StudentX

SP24 Capstone Design: ***Final Design Report***

Team Number SP24-36

Team memebers: ***Peter Chen, Mulugeta Akalu, Nikhil Agarwal, Ishaan Keswani, Long Phan***

Adviser(s): Bo Yuan

Abstract— **Imagine if there was an online peer-to-peer marketplace dedicated to the Rutgers community or any other large university or college. This platform would be less susceptible to fraud and would give students access to a broad selection of relevant products and services from other students. While other online peer-to-peer marketplaces may offer billions of items, they often lack the sense of community and trust that many students seek. Moreover, this platform could provide students with very niche items that are in demand within the community; this can range from specific textbooks to calculators tailored for certain courses, or even furniture left by relocating students.**

Our platform, StudentX, aims to develop a niche online peer-to-peer marketplace for the Rutgers community, with potential expansion to other large communities and institutions. Our goal is to connect students within the community and facilitate the discovery and exchange of a diverse selection of items, whilst also helping students earn some extra money and reduce waste. Any member of the Rutgers community will be able to sign up for the app using their Rutgers email; with that, they'll be able to access our application features like direct messaging and recommended relevant products and services- powered by sophisticated Machine Learning recommendation systems. This ensures that users can confidently purchase from other fellow students, thereby enhancing security and providing a much more affordable selection of relevant products.

Keywords— ***Exchange, Share, Buying, Selling, student, market, machine learning, software engineering, semantic search, recommendation system, peer-2-peer marketplace, peer-to-peer, KNN, E-commerce, Website, Web application.***

I. INTRODUCTION

A peer-to-peer marketplace is a website that connects people who own a product or offer a service with people who want to buy or rent it. Airbnb is a classic example. The role of the website is to help these two groups of people find each other. The site also handles marketplace payments and helps build trust between the parties. Peer-to-peer marketplaces don't need to own or provide any of the products or services offered on their platforms. This makes them relatively inexpensive to start. In addition to Airbnb, famous examples include the product-selling website Etsy and ride-hailing app Uber. Almost any product or service can be sold through a P2P platform. The peer-to-peer (P2P) marketplace ecosystem is estimated to be valued at US$ 1,526.2 million in 2024 and is anticipated to reach US$ 8,474.8 million by 2034.

The rise of specialized marketplaces stands out as a pivotal advancement in the evolution of peer-to-peer marketplace platforms. Instead of aiming to cater to a broad audience, entrepreneurs are now directing their efforts toward creating highly focused markets tailored to specific industries or interests. These specialized markets encompass various niches, such as online boutiques specializing in vintage clothing, artisanal crafts, local food providers, and beyond. By honing in on specific niches, these markets can deliver tailored and unique experiences to consumers, thereby enhancing user engagement and fostering loyalty.

In the contemporary mobile-dominated landscape, adopting a mobile-first strategy is paramount for the prosperity of any online marketplace. Given that most users engage with these platforms through their smartphones, it's imperative for marketplace developers to prioritize creating mobile-responsive and intuitive applications. Consequently, there's been a notable uptick in the requirement for mobile app development services, driven by marketplace owners' endeavors to deliver a smooth and hassle-free experience to their clientele.

Personalization stands out as a pivotal trend in the evolution of peer-to-peer marketplace platforms. Utilizing artificial intelligence and machine learning, these platforms offer users customized product suggestions, search outcomes, and content. Through the analysis of user interactions and preferences, these systems elevate user satisfaction and improve transaction efficacy. This personalized approach not only amplifies user engagement but also generates increased revenue for marketplace operators.

Establishing trust and safety holds utmost importance within peer-to-peer marketplaces. Users must feel confident in their transactions, whether buying or selling on these platforms. To tackle this challenge, marketplaces are adopting diverse trust and safety protocols, including user authentication, background screenings, rating systems, and secure payment gateways. Additionally, there's a growing trend toward transparency in user profiles and transaction records to foster trust among community participants.

The localization trend is rapidly gaining traction as users prioritize accessing products and services in close proximity. Hyperlocal marketplaces facilitate connections between users and nearby businesses, promoting a sense of community and convenience. These platforms commonly cater to services such as food delivery, home repairs, and local event listings. The pandemic has additionally expedited the expansion of hyperlocal marketplaces, as individuals increasingly seek nearby options to fulfill their daily requirements.

The flexibility and adaptability of peer-to-peer marketplace platforms are on the rise thanks to API (application programming interface) integration and open platforms. Marketplace owners are realizing the benefits of enabling third-party developers to craft complementary apps and services, enriching the overall user experience. This strategy fosters innovation and broadens the ecosystem surrounding the marketplace.

There are challenges faced by the peer-to-peer marketplace ecosystem. P2P marketplaces often operate in regulatory gray areas, facing scrutiny from regulatory bodies in various jurisdictions. They also face problems related to compliance with local regulations, including taxation, licensing, and consumer protection laws. Maintaining quality standards across a diverse range of goods and services offered on P2P marketplaces can be challenging.

In most popular peer-to-peer marketplaces customers find a lot of goods and services but not necessarily a community they desire. In a community-centered peer-to-peer marketplace, it is easier for the parties to trust each other and find a large selection of relevant goods and services. We believe the Rutgers community can benefit greatly from a marketplace designed to be accessed by a university credential. Such a platform will offer a large selection of relevant used items at cheaper prices, especially at year-end when graduating students are looking to get rid of a lot of useful material. It will also provide a sense of community they desire, a safer and more convenient transaction experience.

