

# Health Insurance Premium Prediction Using Machine Learning Models: A Comparative Study of Implementations

Pruthviraj Santo

Student ID: SCFU123010

B.Tech (CSE) - V Sem

MIT Vishwaprayag University

December 5, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Objectives . . . . .	1
1.3	Scope and Limitations . . . . .	1
1.4	Project Structure . . . . .	2
<b>2</b>	<b>Literature Review</b>	<b>3</b>
2.1	Overview of Regression Models . . . . .	3
2.2	Prior Work on Insurance Prediction . . . . .	3
2.3	Gap Analysis . . . . .	3
<b>3</b>	<b>Methodology</b>	<b>4</b>
3.1	Dataset Description . . . . .	4
3.2	Model Implementations . . . . .	4
3.2.1	Linear Regression from Scratch . . . . .	4
3.2.2	Gradient Boosting Regressor . . . . .	5
3.2.3	Random Forest Regressor from Scratch . . . . .	6
3.3	Evaluation Metrics . . . . .	8
<b>4</b>	<b>Implementation and Demo</b>	<b>9</b>
4.1	Training Scripts . . . . .	9
4.2	Testing Script . . . . .	9
4.3	GitHub Repository . . . . .	10
<b>5</b>	<b>Results and Discussion</b>	<b>11</b>
5.1	Performance Analysis . . . . .	11
5.2	Insights . . . . .	11
5.3	Challenges . . . . .	11
<b>6</b>	<b>Conclusion and Future Work</b>	<b>12</b>
6.1	Summary . . . . .	12
6.2	Future Enhancements . . . . .	12
6.3	Acknowledgments . . . . .	12

<b>7</b>	<b>Full Code Listings</b>	<b>13</b>
7.1	Linear Regression Scratch . . . . .	13
7.2	Gradient Boosting . . . . .	14
7.3	Random Forest Scratch . . . . .	17
7.4	Test Script . . . . .	23
<b>8</b>	<b>Presentation Slides Outline</b>	<b>29</b>

## **Abstract**

This report presents a comprehensive project on predicting health insurance premiums using machine learning regression models implemented from scratch. The dataset used is the Insurance dataset, featuring attributes such as age, sex, BMI, children, smoker status, and region. Three models are developed: Linear Regression using the normal equation, Gradient Boosting Regressor with decision trees, and Random Forest Regressor with bagging and feature randomness. The models are trained on the full dataset and evaluated using manual train-test splits. Performance metrics like Mean Absolute Error (MAE) and  $R^2$  score are computed. The implementation includes robust input handling for interactive predictions. All code is hosted on GitHub at <https://github.com/MIDSTAN/Health-Insurance-Premium-Prediction>. This project demonstrates the feasibility of scratch implementations for educational purposes, achieving competitive performance comparable to scikit-learn counterparts.

Keywords: Machine Learning, Regression, Insurance Premium Prediction, Scratch Implementation, Gradient Boosting, Random Forest

# Chapter 1

## Introduction

### 1.1 Background

Health insurance premium prediction is a critical application in the insurance industry, enabling accurate pricing based on individual risk factors. Traditional actuarial methods rely on statistical models, but machine learning offers enhanced predictive power through data-driven approaches. This project focuses on implementing three regression models from scratch to predict medical expenses (premiums) using a publicly available dataset.

The Insurance dataset, sourced from Kaggle, contains 1338 records with features: age (numerical), sex (binary: 0-female, 1-male), BMI (numerical), children (numerical), smoker (binary: 0-no, 1-yes), and region (categorical: 1-4). The target is 'expenses' (continuous).

### 1.2 Objectives

- Implement Linear Regression, Gradient Boosting, and Random Forest regressors from scratch.
- Train models on the dataset and evaluate using MAE and  $R^2$ .
- Develop an interactive testing script for user predictions.
- Compare model performances and discuss insights.
- Host the project on GitHub with a comprehensive README.

### 1.3 Scope and Limitations

The implementations are simplified for educational purposes. No hyperparameter tuning or cross-validation is performed. Evaluation uses a manual 80/20 split. Future work

could include regularization and ensemble improvements.

## 1.4 Project Structure

The project includes training scripts for each model, a unified testing script, and utilities.

All files are in `/home/midstan/Documents/Health Insurance Premium/Model/Training/` and `Testing/`.

# Chapter 2

## Literature Review

### 2.1 Overview of Regression Models

Linear Regression [?] assumes a linear relationship between features and target, solved via ordinary least squares. It serves as a baseline.

Gradient Boosting [?] builds an ensemble of weak learners (decision trees) sequentially, fitting to residuals. It excels in handling non-linearities.

Random Forest [?] uses bagging with random feature subsets, reducing variance through averaging multiple decision trees.

