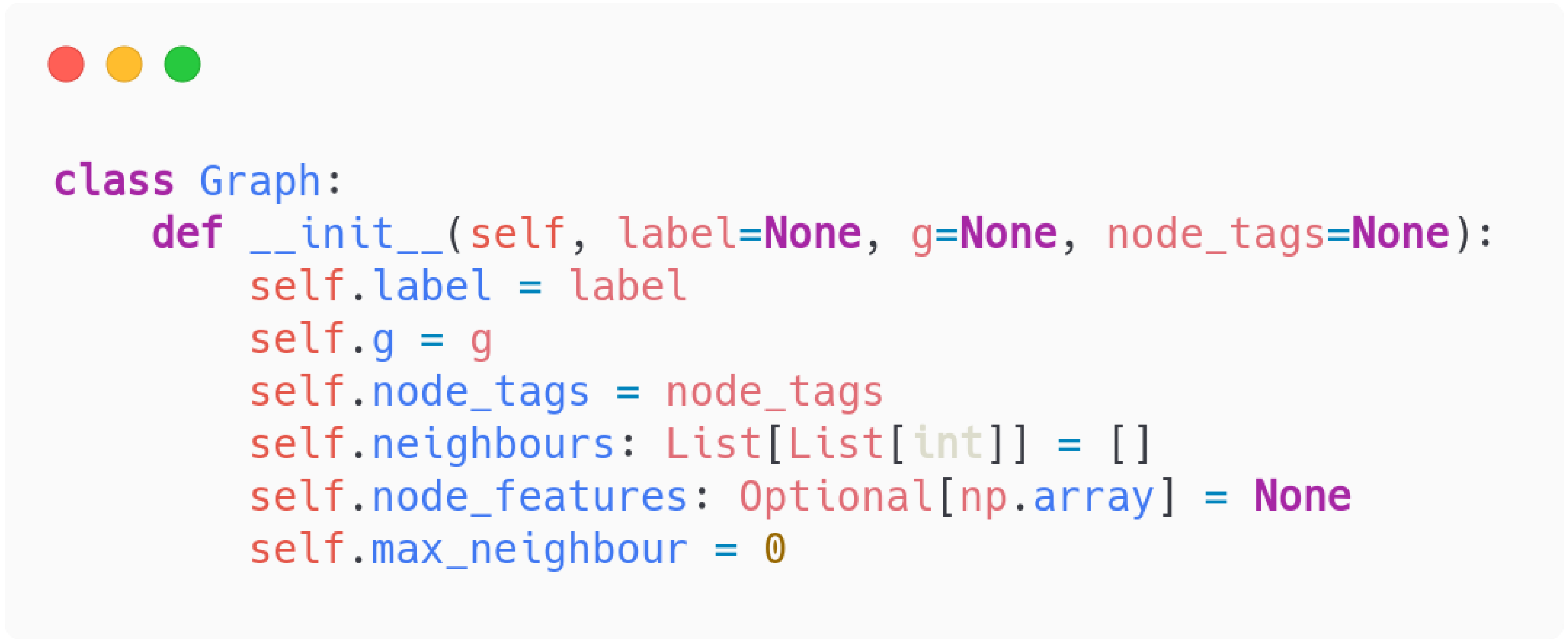
UNDERSTANDING THE POWER OF GRAPH NEURAL NETWORKS

Team 30: AZAP

TABLE OF CONTENTS

AIM

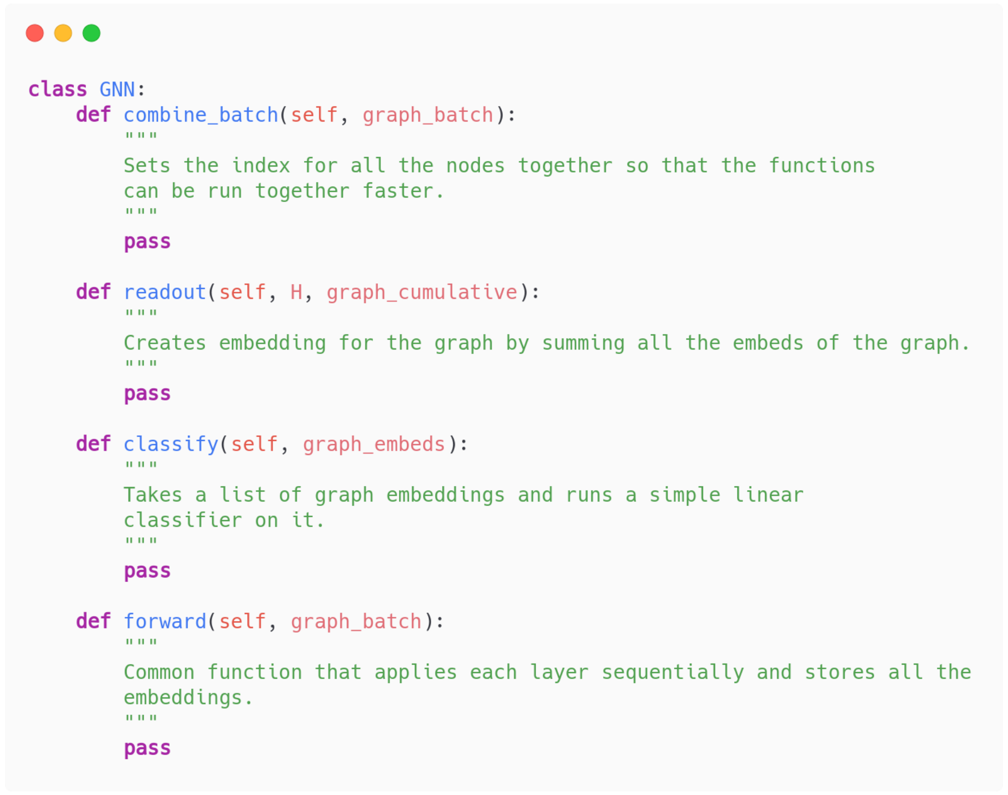
The aim of this project is to implement the different variants of the Graph Neural Networks (primarily GIN), compare it with WL test results and support the results with the help of a theoretical explanation supported by the arguments made by the paper.



