Using Graph Neural Networks for coding assignment plagiarism detection based on source

Tim Peko (TimerErTim)

Neural networks seem like a perfect fit for detecting plagiarism attempts in coding exercises. In the Rust deep learning framework burn-rs, I built a graph convolution network with subpar accuracy, After a quick recap about that, we will explore how graph isomorphism networks and different similarity metrics can be used to improve the accuracy of the model.

Table of Contents

The finished project is available under

1 What has already happened?

The project was started by the simple idea of “How much can I overengineer plagiarism detection?”, an open ended task given to my class by the professor of the Python scripting course. The solution can be as simple as comparing different tree similarity metrics, finding optimal metric weights for best accuracy, and submitting that as results.

Or we apply a more complex machine learning approach, using a graph convolution network to learn the similarity of the source code’s abstract syntax trees.

1.1 Basic conditions

chosen language

Given:

- Supported language: Either Python or **C++**
- GUI of some form
- Comparing two files ⟶ Plagiarism chance percentage ($p \in [0, 1]$).

Assumptions:

- The programming language can reliably be inferred from the file extension
- Every assignment submission is a single file ⟶ multiple files per comparison side is not supported

1.2 Used dataset

We require some ground truth for training, and turns out high quality coding assignment plagiarism datasets are a surprisingly rare occurrence in the wild. I settled on the **Programming Homework Dataset for Plagiarism Detection | IEEE DataPort**, which is a C and C++ dataset with 328 authentic and 120 plagiarized assignment submission pairs.

1.3 High-level approach

Figure 1 shows the general idea behind comparing the different files. We employ a graph compression step, Figure 2, to extract most relevant features from the ASTs and then compare these feature maps with a distance gated exponential similarity output layer. As the graph is further compressed, nodes features are narrowed but deepened, until only a few or a single node with many aggregated features remains (similar to visual CNNs).

For simplicity, layer normalization and dropout are not shown in the figure.

1.4 Applied concepts

You can skip this chapter if you are not interested in the detailed workings of Section 1.3.

1.4.1 Abstract Syntax Trees

With **Tree-sitter** we can parse the source code into an abstract syntax tree (AST). Because it supports many different languages, we can easily extend the same approach to a suitable Python dataset[1].

[1] If found in the future

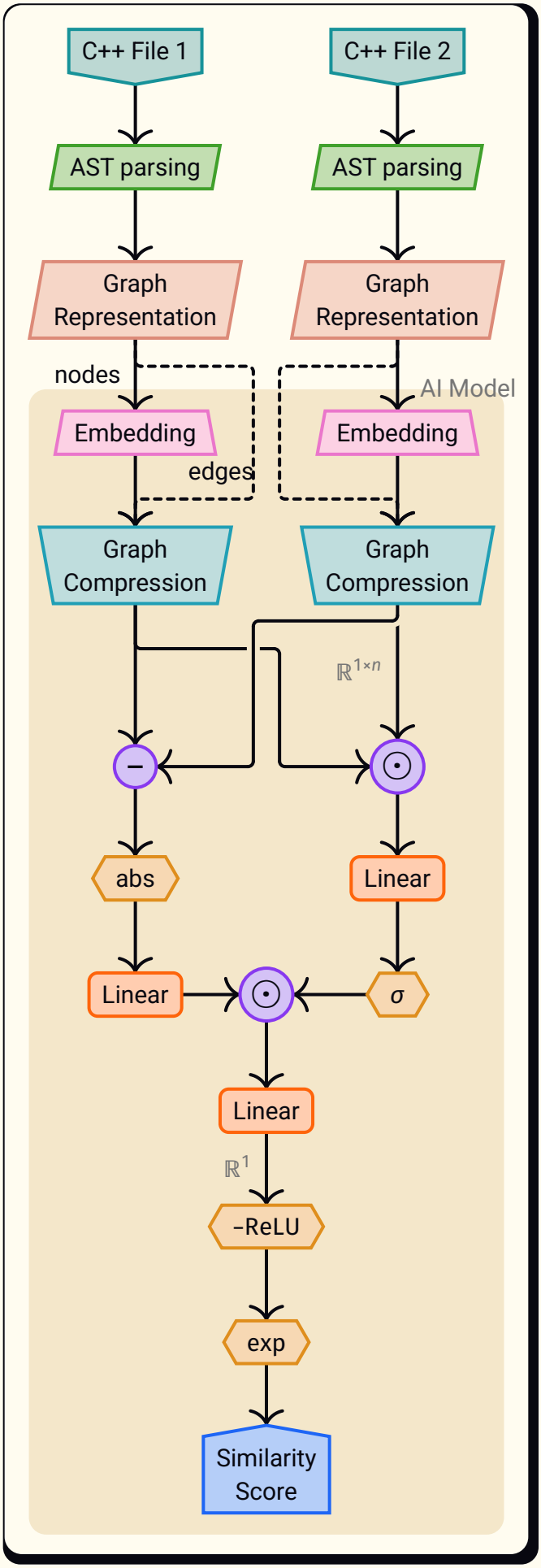


Figure 1: Initial high-level processing pipeline

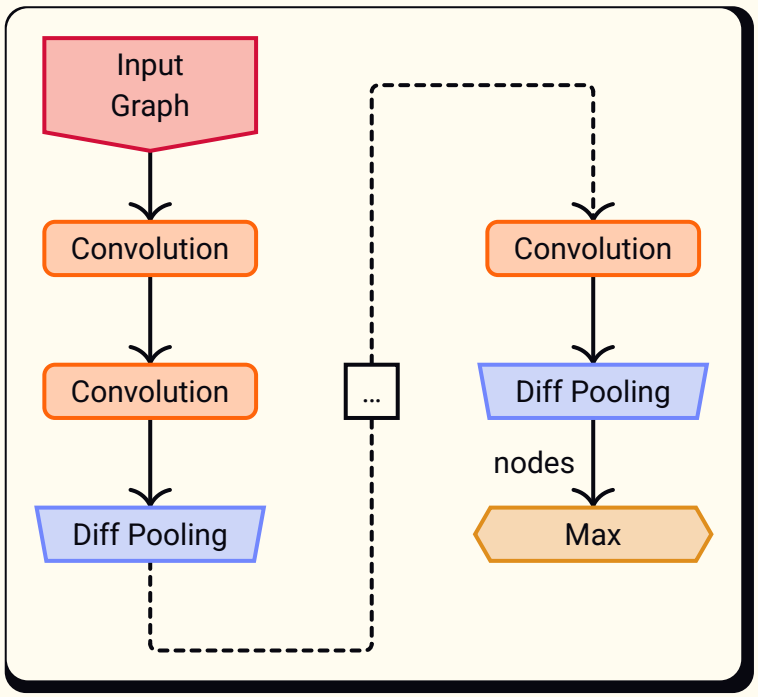


Figure 2: Graph compression process

By having a tree representation of the source code, we have a starting point for extracting features from the semantic relationships between the nodes. See the example in Figure 3, with the corresponding node labels in Table 1.