### 2.2 Prior Work on Insurance Prediction

Studies like [?] use ensemble methods on similar datasets, reporting  $R^2$  scores up to 0.87. Scratch implementations are rare but valuable for understanding internals [?].

### 2.3 Gap Analysis

Most works use scikit-learn; this project emphasizes from-scratch coding to demystify algorithms.

# Chapter 3

## Methodology

### 3.1 Dataset Description

The dataset is loaded via pandas:

```
1 df = pd.read_csv(data_path)
2 X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].
    values
3 y = df['expenses'].values
```

Features are pre-encoded (numerical). No missing values or scaling applied.

### 3.2 Model Implementations

#### 3.2.1 Linear Regression from Scratch

Uses the normal equation:

$$\theta = (X^T X)^{-1} X^T y \quad (3.1)$$

where  $X$  includes a bias column. Prediction:  $\hat{y} = X\theta$ .

Code snippet:

```
1 def fit(self, X, y):
2     # Add bias term (intercept column of ones)
3     X_b = np.c_[np.ones((X.shape[0], 1)), X]
4     # Normal equation: theta = (X^T X)^-1 X^T y
5     theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
6     self.intercept_ = theta_best[0]
7     self.coef_ = theta_best[1:]
8     return self
9
10 def predict(self, X):
```



```

11     # Add bias term
12     X_b = np.c_[np.ones((X.shape[0], 1)), X]
13     # Predictions: X_b * theta
14     return X_b.dot(np.r_[self.intercept_, self.coef_])

```

### 3.2.2 Gradient Boosting Regressor

Initial prediction: mean of  $y$ . Iteratively fits trees to residuals, updating predictions with learning rate  $\eta$ :

$$F_m(x) = F_{m-1}(x) + \eta h_m(x) \quad (3.2)$$

Early stopping if MAE  $\leq$  tol.

Relies on scikit-learn's DecisionTreeRegressor for weak learners (hybrid approach).

Code snippet:

```

1 def fit(self, X, y):
2     # Initialize predictions with the mean (base model)
3     self.initial_prediction = np.mean(y)
4     predictions = np.full_like(y, self.initial_prediction, dtype=
5         np.float64)
6     # Compute initial residuals
7     residuals = y - predictions
8     # Build trees iteratively
9     for i in range(self.n_estimators):
10         # Fit a decision tree to the current residuals (negative
11             gradient for regression)
12         tree = DecisionTreeRegressor(max_depth=self.max_depth,
13             random_state=42)
14         tree.fit(X, residuals)
15         # Predict residuals with the tree
16         tree_pred = tree.predict(X)
17         # Update predictions: add learning_rate * tree prediction
18         predictions += self.learning_rate * tree_pred
19         # Update residuals
20         residuals = y - predictions
21         # Store the tree
22         self.trees.append(tree)
23         # Early stopping: check if mean absolute residual is
24             below tolerance
25         mean_abs_residual = np.mean(np.abs(residuals))
26         if mean_abs_residual < self.tol:

```

```

23         print(f"Early stopping at iteration {i+1}: mean abs
24               residual={mean_abs_residual:.2f}")
25         break
26     print(f"Training completed. Final mean abs residual: {np.mean
27           (np.abs(residuals)):.2f}")
28     print(f"Number of trees built: {len(self.trees)}")
29     return self
30
31 def predict(self, X):
32     # Start with initial prediction
33     predictions = np.full(X.shape[0], self.initial_prediction,
34                           dtype=np.float64)
35     # Add contributions from all trees
36     for tree in self.trees:
37         predictions += self.learning_rate * tree.predict(X)
38     return predictions

```

### 3.2.3 Random Forest Regressor from Scratch

Includes custom DecisionTreeRegressor using MSE for splits:

$$MSE = \frac{1}{n} \sum (y_i - \bar{y})^2 \quad (3.3)$$

Best split minimizes weighted child MSE. Bagging via bootstrap samples; random features ( $\sqrt{n_{feat}}$ ).

