The Regression Engineer Handbook

Numerical Forecasting & Value Estimation

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

Your Core Objectives

1. **The Math:** Build Linear Regression from scratch (using NumPy Gradient Descent).
2. **The Target:** Predict `page_value` (How much Yuan a user spends).
3. **The Bonus:** Build a "Better Model" (Gradient Boosting) from scratch.

Status: CONFIDENTIAL TEAM DOCUMENT
Date: February 2026

1. The Mission: Predicting Value

While the Classifier predicts *if* a user stays, you predict *what they are worth*.

We are building a Reinforcement Learning (RL) system. Your model acts as the **"Reward Function."** The system asks your model: *"If I show Page X to User Y, how much money will we make?"*

1.1 Your Phases of Work

- **Phase 1 (Now – Feb 23):** Build the Standard Models (Linear/Polynomial) using pure NumPy. No `sklearn.linear_model`!
- **Phase 2 (Feb 24 – Mar 2):** Build the "Better Model" (Gradient Boosting from Scratch).
- **Phase 3 (Mar 3 – Mar 9):** Benchmarking and Report Writing.

2. Phase 1: The "From Scratch" Linear Regression

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

2.1 Step 1: Locate Your Workspace

Your code lives here: `src/regression/linear_regression.py`.

2.2 Step 2: The Math (Gradient Descent)

You cannot call `model.fit()`. You must implement the math.

The Hypothesis:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ...$$

The Cost Function (MSE):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

The Update Rule (Gradient Descent):

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

2.3 Step 3: Implementation Guide

Write this code in your file. This is how you implement it efficiently using Vectorization (NumPy).

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

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

```
    def fit(self, X, y):
        # 1. Initialize parameters
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        # 2. Gradient Descent Loop
        for i in range(self.iterations):
            # Prediction (y_pred = wx + b)
            y_predicted = np.dot(X, self.weights) + self.bias

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

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

            # (Optional) Log cost every 100 steps
            cost = (1 / (2*n_samples)) * np.sum((y_predicted - y)**2)
            if i % 100 == 0:
                print(f"Epoch {i}, Cost: {cost}")

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias
```

3. Phase 2: Gradient Boosting from Scratch

To get the full 40 points, you must implement a "Better Model" manually. Linear Regression is too simple for Alibaba data. We will build a **Gradient Boosting Regressor (GBM)**.

3.1 The Concept

Gradient Boosting combines many "Weak Learners" (simple Decision Trees) to make a strong one.

1. Train a Decision Tree to predict y.
2. Calculate the errors (Residuals $= y - y_{pred}$).
3. Train a **second** Tree to predict the *Residuals*.
4. Repeat 100 times.
5. Final Prediction = Sum of all tree predictions.

3.2 Implementation Strategy

Create `src/regression/gbm_scratch.py`.

```
import numpy as np
# You are allowed to use sklearn's Tree as the "Weak Learner"
# ONLY IF you implement the Boosting logic yourself.
from sklearn.tree import DecisionTreeRegressor

class GradientBoostingScratch:
```

```
    def __init__(self, n_estimators=100, learning_rate=0.1):
        self.n_estimators = n_estimators
        self.lr = learning_rate
        self.trees = []
        self.base_pred = None

    def fit(self, X, y):
        # 1. Initial Prediction (Mean)
        self.base_pred = np.mean(y)
        y_pred = np.full(y.shape, self.base_pred)

        for _ in range(self.n_estimators):
            # 2. Calculate Residuals (The Error)
            residuals = y - y_pred

            # 3. Train a weak tree on the residuals
            tree = DecisionTreeRegressor(max_depth=2)
            tree.fit(X, residuals)
            self.trees.append(tree)

            # 4. Update prediction
            update = tree.predict(X)
            y_pred += self.lr * update

    def predict(self, X):
        # Start with the base mean
        y_pred = np.full(X.shape[0], self.base_pred)

        # Add contributions from all trees
        for tree in self.trees:
            y_pred += self.lr * tree.predict(X)

        return y_pred
```

Why this passes the requirement: You are writing the *Boosting Algorithm* yourself. You are managing the residuals and the additive training loop. This counts as "From Scratch."

4. Phase 3: Training and Delivery

The Data Engineer has set up the pipeline. Your job is to run your scripts and save the models to MinIO.

4.1 How to Run Your Code

1. Open `src/regression/value_predictor.py` (The training script). 2. Import your class:

```
    from src.regression.linear_regression import LinearRegressionScratch
    # OR
    from src.regression.gbm_scratch import GradientBoostingScratch
```

3. Run the script in your terminal:

```
    python3 src/regression/value_predictor.py
```

4.2 Metrics to Report

For the final report, you must calculate:

- **MSE (Mean Squared Error):** How far off are our price predictions?
- **R-Squared:** How much of the variance does our model explain?
- **Benchmarking:** Compare your `LinearRegressionScratch` score vs `sklearn.linear_model.LinearReg` They should be nearly identical.

Deliverable Checklist:

☐ `linear_regression.py` (NumPy logic).

☐ `gbm_scratch.py` (Boosting logic).

☐ Benchmarking Table (Scratch vs Sklearn).

☐ Trained `.pkl` files in the MinIO `models` bucket.