NavagantAI

Patrick Dewey, Aaron Gomez, Alex Jaimes, Akhil Kamalesh

1. Product Description

Over the course of this semester, we built a prospecting tool for our client Navagant, a Richmond-based mergers and acquisitions advisory company. The goal of our platform is to assist Navagant in finding new clients using intelligent systems alongside past outreach data. We achieved this goal by building a flexible web server that interacts with an intuitive user interface and various data routines/components such as calls to the Coresignal API and our prospecting model.

2. Functionalities

NavagantAI has 3 main functionalities: Historical CRM Uploads, New Companies Search, and Download Queried Results.

For our first main functionality, the Historical CRM Uploads, a user uploads both opened and unopened CRM data into our application which will be used in our in house machine learning model (spoken about within the model section as well as future directions).

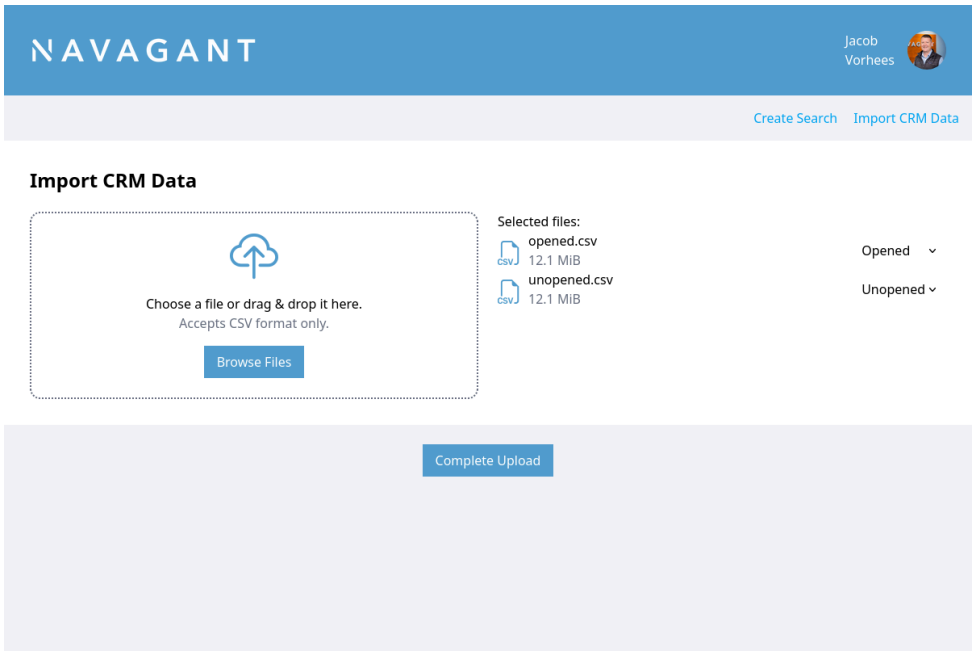

Figure 1: CRM Upload Page

For our second main functionality, New Companies Search, a user can search for companies based on certain parameters such as Sector, Location, Number of Employees, and Revenue. For this functionality, there are also soft constraints that users can search for such as Keywords that is used to provide the similarity score of how close a company is to that given search (Spoken about within the model section under similarity score). These parameters are then used to query the Coresignal API.

This functionality is the backbone of our product, since this transforms the way Navagant prospects for potential clients.

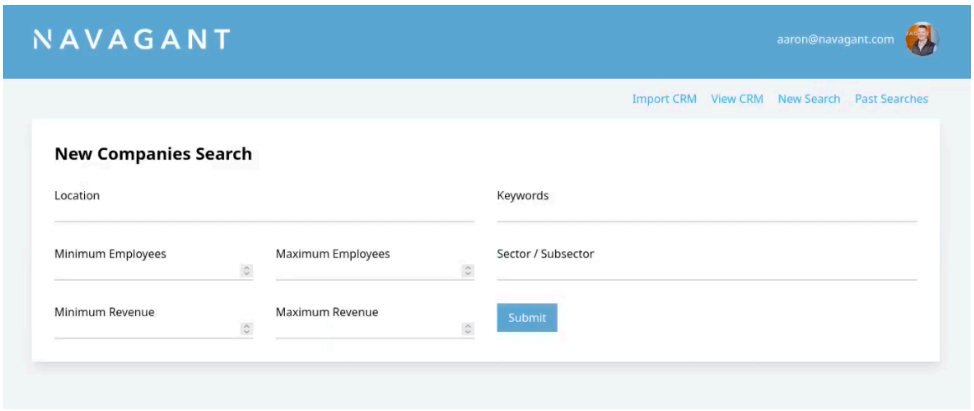

Figure 2: CRM Upload Page

Once the query is completed, our application will take you to our search results page where it displays the list of companies returned alongside with a multitude of attributes such as similarity score, company id, name, contact information, revenue, public status, etc. Within this page, you can also filter/sort through list of companies.

