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

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:

```
df = pd.read_csv(data_path)
X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].
    values
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:

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

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

Code snippet:

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

def predict(self, X):
```

```
    # Add bias term
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    # Predictions: X_b * theta
    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 η:

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

Early stopping if MAE ¡ tol.

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

Code snippet:

```
def fit(self, X, y):
    # Initialize predictions with the mean (base model)
    self.initial_prediction = np.mean(y)
    predictions = np.full_like(y, self.initial_prediction, dtype=
        np.float64)
    # Compute initial residuals
    residuals = y - predictions
    # Build trees iteratively
    for i in range(self.n_estimators):
        # Fit a decision tree to the current residuals (negative
            gradient for regression)
        tree = DecisionTreeRegressor(max_depth=self.max_depth,
            random_state=42)
        tree.fit(X, residuals)
        # Predict residuals with the tree
        tree_pred = tree.predict(X)
        # Update predictions: add learning_rate * tree prediction
        predictions += self.learning_rate * tree_pred
        # Update residuals
        residuals = y - predictions
        # Store the tree
        self.trees.append(tree)
        # Early stopping: check if mean absolute residual is
            below tolerance
        mean_abs_residual = np.mean(np.abs(residuals))
        if mean_abs_residual < self.tol:
```

```
                print(f"Early stopping at iteration {i+1}: mean abs residual = {mean_abs_residual:.2f}")
                break
        print(f"Training completed. Final mean abs residual: {np.mean(np.abs(residuals)):.2f}")
        print(f"Number of trees built: {len(self.trees)}")
        return self

    def predict(self, X):
        # Start with initial prediction
        predictions = np.full(X.shape[0], self.initial_prediction, dtype=np.float64)
        # Add contributions from all trees
        for tree in self.trees:
            predictions += self.learning_rate * tree.predict(X)
        return predictions
```

3.2.3 Random Forest Regressor from Scratch

Includes custom DecisionTreeRegressor using MSE for splits:

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

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

Code snippet (Decision Tree fit):

```
def mse(y: np.ndarray) -> float:
    """
    Mean Squared Error for a set of targets.
    """
    return np.mean((y - np.mean(y)) ** 2)

def best_split(X: np.ndarray, y: np.ndarray, feature_idxs: List[int]) -> Tuple[int, float, float, np.ndarray, np.ndarray]:
    """
    Find the best split: feature and threshold minimizing MSE.
    """
    best_idx, best_thresh, best_mse, best_left, best_right = None, None, float('inf'), None, None
    current_mse = mse(y)

```

```python
    for idx in feature_idxs:
        thresholds = np.unique(X[:, idx])
        for thresh in thresholds:
            left_mask = X[:, idx] <= thresh
            right_mask = ~left_mask
            if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
                continue
            left_y, right_y = y[left_mask], y[right_mask]
            weighted_mse = (len(left_y) * mse(left_y) + len(right_y) * mse(right_y)) / len(y)
            if weighted_mse < best_mse:
                best_idx = idx
                best_thresh = thresh
                best_mse = weighted_mse
                best_left = left_y
                best_right = right_y

    return best_idx, best_thresh, best_mse, best_left, best_right

def build_tree(X: np.ndarray, y: np.ndarray, feature_idxs: List[int], max_depth: int = None, min_samples_split: int = 2, depth: int = 0) -> Node:
    """
    Recursively build the decision tree.
    """
    if len(y) < min_samples_split or (max_depth is not None and depth >= max_depth):
        return Node(value=np.mean(y))

    idx, thresh, _, _, _ = best_split(X, y, feature_idxs)
    if idx is None:
        return Node(value=np.mean(y))

    left_mask = X[:, idx] <= thresh
    right_mask = ~left_mask

    left = build_tree(X[left_mask], y[left_mask], feature_idxs, max_depth, min_samples_split, depth + 1)
    right = build_tree(X[right_mask], y[right_mask], feature_idxs, max_depth, min_samples_split, depth + 1)

```

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

For forest:

