

Student Name	Henry Dowd	Student Number	22535753
Supervisor	Diarmuid O'Donoghue	ECTS Credits	5
Project Title	Plagiarism Detection by Comparing Parse trees		

1) Project Objectives

2) Description of work completed

2.1) Evidence of work completed

2.2) Literature review

2.2) Use of GenAI and tools

3) Future Work

References (Sample below, any valid format is fine – IEEE, APA, etc.)

Dehaerne, Enrique, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. "Code generation using machine learning: A systematic review." *Ieee Access* 10 (2022): 82434-82455.

Appendices

1) Project Objectives

Goal: The goal of this project is to develop software that will be capable of identifying highly similar sentences, particularly those that are paraphrased, to be used in plagiarism detection. While there exist many tools that can effectively flag direct text matches, they are often incapable of identifying similarity when sentence structure changes even when overall meaning has not. To address this common limitation, this detection aims to analyse both the syntactic and semantic context of the input text.

Technical Overview: The project utilizes a dual-branch system. The Structural branch uses a dependency parser like SpaCy [1] or Stanza [2] to extract lexical dependencies. These are converted into graph representations, allowing sentences to be compared as tree structures. Similarity is calculated through the largest mapping between these trees.

The Semantic branch mitigates structural limitations. Since syntax can vary while meaning remains constant, this branch compares semantic content to capture “conceptual plagiarism” that purely syntactic methods might miss.

Key Results: The resulting system will be a tool that accepts sentence pairs and outputs a similarity score, derived from a combination of graph matching and semantic analysis. The systems effectiveness will be evaluated against the Microsoft Research Paraphrase Corpus (MSRP)[3] among others. This will provide a benchmark for the models accuracy in real-world scenarios. A secondary result of this project will be the evaluation of different graph comparison algorithms, determining which is most effective in NLP based structural matching.

2) Description of work completed

The development environment is implemented in Python. The data pipeline is functional and the Microsoft Research Paraphrase Corpus (MSRP) has been chosen as the benchmark dataset, ingested and pre-processed for later analysis. A technical review determined that SpaCy is the most effective tool for this project due to its balance of speed and ease of integration [1]. Initial phases of coding are complete, including scripts for serialization and notebooks for semantic experiments.

2.1) Evidence of work completed

Data Pre-processing: The data pipeline begins with a script to handle the ingestion of the raw MSRP data, using Pandas [4] to parse the text files and serializing the now structured data into Pickle format allowing simple and efficient access during training.

Following ingestion, exploring the data revealed a class imbalance, roughly 67% of the sentence pairs are labelled paraphrases while only 33% non-paraphrases [3]. This is critical for evaluation in the model as a basic classifier could just “guess” paraphrase every time and achieve 67% accuracy. To combat this, performance measurements will rely on F1-score and Precision/Recall rather than raw accuracy. The complexity of the of corpus was also examined, the corpus creators filtered out pairs with a Levenstein distance [5] lower than 8, this ensures there are no “trivial” paraphrases where only a few words differ [3].

Architecture Design: A modular architecture was designed to process the data

- Syntactic Branch: Research evaluated parsers on their ability to generate robust dependency graphs. The priority was finding a parser that balances speed with detail required for tree comparison.
- Semantic Branch: Baseline strategies were implemented using vector-based approaches.
- Fusion Model: The code structure for the final classifier has been started.

2.2) Literature review

Corpora: The Microsoft Research Paraphrase Corpus (MSRP) is the chosen benchmark. It is the industry standard for sentence-level similarity, containing 5,801 pairs [3]. Alternatives like the PAN Plagiarism Corpus [6] and Quora Question Pairs [7] were examined but deemed less suitable for initial benchmarking. The “non-paraphrase” examples in MSRP demonstrate high lexical overlap, acting as “hard negatives” that confuse simple string-matching, making it an ideal stress test.

Parsers: Comparing NLP libraries, SpaCy, Stanza, AllenNLP [8] and others, identified a clear trade-off between processing speed and how much information can be attached to each word. This review guided which of the researched tools could support the “Largest Common Subtree” analysis. The decision as to which software would be suited to this project prioritized libraries that allow the simple extraction of semantic parent-child relationships. SpaCy was selected as the primary parser with Stanza reserved for if SpaCy was found to be inadequate[1], [2].

