Research Project on Sequence Classification via Graph

Final Report

CSC 6850 Machine Learning

SPRING 2024

Alenka Tang, Omar Madjitov

❖ Introduction

- ➢ **Project Type:** research project in sequence classification using graph models
- ➢ **Background and Motivation**

  Classification is not a new topic in bioinformatics. In the analysis of gene expression data, genes obtained from microarray data are clustered and genes in the same cluster are considered to trigger the same function. Sequence classification is an important task in bioinformatics to produce stable, quick, and accurate results.

- ➢ **Problem Definition**

  This project proposed a method using graphical models for sequence classification where the graph model will provide the features for training classifiers. We will take existing models such as the de-Bruijn graph[1], and OverlapGraphs[5], then use Node2vec [7], and Graph2vec embedding models to create input vectors for each sequence data which will be used to train Support Vector Machine (SVM) and Neural Network (NN) classifiers and test them with different parameters to find the combination with optimal results for sequence classification.

❖ Methodology

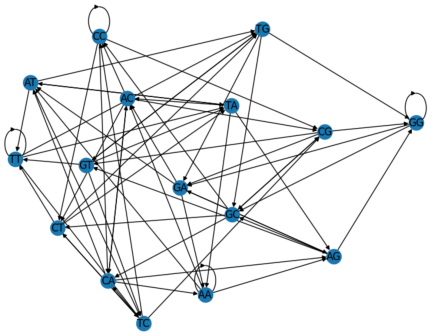
- ➢ **Datasets**

  This project will use the Human DNA (HD) sequence database consisting of 4380 DNA sequences with each DNA sequence corresponding to a specific gene family

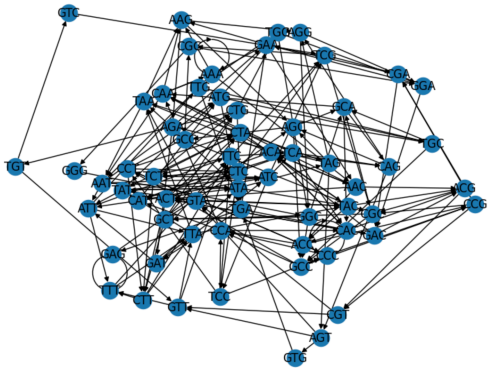
which can be used as labels for training and testing the model, with a total of seven labels. Due to time constraints, we only used the first 1000 DNA sequences to test the model.

➢ **De-Bruijn Graph Construction**

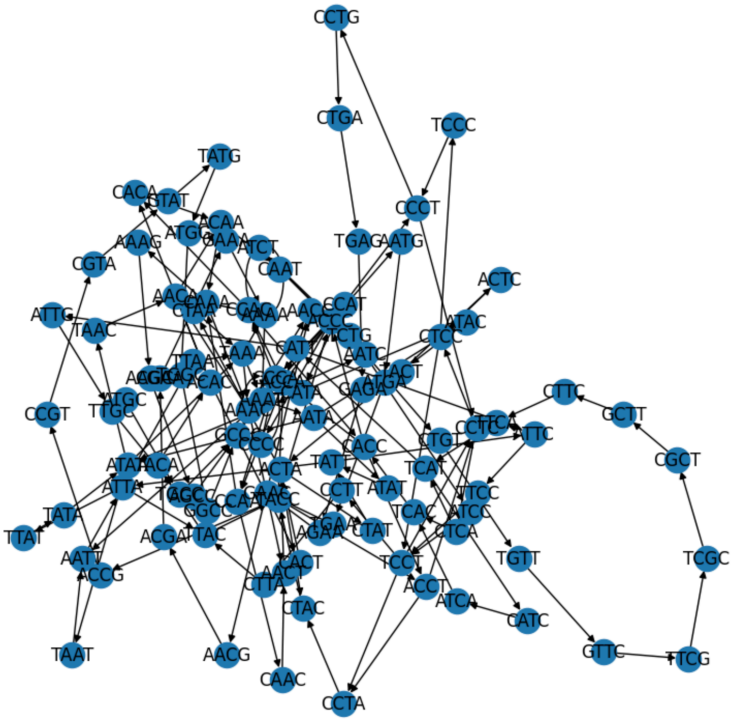
This project will use the de-Bruijn graph model, as well as Overlap Graphs to convert the sequence data to graphs with each sequence data entry being one graph. For constructing the de-Bruijn graph for each data entry, we will try different k-mer values and break the sequence into overlapping k-mers. The graph will be constructed by creating nodes for each k-mer and connecting overlapping k-mers with direct edges. Lastly, we will simplify the graph by merging nodes with common overlaps with each sequence data in the dataset being an individual graph. The complexity of the graph will first increase and then decrease as we increase the value of kmers.



kmers = 3



kmers = 4



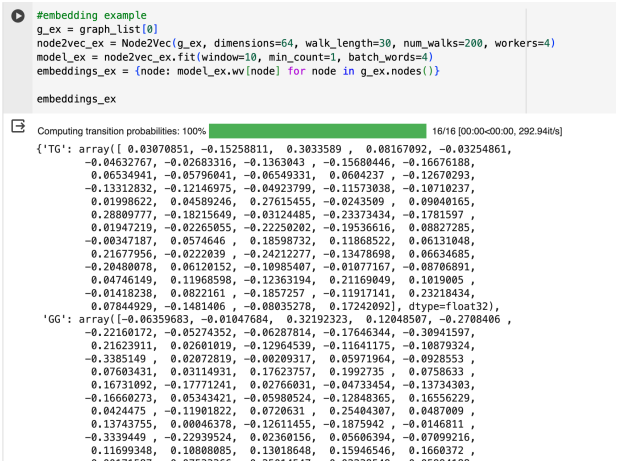
Kmer = 5

➢ **Node Embedding Models**

After constructing the graphs from the sequence data, we use Node2vec and embedding models to create the input vectors from each graph.

The Node2vec embedding model uses random walks to explore the graph and outputs a matrix with information for each node. Increasing the kmer value when constructing the de-bruijn graph causing the increase in several nodes and the complexity of the graph would also increase the processing time for the Node2vec embedding models. When we increased the kmer value to 10, the Node2vec embedding model took 3 hours to complete 50 graphs which limited the amount of data we could use as the sample in a short period of time.

The output of the Node2vec model is a matrix where each node has its own vector. We then take the sum of the feature values as well as the average of each node and create the input feature vector with dimension = 64.



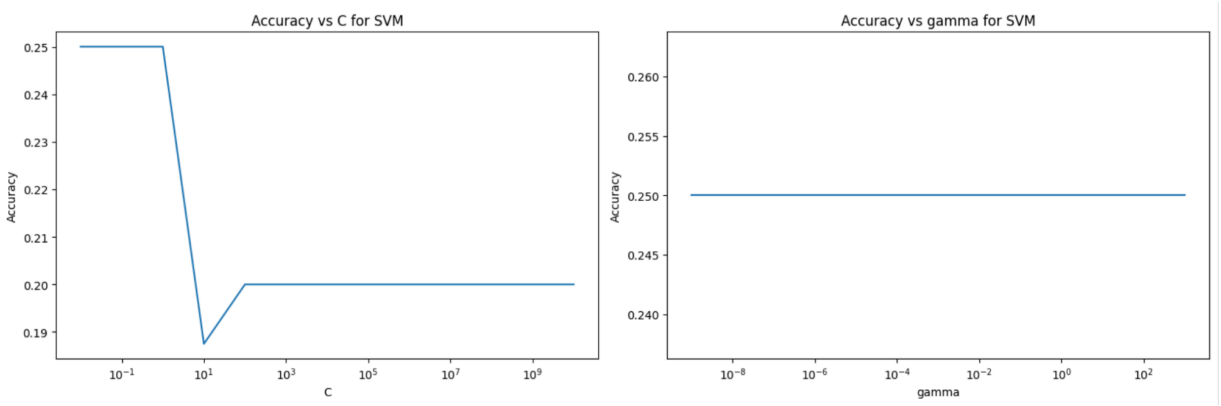
We then stored the feature vectors for each graph constructed from the same kmer value into a CSV file which will later be used as inputs for the different classifiers we are testing.