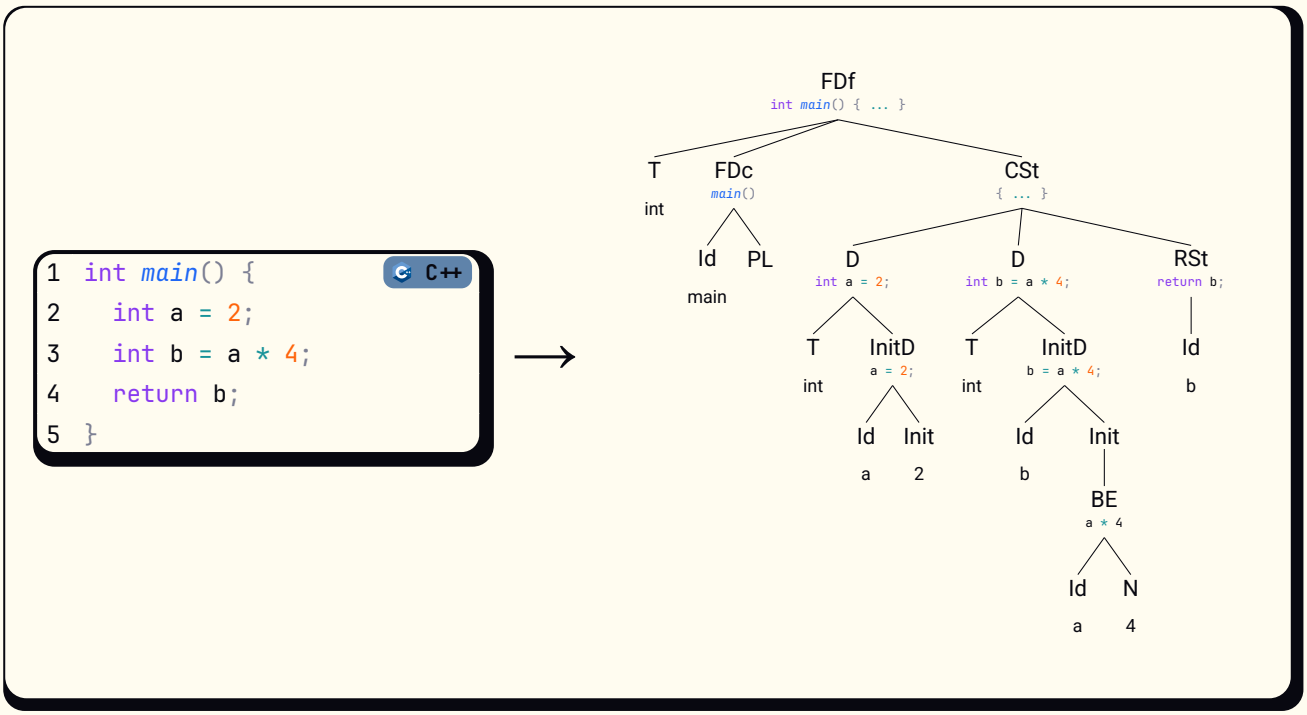


Figure 3: AST representation of a simple C++ program

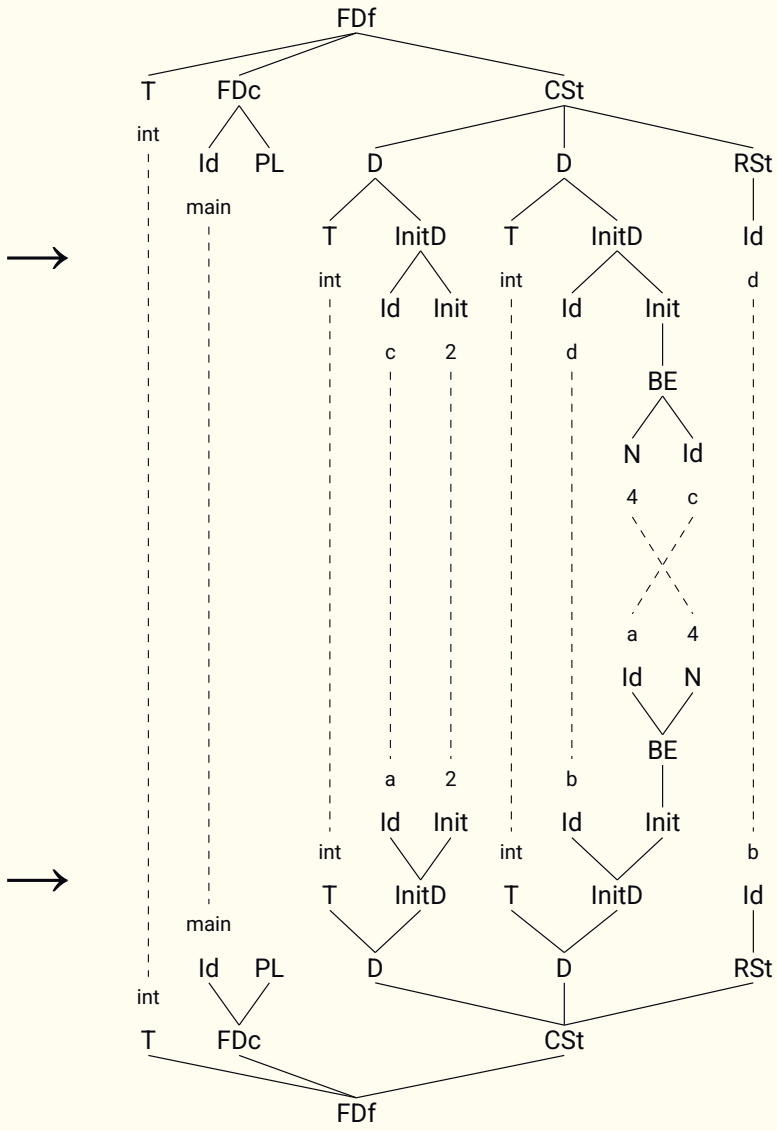
Abbreviation	Full
FDf	FunctionDefinition
T	Type
FDc	FunctionDeclarator
CSt	CompoundStatement
D	Declaration
InitD	InitDeclarator
Init	Initialiser
BE	BinaryExpression
N	Number
RSt	ReturnStatement
Id	Identifier
PL	ParameterList

Table 1: abbreviated AST node labels

AST Advantages

AST representation is agnostic of symbolic names, it only cares about the structure of the code, which is harder to cover up when plagiarizing.

```cpp
int main() {
  int a = 2;
  int b = a * 4;
  return b;
}
```




```cpp
int main() {
  int c = 2;
  int d = 4 * c;
  return d;
}
```

The above code samples lead to structurally equivalent ASTs, as indicated by the correspondents of the dashed lines.

1.4.2 Node embeddings

Raw node labels and ids in the range of 0-65535 are barely useful as input features for the neural network. An embedding layer is used to transform the ids into 16-dimensional vectors.

