

Ahmet Zorer | CMPE 480 | December 18, 2016

MOVIE RATER

aRTIFICIAL INTELLIGENCE PROJECT

# INTRODUCTION

In this project, the concept of artificial intelligence is exemplified by a movie review rater system. It is assumed that a given review consists from the words in the previous reviews. Basically, the program executes by getting the weighted score of each word in the review and calculates the overall score of the review.

To solve this problem, for each word in the given review, the program finds all the occurrences of the word and the score of the review in the previous reviews. Then, it calculates weighted average of the scores by multiplying the score of the reviews that includes this word. By this way, we have a learned score for each word in given review.

Though we have scores of the words, the importance of the words in the review differs. Adjectives, for example, has generally much more effect on the overall score of the review. Therefore, we need to keep the category of the word and each word has effect proportional to its category. Eventually, the program gives reasonable scores to the given reviews.

# PROG RAM INTERFACE

There are 4 files in the program. *Corpus.pl* file contains reviews, *dictionary.pl* file holds the categories of the words (i.e. “huge” is an adjective and “it” is a pronoun). Weights of the categories are placed in *category.pl* file and the main prolog program is in *movieRater.pl* file.

To execute the program, there are a few options. One of them is consulting the program in SWI prolog application and executing *calculate\_scores(NewReview, S)* or *calculate\_scores2(NewReview, S)* predicatesdepending on the purpose of execution. The first method divides the weighted sum of words to the number of words. However, it is more proper to divide the weighted sum of words to the sum of weights of categories of that words which is the algorithm of second method.

Another way of running the program is directly running it on sublime text editor if prolog syntax highlighting package is installed. This time directly *main* method is called. This time, the predicates defined above should be called in the main predicate.

# PROGRAM EXECUTION

A review that has only words that are in the reviews in the corpus file is given as input and the calculated score of the review is shown as output. Since the aim of the system is straightforward, this is the only manner of user interaction through the project.

# INPUT AND OUTPUT

The input is given as the first parameter of the *calculate\_scores* predicate and the words in it are separated by 1 space as normal. The program automatically removes any commas and dots. The words of the review is lowercased for the ease of the use as well.

# PROGRAM STRUCTURE

In this section, technical details of the program is discussed. Since the program is composed of predicates, the definition of the predicates will be given in this section.

## review (Review, score)

This predicate holds the previously written reviews and their corresponding scores in *corpus.pl* file.

## word\_Category(Word)

This predicate returns true if the given word is in the given category(i.e. *adjective(“funny”)* returns true). This predicate is in *dictionary.pl* file.

## defined (Word)

This predicate returns true if the given word is in the given any category. This predicate is used for checking if the word is added to dictionary file. This predicate is in *dictionary.pl* file.

## weight (Word, Weight)

This predicate returns the weight of the word with respect to its category. This predicate is in *category.pl* file. If the type of word is unknown the given weight is 0.

## main

This predicate is called automatically when building is finished on sublime text editor. Therefore, I called *calculate\_scores* method with given new review for use prolog in sublime.

calculate\_scores(+NewReview,-S).

Calculates the score of the given review by using reviews on the carpus with the formula given in the project description. This predicate downcases the given review, splits it to words and gets weighted sum of the word scores. Finally, it divides this sum to the number of words to find the expected score of the review as in the project description.

%calculates the score of the given review by using reviews on the carpus - the proper one

%calculate\_scores(+NewReview,-S).

calculate\_scores2(NewReview, S):-

downcase\_atom(NewReview, NewReviewDowncase),

split\_string(NewReviewDowncase, " ", ".,", Words),

sumScores(Words,Weights,Sum),sumlist(Weights,TotalWeight), S is Sum / TotalWeight.

%returns the sum of scores of a word

%sumScores(Words,TotalWeight,Score)

sumScores([],[],0).

sumScores([Word|T],[Factor|Factors],K+FactorScore):-

findall([P|N],(get\_review(Words,S),count(Word,Words,N),P is S\*N),WeightedList),

calculate\_score\_from\_weighted\_list(WeightedList,Score),

weight(Word,Factor),

FactorScore is Factor \* Score,

sumScores(T,Factors,K).