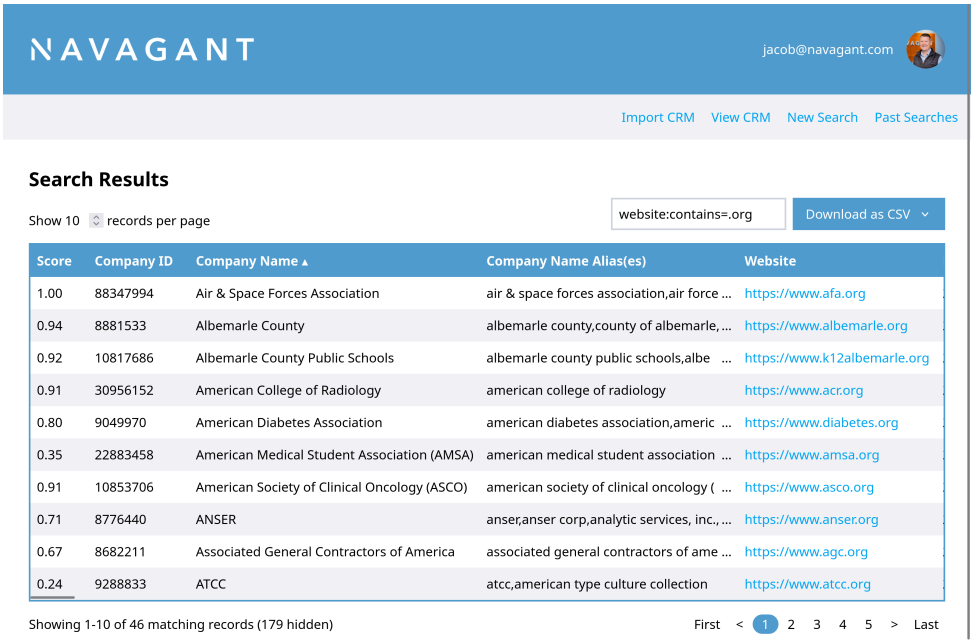

Figure 3: Search Results Page

Our last main functionality, Download Queried Results, allows a user to download the tabular results of the CoreSignal query in .csv format. This way, the bankers can have local access to the results, and this functionality reduces our storage overhead since we don't need to store the results.

There are other functionalities of our application such as Authentication and User Management, but the core of our application revolves around those three main functionalities.

2.1. User Walkthrough

As a user, the first step I take when using NavagantAI is logging in. Currently, authentication is set up in a way where the system admin has to add user permissions, so with my set username/ password, I log in to the system.

After logging in, the user is taking to the new companies search page. Within this page, a user can search for companies based on certain parameters such as Sector, Location, Number of Employees,

and Revenue. A user can also enter in key words which is used to give a similarity score for the companies returned. After a user enters in their specific parameters, they press submit.

This then leads the user to the search results page. This page displays the list of companies returned alongside with a multitude of attributes such as similarity score, company id, name, contact information, revenue, public status, etc. A user can sort the returned records by clicking on the column name. A user can filter through the returned records by entering their filter query in the filter box.

A user can then download the returned records as a CSV by clicking the Download as CSV button.

If a user has historic CRM data to input to the system, they would navigate to the Import CRM tab. They will input their “opened” and “unopened” CRM CSVs to the system. The user will press complete upload, and they will be directed to a page where they can view their CRM data. Similar to the view search results page, a user can sort/filter the same way as well as download the CRM data as a CSV.

If a user wants to look at their past searches, they click the past searches tab. This page displays the users past searches and the parameters associated with that.

If a user needs to change their account information, they click on their profile icon and select account. A user can edit their first name, last name, email, and password.

Lastly, if a user is a system administrator, they can manage user accounts. Within this page, a user can view the other users of the application and perform CRUD operations on user accounts.

3. Design

3.1. System Design

From a high level, our application is broken down into three subsystems: the backend, model, and frontend. Users interact directly with the user interface (frontend), which provides a number of different functionalities outlined above, and communicates with the backend through an internal API. The model and backend both run as web servers, with the model server only serving one endpoint `/predict`–which is used by the backend to solicit predictions–and with the backend serving a more expansive API that is used by the frontend. The backend also issues requests to the external Coresignal API to obtain company data.

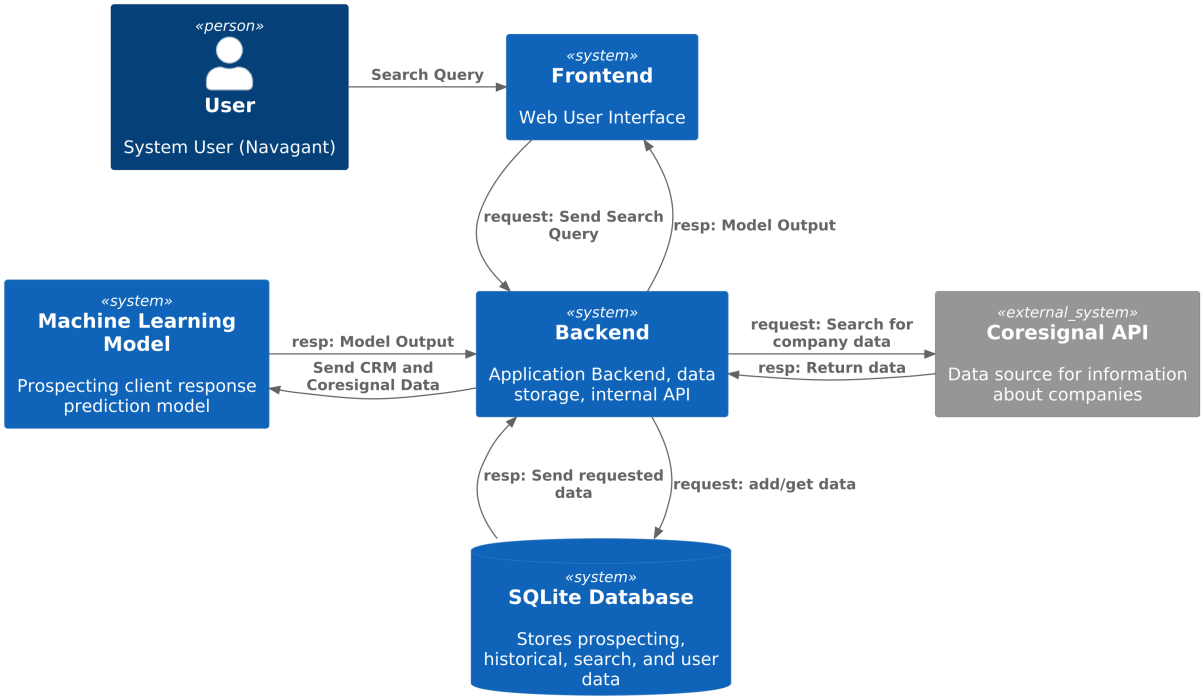

Figure 4: System Context

As seen in Figure 4, the backend facilitates communication between each of the other systems, allowing those systems to exist without needing in-depth knowledge of the whole system–each subsystem operates with only the knowledge of other subsystems required to operate.