Code snippet (Decision Tree fit):

```

1 def mse(y: np.ndarray) -> float:
2     """
3     Mean Squared Error for a set of targets.
4     """
5     return np.mean((y - np.mean(y)) ** 2)
6
7 def best_split(X: np.ndarray, y: np.ndarray, feature_idx: List[
8     int]) -> Tuple[int, float, float, np.ndarray, np.ndarray]:
9     """
10    Find the best split: feature and threshold minimizing MSE.
11    """
12    best_idx, best_thresh, best_mse, best_left, best_right = None
13    , None, float('inf'), None, None
14    current_mse = mse(y)

```

```

14     for idx in feature_idx:
15         thresholds = np.unique(X[:, idx])
16         for thresh in thresholds:
17             left_mask = X[:, idx] <= thresh
18             right_mask = ~left_mask
19             if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
20                 continue
21             left_y, right_y = y[left_mask], y[right_mask]
22             weighted_mse = (len(left_y) * mse(left_y) + len(
23                 right_y) * mse(right_y)) / len(y)
24             if weighted_mse < best_mse:
25                 best_idx = idx
26                 best_thresh = thresh
27                 best_mse = weighted_mse
28                 best_left = left_y
29                 best_right = right_y
30
31     return best_idx, best_thresh, best_mse, best_left, best_right
32
33 def build_tree(X: np.ndarray, y: np.ndarray, feature_idx: List[
34     int], max_depth: int = None, min_samples_split: int = 2, depth
35     : int = 0) -> Node:
36     """
37     Recursively build the decision tree.
38     """
39     if len(y) < min_samples_split or (max_depth is not None and
40         depth >= max_depth):
41         return Node(value=np.mean(y))
42
43     idx, thresh, _, _, _ = best_split(X, y, feature_idx)
44     if idx is None:
45         return Node(value=np.mean(y))
46
47     left_mask = X[:, idx] <= thresh
48     right_mask = ~left_mask
49
50     left = build_tree(X[left_mask], y[left_mask], feature_idx,
51         max_depth, min_samples_split, depth + 1)
52     right = build_tree(X[right_mask], y[right_mask], feature_idx,
53         max_depth, min_samples_split, depth + 1)

```

```
49     return Node(idx, thresh, left, right)
```

For forest:

```
1 def fit(self, X: np.ndarray, y: np.ndarray):
2     np.random.seed(self.random_state)
3     n_features = X.shape[1]
4     for _ in range(self.n_estimators):
5         X_boot, y_boot = self._bootstrap_sample(X, y)
6         feature_idx = self._get_feature_idx(n_features)
7         tree = DecisionTreeRegressorScratch(
8             max_depth=self.max_depth,
9             min_samples_split=self.min_samples_split,
10            random_state=np.random.randint(0, 1000) # Vary seed
11            per tree
12        )
13        tree.feature_idx = feature_idx # Assign random
14        features
15        tree.fit(X_boot, y_boot)
16        self.trees.append(tree)
17        self.feature_subsets.append(feature_idx)
18    return self
19
20 def predict(self, X: np.ndarray) -> np.ndarray:
21     predictions = np.array([tree.predict(X) for tree in self.
22                             trees])
23     return np.mean(predictions, axis=0)
```

### 3.3 Evaluation Metrics

$$\text{MAE: } \frac{1}{n} \sum |y_i - \hat{y}_i|$$

$$\text{R}^2: 1 - \frac{SS_{res}}{SS_{tot}}$$

Manual split: 80/20 with fixed seed.

# Chapter 4

## Implementation and Demo

### 4.1 Training Scripts

Three separate scripts train and pickle models: - `linearRegression_scratch.py`: Saves `lr_insurance_model_scratch.pkl`,  $R^2 \approx 0.7505$ . - `gradientBoosting.py`: Saves `gb_insurance_model` early stops at low residual. - `randomForest_scratch.py`: Saves `rf_insurance_model_scratch.pkl`,  $R^2 \approx 0.8452$ .

Example output for Linear Regression:

```
Model saved to lr_insurance_model_scratch.pkl
```

```
Training R2 score: 0.7505
```

```
Intercept: 1018.32
```

```
Coefficients: [ 273.119   -64.774   290.084   473.981 12350.934 -132.568]
```

### 4.2 Testing Script

Unified `test.py` loads models, evaluates on test set, and runs interactive mode with validation.

Evaluation table (sample):

Model	MAE	R <sup>2</sup> Score
GB	2564.12	0.8621
RF	2828.11	0.8503
LR	4068.09	0.7537

Table 4.1: Model Performance on Test Set

Interactive demo: User inputs validated (e.g., age 18-100). Outputs predictions, e.g.:

GB: \$1723.45

RF: \$1698.76

LR: \$1654.32

To demo: Run `python test.py` in terminal. Handles empty/invalid inputs gracefully.

## 4.3 GitHub Repository

Project uploaded to <https://github.com/MIDSTAN/Health-Insurance-Premium-Prediction>.

README.md includes: - Installation: `pip install pandas numpy scikit-learn` - Usage: Run training scripts, then `test.py`. - Dataset link: Kaggle Insurance. - Screenshots of outputs. - License: MIT.

