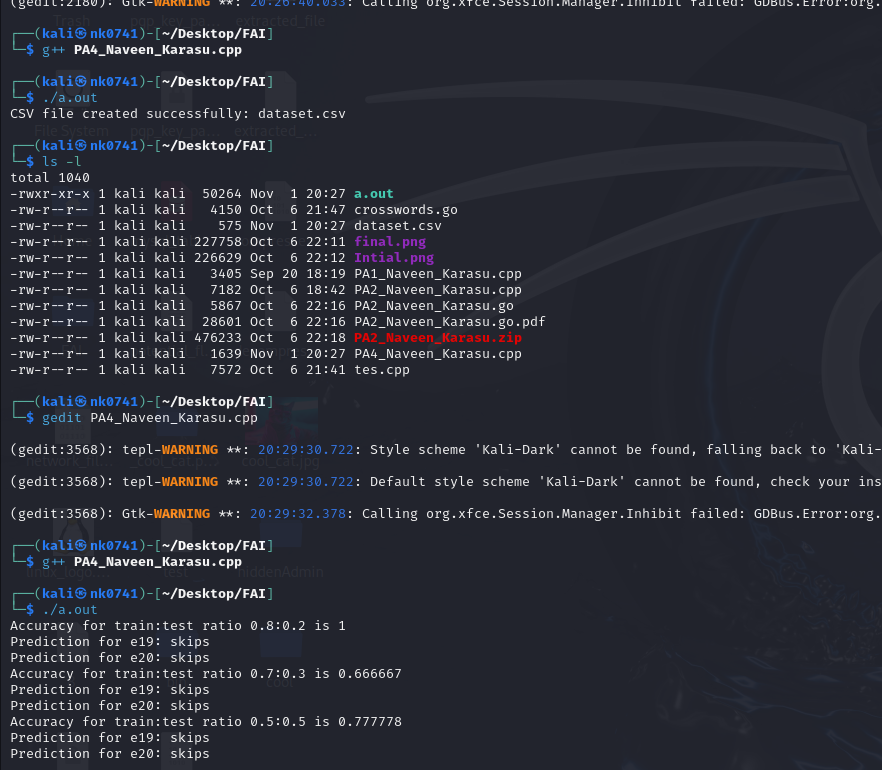
Programming Asssignment -4

This C++ code is used to design a decision tree model which is used for classifying the data present on a CSV dataset. Loads the dataset using load\_data function which then parses each entry into a structured format. Next, split\_data function divides the dataset into training and testing sets with different ratios: 80:20, 70:30, 50:50 to assess the performance model under different conditions. The train\_decision\_tree function recursively builds the tree by selecting the best feature at each node to maximize the information gain and using the entropy calculations which is used for measure data purity. The trained tree is then tested by using it do the predict function to classify new examples by following feature-based paths.

The calculate\_accuracy function evaluates the model by checking its predictions against actual labels on the test set, providing an accuracy metric for each split.

**output:**  
  
**Code:**  
#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <map>

#include <cmath>

#include <algorithm>

#include <cstdlib>

using namespace std;

// Data structure to store each example's attributes and user action

struct Example {

string author;

string thread;

string length;

string where\_read;

string user\_action;

};

/\*\*

\* @brief Loads data from a CSV file

\*

\* Reads a CSV file where each line represents an example with the following

\* columns: Author, Thread, Length, Where\_read, and User\_action.

\* These columns correspond to characteristics of each example.

\*

\* @param filename Name of the CSV file to load data from.

\* @return vector<Example> The parsed dataset, where each Example contains

\* values for the above-mentioned columns.

\*

\* Example CSV structure:

\* Author, Thread, Length, Where\_read, User\_action

\* known, new, long, home, skips

\* unknown, new, short, work, reads

\*/

vector<Example> load\_data(const string &filename) {

vector<Example> data;

ifstream file(filename);

string line, word;

// Skip header line

getline(file, line);

while (getline(file, line)) {

stringstream ss(line);

Example example;

getline(ss, example.author, ',');

getline(ss, example.thread, ',');

getline(ss, example.length, ',');

getline(ss, example.where\_read, ',');

getline(ss, example.user\_action, ',');

data.push\_back(example);

}

return data;

}

/\*\*

\* @brief Splits the dataset into training and testing sets

\*

\* This function divides the dataset into two parts: training and testing sets,

\* based on a specified ratio. We create multiple sets with different ratios

\* (80:20, 70:30, 50:50) to evaluate the model's performance on different data splits.