3.2. Backend

Our backend system is responsible for storing and managing data, and for facilitating the flow of information within the application. It is also the system that performs all requests for searching and data retrieval with the Coresignal API.

3.2.1. Tools Used

We chose to write our backend using the Go programming language as it is generally well suited for web application backends, because it is very performant, for ease of adoption (it is a relatively easy language to learn), and because some of us have worked with it extensively in the past. Due to the size and strength of the Go standard library, our backend only has required direct dependencies, the "jmoiron/sqlx" [1] library for better SQL support, and the "gocarina/gocsv" [2] library for CSV serialization and deserialization.

For our data storage, we chose to use SQLite due to its very small footprint and ability to be embedded into the application itself. Due to the relatively low volume of data being stored, and the low likelihood of multiple concurrent reads/writes occurring, we did not think that the downsides of SQLite outweighed the additional technical complexity that would have been required for alternative database software.

Other tools that were pivotal to the development of the backend were the "gopls" language server for Go for completion and code diagnostics, the "gofmt" formatter for ensuring consistent code standards, Docker for deployment, and Neovim as the primary text editor used for the system.

3.2.2. Implementation

As seen in Figure 5, the backend system is broken down into a series of modules based on specific core application functionalities–those modules being the data module, the API handler, the aggregation module, the Coresignal module, and the coordinator module.

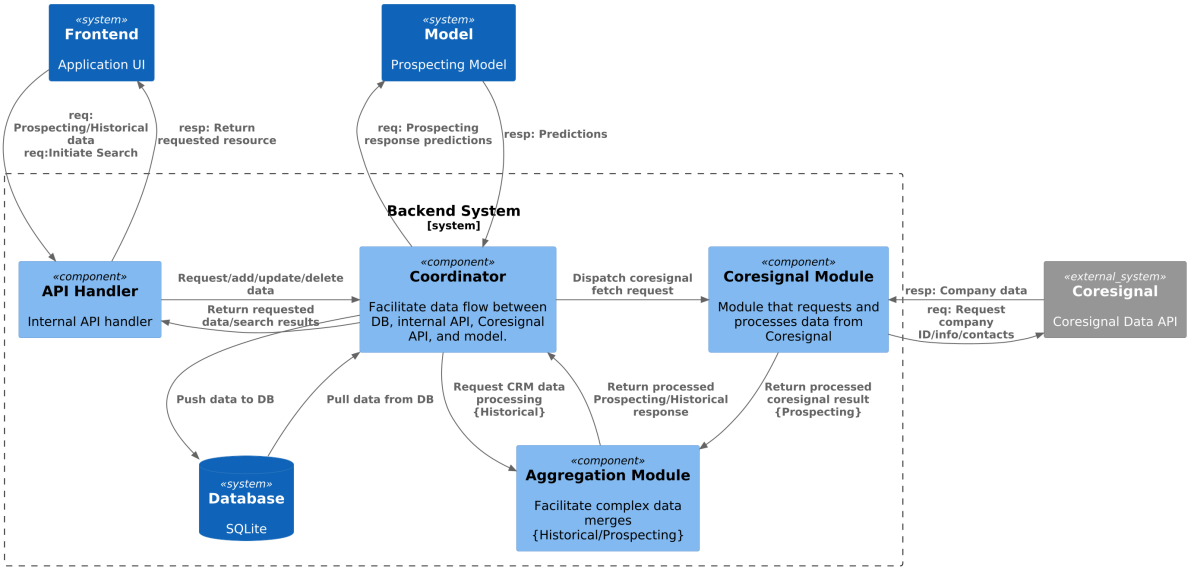

Figure 5: Backend System Architecture

The data module defines important data types (most notably `CRMData` and `CoresignalData`) and functions for working with them. It also initializes the SQLite database, and defines functions for creating, inserting, updating, and deleting entries in each table. The database creates and utilizes the `Searches`, `Users`, `Prospecting`, and `Historical` tables. The `Searches` table stores query terms used in Coresignal searches, and is used by the frontend to look up previous search data. The `Users` table is used for authentication and defines permissions for application users. The `Historical` table stores uploaded CRM data with additional data populated from Coresignal (schema is defined from the `aggregation.HistoricalData` struct type). The `Prospecting` table is populated with data collected from Coresignal and our model (schema defined from the `aggregation.ProspectingData` struct type).

The Coresignal module defines functionalities pertaining to searching and pulling data from the external Coresignal API. It includes functions for fetching company IDs that are used in prospecting searches, and additional functions for pulling information for each of the resulting companies. There are also functions for pulling company contacts for prospecting results (that prioritizes higher-level employees at the company), and functions for pulling additional information for companies included in uploaded CRM historical data.

The aggregation is the smallest module, as its role is simply merging data structures for historical and prospecting results. It merges CRM data with additional information from Coresignal, and merges Coresignal data with model predictions into the `ProspectingData` type.

The API handler module contains the code for our application's internal API. It defines endpoints for historical and prospecting data retrieval, CRM data upload, user authentication and management, issuing Coresignal searches, and retrieving previous search queries. After receiving and validating a request from the frontend, the API handler hands off tasks to the coordinator module.

