**1) train\_neural\_network.py — your end-to-end, production-leaning pipeline**

**a) Load + target setup + leakage control**

**Concept:** Load the cleaned data, make y numeric, and drop duration (leakage—only known after a call).

df = pd.read\_csv("data/processed/cleaned\_full\_data.csv", sep=',')

drop\_cols = ['duration']

X = df.drop(columns=['y'] + drop\_cols)

y = (df['y'] == 'yes').astype(int)

**Why:** If you keep duration, your model “cheats” and looks great on paper but fails in production.

**b) Split numeric vs categorical, then build a single preprocessor**

**Concept:** NN needs scaled numeric features and one-hot encoded categoricals—done neatly in a ColumnTransformer so train/inference match 1:1.

num\_cols = X.select\_dtypes(include=["int", "float"]).columns.tolist()

cat\_cols = X.select\_dtypes(exclude=["int", "float"]).columns.tolist()

preproc = ColumnTransformer([

('num', Pipeline([('scaler', StandardScaler())]), num\_cols),

('cat', Pipeline([('encoder', OneHotEncoder(handle\_unknown='ignore'))]), cat\_cols)

], remainder='drop')

X\_proc = preproc.fit\_transform(X)

**Why:** Prevents “false ordering” of categories and keeps the exact same transforms for production.

**c) SMOTE on the training data (after transform)**

**Concept:** Your target is imbalanced (~88% “no”). SMOTE creates **synthetic** “yes” points so the model sees enough positives to learn.

smote = SMOTE(random\_state=42)

X\_res, y\_res = smote.fit\_resample(X\_proc, y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_res, y\_res, test\_size=0.2, random\_state=42)

**Why not before split?** That would leak synthetic info into test data and inflate your metrics.

**d) Save the exact preprocessing artifacts (for reproducibility)**

**Concept:** Persist scaler/encoder/preproc so inference uses *identical* logic—no “works on my laptop” drama.

with open("models/neural\_network/preprocessor\_nn.pkl", "wb") as f:

pickle.dump(preproc, f)

**Why:** Reusing these artifacts guarantees the model sees the same feature space in prod.

**e) Define the neural net architecture**

**Concept (beginner-friendly):**

* **Input layer**—width equals the number of processed features.
* **Hidden layers**—Dense layers learn feature interactions; **ReLU** keeps gradients healthy; **BatchNorm** stabilizes training; **Dropout** randomly “drops” neurons to prevent memorization (overfitting).
* **Output layer (Sigmoid)**—outputs probability in [0,1] for binary yes/no.

model = models.Sequential([

layers.Input(shape=(X\_train.shape[1],)),

layers.Dense(128, activation='relu'),

layers.BatchNormalization(),

layers.Dropout(0.5),

layers.Dense(64, activation='relu'),

layers.Dropout(0.3),

layers.Dense(1, activation='sigmoid')

])

**Overfitting (plain English):** memorizing the training set (like rote learning) and then struggling with new questions. Dropout + EarlyStopping help stop that.

**f) Compile with beginner-friendly definitions**

**Concept:**

* **Loss: binary\_crossentropy**—penalizes wrong probabilities for yes/no. If true=1 and pred=0.1 → big penalty; pred=0.9 → small penalty.
* **Optimizer: Adam**—a smart weight updater; faster convergence than vanilla gradient descent.
* **Metrics: accuracy, AUC**—AUC is threshold-free and great for imbalanced targets.

optimizer = optimizers.Adam(learning\_rate=0.0005)

model.compile(optimizer=optimizer, loss='binary\_crossentropy', metrics=['accuracy', tf.keras.metrics.AUC(name='auc')])

**g) Train with EarlyStopping + ReduceLROnPlateau**

**Concept:** Stop when val loss stops improving (avoid overfitting) and lower the learning rate if progress stalls.

early\_stop = callbacks.EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

reduce\_lr = callbacks.ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3)

history = model.fit(X\_train, y\_train, validation\_split=0.2, epochs=100, batch\_size=64,

callbacks=[early\_stop, reduce\_lr], verbose=1)

**Overfitting refresher:** If training loss goes down but validation loss goes up, you’re memorizing noise—this halts it.

**h) Evaluate + plots you can show**

**Concept:** Confusion matrix, classification report, ROC/AUC—your proof.

y\_pred = (model.predict(X\_test) > 0.5).astype(int).ravel()

print("Classification report:\n", classification\_report(y\_test, y\_pred))

print("ROC AUC score:", roc\_auc\_score(y\_test, model.predict(X\_test)))

**i) Save the final model**

model.save("models/neural\_network/model\_nn.h5")

**j) SHAP explainability (and alternatives)**

**Concept:** SHAP tells you *how much each feature pushed a prediction up or down*—perfect for stakeholder trust.

explainer = shap.Explainer(model, X\_background, feature\_names=preproc.get\_feature\_names\_out())

shap\_values = explainer(X\_test\_dense)

shap.plots.bar(shap\_values)

**Alternatives:** LIME, permutation importance, feature importances (trees).  
**When I’d pick SHAP:** When you need **local** explanations per customer and a defensible story for a regulator or marketing lead.

