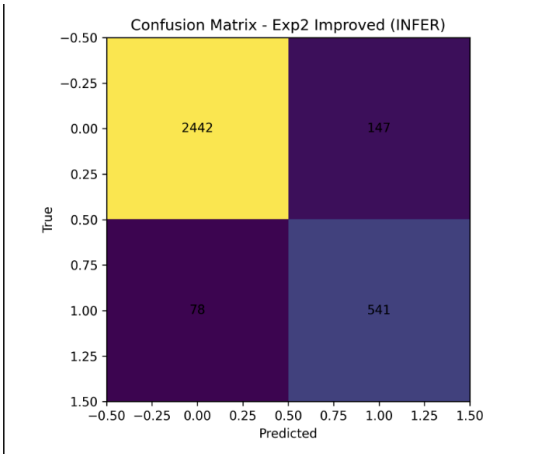
- **Validation during training:**
    - **validation_split = 0.2** (20% of training set used as validation inside .fit())
    - Reason: used for early stopping (check if model starts overfitting)
- **Epochs:** 100
- **Batch size:** 32
- **Early stopping:** monitor **val_loss**, patience = **10**, restore best weights
    - Reason: stop when validation no longer improves and keep the best model

Results:

```
============================================================
EXPERIMENT 2 (IMPROVED) — TEST RESULTS
============================================================
Threshold: 0.58
Accuracy : 0.8890
Precision: 0.7037
Recall   : 0.7355
F1-score : 0.7192
AUC      : 0.9047

Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.93      0.93       647
           1       0.70      0.74      0.72       155

    accuracy                           0.89       802
   macro avg       0.82      0.83      0.83       802
weighted avg       0.89      0.89      0.89       802


Confusion Matrix:
 [[599  48]
 [ 41 114]]
```

```
============================================================
INFERENCE (EXP2 IMPROVED)
============================================================
Threshold: 0.58
Accuracy : 0.9299
Precision: 0.7863
Recall   : 0.8740
F1-score : 0.8279
AUC      : 0.9648

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.94      0.96      2589
           1       0.79      0.87      0.83       619

    accuracy                           0.93      3208
   macro avg       0.88      0.91      0.89      3208
weighted avg       0.93      0.93      0.93      3208


Confusion Matrix:
 [[2442  147]
 [  78  541]]
```





What I changed from Version 1 → Version 2 (Improved):

From version 1 to version 2, I mainly changed the model setup to improve **F1-score** (not just accuracy) by focusing on the imbalanced target class. In version 1, the model used the default prediction threshold **0.50** and standard **binary crossentropy**, which often misses class 1 and gives low recall/F1. In version 2, I added **imbalance handling** (using **class weights** or **weighted binary crossentropy**) so the model pays more attention to class 1 mistakes, and I also used a proper **Train/Validation/Test split** so I could tune the **best threshold** on the validation set instead of guessing. After training, I selected the threshold that gives the highest validation F1 and then evaluated on the test set, showing improvement with evidence like the **classification report and confusion matrix** before vs after.

Evaluation & Results (Results in accuracy and recall)

In Experiment 1 (baseline), I trained a simple neural network with default binary cross-entropy and used the default decision threshold = 0.50. On the test set, the model got Accuracy = 0.8716 and AUC = 0.8692, but the F1-score was only 0.5830 because Recall was low (0.4645), meaning it missed many “ProdTaken = 1” cases (class imbalance problem). In Experiment 2 (improved), I focused on improving minority-class detection by tuning the threshold to 0.58 (chosen to improve F1) and using an imbalance-aware training approach, which increased positive predictions correctly. As a result, the test Recall increased to 0.7355 and test F1-score increased to 0.7192 (Accuracy = 0.8890, AUC = 0.9047). In the inference results, Experiment 2 also showed stronger overall separation (AUC = 0.9648) and much better balance between Precision and Recall (F1 = 0.8279) compared to Experiment 1 (F1 = 0.6457). Evidence: include the printed metric screenshots for both experiments + both confusion matrix images (baseline vs improved) to show the reduction in false negatives for class 1.