➢ **SVM Classification**

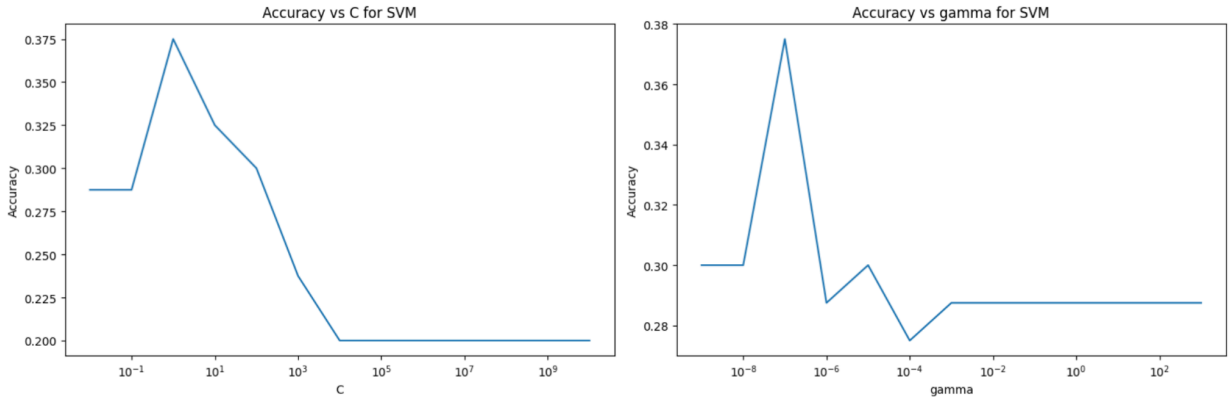
For the SVM classifier, we tested the linear kernel and rbf kernel with a range of different C and gamma values for the parameters. The accuracy result was lower

than expected but it could be due to the reduced sample size as we had limited time to run the node embedding models. We also found out that there is a trend of the accuracy increasing when we test higher Kmer values with the highest being Kmer = 10.

Another discovery while testing the different C and gamma values in the RBF kernel is that the accuracy does not change with the change of gamma value until we increase kmer to 10.



C and gamma value with kmer = 5



C and gamma value with kmer = 10

➢ **Result Analysis**

Although we ended up using a smaller sample size and three different Kmer values, we can still see the accuracy is affected by the Kmer value as well as the

impact on the testing accuracy from changing the parameter value with the RBF kernel while using the same Kmer value.

Kmer Value	Lowest Accuracy	Highest Accuracy
3	16%	27%
5	19%	25%
10	20%	37.5%

❖ **Overlap Graph Construction:**

As part of our project, we have also attempted to adapt an overlap graph in parallel with the De-Bruijn graph construction. In the process of reading the related literature such as Overlap Layout Consensus Assembly[10] from Ben Langmead, we have come across the OLC method which we have adapted as a primary approach to Overlap Graph Construction. The Algorithm:

1. Overlap

Goal: Identify all possible overlaps between the fragments (reads).

Process: Compare every fragment to every other fragment to find regions where they align or overlap. This is usually done using a pairwise alignment algorithm. The length of the overlap must exceed a certain threshold to be considered valid, ensuring reliability.

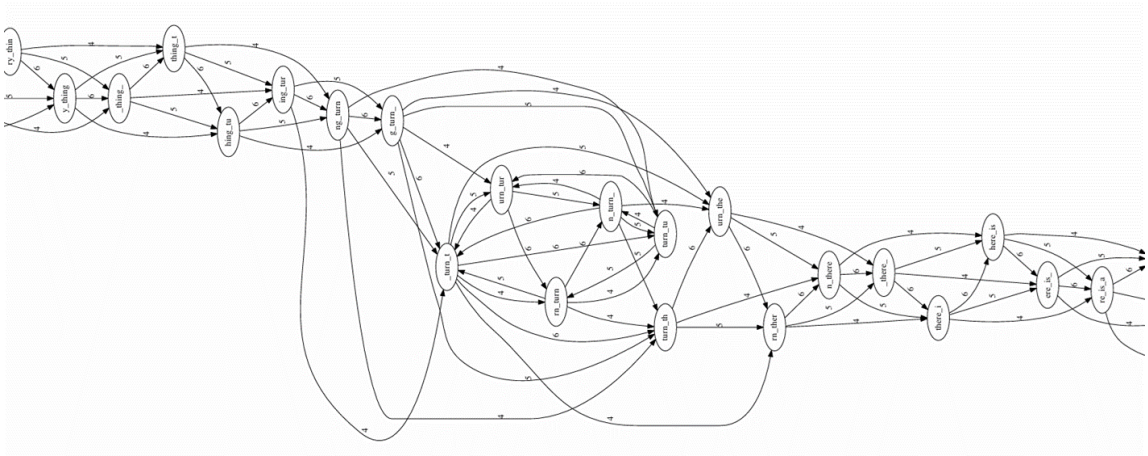
Output: A set of identified overlaps that indicates how reads might extend each other.

2. Layout

Goal: Arrange the reads into a continuous and coherent order based on their overlaps.

Process: Use the overlaps to construct an overlap graph, where each node represents a read, and each edge represents an overlap between two reads.

Output: The layout of the reads in a graph form, often visualized to show connections that help in reconstructing the original sequence.

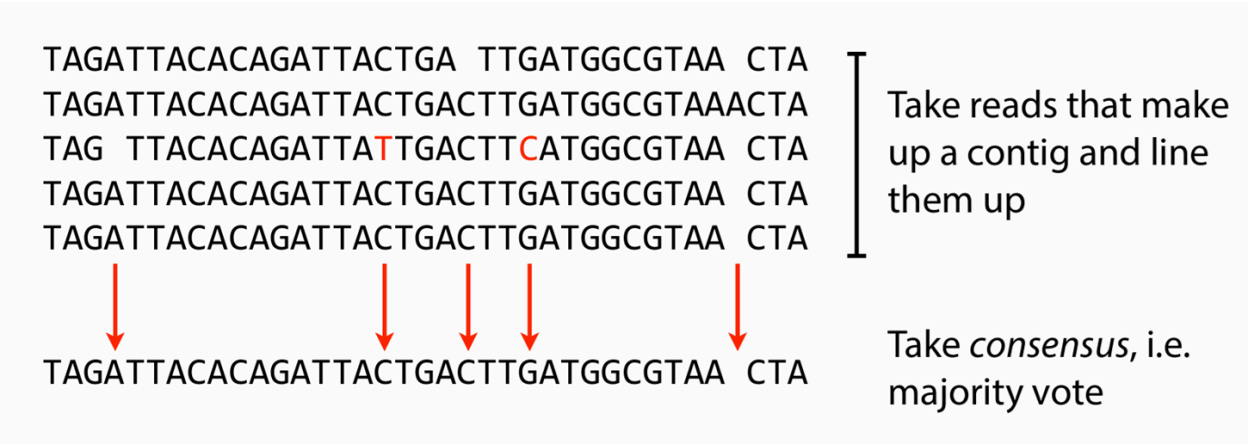


3. Consensus

Goal: Generate a single consensus sequence that best represents the data from all the overlapping reads.

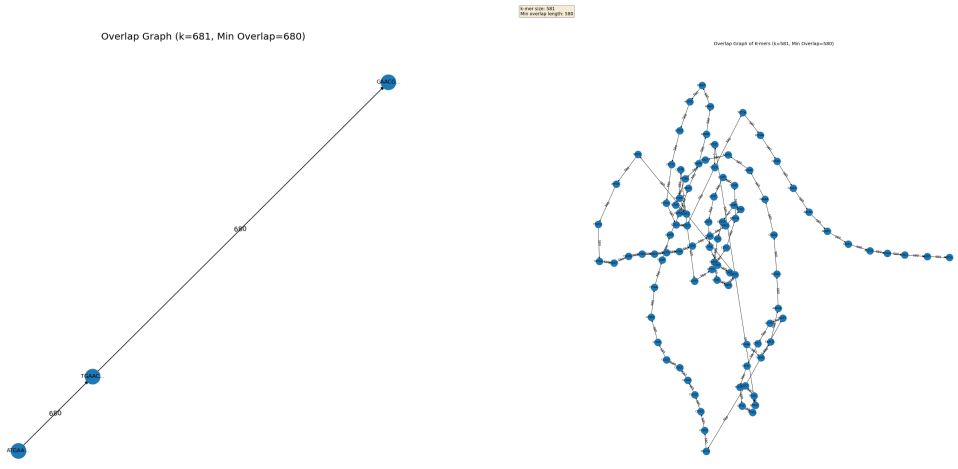
Process: Traverse the overlap graph to merge overlaps and resolve conflicts where different reads suggest different sequences for the same region. This often involves sophisticated algorithms to deal with errors in the reads and to choose the most likely base for each position in the sequence.