```
def fit(self, X: np.ndarray, y: np.ndarray):
    np.random.seed(self.random_state)
    n_features = X.shape[1]
    for _ in range(self.n_estimators):
        X_boot, y_boot = self._bootstrap_sample(X, y)
        feature_idxs = self._get_feature_idxs(n_features)
        tree = DecisionTreeRegressorScratch(
            max_depth=self.max_depth,
            min_samples_split=self.min_samples_split,
            random_state=np.random.randint(0, 1000)  # Vary seed
                per tree
        )
        tree.feature_idxs = feature_idxs  # Assign random
            features
        tree.fit(X_boot, y_boot)
        self.trees.append(tree)
        self.feature_subsets.append(feature_idxs)
    return self

def predict(self, X: np.ndarray) -> np.ndarray:
    predictions = np.array([tree.predict(X) for tree in self.
        trees])
    return np.mean(predictions, axis=0)
```

3.3 Evaluation Metrics

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

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

5.2 Insights

- Smoker feature dominates coefficients (≈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

```
import pandas as pd
import numpy as np
import pickle
import os

# Path to the dataset
data_path = "/home/midstan/Documents/Health Insurance Premium/Model/Converting Dataset for Training/Dataset/insurance_converted.csv"

# Load the dataset
df = pd.read_csv(data_path)

# Prepare features (X) and target (y)
X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].values
y = df['expenses'].values

class LinearRegressionScratch:
    """
    Linear Regression implemented from scratch using the normal equation.
    """
    def __init__(self):
        self.coef_ = None
        self.intercept_ = None

```

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

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

# Instantiate the model
lr_model = LinearRegressionScratch()

# Train the model on the full dataset
lr_model.fit(X, y)

# Compute R  score manually
y_pred = lr_model.predict(X)
ss_res = np.sum((y - y_pred) ** 2)
ss_tot = np.sum((y - np.mean(y)) ** 2)
r2_score = 1 - (ss_res / ss_tot)

# Save the trained model to a file for later use
model_path = 'lr_insurance_model_scratch.pkl'
with open(model_path, 'wb') as f:
    pickle.dump(lr_model, f)

print(f"Model saved to {model_path}")
print(f"Training R  score: {r2_score:.4f}")
print(f"Intercept: {lr_model.intercept_:.2f}")
print(f"Coefficients: {lr_model.coef_}")
```

7.2 Gradient Boosting

```python
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeRegressor
import pickle
import os

# Path to the dataset
data_path = "/home/midstan/Documents/Health Insurance Premium/Model/Converting Dataset for Training/Dataset/insurance_converted.csv"

# Load the dataset
df = pd.read_csv(data_path)

# Prepare features (X) and target (y)
X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].values
y = df['expenses'].values

class GradientBoostingRegressor:
    """
    A simple implementation of Gradient Boosting for Regression.
    - Base model: Mean of the target values.
    - Weak learners: Decision Trees fitted to negative gradients (residuals).
    - Stops early if mean absolute residual is below a tolerance.
    """
    def __init__(self, n_estimators=100, learning_rate=0.1, max_depth=3, tol=1.0):
        """
        :param n_estimators: Maximum number of trees.
        :param learning_rate: Shrinkage factor for each tree's contribution.
        :param max_depth: Maximum depth of each decision tree.
        :param tol: Tolerance for mean absolute residual to stop early.
        """
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.max_depth = max_depth
```

```
        self.tol = tol # Tolerance for residuals (adjusted for expenses scale ~thousands)
        self.trees = [] # List to store decision trees
        self.initial_prediction = None # Mean as base prediction

    def fit(self, X, y):
        # Initialize predictions with the mean (base model)
        self.initial_prediction = np.mean(y)
        predictions = np.full_like(y, self.initial_prediction, dtype=np.float64)
        # Compute initial residuals
        residuals = y - predictions
        # Build trees iteratively
        for i in range(self.n_estimators):
            # Fit a decision tree to the current residuals (negative gradient for regression)
            tree = DecisionTreeRegressor(max_depth=self.max_depth, random_state=42)
            tree.fit(X, residuals)
            # Predict residuals with the tree
            tree_pred = tree.predict(X)
            # Update predictions: add learning_rate * tree prediction
            predictions += self.learning_rate * tree_pred
            # Update residuals
            residuals = y - predictions
            # Store the tree
            self.trees.append(tree)
            # Early stopping: check if mean absolute residual is below tolerance
            mean_abs_residual = np.mean(np.abs(residuals))
            if mean_abs_residual < self.tol:
                print(f"Early stopping at iteration {i+1}: mean abs residual = {mean_abs_residual:.2f}")
                break
        print(f"Training completed. Final mean abs residual: {np.mean(np.abs(residuals)):.2f}")
        print(f"Number of trees built: {len(self.trees)}")
        return self

    def predict(self, X):
```

```
        # Start with initial prediction
        predictions = np.full(X.shape[0], self.initial_prediction
            , dtype=np.float64)
        # Add contributions from all trees
        for tree in self.trees:
            predictions += self.learning_rate * tree.predict(X)
        return predictions

# Instantiate the model (adjust hyperparameters as needed)
gb_model = GradientBoostingRegressor(
    n_estimators=100000,
    learning_rate=0.1,
    max_depth=3,
    tol=1.0 # Stop if mean abs residual < 1 (small relative to
        expenses)
)

# Train the model on the full dataset
gb_model.fit(X, y)

# Save the trained model to a file for later use
model_path = 'gb_insurance_model.pkl'
with open(model_path, 'wb') as f:
    pickle.dump(gb_model, f)
print(f"Model saved to {model_path}")

# Example: How to load and predict after training
# (Uncomment to test if you have sample input data)
# Load the model
with open(model_path, 'rb') as f:
    loaded_model = pickle.load(f)
# Example new input (must match feature order and types)
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