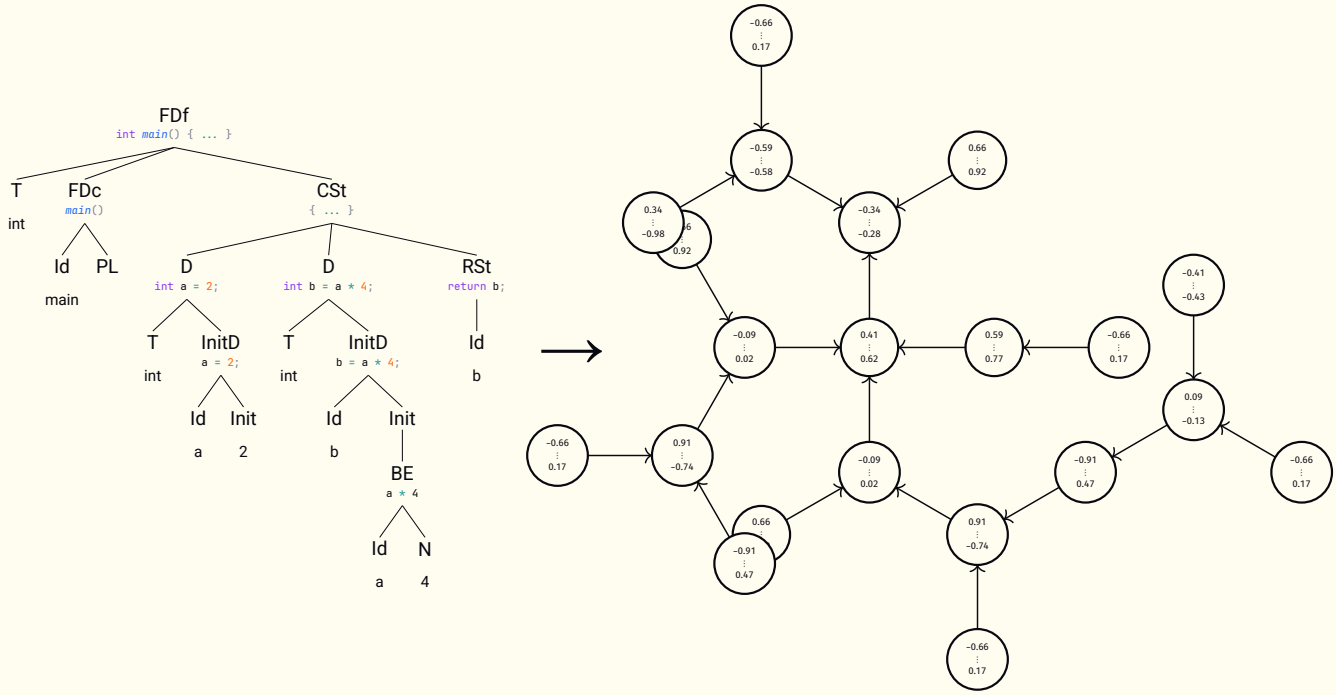
burn-rs node embedding implementation

```rust
impl<B: Backend> PlagiarismDecider<B> {
  pub fn embed_nodes(&self, nodes: Tensor<B, 2, Int> [n, 1]) -> Tensor<B, 2> {
```

```
        let embedding_layer = EmbeddingConfig::new(self.num_classes = 65536,
3       self.embedding_size = 16).init(&self.device);
4       embedding_layer
5               .forward(nodes.clone().swap_dims(0, 1) [1, n])
6               · squeeze_dim(0) [1, n, 16]
7   }
8 }
```

Stable Node IDs

This only works because node types have stable associated ids across the same Tree-sitter and language spec version. A `Type` node will always have the same id and thus embedding vector.



1.4.3 Graph Convolution Networks

Graphs are represented as a simple struct containg node features of size $n \times d$ and adjacency matrix of size $n \times n$ where n is the number of nodes and d is the number of features per node.

```rust
#[derive(Debug, Clone)]
pub struct Graph<B: Backend, D: TensorKind<B> = Float> {
    /// Shape: [num_nodes, num_features]
    pub nodes: Tensor<B, 2, D>,
    /// Shape: [num_nodes, num_nodes]
    /// Adjacency matrix, [target, src]
    pub edges: Tensor<B, 2, Float>,
}
```

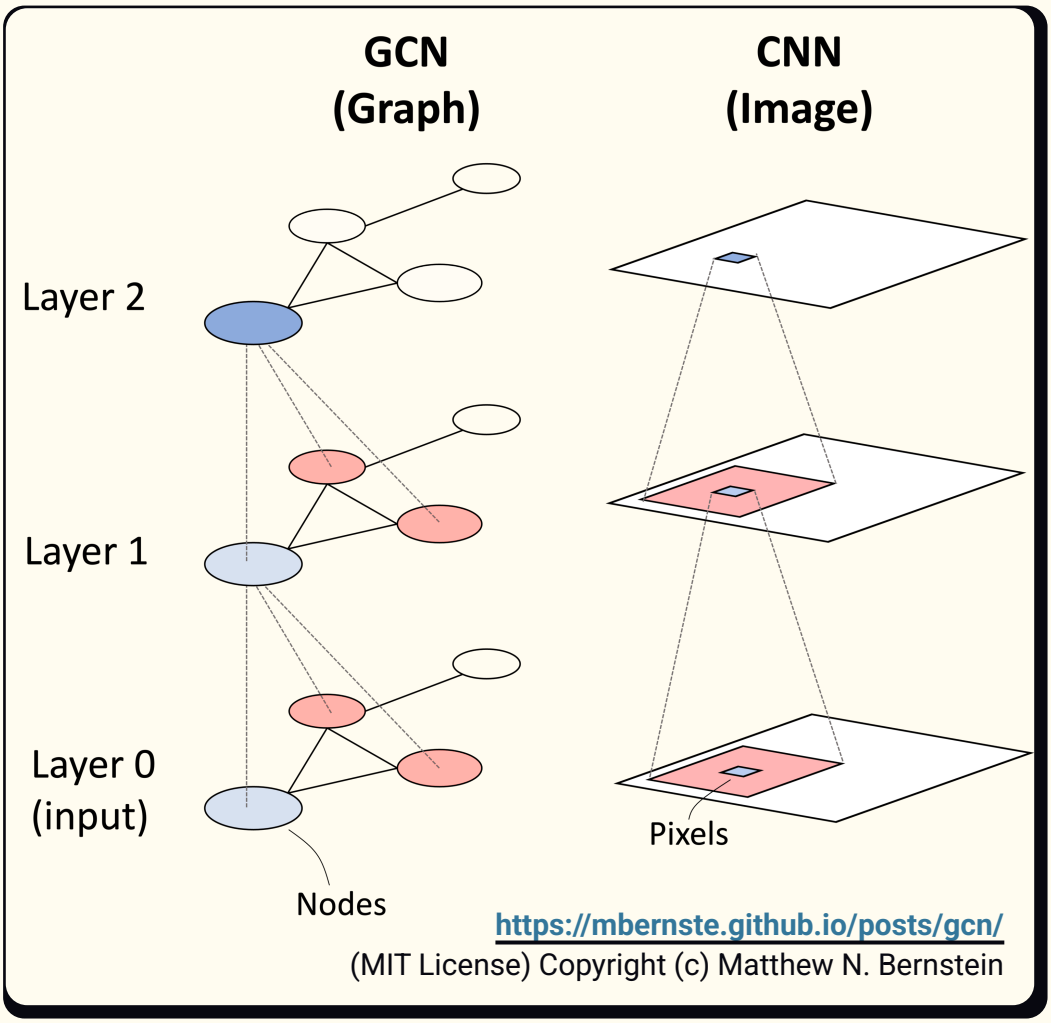


Figure 4: Principle of the graph convolution layer

A graph convolution layer basically performs a feedforward step for each node's neighbors and averages the result into said node.

Mathematically, this is represented in Equation 1:

$$N \in R^{n \times d} \wedge A \in R^{n \times n} \wedge D_i = 1 + \sum_{j=1}^{n} A_{ij} \Rightarrow$$
$$\mathrm{GCN}(N, A) = \mathrm{SwiGLU}\left(A \odot \frac{1}{\sqrt{D}} \odot \left(\frac{1}{\sqrt{D}}\right)^{\mathsf{T}} \cdot N\right) \quad (1$$

We use the SwiGLU activation function due to its ability for faster convergence. GCN are the new node features for the graph, with the feature size being dependant on SwiGLU's output size. The above equation closely mirrors the implementation in burn-rs.

burn-rs graph convolution layer implementation — Rust

```rust
impl Graph {
  pub fn normalized_adjacency_matrix(&self) -> Tensor<B, 2, Float> {
    let degree_matrix = self.edges.clone().sum_dim(1) + 1 ; // 1 + ∑_{i=1}^{n} A_{ij} → [n, 1]
    // Inverse square root of the degree matrix
    let degree_matrix_inv_sqrt = degree_matrix.powf_scalar(-0.5);
```

```
6
7       self.edges.clone()
8         // Normalize by deg(i) of the target node
9         .mul(degree_matrix_inv_sqrt.clone())
10        // Normalize by deg(j) of all the source nodes j in N_i
11        .mul(degree_matrix_inv_sqrt.transpose())
12    }
13 }
14
15 pub fn graph_convolution<B: Backend>(
16     graph: Graph<B, Float>,
17     swiglu: SwiGlu<B>,
18 ) -> Graph<B> {
19     let normalized_adjacency_matrix = graph.normalized_adjacency_matrix();
20     let neighbor_features = normalized_adjacency_matrix
21         .matmul(graph.nodes [n, d]);
22     let neighbor_features = swiglu.forward(neighbor_features) [n, d'];
23     Graph {
24         nodes: neighbor_features,
25         edges: graph.edges,
26     }
27 }
```

Only one neighbor deep!

