Assignment 1: Getting Started with NetworkX

Spring 2025

Overview

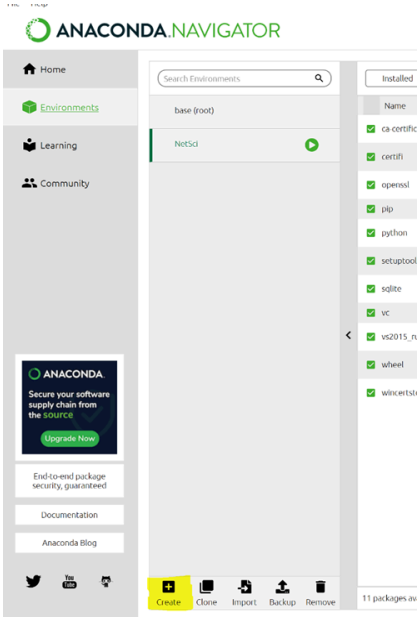
The objective of this assignment is to provide an introduction to working with NetworkX and review some linear algebra concepts. By the end of this assignment you will able to:

- Generate graphs with NetworkX.
- Import datasets for analysis.
- Understand techniques for working with undirected, directed, unconnected, acyclic, and bipartite graphs.
- Begin applying linear algebra to your analysis.

Getting Started

In this section we will provide a quick start tutorial for setting up your environment. **You are not required to use Anaconda**, but it is recommended if you are not familiar with using Jupyter. If you are already familiar with Jupyter notebooks, you can skip to the required packages section and install for your preferred environment set up.

