Proposal presentation

presentation ( why this topic, why not wikipedia or flight)

We chose the topic of **meal recipes** because food is something that connects all of us. **Everyone** cooks, eats, or at least has preferences, it is a very engaging topic.

Representing recipes as a graph isn't just an abstract idea, it actually opens up information for **useful** applications. For example:

* It can help with **meal prep planning**. By finding **clusters** of recipes that use similar ingredients, users can group meals and create more efficient grocery lists.
* It can also help with **recipe recommendations**. If two recipes share a lot of ingredients, they're likely to be similar. So recipes that are directly connected or only a few steps apart, can be recommended to each other.

This **graph** is also technically a good choice, because:

* Its **interconnected: relations** between recipes can be traced and analyzed for the explained ideas.
* It's **scalable**: we can shrink or grow the dataset without breaking the structure.
* It's **customizable**: for example, we can ignore common ingredients like salt or water, so that unrelated recipes aren't mistakenly grouped together

dataset presentation (kaggle, how are data in the files, columns/rows,...) + code

The dataset used in this project comes from Food.com and is available on Kaggle. It contains three main groups of data: raw recipe and interaction data, user interaction data, and preprocessed data.

The file RAW\_recipes.csv includes detailed information for over 200,000 recipes, such as their unique identifiers (id), names (name), ingredients in text format, preparation and cooking times, and step-by-step instructions.

To complement this, RAW\_interactions.csv contains over one million user interactions, including ratings, reviews, and timestamps for specific recipes, making it useful for evaluating recipe popularity and user preferences.

The dataset also provides three interaction split files interactions\_train.csv, interactions\_validation.csv, and interactions\_test.csv which divide user interactions for training and evaluation.

For structured experiments, the dataset includes preprocessed versions of the raw data. PP\_recipes.csv contains tokenized ingredient information for each recipe, where ingredients have been standardized and converted into token ID sequences. These tokens correspond to normalized ingredient names found in the ingr\_map.pkl file, which maps each token ID to its cleaned version.

Finally, PP\_users.csv holds preprocessed information about user behavior for downstream personalization tasks.

In this project, we use RAW\_recipes.csv to retrieve the human-readable names of recipes based on their IDs, PP\_recipes.csv to access the tokenized ingredient lists, and ingr\_map.pkl to decode those tokens into interpretable ingredient names.

(nodes / edges/ visualization/ cluster ? /why we limit the number of recipes)

In our graph based on recipes **each node** represents a **single recipe**.  
An edge connects two recipes if they **share at least one ingredient**. The weight of an edge is the **number** of shared ingredients.

For **example** if we take **lasagna** and **pizza** they both share tomato sauce and cheese. In that case the node lasagna and the node pizza share an edge with weight 2.

This first results in a very dense and interconnected graph.

The original dataset contains over 200.000 recipes, but using all of them would make the graph **extremely large and difficult** to work with. That’s because as we add more nodes, the number of edges grows exponentially, especially in a graph like this where many nodes are connected.

So, we decided to work with a **smaller subset** of 200 recipes. This makes it easier to compute, visualize, and analyze the graph, without losing the interesting structure.

To make the connections more clear and interactive, we also use **3D visualization**. This helps to show clusters of similar recipes, and lets us explore relationships that might be hidden in 2D. We did the implementation with the NetworkX library for the graph and we use Plotly to display an interactive 3D visualization. In the pictures you can see how the graph looks from two perspectives.

(optimisation -> inverted index / weights / remove the most common ingredients

To efficiently compare recipes based on their ingredients, we first constructed an inverted index that maps each standardized ingredient to the set of recipes that contain it. This structure allows for quick retrieval of recipe pairs sharing specific ingredients, avoiding expensive full pairwise comparisons.

Using this inverted index, we then computed edge weights. Edges were added between nodes only if the corresponding recipes shared a significant number of ingredients (for example at least 5 ingredients). This weighting strategy highlighted strong culinary similarities while filtering out weaker or less meaningful connections.

To further refine the graph and reduce noise, we removed the most frequent ingredients, those appearing in a large number of recipes as they tend to be generic and don’t contribute much to recipe specificity (like salt, water, oil). This step leads to a more informative network of recipes.

Analytics median, ...., (add more stats → cliques...)

After building our recipe graph, we **analyzed** it to learn more about its structure and to identify recipes that play key roles.

In general, we can analyze things like the **number of nodes and edges**, how **dense** the graph is, and how **many connections** each node has on average. We can also look deeper by checking **centrality**, **cliques**, how **far** apart nodes are, and whether the graph shows the **small-world** phenomenon.

For our graph with **200 example** recipes (and after the cut of most common ingredients?), we found:

Over **3,000 edges**, which makes an average of **15 connections** per recipe. And each connection has on average **6.6 shared ingredients**.

One recipe, Simple Corn Salad, had the highest degree, with over 100 direct neighbors.