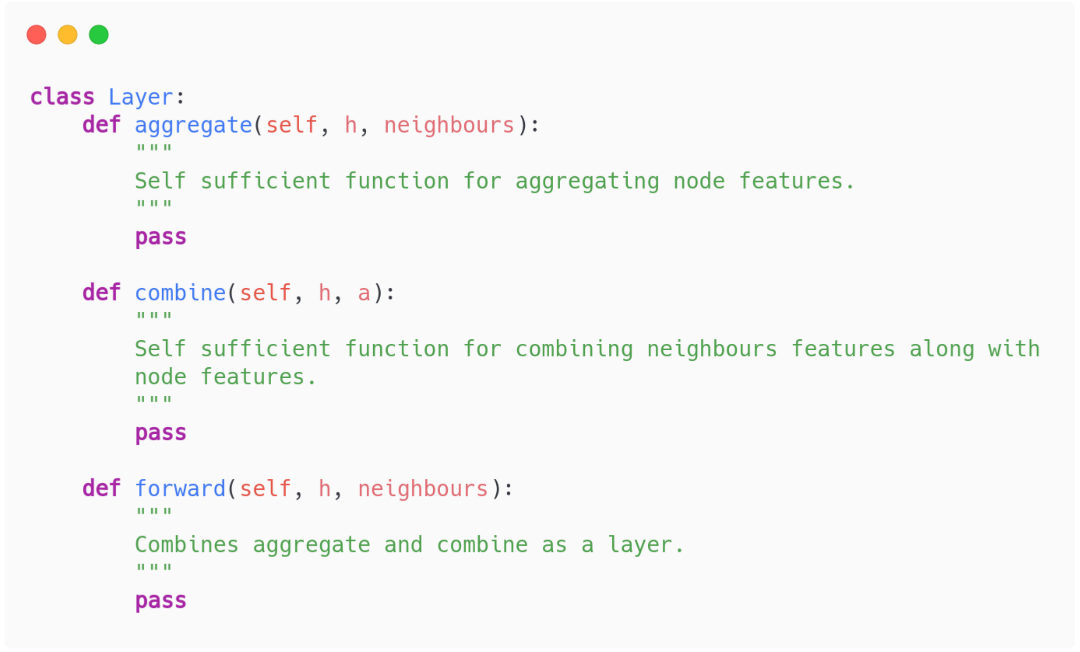
```
class Graph:
    def __init__(self, label=None, g=None, node_tags=None):
        self.label = label
        self.g = g
        self.node_tags = node_tags
        self.neighbours: List[List[int]] = []
        self.node_features: Optional[np.array] = None
        self.max_neighbour = 0
```