II. Conceptual Design

A. Web Application

We will begin by developing the web application for StudentX. This web application will encompass several key features aimed at creating a seamless platform for Rutgers University students to exchange stuff. We will incorporate our own user authentication system, and get sellers to verify via Rutgers NETID, to ensure security. The application will empower users to efficiently manage their listings. Sellers will easily be able to manage creating listings, editing them, and deleting them. They'll be able to add information about the item they're selling such as descriptions, prices, and categories. We'll have designated categories for students such as parking spots, places for rent, and other student items. We'll also incorporate robust search and filtering functionalities to help users to find specific items easily. A direct messaging system will facilitate communication between buyers and sellers while prioritizing user privacy. To streamline transactions, our web app will have a secure payment processing system, with escrow, that will be integrated using a third-party API like Stripe or PayPal. Our web design will also be responsive, to ensure optimal user experiences across various devices, including desktop and mobile platforms. We will ensure that our site is responsive by incorporating well-known industry standard frameworks and libraries such as TailWind CSS. We will also couple this together with a sophisticated backend system built with Python. While originally, we were planning to utilize React for the frontend side of our web application, we realized how difficult it can be– as well as the notable learning curve for the entire team- to build different parts of the application using different programming languages. So instead, we now opt to use Flask to help build and manage our web application. Coupled with Google's Firebase, we'll be able to seamlessly complete the project at a relatively low cost. Alongside this, we will utilize Docker containers to help us create a scalable and transferable application which we'll be able to run everywhere.

B. Machine Learning

We will also focus on a machine learning component of StudentX. The purpose of this will be to implement a recommendation system to enhance the user experience. By analyzing user behavior, including time spent on items, purchase history, and various user interactions, we will implement an algorithm to suggest relevant items to individual users. The model will be designed for continuous improvement, tailoring recommendations based on evolving user preferences. The technology stack for this component will include Python for machine learning development, with considerations for popular libraries and frameworks such as PyTorch. Our data will come from our user base, and we will test numerous models to validate the accuracy of our model using historical user data. We'll also seamlessly integrate the machine learning model into both the web and mobile applications to ensure real-time updating of recommendations based on user interactions..

III. Methods / Results (any relevant) / Approach

Our idea for this system's behavior and how it will look should be quite simple. It will have a few key interfaces that the user can navigate between, including a store page, a marketplace page, and a user information page. On top of that, we had been contemplating a chat page where users could message each other about products and buy and sell them. This was the high-level overview of what we think this system would be like. That also covers the behavior, a simple application that would give users the freedom to set their own prices and talk freely with one another about items of interest. Even more, we thought that to make our project unique, we would integrate it with some machine learning algorithms such that users are able to optimally search and find items that they need.

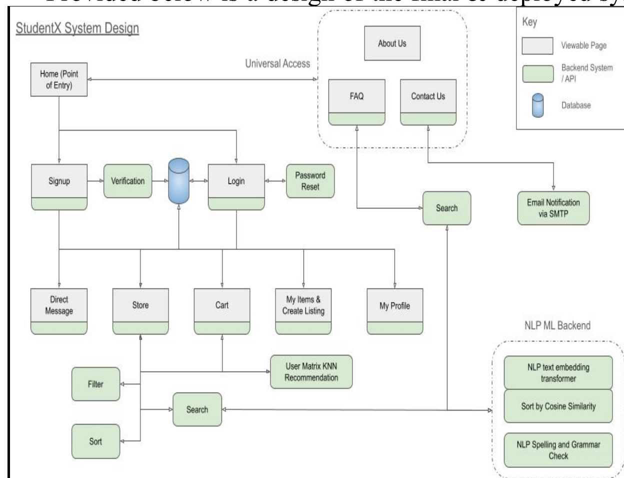
From the start, we were thinking of building a mobile application or a web application. However, after considerations and estimates of the scope of work, we settled on focusing on aa web application given the time constraints of the project.

After deciding on creating a web application, we had to figure out what technologies we would use to build it, such as what kind of database, what programming languages, etc. Give that we wanted to build a recommendation system and incorporate NLP, we decided on a Python web development framework, Flask. This allowed us to seamlessly incorporate

PyTorch into our backend. Using plain HTML and CSS, we built the front end with JQuery and Bootstrap so that the website would be responsive across different screen sizes; this will also make it easier for our website style to appear consistent across different website pages.

This led to the next design/implementation challenge: selecting a database at a low cost. So, to optimize that and to minimize the use of any server, we decided to go with a serverless design and utilize GCP's- Google Cloud's- various features to do this. Due to this architecture, which we chose to save money, we ended up having to use a NoSQL database; and what better one to use than Google's own Firebase? This allows us to find and access data extremely quickly and efficiently.

Provided below is a design of the final & deployed system.



And so this was our overall plan: first focus on building out the database structure, then the backend (mostly referring to our web APIs), and finally the UI/frontend. In this manner, we can intuitively figure out and plan out the core parts of the system and then build on top of that. Finally, when all of this was completed, we could add on our machine learning models and algorithms.

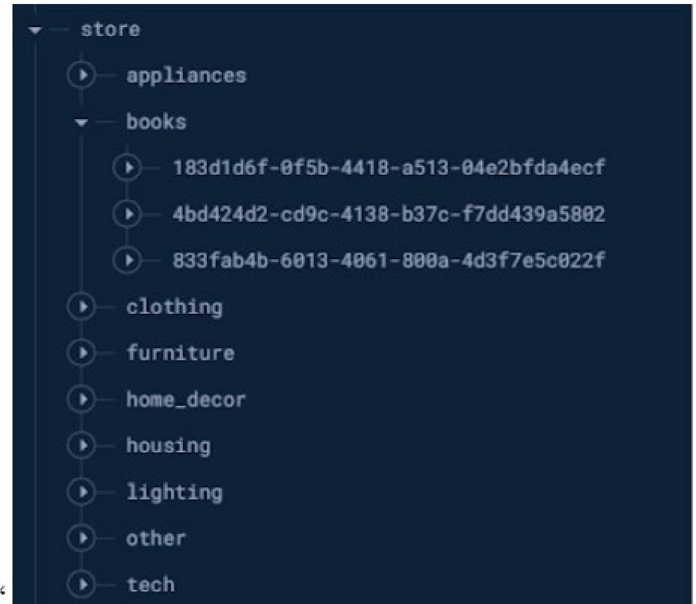
Provided below are more in-depth explanations about the technologies and implementations for the Semantic Search, KNN recommendation system, Chat, Store & UI functions/design.