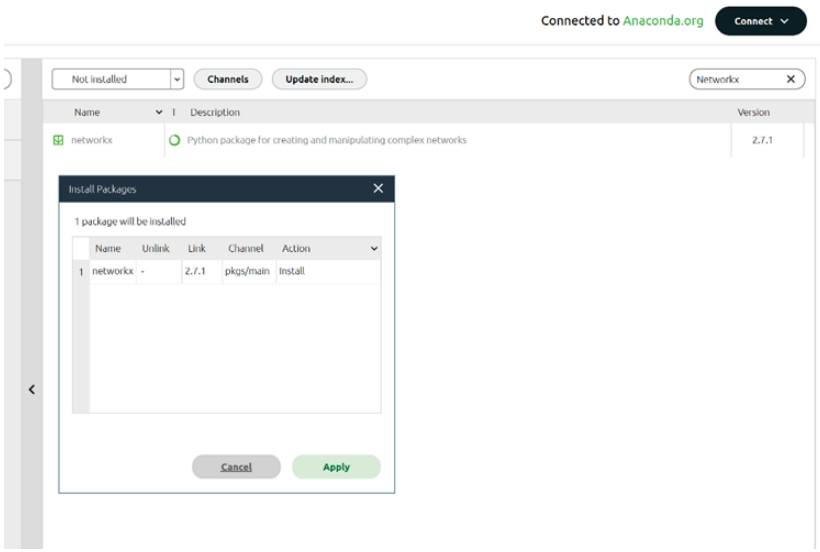
Installing Jupyter - Quick Start



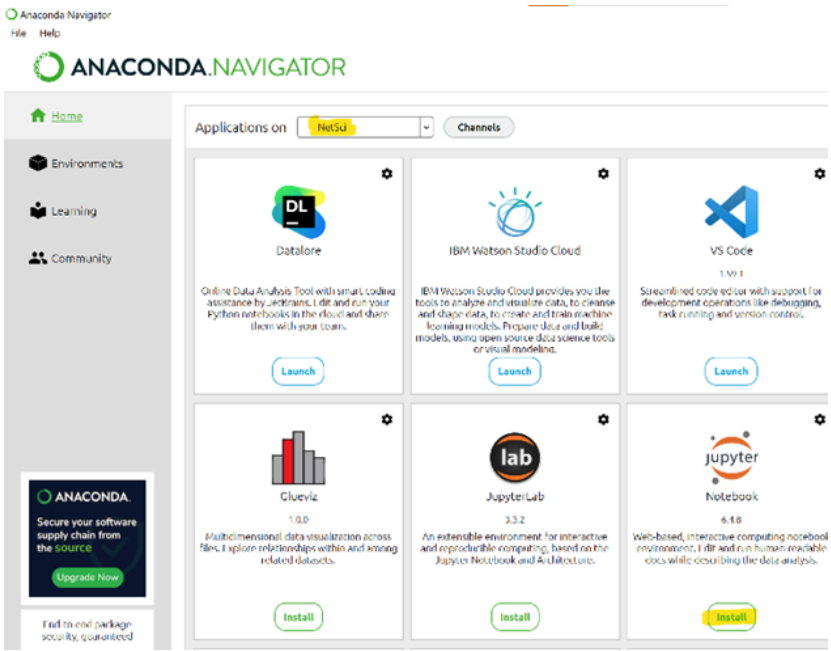
The simplest way to use Jupyter notebooks is through Anaconda. If you do not have Anaconda already installed on your machine, go to https://www.anaconda.com/ and select the installation for your machine. Installing Anaconda ensures that most of the libraries you will need come pre-loaded.

Once the executable is downloaded, follow the installation prompts and use the default settings.

Next use the search bar to find and launch the Anaconda Navigator. Once the Navigator launches, you'll want to navigate to the environments and create one for this class. (Or use the base if you prefer.)

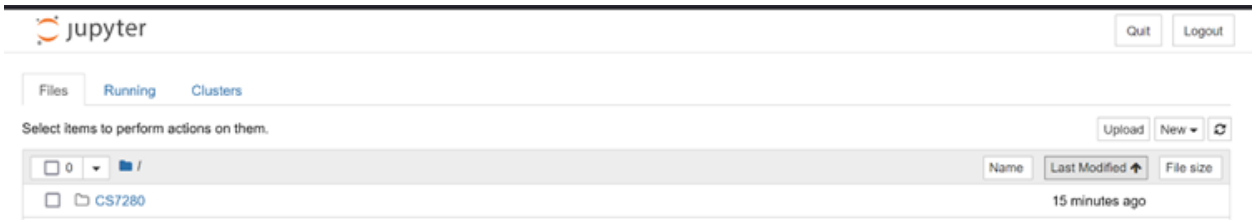


Next, verify that you have the necessary packages installed. Most of them will come pre-installed with Anaconda, but you will need to install NetworkX manually.



After the packages are installed, navigate back to the home page and install then launch Jupyter Notebook.

This will launch the notebook locally in your browser where you can upload and read files from your directory.



From here, you can upload the assignment file and open the .ipynb to begin the assignment.

Required Packages

The assignments are currently validated with python 3.8. If you do not have a python distribution set up already, we recommend installing Anaconda (or miniconda) with python 3.8 or higher.

If you are not using Anaconda, you will need to ensure you have the necessary packages installed. Some of the recommended libraries are:

- networkx (required) **>=v3.0 required**
- numpy (required)
- scipy (required)
- matplotlib (Optional, you can use any plotting library of your choice.)
- Seaborn (Optional, you can use any plotting library of your choice.)
- pandas (Optional)

Submission

Please ensure when submitting that you also include your requirements.txt so that we may replicate your Python dependencies to run your code as needed.