DATA CLEANING

Datasets are cleaned and are stored as Graph objects, using NetworkX library



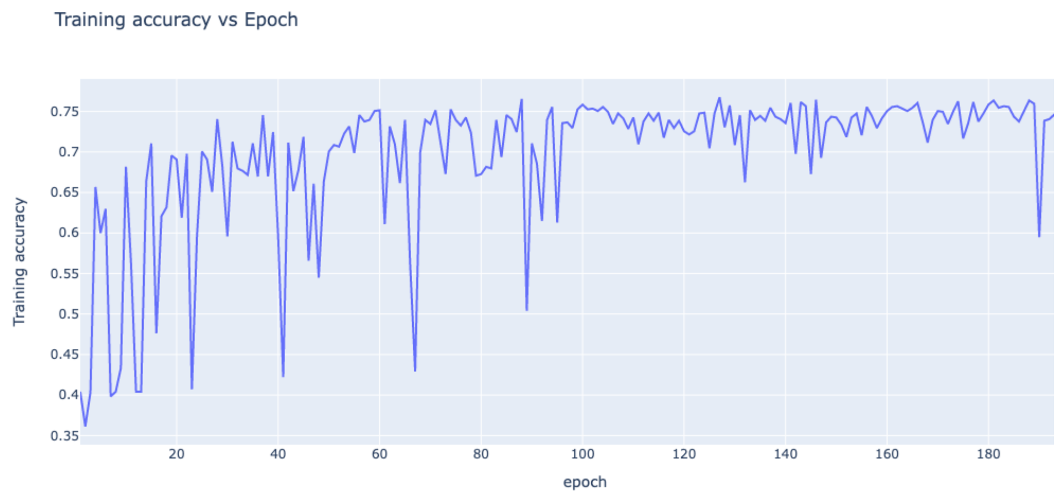
```python
class GNN:
    def combine_batch(self, graph_batch):
        """
        Sets the index for all the nodes together so that the functions
        can be run together faster.
        """
        pass

    def readout(self, H, graph_cumulative):
        """
        Creates embedding for the graph by summing all the embeds of the graph.
        """
        pass

    def classify(self, graph_embeds):
        """
        Takes a list of graph embeddings and runs a simple linear
        classifier on it.
        """
        pass

    def forward(self, graph_batch):
        """
        Common function that applies each layer sequentially and stores all the
        embeddings.
        """
        pass
```

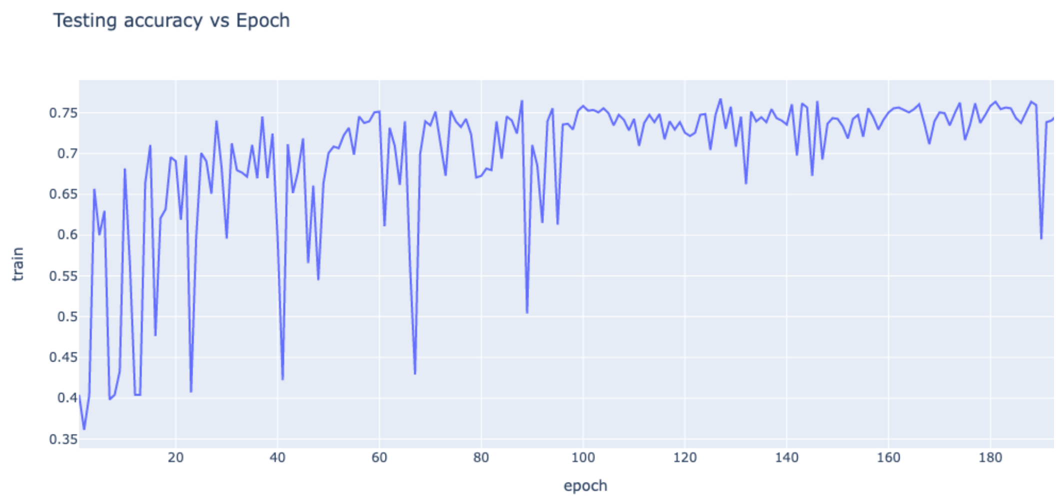


```python
class Layer:
    def aggregate(self, h, neighbours):
        """
        Self sufficient function for aggregating node features.
        """
        pass

    def combine(self, h, a):
        """
        Self sufficient function for combining neighbours features along with
        node features.
        """
        pass

    def forward(self, h, neighbours):
        """
        Combines aggregate and combine as a layer.
        """
        pass
```

CODE STRUCTURE

Ready to create any form aggregate, combine, or readout functions using inheritance.