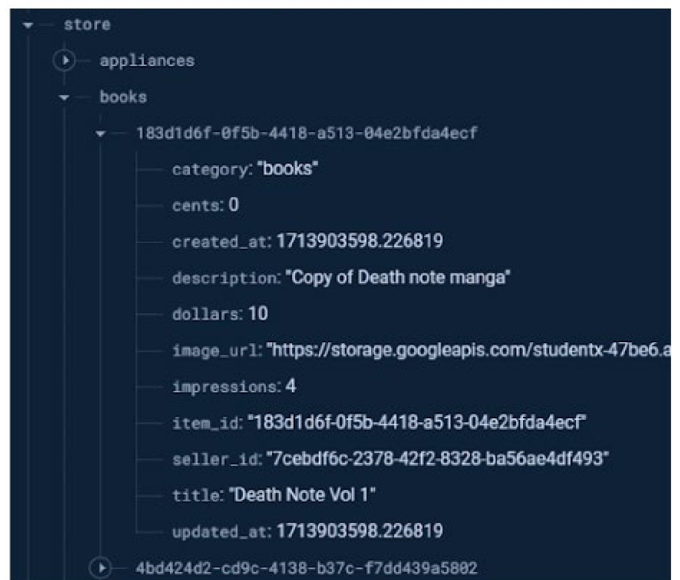
A. Database & File Storage

For this project, we used Google's Firebase real-time database & and file storage. The database is key-value and nonrelational which provides certain advantages with tradeoffs. Central to the database are the unique IDs we generate for all users, chatrooms, and items. uuid.uuid4() generates a unique identifier using random numbers. This UUID (Universally Unique Identifier) is a 128-bit number, typically represented in hexadecimal format and used to ensure that identifiers are unique across different systems and contexts without a central coordinating entity.

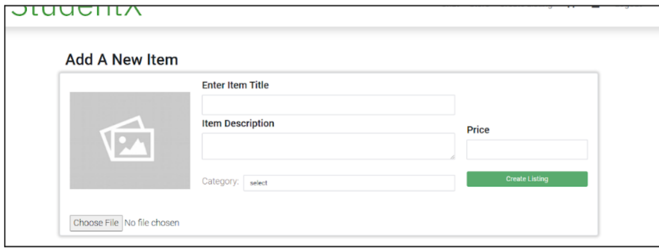
The database has four main paths: 'users', 'store', 'chats', and 'surveys'.

The store path holds every item that is uploaded to the store. The store path is divided into categories followed by item_ids of the items in that category. The reference to a specific item in the store would then be /store/<category>/<item_id>. As shown below on the left there are three items in the store corresponding to the book category.



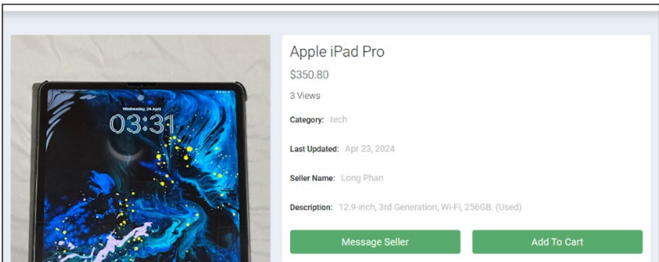


Above in the seconds image, we see the attributes stored in an item reference for item_id '183d1d6f-0f5b-4418-a513-04e2bfda4ecf'. We store the category, time the item was created, description, price, a link to the image in file storage, the number of views (impressions), the item_id, the user ID of the seller, the title of the item, and the time the item was updated (same as created time if no updates). Users use the following form to create items (listings).



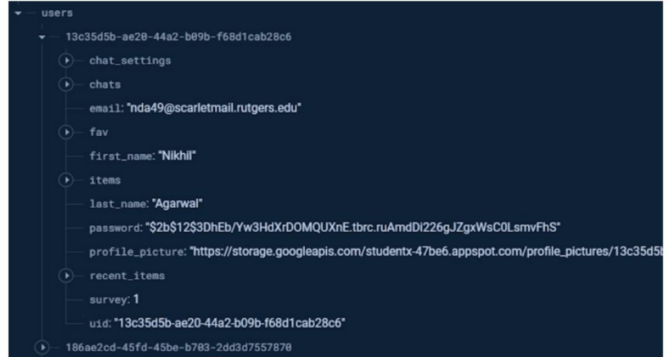
The chat path holds every chat room that is created on our platform. Chatrooms can only be created between two users and are assigned a unique chat_id. To retrieve all the information about a specific chatroom, we make a get call to the /chat/<chat_id> path. Each chat holds the user IDs of the two users in that chat room, as well as all the messages sent in that chatroom (in chronological order). Ever message object that is stored holds the chat_id of the chatroom it was sent in, the message, the time the message was processed by the server, as well as the user ID of the sender.

As shown above, there exists a chat with chat_id '0c51acb9-cebe-4514-a04d-452498f37469' that holds three messages sent between users '13c35d5b-ae20-44a2-b09b-f68d1cab28c6' and '8467b3d0-d26b-46d4-999c-952a2245089f'. Users can create chatrooms by clicking the message seller button on any item page. If they already have a chatroom with the seller, then we do not create a new room. If they do not share a room, we create a new one and store it under 'chats'.



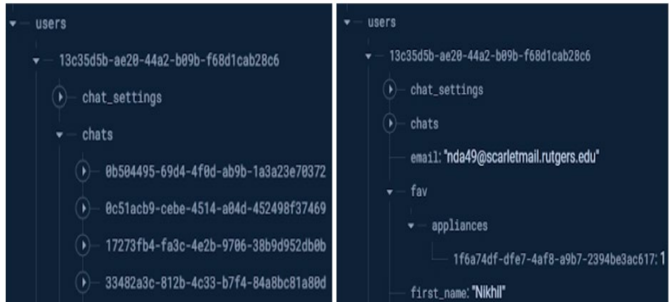
The survey path holds all the responses from every user who took the survey for our recommendation system. To get the survey information for a specific user we make a get call on the path /surveys/<user_id>.