If you are exceptionally meticulous (or have preknowledge) you notice that per convolution layer we can only aggregate features from the direct neighbors. We can aggregate features from more distant neighbors by stacking multiple convolution layers. That's why in Figure 2 we require multiple convolutions before pooling in order to consider meaningful AST macro structures.

1.4.4 Graph Diff Pooling

Graph DiffPool is a technique to reduce the number of nodes in a graph by calculating an assignment matrix $S \in R^{n \times n'}$ where n is the number of nodes in the original graph and m is the number of supernodes in the pooled graph. $N \in R^{n \times d}$ is the node features of the original graph and $F \in R^{n \times d'}$ is the embeded node features of the graph. Then $N' \in R^{n' \times d'}$ is the node features of the pooled graph. Formally expressed in Equation 2:

$$\begin{aligned} S &= \text{softmax}\left(\text{GNN}_{\text{Assigment}}(N)\right) \\ F &= \text{GNN}_{\text{Embed}}(N) \\ N' &= S^{\mathsf{T}} \cdot F \\ A' &= S^{\mathsf{T}} \cdot A \cdot S \end{aligned} \tag{2}$$

In code, that is easy to implement with the GCN:

```
1  impl<B: Backend> GraphDiffPool<B> {                                  Rust
```

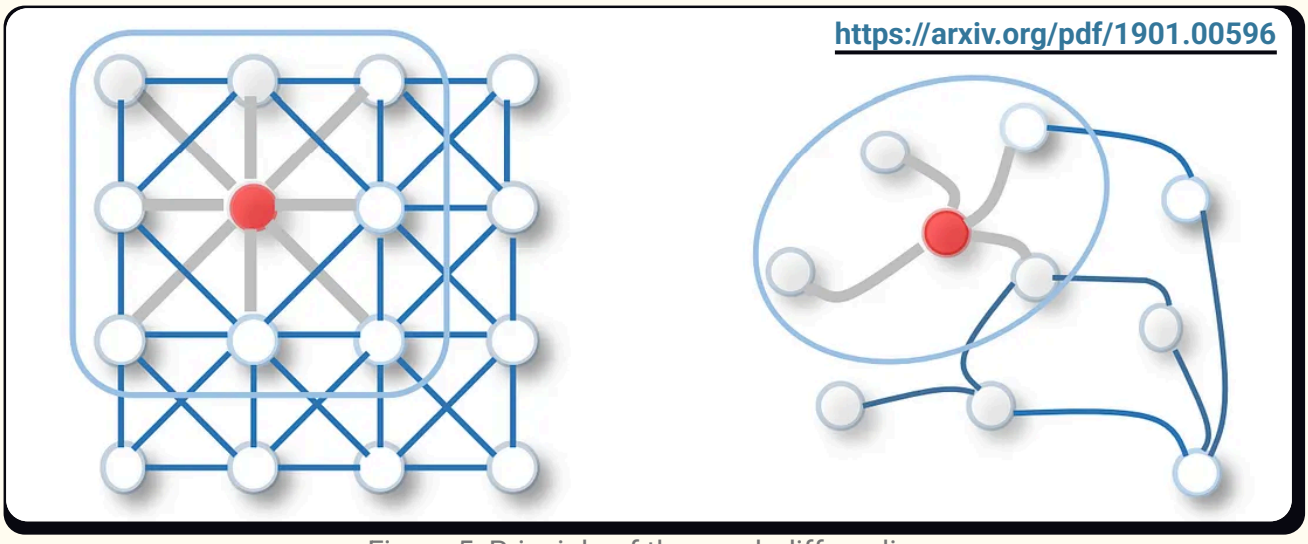


Figure 5: Principle of the graph diff pooling

```
    pub fn forward(&self, input: Graph<B>) -> Graph<B> {
        let soft_assignments [n, n'] = softmax(
          self.assignments_layer.forward(input.clone()).nodes,
          0
        );
        let node_embeddings [n, d'] =
        self.embed_layer.forward(input.clone()).nodes;
        let mut super_nodes [n', d'] =
        soft_assignments.clone().transpose().matmul(node_embeddings);
        let super_edges [n', n'] = soft_assignments
            .clone()
            .transpose()
            .matmul(input.edges)
            .matmul(soft_assignments);
        Graph::new(super_nodes, super_edges)
    }
}
```

Missing loss function 😯

The technique is designed to

- favor neighboring nodes to be assigned to the same supernode
- and maximize heterogeneity of the assignment matrix.

The loss functions for this were never implemented. I did not find out about them until researching for this blog post. Since I am transitioning away from GCN approach, there is no appeal in implementing them right now.

Now all the building blocks for the Figure 2 are available. With that, the architecture is similar to Figure 6.

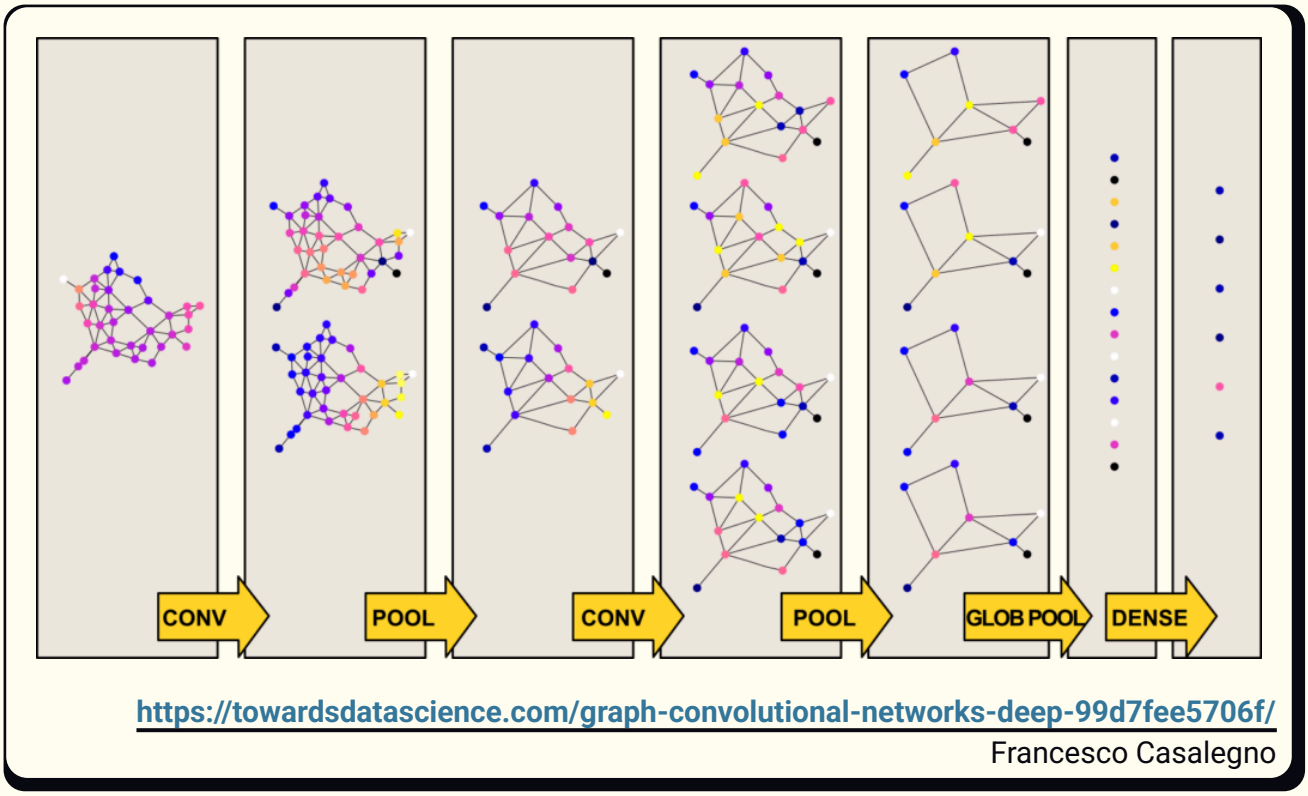


Figure 6: Stepwise graph feature condensation used for inspiration

1.5 Evaluation of GCN approach

Figure 7 shows an AUC score of **0.94**, which indicates very good classification performance of the GCN approach. Figure 8 shows an AUC **0.9**, which is not bad but nothing extraordinary.