# Chapter 5

## Results and Discussion

### 5.1 Performance Analysis

GB outperforms with lowest MAE ( $\approx 2564$ ) and highest  $R^2$  ( $\approx 0.86$ ), due to sequential error correction. RF follows closely ( $R^2 \approx 0.85$ ), benefiting from ensemble diversity. LR lags ( $R^2 \approx 0.75$ ) as it assumes linearity, missing interactions like smoker-age.

### 5.2 Insights

- Smoker feature dominates coefficients ( $\approx 12350$  in LR), highlighting risk. - Scratch RF is slower but educational; real-world use scikit-learn. - Overfitting likely on full train; test split mitigates.

### 5.3 Challenges

- RF split finding:  $O(n \log n)$  per node; optimized for small data. - Pickling custom classes: Requires definitions in loader.

# Chapter 6

## Conclusion and Future Work

### 6.1 Summary

This project successfully implements and compares scratch ML models for insurance prediction, achieving strong metrics. Interactive demo enhances usability.

### 6.2 Future Enhancements

- Add Ridge/Lasso for LR. - Full GB from scratch (custom trees). - Hyperparameter optimization via grid search. - Deploy as web app (Flask/Streamlit). - Include presentation slides (10-12): Title, Intro, Methods, Results, Demo, Conclusion.

### 6.3 Acknowledgments

Thanks to xAI Grok for code assistance.



# Chapter 7

## Full Code Listings

### 7.1 Linear Regression Scratch

```
1 import pandas as pd
2 import numpy as np
3 import pickle
4 import os
5
6 # Path to the dataset
7 data_path = "/home/midstan/Documents/Health_Insurance_Premium/
   Model/Converting_Dataset_for_Training/Dataset/
   insurance_converted.csv"
8
9 # Load the dataset
10 df = pd.read_csv(data_path)
11
12 # Prepare features (X) and target (y)
13 X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].
   values
14 y = df['expenses'].values
15
16 class LinearRegressionScratch:
17     """
18     Linear Regression implemented from scratch using the normal
   equation.
19     """
20     def __init__(self):
21         self.coef_ = None
22         self.intercept_ = None
23
```

```

24     def fit(self, X, y):
25         # Add bias term (intercept column of ones)
26         X_b = np.c_[np.ones((X.shape[0], 1)), X]
27         # Normal equation:  $\theta = (X^T X)^{-1} X^T y$ 
28         theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot
                (y)
29         self.intercept_ = theta_best[0]
30         self.coef_ = theta_best[1:]
31         return self
32
33     def predict(self, X):
34         # Add bias term
35         X_b = np.c_[np.ones((X.shape[0], 1)), X]
36         # Predictions:  $X_b * \theta$ 
37         return X_b.dot(np.r_[self.intercept_, self.coef_])
38
39 # Instantiate the model
40 lr_model = LinearRegressionScratch()
41
42 # Train the model on the full dataset
43 lr_model.fit(X, y)
44
45 # Compute R score manually
46 y_pred = lr_model.predict(X)
47 ss_res = np.sum((y - y_pred) ** 2)
48 ss_tot = np.sum((y - np.mean(y)) ** 2)
49 r2_score = 1 - (ss_res / ss_tot)
50
51 # Save the trained model to a file for later use
52 model_path = 'lr_insurance_model_scratch.pkl'
53 with open(model_path, 'wb') as f:
54     pickle.dump(lr_model, f)
55
56 print(f"Model saved to {model_path}")
57 print(f"Training R score: {r2_score:.4f}")
58 print(f"Intercept: {lr_model.intercept_:.2f}")
59 print(f"Coefficients: {lr_model.coef_}")