The coordinator module acts as a middleman between the other modules (and the model system); it receives requests from the API and hands them off to the relevant module(s), returning responses where necessary. It is most closely linked with the API module, as the API handler largely does not

directly consume from the data, aggregation, or Coresignal modules–with the goal of minimizing coupling between packages.

3.3. Model

To the grader: This section is quite long and we go somewhat in-depth into some of the approaches we tried and failed. We would recommend going over sections 3.3 and 3.3.1. Sections 3.3.2 and 3.3.2 are more meant for any future team that decides to build on top of our product. While we do not discourage reading these sections, they are not reflected in the final product. We now cover a bit of background information necessary.

Navagant uses MadeMarket as their Customer Relationship Manager (CRM) software. CRM software is meant to help companies keep track of customer relationships and reach out to new customers via campaigns. MadeMarket manages email campaigns and keeps track of outreach emails sent out to potential customers. The "open state" of the email is also kept track of. The possible values of this state are "opened", "unopened", "invalid", and "bounced". Since "invalid" and "bounced" represent such a small percentage of all emails (< 5%), we simply ignore them or treat them as "unopened". Most of the modeling consisted of augmenting the CRM data via external data sources to fill out company and contact information and data preprocessing to attempt to predict whether or not an email would be opened or not. A successful model, i.e., a model that could, with high accuracy, predict who would open Navagant's outreach emails, would likely result in a higher number of clients. This would provide a great deal of value to any company.

Of course, any supervised model needs data to learn. This need for data is the main reason why we have the CRM data upload page, so that the analysts can upload information that would guide the modeling. While we have made the upload as painless as possible, we recognize that this might get tedious if there are a large number of files to be uploaded. As far as we know, MadeMarket does not offer any way to export distribution data programmatically, so we did not do this with code. Some potential future work could be accessing the files via Selenium or similar web navigation software so that the user does not have to do this manually. While this might not be possible, it is certainly worth exploring.

Unfortunately, our modeling efforts were fruitless in the email state prediction. While this is unfortunate, there is precedent to believe that this is possible and could be achieved with more efforts. Since Navagant requested an intelligent system, we still needed to create some sort of value with data science and machine learning tools. We have created an intuitive metric that should still provide significant value to the user in the search. The CRM data uploaded to the program remains largely unused, but we will outline why the page is still important in the *Future Directions* section. We will now go over our metric, our unsuccessful modeling attempts, and future directions that show promise.

3.3.1. Similarity Metric

Given Navagant's desire for an intelligent search system, we have created a metric using machine learning tools that provides great value to the users. This metric is a *search similarity* score. The way that we have built our search, we have *hard* constraints and *soft* constraints. Hard constraints are things that **must** be met such as a maximum revenue or a minimum number of employees. A soft constraint is more relaxed. The user has the option to input a number of keywords and industries about the company to guide the search. A company will match if the hard constraints are met and at least some of the soft constraints are met. There are pros and cons in how we've implemented our search. It is very broad and flexible, but it can sometimes catch companies that are not that relevant or do not "carry the intent or spirit" of the user's wishes. Creating a good search can be a difficult task, so we have tried to alleviate this with the creation of this metric. The *search similarity* score

helps us quantify how close each company is to the actual search given. It is a metric defined between 0 and 1, with 0 being a very poor fit, and 1 being a very strong fit. This allows the user to filter the companies if we only want some above a certain threshold for the “best candidates”.

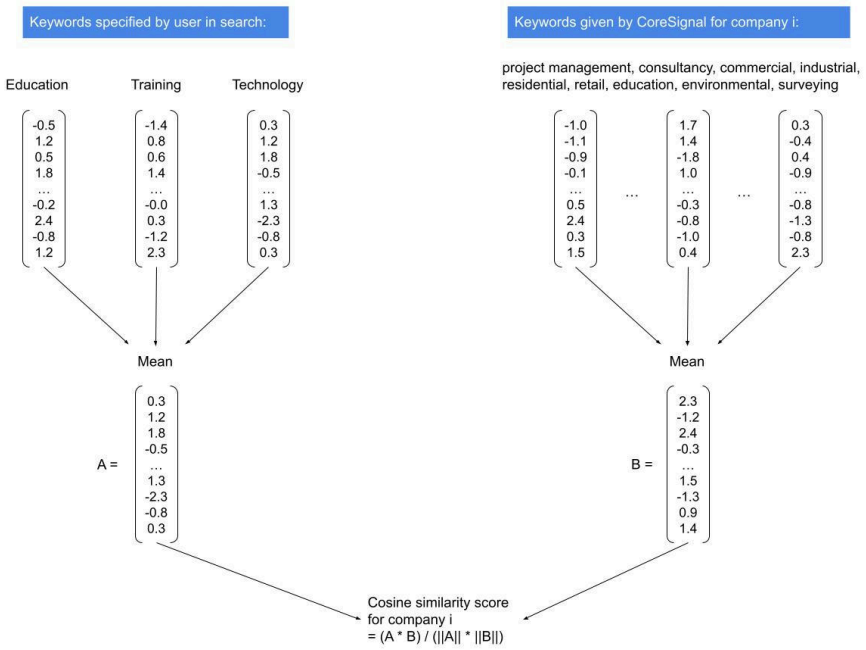

Figure 6: Our similarity metric process illustrated.