2.3) Use of GenAI and tools

Generative AI tools were used throughout the development process they served as supporting and technical aids. Large Language Models (LLMs) in particular served three distinct functions:

- **Concept Explanation:** Tools like Microsoft Copilot[9] were consulted to clarify and explain theoretical constraints of complex graph algorithms, specifically the computational bottlenecks of Tree Edit Distance.
- **Coding Assistance:** GitHub Copilot[10] was used to accelerate writing boilerplate code for data ingestion and cleaning, allowing focus to stay on the core concepts and logic of the code.
- **Writing Refinement:** LLMs were used to assist with reviewing the technical report’s clarity and tone with the goal of ensuring language remained objective and formatted professionally. While tools assist the process, to ensure correctness and academic integrity, the final say on architecture and code is a human responsibility.

3) Future Work

Semantic Analysis: A major challenge in capturing sentence meanings is polysemy, words having different meaning depending on the context they are used in, simple word vectors struggle with this. The Semantic module will be upgraded to use contextual embeddings from a pre-trained Transformer model like BERT or RoBERTa[11], [12]. Unlike static embeddings, these models generate dynamic representations of words based on the surrounding text, offering much higher precision and avoiding misunderstanding words

intended meanings. Experiments will also include SIF (Smooth Inverse Frequency) sentence embeddings as a computationally lighter alternative to compare against transformer results.

Syntactic Analysis: The syntactic module relies on a graph comparison algorithm, the primary candidate for which being Tree Edit Distance (TED) [13], which calculates the minimum number of edits needed to turn one parse tree into another. An alternative approach, the Largest Common Substructure algorithm, will also be explored. This phase has high algorithmic complexity, requiring optimization to run efficiently on the full corpus.

Model Training and Evaluation: The final step is training a machine learning model, a supervised model like Logistic Regression or an SVM is ideal. The model will take the output of the syntactic and semantic comparison systems as features and generate a final probability score (0-100%). The evaluation will cover standard metrics like Precision, Recall and F1-score to the system can identify plagiarism without triggering too many false positives.