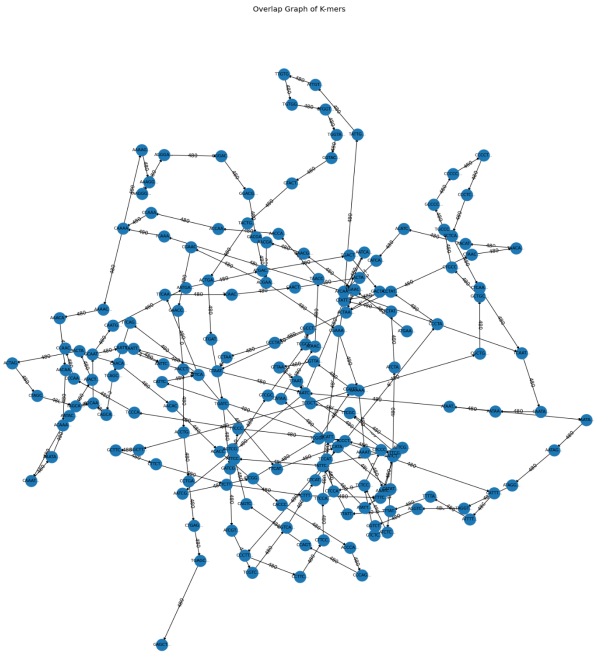
Output: A final, reconstructed sequence that represents the consensus of all the reads, intended to be the closest possible approximation to the original DNA or RNA sequence being sequenced.



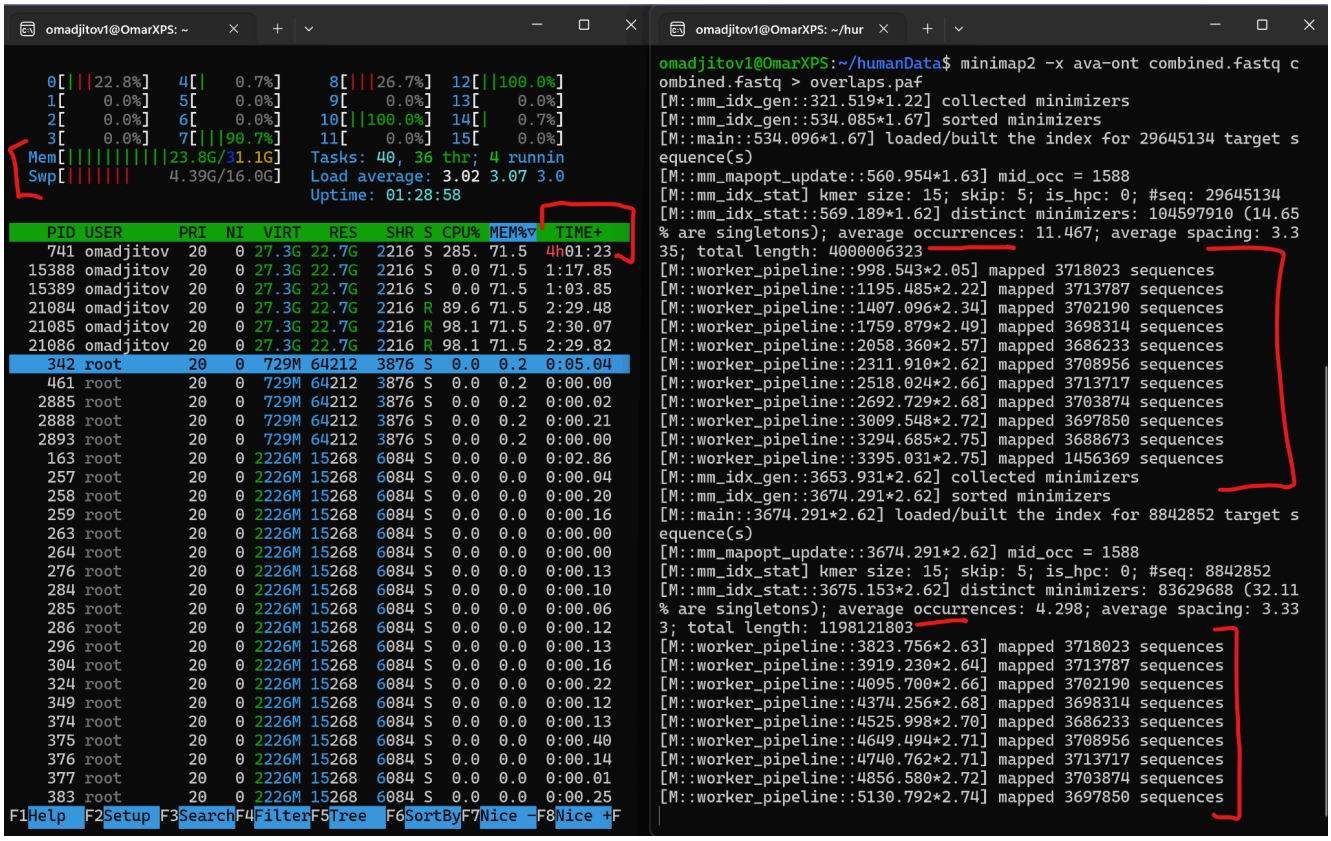
While constructing an overlap graph initial data to be used was the chimpanzee genome dataset stored as a series of sequences in .txt format, which after several experiments proved to be not organized enough to be used for any sort of accurate graph construction, for any existing approach for Overlap graph construction as we discovered in a process of implementation the data must be in .fastq format, our initial dataset has not met this requirement, and even more data was lost the data clean up and conversion.

The following are the initial experimental results of the resulting graphs:



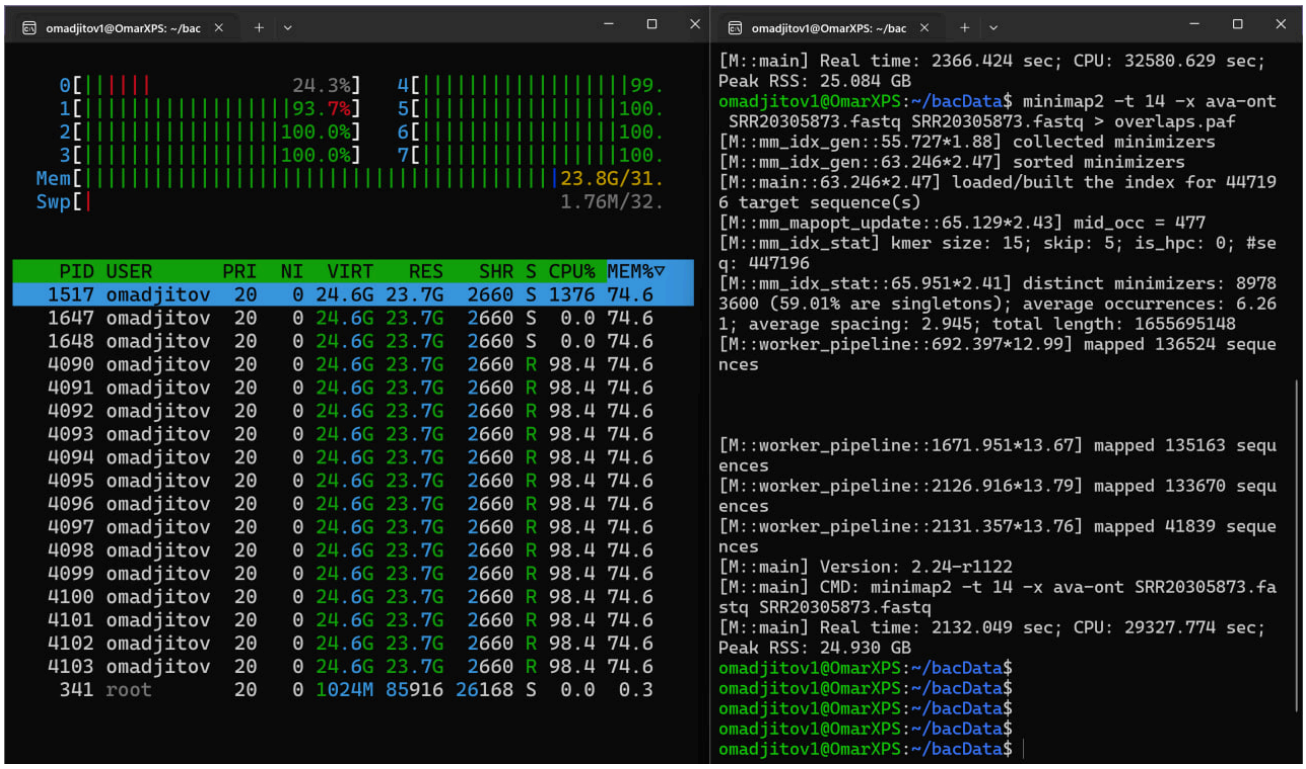


While continuing to follow the research papers that are related to the assembly topic. I didn't realize how we have side skewed from the initial objective of a topic which is sequence classification instead of genome assembly. As a result of this misunderstanding, we ended up spending much time on genome assembly tools such as Celera and Minimap2+Miniasm, as a result of this unnecessary research we ended up spending much time even assembling the Genome of a Fruit Fly. This process turned out to be just as time-consuming. It was decided to first move forward with a better quality human DNA Sequence produced by the Institute of Hydrobiology, Chinese Academy of Sciences [8] to preserve the quality of the project. The Overlap Stage:

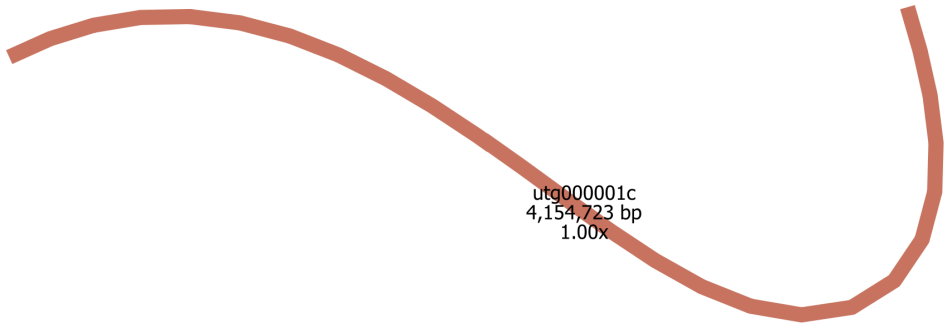


We can see how computationally intensive the overlap graph construction is. The 1st stage of the overlap stage only starts with 29 million sequences to compare meters of the length 15 against has and is still mapping the sequences.

Because of how computationally intensive it was to work with human DNA we started experimenting with the algorithm to enhance the performance of this process. When the Overlap Graph construction algorithm is run on simpler data sets such as simpler DNA sequences, more specifically on *Long read DNA sequence of Bacillus amyloliquefaciens KNU-28: bacteria*[9], although it took considerable time and computations we completely assembled the genome.



We have completely reassembled the Sequence, which when represented as a graph shows 1 continuous node:



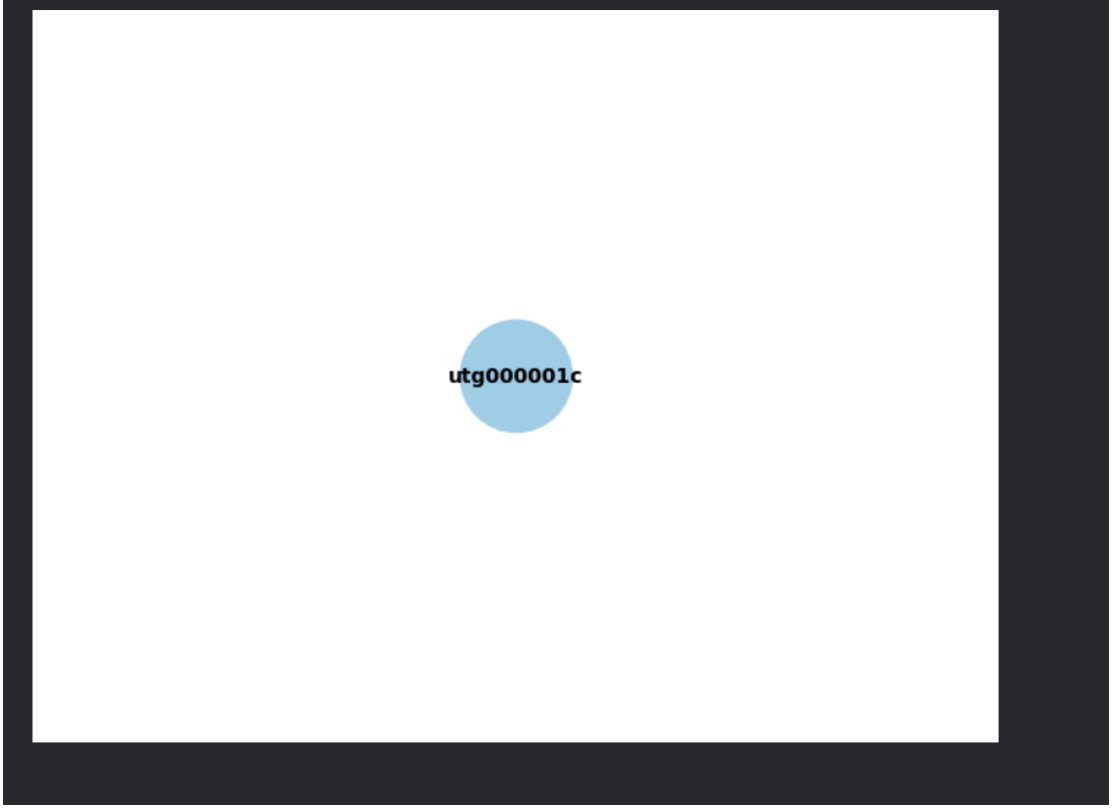
The node contains 4,154,723 base pairs.

When represented using a MatPlotLib, we also see 1 node which confirms our findings, we have reconstructed a sequence:

```
# Load the .gfa file and create the NetworkX graph
# gfa_file = 'D:\HOMEWORK\I. MACHINE LEARNING\PROJECT\GRAPH CONSTRUCTION\Software\Graph Visua
gfa_file = 'D:\HOMEWORK\I. MACHINE LEARNING\PROJECT\GRAPH CONSTRUCTION\data\\bac_assembly.gfa
graph = parse_gfa(gfa_file)

# Visualize the graph using Matplotlib
pos = nx.spring_layout(graph)  # Positions for all nodes
nx.draw(graph, pos, with_labels=True, node_size=3000, node_color='skyblue', font_size=10, fon
plt.show()
```