The user path holds all the information about a specific user. The attributes for each user are added at the time of login/signup and throughout our platform experience in the form of preferences, item favorites, cart items, profile pictures, etc. Each user is assigned a unique user ID using uuid.uuid4() in Python. Below is Nikhil's user profile stored under the user/<user_id> path.



The basic attributes of email, uid, password (encrypted), and name are all present and generated when the user signs up. The user can also edit these attributes on the profile page along with their profile picture. Some of the most crucial parts of our database design occur in the /user/<user_id>/fav and /user/<user_id>/chat paths within each user object.

Recall that we store uid information in the 'chats' path {"user1": <user1_id>, "user2": <user2_id>}. To load all the chats that Nikhil is in, we would need to loop through each chat_id attribute and check if Nikhil's uid is present as either user1 or user2. This becomes highly inefficient as the number of chatrooms grows. To fix this, we keep a list of chat_ids that Nikhil belongs to in the user attribute itself. Then for each chat_id, we make a call to the path /chats/<chat_id> to compile a list of all chats that Nikhil is a part of. We do the same thing for items that a user favorites as well as items a user adds to the cart.



Above we see that the user has one item in their favorites list, storing both the category of that item as well as the item_id which allows us to make a call to the path /store/<category>/<item_id> to get the information of that item, instead of having to loop through every item in the store.

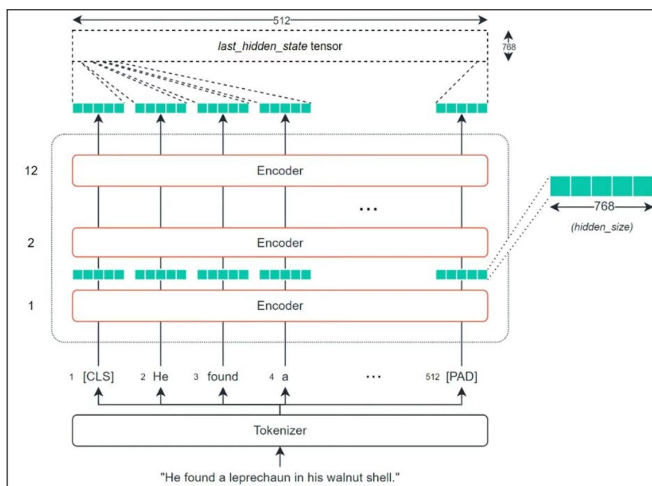
B. NLP Semantic Search

Backend Code Files: store.py, faq.py
Frontend Code Files: shop.html, faq.html, faq.js, shop.js

Semantic search refers to a search technique that seeks to improve search accuracy by understanding the searcher's intent and the contextual meaning of terms as they appear in the searchable dataspace. Unlike traditional keyword-based searches, which only look for matches of the exact query words or phrases, semantic search considers various factors including the context of the search query, the relationship between the words, synonyms, generalized and specialized queries, and natural language as it is used in everyday communication.

This approach enables the search algorithm to understand the query at a deeper level and return results that are more aligned with the user's intent, even if the exact words from the query aren't present in the results. Semantic search is widely used in various applications, including web search engines, information retrieval systems, and artificial intelligence models, to provide more relevant and contextually appropriate results.

For this project, we used a Hugging Face pre-trained transformer to generate text embeddings for strings. In other words, converting strings to numerical vectors. A diagram of the general transformer is provided below.



On our store and FAQ pages, the search function takes the user search input and feeds it to the transformer to generate a text embedding. We then compare the text embedding to the embeddings of items in the store or questions in the FAQ using cosine similarity. The input strings for items in the store are the “item name” + “item category”.

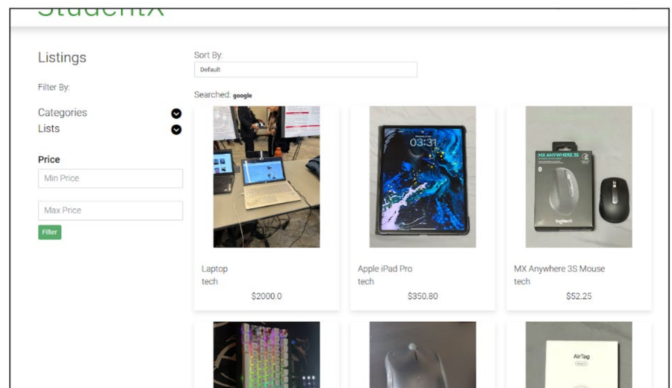
Cosine similarity is a metric used to determine how similar two vectors are by measuring the cosine of the angle between them. It's particularly useful in fields like text analysis and information retrieval, where each vector might represent a document or a set of features. In our project's case, it is a vector indicating the semantic meaning of the user's search and the products in our store. In essence, cosine similarity assesses the orientation of two vectors in a multidimensional space, rather than their magnitude. The outcome of this measurement ranges from -1 to 1. A value of 1 indicates that the vectors are pointing in the same direction, suggesting high similarity. A value of 0 means the vectors are perpendicular, showing no similarity, while a value of -1 implies that the vectors are in opposite directions, indicating complete dissimilarity.