References

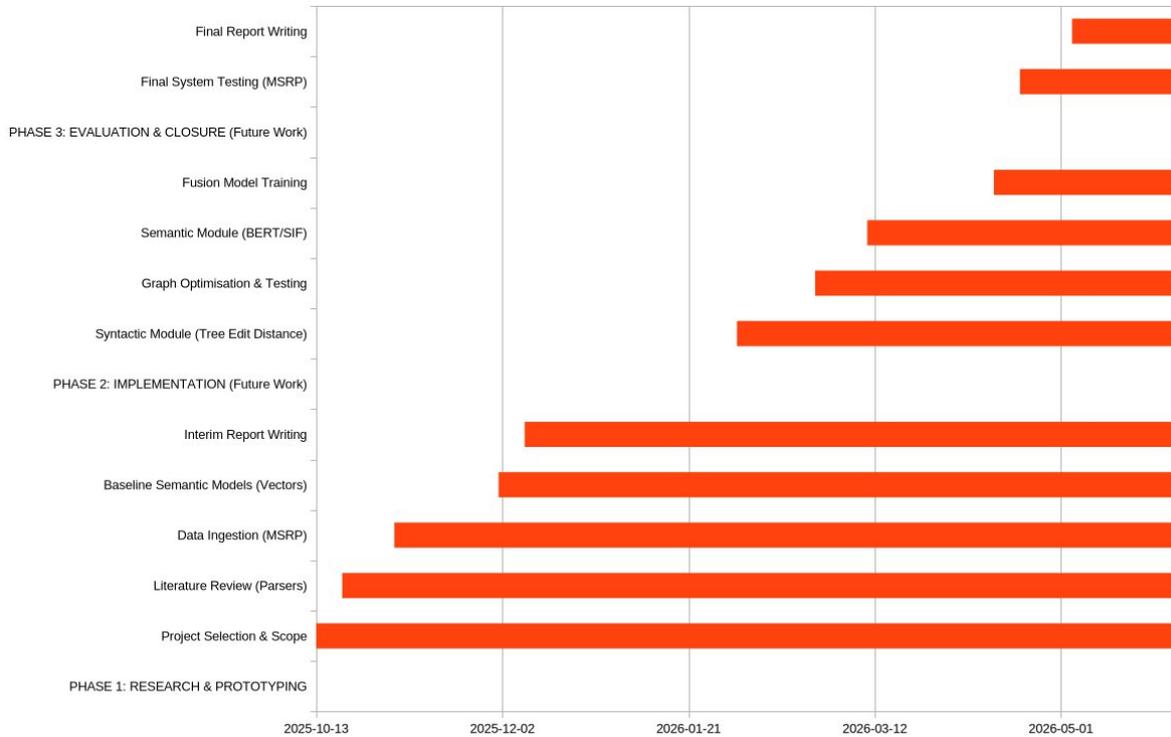
- [1] 'spaCy · Industrial-strength Natural Language Processing in Python'. Accessed: Nov. 30, 2025. [Online]. Available: <https://spacy.io/>
- [2] 'Overview', Stanza. Accessed: Nov. 30, 2025. [Online]. Available: <https://stanfordnlp.github.io/stanza/>
- [3] B. Dolan and C. Brockett, 'Microsoft Paraphrase Corups (MSRP)', presented at the Third International Workshop on Paraphrasing (IWP2005), Jan. 2005. Accessed: Nov. 30, 2025. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/automatically-constructing-a-corpus-of-sentential-paraphrases/>
- [4] 'pandas - Python Data Analysis Library'. Accessed: Nov. 30, 2025. [Online]. Available: <https://pandas.pydata.org/>
- [5] 'Levenshtein distance', *Wikipedia*. Nov. 17, 2025. Accessed: Nov. 30, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=1322698862
- [6] 'PAN'. Accessed: Nov. 30, 2025. [Online]. Available: <https://pan.webis.de/>
- [7] 'Question Pairs Dataset'. Accessed: Nov. 30, 2025. [Online]. Available: <https://www.kaggle.com/datasets/quora/question-pairs-dataset>
- [8] M. Gardner *et al.*, 'AllenNLP: A Deep Semantic Natural Language Processing Platform', May 31, 2018, *arXiv*: arXiv:1803.07640. doi: 10.48550/arXiv.1803.07640.
- [9] 'Microsoft Copilot: Your AI companion', Microsoft Copilot: Your AI companion. Accessed: Nov. 30, 2025. [Online]. Available: <https://copilot.microsoft.com>
- [10] 'GitHub Copilot', GitHub. Accessed: Nov. 30, 2025. [Online]. Available: <https://github.com/copilot>
- [11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding', May 24, 2019, *arXiv*: arXiv:1810.04805. doi: 10.48550/arXiv.1810.04805.
- [12] Y. Liu *et al.*, 'RoBERTa: A Robustly Optimized BERT Pretraining Approach', July 26, 2019, *arXiv*: arXiv:1907.11692. doi: 10.48550/arXiv.1907.11692.
- [13] K. Zhang and D. Shasha, 'Tree Edit Distance', *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989, doi: 10.1137/0218082.

Appendices

Notebooks: [Notebooks](#)
(code and visualizations)

A.

Gantt Chart



B.

Data Pipeline:

```
def load_msr_data(file_path):
    """Load the MSR Paraphrase Corpus from a TSV file."""
    df = pd.read_csv(file_path, sep='\t', quoting=3) # quoting=3 for ignoring quotes

    print(f"Loaded {len(df)} sentence pairs")
    #print(f"Positive examples (paraphrases): {df['quality'].sum()}")
    #print(f"Negative examples: {len(df) - df['quality'].sum()}")

    return df

def save_to_pickle(df, pickle_path):
    """Save the DataFrame to a pickle file."""
    df.to_pickle(pickle_path)
    print(f"DataFrame saved to {pickle_path}")

def load_and_save_data():
    """Load paraphrase data from user input and save as pickle"""
    print("Enter current relative path to MSR Corpus\n")
    relative_path = input("./ : ").strip()
    #full_path = os.path.join(raw_data_path, relative_path)

    try:
        df = load_msr_data(relative_path)
    except Exception as e:
        print(f"Error loading data: {e}")
        return None

    pkl_save_path = input("Enter relative path to save pickle: ").strip()
    if not os.path.isdir(pkl_save_path):
        print(f"Directory does not exist: ./{pkl_save_path}")
        return None

    pkl_filename = input("Enter pickle filename: ").strip() + ".pkl"
    full_pkl_path = os.path.join(pkl_save_path, pkl_filename)

    try:
        save_to_pickle(df, full_pkl_path)
    except Exception as e:
        print(f"Error saving pickle: {e}")
        return None

    print("Data loading and saving completed successfully.")
    print(f"Pickle saved at: {full_pkl_path}")

load_and_save_data()
```