%given the list of weighted scores and their occurences,returns the overall score of a word

%calculate\_score\_from\_weighted\_list(+WeightedList,-Score).

calculate\_score\_from\_weighted\_list(WeightedList,Score):-

calculate\_total\_score\_from\_weighted\_list(WeightedList,Scores,Counts),Counts>0,

Score is Scores/Counts.

%given the list of weighted scores and their occurences,returns the sum of scores and occurences of a word

%calculate\_total\_score\_from\_weighted\_list(+WeightedList,-TotalScore,-Count).

calculate\_total\_score\_from\_weighted\_list( [], 0, 0 ).

calculate\_total\_score\_from\_weighted\_list( [[Score| Count] | Tail], NewScores, NewCounts ) :-

calculate\_total\_score\_from\_weighted\_list( Tail, Scores, Counts ),

NewScores is Scores+Score,

NewCounts is Counts+Count.

%counts the number of occurences of X in the given list

%count(+Elem, +List, -NumOfOccurence).

count(\_, [], 0) :- !.

count(X, [X|T], N) :- count(X, T, N2), N is N2 + 1.

count(X, [Y|T], N) :- X \= Y, count(X, T, N).

%returns all the words on any review and its score

get\_review(Words,Score):-

review(R,Score),

downcase\_atom(R, L),

split\_string(L, " ", ".,", Words).

%returns a list of all different words on reviews that isn't put on any category, used for producing dictionary file

%not\_defined(-Words):-

not\_defined(Words):-

uniq(Uniques),

findall(Word,(defined(Word)),Defined),

subtract(Uniques,Defined,Words).

%returns a list of all different words on reviews, used for producing dictionary file

%uniq(-Uniques).

uniq(Uniques) :-

findall(Word,(get\_review(Words, \_),member(Word,Words)),Data),

sort(Data,Uniques).

# SAMPLES

Since samples are given, I didn’t put them to this section.

# IMPROVEMENTS AND EXTENSIONS

Both of the programs need to have some improvements. This section includes some of them.

One of them is the number of processors. It should be more than 1. Since it is designed to use 1 processor as master and master processor just distribute maze and manage processors, it doesn’t actually solve the maze itself. Instead make slave processors to solve their local maze parts and decide if they finished solving the maze.

Another issue that should be improved is that in each iteration all the cells are visited in loop. This may be reduced in the future by some improvements.

# DIFFICULTIES ENCOUNTERED

This section is discussed in two categories: difficulties in the first part – *1d\_mesh* and in addition to the first part issues in the trickier second part *– 2d\_mesh.*

## difficulties encountered in the first part

Since this is the first time I used multiprogramming environment, getting used to mpi is essential part of this project for me. Sending and receiving messages without causing deadlock and handling lots of errors enhances my perspective of multiprogramming concept. I ended up with the idea of ensuring the equity of sending and receiving messages; and doing communication between adjacent processors at each iteration. This was the issue concerning slave processors. Another issue was managing all the slave processors by master processor. This time I solved this by sending and receiving total number of changes at each iteration. Hence, I can find when the solution found and when to terminate the program.

## difficulties encountered in the second part

This part is more difficult than the first part. This time instead of two, we have four adjacent processors and we have corner problem here.

Handling with adjacent processors was not so difficult since it was similar to that of the first part. By adding some more constraints and cases this step was over. The crucial part of the second part was reading and printing file. In the first part those was straightforward; read N/p lines and send to a slave processor or send whole data to master processor at once. This time each line should be send separately and read back separately; send first slave’s first line, send second slave’s first line, send third slave’s first line and so on…

The last issue is handling corners. Since corner cells has adjacent cells from two different cells, those should be handled differently from others. When there are two walls around a corner, then it is similar to other cells in terms of detecting dead-ends. However, when there is just one then we need to ask both of two adjacent processors to determine if it is a dead-end. I solved this by adding some more cases and holding integers for corners to count walls around them.

# CONCLUSION

Briefly, multiprocessor environment is different from sequential one and should be thought differently. In this project, both one dimensional and two dimensional mesh is implemented and given maze is solved by distributing tasks over the processors. In the file *main\_1D.cc* the maze is solved with one dimensional mesh and communication between processors is implemented accordingly. On the other hand, *main\_2D.cc* file solves maze using two dimensional mesh and each processor can communicate with its left, right, up, and down adjacent processors. Hence, each processor handles smaller data and can communicate more processors.