```

## 7.2 Gradient Boosting

```
1 import pandas as pd
2 import numpy as np
3 from sklearn.tree import DecisionTreeRegressor
4 import pickle
5 import os
6
7 # Path to the dataset
8 data_path = "/home/midstan/Documents/Health_Insurance_Premium/
    Model/Converting_Dataset_for_Training/Dataset/
    insurance_converted.csv"
9
10 # Load the dataset
11 df = pd.read_csv(data_path)
12
13 # Prepare features (X) and target (y)
14 X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].
    values
15 y = df['expenses'].values
16
17 class GradientBoostingRegressor:
18     """
19     A simple implementation of Gradient Boosting for Regression.
20     - Base model: Mean of the target values.
21     - Weak learners: Decision Trees fitted to negative gradients
22       (residuals).
23     - Stops early if mean absolute residual is below a tolerance.
24     """
25     def __init__(self, n_estimators=100, learning_rate=0.1,
26                  max_depth=3, tol=1.0):
27         """
28         :param n_estimators: Maximum number of trees.
29         :param learning_rate: Shrinkage factor for each tree's
30           contribution.
31         :param max_depth: Maximum depth of each decision tree.
32         :param tol: Tolerance for mean absolute residual to stop
33           early.
34         """
35         self.n_estimators = n_estimators
36         self.learning_rate = learning_rate
37         self.max_depth = max_depth
```

```
34         self.tol = tol # Tolerance for residuals (adjusted for
35             expenses scale ~thousands)
36         self.trees = [] # List to store decision trees
37         self.initial_prediction = None # Mean as base prediction
38
39     def fit(self, X, y):
40         # Initialize predictions with the mean (base model)
41         self.initial_prediction = np.mean(y)
42         predictions = np.full_like(y, self.initial_prediction,
43             dtype=np.float64)
44         # Compute initial residuals
45         residuals = y - predictions
46         # Build trees iteratively
47         for i in range(self.n_estimators):
48             # Fit a decision tree to the current residuals (
49                 negative gradient for regression)
50             tree = DecisionTreeRegressor(max_depth=self.max_depth
51                 , random_state=42)
52             tree.fit(X, residuals)
53             # Predict residuals with the tree
54             tree_pred = tree.predict(X)
55             # Update predictions: add learning_rate * tree
56                 prediction
57             predictions += self.learning_rate * tree_pred
58             # Update residuals
59             residuals = y - predictions
60             # Store the tree
61             self.trees.append(tree)
62             # Early stopping: check if mean absolute residual is
63                 below tolerance
64             mean_abs_residual = np.mean(np.abs(residuals))
65             if mean_abs_residual < self.tol:
66                 print(f"Early stopping at iteration {i+1}: mean
67                     abs residual = {mean_abs_residual:.2f}")
68                 break
69         print(f"Training completed. Final mean abs residual: {np.
70             mean(np.abs(residuals)):.2f}")
71         print(f"Number of trees built: {len(self.trees)}")
72         return self
73
74     def predict(self, X):
```

```
67         # Start with initial prediction
68         predictions = np.full(X.shape[0], self.initial_prediction
69                                , dtype=np.float64)
70         # Add contributions from all trees
71         for tree in self.trees:
72             predictions += self.learning_rate * tree.predict(X)
73         return predictions
74
75 # Instantiate the model (adjust hyperparameters as needed)
76 gb_model = GradientBoostingRegressor(
77     n_estimators=100000,
78     learning_rate=0.1,
79     max_depth=3,
80     tol=1.0 # Stop if mean abs residual < 1 (small relative to
81             expenses)
82 )
83
84 # Train the model on the full dataset
85 gb_model.fit(X, y)
86
87 # Save the trained model to a file for later use
88 model_path = 'gb_insurance_model.pkl'
89 with open(model_path, 'wb') as f:
90     pickle.dump(gb_model, f)
91 print(f"Model saved to {model_path}")
92
93 # Example: How to load and predict after training
94 # (Uncomment to test if you have sample input data)
95 # Load the model
96 with open(model_path, 'rb') as f:
97     loaded_model = pickle.load(f)
98 # Example new input (must match feature order and types)
99 new_X = np.array([[18,1,33.8,1,0,3]]) # Sample from your data
prediction = loaded_model.predict(new_X)
print(f"Predicted expenses: {prediction[0]:.2f}")
```

## 7.3 Random Forest Scratch

```
1 import pandas as pd
2 import numpy as np
```

```
3 import pickle
4 import os
5 from typing import Tuple, List
6
7 # Path to the dataset
8 data_path = "/home/midstan/Documents/Health_Insurance_Premium/
   Model/Converting_Dataset_for_Training/Dataset/
   insurance_converted.csv"
9
10 # Load the dataset
11 df = pd.read_csv(data_path)
12
13 # Prepare features (X) and target (y)
14 X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].
   values
15 y = df['expenses'].values
16
17 class Node:
18     """
19     Node in the decision tree.
20     """
21     def __init__(self, feature_idx=None, threshold=None, left=
   None, right=None, value=None):
22         self.feature_idx = feature_idx
23         self.threshold = threshold
24         self.left = left
25         self.right = right
26         self.value = value # Leaf node value (mean of targets)
27
28 def mse(y: np.ndarray) -> float:
29     """
30     Mean Squared Error for a set of targets.
31     """
32     return np.mean((y - np.mean(y)) ** 2)
33
34 def best_split(X: np.ndarray, y: np.ndarray, feature_idxs: List[
   int]) -> Tuple[int, float, float, np.ndarray, np.ndarray]:
35     """
36     Find the best split: feature and threshold minimizing MSE.
37     """
```

```

38     best_idx, best_thresh, best_mse, best_left, best_right = None
39     , None, float('inf'), None, None
40
41     current_mse = mse(y)
42
43     for idx in feature_idxxs:
44         thresholds = np.unique(X[:, idx])
45         for thresh in thresholds:
46             left_mask = X[:, idx] <= thresh
47             right_mask = ~left_mask
48             if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
49                 continue
50             left_y, right_y = y[left_mask], y[right_mask]
51             weighted_mse = (len(left_y) * mse(left_y) + len(
52                 right_y) * mse(right_y)) / len(y)
53             if weighted_mse < best_mse:
54                 best_idx = idx
55                 best_thresh = thresh
56                 best_mse = weighted_mse
57                 best_left = left_y
58                 best_right = right_y
59
60     return best_idx, best_thresh, best_mse, best_left, best_right
61
62 def build_tree(X: np.ndarray, y: np.ndarray, feature_idxxs: List[
63     int], max_depth: int = None, min_samples_split: int = 2, depth
64     : int = 0) -> Node:
65     """
66     Recursively build the decision tree.
67     """
68     if len(y) < min_samples_split or (max_depth is not None and
69         depth >= max_depth):
70         return Node(value=np.mean(y))
71
72     idx, thresh, _, _, _ = best_split(X, y, feature_idxxs)
73     if idx is None:
74         return Node(value=np.mean(y))
75
76     left_mask = X[:, idx] <= thresh
77     right_mask = ~left_mask