Similarity Logic:

Similarity Methods

--- Jaccard Similarity ---

```
def jaccard_similarity(sent1, sent2):
    # make lowercase and split into words
    words1 = set(sent1.lower().split())
    words2 = set(sent2.lower().split())
    intersection = words1.intersection(words2)
    union = words1.union(words2)
    return float(len(intersection)) / len(union) if union else 0.0

print(f"{'Sentence 1':<41} {'Sentence 2':<40} {'Similarity Score:}'")
print("-" * 100)

for sent1, sent2 in test_pairs:
    similarity = jaccard_similarity(sent1, sent2)
    print(f"{sent1:<30} {'vs':<10} {sent2:<40} {similarity:.3f}") # 3 decimal places
    #print("-"* 50)
```

✓ 0.0s Python

Sentence 1		Sentence 2:	Similarity Score:
The cat sat on the mat.	vs	The cat sat on the mat.	1.000
The cat sat on the mat.	vs	The cat sat on the mat	0.667
The cat sat on the mat.	vs	The cat sat on the mat.	1.000
The cat sat on the mat.	vs	On the mat, the cat was sitting.	0.375
The cat sat on the mat.	vs	The feline rested on the rug.	0.250
The quick brown fox jumps.	vs	A fast brown fox leaps.	0.250
The cat sat on the mat.	vs	The dog ran in the park.	0.111
I love programming.	vs	She enjoys reading books.	0.000
The weather is nice today.	vs	It's raining outside.	0.000
Short.	vs	Short.	1.000
A B C D E F G	vs	A B C D E F G	1.000
	vs		0.000

--- Levenshtein Similarity ---

Character & Word
▷ 1 cell hidden ...

--- Cosine Similarity ---

▷ 1 cell hidden ...

--- Fuzzy ratios ---

▷ 1 cell hidden ...

--- Longest common sub-sequence ----

▷ 1 cell hidden ...

--- Containment Similarity ---

Percentage of Sentence A in Sentence B
 $\text{containment}(A, B) = \frac{|\text{words}(A) \cap \text{words}(B)|}{|\text{words}(A)|}$

▷ 1 cell hidden ...

Compare Performance

Analyze and compare each methods performance

▷ 4 cells hidden ...

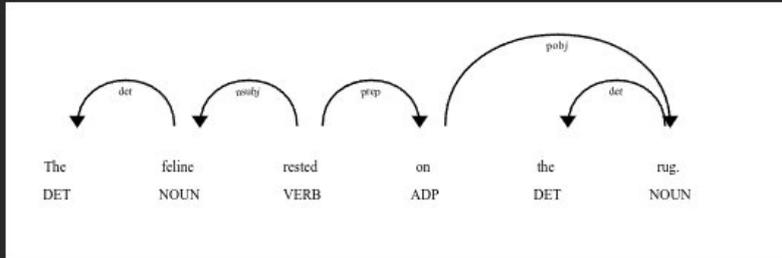
Parsing Visualization:

Sentence 4: 'The feline rested on the rug.'
 Dependency Parse:

```

-----
The      det      feline      []
feline   nsubj     rested      ['The']
rested   ROOT      rested      ['feline', 'on', '.']
on       prep      rested      ['rug']
the      det      rug         []
rug      pobj     on          ['the']
.        punct   rested      []
    
```

Dependency Tree Visualization - Sentence 4:



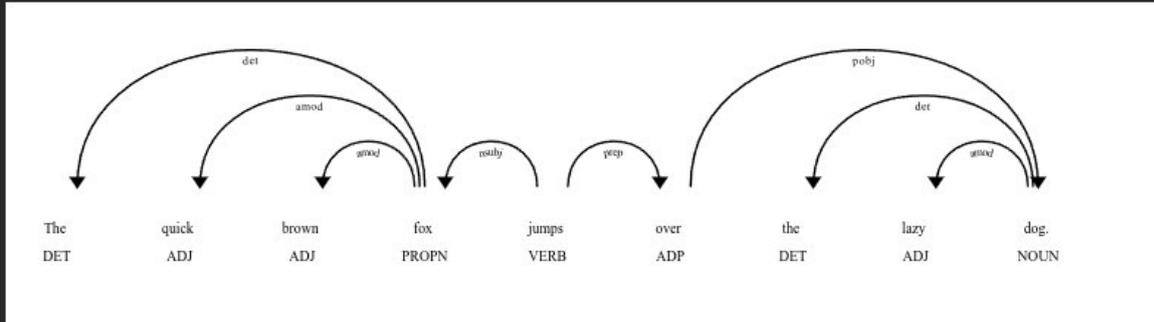
=====

Sentence 5: 'The quick brown fox jumps over the lazy dog.'
 Dependency Parse:

```

-----
The      det      fox         []
quick    amod     fox         []
brown    amod     fox         []
fox      nsubj     jumps       ['The', 'quick', 'brown']
jumps    ROOT     jumps       ['fox', 'over', '.']
over     prep     jumps       ['dog']
the      det      dog         []
lazy     amod     dog         []
dog      pobj     over        ['the', 'lazy']
.        punct   jumps       []
    
```

Dependency Tree Visualization - Sentence 5:



C. (AI)

1.

Prompt:

Explain the time complexity of the Zhang-Shasha algorithm for Tree Edit Distance. Why does it become a bottleneck when comparing thousands of sentence parse trees?

Response:

Great question — let's break this down clearly.

Zhang–Shasha Algorithm for Tree Edit Distance

The **Zhang–Shasha algorithm** (1989) is the classic dynamic programming approach for computing the **tree edit distance (TED)** — the minimum cost of transforming one ordered labeled tree into another using operations like insert, delete, and relabel.

Time Complexity

- For two trees with sizes n and m (number of nodes):
 - Worst-case time complexity:
$$O(n \cdot m \cdot \min(\text{depth}(T_1), \text{leaves}(T_1)) \cdot \min(\dots))$$
- Simplified: often cited as $O(n^3)$ in the worst case for trees of similar size.
- Space complexity: $O(n \cdot m)$.

This cubic behavior arises because:

- The algorithm decomposes trees into subforests.
- It computes edit distances between all pairs of subforests using dynamic programming.
- The number of subforest pairs grows quickly with tree size.

Why It Becomes a Bottleneck for Parse Trees

Sentence parse trees (from NLP) typically have:

- **Dozens of nodes per sentence** (sometimes hundreds for complex sentences).
- Thousands of sentences in a corpus → millions of pairwise comparisons.

Bottleneck Factors

- **Quadratic-to-cubic scaling:** Even if each tree has ~50 nodes, comparing two trees can take on the order of $50^3 = 125,000$ operations. Multiply that by thousands of comparisons, and the cost explodes.
- **Dynamic programming overhead:** Each comparison requires filling large DP tables, not easily parallelized at fine granularity.

- **Corpus size:** Comparing all pairs of parse trees in a dataset of size N requires $O(N^2)$ comparisons. With thousands of sentences, this means millions of expensive TED computations.

⚡ Intuition

- For small trees (like XML fragments or short phrases), Zhang–Shasha is fine.
- For **large-scale NLP tasks** (clustering, similarity search, paraphrase detection), the cubic complexity makes it impractical.
- That’s why researchers often use **approximations** (e.g., tree kernels, embeddings, or restricted edit distance variants) instead of exact TED.

☞ So in short: the Zhang–Shasha algorithm is **cubic in tree size**, and when combined with the quadratic number of comparisons across thousands of parse trees, it becomes computationally prohibitive.

Would you like me to also walk through **modern alternatives** (like APTED, tree kernels, or neural embeddings) that are used to overcome this bottleneck?

2.

Prompt:

What are the main computational differences between calculating the 'Largest Common Substructure' versus 'Tree Edit Distance' for dependency graphs? Which is faster for NLP tasks?