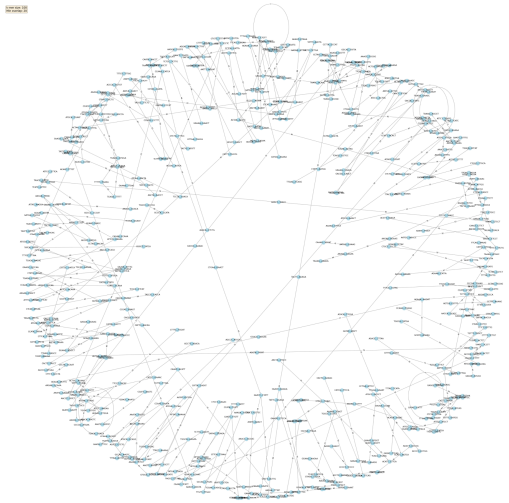
✓ 0.0s

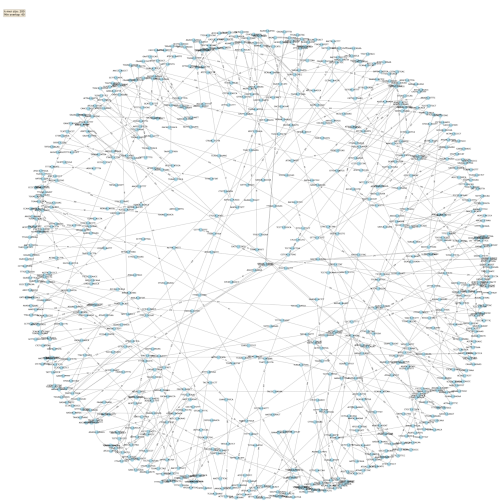


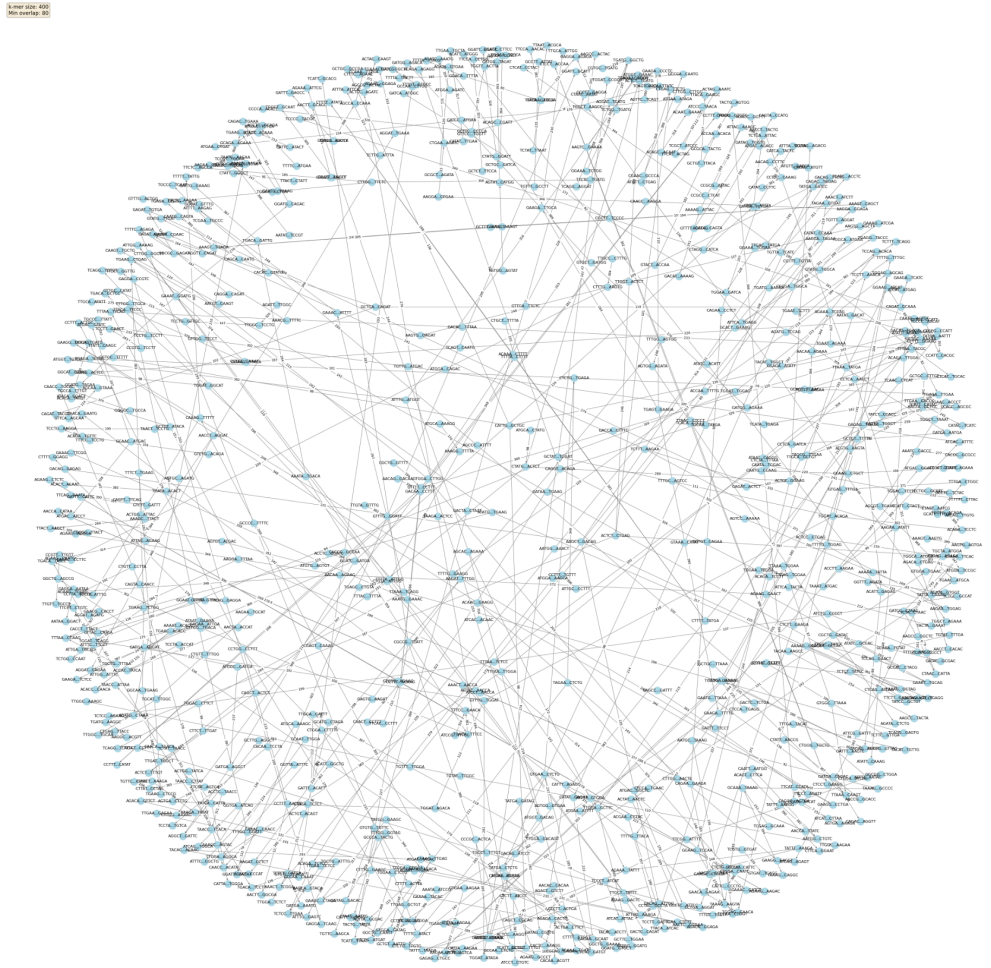
❖ **Error Correction:**

After realizing the mistake we returned to back on track of classification task instead of assembly, where we had to start over completely. From the overlap graph construction, see the graphs:


GGACAGTAATTACCGGTTTCCGCTATAAAACCAAAGCATCCTTAGCACACTTTCTACCCCAAGGCACCCCTCTCCCCCTAATTCCAATACTAGTAGTCATCGAAACTA
CCACAACACAACTCTCTATAAACCTCGGAATAGCAATTCCCCTATGAGCAGGGACAGTAATTACCGGTTTCCGCTATAAAACCAAAGCATCCTTAGCACACTTTCTAC
ACTAATCTACTTGGACTATTACCTCACTCATTTACGCCCACAACACAACTCTCTATAAACCTCGGAATAGCAATTCCCCTATGAGCAGGGACAGTAATTACCGGTTTCC






k-mer size: 400
Min overlap: 80

We have eventually, managed to compose the following code:

```python
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from karateclub import Graph2Vec
import logging
import os

# Setup logging to track the progress and any issues during execution.
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Function to read sequences and their labels from a file, expecting tab-separated values.
def read_sequences_from_file(filename):
    with open(filename, 'r') as file:
        # Read each line, split by tab, and strip whitespace.
        return [line.strip().split('\t') for line in file if line.strip()]

# Function to generate k-mers from sequences.
def chop_into_kmers(sequences, k):
    kmers = []
    # Iterate over each sequence and its associated label.
    for seq, label in sequences:
        # Generate overlapping k-mers for the length of the sequence.
        for i in range(len(seq) - k + 1):
            kmers.append((seq[i:i+k], label))
    return kmers

# Function to find overlapping k-mers based on a minimum overlap length.
def find_overlaps(kmers, min_overlap_length):
    overlaps = []
    # Compare each k-mer with every other k-mer to find overlaps.
    for i, (kmer1, _) in enumerate(kmers):
        for j, (kmer2, _) in enumerate(kmers):
            if i != j:
                # Determine the smallest length between two k-mers.
                length = min(len(kmer1), len(kmer2))
                # Check for overlaps starting from the minimum length down to 1.
                for ol in range(min_overlap_length, length + 1):
                    if kmer1.endswith(kmer2[:ol]):
                        overlaps.append((kmer1, kmer2, ol))
                        break
    return overlaps

# Function to construct a graph from the overlaps of k-mers.
def construct_overlap_graph(overlaps):
    graph = nx.Graph()  # Initialize an undirected graph.
    kmer_to_index = {}
    current_index = 0
    # Create nodes for each unique k-mer and edges for their overlaps.
    for kmer1, kmer2, weight in overlaps:
        if kmer1 not in kmer_to_index:
            kmer_to_index[kmer1] = current_index
            current_index += 1
        if kmer2 not in kmer_to_index:
            kmer_to_index[kmer2] = current_index
            current_index += 1
        graph.add_edge(kmer_to_index[kmer1], kmer_to_index[kmer2], weight=weight)
    return graph

# Function to plot the graph to a PNG file.
def plot_graph(graph, file_name):
    plt.figure(figsize=(10, 10))
    labels = nx.get_node_attributes(graph, 'label')
    pos = nx.spring_layout(graph)
    nx.draw(graph, pos, labels=labels, with_labels=True)
    plt.savefig(file_name)
    plt.close()
```

```python
# Function to embed the graph using the Graph2Vec algorithm.
def graph2vec_embedding(graph):
    graph2vec = Graph2Vec(dimensions=64)  # Initialize with 64 dimensions.
    graph2vec.fit([graph])
    return graph2vec.get_embedding()

# Function to export the graph structure to a GraphML file.
def export_graph(graph, export_file_name):
    nx.write_graphml(graph, export_file_name)

# Main processing function to handle k-mer processing and classification.
def process_kmers(filename, k, min_overlap_values):
    logging.info(f"Processing k={k} with min_overlap_values: {min_overlap_values}")
    results = {}
    sequences = read_sequences_from_file(filename)
    kmers = chop_into_kmers(sequences, k)
    result_path = f'results_k{k}'
    if not os.path.exists(result_path):
        os.makedirs(result_path)

    for min_overlap in min_overlap_values:
        overlaps = find_overlaps(kmers, min_overlap)
        graph = construct_overlap_graph(overlaps)
        graph_file_name = f'{result_path}/overlap_graph_k{k}_min{min_overlap}.png'
        plot_graph(graph, graph_file_name)
        export_graph(graph, f'{result_path}/graph_k{k}_min{min_overlap}.graphml')

        embeddings = graph2vec_embedding(graph).squeeze()
        embeddings_df = pd.DataFrame([embeddings], index=['Graph'])
        embeddings_csv = f'{result_path}/embeddings_k{k}_min{min_overlap}.csv'
        embeddings_df.to_csv(embeddings_csv)

        y = pd.Series([k[:5]+'...'+k[-5:] label for k, label in kmers])
        X = embeddings_df
        y = pd.Series(y.iloc[0])
        if len(y) < 2:
            logging.warning(f"Not enough samples to train and test for k={k} and min_overlap={min_overlap}. Need at least 2 data points.")
            continue
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

        classifier = svm.SVC()
        classifier.fit(X_train, y_train)
        y_pred = classifier.predict(X_test)

        report = classification_report(y_test, y_pred, output_dict=True)
        accuracy = accuracy_score(y_test, y_pred)
        results[(k, min_overlap)] = {'report': report, 'accuracy': accuracy}
        logging.info(f"Completed min_overlap={min_overlap} for k={k}")

    return results

# Example usage for each k value separately to manage large datasets or long processing times
filename = 'hum_med_2.txt'
k_values = [100]  # Different k-mer lengths to try
min_overlap_values = [50]  # Different minimum overlaps to try
for k in k_values:
    results = process_kmers(filename, k, min_overlap_values)
    logging.info(f"Results for k={k}: {results}")
```

