This file includes pseudo code of:

- ModelTraining: The workflow to train the embedding, HyLSTM, and Multi_HyLSTM.
- Multi_HyLSTM: The algorithm of how a trained Multi_HyLSTM model produces the recommended API based on the multi-path input.
- MultiPathSelection: The multi-path selection algorithm to obtain the input of the Multi_HyLSTM.
- PickAPath: The algorithm used in MultiPathSelection to traverse a path.

Algorithm 1 ModelTraining(D): The workflow of training the embedding, HyLSTM and Multi_HyLSTM

Input: D, where D is a dataset of the API dependence graphs. Each graph G has a slicing criterion $G.sc$
Output: θ, where θ is the parameters of the trained Multi_HyLSTM
$P \leftarrow$ extract paths from D
$\theta_{emb} \leftarrow$ train $word2vec(P)$ //θ_{emb} is the parameters of the trained embedding layer.
$\theta_{HyLSTM} \leftarrow$ train $HyLSTM(P)$ //θ_{HyLSTM} is the parameters of the trained HyLSTM model
$I \leftarrow \emptyset$
$L \leftarrow \emptyset$
for Graph G in D **do**
    $label \leftarrow G.sc$
    $(P_1, \ldots, P_n) \leftarrow$ MultiPathSection$(G, label)$
    $C.add((P_1, \ldots, P_n))$
    $L.add(label)$
end for
//Initialize Multi_HyLSTM with $\theta_{emb}, \theta_{HyLSTM}$
$\theta_{pretrain} \leftarrow (\theta_{emb}, \theta_{HyLSTM})$
//Start training
for $(P_1, \ldots, P_n)$ in I, $label$ in L **do**
    $output \leftarrow multi\_HyLSTM(P_1, \ldots, P_n, \theta_{pretrain})$
    $\theta \leftarrow backward\_propagation(output, label)$
end for
return θ

Algorithm 2 Multi_HyLSTM($P_1, P_2, \ldots, P_n, \theta$): The API recommendation process that accepts multiple data-flow paths and produces the recommended API

Input: $(P_1, P_2, \ldots, P_n, \theta)$, where P_i is a data-flow path, θ represents the parameters of a trained Multi_HyLSTM model
Output: R, where R is the recommended API
$emb \leftarrow \theta$ //Initialize embedding layer
$HyLSTM \leftarrow \theta$ //Initialize HyLSTM network
for each path P_i **do**
    for each API/constant t_{ij} in P_i **do**
        $v_{ij} \leftarrow emb[t_{ij}]$.
    end for
    $output_i \leftarrow HyLSTM(v_{i1}, v_{i2}, \ldots, v_{in})$
end for
$output_{aggre} \leftarrow Ave\_pooling(output_1, output_2, \ldots, output_n)$//$Ave\_pooling$ is a standard neural network layer
$R \leftarrow Softmax(output_{aggre})$ //$Softmax$ is a standard layer for multi-class classification
return R

Algorithm 3 MultiPathSelection(n, G, sc): An important building block of ModelTraining for identifying n data-flow paths originating from the slicing criteria, with the constraint of being as non-overlapping as possible

Input: (n, G, sc), where n is the path budget and G is an API dependence graph. G includes a set of nodes $G.nodes$. A node N contains a code statement $N.code$, the control dependence $N.cd$ of this code statement. sc is a node in G representing the slicing criterion.
Output: C, where C includes i data-flow paths ($i \leq n$).
path $\leftarrow \emptyset$
path.append(sc)
Q $\leftarrow \emptyset$
Q.enqueue(sc)
while $Q \neq \emptyset$ **do**
    $curr\_node \leftarrow Q$.dequeue()
    $D_{control} \leftarrow \emptyset$
    for Node *node* in $curr\_node$.predecessor() **do**
        $D_{control}$.add(node.cd)
    end for
    for each cd in $D_{control}$ **do**
        for Node *node* in $curr\_node$.predecessor() **do**
            if *node*.cd == cd **then**
                Q.enqueue(*node*)
                path.append(*node*)
                new_path = PickAPath($path$, G)
                C.add(new_path)
                if C.length == n **then**
                    return
                end if
            end if
        end for
    end for
end while

Algorithm 4 PickAPath(*path*, *G*): Traverse a data-flow path by random walk in an API dependence graph given several beginning nodes of the path

Input: (*path*, *G*), where *path* is an incomplete path only including several beginning nodes. *G* is an API dependence graph.
Output: *new_path*
new_path ← ∅
curr_node ← last node in *path*
while *curr_node.predecessors*() ≠ ∅ **do**
    next_node ← randomly pick a predecessor
    new_path.append(*next_node*)
    curr_node ← *next_node*
end while
return *new_path*