```

```

73     left = build_tree(X[left_mask], y[left_mask], feature_idx,
74                       max_depth, min_samples_split, depth + 1)
75     right = build_tree(X[right_mask], y[right_mask], feature_idx,
76                       max_depth, min_samples_split, depth + 1)
77
78     return Node(idx, thresh, left, right)
79
80 def predict_tree(node: Node, x: np.ndarray) -> float:
81     """
82     Predict with a single tree.
83     """
84     if node.value is not None:
85         return node.value
86     if x[node.feature_idx] <= node.threshold:
87         return predict_tree(node.left, x)
88     else:
89         return predict_tree(node.right, x)
90
91 class DecisionTreeRegressorScratch:
92     """
93     Decision Tree Regressor from scratch (CART-like, MSE
94     criterion).
95     """
96     def __init__(self, max_depth: int = None, min_samples_split:
97                   int = 2, random_state: int = 42):
98         self.max_depth = max_depth
99         self.min_samples_split = min_samples_split
100        self.random_state = random_state
101        self.root = None
102        self.feature_idx = None
103
104        def fit(self, X: np.ndarray, y: np.ndarray):
105            np.random.seed(self.random_state)
106            self.n_features = X.shape[1]
107            self.feature_idx = list(range(self.n_features))
108            self.root = build_tree(X, y, self.feature_idx, self.
109                                  max_depth, self.min_samples_split)
110            return self
111
112        def predict(self, X: np.ndarray) -> np.ndarray:
113            return np.array([predict_tree(self.root, x) for x in X])

```



```

109
110 class RandomForestRegressorScratch:
111     """
112     Random Forest Regressor from scratch: Ensemble of decision
113     trees with bagging and random features.
114     """
115     def __init__(self, n_estimators: int = 100, max_depth: int =
116         3, min_samples_split: int = 2, max_features: str = 'sqrt',
117         random_state: int = 42):
118         self.n_estimators = n_estimators
119         self.max_depth = max_depth
120         self.min_samples_split = min_samples_split
121         self.max_features = max_features
122         self.random_state = random_state
123         self.trees = []
124         self.feature_subsets = []
125
126     def _bootstrap_sample(self, X: np.ndarray, y: np.ndarray) ->
127         Tuple[np.ndarray, np.ndarray]:
128         n_samples = X.shape[0]
129         idxs = np.random.choice(n_samples, n_samples, replace=
130             True)
131         return X[idxs], y[idxs]
132
133     def _get_feature_idxes(self, n_features: int) -> List[int]:
134         if self.max_features == 'sqrt':
135             return np.random.choice(n_features, max(1, int(np.
136                 sqrt(n_features))), replace=False).tolist()
137         elif self.max_features == 'log2':
138             return np.random.choice(n_features, max(1, int(np.
139                 log2(n_features))), replace=False).tolist()
140         else:
141             raise ValueError("max_features must be 'sqrt' or '
142                 log2'")
143
144     def fit(self, X: np.ndarray, y: np.ndarray):
145         np.random.seed(self.random_state)
146         n_features = X.shape[1]
147         for _ in range(self.n_estimators):
148             X_boot, y_boot = self._bootstrap_sample(X, y)
149             feature_idxes = self._get_feature_idxes(n_features)