The equation is quite simple and involves dot products and magnitude calculations:

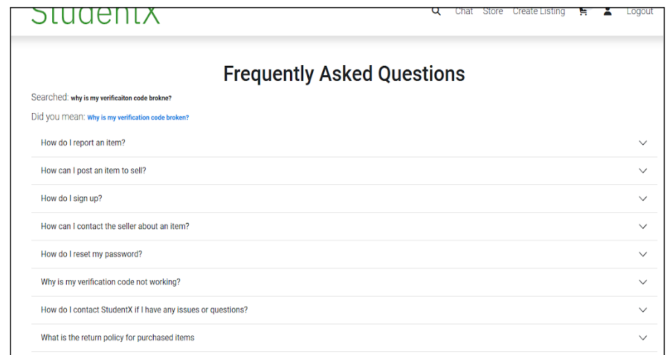
$$\frac{A \cdot B}{||A||\ ||B||}$$

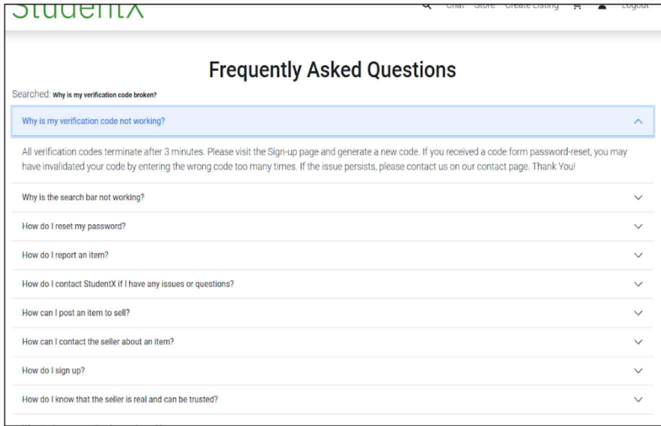
To utilize this equation efficiently and calculate the similarity scores for multiple vectors, we used sci-kit learn’s functions for cosine similarity.

This function is fully implemented in our store. Take the search “google” for example. We do not have any Google tech products in the store, but we do have Apple tech products. If we were using a keyword search, the search would not return any items to the user. However, since we are able to calculate the semantic meaning of the word Google and find similar items such as related tech products or Apple tech products, we can display these items to the user instead.



Further, we use a pipeline to a pre-trained Hugging Face model that detects and corrects typos. Since we do not know exactly what the user would like to search for, we do not automatically correct the search. Instead, we suggest the correct search and give the user the option to utilize it.





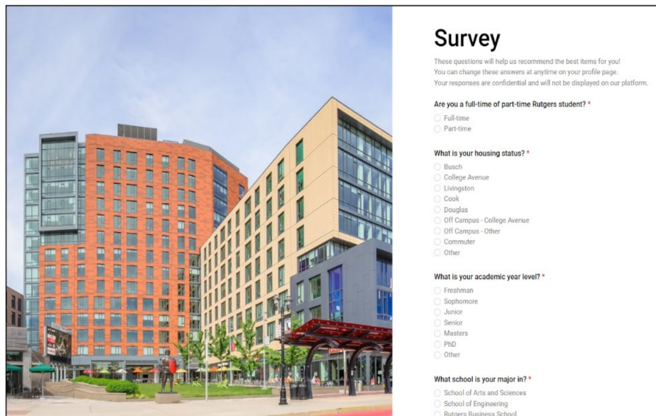
As shown above, the correct search is suggested and utilized by the user. In the end, the most semantically similar question is prioritized at the top and enables the user to find the answer.

C. KNN Recommendation System

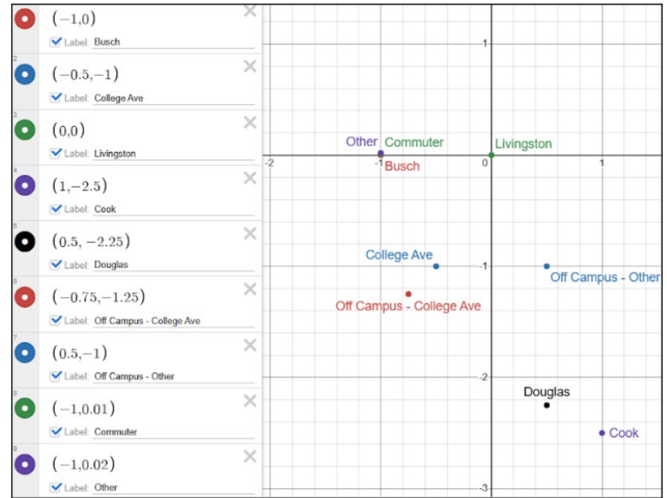
Backend Code Files: questionnaire.py, store.py, user_matrix.py
Frontend Code Files: questionnaire.html, shop.html, shop.js

As part of the project requirements we set for ourselves, we built a KNN recommendation system for users. The system consists of two independent parts. First is the user matrix that is generated for each user, and second is the tracking of items viewed by a user. The idea is that we show similar items to similar users.

We generate a user matrix for each user by providing a user with a survey before they can access the recommendation category in the store. The survey is tailored specifically to Rutgers students and questions about what school a user is in and what campus they live on etc.



The answers are stored in our database using one-hot encoding. This allows us to calculate the distance between user answers without giving bias to one answer over another. From here the distance calculation is straightforward. Let's say user1 selected "sophomore" and user2 selected "senior", we would store the answers in our database as [0, 1, 0, 0, 0, 0, 0] & [0, 0, 0, 1, 0, 0, 0], so the distance would be 2. If both users had selected "sophomore" then the distance would be 0. Further, we calculate the distance differently for the housing category.



Now comes the challenge of selecting weights for each distance metric. After calculating distances for each question on the survey, we are left with 6 distance values. To generate a single numerical value that we can use to sort users in our KNN model, we take a weighted average of all the distance values calculated. The weightings are as follows:

12.5% → Drive

7.5% → Enrollment

25% → Housing

15% → Academic Year

12.5% → School

27.5% → Category Preferences

The most weight is given to where a student is on campus as well as to what categories a student prefers. In this way, we sort and compare user matrices for KNN by giving larger weights to where students are located approximately and what students want to see. Users can retake the survey whenever they want and can change their responses to whatever they choose.

The second part of the recommendation system requires that we keep track of the items that a user has most recently viewed. We specifically keep track of the 5 items that a user has most recently viewed in the database.