**2) evaluation\_model\_comparison.py — apples-to-apples scoreboard**

**a) Load common test set + models + their preprocessors**

**Concept:** Fair comparison = test each model on the **same** test data using its **own** scaler/encoder.

X\_test = pd.read\_csv('data/X\_test.csv')

y\_test = pd.read\_csv('data/y\_test.csv')['y']

model\_dt = joblib.load('models/decision\_tree/model\_dt.pkl')

scaler\_dt = joblib.load('models/logistic\_regression/scaler\_dt.pkl')

model\_rf = joblib.load('models/random\_forest/model\_rf.pkl')

scaler\_rf = joblib.load('models/logistic\_regression/scaler\_rf.pkl')

model\_nn = load\_model('models/neural\_network/model\_nn.h5')

preprocessor\_nn = pickle.load(open('models/neural\_network/preprocessor\_nn.pkl', 'rb'))

**Why:** Each model expects a specific feature space; mixing preprocessors = garbage results.

**b) Predict each model correctly**

y\_pred\_dt = predict\_model(model\_dt, scaler\_dt.transform(X\_test))

y\_pred\_rf = predict\_model(model\_rf, scaler\_rf.transform(X\_test))

X\_test\_nn = preprocessor\_nn.transform(X\_test)

y\_pred\_nn = (model\_nn.predict(X\_test\_nn) > 0.5).astype(int).ravel()

**c) Collect and plot metrics**

df = pd.DataFrame({

"Model":["Decision Tree","Random Forest","Neural Network"],

"Accuracy":[accuracy\_score(...), ...],

"Precision":[precision\_score(...), ...],

"Recall":[recall\_score(...), ...],

"F1":[f1\_score(...), ...]

})

df.set\_index("Model").plot(kind="bar", figsize=(10,6))

**Interview angle:** “Same test, model-specific preprocessors, consistent metrics—this is a proper bake-off.”

**3) train\_random\_forest.py — quick, strong baseline (trees)**

**a) LabelEncode categoricals + scale features**

for col in df.select\_dtypes(include='object').columns:

le = LabelEncoder(); df[col] = le.fit\_transform(df[col])

features = df.drop('y', axis=1); target = df['y']

scaler = StandardScaler(); X\_scaled = scaler.fit\_transform(features)

**Note:** LabelEncoding imposes an *artificial order*; it’s fine as a quick baseline, but One-Hot is safer (what you did in NN). Good to mention this trade-off.

**b) Train/save model + artifacts**

rf\_model = RandomForestClassifier(n\_estimators=100, random\_state=42).fit(X\_train, y\_train)

joblib.dump(rf\_model, "models/random\_forest/model\_rf.pkl")

pickle.dump(scaler, open("models/logistic\_regression/scaler\_rf.pkl","wb"))

pickle.dump(label\_encoders, open("models/random\_forest/label\_encoder\_rf.pkl","wb"))

**Why save:** So inference uses the same encodings & scaling.

**4) train\_decision\_tree.py — interpretable baseline**

**Same pattern** as RF (LabelEncoder + StandardScaler + joblib/pickle). Simpler model; handy for sanity checks and quick “what splits did it learn?” discussions.

dt\_model = DecisionTreeClassifier(random\_state=42).fit(X\_train, y\_train)

joblib.dump(dt\_model, "models/decision\_tree/model\_dt.pkl")

**5) logistic\_regression\_bank\_marketing.py — classic linear baseline**

**a) Drop leakage, one-hot encode, scale**

# Drop 'duration' (leakage)

df = pd.get\_dummies(df, drop\_first=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(..., stratify=y)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

**b) Train + simple evaluation**

model = LogisticRegression(max\_iter=1000).fit(X\_train\_scaled, y\_train)

print(confusion\_matrix(y\_test, model.predict(X\_test\_scaled)))

print(classification\_report(y\_test, model.predict(X\_test\_scaled)))

**Story to tell:** “This gives an interpretable baseline (coefficients), but because of class imbalance, precision/recall on ‘yes’ are lower—hence SMOTE + NN/RF.”

**6) data\_preprocessing\_and\_EDA.ipynb — exploration + proof of baseline realities**

**What you usually cover in this notebook (and you should say in the interview):**

* Class imbalance visualization (~88/12).
* Target leakage check (drop duration).
* Correlations/seasonality (month effects, previous campaign outcome).
* Outlier notes (balance heavy-tailed).
* First pass models or sanity checks.

You don’t need to show the whole notebook—just say what you validated and the decisions that flowed into training scripts.

**7) eda\_bank\_marketing.md — stakeholder-friendly EDA summary**

**Use this to talk to non-tech folks:**

* What the dataset is, target definition, and class split.
* Key signals: poutcome\_success, some month effects, loan/housing patterns.
* Decision: “We will drop duration and address imbalance.”

This doc = your narrative bridge from “raw data” to “why our model will help marketing.”