```

```
142         tree = DecisionTreeRegressorScratch(  
143             max_depth=self.max_depth,  
144             min_samples_split=self.min_samples_split,  
145             random_state=np.random.randint(0, 1000) # Vary  
                seed per tree  
146         )  
147         tree.feature_idx = feature_idx # Assign random  
                features  
148         tree.fit(X_boot, y_boot)  
149         self.trees.append(tree)  
150         self.feature_subsets.append(feature_idx)  
151     return self  
152  
153     def predict(self, X: np.ndarray) -> np.ndarray:  
154         predictions = np.array([tree.predict(X) for tree in self.  
                trees])  
155         return np.mean(predictions, axis=0)  
156  
157 # Instantiate the model (hyperparameters to roughly match  
    original: 100 trees, depth 3, sqrt features)  
158 rf_model = RandomForestRegressorScratch(  
159     n_estimators=100,  
160     max_depth=3,  
161     min_samples_split=2,  
162     max_features='sqrt',  
163     random_state=42  
164 )  
165  
166 # Train the model on the full dataset  
167 rf_model.fit(X, y)  
168  
169 # Compute R score manually  
170 y_pred = rf_model.predict(X)  
171 ss_res = np.sum((y - y_pred) ** 2)  
172 ss_tot = np.sum((y - np.mean(y)) ** 2)  
173 r2_score = 1 - (ss_res / ss_tot)  
174  
175 # Save the trained model to a file for later use  
176 model_path = 'rf_insurance_model_scratch.pkl'  
177 with open(model_path, 'wb') as f:  
178     pickle.dump(rf_model, f)
```

```
179
180 print(f"Model_saved_to_{model_path}")
181 print(f"Training_R_score:{r2_score:.4f}")
182 print(f"Number_of_trees:{len(rf_model.trees)}")
```

## 7.4 Test Script

```
1 import numpy as np
2 import pickle
3 import pandas as pd
4 from sklearn.tree import DecisionTreeRegressor # Only for GB
5 prediction (loaded trees)
6 import os
7 from typing import Tuple, List
8
9 # Path to the dataset (for evaluation)
10 data_path = "/home/midstan/Documents/Health_Insurance_Premium/
11             Model/Converting_Dataset_for_Training/Dataset/
12             insurance_converted.csv"
13
14 # Define the GradientBoostingRegressor class (must match the one
15 # used during training)
16 class GradientBoostingRegressor:
17     # ... (full class definition as above)
18
19 # Define LinearRegressionScratch class
20 class LinearRegressionScratch:
21     # ... (full class definition as above)
22
23 # Define Random Forest Scratch classes
24 class Node:
25     # ... (full class definition as above)
26
27 # ... (all other functions and classes as in training script)
28
29 # Manual train_test_split function
30 def manual_train_test_split(X, y, test_size=0.2, random_state=42)
31 :
32     np.random.seed(random_state)
33     indices = np.arange(X.shape[0])
```

```
29     np.random.shuffle(indices)
30     split = int((1 - test_size) * len(indices))
31     train_idx = indices[:split]
32     test_idx = indices[split:]
33     return X[train_idx], X[test_idx], y[train_idx], y[test_idx]
34
35 # Manual MAE and R2 functions
36 def manual_mean_absolute_error(y_true, y_pred):
37     return np.mean(np.abs(y_true - y_pred))
38
39 def manual_r2_score(y_true, y_pred):
40     ss_res = np.sum((y_true - y_pred) ** 2)
41     ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
42     return 1 - (ss_res / ss_tot)
43
44 # Model paths (full paths for scratch models; adjust GB if needed
45 )
46 model_paths = {
47     'gb': '/home/midstan/Documents/Health_Insurance_Premium/
48         gb_insurance_model.pkl',
49     'rf': '/home/midstan/Documents/Health_Insurance_Premium/
50         rf_insurance_model_scratch.pkl',
51     'lr': '/home/midstan/Documents/Health_Insurance_Premium/
52         lr_insurance_model_scratch.pkl'
53 }
54
55 # Load models
56 models = {}
57 loaded_successfully = []
58 for name, path in model_paths.items():
59     try:
60         with open(path, 'rb') as f:
61             if name == 'gb':
62                 models[name] = pickle.load(f)
63             elif name == 'rf':
64                 models[name] = pickle.load(f)
65             elif name == 'lr':
66                 models[name] = pickle.load(f)
67             print(f"{name.upper()}_model_loaded_successfully!")
68             loaded_successfully.append(name)
69     except FileNotFoundError:
```

```

66         print(f"Error: Model file '{path}' not found. Skipping {
            name}_model.")
67     except Exception as e:
68         print(f"Error loading {name}_model: {e}. Skipping.")
69
70 if not loaded_successfully:
71     print("No models loaded. Exiting.")
72     exit(1)
73
74 # Evaluation on test set
75 print("\n" + "="*50)
76 print("MODEL PERFORMANCE COMPARISON ON TEST SET (80/20 split)")
77 print("="*50)
78 try:
79     # Load data for evaluation
80     df = pd.read_csv(data_path)
81     X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region'
82             ]].values
83     y = df['expenses'].values
84     X_train, X_test, y_train, y_test = manual_train_test_split(X,
85         y, test_size=0.2, random_state=42)
86     print("Model\t\tMAE\t\tR  _Score")
87     print("-" * 40)
88     for name in loaded_successfully:
89         model = models[name]
90         y_pred = model.predict(X_test)
91         mae = manual_mean_absolute_error(y_test, y_pred)
92         r2 = manual_r2_score(y_test, y_pred)
93         print(f"{name}\t\t{mae:.2f}\t\t{r2:.4f}")
94     print("Evaluation completed successfully!")
95 except Exception as e:
96     print(f"Error during evaluation: {e}")
97     import traceback
98     traceback.print_exc() # Print full traceback for debugging
99
100 print("\n" + "="*50)
101 print("INTERACTIVE PREDICTION MODE")
102 print("="*50)
103
104 def get_user_input():
105     """