We now present a short explanation of how the *search similarity score* was created. We have taken the sector/subsector parameters and keywords given by the user and combined the two into a list. For each company, we are given a list of keywords associated with it from Coresignal. We take both the lists of user-given keywords and sector/subsector and each company’s associated keyword lists and turn them into a list of the corresponding GloVe word embeddings. GloVe word embeddings are vector representations of words and short phrases that convey syntactic meaning. What that means is that, in the vector space (in this case 50-dimensional), there is a direct relationship between (linear) distance and meaning. For example, the vectors representing frog and toad are very similar and thus very close in distance. Figure 7 provides a visualization of GloVe embeddings projected into two dimensional space. Now, we take the average of the vectors representing the search parameters (sector/subsector and keywords); we do the same with each company’s keywords. This average gives us a sort of ‘average’ representation of each company and the company that we are searching for. We then take the cosine similarity between them to obtain a metric of how ‘similar’ each company is to our search. Figure 6 is a diagram of the process just described. Of course, we decided on this GloVe approach as to avoid having very low similarity scores if we compared the raw words as that would be extremely restrictive (‘health’ vs ‘healthcare’ would result in no similarity.)

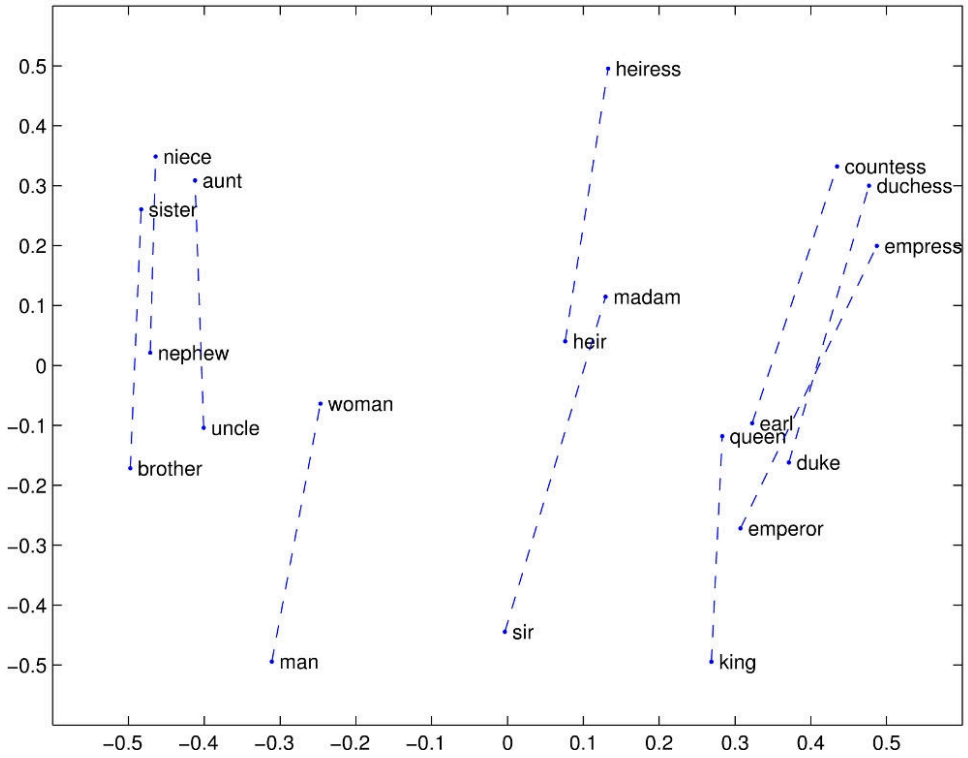

Figure 7: GloVe embedding example, image from https://nlp.stanford.edu/projects/glove/

In our testing, most of the values ended up somewhat close to 1. This makes sense as we would expect the companies to be similar to what we searched for. While the theoretical minimum score is −1, this would be nearly impossible to obtain as a company would need to be associated with keywords completely opposite to the search parameters specified. We tested our score some healthcare education companies with GloVe vectors for words like “kill” and “death” and obtained values around 0.1. Overall, we were very happy with how this score turned out.

3.3.2. Unsuccessful Approaches

We will now discuss approaches that did not end up making it to production. We document this mainly for two reasons:

- To serve as a guide for future students/workers that may want to attempt to build a machine learning model using the past data for email state prediction.
- To document work that was done, even if it is not included in the final code.

Before the capstone project officially started, we had already experimented with a number of approaches and data sources. These approaches generally consisted of joining company information obtained via some sort of data provider into existing MadeMarket data and training some non-linear model to predict whether a company would open an email or not. Some of the data sources which we tried include:

- Zoominfo: This is a platform which Navagant has a subscription to that allows users to manually conduct searches. We were limited to gathering information manually as Navagant’s subscription did not allow for programmatic access.
- Mixrank: This is a very similar platform to Coresignal. We detail why we picked Coresignal in the following paragraph.
- PitchBook: This is a platform similar to Zoominfo that Navagant also has a subscription to. The prices for programmatic access to the data were substantially higher than any other platform.
- Crunchbase: A very cost-effective option that we considered, but ended up dropping for higher data-quality platforms.

Through a number of months, we experimented with the data on these platforms, and, eventually, we landed on Coresignal. The main reasons why we picked Coresignal as our data provider are the following:

- Coresignal has a good price compared to other data providers.
- Coresignal has a number of useful variables on companies such as "description_enriched" (LLM description of a company), internet traffic, and funding information that could provide great value during modeling.

The main models that we tested were decision trees and random forests. This is because they tend to perform great empirically in most real-world situations [3] and, in contrast to other empirically successful non-linear models like Artificial Neural Networks (ANNs), the work great out-of-the-box so not too much effort is spent on getting a model ready. One big worry with trees and ensemble trees is their capacity for overfitting, but we saw that as an advantage. If we could get a model to learn the structure of the data, then overfitting would only be a small issue to fix. On the other hand, if our models simply lacked the ability to learn the desired mapping and they underfitted, then the problem would become unsolvable. This is why we picked the more complex models as opposed to starting with simpler linear models. Unfortunately, our complex (referring to the high number of degrees of freedom in the model) models were not able to make a significant improvement over random guessing. We mainly attribute this to the explanatory variables simply having no mapping to the response variable. In other words, we conjecture that no model, regardless of complexity, would be able to successfully predict the email state given the input features. We now outline some of the data augmentation and different approaches that we spent much of the semester on.