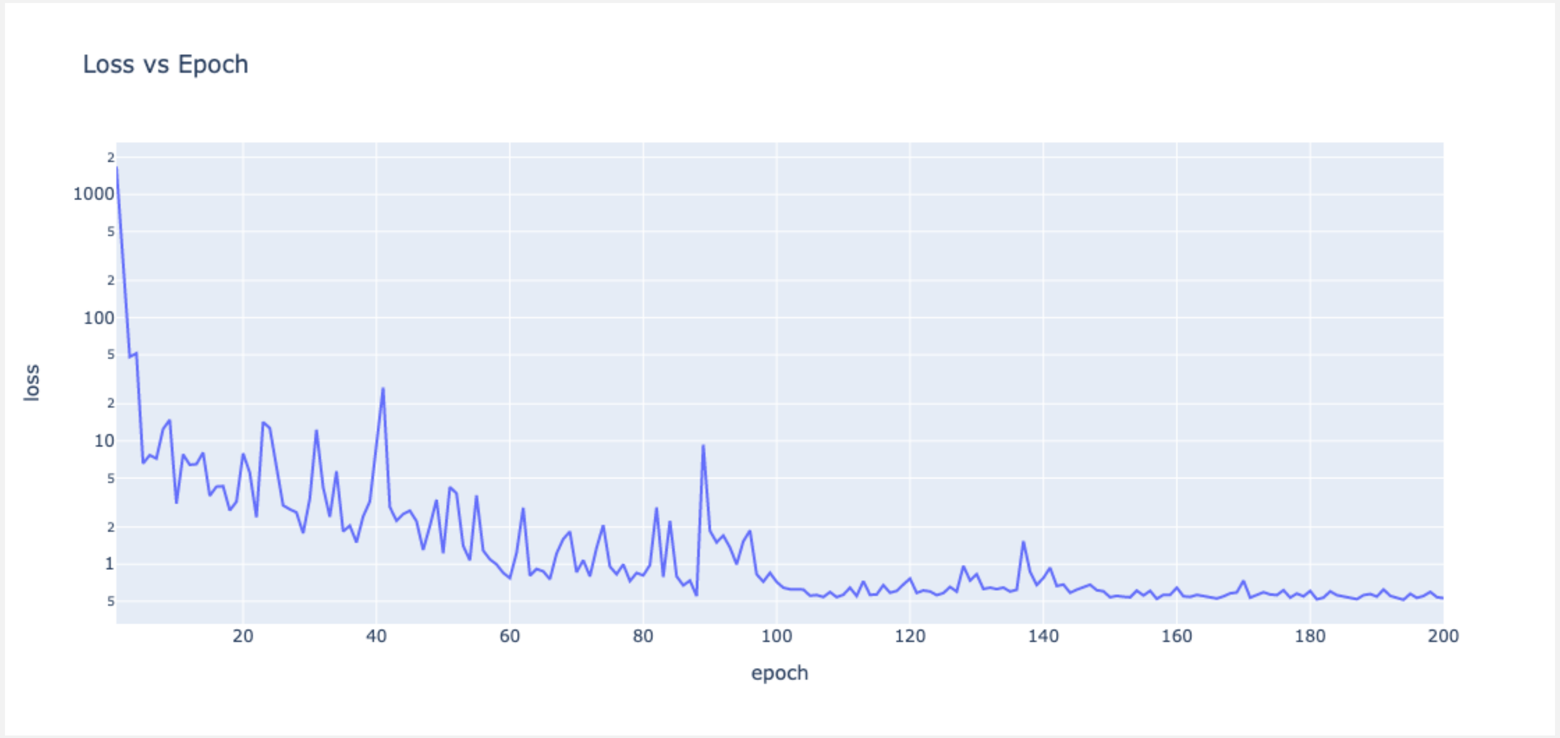


PROTEINS DATA

75% accuracy using max pool and one layer classifier.

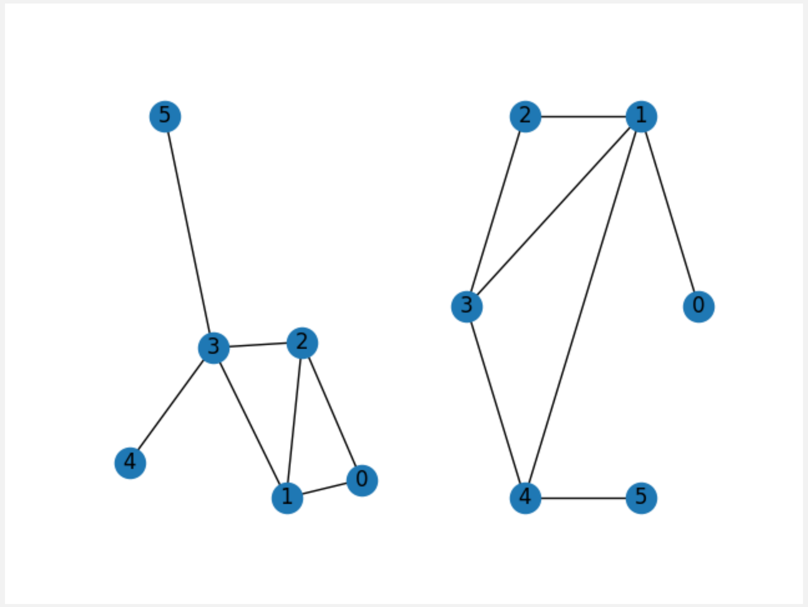
REPLICATED GRAPH SAGE LOSS VS EPOCH

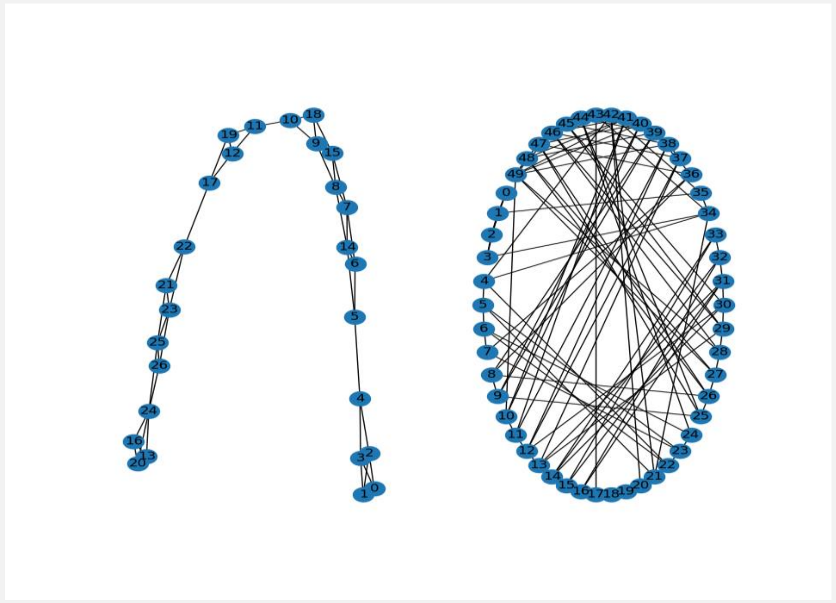
- In each epoch we train over the first fold of the 10 folds of cross validation
- The average loss starts high at 1681 for the first epoch then quickly falls to less than 10 over the next few epochs