\*

\* Why different splits?

\* - Using multiple train-test splits allows us to understand how the model performs

\* with different amounts of training data.

\* - The more training data, the more accurate the model might be, but it’s good to

\* evaluate different scenarios.

\*

\* @param data Original dataset to split.

\* @param train Vector to store the training set.

\* @param test Vector to store the testing set.

\* @param train\_ratio Ratio of training data to total dataset.

\*/

void split\_data(const vector<Example>& data, vector<Example>& train, vector<Example>& test, float train\_ratio) {

int train\_size = static\_cast<int>(data.size() \* train\_ratio);

train.assign(data.begin(), data.begin() + train\_size);

test.assign(data.begin() + train\_size, data.end());

}

// Node structure representing a decision tree node

struct TreeNode {

string feature;

map<string, TreeNode\*> children;

string label;

};

/\*\*

\* @brief Calculates entropy of a dataset

\*

\* Entropy is a measure of the disorder or impurity in the data.

\* For a dataset, entropy is calculated based on the distribution of

\* labels (user actions in this case).

\*

\* Formula: Entropy = -Σ (p \* log2(p))

\* where p is the probability of each label in the data.

\*

\* Example Calculation:

\* - Suppose the dataset has 10 examples, 6 labeled "reads" and 4 labeled "skips".

\* - Probability of "reads" = 6/10 = 0.6

\* - Probability of "skips" = 4/10 = 0.4

\* - Entropy = -(0.6 \* log2(0.6) + 0.4 \* log2(0.4)) ≈ 0.971

\*

\* Use in Decision Tree:

\* - Entropy helps us determine the best feature to split the data. Lower entropy

\* means a feature creates "purer" subsets.

\*

\* @param data The dataset for which entropy is calculated.

\* @return double Entropy value.

\*/

double calculate\_entropy(const vector<Example>& data) {

map<string, int> label\_counts;

for (const auto& ex : data) {

label\_counts[ex.user\_action]++;

}

double entropy = 0.0;

for (const auto& pair : label\_counts) {

double p = static\_cast<double>(pair.second) / data.size();

entropy -= p \* log2(p);

}

return entropy;

}

/\*\*

\* @brief Trains a decision tree based on the dataset

\*

\* This function builds a decision tree by recursively splitting the data

\* using features that minimize entropy (i.e., create the "purest" subsets).

\*

\* Steps:

\* 1. If all examples have the same label (e.g., all "reads"), create a leaf node

\* with that label and stop.

\* 2. If there are mixed labels, calculate entropy for each feature to find the

\* best split. The best feature is the one that gives the highest information gain.

\* 3. Divide data based on this feature and recursively build child nodes.

\*

\* Example:

\* Let's assume we have a small dataset with 4 examples:

\* - {Author: "known", Thread: "new", Length: "short", User\_action: "reads"}

\* - {Author: "unknown", Thread: "new", Length: "long", User\_action: "skips"}

\* - {Author: "known", Thread: "followup", Length: "short", User\_action: "reads"}

\* - {Author: "unknown", Thread: "followup", Length: "long", User\_action: "skips"}

\*

\* If splitting by "Author" reduces entropy the most, we split on "Author" and

\* create branches for "known" and "unknown". Each branch is further split

\* based on remaining features.

\*

\* @param data The training dataset.

\* @param features List of features to consider for splits.

\* @return TreeNode\* Pointer to the root node of the trained tree.