Coresignal gives us a number of qualitative (non-numeric) variables for each company. Some of these include the company's sector/industry, a list of keywords associated with the company, the company's Linkedin description, and an LLM-generated paragraph that describes the company. We tried to see if the well-established and powerful BERT model would be able to predict the email state using these variables. We placed some hope in this very advanced model as its transformer architecture makes difficult problems solvable. Unfortunately, the results were barely better than guessing. We also floated around the idea of including the email content as a predictor variable, but, upon closer inspection, all of the outreach emails were very similar and generic, so we decided to not test this as gathering the data would have taken too much time for what we thought was a low probability of success.

One of our approaches, initially suggested by Jacob, was to examine response rates when taking into account the contact's title. We approached this by turning the CRM variable "Job Title", which is a string containing the contact's job title(s), into a number of dummy variables. The logic for transforming these strings into dummy variables is somewhat complex as the CRM data formatting was inconsistent, a contact could hold multiple positions, and, depending on the processing, it is easy to accidently mix titles such as "Owner" and "Co-Owner". Our logic worked perfectly in the small sample that we manually tested, so we assume that there was no major fault in the preprocessing. In these variables, we capture whether a contact is a CEO, Co-Founder, Founder, President, Co-Owner, Owner, or none of these (i.e., something else). These values were picked because they comprise the large majority of the contacts in the CRM data. We added these variables to our models, but, unfortunately, they proved useless. We expected there to be some trends and/or interactions with some other variables such as CEOs of large companies being less likely to reply compared to CEOs of small companies, but no such trends were found. The dummy variables contributed near-zero explanatory power.

One of the main ways in which Navagant obtains clients is through attending a number of yearly industry conferences. This is where they get a list of contacts which they upload the MadeMarket.

We hypothesized that different conferences would have different email state proportions; perhaps companies that attend conferences specializing in medical training would be more likely to open the outreach email compared to companies in vocational training conferences. Unfortunately, MadeMarket does not record the conference as one of the variables. We manually added this conference variable to test our hypothesis. Unfortunately, we found no relationship between the conference and the email state proportions. Figure 8 displays this finding. This new variable was not tested in isolation, it was tested with other variables to find any potential interactions, but the search was in vain.

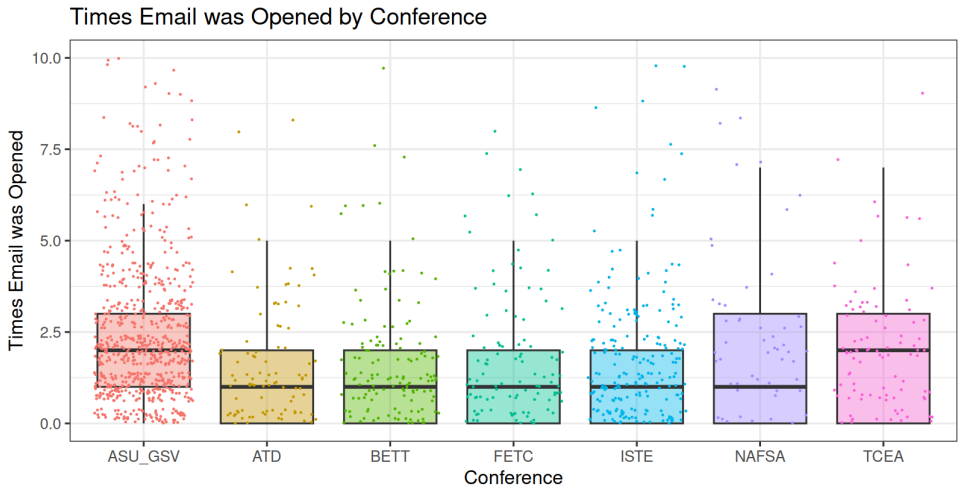

Figure 8: The number of times each outreach email was opened per conference, limited to 10 max.

In our desperation, we kept looking for any technique that would help out the classifier. Among them, we found a number of papers that claimed that clustering could enhance classification; one prominent paper we followed was [4]. The main idea that we experimented with was to cluster the data in hopes that the companies within each cluster mostly fall into either “opened” or “unopened”. This was one way to test if there were any spatial differences between the two classes which a classifier would be able to exploit. This clustering was made with k-means, a stochastic algorithm. Results initially appeared successful, as shown by Table 1 where each cluster seems to feature one class more prominently than the other one. Unfortunately, we found that this initial “lucky” distribution was likely due to random chance as the results were highly dependent on the random seed chosen and none of the other results were as great as the initial ones. This served as one final confirmation that there was no structural difference between the classes even with our newly created variables and the classification task was impossible with the current data.

Cluster	Opened	Unopened
0	52	31
1	1	0
2	1	1
3	7	4
4	1	0
5	1	0
6	34	19
7	96	76

Table 1: The class inbalance in each of the clusters seemed like a promising sign, but this was likely due to chance.

Because the data gathering and automating was sequential, our dataset slowly got bigger and more general. Since we tried very powerful, nonlinear models from the start, we did not believe weaker models to be worthy of testing. With our best models and approaches, we were able to achieve a balanced accuracy of around 55%. Balanced accuracy is defined as $\frac{\text{specificity} + \text{recall}}{2}$. We used this as our main metric as there were points in our testing where we had very unbalanced datasets, so this metric provided a more unbiased assessment of our model's performance than if we had just used plain accuracy. While this balanced accuracy is certainly better than guessing (especially given our sample size), we were only able to capture a small signal from the data. We do not believe this is a model worthy of deployment as it would not, in our opinion, make a meaningful difference in Navagant's outreach process.