```
import pandas as pd
import numpy as np
```

```python
import pickle
import os
from typing import Tuple, List

# Path to the dataset
data_path = "/home/midstan/Documents/Health Insurance Premium/Model/Converting Dataset for Training/Dataset/insurance_converted.csv"

# Load the dataset
df = pd.read_csv(data_path)

# Prepare features (X) and target (y)
X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].values
y = df['expenses'].values

class Node:
    """
    Node in the decision tree.
    """
    def __init__(self, feature_idx=None, threshold=None, left=None, right=None, value=None):
        self.feature_idx = feature_idx
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value  # Leaf node value (mean of targets)

def mse(y: np.ndarray) -> float:
    """
    Mean Squared Error for a set of targets.
    """
    return np.mean((y - np.mean(y)) ** 2)

def best_split(X: np.ndarray, y: np.ndarray, feature_idxs: List[int]) -> Tuple[int, float, float, np.ndarray, np.ndarray]:
    """
    Find the best split: feature and threshold minimizing MSE.
    """
```

```python
    best_idx, best_thresh, best_mse, best_left, best_right = None, None, float('inf'), None, None
    current_mse = mse(y)

    for idx in feature_idxs:
        thresholds = np.unique(X[:, idx])
        for thresh in thresholds:
            left_mask = X[:, idx] <= thresh
            right_mask = ~left_mask
            if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
                continue
            left_y, right_y = y[left_mask], y[right_mask]
            weighted_mse = (len(left_y) * mse(left_y) + len(right_y) * mse(right_y)) / len(y)
            if weighted_mse < best_mse:
                best_idx = idx
                best_thresh = thresh
                best_mse = weighted_mse
                best_left = left_y
                best_right = right_y

    return best_idx, best_thresh, best_mse, best_left, best_right

def build_tree(X: np.ndarray, y: np.ndarray, feature_idxs: List[int], max_depth: int = None, min_samples_split: int = 2, depth: int = 0) -> Node:
    """
    Recursively build the decision tree.
    """
    if len(y) < min_samples_split or (max_depth is not None and depth >= max_depth):
        return Node(value=np.mean(y))

    idx, thresh, _, _, _ = best_split(X, y, feature_idxs)
    if idx is None:
        return Node(value=np.mean(y))

    left_mask = X[:, idx] <= thresh
    right_mask = ~left_mask

```

```
    left = build_tree(X[left_mask], y[left_mask], feature_idxs,
        max_depth, min_samples_split, depth + 1)
    right = build_tree(X[right_mask], y[right_mask], feature_idxs
        , max_depth, min_samples_split, depth + 1)

    return Node(idx, thresh, left, right)

def predict_tree(node: Node, x: np.ndarray) -> float:
    """
    Predict with a single tree.
    """
    if node.value is not None:
        return node.value
    if x[node.feature_idx] <= node.threshold:
        return predict_tree(node.left, x)
    else:
        return predict_tree(node.right, x)

class DecisionTreeRegressorScratch:
    """
    Decision Tree Regressor from scratch (CART-like, MSE
       criterion).
    """
    def __init__(self, max_depth: int = None, min_samples_split:
        int = 2, random_state: int = 42):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.random_state = random_state
        self.root = None
        self.feature_idxs = None

    def fit(self, X: np.ndarray, y: np.ndarray):
        np.random.seed(self.random_state)
        self.n_features = X.shape[1]
        self.feature_idxs = list(range(self.n_features))
        self.root = build_tree(X, y, self.feature_idxs, self.
            max_depth, self.min_samples_split)
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        return np.array([predict_tree(self.root, x) for x in X])
```