Here's the breakdown of the code in Detail:

●Read Sequences and Create k-mers:

read_sequences_from_file: Extracts sequences and their associated classes from a tab-delimited file.

chop_into_kmers: Chops these sequences into k-mers of specified length k, retaining the associated class label for each k-mer.

●Detect Overlaps Among K-mers:

find_overlaps: Checks each pair of k-mers for overlaps where the suffix of one k-mer matches the prefix of another. This overlap must meet a specified minimum length to be considered significant.

●Construct the Overlap Graph:

construct_overlap_graph: Creates a graph using NetworkX where each node represents a unique k-mer, and edges represent detected overlaps. The edge weights are determined by the length of the overlap.

●Graph Visualization:

plot_graph: Visualizes the constructed graph and saves the visualization to a file.

●Feature Extraction via Graph Embedding:

graph2vec_embedding: Converts the graph structure into a fixed-size feature vector using the Graph2Vec algorithm, facilitating the use of graph data in machine learning models.

●**Export Graph:**

export_graph: Saves the graph in a GraphML format for further analysis or use outside the Python environment.

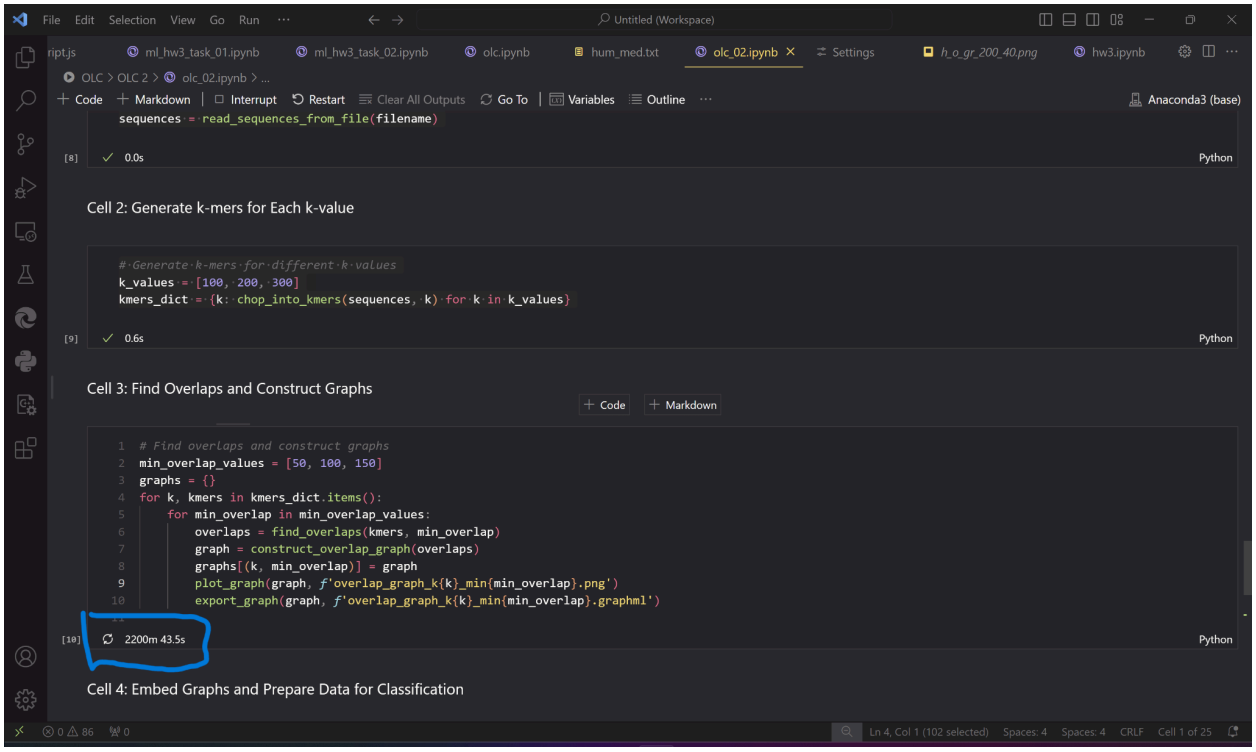
●**Classification:**

The k-mer sequences are prepared for classification by associating each k-mer with its original sequence label. The embeddings from the graph are used as features in a Support Vector Machine (SVM) classifier. The dataset is split into training and testing sets, the SVM model is trained, and its performance is evaluated using classification reports and accuracy metrics.

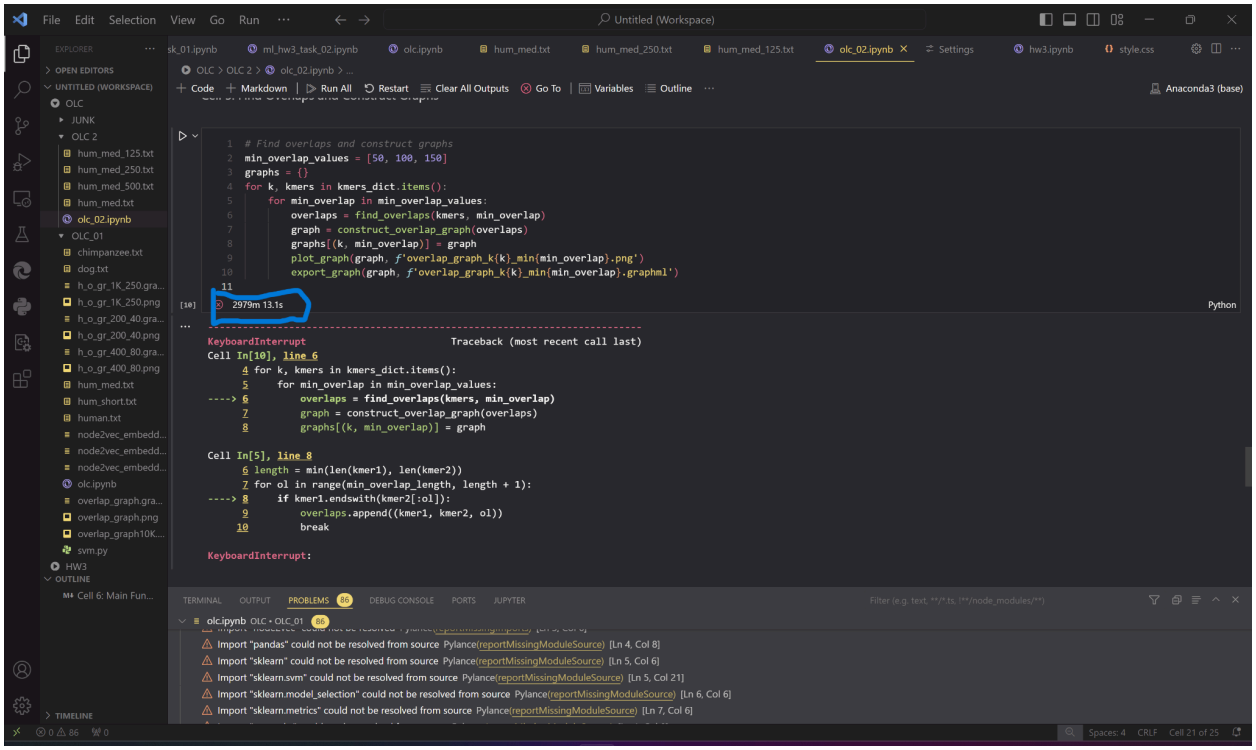
●**Process Integration and Logging:**

process_kmers: Integrates all previous steps into a single function that processes k-mers for different values of k and minimum overlap lengths. Results for each configuration are logged and returned. Incorporates logging throughout to track progress and debug issues.

