Documentation for core logic

We have implemented our checker for two languages, C++ and Python. All the functions used for these are the same except the function which manages function calls and it's definitions. The C++ checker can be used approximately for C language as well. Depending on the language (C++ or Python), add the prefix, "c_" or "p_" to get the function name used in the project.

The functions:-

evaluate():

Parameters: a zip file given for the check.
Returns: an nxn matrix where "n" is the number of files in the zip file.

Description: This function first unzips the zip file using a helper function "unzip()", calculates the signature vector for each file using the helper function "find_signature()", pad these vectors and sort them using the "sort_pad" helper function, and finally calculates the **similarity percentage** between the files using the "similar" helper function and stores them in a matrix and return this matrix.

unzip():

Parameters: a zip file given for the check.
Returns: a list of the files which are extracted and returns this list.

Description: Here, we use the ZipFile function of the zipfile module and extract all the files to a directory which has the same name as the zip file. Now, we use the os module and its functions listdir, isfile and join to generate a list of the files which are present in the created directory.

find_signature(): (for C++)

Parameters: a list of files as input.
Returns: the list of word counts and the lengths of these word counts in the form of another list.

Description: For every file, we create a list of lines in the file. Then we remove the comments using the "eliminate_comments" function for better results. We merge these lines into a single string using the "merge" helper function. Then, we use the "remove_functions" to replace all the function calls with their definitions and delete their definitions in the global part. Now, after this, we create a vector of word counts for all of these files and store them. We also store the lengths of all these vectors.

eliminate_comments(): (for C++)

Parameters: a list of strings which are assumed to lines in a C++ file.
Returns: a list of strings which contains the content of the input list without the comments.

Description: We first delete the paragraph comments (starting with "/*" and ending with "*/"). For this, we iterate through every line to find a "/*" and then we check through every line starting from this line for "*/" and after we find a pair of "/*" and "*/", we delete everything in between them including these endpoints. And wherever necessary, we add a "\n" at the end of each string of the list if it was deleted by the before process to maintain the structure of the code. Now, we delete the line comments (starting with a "//"), we iterate through the updated list and if we find "//", then we delete everything that comes after it and add a "\n".

merge():

Parameters: a list of strings
Returns: a concatenated string (separated by a space) with the character “\n” replaced by a space.

Description: Self-explanatory

remove_functions(): (for C++)

Parameters: a string (this string is assumed to devoid of comments and that this is the merged string of the lines in the file)
Returns: a string, which contains all the info of the input string except the function definitions and the function calls will be replaced by these definitions.

Description: We first separate the global part and the part in "int main" because all the function definitions would be in the global part of the code and the function calls would be inside the "main". In the global part, we check for the first occurence of "{", this must correspond to a function, then we try to find the closing "}" using an inner while which takes of all the flower brackets that may be present inside the function because of for or while or if statements. Using these limits, we find the function name and isolate the function definition, store this in a dictionary and then delete this definition along with all the brackets and repeat the process until no function definition is left. After all the functions and their definitions are collected, we update the global part of the original string and replace all the function calls with their definitions

find_signature(): (for Python)

Parameters: a list of files as input.
Returns: the list of word counts and the lengths of these word counts in the form of another list.

Description: For every file, we create a list of lines in the file. Then we remove the comments using the "eliminate_comments" function for better results. Then, we use the "edit_functions" helper function to delete their definitions in the file and then we replace all the function calls with their definitions. We merge these lines into a single string using the “merge” helper

function. Now, after this, we create a vector of word counts for all of these files and store them. We also store the lengths of all these vectors

eliminate_comments(): (for Python)

Parameters: a list of strings which are assumed to lines in a Python file.
Returns: a list of strings which contains the content of the input list without the comments.

Description: We delete the line comments (starting with a "#"), we iterate through the updated list and if we find "#", then we delete everything that comes after it and add a "\n".

edit_functions(): (for Python)

Parameters: a list of strings as input which is assumed to be lines in a python file.
Returns: a list of strings which contains all the content of the input with the function definitions removed and the dictionary of functions as another list.

Description: We first find the occurence of "def" in in each line, if present, we calculate the index of the first character in the next line, which indicates the indentation level of each line in the function which is important for python language and based on this level, collects the function definition and stores it and deletes it in the input list.

sort_pad():

Parameters: two lists as inputs, the first one is assumed to be the lengths of word count vectors and the other is the word count vector itself.
Returns: maximum length and the updated word count vectors in the form of another list.

Description: We first find the maximum length using the lengths list and increase the length of each word count vector to it by padding zeros. (we can pad zeros because it just implies that a word foreign to a file is not there in the file, which is true and doesn't disturb our estimations). Then we sort these vectors to get a distribution of words and their counts which is uniform over all files (increasing or decreasing).

similar():

Parameters: a list of lists as input which is assumed to be the word count vectors.
Returns: a symmetric matrix containing all the similarity percentages.

Descriptions: For every pair of vectors, we calculate the norm of the difference of those vectors and divide by the maximum norm of the vectors. This will always be less than 1 because, as the vectors are assumed to be sorted, any two vectors will have an acute angle in between them (because all of them will be on the same side of the hyperplane x1=x2=...=xn for an n-dimensional space of vectors) and the maximum value of the difference is the maximum of norm of the two vectors at best (equality when a vector

consists of all zeros). Now, this can be used as an estimate for similarity because similar files will be "closer" to each other. For the percentage similarity, we subtract this value from 1 and multiply by 100.

csv_write():

Parameters: a numpy array which is assumed to be the symmetric matrix containing all the similarity percentages. i.e a matrix returned from the “similar()” function.
Returns: a csv file containing similarity percentages of files which are in the zipfile.

Descriptions: First we create a csv file named REDPLAG.csv in the working directory.By using os.listdir(), we first obtain the list of files in the zipfile. Using this list, we now create a header row for the csv file. We then modify the matrix containing similarity percentages, by adding an extra column ,containing names of respective files in the zipfile, using hstack() function. Now, using iterators we add rows to the created csv file.

plots():

Parameters: a numpy array which is assumed to be the symmetric matrix containing all the similarity percentages. i.e a matrix returned from the “similar()” function.
Returns: a heatmap which represents the similarity percentages of files which are in the zipfile.

Descriptions: A usual procedure of plotting a heatmap is followed, where limits have been set to the colormap (between 0 to 100 including both 0 and 100).For high percentage similarity white color is assigned , whereas for moderate percentage similarity yellowish - orange is assigned, and for low percentage similarity black color is assigned.And percentage similarity of files can be seen in respective cells of the heatmap.And this heatmap is saved, as REDPLAG.png, into the working directory using savefig() function