Response:

⚙️ Computational Differences

Largest Common Substructure (LCS)

- **Definition:** Finds the largest overlapping subgraph (or subtree) between two dependency graphs.
- **Complexity:** Equivalent to solving subgraph isomorphism, which is **NP-complete**.
- **Operations:** Requires exhaustive search or heuristics to identify maximal shared substructures.
- **Strengths:** Captures *global structural similarity* and exact overlap.
- **Weaknesses:** Computationally expensive, especially for large graphs in NLP.

Tree Edit Distance (TED)

- **Definition:** Measures the minimum number of edit operations (insert, delete, substitute) to transform one tree into another.

- **Complexity:** Polynomial-time algorithms exist (e.g., $O(n^3)$ for ordered trees)

3.

Prompt:

I am building a plagiarism detector using dependency trees. Is Tree Edit Distance (TED) feasible for a dataset of 5,000 sentence pairs (MSRP), or is it too slow? Suggest optimizations

Response:

⚙️ Feasibility of Tree Edit Distance (TED) on MSRP

- **Complexity:** The standard TED algorithm (Zhang & Shasha, 1989) runs in $O(|T_1| \cdot |T_2| \cdot \min(\text{depth}(T_1), \text{leaves}(T_1)) \cdot \min(\dots))$. For dependency trees, sentence length is usually ≤ 30 tokens, so trees are relatively small.
- **Dataset size:** MSRP has $\sim 5,800$ sentence pairs. Computing TED for each pair is feasible if optimized — but naïve implementations can be slow (seconds per pair \rightarrow hours total).
- **Practicality:** With efficient implementations (e.g., APTED algorithm), TED can handle thousands of pairs in minutes, not hours. So yes, it's feasible, but you'll want optimizations.

Optimizations

Here are strategies to make TED practical:

- **Use APTED (All Path TED)**
 - A modern algorithm that improves TED runtime significantly compared to Zhang-Shasha.
 - Implementations exist in C++/Java/Python.
- **Tree simplification**
 - Collapse function words (articles, auxiliaries) into parent nodes.
 - Normalize dependency labels (e.g., treat nsubj and csubj as equivalent).
 - Reduces tree size \rightarrow faster TED.
- **Parallelization**
 - Each sentence pair is independent. Run TED computations in parallel across CPU cores or on a cluster.
- **Approximate TED**

- o Instead of exact TED, use heuristics like subtree overlap, bag-of-dependencies, or partial matching.
- o Faster and often sufficient for plagiarism detection.
- **Pruning**
 - o If sentence lengths differ drastically, skip TED (likely non-paraphrase).
 - o Use cheap similarity measures (e.g., cosine on embeddings) as a filter before TED.
- **Hybrid approach**
 - o Use embeddings (e.g., SBERT) for coarse similarity.
 - o Apply TED only on borderline cases to refine decisions.

📌 Practical Workflow Suggestion

1. **Pre-filter** with embeddings (fast cosine similarity).
2. **Run TED** only on pairs above a similarity threshold.
3. **Optimize trees** (collapse function words, normalize labels).
4. **Parallelize** across cores.

This way, you combine **semantic embeddings** (speed) with **syntactic TED** (precision).

🗨️ Would you like me to sketch out a **runtime estimate** (e.g., how many minutes TED might take on MSRP with APTED vs naïve Zhang-Shasha), or a **code outline** showing how to integrate TED with a pre-filtering step?

4.

Prompt:

Explain the concept of 'graph isomorphism' in the context of dependency parsing. How is it different from simple tree matching?

Response:

🌐 Graph Isomorphism in Dependency Parsing

- **Dependency parsing** represents the syntactic structure of a sentence as a **graph**:
 - o Words are **nodes**.
 - o Dependency relations (like *subject*, *object*, *modifier*) are **edges**.
- Two dependency graphs are **isomorphic** if there exists a **one-to-one mapping** between their nodes such that:
 - o The mapping preserves **labels** (word identity or part-of-speech).

1000-word limit (not including Appendices), A4, 12-point single spaced Times/Arial font (or equivalent).

- o The mapping preserves **edges** (dependency relations).
- In other words, the graphs have the **same structure**, even if the nodes are named differently. Example:
 - o "The cat chased the mouse."
 - o "A dog pursued a rabbit." If both parse into graphs with identical dependency relations (subject → verb → object), they are **isomorphic**.