With an F1-Score of **0.74** at a threshold of 0.5, the GCN approach is not the best performing model. However, it is still a good baseline for the project and the next steps will focus on improving the accuracy of the model.

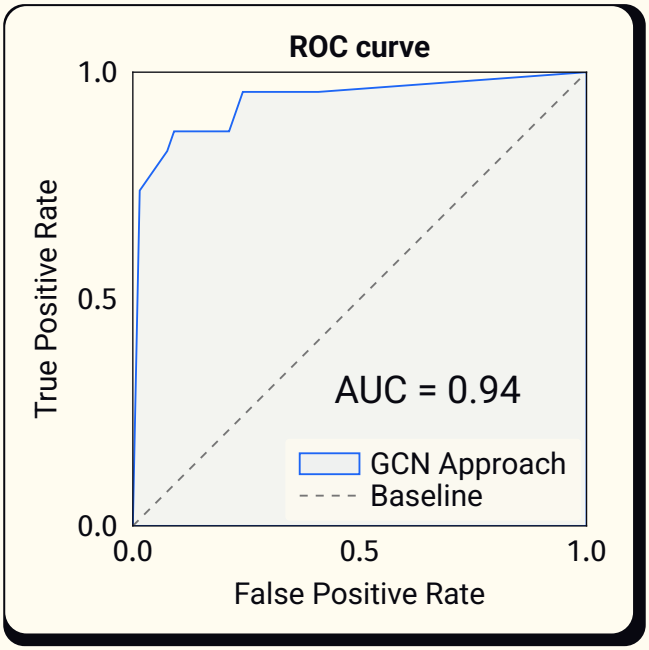


Figure 7: ROC curve for the GCN approach

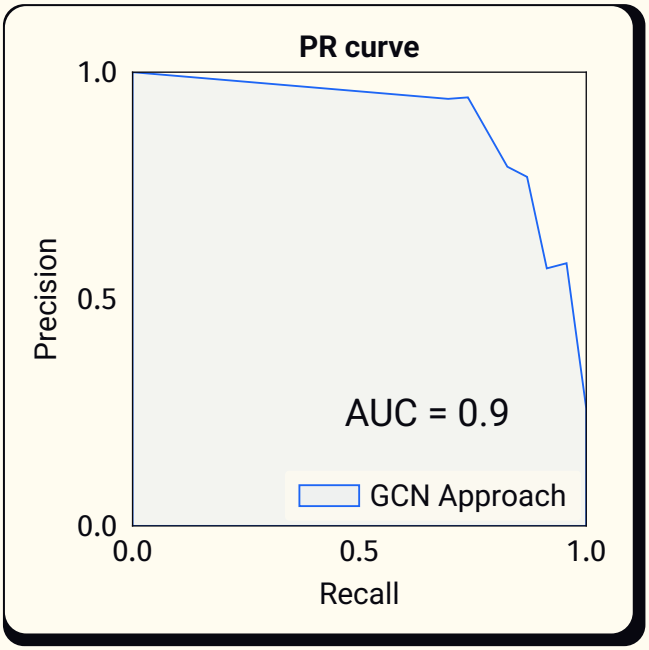


Figure 8: Precision-Recall curve for the GCN approach

2 GIN and Cosine Similarity

2.1 Expanding the Data Variety

One huge problem with the approach from the recap is the lackluster data variance in the used datasets. The amount of plagiarized assignments is significantly lower than expected.

So with Cursor Composer 2.5, we generated more data to train the model on:

- 640 Plagiarized pairs = 20 Tasks × 8 Authentic solutions × 4 Plagiarized solutions
- 560 Non-plagiarized pairs = 20 Tasks × $\frac{8 \text{ Authentic solutions} \times 7 \text{ Other authentics}}{2}$

The model was instructed to

1. Generate 20 task descriptions for C++ assignments, solvable in a single source code file of around ~100-ish lines of code.
2. Generate 8 authentic solutions for each task, writing them independently and without any knowledge of the other solutions.
3. For each authentic solution, generate 4 plagiarized solutions with different degrees of blatancy, employing same techniques as humans would.

Variant	Typical edits
plag1	Include reorder, member renames (`data_` → `cells_`), `i++` vs `++i`, `'\n'` ↔ `std::endl`
plag2	Extract helpers (e.g. `dotRow()`), Allman braces, swap input loop order (column-major), accumulate into local before assign
plag3	Heavy renames, casual comment, decrement loops `for (k = n; k-- > 0;)`
plag4	Lightest touch — still not copy-paste: a few renames + main variable renames

Task 2: Template Singly Linked List

Summary

Implement a generic singly linked list template with iterator support and use it to merge two sorted integer sequences from input.

Problem

Create a class template `LinkedList<T>` backed by nodes allocated on the heap. Support push_back, size, forward iteration, and a free function template `merge_sorted(const LinkedList<T>&, const LinkedList<T>&)` that returns a new sorted list without mutating the inputs (stable merge).

Read two whitespace-separated ascending integer sequences from stdin (sequence ends at newline; second sequence on the next line). Output the merged sequence space-separated on one line.

Authentic solution 1 for task 2 C++

```
⋮                                                    …
51  template <typename T>
52  LinkedList<T> merge_sorted(const LinkedList<T>& a, const LinkedList<T>& b) {
53      LinkedList<T> result;
54      auto ia = a.begin(), ib = b.begin();
55      auto ea = a.end(), eb = b.end();
```

```
    while (ia != ea && ib != eb) {
        if (*ia <= *ib) { result.push_back(*ia); ++ia; }
        else { result.push_back(*ib); ++ib; }
    }
    while (ia != ea) { result.push_back(*ia); ++ia; }
    while (ib != eb) { result.push_back(*ib); ++ib; }
    return result;
}
⋮                                                  …
```

Plagiarized solution 1 for task 2 C++

```
⋮                                                  …
template <typename T>
LinkedList<T> merge_sorted(const LinkedList<T>& a, const LinkedList<T>& b) {
    LinkedList<T> result;
    auto itA = a.begin(), itB = b.begin();
    auto endA = a.end(), endB = b.end();
    while (itA != endA && itB != endB) {
        if (*itA <= *itB) { result.emplace_back(*itA); ++itA; }
        else { result.push_back(*itB); ++itB; }
    }
    while (itA != endA) { result.emplace_back(*itA); ++itA; }
    while (itB != endB) { result.push_back(*itB); ++itB; }
    return result;
}
⋮                                                  …
```

Authentic solution 2 for task 2 C++

```
⋮                                                  …
template <typename T>
LinkedList<T> merge_sorted(const LinkedList<T>& L, const LinkedList<T>& R)
{
    LinkedList<T> out;
    auto i = L.begin(), j = R.begin();
    while (i != L.end() && j != R.end()) {
        if (*i <= *j) { out.push_back(*i); ++i; }
        else { out.push_back(*j); ++j; }
    }
    for (; i != L.end(); ++i) out.push_back(*i);
    for (; j != R.end(); ++j) out.push_back(*j);
    return out;
}
⋮                                                  …
```

This synthetic dataset is available under the projects repository in Section 4.