KNN can be quite computationally expensive, especially as the number of users increases. To help mitigate this, our model randomly selects 100 other users (that have filled out the survey) to compare with so that we only calculate the distances between user matrices for 100 users at max. After determining the weighted average distances between the current user matrix and the 100 randomly selected matrices, we sort the distances and choose the "closest" most similar 5 users. We then compile a list of all the items those 5 users have looked at most recently and display them to our current user. If there are less than 100 users at any time, we simply use the user matrices for all the users that are currently available.

D. Direct Messaging

Backend Code Files: chat.py, app.py

Frontend Code Files: chat.html, chat.js

To build the direct messaging system, we utilized Flask-SocketIO.

Flask-SocketIO is an extension of the Flask web framework that enhances it with real-time communication capabilities, integrating seamlessly with the Socket.IO library to support bi-directional communication via WebSockets. This setup allows servers to send real-time updates to clients without requiring continuous polling. To start using Flask-SocketIO, you first install the library, create a SocketIO instance, and integrate it with your Flask application. It primarily works through event handling, where both the server and clients can emit and listen for events, enabling reactive functionalities like in chat applications where the server broadcasts new messages to all clients instantly. Flask-SocketIO also manages WebSocket connections and automatically resorts to long-polling if necessary, ensuring broad compatibility. Additionally, it supports organizing sockets into namespaces and rooms for efficient message targeting, making it ideal for scenarios like specific chat rooms or user groups. Integrated deeply with Flask, it works smoothly with both development and production setups, particularly when paired with WSGI servers and tools like Gunicorn for handling production traffic.

First, we need to create a chat room between two users in the database. After generating a unique chat ID using the inbuilt uuid library in Python, we know exactly which users can connect to that room, where to store the messages for that chatroom, and what the most recent message is in that chatroom.

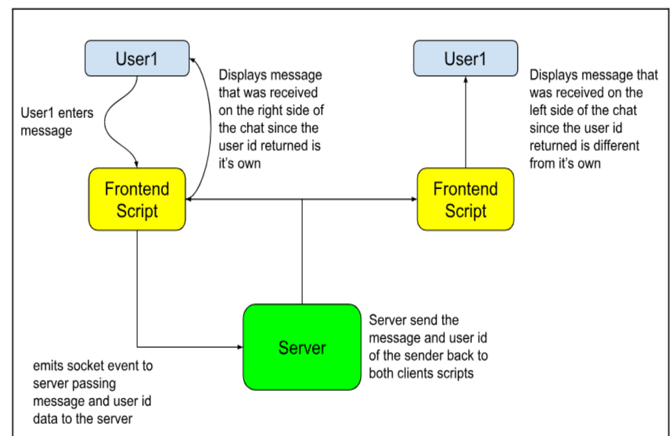
Second, when a user accesses the chat page we immediately connect the user to the socket via a route (@socketio.on("connect")) and get all of the chat IDs that the user belongs to. Separately we keep a server that keeps track of all the active connections in chat. This tracker is in the form of a Python dictionary (map) that holds chat IDs as keys. The value of each chat ID key is another dictionary that has key's 'count', 'messages', 'new_message', 'recent_time_stamp', 'recent_message', etc.

When the @socketio.on("connect") is triggered we check to see if the chat_id key exists in our tracker. If it does not we create it and load messages from the db into tracker[chat_id]['message'] as well as set tracker[chat_id]['count'] to 1. This specific count value refers to the number of users that are currently connected in chat. In this way, we know if we need to send email notifications or when to update messages to the database, etc. For example, if both users are connected to the chat we know that both users have read the message, if not, and tracker[chat_id]['count'] is 1 we know that we need to send an email notification to the user that is not connected to the chat. Any new messages that any user sends, whether both users are connected to the chat or not, are stored in tracker[chat_id]['new_messages'].

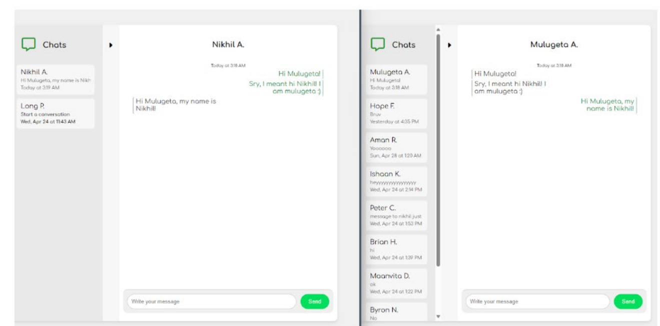
Further, when a user connects to the socket if tracker[chat_id] already exists, then we display previously loaded messages in tracker[chat_id]['messages'] as well as any new messages that exist in tracker[chat_id]['new_messages'] and set tracker[chat_id]['count'] to 2. In this way, when the second person joins the chat, they are able to load the new messages without accessing the database since we are keeping track of new messages on the server. Additionally, when either user joins and there are messages that have not been read by that user, the message is in bold on the left had side similar to how you would see it in iMessage.

Now when @socketio.on("disconnect") is triggered, we subtract 1 from tracker[chat_id]['count'] and check to see if that value has dropped to 0 (indicating both parties have left the chat). If 'count' is still greater than 0, we do nothing. However, if both parties have left the chat, we take all the messages in tracker[chat_id]['new_message'] and upload them to the database. We then delete tracker[chat_id] until a user from that chat tries to join again.

Let's take a look at how exactly messages are displayed in real-time through our chat feature. Take for example if user1 messages something to user2.



As shown, the clients never directly talk to each other. Instead, they communicate with the server and the server emits messages back to them while providing the uid of the origin user which allows both users to process the incoming message accordingly. Below is a screenshot displaying the same chat room from the perspective of different users.