With Anaconda, you can do this by running:
conda list -e > requirements.txt

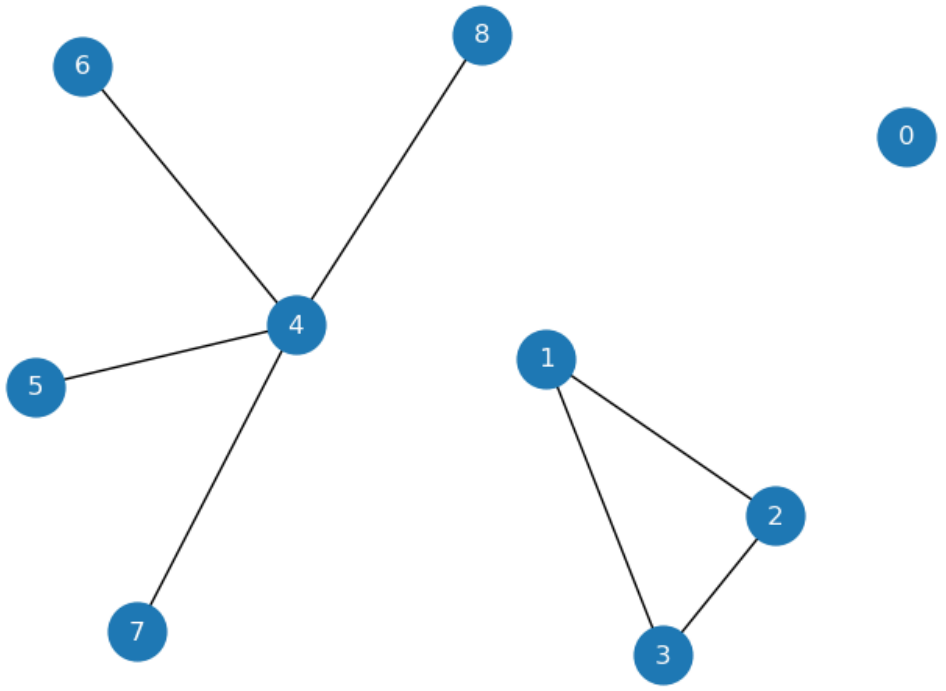
Ensure all graphs and plots are properly labeled with unit labels and titles for x & y axes. Producing readable, interpretable graphics is part of the grade as it indicates understanding of the content – **there may be point deductions if plots are not properly labeled.**

Part 1- Intro to NetworkX

[20 points]

Generating graphs with NetworkX [10 points]

1. [5 points] First, familiarize yourself with the process of creating a graph structure in NetworkX. Be sure to include **readable labels**, ensuring each node has the same structure as in the image. Using the function ***practice_graphs()*** recreate the graph shown below.
   Note: Your layout does not have to match to picture below exactly. That is, as long as there is an edge between 1 and 2, 2 and 3, 3 and 1, etc. that is acceptable, even if they are in different spots on the canvas from what is depicted below.



   Hint: adjust the distance between nodes using ***pos=nx.spring_layout(G, k=None)*** *and setting the* ***k*** *parameter.*

2. [5 points] Use the function ***create_toy_graphs()*** to create the following three 10-node toy networks using the graph generators in NetworkX.
   a. Cycle network
   b. Clique (complete) network

c. Star network

Then, in the calculate_leading_eigenvalue(G) function, compute the leading eigenvalue of an arbitrary graph G. You will use this function to find the leading eigenvalue of the 3 graphs you created in create_toy_graphs().
In an markdown cell below, answer the question, "What is the relationship between the eigenvalue of the adjacency matrix, the maximum degree, and average degree of each network?"
Hint: **`linalg.eig()`***of Numpy or* **`to_numpy_array()`** *of NetworkX*

Importing Map Data [10 points]

The first data set you will need to import is a graph that lists cities as nodes and the distance between the cities as edge weights.

3. [5 points] Using the ***load_cities_data()*** function, import the **cities_data.graphml** data set using the appropriate function and parse it into a Graph object *G*. Use the NetworkX documentation to find the correct function.

   To see what the node and edges represent, you can use G.nodes(data=True) and G.edges(data=True). *Note that this is just to help you view the data and you may clear the outputs once you are finished.*

   Then answer the following questions about **cities_data**:
   a. How many nodes are there? Use the ***find_number_of_nodes(G)*** function to find this.
   b. How many edges? Use the ***find_number_of_edges(G)*** function to find this.
   c. How many pairs of cities are less than **50 miles** apart? Use ***cities_within_50(G, city_list)*** to find this.

4. [5 points] Now we want to implement a function that returns a subgraph of input cities and all directly neighboring cities that are less than **100 miles** from the input cities. Use the pseudocode below to implement the function.