2.2 Graph Isomorphism Networks

Graph Isomorphism Networks (GINs) can be thought of pretty similar to GCNs, introduced in Section 1.4.3, where the main difference is the aggregation function being a sum of the node's neighbors' embeddings instead of a mean. The single GIN passes are repeated in order to extend the aggregation reach to multiple hops. For further processing, the residuals (the individual passes) are summed up along all nodes in the graph and concatenated to form a final graph embedding in the shape of number of passes × embedding size.

> **GINs are injective**
>
> Assuming all hidden layers also use injective activation functions, the GIN as a whole is also injective. Put differently, every different input graph will have a different output embedding. This makes it great for classification tasks where we want to be able to distinguish between different graphs.

With that, the new high level architecture becomes:

2.3 Cosine Similarity

After embedding the source files into a high-dimensional vector space, we can compute the cosine similarity between the vectors. The cosine similarity is a measure of how similar two vectors are, and is calculated as the dot product of the two vectors divided by the product of the magnitudes of the two vectors, as shown in Equation 3.

$$\cos(\theta) = \frac{a \cdot b}{\|a\| \times \|b\|} \tag{3}$$

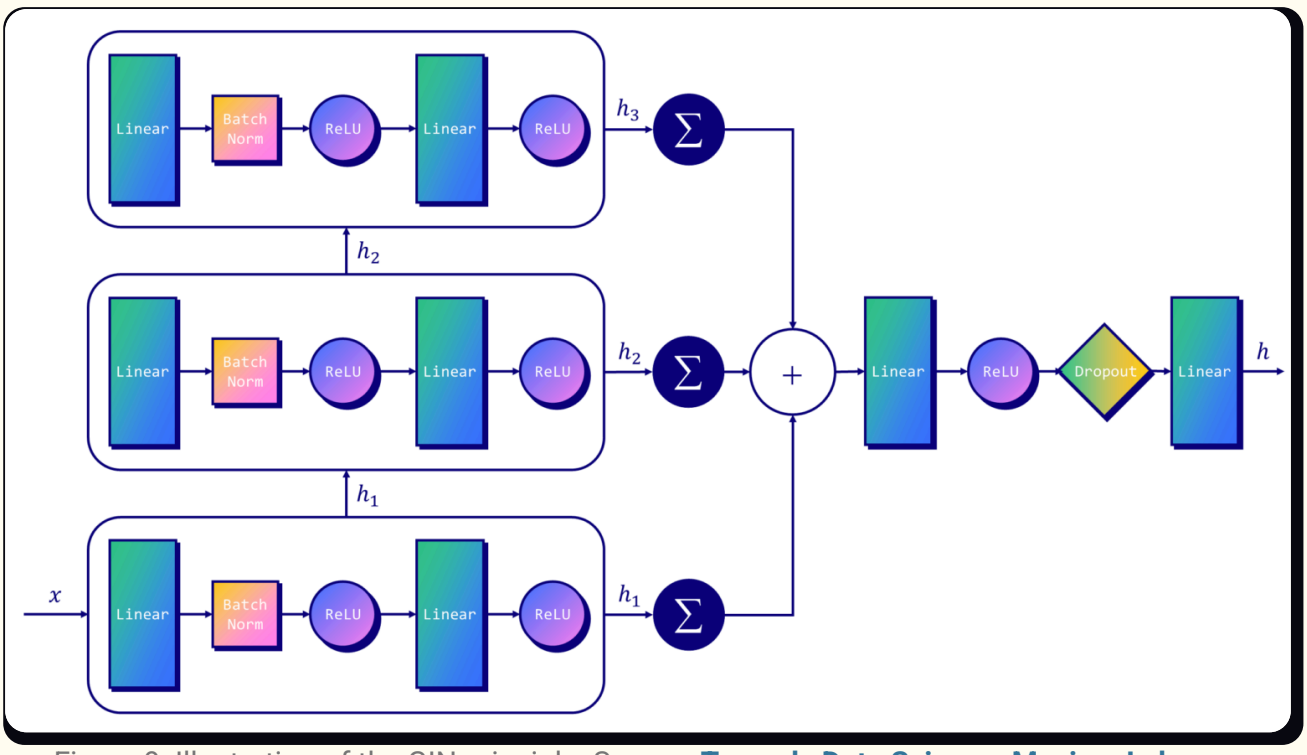


Figure 9: Illustration of the GIN principle. Source: [Towards Data Science, Maxime Labonne]