Unfortunately, we do not possess adequate powers to run this code, you can see that it takes ~50 hours to construct the overlap graph even with a reduced data size of under 1000 strands.



At this stage, the code has been running for ~36 hours



At this stage, the code has been running for ~50 hours

Unfortunately, we didn't have enough time to complete running the full model, by the end of the set deadline, due to insufficient computational capabilities we possessed.

Additional Work:

Since we were provided some additional time before the submission we have worked and improved our existing code that utilizes the overlap graph (both initial and upgraded codes will be attached as .ipynb files).

Improved code:

SELECT PROPER FILENAME K VALUE AND MIN_OVERLAP VALUES

```python
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from node2vec import Node2Vec
import logging
import os

# Set up logging to help with debugging and tracking the progress of the program.
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def read_sequences_from_file(filename):
    """Read DNA sequences and their associated class labels from a file.

    Args:
        filename (str): The path to the file containing sequences and labels, separated by tabs.

    Returns:
        list of tuples: A list where each tuple contains a sequence and its corresponding label.
    """
    with open(filename, 'r') as file:
        return [line.strip().split('\t') for line in file if line.strip()]

def chop_into_kmers(sequences, k):
    """Break down each sequence into k-mers of specified length.

    Args:
        sequences (list): A list of tuples containing sequences and their labels.
        k (int): The length of each k-mer.

    Returns:
        list of tuples: Each tuple contains a k-mer and its associated label.
    """
    return [(seq[i:i+k], label) for seq, label in sequences for i in range(len(seq) - k + 1)]

def find_overlaps(kmers, min_overlap_length):
    """Identify all k-mers that overlap by at least the minimum specified length.

    Args:
        kmers (list): A list of k-mers.
        min_overlap_length (int): Minimum length of overlap.

    Returns:
        list of tuples: Each tuple represents an overlap between two k-mers, including the overlap size.
    """
    overlaps = []
    for i, (kmer1, _) in enumerate(kmers):
        for j, (kmer2, _) in enumerate(kmers):
            if i != j:
                length = min(len(kmer1), len(kmer2))
                for ol in range(min_overlap_length, length + 1):
                    if kmer1.endswith(kmer2[:ol]):
                        overlaps.append((kmer1, kmer2, ol))
                        break
    return overlaps

def construct_overlap_graph(overlaps):
    """Construct a graph where each node represents a k-mer and edges represent overlaps.

    Args:
        overlaps (list): A list of overlaps between k-mers.

    Returns:
        networkx.Graph: A graph object with k-mers as nodes connected by their overlaps.
    """
    graph = nx.Graph()
    for kmer1, kmer2, weight in overlaps:
        graph.add_edge(kmer1, kmer2, weight=weight)
    return graph
```

```python
def plot_graph(graph, file_name):
    """Visualize the overlap graph using Matplotlib.

    Args:
        graph (networkx.Graph): The graph to visualize.
        file_name (str): Path to save the visualization.
    """
    plt.figure(figsize=(10, 10))
    pos = nx.spring_layout(graph, scale=2, k=1/(graph.order()**0.5)*2)
    labels = {node: node[:5] + '...' + node[-5:] for node in graph.nodes()}
    nx.draw(graph, pos, labels=labels, with_labels=True, node_size=50, font_size=8)
    plt.savefig(file_name)
    plt.close()

def node2vec_embedding(graph):
    """Embed graph nodes into a vector space using the Node2Vec algorithm.

    Args:
        graph (networkx.Graph): The graph to embed.

    Returns:
        gensim.models.Word2Vec: A trained Node2Vec model, or None if the graph is empty.
    """
    if graph.number_of_nodes() == 0:
        logging.warning("Graph is empty. Skipping embedding.")
        return None
    node2vec = Node2Vec(graph, dimensions=64, walk_length=30, num_walks=200, workers=4)
    return node2vec.fit(window=10, min_count=1, batch_words=4)

def export_graph(graph, export_file_name):
    """Export the graph to a GraphML file format.

    Args:
        graph (networkx.Graph): The graph to export.
        export_file_name (str): Path where the GraphML file will be saved.
    """
    nx.write_graphml(graph, export_file_name)

def evaluate_graph(graph):
    """Calculate and log key graph metrics to assess its structure and connectivity.

    Args:
        graph (networkx.Graph): The graph to evaluate.

    Returns:
        dict: Contains the density, number of connected components, and average clustering coefficient.
    """
    density = nx.density(graph)
    num_connected_components = nx.number_connected_components(graph)
    avg_clustering_coefficient = nx.average_clustering(graph)
    logging.info(f"Graph Density: {density}, Connected Components: {num_connected_components}, Average Clustering Coefficient: {avg_clustering_coefficient}")
    return {'density': density, 'num_components': num_connected_components, 'avg_clustering': avg_clustering_coefficient}
```

```python
def process_kmers(filename, k, min_overlap_values):
    """Main function to process k-mers, construct graphs, and perform classification based on the Node2Vec embeddings.

    Args:
        filename (str): Path to the DNA sequences file.
        k (int): Length of each k-mer.
        min_overlap_values (list): List of minimum overlap lengths to consider.

    Returns:
        dict: Contains classification reports and accuracy for each combination of k and minimum overlap.
    """
    base_filename = os.path.splitext(os.path.basename(filename))[0]
    results = {}
    sequences = read_sequences_from_file(filename)
    kmers = chop_into_kmers(sequences, k)
    kmer_dict = {kmer: label for kmer, label in kmers}

    for min_overlap in min_overlap_values:
        if min_overlap > k:
            logging.warning(f"Skipping min_overlap={min_overlap} as it is greater than k={k}")
            continue
        overlaps = find_overlaps(kmers, min_overlap)
        graph = construct_overlap_graph(overlaps)
        if not graph.number_of_edges():
            logging.warning(f"Graph is empty for k={k} and min_overlap={min_overlap}. Skipping.")
            continue

        graph_metrics = evaluate_graph(graph)
        graph_dir = f'graphs/{base_filename}_k{k}_min{min_overlap}'
        os.makedirs(graph_dir, exist_ok=True)
        graph_file_name = f'{graph_dir}/overlap_graph_k{k}_min{min_overlap}.png'
        plot_graph(graph, graph_file_name)
        export_graph(graph, f'{graph_dir}/graph_k{k}_min{min_overlap}.graphml')

        model = node2vec_embedding(graph)
        if model is None:
            continue
        embeddings = model.wv.vectors
        node_ids = model.wv.index_to_key
        embeddings_df = pd.DataFrame(embeddings, index=node_ids)

        y = pd.Series({node: kmer_dict[node] for node in node_ids if node in kmer_dict})
        X = embeddings_df
        y = y.reindex(X.index)  # Ensure y is aligned with X's index

        if y.nunique() < 2:
            logging.warning(f"Not enough classes to train SVM for k={k} and min_overlap={min_overlap}. Need at least 2 classes.")
            continue

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
        classifier = svm.SVC()
        classifier.fit(X_train, y_train)
        y_pred = classifier.predict(X_test)

        report = classification_report(y_test, y_pred, output_dict=True)
        accuracy = accuracy_score(y_test, y_pred)
        results[(k, min_overlap)] = {'report': report, 'accuracy': accuracy, 'graph_metrics': graph_metrics}
        logging.info(f"Completed min_overlap={min_overlap} for k={k}")

    return results
```

```
filename = 'input_data/hum_med_2.txt'
k_values = [150, 250, 350]
min_overlap_values = [50, 100, 149]
for k in k_values:
    results = process_kmers(filename, k, min_overlap_values)
    logging.info(f"Results for k={k}: {results}")

# Print the results
for key, value in results.items():
    print(f"Results for k={key[0]} and min_overlap={key[1]}:")
    print(f"Accuracy: {value['accuracy']}")
    print("Classification Report:")
    print(value['report'])
    print("Graph Metrics:")
    print(value['graph_metrics']['density'], value['graph_metrics']['num_components'], value['graph_metrics']['avg_clustering'])
```