Tree Matching

- **Tree matching** is a simpler comparison:
 - o It checks whether two **tree structures** (often rooted, ordered trees) are identical in shape.
 - o Typically used when dependency parses are constrained to trees (no cycles, one root).
- Tree matching does **not allow flexible relabeling** of nodes.
 - o It requires exact matches of node labels and positions.
- Example:
 - o "The cat chased the mouse." vs. "The dog chased the rabbit." Tree matching would fail because the words differ, even though the structure is the same.

Key Differences

Aspect	Graph Isomorphism	Tree Matching
Structure type	Works on general graphs (trees + possible DAGs)	Works only on trees
Node labels	Can ignore or remap labels if structure is preserved	Requires exact label matches
Flexibility	More abstract — focuses on structural equivalence	More rigid — focuses on identical shape + labels
Use case in parsing	Identifies structurally equivalent parses across sentences	Checks if two parses are literally the same tree

Why It Matters in Dependency Parsing

- **Graph isomorphism** helps in recognizing **paraphrases** or structurally equivalent sentences with different words.
- **Tree matching** is stricter, useful for exact syntactic identity checks (e.g., in grammar validation).

So, in short:

- **Graph isomorphism** = "Do these parses have the same *structure* regardless of labels?"
- **Tree matching** = "Are these parses *exactly identical* in both structure and labels?"

Would you like me to illustrate this with a **visual example of two dependency graphs** — one isomorphic and one not — so you can see the difference more concretely?

5.

Prompt:

Write a Python function using Pandas to load a tab-separated file (TSV), treating quotes as literal characters (quoting=3), and print the number of rows loaded

Response:

Here's a clean Python function that does exactly what you asked:

```
python
```

```
import pandas as pd
```

```
import csv
```

```
def load_tsv(file_path: str):
```

```
    """
```

```
    Load a tab-separated file (TSV) using pandas,  
    treating quotes as literal characters (quoting=3),  
    and print the number of rows loaded.
```

```
    Parameters:
```

```
    file_path (str): Path to the TSV file.
```

```
    """
```

```
    try:
```

```
        # quoting=csv.QUOTE_NONE is equivalent to 3
```

```
        df = pd.read_csv(file_path, sep='\t', quoting=csv.QUOTE_NONE)
```

```
        print(f"Number of rows loaded: {len(df)}")
```

```
        return df
```

```
    except Exception as e:
```

```
        print(f"Error loading file: {e}")
```

```
    return None
```

Explanation

- `sep='\t'` → ensures the file is treated as tab-separated.
- `quoting=csv.QUOTE_NONE` (which equals 3) → treats quotes as literal characters instead of special delimiters.
- `len(df)` → gives the number of rows loaded.
- The function prints the row count and returns the DataFrame for further use.

Would you like me to also show you how to **test this function with a sample TSV file** (created in code), so you can see it working end-to-end?

6.

Prompt:

I have a text file where some lines have bad formatting. Write a script to read the file line by line and try-except the parsing logic to skip errors without crashing.

Response:

Got it 🙌 — you want a robust script that processes a text file line by line, but gracefully skips over any badly formatted lines instead of crashing. Here's a clean Python example:

```
def process_line(line):
    """
    Example parsing logic.
    Replace this with whatever parsing you need.
    For demonstration, we'll assume each line should be 'key:value'.
    """
    key, value = line.strip().split(":")
    return key, value

def read_file(filename):
    results = []
    with open(filename, "r", encoding="utf-8") as f:
        for line_number, line in enumerate(f, start=1):
            try:
                parsed = process_line(line)
                results.append(parsed)
            except Exception as e:
                # Skip bad lines but log the error
                print(f"Skipping line {line_number}: {line.strip()} (Error: {e})")
    return results

if __name__ == "__main__":
    filename = "data.txt" # replace with your file path
    parsed_data = read_file(filename)
    print("Successfully parsed lines:")
    for item in parsed_data:
```

```
        print(item)
```

Key points

- process_line function: Customize this with your actual parsing logic.
- try-except block: Ensures the script won't crash if a line is malformed.
- line_number tracking: Helps you identify which lines failed.

- logging skipped lines: Prints out the error so you know what went wrong, but continues processing.

This way, the script is resilient: it processes all valid lines and skips over the bad ones without stopping execution.

Would you like me to adapt this for a **CSV-style file** (comma-separated values) or keep it general-purpose like above?