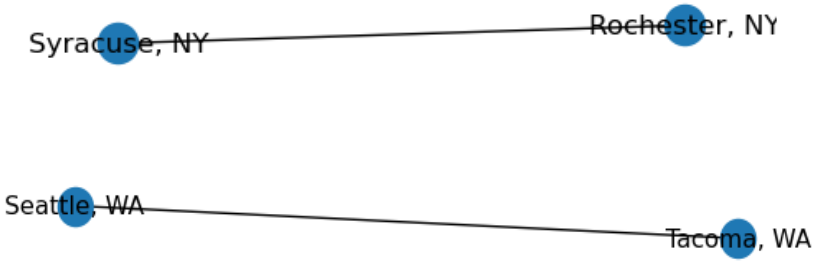
```
def cities_within_100(G, city_list):
# Input
# G: graph from cities_data
# city_list: list of strings (names of cities in G)
```

```
# Output
# S: subgraph of G that only contains edges between cities in
# "city_list" and directly neighboring cities that are less than 100 # miles away
return S
```

Once you have implemented the function, you may test it by running:

```
cities_within_100(G, ["Rochester, NY","Tacoma, WA"])
```

Which should give you the following graphs:



Now run the code for the following list of cities and save the output. Again, make sure the labels are readable.

```
cities_within_100(G, [ "Toledo, OH", "Stockton, CA", "San Francisco, CA" ])
```

Part 2 - Random Walks of Les Misérables

[20 points]

Next we are going to be looking at an undirected weighted network of character co-appearances in Victor Hugo's novel "Les Miserables'' using **"lesmis_data.gml"**. The nodes represent characters (as indicated by the labels) and the edges connect pairs of characters that appear together in the same chapter. The weights on the edges represent the number of times those two characters are seen together. Import the graph, then answer the following questions.

1. [2 points] Using NetworkX built in methods, load **lesmis_data.gml** into a networkx graph object in the ***load_lesmis_data()*** function.
2. [2 points] Use a Networkx method to detemine is the graph connected or unconnected? Use the ***lesmis_connected()*** function.
3. [8 points] Next, you will use the function ***calculate_shortest_paths(G)*** to find the unweighted shortest paths between all node pairs in the graph. Then in the function ***plot_shortest_paths(G)*** you will create a **histogram** or **barplot** to show the distribution of shortest paths you calculated from calculate_shortest_paths(G). Finally, use the ***average_shortest_path_length(G)*** and ***maximum_shortest_path(G)*** functions to find the average of all shortest paths between node pairs in G and the maximum shortest path between all node pairs in G, respectively.
4. [8 points] Next, use the concept of a random walk to find stationary distribution of the network. For this network, a random walk would be meeting one character, then randomly moving to meet another character using the edge weights as the probability of moving along that edge). More mathematically, if you are currently at node u and (u,v) is an edge with node v, then:

   Probability of visiting v from u = The weight of edge(u,v)/sum of all weights adjacent to u

   This means that if the total weight of all edges of Valjean is 158 and the weight of the edge between Cosette and Valjean is 38. In this case, the probability of moving to Cosette in the next step is 38/158.

   Then using the stationary distribution, **find the names of the three most commonly mentioned characters in the novel using the *popular_characters(G)* function.** *No points will be awarded for answers calculated without using the stationary distribution.*

   Hint: There are two different methods for calculating the stationary distribution. Whichever method you use, it is recommended that you test your code on a smaller graph with a known solution as some library functions don't work as students expect and cause them to lose points. See Ed Discussion for more information on each method and how to test your code.

Part 3 - Components of Drosophila Optic Medulla

[20 points]

This third data set, "**drosophila_medulla_data.graphml**", is a directed dataset of synapses among neurons in the drosophila optic medulla. The nodes represent neurons and the edges

are chemical synapses from a neuron to the other. Import this data set and answer the following questions:

1. [2 points] Using NetworkX built-in methods, load **drosophila_medulla_data.graphml** into a networkx graph object in the ***load*_drosophila_medulla_*data()*** function.
2. [4 points] Using the connected components functions in NetworkX, find the weakly connected components of this network. How many weakly connected components are there? What percentage of the nodes are in the largest weakly connected component? Perform this task in the function ***weakly_connected(G)*** (Round to 2 decimals)
3. [4 points] Find the strongly connected components of this network. How many strongly connected components are there? What percentage of the nodes are in the largest strongly connected component? Perform this task in the function ***strongly_connected(G)*** (Round to 2 decimals)
4. [10 points] Compute the shortest path length between every pair of nodes in the largest strongly connected component. **Show the distribution of shortest path lengths using a histogram or barplot.** What is the average shortest path length in the largest SCC? What is the max shortest path length of the largest SCC? How does the distribution compare to the undirected data set in part 2?