PROGRESS OVERVIEW: WL TEST

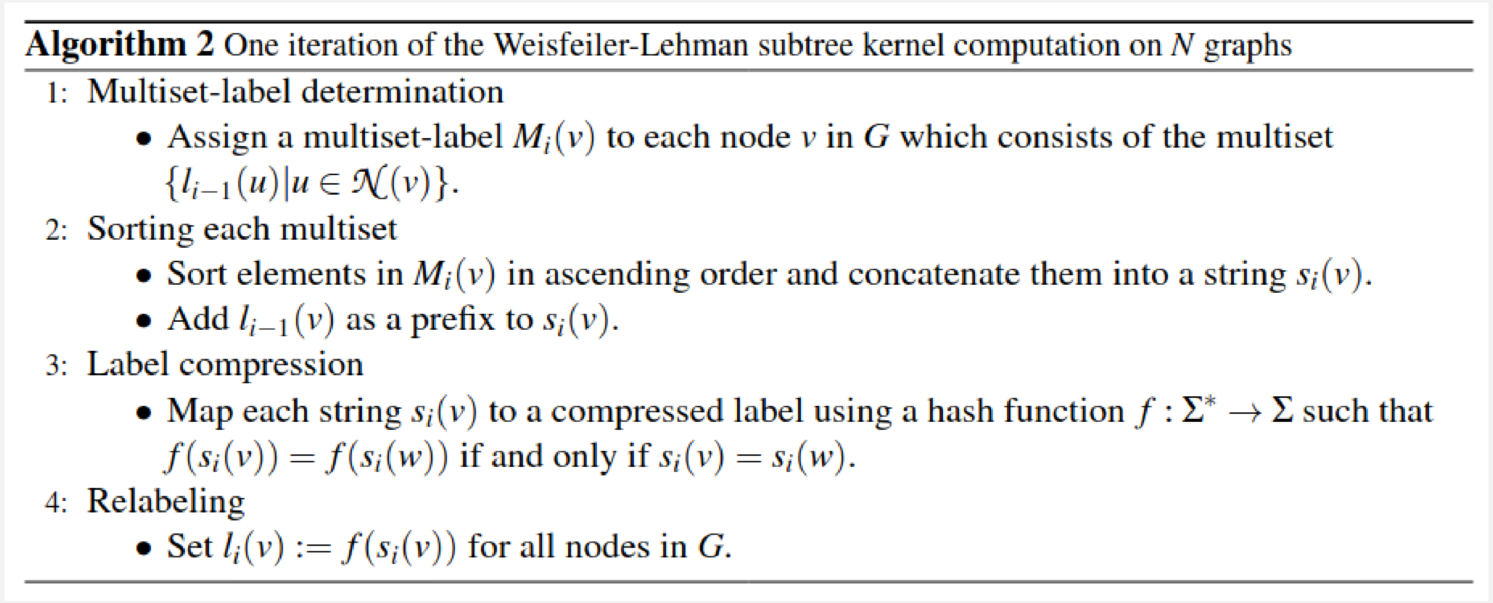
- Implemented the binary 1D-WL Test.
- Tested over a set of non-regular graphs.





PROGRESS OVERVIEW: WL TEST

- Implemented the WL subtree kernel.
- Tested over PROTEINS dataset.