Important to note is that the user is connected to every chat room that the user is a part of. So if any new messages are sent, regardless of the room that is currently displayed on the screen, the most recent message in the chat room will automatically update on the left-hand side of the chat. The user can then freely navigate to any chat to view new messages. We made sure to build the messaging system in a way that allows the

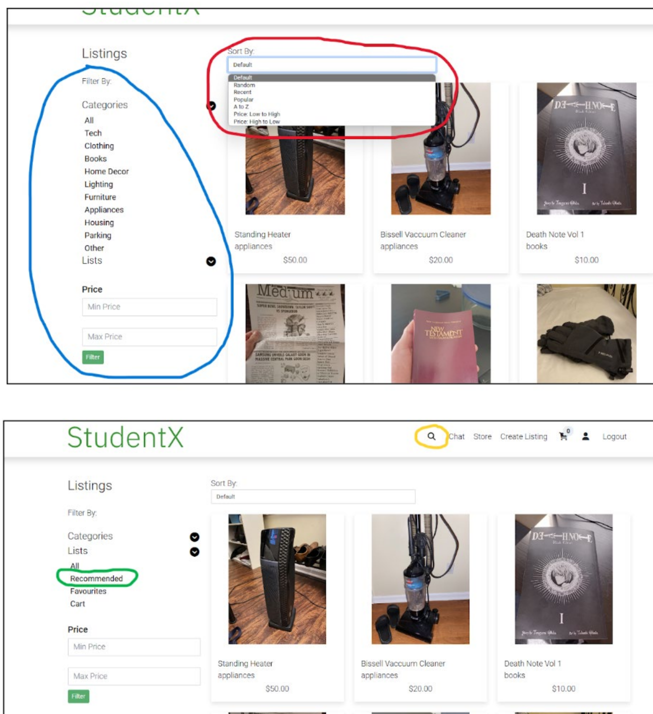
user to utilize and travel to any chat without accessing the database. The database is only accessed when you load the chat page itself, not while navigating the chat page.

We also keep track of user information in the backend like user_id to name conversions so that we can display names, timestamps, etc. to the user in a convenient and user-friendly manner.

D. Store

Backend Code Files: store.py
Frontend Code Files: shop.js, shop.html

The store page contains four major functionalities and utilizes jinja2 templating to pass properly structured items to the front-end HTML page. Paramount to the store page are the sorting and filtering functions for items, as well as the KNN recommendation system and the NLP semantic search for items.



The Blue denotes all of the categories you can filter by as well as by a ‘min price’, ‘max price’, or both. The Red allows users to sort all the returned items from a query or filter by the selected sort. Using Python’s built-in sort functions (lambda functions) with keys, we can efficiently sort all the items returned to the user.

The Green is a filter button for recommended items. When this category is selected, the user is fed into the KNN recommendation model, and the appropriate items are displayed to the user. The Yellow circle denotes the search bar which allows for NLP semantic search capabilities.

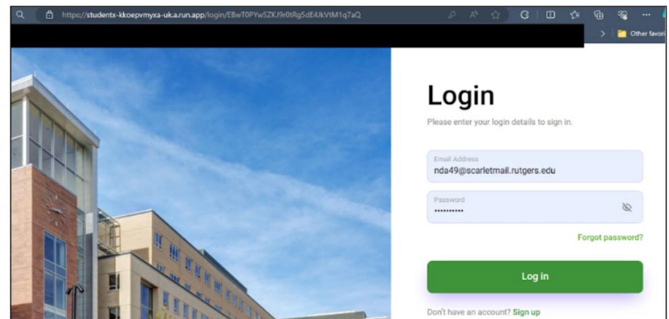
Additionally, we feed the user the items in chunks to limit the computation expense of loading an increasing number of items as the store grows. Users can navigate multiple pages of items.

E. Automatic Routing

Backend Code Files: app.py
Frontend Code Files: route.html, login_route.html, signup_route.html, verify_route.html, route.js

Some of the pages on our website like the chat, store, and profile pages can only be accessed if you are logged in as a verified user. To check if a user is logged in we simply access the flask session and try to fetch the ‘uid’ key. If the return result is None we redirect the user to the login page. This is relatively straightforward and ensures that users do not access pages that require authentication. However, after the user logs in, the route is static and programmed to send the user to the home page. So, for example, if I tried to access the chat page, and I was not logged in, the website would route me to the login but not route me back to the chat page after logging in. Therefore, for user convenience, we implemented automatic page routing after logging in depending on how they were first routed to the login page.

The fix was implementing routing using a routing key. The routing keys are all randomly generated secret keys that are stored in our .env file. So if we access the store page and we are not logged in the backend will route the user to /login/<store_routing_key>. For chat the route would be /login/<chat_routing_key> and so on. The following URLs route the user to an intermediary page called ‘route.html’ that takes a routing_key input via Jinja2 templating and routes the user to the appropriate page by submitting a form with a GET method and the appropriate action. All of this is done almost instantaneously and the user is displayed a loading sign on the intermediary. Below is a screenshot of the login page with a routing key to the books category in the store.



F. UI

Frontend Code Files: Everything under static /and html/

Originally, we had been planning to build our website using some JS framework. This way we could create and build a wide range of functionalities without having to write everything ourselves. However, this idea quickly fell through as we didn’t have many people in our team that were experienced with using JS frameworks. So, given the fact that we were already using Flask we decided to pivot and just use plain old HTML and CSS as well as a little sprinkle of JS.

Creating a user interface that works smoothly across all devices was a big challenge for us. We wanted our website to

look good and be easy to use whether you were on a computer or a phone, meaning we wanted it to be dynamically responsive across different screen sizes. That's where Bootstrap came in. It's a collection of all these premade CSS classes that helped us make sure everything looked right no matter what screen size you were using. On top of that it saved us so much time from having to individually write and configure every single element and style. We could organize stuff neatly and make sure everything looked the same, thanks to some fancy premade styles given to us by Bootstrap. Even more, to decrease the size of our deployment, we utilized Bootstrap CDNs which are content delivery networks so that we could easily and quickly download the stuff that we needed from Bootstrap without having to store these massive files in our codebase. Another unique feature about Bootstrap is that it has a few built in animations and it also helps us integrate a few other lesser known but quite useful libraries into our codebase and website. This way, we didn't have to import a large number of other libraries and could easily just gain access to countless JS scripts just from Bootstrap alone.