**8) cleaning\_summary.md — what you cleaned and why it matters**

**Pull these bullets in interviews:**

* Clarified that unknown and special values (e.g., pdays = -1) are not missing but carry meaning.
* Standardized categories (e.g., job labels).
* Dropped duration everywhere to avoid leakage.
* Documented impacts (row counts, feature changes) so the process is reproducible and auditable.

This file shows you’re disciplined—not just coding, but **governance-ready**.

**Mini-FAQ (rapid-fire answers you can use)**

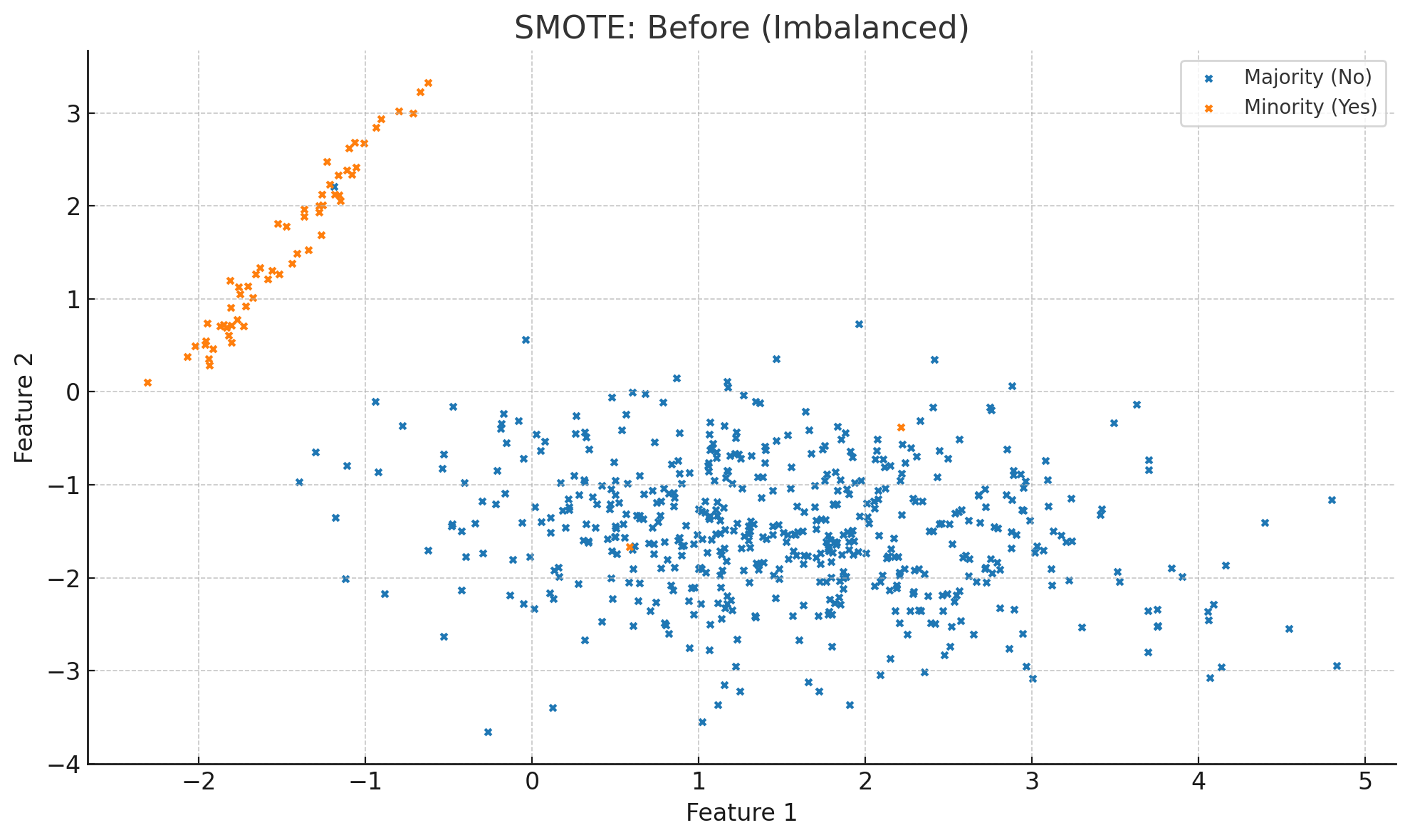
* **What’s overfitting? Why does it happen?**  
  Memorizing training noise instead of learning general patterns—too many parameters, too long training, or noisy data. Fixed with **Dropout**, **EarlyStopping**, **regularization**, and **validation monitoring**.
* **Why joblib/pickle?**  
  To **persist** models and preprocessors exactly as trained. Then prod can load() and transform/predict identically—no re-fitting, no surprises.
* **Why SHAP? Any alternatives?**  
  SHAP gives **local** explanations per prediction and **global** importance. Alternatives: **LIME**, **Permutation Importance**, tree **feature\_importances\_**. I like SHAP for business trust and regulator-friendly narratives.
* **Why SMOTE?**  
  Because with 88/12 skew, a model can game accuracy by predicting “no” for everyone. SMOTE gives the NN enough **positive** examples to learn meaningful decision boundaries. Always apply it **after** splitting (otherwise test leakage).
* **Why One-Hot for NN, LabelEncoder for RF/DT?**  
  NN requires numeric inputs without fake order; One-Hot is correct. Trees are less sensitive but still, One-Hot is cleaner. Your NN path is the production-ready preprocessing; RF/DT scripts are quick baselines.