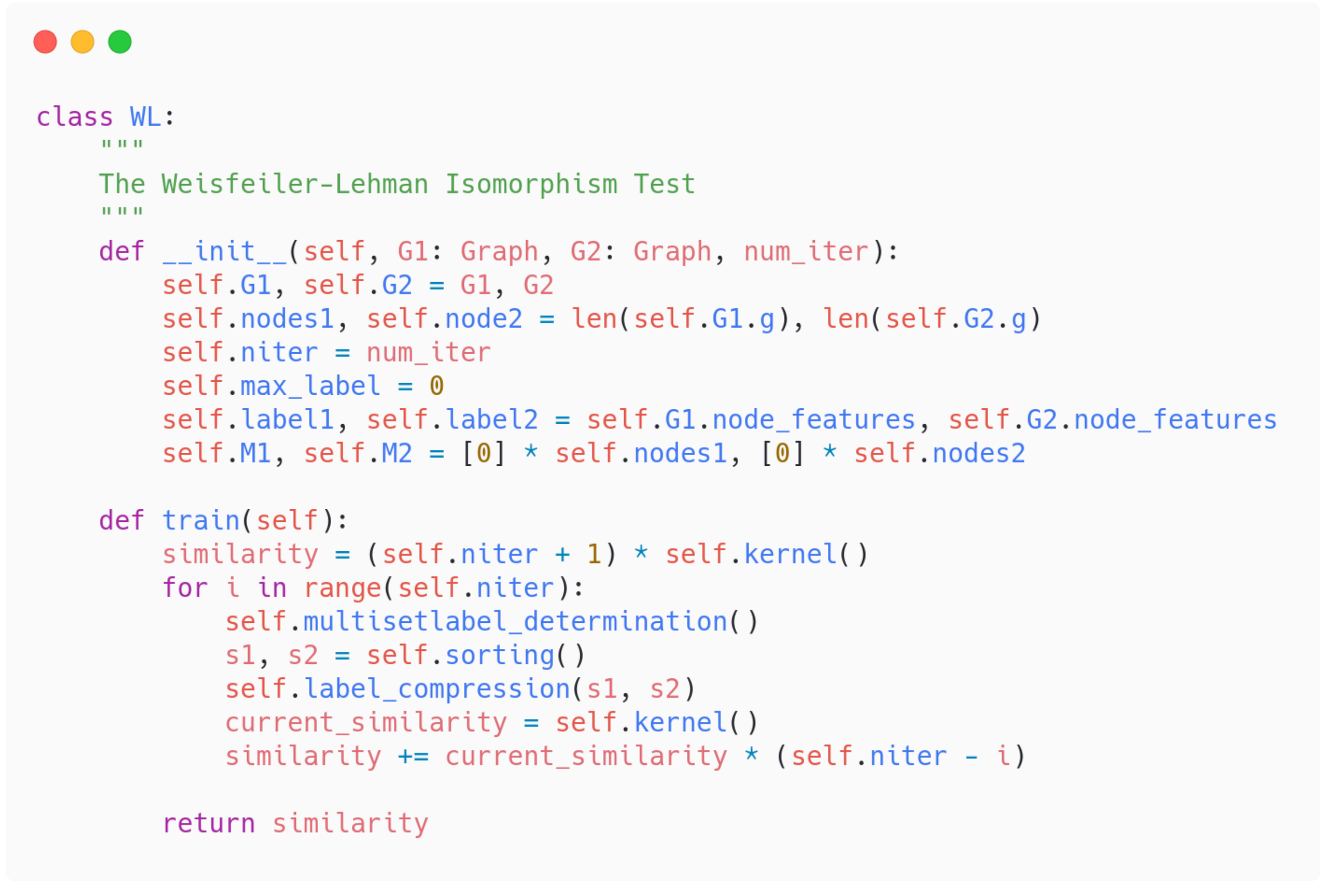


Algorithm 2 One iteration of the Weisfeiler-Lehman subtree kernel computation on N graphs

```
1: Multiset-label determination
       • Assign a multiset-label $M_i(v)$ to each node $v$ in $G$ which consists of the multiset
         $\{l_{i-1}(u) | u \in \mathcal{N}(v)\}$.
2: Sorting each multiset
       • Sort elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$.
       • Add $l_{i-1}(v)$ as a prefix to $s_i(v)$.
3: Label compression
       • Map each string $s_i(v)$ to a compressed label using a hash function $f : \Sigma^* \to \Sigma$ such that
         $f(s_i(v)) = f(s_i(w))$ if and only if $s_i(v) = s_i(w)$.
4: Relabeling
       • Set $l_i(v) := f(s_i(v))$ for all nodes in $G$.
```