While we spearheaded our UI development using Bootstrap, we faced a number of other challenges too, with one of them being image loading. Oftentimes, we would have many images trying to load onto the screen at once. This would slow down the performance of our website, and lead to a bad user experience. So we added lazy loading. Essentially, instead of loading all the images at once when you open the page, they only load when you scroll down to them and when they appear on the portion of the screen that the user sees. It made the site a lot faster and smoother to use. On top of this, we sent most of our queries such as sorts to the server-side instead of finishing it on the client side. This was a way for us to cache and organize and keep track of different searches in one place while at the same time boosting the website performance for the user.

In the end, we managed to create a UI that not only looks good but also works well, thanks to the combination of HTML, CSS, JavaScript, and Bootstrap. It was a bit tricky at times, but overcoming those challenges made the final result even more satisfying. It was scalable across multiple screen sizes and we were able to have all of the functionalities work regardless of device and platform.

IV. Cost and Sustainability Analysis

For us, the total cost of the prototype that we developed- the website that ran locally in a controlled environment- is $0. Although we used Firebase and deployed it via GCP, we landed within the free tier limits of each of these services, thus we did not need to pay anything. However, if our app receives greater usage, we would need to pay a significantly greater amount for both storage, API calls, and calculations by our Machine Learning model on GCP's servers. In short, we use authentication, image storage, and real-time storage with Firebase, and all of these increase with user count, which will prompt us to move on to Google's Blaze plan from our current Spark plan. Although it is hard to exactly calculate how much, depending on how much of each kind of data we have, it can potentially reach $730 a month for storage for roughly 50,000 users.

For our real-time database, it costs about $5/GB to store and $1/GB to download. On top of that, for cloud storage, it costs about $0.026/GB stored, this is more for stuff like images and larger files that we may store. For our machine learning model, it is hard to exactly put a number on how much this will cost since the cost heavily depends on not just who the model is generating recommendations for, but also how much data it must scan to calculate the recommendations. Thus, overall, with all of these costs in mind, we came up with roughly $730 per month.

Other annual one-time costs that we did not mention include our domain name, SSL certificates, and Cloudflare subscriptions to protect us from cyber-attacks. All of this should be relatively cheap and should be no more than $100 per year. It is important to note that displaying ads on our platform could help us offset these costs given that we do not charge users any fees for using our platform.

For users, our platform is extremely cheap. Again, we highlight the fact that we give users complete freedom as we do not charge users anything for making transactions. This means that users will pay us nothing ($0) to list and use our platform and they will keep 100% of the money that they make.

Our project should have a significant positive impact on people's lives, especially college students based on our current platform focus. Our project will help significantly reduce waste by allowing students to give away/sell their items to other students. At the same time, users will be able to earn extra money on the side and find exclusive items that they might need at a lower price. We do believe that this product can significantly change consumption patterns. Users, mostly students for now, will no longer need to buy expensive items from stores such as Barnes and Noble or purchase hefty furniture. Instead, via our platform they will be able to save significant money by utilizing our platform to find other students who are trying to get rid of unused items.

As our product isn't automating anything necessarily, it won't impact employment. Although we aren't creating a new field with our project, if we succeed and gain enough users, we will be able to expand and help create new jobs. We envision the possibility of the existence of student ambassadors who can help us establish our platform in various communities. By doing so, we can help students gain experience and find employment much more easily.

Finally, as our product is purely software, it poses little to no threat to our user's health or safety. Although extremely unlikely, it is possible that we could face regulatory action- not so much from the government, but from colleges more likely. Since we are helping students find good deals and exchange with one another, it is possible that we are redirecting many students who would potentially be further customers of the university - by buying stuff such as new school materials or new textbooks. Because of this loss of business, it is possible that schools could potentially ban our application to increase their own sales and revenue. This would be the only possible type of foreseeable "regulatory" action that we could face.

V. Conclusion/Summary

The StudentX project aimed to create a special online marketplace for Rutgers students, with the potential for

expansion to other universities. It aimed to build trust and community while helping students find relevant items and earn money. The project used a mix of technical tools like databases, user interfaces, and machine learning to make the platform work well. It successfully launched the StudentX website with features like user registration, item listing, and real-time messaging. StudentX is important because it fills a gap in the market by offering a safe and community-focused platform for students to buy and sell items. It helps reduce clutter during dorm move-outs and makes buying safer for students. In the future, StudentX could improve its recommendation system and expand to more universities. It could also consider showing ads to cover costs and keep the platform free for users. In conclusion, StudentX is a useful platform for students, making buying and selling easier and safer within the campus community. It helps build trust and connections among students, making campus life better.

Acknowledgment

Thank you to Professor Bo Yuan for advising the team to complete this project. We appreciate support from Rutgers University and Professor Sasan Haghani for the Introduction to Capstone and Capstone Design classes.

References

[1] https://firebase.google.com/docs
[2] https://flask.palletsprojects.com/en/3.0.x/
[3] https://pytorch.org/tutorials/
[4] https://console.cloud.google.com/billing
[5] CodeBrewLabs. "The Current Trends in Peer-to-Peer Marketplace Development." Medium, 06 November 2023, www.medium.com/@samiksha.shukla_2591/the-current-trends-in-peer-to-peer-marketplace-development-6f38e3c262ea. Accessed 29 April 2024.
[6] Dmytro. Oleksandra "Top 15 Online Marketplace Trends to Watch Out for in 2024." Codica, 18 January 2024, www.codica.com/blog/online-marketplace-trends. Accessed 29 April 2024.
[7] Dr. Bajwa's Introduction to Machine Learning ECE443 / ECE539 course lecture slides https://rutgers.instructure.com/courses/243699/files,