Key Differences and Improvements

1. **Logging Integration**:
    - **Original Version:** Did not include any logging, which can limit debuggability and traceability of the code execution.
    - **Improved Version:** I integrated Python's logging module, which allows tracking the execution process and debugging more effectively. This is especially useful in production environments or during further development to understand flow and errors.
2. **Function Enhancements**:
    - **Original Version:** Functions are straightforward but lack efficiency in handling large datasets or minimizing repetitive computations.
    - **Improved Version:** I optimized the chop_into_kmers function using a list comprehension for better performance. Additionally, I enhanced overlap detection logic to exit early once a sufficient overlap is found, reducing unnecessary comparisons.
3. **Graph Construction**:
    - **Original Version:** The graph construction was somewhat basic, mapping k-mers to unique indices but not ensuring efficient memory usage or scalability.
    - **Improved Version:** In constructing the overlap graph, I retained the efficient mapping but streamlined the addition of edges and nodes to ensure it scales better with larger datasets. I also included explicit logging at this stage to notify when a graph has insufficient data, enhancing the robustness of the dataset handling.

4. **Graph Analysis and Exporting**:
    - **Original Version:** Lacked detailed analysis or exporting functionality which could limit the utility in real-world applications.
    - **Improved Version:** I added a function, evaluate_graph, to compute and log various graph metrics like density and clustering coefficient, providing insights into the graph structure. This is critical for understanding the quality of the graph in terms of connectivity and clustering properties.
5. **Node2Vec Integration**:
    - **Original Version:** Utilized Graph2Vec without considerations for empty graphs or alignment issues between node embeddings and labels.
    - **Improved Version:** I switched to Node2Vec for more customizable node embedding and handled cases where the graph might be empty. This flexibility allows for better tuning according to specific characteristics of the data. I ensured that the node labels align correctly with their embeddings, which is crucial for accurate classification.
6. **Error Handling and Data Alignment**:
    - **Original Version:** Did not focus on error handling or data misalignment, which could lead to failures or incorrect results during classification.
    - **Improved Version:** I implemented checks to ensure there are at least two classes before proceeding with SVM training, and aligned labels directly with embeddings to prevent misclassification due to misaligned data.
7. **Scalable and Organized File Management**:
    - **Original Version:** Simplistic handling of output files.
    - **Improved Version:** I introduced organized directory structures for saving graphs and results based on k-mer and overlap values, which simplifies managing outputs from multiple runs or datasets.

Overlap Graph Conclusions.

The improvements I implemented are designed to enhance the scalability, robustness, and utility of the sequence classification using overlap graphs. By integrating better logging, optimizing functions, adding graph evaluations, and ensuring error handling and data alignment, the script is

now more suitable for larger datasets and provides more insights into the generated models and graphs.

- **Conclusions:** As a result of our work we have come to the following conclusions:
  - What are overlap graphs and how to construct them?
  - Overlap-Layout-Consensus is a very effective algorithm to assemble a genome when written in C lang.
  - The difference between Genome assembly and Sequence Classification.
  - Minimap2 + Miniasm is another very efficient approach to assembling a genome for long reads.
  - Although it's probably feasible, but not very sensible to use an overlap graph for a problem of sequence classification. It's worth considering using the De-Bruijn Graph for such problems.
  - Python is slow. One may consider using faster languages such as C, to
  - Implement the algorithms that work with large datasets, such as DNA.

Timeline:

- **Phase 1** (2/18 - 3/8) - Literature review on de-Bruijn graph model, Node2vec and Graph2vec embeddings as well as SVM and NN classifiers. (Complete)
- **Phase 2** (3/9 - 31) - Construction of the de-Bruijn graph with different kmer values for both datasets and construction of the Overlap Graph using the Overlap-Layout-Consensus algorithm (Finished construction of de-Bruijn graphs with different kmers, in progress with graph construction with overlap graph)
- **Phase 3** (3/31 - 4/10)
  - Implementation of Node2vec and Graph2vec model with SVM, and NN for both datasets and testing different parameters. (in-progress with Node2vec embedding with de-Bruijn graphs)

- **Phase 4** (4/10 - 14) - Trying the combination of de-Bruijn graph construction and models with classifiers for final comparison.

❖ Work Distribution

- Alenka: de-Bruijn graph construction for both datasets, implementation of Node2vec embedding for graph features then testing with SVM.
- Omar: Overlap Graph for both datasets, perform feature extraction for the machine learning mode, implementation of Graph2vec with SVM testing different parameters.

❖ Supplement Materials

[1] Compeau, P., Pevzner, P. & Tesler, G. How to apply de Bruijn graphs to genome assembly. *Nat Biotechnol* 29, 987–991 (2011). https://doi.org/10.1038/nbt.2023

[2] Nagesh Singh Chauhan. [n. d.]. Demystify DNA Sequencing with Machine Learning|https://www.kaggle.com/code/nageshsingh/demystify-dna-sequencing-with-machine-learning/notebook

[3] Ali, Sarwan, Babatunde Bello, Prakash Chourasia, Ria Thazhe Punathil, Yijing Zhou, and Murray Patterson. 2022. "PWM2Vec: An Efficient Embedding Approach for Viral Host Specification from Coronavirus Spike Sequences" *Biology* 11, no. 3: 418. https://doi.org/10.3390/biology11030418

[4] Kiril Kuzmin, Ayotomiwa Ezekiel Adeniyi, Arthur Kevin DaSouza, Deuk Lim, Huyen Nguyen, Nuria Ramirez Molina, Lanqiao Xiong, Irene T. Weber, and Robert W. Harrison. 2020. Machine learning methods accurately predict the host specificity of coronaviruses based on spike sequences alone. Biochemical and Biophysical Research Communications 533 (12 2020), 553–558. Issue 3.https://doi.org/10.1016/j.bbrc.2020.09.010

[5] Rizzi, R., Beretta, S., Patterson, M. et al. Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era. Quant Biol 7, 278–292 (2019). https://doi.org/10.1007/s40484-019-0181-x

[6] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, Bicheng Yang, Wei Fan, Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-Bruijn-graph, *Briefings in Functional Genomics*, Volume 11, Issue 1, January 2012, Pages 25–37, https://doi.org/10.1093/bfgp/elr035

[7] Neri Van Otten, Node2Vec: Extensive Guide & How To In Python, https://spotintelligence.com/2024/01/18/node2vec/

[8] Institute of Hydrobiology, Chinese Academy of Sciences (Institute of Hydrobiology, Chinese Academy of Science). DNA sequence of one male, one female, and 48 females. (SRR5119618)

https://www.ncbi.nlm.nih.gov/sra/SRX2425041[accn]

[9] Kyungpook National University, Bacillus amyloliquefaciens strain: KNU-28 Genome sequencing

https://www.ncbi.nlm.nih.gov/sra/SRX16339329[accn]

[10] Ben Langmead, "Overlap Layout Consensus assembly," Langmead Lab, Accessed 20 Apr. 2024.

assembly_olc.pdf (jhu.edu)

Overlap Graph Github:

workoholyguy/Sequence-Classification-Machine-Learning: Seqeunce Classification Algorithm (github.com)