The Classification Engineer Handbook

Subscription Retention & Churn Prediction

Project: Customer Behavior Analysis for Subscription Retention
Dataset: Alibaba User Behavior (RL)

Your Core Objectives

1. **The Math:** Build Logistic Regression from scratch (Gradient Descent & Sigmoid).
2. **The Logic:** Build Decision Trees from scratch (Entropy & Information Gain).
3. **The Better Model:** Implement a Random Forest manually (Bagging).
4. **The Target:** Predict `is_churn` (Terminal state).

Status: CONFIDENTIAL TEAM DOCUMENT
Date: February 2026

1. The Mission: Predicting Retention

While the Regression Engineer predicts revenue, you predict **Survival**.

In our Reinforcement Learning context, your model acts as the **"Termination Function."** The system asks your model: *"If I show Page X to User Y, will they get bored and quit?"*

1.1 Your Phases of Work

- **Phase 1 (Now – Feb 23):** Build **Logistic Regression** from scratch. This establishes our baseline.
- **Phase 2 (Feb 24 – Mar 2):** Build **Decision Trees** and the **Random Forest** from scratch. This covers the "Better Model" requirement.
- **Phase 3 (Mar 3 – Mar 9):** Quantitative Analysis (Confusion Matrix, F1-Score).

2. Phase 1: Logistic Regression from Scratch

You must strictly follow the `BaseModel` contract provided by the Data Engineer.

2.1 The Math (Sigmoid & Log Loss)

Logistic Regression is Linear Regression passed through an activation function.

1. The Sigmoid Activation:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

where $z = wx + b$.

2. The Cost Function (Log Loss / Binary Cross Entropy):

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

3. The Update Rule (Gradient Descent):

$$w := w - \alpha \cdot \frac{1}{m}X^T(\sigma(Xw+b)-y)$$

2.2 Implementation Guide

Create file: `src/classification/logistic_regression.py`.

```python
import numpy as np
from src.models_scratch.base import BaseModel

class LogisticRegressionScratch(BaseModel):
    def __init__(self, learning_rate=0.01, iterations=1000):
        super().__init__()
        self.lr = learning_rate
        self.iterations = iterations
        self.weights = None
        self.bias = 0

    def _sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)

        for _ in range(self.iterations):
            # Linear Model
            linear_model = np.dot(X, self.weights) + self.bias
            # Activation
            y_predicted = self._sigmoid(linear_model)

            # Gradients
            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            # Update
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self._sigmoid(linear_model)
        # Convert probability to class (Threshold 0.5)
        return [1 if i > 0.5 else 0 for i in y_predicted]
```

3. Phase 2: The ”Better Model” (Tree Ensembles)

To secure the highest grade, you cannot just use a simple linear model. You must implement a **Decision Tree** manually, and then combine many of them into a **Random Forest**.

3.1 Step 1: The Decision Tree Logic (Entropy)

You need to write a class that recursively splits data.

Entropy Formula:

$$E = -\sum p_i \log_2(p_i)$$

Information Gain:

$$IG = E(parent) - \sum w_i E(child_i)$$

3.2 Step 2: Implementation Strategy

Create `src/classification/decision_tree.py`.

```
import numpy as np

class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None
    ):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value  # Leaf node value

class DecisionTreeScratch:
    def __init__(self, max_depth=10):
        self.max_depth = max_depth
        self.root = None
```

```python
    def fit(self, X, y):
        self.root = self._grow_tree(X, y)

    def _grow_tree(self, X, y, depth=0):
        n_samples, n_features = X.shape
        n_labels = len(np.unique(y))

        # Stopping criteria
        if depth >= self.max_depth or n_labels == 1:
            leaf_value = self._most_common_label(y)
            return Node(value=leaf_value)

        # Find best split (Implement Information Gain logic here)
        best_feat, best_thresh = self._best_split(X, y, n_features)

        # Recursion
        left_idxs, right_idxs = self._split(X[:, best_feat], best_thresh)
        left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth+1)
        right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth+1)
        return Node(best_feat, best_thresh, left, right)
```

3.3 Step 3: The Random Forest (Bagging)

Create `src/classification/random_forest.py`. This class assumes you have the DecisionTree class working.

```python
from src.classification.decision_tree import DecisionTreeScratch

class RandomForestScratch:
    def __init__(self, n_trees=10, max_depth=10):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        for _ in range(self.n_trees):
            # Bootstrap Sampling (Random selection with replacement)
            idxs = np.random.choice(len(y), size=len(y), replace=True)
            X_sample, y_sample = X[idxs], y[idxs]

            tree = DecisionTreeScratch(max_depth=self.max_depth)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)

    def predict(self, X):
        # Gather predictions from all trees
        tree_preds = np.array([tree.predict(X) for tree in self.trees])
        # Majority Vote
        final_preds = []
        for i in range(tree_preds.shape[1]):
            counts = np.bincount(tree_preds[:, i])
            final_preds.append(np.argmax(counts))
        return np.array(final_preds)
```

4. Phase 3: Quantitative Analysis

The professor requires specific metrics. Since you are building from scratch, you should ideally calculate these from scratch too (or at least understand how).

4.1 Required Metrics

- **Confusion Matrix:** Calculate TP (True Pos), TN, FP, FN.
- **Accuracy:** $(TP + TN)/Total$
- **Precision:** $TP/(TP + FP)$
- **Recall (Sensitivity):** $TP/(TP + FN)$
- **F1-Score:** $2 * (Precision * Recall)/(Precision + Recall)$

4.2 Running Your Code

The Data Engineer has created a script `src/classification/churn_predictor.py`. You need to edit it to import YOUR new classes.

```python
# In churn_predictor.py
from src.classification.logistic_regression import LogisticRegressionScratch
from src.classification.random_forest import RandomForestScratch

# Train
model = RandomForestScratch(n_trees=20)
model.fit(X_train, y_train)
```

Deliverable Checklist:

- ☐ `logistic_regression.py` (Sigmoid logic).
- ☐ `decision_tree.py` (Recursive logic).
- ☐ `random_forest.py` (Bagging logic).
- ☐ Benchmarking Table (Your Model vs Sklearn).
- ☐ Trained `.pkl` files in the MinIO `models` bucket.