We conjecture that perhaps we had a flawed vision from the start. Predictions for each individual business might not be the best way to approach this. We envision possible better systems in the next section.

3.3.3. Future Directions

One of our major breakthroughs in the modeling came from examining in-depth how one of our variables was being generated. In the CRM data, there exists a variable named "date_added". This was initially assumed to be the date when the outreach emails were sent out. Upon closer inspection, however, we realized that this was not the case. Unfortunately, MadeMarket did not add this "date_sent" as one of the variables in the data, but we were able to locate it for each batch of emails sent. From this, we manually and painstakingly added the variables into the dataset. We performed some exploratory analysis and came up with some very exciting findings. We first plotted (Figure 9) the email state counts through the years and found that there were a couple of large spikes while most of the time there were little emails sent. Secondly (Figure 10), we wished to see if, regardless of email counts, there were periods of higher receptivity for the emails being sent out. We identified some potential seasonal components. Finally, we plotted the mean email-opening rates per month in Figure 11. This final plot revealed some potentially strong seasonality that indicates when businesses are more likely to reply to outreach emails. Unfortunately, due to a lack of time we were not able to

explore this further, but this might be a genuinely interesting finding that could boost email-opening rates. We would also like to highlight that we only have around 18 months of data, so the results might not be truly representative of a true seasonality.

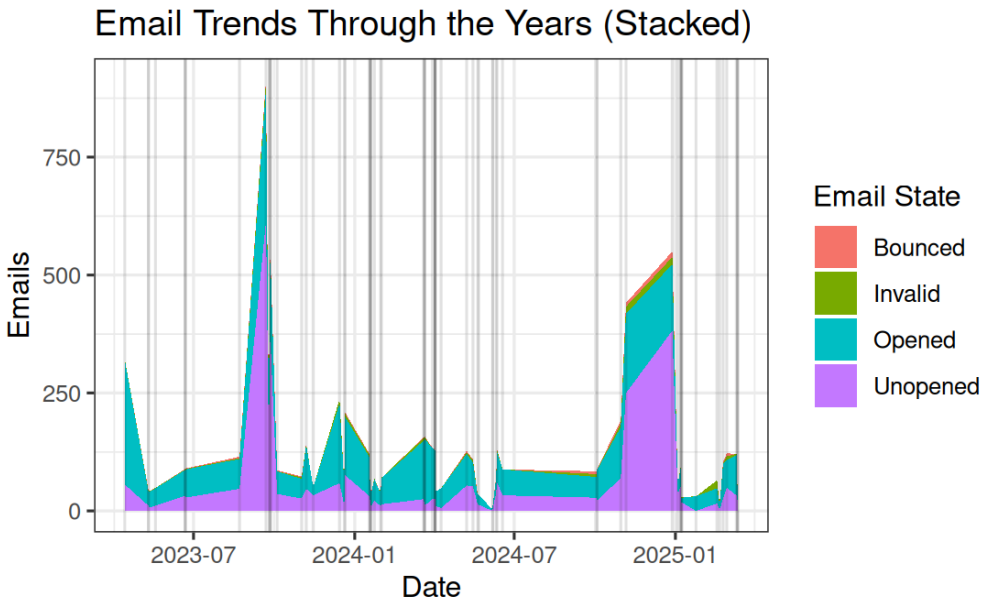

Figure 9: The email state proportions of the entirety of the CRM data. The vertical black lines represent when batches of emails were sent out.

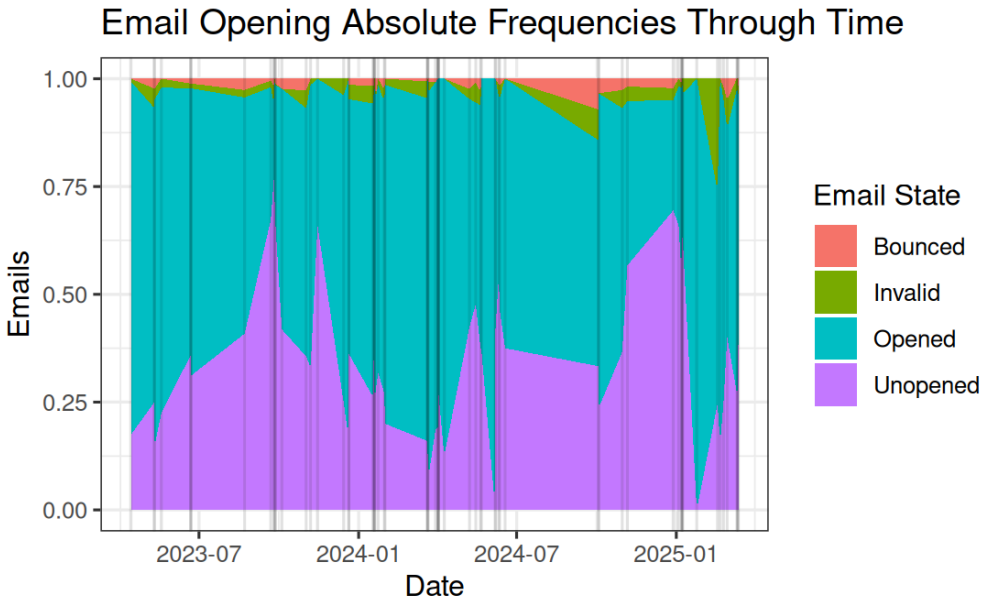

Figure 10: The absolute frequencies of the email state proportions suggest a seasonal component. The vertical black lines represent when batches of emails were sent out.