If you want, I can turn this into a **one-pager cheat sheet** (PDF) with diagrams:

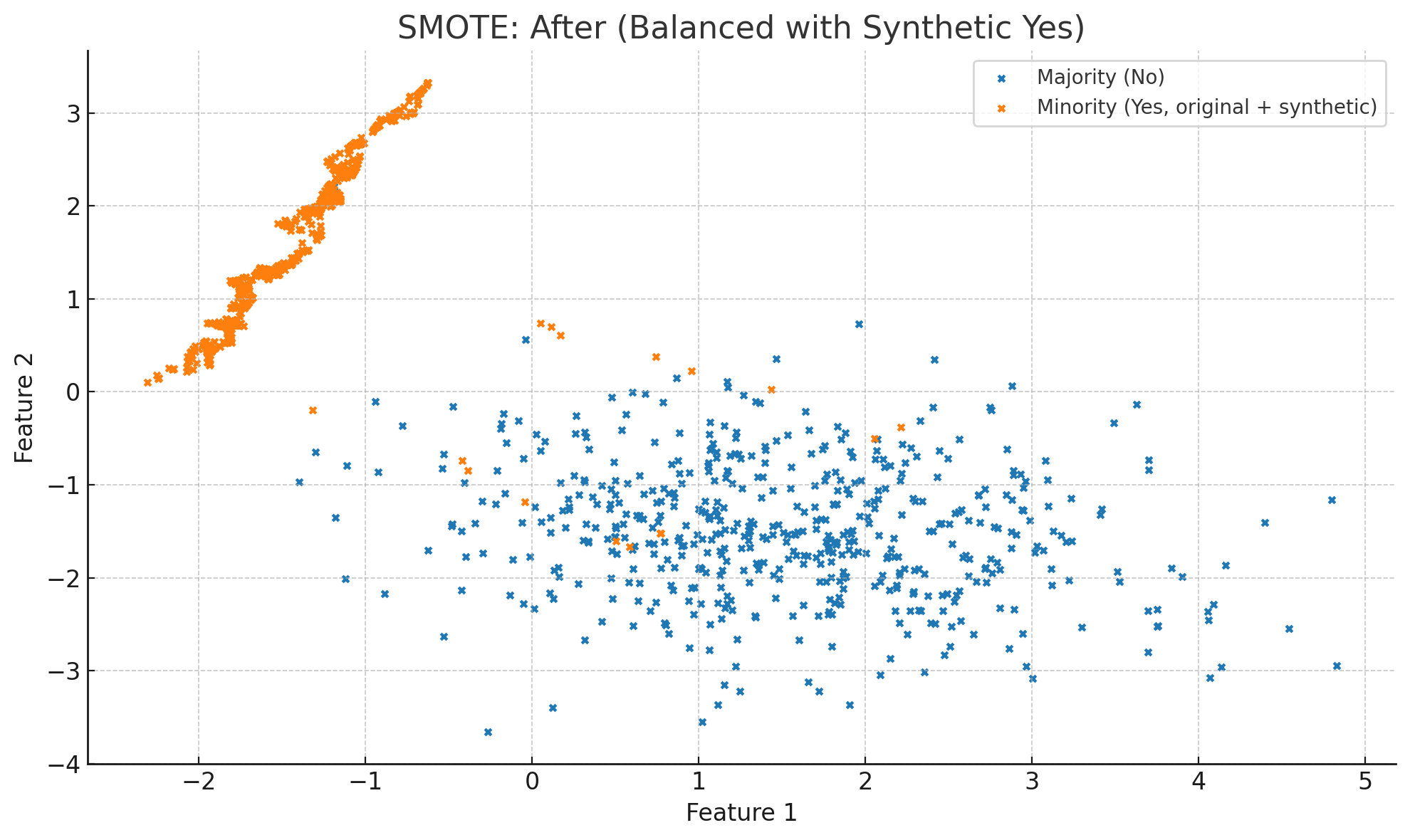
* SMOTE before/after,
* One-hot example,
* NN architecture sketch,
* AUC vs accuracy mini-chart,
* SHAP bar example.

Thought for 15s

SMOTE: Before (Imbalanced)



SMOTE: After (Balanced with Synthetic Yes)



One-Hot Encoding - Raw

Now interactive!