Part 4 - Topologically Ordered Languages

[20 points]

The fourth data set can be found in "**language_data.txt**". It represents the relationships of influence among programming languages. Each line in the file consists of two programming languages separated by a space. The language on the left was influenced by the language on the right. Use the directed acyclic graph functions in NetworkX to answer the following questions.

1. [2 points] Using NetworkX built-in methods, load **language_data.txt** into a networkx graph object in the ***load_language_data()*** function.
2. [4 points] Use the ***is_graph_dag(G)*** function to determine whether G is a directed acyclic graph. Return a boolean value representing if G is a dag. .
3. [4 points] If the graph is not a DAG, make it one by removing the first edge from each cycle in alphabetical order. For example, if a cycle consists of the edges ('Basic', 'COBOL') and ('COBOL','Basic'), remove (`Basic', `COBOL').
4. [10 points] Answer the following questions:

a. How many source languages are there in this DAG? (A source language is a language that is not influenced by any other language.)
b. Which source language had the highest influence? Influence can be both direct (there is an edge from C to C++) and indirect (C++ influences Rust, therefore C also influences Rust indirectly).

Hint: NetworkX has a type nx.Graph and a type nx.DiGraph (representing a directed graph). Think about which one you should be using.
Pay careful attention to the input file. It shows the language on the left column is influenced by the language in the right column. For example, you will see `c++  c`*. That means that C++ is influenced by C. Make sure your graph has an edge from c to c++ and not c++ to c.*

Part 5 - Bipartite Projects of Github

[20 points]

The fifth and final data set "**github_data.txt**", is a bipartite data set where some of the nodes represent users, some represent projects, and an edge between a user and a project indicates that a user is a member of that project. Similar to the recommendation system example in the lectures, we can use one-mode projections to find projects and users that have strong associations with each other.

The format of each line in the text file consists of two numbers separated by a space. The left number corresponds to the user ID and the right number corresponds to the project ID. Load the text file and parse the bipartite graph.

1. [2 points] First import this data using the ***load_github_data()*** function. Check that there are no edges between nodes of the same type. A correctly-loaded graph will be bipartite, with 177,368 nodes and 440,237 edges.
    - *Be careful when creating your graph. You will notice that the ids for projects and users are integer values and can contain the same id. You may want to perform some action to make them unique to each bipartite set, such as prepending a 'u' to each user id and a 'p' to each project id. That is, there can exist a user '1' and a project '1', you may find it easier to change them into 'u1' and 'p1' when making your graph. In the github_data.txt file, the user is always the left column and the project is always the right column.*
    - *Lastly, you will also notice that load_github_data() returns a graph plus two lists. Those lists are the list of user ids and project ids and they are necessary to perform projections (notice they are passed as input to both projection functions). You will also need those lists to find the correct project or user ids to return in the get_user_pair or get_project_pair function.*

2. [8 points] Then, compute the two one-mode projections using the adjacency matrix using the ***calculate_projections****(G, uid_list, pid_list)* function.
    - *Note that while it is acceptable to compute the adjacency and adjacency matrices of the graph using NetworkX's native functionality (nx.adjacency_matrix and nx.bipartite.biadjacency_matrix, respectively), one should* ***not*** *use NetworkX's native functionality to compute the projections (ie, nx.projected_graph). Compute the projections directly from the adjacency and/or biadjacency matrix/matrices.*
    - *Be careful when performing matrix multiplication in python. There are several methods, but using the "@" operator requires python 3.5 or higher.*
3. [5 points] What pair of users share the most github projects? Use the ***get_user_pair****(M, uid_list)* function to find the correct pair.
4. [5 points] What pair of projects share the most users? Use ***get_project_pair****(G, pid_list) function*.