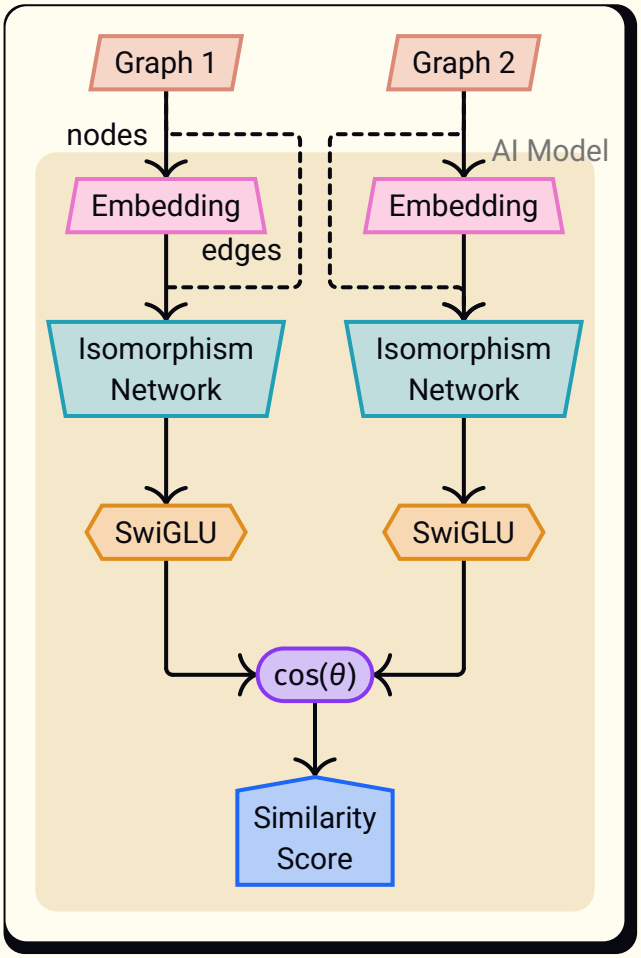


Figure 10: High-level ML pipeline for the GIN approach

Advantages of cosine similarity 🤯

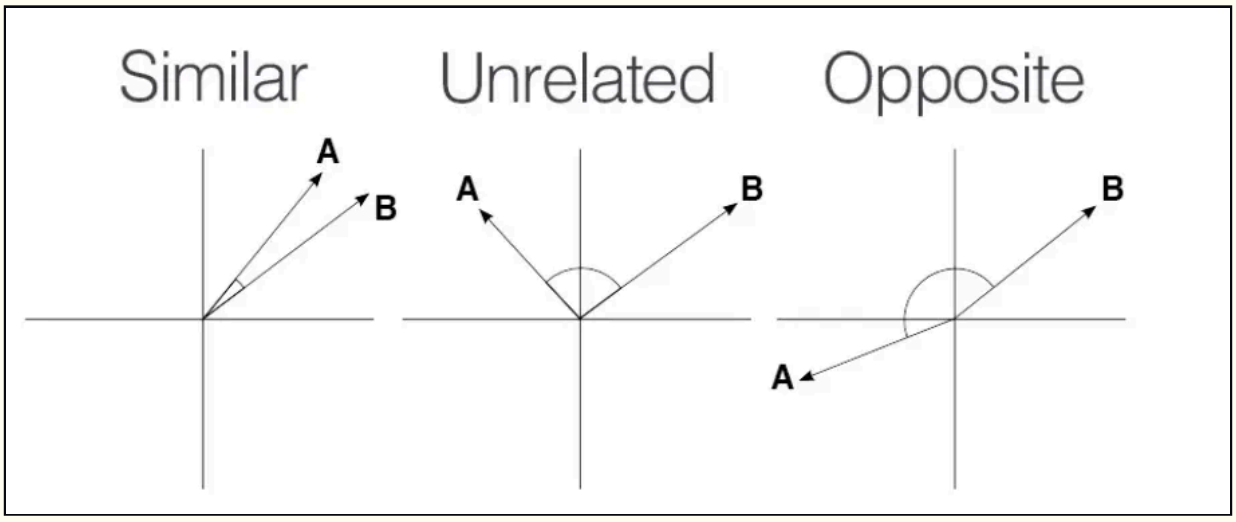


Figure 11: Cosine similarity visualization. Source: [Medium, Milana Shkhanukova]

The GIN inherent property of being injective guarantees longer embedding vectors for bigger ASTs, which is not meaningful for similarity of the solution approach. Through cosine similarity, we can compare the “direction” of the encoded solutions independent of their length.

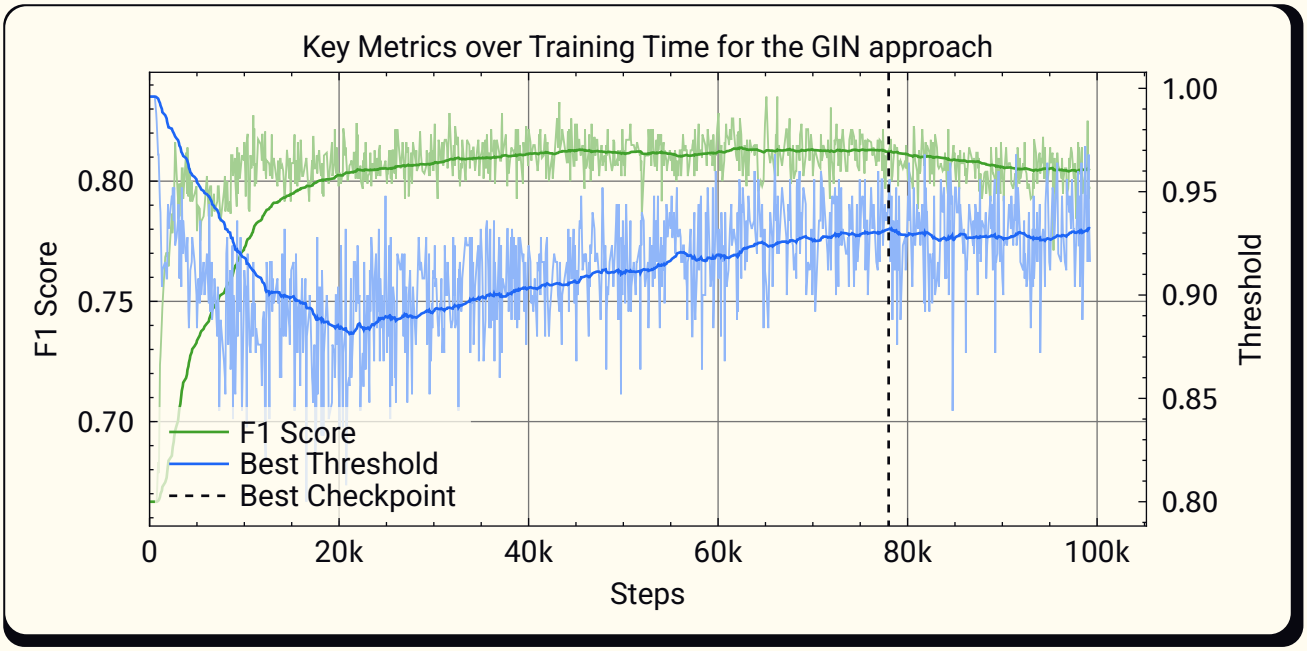


Figure 12: F1 Score and Best Threshold over Training Time for the GIN approach

2.4 Evaluation of GIN approach

After training for 99250 steps, the best performing model was at 78000, with an ideal threshold of 0.93 and an F1-Score (assuming 0.5 threshold) of **0.81** on the validation set. This is significantly higher than our baseline of 0.74 in the GCN approach.

Figure 13 shows an AUC score of **0.99**, which indicates very good classification performance of the GIN approach. However, Figure 14 shows a lower AUC of only **0.98**, indicating struggles with the precision in the minority cases (= plagiarism cases).

The sharp corner in these curves shows that the model is very good at differentiating between non-plagiarized and plagiarized solutions. Only a small change in the classification threshold is required in order to accurately predict all the cases. This indicates that nearly all positive cases are above a certain threshold and nearly all negative cases are below that same threshold.

I love the idea of embeddings. There is something intriguing about encoding some object into a high-dimensional vector space and then being able to reason about it in that space. Naturally, I performed some dimensionality reduction on the embeddings to visualize the source file representations in 2D using **annembed**, a crate inspired by t-SNE and UMAP.

Figure 15 colorizes every dot according the exercise the source file should solve. The green crosses mark a plagiarized group of source files. The embedding space clearly captures the similarity between high level task and low level implementation details, as evident by similarly colored clusters.

3 Shipping

Making this whole thing ready as web application.