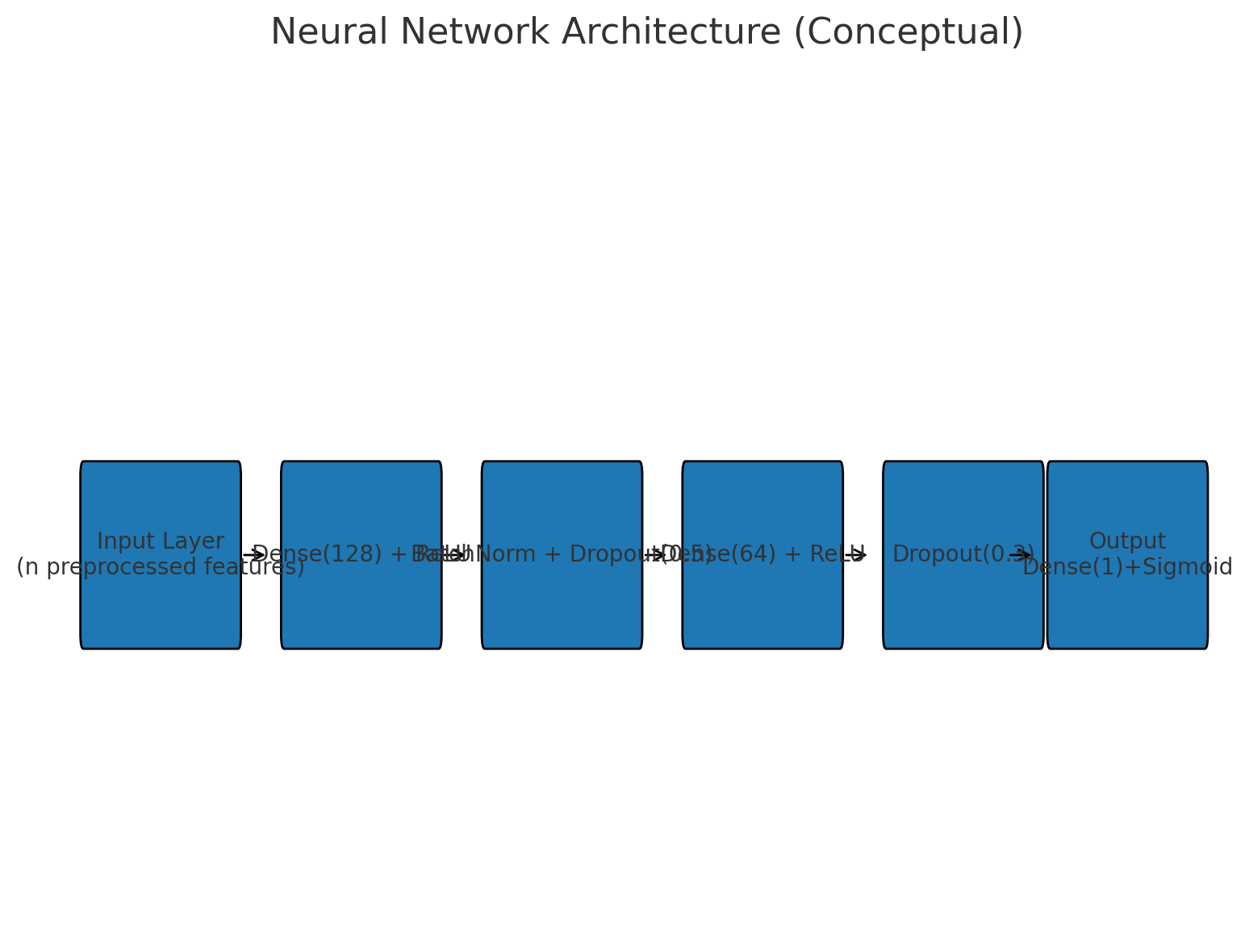
| **age** | **marital** | **housing** |
| --- | --- | --- |
| 28 | single | yes |
| 45 | married | no |
| 39 | divorced | yes |
| 52 | married | no |