STRUCTURE OF WL SUBTREE KERNEL

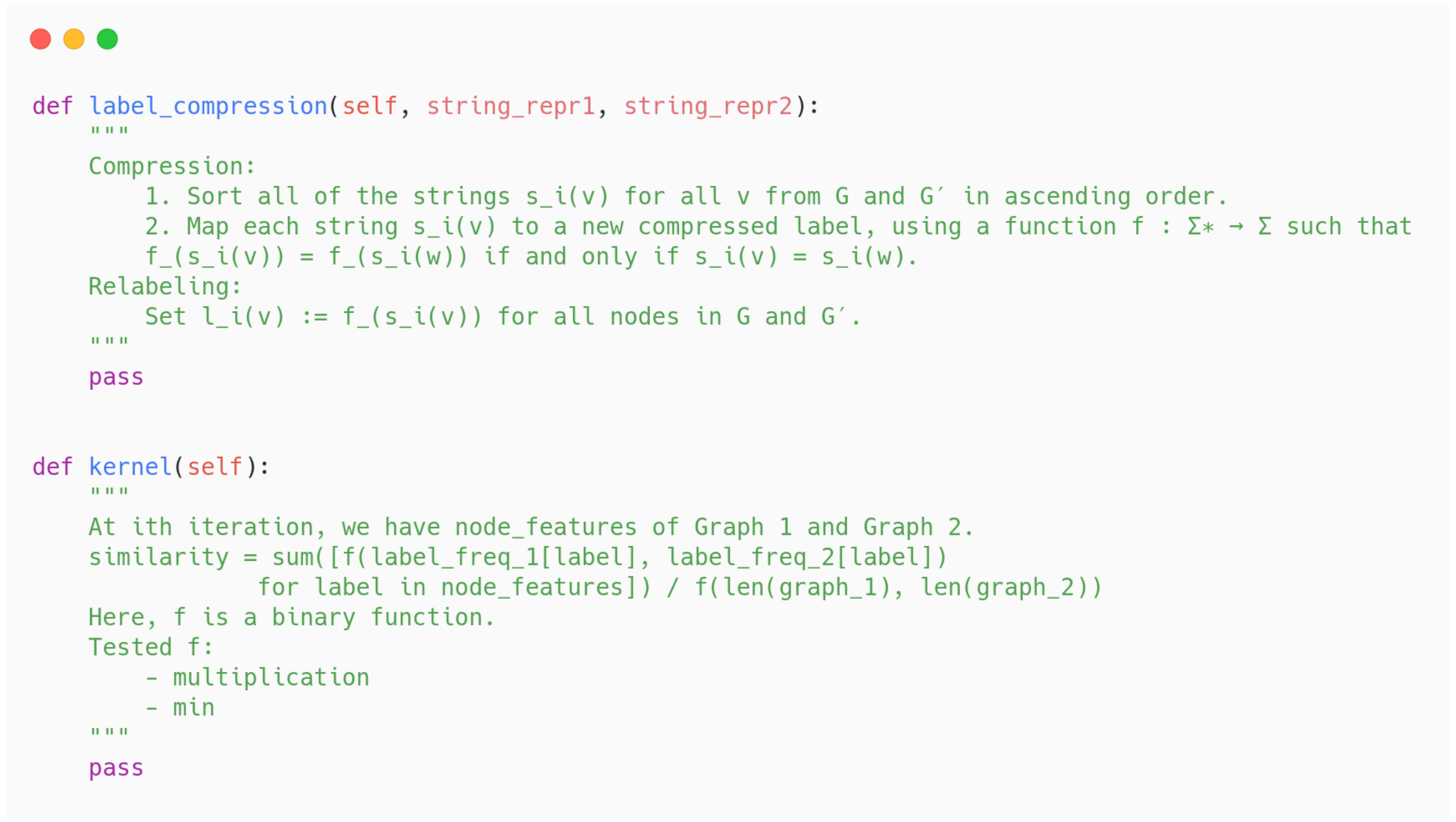
Definition and training



```python
class WL:
    """
    The Weisfeiler-Lehman Isomorphism Test
    """
    def __init__(self, G1: Graph, G2: Graph, num_iter):
        self.G1, self.G2 = G1, G2
        self.nodes1, self.node2 = len(self.G1.g), len(self.G2.g)
        self.niter = num_iter
        self.max_label = 0
        self.label1, self.label2 = self.G1.node_features, self.G2.node_features
        self.M1, self.M2 = [0] * self.nodes1, [0] * self.nodes2

    def train(self):
        similarity = (self.niter + 1) * self.kernel()
        for i in range(self.niter):
            self.multisetlabel_determination()
            s1, s2 = self.sorting()
            self.label_compression(s1, s2)
            current_similarity = self.kernel()
            similarity += current_similarity * (self.niter - i)

        return similarity
```