```

```
104 Collects user input for the features in the correct order: [
    age, sex, bmi, children, smoker, region]
105 With robust error handling for invalid inputs.
106 """
107 print("\nEnter the following details for insurance premium
    prediction:")
108
109 # Helper function to get validated int input
110 def get_valid_int(prompt, min_val=None, max_val=None):
111     while True:
112         try:
113             value = input(prompt).strip()
114             if not value:
115                 print("Input cannot be empty. Please try
                    again.")
116                 continue
117             int_val = int(value)
118             if min_val is not None and int_val < min_val:
119                 print(f"Value must be at least {min_val}.
                    Please try again.")
120                 continue
121             if max_val is not None and int_val > max_val:
122                 print(f"Value must be at most {max_val}.
                    Please try again.")
123                 continue
124             return int_val
125         except ValueError:
126             print("Invalid integer value. Please enter a
                    valid number.")
127
128 # Helper function to get validated float input
129 def get_valid_float(prompt, min_val=None, max_val=None):
130     while True:
131         try:
132             value = input(prompt).strip()
133             if not value:
134                 print("Input cannot be empty. Please try
                    again.")
135                 continue
136             float_val = float(value)
137             if min_val is not None and float_val < min_val:
```

```
138         print(f"Value must be at least {min_val}.  
139             Please try again.")  
140         continue  
141     if max_val is not None and float_val > max_val:  
142         print(f"Value must be at most {max_val}.  
143             Please try again.")  
144         continue  
145     return float_val  
146 except ValueError:  
147     print("Invalid float value. Please enter a valid  
148         number.")  
149  
150 age = get_valid_int("Age (18-100):", min_val=18, max_val  
151     =100)  
152 sex = get_valid_int("Sex (0 for female, 1 for male):",  
153     min_val=0, max_val=1)  
154 bmi = get_valid_float("BMI (10-60):", min_val=10, max_val  
155     =60)  
156 children = get_valid_int("Number of children (0-5):",  
157     min_val=0, max_val=5)  
158 smoker = get_valid_int("Smoker (0 for no, 1 for yes):",  
159     min_val=0, max_val=1)  
160 region = get_valid_int("Region (1-4):", min_val=1, max_val  
161     =4)  
162  
163 return np.array([[age, sex, bmi, children, smoker, region]])  
164  
165 # Interactive prediction loop  
166 while True:  
167     new_X = get_user_input()  
168     # Make predictions with all loaded models  
169     print("\nPredicted insurance expenses from all models:")  
170     print("-" * 40)  
171     for name in loaded_successfully:  
172         model = models[name]  
173         prediction = model.predict(new_X)  
174         print(f"{name.upper()}: ${prediction[0]:.2f}")  
175     # Ask if user wants to predict another  
176     while True:  
177         continue_choice = input("\nPredict another? (y/n):").  
178             lower().strip()
```

```
169         if continue_choice in ['y', 'yes']:
170             break
171         elif continue_choice in ['n', 'no']:
172             print("Goodbye!")
173             exit(0)
174         else:
175             print("Please enter 'y' for yes or 'n' for no.")
```



# Chapter 8

## Presentation Slides Outline

- Slide 1: Title and Agenda
- Slide 2: Problem Statement
- Slide 3: Dataset Overview
- Slide 4: Linear Regression Details
- Slide 5: Gradient Boosting
- Slide 6: Random Forest
- Slide 7: Implementation Flow
- Slide 8: Results Table
- Slide 9: Demo Screenshots
- Slide 10: Challenges and Insights
- Slide 11: Conclusion
- Slide 12: Q&A

Note: Slides prepared in PowerPoint/Google Slides with code snippets and plots.