```python

class RandomForestRegressorScratch:
    """
    Random Forest Regressor from scratch: Ensemble of decision trees with bagging and random features.
    """
    def __init__(self, n_estimators: int = 100, max_depth: int = 3, min_samples_split: int = 2, max_features: str = 'sqrt', random_state: int = 42):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.max_features = max_features
        self.random_state = random_state
        self.trees = []
        self.feature_subsets = []

    def _bootstrap_sample(self, X: np.ndarray, y: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        n_samples = X.shape[0]
        idxs = np.random.choice(n_samples, n_samples, replace=True)
        return X[idxs], y[idxs]

    def _get_feature_idxs(self, n_features: int) -> List[int]:
        if self.max_features == 'sqrt':
            return np.random.choice(n_features, max(1, int(np.sqrt(n_features))), replace=False).tolist()
        elif self.max_features == 'log2':
            return np.random.choice(n_features, max(1, int(np.log2(n_features))), replace=False).tolist()
        else:
            raise ValueError("max_features must be 'sqrt' or 'log2'")

    def fit(self, X: np.ndarray, y: np.ndarray):
        np.random.seed(self.random_state)
        n_features = X.shape[1]
        for _ in range(self.n_estimators):
            X_boot, y_boot = self._bootstrap_sample(X, y)
            feature_idxs = self._get_feature_idxs(n_features)
```

```
            tree = DecisionTreeRegressorScratch(
                max_depth=self.max_depth,
                min_samples_split=self.min_samples_split,
                random_state=np.random.randint(0, 1000)  # Vary seed per tree
            )
            tree.feature_idxs = feature_idxs  # Assign random features
            tree.fit(X_boot, y_boot)
            self.trees.append(tree)
            self.feature_subsets.append(feature_idxs)
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        predictions = np.array([tree.predict(X) for tree in self.trees])
        return np.mean(predictions, axis=0)

# Instantiate the model (hyperparameters to roughly match original: 100 trees, depth 3, sqrt features)
rf_model = RandomForestRegressorScratch(
    n_estimators=100,
    max_depth=3,
    min_samples_split=2,
    max_features='sqrt',
    random_state=42
)

# Train the model on the full dataset
rf_model.fit(X, y)

# Compute R  score manually
y_pred = rf_model.predict(X)
ss_res = np.sum((y - y_pred) ** 2)
ss_tot = np.sum((y - np.mean(y)) ** 2)
r2_score = 1 - (ss_res / ss_tot)

# Save the trained model to a file for later use
model_path = 'rf_insurance_model_scratch.pkl'
with open(model_path, 'wb') as f:
    pickle.dump(rf_model, f)
```

```python

print(f"Model saved to {model_path}")
print(f"Training R  score: {r2_score:.4f}")
print(f"Number of trees: {len(rf_model.trees)}")
```

7.4 Test Script

```python
import numpy as np
import pickle
import pandas as pd
from sklearn.tree import DecisionTreeRegressor # Only for GB prediction (loaded trees)
import os
from typing import Tuple, List

# Path to the dataset (for evaluation)
data_path = "/home/midstan/Documents/Health Insurance Premium/Model/Converting Dataset for Training/Dataset/insurance_converted.csv"

# Define the GradientBoostingRegressor class (must match the one used during training)
class GradientBoostingRegressor:
    # ... (full class definition as above)

# Define LinearRegressionScratch class
class LinearRegressionScratch:
    # ... (full class definition as above)

# Define Random Forest Scratch classes
class Node:
    # ... (full class definition as above)

# ... (all other functions and classes as in training script)

# Manual train_test_split function
def manual_train_test_split(X, y, test_size=0.2, random_state=42):
    np.random.seed(random_state)
    indices = np.arange(X.shape[0])
```

```
    np.random.shuffle(indices)
    split = int((1 - test_size) * len(indices))
    train_idx = indices[:split]
    test_idx = indices[split:]
    return X[train_idx], X[test_idx], y[train_idx], y[test_idx]

# Manual MAE and R2 functions
def manual_mean_absolute_error(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

def manual_r2_score(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)

# Model paths (full paths for scratch models; adjust GB if needed
    )
model_paths = {
    'gb': '/home/midstan/Documents/Health Insurance Premium/
        gb_insurance_model.pkl',
    'rf': '/home/midstan/Documents/Health Insurance Premium/
        rf_insurance_model_scratch.pkl',
    'lr': '/home/midstan/Documents/Health Insurance Premium/
        lr_insurance_model_scratch.pkl'
}