One-Hot Encoding - Encoded

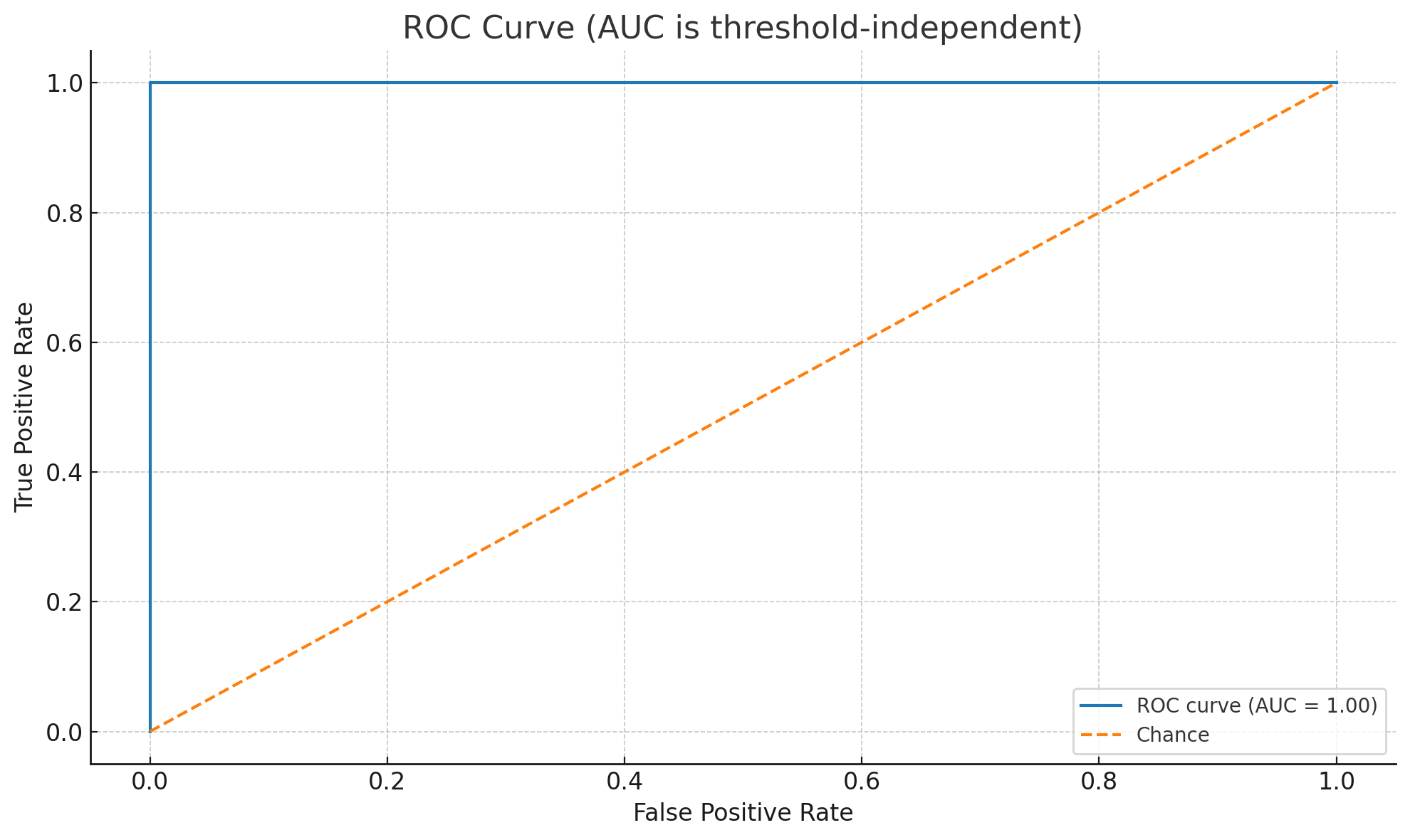
Now interactive!

| **age** | **marital\_divorced** | **marital\_married** | **marital\_single** |
| --- | --- | --- | --- |
| 28 | 0 | 0 | 1 |
| 45 | 0 | 1 | 0 |
| 39 | 1 | 0 | 0 |
| 52 | 0 | 1 | 0 |

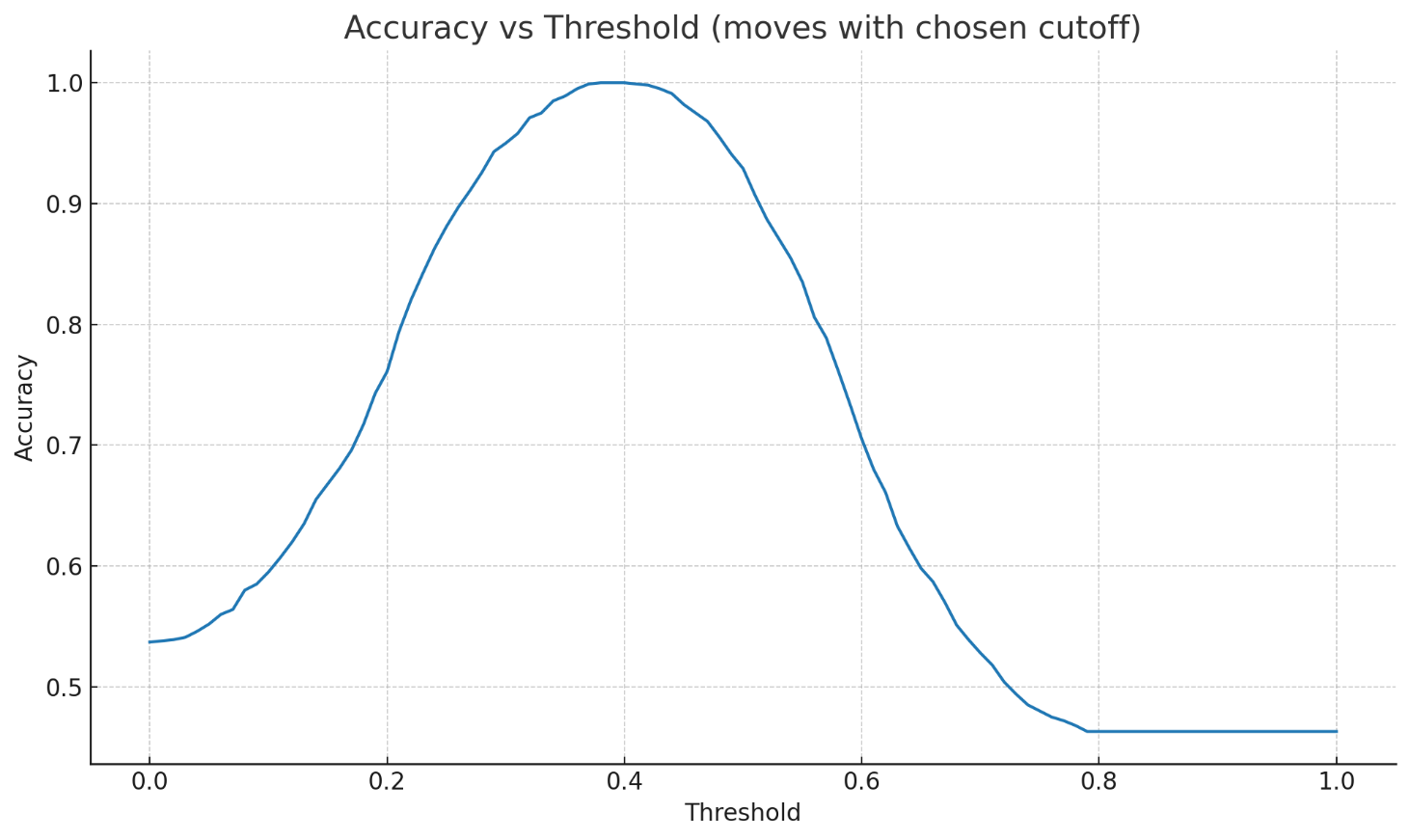
Neural Network Architecture (Conceptual)