\*/

TreeNode\* train\_decision\_tree(const vector<Example>& data, const vector<string>& features) {

if (data.empty()) return nullptr;

// Check if all labels are the same

map<string, int> label\_counts;

for (const auto& ex : data) {

label\_counts[ex.user\_action]++;

}

if (label\_counts.size() == 1) {

TreeNode\* leaf = new TreeNode;

leaf->label = data[0].user\_action;

return leaf;

}

// Find the best feature for splitting

double base\_entropy = calculate\_entropy(data);

double best\_gain = 0.0;

string best\_feature;

map<string, vector<Example>> best\_splits;

for (const auto& feature : features) {

map<string, vector<Example>> splits;

for (const auto& ex : data) {

string value = feature == "author" ? ex.author :

feature == "thread" ? ex.thread :

feature == "length" ? ex.length :

ex.where\_read;

splits[value].push\_back(ex);

}

double new\_entropy = 0.0;

for (const auto& split : splits) {

double weight = static\_cast<double>(split.second.size()) / data.size();

new\_entropy += weight \* calculate\_entropy(split.second);

}

double gain = base\_entropy - new\_entropy;

if (gain > best\_gain) {

best\_gain = gain;

best\_feature = feature;

best\_splits = splits;

}

}

// Stop if no gain in entropy

if (best\_gain == 0) {

TreeNode\* leaf = new TreeNode;

leaf->label = data[0].user\_action;

return leaf;

}

TreeNode\* node = new TreeNode;

node->feature = best\_feature;

vector<string> remaining\_features;

for (const auto& feature : features) {

if (feature != best\_feature) remaining\_features.push\_back(feature);

}

for (const auto& split : best\_splits) {

node->children[split.first] = train\_decision\_tree(split.second, remaining\_features);

}

return node;

}

/\*\*

\* @brief Predicts the user action based on the trained tree

\*

\* This function traverses the decision tree to predict the user action for a new example.

\* Starting from the root, it checks the example’s feature values and moves down the tree

\* until it reaches a leaf node, which holds the prediction.

\*

\* @param root Pointer to the root of the decision tree.

\* @param example New example to classify.

\* @return string Predicted user action.

\*/

string predict(TreeNode\* root, const Example& example) {

if (root->children.empty()) return root->label;

string value = root->feature == "author" ? example.author :

root->feature == "thread" ? example.thread :

root->feature == "length" ? example.length :

example.where\_read;

if (root->children.count(value)) {

return predict(root->children[value], example);

} else {

return "unknown"; // Handle unseen data

}

}

/\*\*

\* @brief Calculates accuracy of the model on the test dataset

\*

\* This function compares the predicted labels to actual labels to measure

\* the model's performance on the test dataset.

\*

\* Accuracy Formula: (Number of Correct Predictions / Total Predictions)

\*

\* @param root Pointer to the root of the decision tree.

\* @param test\_data Test dataset for evaluation.

\* @return double Accuracy of the model.

\*/

double calculate\_accuracy(TreeNode\* root, const vector<Example>& test\_data) {

int correct = 0;

for (const auto& ex : test\_data) {

if (predict(root, ex) == ex.user\_action) {

correct++;

}

}

return static\_cast<double>(correct) / test\_data.size();

}

/\*\*

\* @brief Main function to train, test, and evaluate the decision tree model

\*/

int main() {

string filename = "dataset.csv"; // The generated CSV file

vector<Example> data = load\_data(filename);

// Define train-test split ratios

vector<float> ratios = {0.8, 0.7, 0.5};

for (float ratio : ratios) {

vector<Example> train\_data, test\_data;

split\_data(data, train\_data, test\_data, ratio);

// Train decision tree

vector<string> features = {"author", "thread", "length", "where\_read"};

TreeNode\* tree = train\_decision\_tree(train\_data, features);

// Calculate and display accuracy

double accuracy = calculate\_accuracy(tree, test\_data);

cout << "Accuracy for train:test ratio " << ratio << ":" << (1 - ratio) << " is " << accuracy << endl;

// Predict user actions for examples e19 and e20

Example e19 = {"unknown", "new", "long", "work", "?"};

Example e20 = {"unknown", "followup", "short", "home", "?"};

cout << "Prediction for e19: " << predict(tree, e19) << endl;

cout << "Prediction for e20: " << predict(tree, e20) << endl;

// Note: Add code to delete the tree if necessary to avoid memory leaks

}

return 0;

}