Another, Sugar Glazed Walnuts, had the most total shared ingredients with its neighbors with a score of 718 ingredients.

This shows us two things: One is, there are recipes serving as **recipe hubs**, meaning they have many neighbours, and other recipes that are very **similar to many** recipes in terms of number of shared ingredients.

This distinction helps us to design a better project with good food recommendations considering both degree and weight.

A recipe that connects many recipes is a good starting point: if a user likes this recipe, they will have many similar options to choose from.

And a recipe with high total similarity in ingrediences is great for finding nearly identical alternatives.

Our recipe graph is not fully connected, it has **7 separate components**. This happens by design, because we only connect recipes that share at least 5 ingredients.

The **largest connected component contains 194 recipe**s, which is almost the entire graph.

Inside this component, we measured an **average shortest path length of only 2.15**. This means that most recipes are just 2 to 3 steps apart. Also the average clustering coefficient is 0.581.

Even though the graph is highly clustered and most recipes closely connected, it is not a **Small-World Network** by definition, as we have nodes that cannot be reached.

+ diagram

To better understand the structure of the recipe similarity graph, we generated a histogram illustrating the distribution of edge weights representing the number of shared ingredients between connected recipes.

This visualization provides insight into how frequently recipes are linked based on strong ingredient overlap. A concentration of low-weight edges suggests that many recipes share only a few common ingredients, while the presence of high-weight edges indicates tightly related recipes, possibly variations of the same dish or recipes.

This distribution helps us identify whether our filtering threshold effectively captures meaningful relationships. It also guides future decisions in refining the graph, such as increasing the threshold for denser subgraphs or identifying clusters of highly similar recipes for recommendation purposes.

recipe recommendations (top 10 with the most connected (+ rating ? calorie level ? number of steps ? depending on user preferences ) input : recipe or ingredients → output : most connected recipes

Our final idea for this **project** is to make the graph valuable and user-friendly by developing a recipe recommendation tool.

The idea is simple: the user picks a recipe they like, or **enters** a **list of ingredient**s. Then, based on the graph, we return the top **10 most similar recipes**.

Similarity here is based on the edges and therefore the number of **shared ingredients**. The more ingredients two recipes have in common, the stronger the connection, meaning the higher the edge weight. So, neighbors with high weights are ranked higher.

We will write an **algorithm** that finds the top 10 nearest neighbors of a given recipe node. If there are not enough neighbours, second degree neighbours are considered too.

Optionally, we could include **filters**, so users can choose recipes based on their preferred ingredients.

The first version will be a simple **Command-line application** like this. (example on slide)

Later, this can be expanded into a **web application** where users might explore recipe cliques and get smart recommendations based on what they like or what ingredients they have at home. We are here planning on only **including statistics** that are meaningful for the user experience, not just general graph theory numbers, but ones that help explain **why** recipes are recommended, or how they are connected.

Conclusion : project planning = diagram (when we choose the topic, when we code, ...)

+ distribution of tasks

In conclusion, this project evolved through several well-structured phases, beginning with a brainstorming session that led us to explore the relationships between recipes through their shared ingredients.

After selecting the Food.com dataset from Kaggle for its comprehensive recipe and user interaction data, we moved on to building a graph-based representation of the data. Each recipe was treated as a node, with edges representing the number of shared ingredients.

We developed several tools to process and clean the data, including decoding tokenized ingredients and linking recipe IDs to their names. This allowed us to create a meaningful and interactive 3D visualization of the graph. We then performed statistical analyses, such as examining edge weight distribution and node connectivity, to better understand the structure of the recipe network.

Moreover, we ensured a balanced collaboration by dividing the tasks between the two of us. One of us focused on data preprocessing and integration of complementary files, while the other built and optimized the graph and developed its visual representation. We worked together to interpret the results and prepare for the next phase of the project.

Looking forward, our final objective is to design a recommendation feature that can suggest similar recipes to a user based on a specific recipe they like or based on ingredients they already have at home.

**Progress presentation**

**Project overview**

During our first presentation, we introduced the topic of our project: food recipes. Let me give you a quick recap. We chose this topic because food is something that brings people together, making it an engaging subject with rich data. Representing recipes as a graph opens the door to concrete applications. For example, it can help improve recommendation systems—if two recipes share many ingredients, it is likely that they are similar.

To build this graph, we used a dataset from Kaggle, originally sourced from Food.com. This dataset contains over 200,000 recipes. The graph we constructed follows a simple logic: each node represents a recipe, and an edge connects two recipes if they share at least one ingredient. The weight of the edge corresponds to the number of shared ingredients. This quickly leads to a very dense graph. For this reason, we limited our study to a subset of recipes.

To optimize the graph construction, we implemented an inverted index. This method allows us to quickly identify recipe pairs that share a given ingredient. Using this index, we calculated edge weights and only added edges when the recipes shared a significant number of ingredients. At the same time, we removed the most frequent ingredients, since they are often generic and less useful for distinguishing between recipes.