burn-rs makes this easy

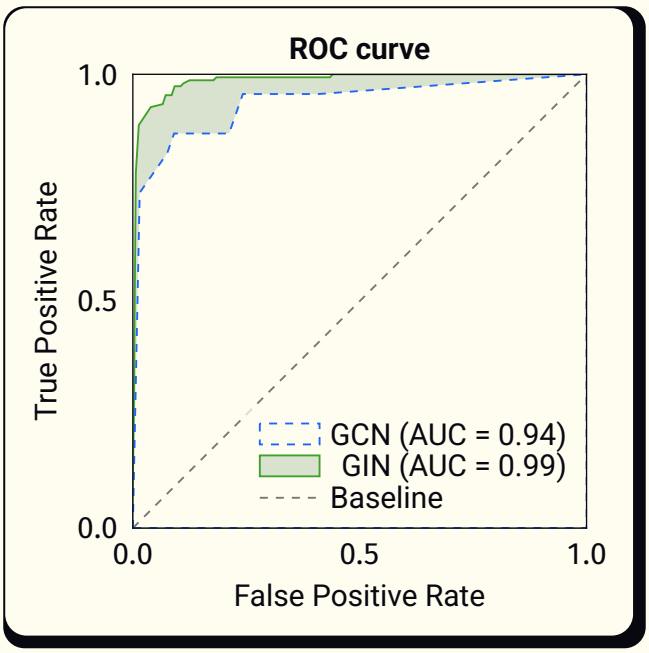


Figure 13: ROC curve comparing GCN and GIN approaches

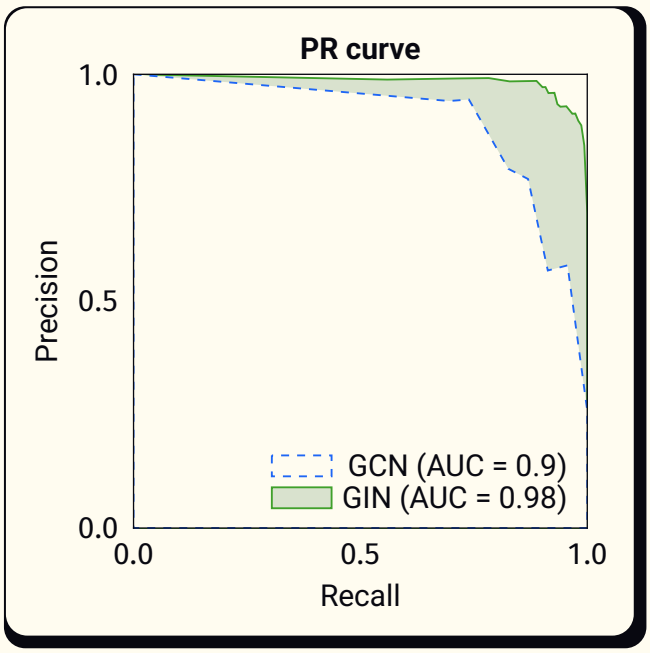


Figure 14: Precision-Recall curve comparing GCN and GIN approaches

With burn and Rust's wasm32 target, we can easily compile the model to a self contained binary executable by the browser. `include_bytes!("resources/production.bpk")` bundles all model weights and the model itself into a single file.

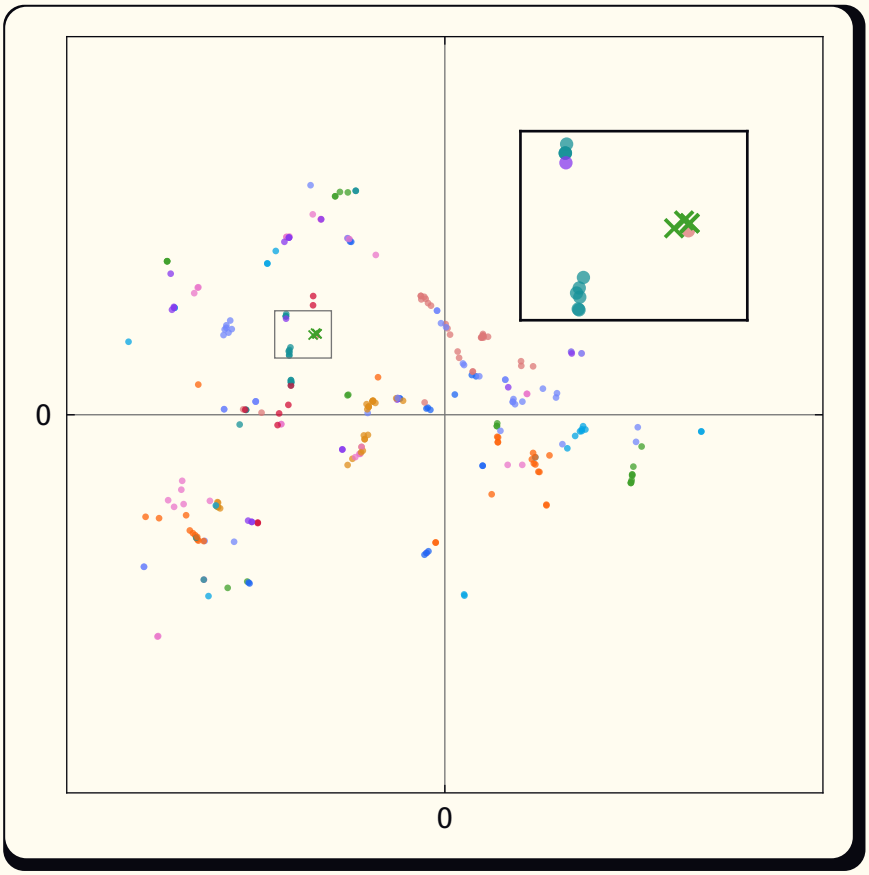


Figure 15: 2D source code embedding space for the GIN approach

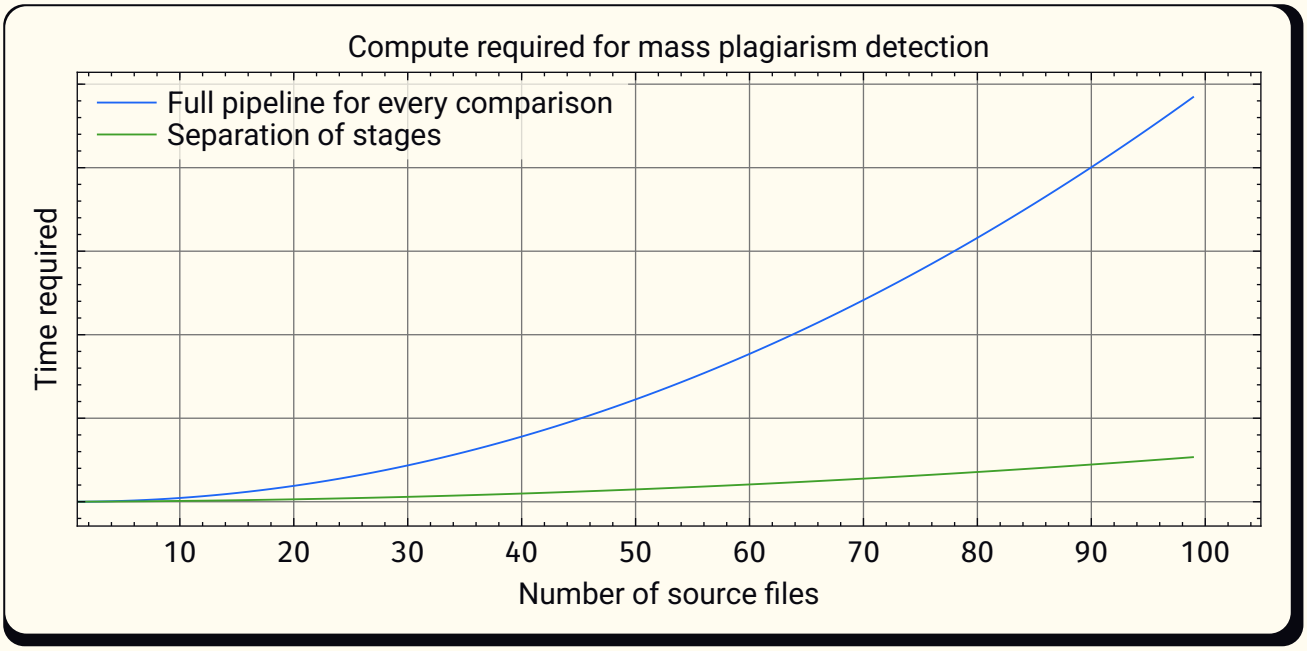


Figure 16: Time behavior comparison between full pipeline and stage separation

3.1 Mass plagiarism detection

The separation between heavy embedding computation and performant similarity calculation allows for efficient mass cross-file plagiarism detection. This becomes a two step process:

1. Compute the embeddings for all source files in the dataset
2. Compute the similarity of every embedding with every other embedding

Figure 16 illustrates the exponential savings in compute power with increasing number of source files.

3.2 Node decision importance

Backpropagation is great! Using the gradients for a single forward pass, we get a lot of nice information. Mainly, the importance of each node in the graph for the final decision. Every embedded AST node has a gradient vector $G \in R^{\text{embedding\_size}}$. The node importance is simply the magnitude of the gradient vector $\|G\|$.

~~You can try this out yourself in the interactive demo just down below.~~ Currently, you can't, because burn-rs has problems with autodifferentiation on web.

3.3 Try it out yourself

GitHub Pages Deployment

4 References

Project Repository: TimerErTim/plagiarinator

C++ Dataset: Programming Homework Dataset for Plagiarism Detection | IEEE DataPort

Dimensionality Reduction: annembed