7.

Prompt:

How do I save a Pandas DataFrame to a pickle file in Python to preserve the data types exactly?

Response:

To save a **Pandas DataFrame** to a **pickle file** while preserving the data types exactly, you can use the built-in `to_pickle()` method. Pickle serialization stores the DataFrame in binary format, ensuring that when you load it back, all data types (including complex ones like categorical, datetime, or custom objects) are preserved.

Here's how you can do it:

Saving a DataFrame to Pickle

```
import pandas as pd
```

```
# Example DataFrame
```

```
df = pd.DataFrame({  
    "id": [1, 2, 3],  
    "name": ["Alice", "Bob", "Charlie"],  
    "score": [95.5, 82.3, 74.8],  
    "passed": [True, True, False]  
})
```

```
# Save to pickle file
```

```
df.to_pickle("dataframe.pkl")
```

Loading

the

Pickle

File

Back

```
# Load the DataFrame back
```

```
df_loaded = pd.read_pickle("dataframe.pkl")
```

```
print(df_loaded)
```

```
print(df_loaded.dtypes) # Data types are preserved
```

Key Points

- **Pickle preserves data types:** Unlike CSV or Excel formats, which may coerce types (e.g., turning integers into floats or losing categorical info), pickle keeps the exact Python objects.

- **File extension:** Conventionally `.pkl` or `.pickle`.
- **Cross-version caution:** Pickle files are Python-specific. They may not be portable across different Python versions or Pandas versions. For long-term storage or sharing, formats like **Parquet** or **Feather** are safer.
- **Security note:** Only load pickle files you trust, since they can execute arbitrary code when unpickled.

Would you like me to also show you how to save the same DataFrame in a **Parquet format** (which is more portable and still preserves data types), or do you want to stick with **pickle only**?

8.

Prompt:

Review the following paragraph for academic tone. Remove any personal pronouns like 'I' or 'we' and make it sound more objective: My primary aim in this project is to build a software system that I designed to identify highly similar sentences, specifically so I can catch plagiarism and paraphrased text. I found that current detection tools often flag direct text matches but fail when words are swapped, even if the core message remains the same. To address this limitation, my detector analyzes two specific elements: syntactic structure (grammar) and semantic context (meaning).

Response:

Here's a revised version of your paragraph with a more academic tone and without personal pronouns:

Revised Paragraph (Academic Tone): The primary aim of this project is to develop a software system designed to identify highly similar sentences, with a particular focus on detecting plagiarism and paraphrased text. Existing detection tools frequently flag direct text matches but often fail when words are substituted, even if the underlying meaning remains unchanged. To address this limitation, the proposed detector analyzes two key elements: syntactic structure (grammar) and semantic context (meaning).

This version emphasizes objectivity, avoids personal pronouns, and uses more formal phrasing. Would you like me to make it even more research-oriented by adding phrases such as "this study proposes" or "the system demonstrates" to align it with typical academic writing conventions?

9.

Prompt:

Reword this sentence to be more formal: 'The parser was really slow so I decided to switch to SpaCy because it is faster.'

Response:

Here's a more formal rewording of your sentence:

"The parser exhibited significant latency, so I opted to transition to SpaCy due to its superior processing speed."

Alternative variations you might also consider, depending on tone:

- "Because the parser performed inefficiently, I chose to adopt SpaCy for its faster execution."
- "Given the parser's slow performance, I decided to migrate to SpaCy, which offers greater efficiency."

Would you like me to make it academic-style (e.g., suitable for a research paper) or professional-business style (e.g., suitable for a report or email)?

10.

Prompt:

Check this section for grammatical errors and ensure the technical terminology regarding 'dependency parsing' is used correctly.

Response:

Could you please share the section you'd like me to review? Right now I don't see any text attached. If you paste the passage here, I'll go through it carefully, correct any grammatical issues, and make sure the terminology around dependency parsing is precise and used correctly.

For example, if your section says something like:

"Dependency parsing is a technique that breaks down sentence into relations between words."

I would refine it to:

"Dependency parsing is a technique that analyzes a sentence by identifying grammatical relations between words."

This way, the grammar is fixed ("sentence" → "a sentence"), and the technical definition is accurate.

Go ahead and drop the text, and I'll polish it for you.