Once the graph was built, we analyzed it to better understand its structure and to identify key recipes. The results revealed two types of centrality: some recipes act as hubs, meaning they are connected to many others, while others show a high level of similarity with their neighbors. This distinction helps us design better recommendation systems.

First Interactive Script

In this project, so far, we implemented **two python classes**, a **RecipeRecommender** and a **GraphManager**. These classes are central to how our system works and are shown in the ER diagram along with their attributes. The GraphManager contains all the **core logic.** This includes **building** the recipe graph, conducting the **similarity** **search** to find the best recipe matches, and **retrieving** recipe names or ingredient names based on their ids.

The RecipeRecommender class wraps around the GraphManager to make it more user-friendly during this development stage. It **acts as an interface** between the system and the user. This is what we use for the command line application.

The RecipeRecommender class has **three main attributes**. First, the graph manager itself. Then, the **number of recommendations** to show. Finally, the **normalization type**, which determines how similarity scores are calculated. These last two attributes can be set by the user or left at default values. By **default**, the recommender returns **10** recipe recommendations and uses normalization type **1**.

If the user wants, they can **override** these defaults either by passing in different values when calling the interactive evaluation function or by changing them in another interactive script.

There are **3 normalization options** to choose from. Option **0** uses the **number** of shared ingredients as the similarity measure, which is one edges weight. Option **1 normalizes** that value by dividing the number of shared ingredients by the total number of ingredients in the neighboring recipe. Option **2** goes one step further and also includes the **recipe rating**. This is done by adding the normalized rating, which is the percentage of the rating out of five, to the normalized ingredient similarity. That way the rating and similarity are **50:50** distributed.

When the script is run, the user is prompted to enter the name of a recipe. This input acts as the starting point for the search. The system then looks for similar recipes in the graph based on the selected normalization method and returns the top matches.

For example, if the user **searches for Almond Cow**, which is a recipe for an almond-based drink, the GraphManager processes the query and returns a list of similar recipes. Internally, each result is stored as **a JSON object** in a list. Each object includes the identifier of the recipe, its name, its similarity score, its list of ingredients, the ingredients shared with the queried recipe, and the recipe’s rating.

These results are sorted by similarity score, which depends on the chosen normalization method.

In our terminal-based implementation, the user sees a list of similar recipes with detailed information. This includes the number and list of shared ingredients, the rating, and the total number of ingredients in each recipe as you can see.

We quickly found that users do **not always know exactly** which recipes exist in the graph. To solve this problem, we implemented a simple **fuzzy matching** system using Python’s ***difflib*** library. When a user enters a recipe name that is not recognized, the system suggests recipes with similar names. This helps users find the right recipe even if their input has typos or does not exactly match any known recipe.

**UI library comparison**

As part of our project, we also started thinking about how users could interact with our recipe recommendation system. The goal is to offer a simple interface where someone can input a recipe and receive automatic suggestions of similar recipes based on the number of shared ingredients. While this interface is not yet integrated into our final pipeline, we began exploring different technical options to prepare for this next step.

We compared three Python libraries commonly used to create user interfaces: Tkinter, Dash, and Streamlit.

Tkinter, which is included by default in Python, allows for building desktop GUI applications. It is lightweight and requires no additional installations, but it’s also quite basic. The interfaces created with Tkinter often look outdated, and even simple layouts require more lines of code. As a result, it’s not the best fit for a project that depends on dynamic visualizations and a clean user experience.

Dash enables the development of interactive web applications and is well suited for data visualization, especially since it is built on top of Flask and integrates seamlessly with Plotly. It is a powerful tool for more complex web dashboards. However, its architecture—especially the use of callbacks—adds a level of complexity that makes rapid prototyping more difficult. For a relatively simple interface like ours, it felt like too heavy a solution at this stage.

Streamlit, on the other hand, proved to be a much smoother option for our needs. It allows for the creation of interactive web applications using very simple Python code, as if writing a normal script. With just a few lines, we can generate input fields, display results, and include interactive visualizations using Plotly or NetworkX.

So this was our initial idea, a home page where the user can enter a recipe and search the top 10 recommendations. The user can also click on a specific recipe and access the recipe page where we can find some information about the recipe.

To test its potential, we built a first mockup using fake data, and the results were promising. The simplicity, speed, and clean appearance of Streamlit make it particularly well suited for our project.

End schedule

The **schedule** is going **well** so far. We have completed the command line interface and the basic recommending logic. We have also created a mock-up for the user interface and conducted some technology tests to ensure that our chosen tools will work for the UI.

In the **next few weeks**, we plan to complete the implementation of the graphical user interface. Therefore we distribute the tasks in a way where one of us works on how the UI handles the retrieved data and the other one adds the last missing recipe data like the preparation steps and ensures all necessary data can be retrieved.