STRUCTURE OF WL SUBTREE KERNEL

Label Compression and kernel

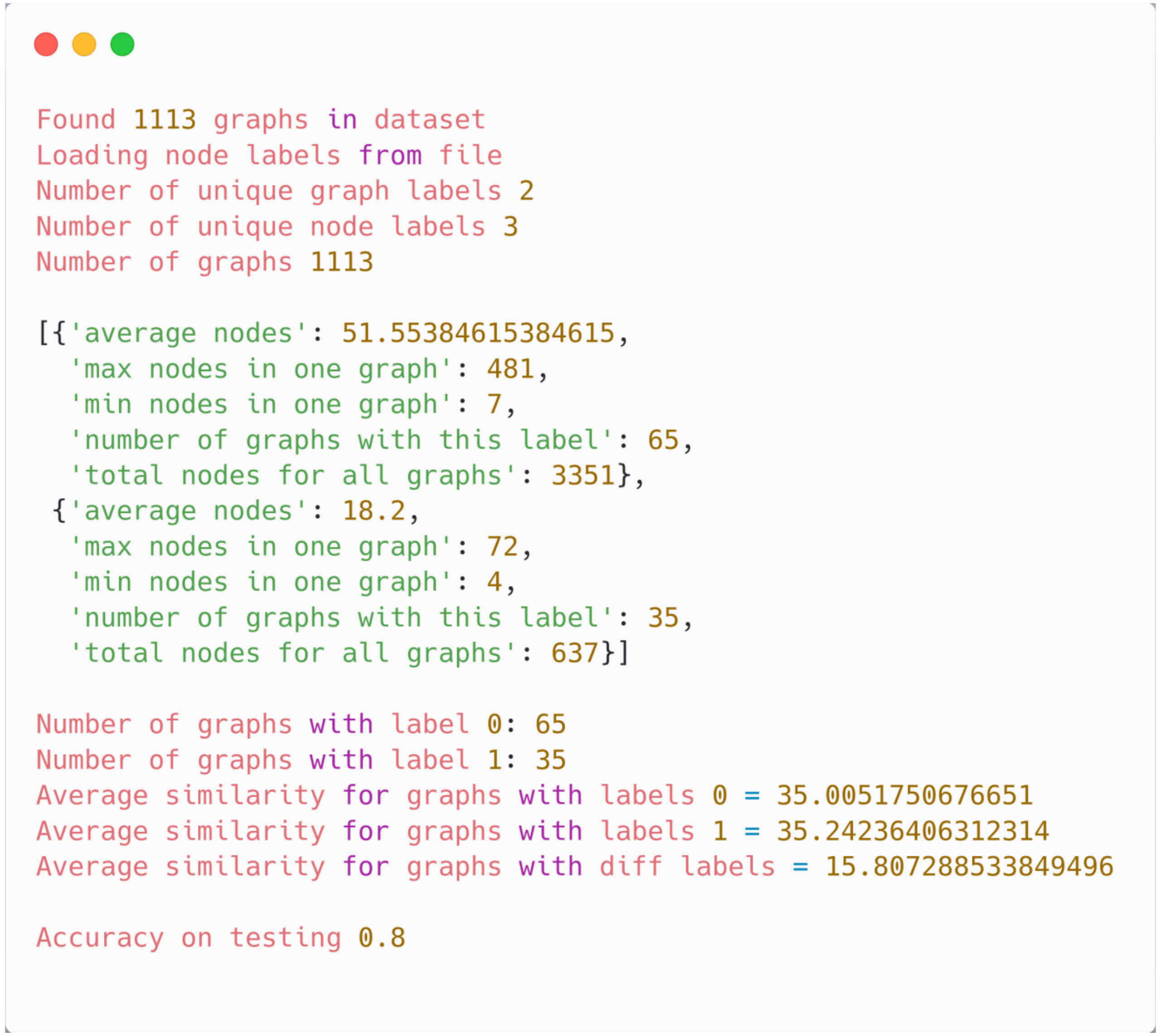


```
def label_compression(self, string_repr1, string_repr2):
    """
    Compression:
        1. Sort all of the strings s_i(v) for all v from G and G′ in ascending order.
        2. Map each string s_i(v) to a new compressed label, using a function f : Σ∗ → Σ such that
        f_(s_i(v)) = f_(s_i(w)) if and only if s_i(v) = s_i(w).
    Relabeling:
        Set l_i(v) := f_(s_i(v)) for all nodes in G and G′.
    """
    pass


def kernel(self):
    """
    At ith iteration, we have node_features of Graph 1 and Graph 2.
    similarity = sum([f(label_freq_1[label], label_freq_2[label])
                for label in node_features]) / f(len(graph_1), len(graph_2))
    Here, f is a binary function.
    Tested f:
        - multiplication
        - min
    """
    pass
```

RESULTS OF WL SUBTREE KERNEL

Accuracy in the given case: 80%

Note: WL currently is prepared only to be tested for PROTEINS



```
Found 1113 graphs in dataset
Loading node labels from file
Number of unique graph labels 2
Number of unique node labels 3
Number of graphs 1113

[{'average nodes': 51.55384615384615,
  'max nodes in one graph': 481,
  'min nodes in one graph': 7,
  'number of graphs with this label': 65,
  'total nodes for all graphs': 3351},
 {'average nodes': 18.2,
  'max nodes in one graph': 72,
  'min nodes in one graph': 4,
  'number of graphs with this label': 35,
  'total nodes for all graphs': 637}]

Number of graphs with label 0: 65
Number of graphs with label 1: 35
Average similarity for graphs with labels 0 = 35.0051750676651
Average similarity for graphs with labels 1 = 35.24236406312314
Average similarity for graphs with diff labels = 15.807288533849496

Accuracy on testing 0.8
```

WHAT'S NEXT?

GNN	Try different aggregate functions on the GNN. Add small features mentioned in the paper such as dropout.
GIN	Connect MLP with GNN to make GIN and run tests on all the datasets mentioned in the paper.
SVM	To use WL test along with SVM classification to create a graph classifier as described in the paper.
Report	Compile all the results into a report.

REFERENCES

- How Powerful are Graph Neural Networks, Xu et al
- Inductive Representation Learning on Large Graphs, Hamilton et al
- Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. Journal of Machine Learning Research, 12(Sep): 2539–2561, 2011.