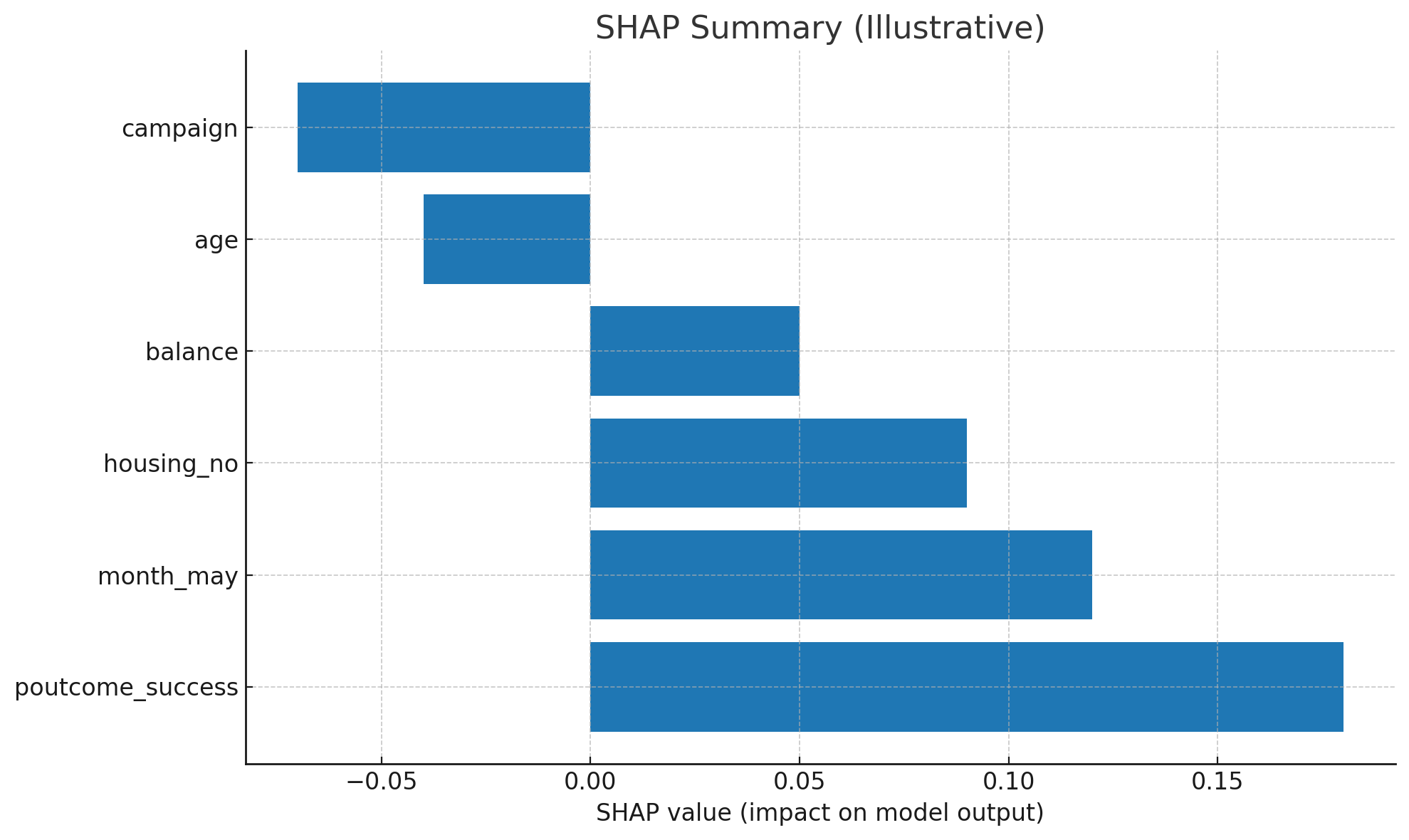
ROC Curve (AUC is threshold-independent)