# Load models
models = {}
loaded_successfully = []
for name, path in model_paths.items():
    try:
        with open(path, 'rb') as f:
            if name == 'gb':
                models[name] = pickle.load(f)
            elif name == 'rf':
                models[name] = pickle.load(f)
            elif name == 'lr':
                models[name] = pickle.load(f)
        print(f"{name.upper()} model loaded successfully!")
        loaded_successfully.append(name)
    except FileNotFoundError:
```

```
            print(f"Error: Model file '{path}' not found. Skipping {name} model.")
        except Exception as e:
            print(f"Error loading {name} model: {e}. Skipping.")

if not loaded_successfully:
    print("No models loaded. Exiting.")
    exit(1)

# Evaluation on test set
print("\n" + "="*50)
print("MODEL PERFORMANCE COMPARISON ON TEST SET (80/20 split)")
print("="*50)
try:
    # Load data for evaluation
    df = pd.read_csv(data_path)
    X = df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].values
    y = df['expenses'].values
    X_train, X_test, y_train, y_test = manual_train_test_split(X, y, test_size=0.2, random_state=42)
    print("Model\t\tMAE\t\tR  Score")
    print("-" * 40)
    for name in loaded_successfully:
        model = models[name]
        y_pred = model.predict(X_test)
        mae = manual_mean_absolute_error(y_test, y_pred)
        r2 = manual_r2_score(y_test, y_pred)
        print(f"{name}\t\t{mae:.2f}\t\t{r2:.4f}")
    print("Evaluation completed successfully!")
except Exception as e:
    print(f"Error during evaluation: {e}")
    import traceback
    traceback.print_exc()  # Print full traceback for debugging

print("\n" + "="*50)
print("INTERACTIVE PREDICTION MODE")
print("="*50)

def get_user_input():
    """
```

```
    Collects user input for the features in the correct order: [age, sex, bmi, children, smoker, region]
    with robust error handling for invalid inputs.
    """
    print("\nEnter the following details for insurance premium prediction:")

    # Helper function to get validated int input
    def get_valid_int(prompt, min_val=None, max_val=None):
        while True:
            try:
                value = input(prompt).strip()
                if not value:
                    print("Input cannot be empty. Please try again.")
                    continue
                int_val = int(value)
                if min_val is not None and int_val < min_val:
                    print(f"Value must be at least {min_val}. Please try again.")
                    continue
                if max_val is not None and int_val > max_val:
                    print(f"Value must be at most {max_val}. Please try again.")
                    continue
                return int_val
            except ValueError:
                print("Invalid integer value. Please enter a valid number.")

    # Helper function to get validated float input
    def get_valid_float(prompt, min_val=None, max_val=None):
        while True:
            try:
                value = input(prompt).strip()
                if not value:
                    print("Input cannot be empty. Please try again.")
                    continue
                float_val = float(value)
                if min_val is not None and float_val < min_val:
```

```
                    print(f"Value must be at least {min_val}. Please try again.")
                    continue
                if max_val is not None and float_val > max_val:
                    print(f"Value must be at most {max_val}. Please try again.")
                    continue
                return float_val
            except ValueError:
                print("Invalid float value. Please enter a valid number.")

    age = get_valid_int("Age (18-100): ", min_val=18, max_val=100)
    sex = get_valid_int("Sex (0 for female, 1 for male): ", min_val=0, max_val=1)
    bmi = get_valid_float("BMI (10-60): ", min_val=10, max_val=60)
    children = get_valid_int("Number of children (0-5): ", min_val=0, max_val=5)
    smoker = get_valid_int("Smoker (0 for no, 1 for yes): ", min_val=0, max_val=1)
    region = get_valid_int("Region (1-4): ", min_val=1, max_val=4)

    return np.array([[age, sex, bmi, children, smoker, region]])

# Interactive prediction loop
while True:
    new_X = get_user_input()
    # Make predictions with all loaded models
    print("\nPredicted insurance expenses from all models:")
    print("-" * 40)
    for name in loaded_successfully:
        model = models[name]
        prediction = model.predict(new_X)
        print(f"{name.upper()}: ${prediction[0]:.2f}")
    # Ask if user wants to predict another
    while True:
        continue_choice = input("\nPredict another? (y/n): ").lower().strip()
```

```python
        if continue_choice in ['y', 'yes']:
            break
        elif continue_choice in ['n', 'no']:
            print("Goodbye!")
            exit(0)
        else:
            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.