Accuracy vs Threshold (moves with chosen cutoff)



SHAP Summary (Illustrative)



Perfect—here are **clear explanations** plus the **visuals** I just generated for you. If anything looks off or you want them exported to a PDF/PowerPoint, say the word and I’ll package them up.

**1) SMOTE — Before/After (what, why, how)**

**What you’re seeing:**

* **Before:** blue points (majority “No”) dominate; only a few orange points (minority “Yes”).
* **After:** SMOTE creates **synthetic “Yes”** points by interpolating between real “Yes” neighbours → the classes look more balanced.

**Why it matters:**

* With 88% “No” vs 12% “Yes”, a lazy model can predict “No” for everyone and still brag high accuracy.
* SMOTE **only on the training set** gives the model enough positive examples to learn patterns without contaminating the test set (that would be **test leakage**).

**How to say it in the interview (simple):**

“We used SMOTE after splitting the data. It synthesizes realistic ‘Yes’ customers by interpolating between actual ‘Yes’ neighbours, so the neural net learns meaningful boundaries instead of defaulting to ‘No’.”

You can scroll up to see the **two scatter plots** titled:

* *SMOTE: Before (Imbalanced)*
* *SMOTE: After (Balanced with Synthetic Yes)*

**2) One-Hot Encoding — tiny, concrete example**

**Why we need it:**  
Categorical values like marital = {'single','married','divorced'} don’t have a natural order. If we encode them as 0/1/2, the model might think “divorced > married > single.” That’s nonsense.  
**One-hot** creates a **binary column per category**, avoiding fake ordering.

I showed you two tables:

* **“One-Hot Encoding – Raw”** (small sample)
* **“One-Hot Encoding – Encoded”** (the one-hot columns added)

**Interview line:**

“Neural nets want numbers. One-hot encoding turns each category into its own 0/1 flag, so the model doesn’t assume an order that doesn’t exist.”

**3) Neural Network Architecture — the sketch**

I drew a simple **box-and-arrow** diagram with:

* **Input Layer**: “n preprocessed features” (after scaling + one-hot).
* **Dense(128) + ReLU**: Learns non-linear patterns; ReLU speeds training and avoids vanishing gradients.
* **BatchNorm + Dropout(0.5)**: BatchNorm stabilizes training; Dropout randomly “drops” neurons so the model doesn’t memorize noise.
* **Dense(64) + ReLU** then **Dropout(0.3)**: More representation power, still regularized.
* **Output Dense(1)+Sigmoid**: Probability of “Yes” between 0 and 1.

**Overfitting in plain English:**  
Like cramming answers by heart — you ace the training questions but stumble on new ones. **Dropout** + **EarlyStopping** are your “don’t cram” policies.

**Interview line:**

“Our NN takes the preprocessed feature vector, passes it through ReLU layers for non-linear patterns, uses BatchNorm and Dropout to reduce overfitting, and outputs a single Sigmoid probability for the term-deposit ‘Yes’.”

**4) AUC vs Accuracy — why AUC is steadier**

I generated **two charts**:

* **ROC Curve (AUC)**: Measures ranking quality *across all thresholds*. It doesn’t care where you set the cutoff.
* **Accuracy vs Threshold**: Accuracy bounces around as you slide the threshold (0.3 vs 0.5 vs 0.7 can look very different), especially with imbalance.

**Interview line:**

“We report AUC because it’s threshold-independent, so it’s a stable measure of how well we rank ‘Yes’ above ‘No’. Accuracy can mislead on imbalanced data and shifts wildly if you change the cutoff.”

**5) SHAP Bar — what it means**

I drew a **horizontal bar** (illustrative) with features like poutcome\_success, month\_may, housing\_no, etc.

* **Positive bars** → push the prediction **toward ‘Yes’**.
* **Negative bars** → push **toward ‘No’**.

**Why SHAP:**

* **Local explanations** (per client) for trust: “This person’s previous success and month contribute +0.30 to their score.”
* **Global view** (summary bars) for strategy: “These features matter most across the portfolio.”

**Alternatives you can name:** LIME, permutation importance, tree feature importances.  
**When I’d pick SHAP:** Regulated/banking context; you need both **per-prediction** and **overall** explanations.