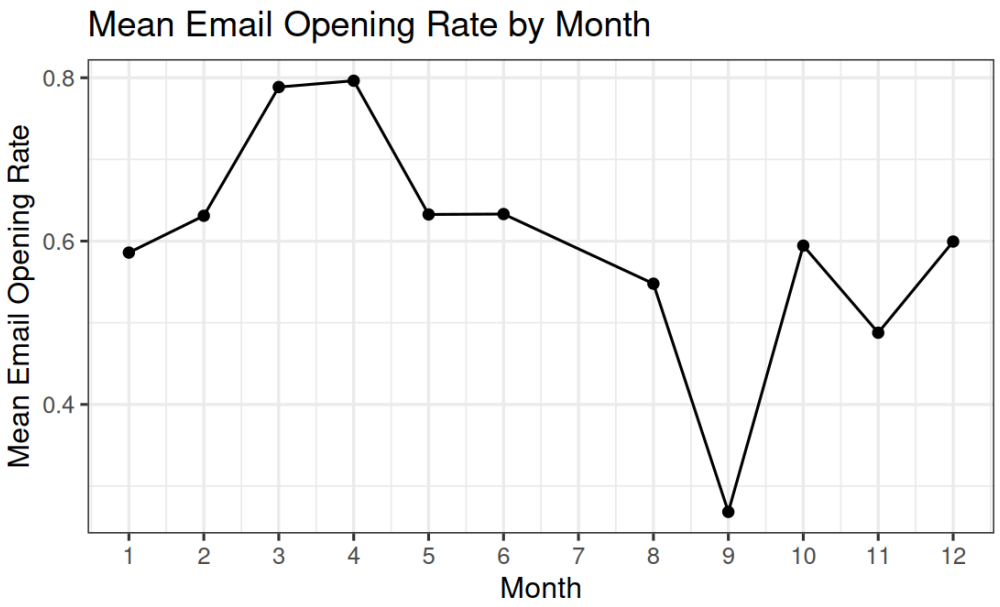

Figure 11: We noticed a clear seasonal aspect in email state through the year.

There were a number of papers that influenced this whole prospecting modeling exercise [5], [6], [7]. We would recommend going over the literature before deciding on how to continue this project's work. Some of the papers explored achieved great customer prediction accuracies, however, they often had access to variables which we did not have access to. A variable which some papers alluded to and Jacob mentioned too was whether a company was referred from a past customer or not. Another idea often floated around is that of "rival" companies. Companies similar to ones where a deal has been recently executed might make great customers as they have seen what deals Navagant is capable of. These two approaches/ideas (referrals and rivals) have not been implemented as we lack the infrastructure, but we think that they have a lot of potential.

Navagant has historically been a company that specializes in education and training companies, yet they have shared with us a desire to expand into different industries. Being able to find good customers in new industries is part of why they have asked us to come up with this prospecting software. A big difficulty is deciding how to build a system that incorporates past data regarding a certain industry and can still obtain customers in unknown markets. Our original vision for the system was a model that, akin to a Bayesian process, would use prior information on the industries in which some behavior had been previously established and could be exploited to obtain new customers and make little assumptions on new industries (uninformative prior). We still believe an approach like this would work great, once a successful modeling approach has been discovered and implemented for the known industries.

While we have floated the idea of writing personalized outreach emails using some language model to Navagant, they have not been too receptive to this. We would recommend reviewing the literature on the subject to see if a personalized email would yield better results than a generic one. Before starting this work, it is also worth to think about whether this would play nice with MadeMarket or if another section of NavagantAI would have to be built to track emails.

To combine and summarize our suggestions, we would recommend a future team to build some sort of model that captures the seasonalities across industries and takes referrals and rivals into account. A number of new pages would likely have to be added, for example a page that recommends which industry to target given the time of the year and a page that finds companies similar to those with who Navagant (or other firms) have struck a deal with. Better CRM integration will also likely be

needed to make the process more efficient and seamless. We would even go as far as potentially suggesting a change of CRM to one that is code-friendly. Of course, we recognize that this would be a great effort that would take significant commitment from the development and Navagant teams. Given the experimental nature of this whole project, it is difficult to know whether or not it would pay off, but the rewards could be enormous if it did.

3.4. Frontend

The frontend serves as the user's primary interface for interacting with the backend to create searches and manage CRM data.

3.4.1. Tools Used

For building the frontend, we opted to use the SvelteKit framework, as we found it to be significantly lighter than other more mainstream frameworks like React, without compromising on the features available. Though our team was overall more familiar with React, we decided to trust Svelte's consistently high rankings among other developers take the leap. Svelte and SvelteKit were quick to pick up and felt far less clunky than React projects have in the past. It offered plenty of state management and routing/templating options that were more than adequate for an interactive web app such as ours. We also chose to write the frontend in the TypeScript language, as it ensured type safety in the codebase and should improve project maintainability. Aside from those required by Svelte(Kit) and TypeScript, our project had a few other dependencies:

- `bootstrap-icons`: Provided a free, high-quality collection of icons for various parts of the UI.
- `csv-stringify`: Enabled exporting table records in CSV format following the (loose) specification.
- `tailwindcss`: Provided several utility CSS classes for consistent UI styling.

The final version is served from inside a Docker container for ease of deployment as static HTML/CSS/JS by the Nginx web server. Though Nginx's feature set may be overkill for our use case, it is a robust and widely-tested option with reasonable defaults and a relatively light resource footprint. Our team also found the Svelte extension for VS Code useful for the improved autocomplete over JS functions alone, as well as the Prettier formatter and ESLint for a clean, consistent code style.

3.4.2. Implementation

The frontend is split into a collection of reusable Svelte components, each responsible for a specific part of the interface and containing the accompanying logic. The most significant of these was the paginated table, used on several different pages of the web interface. It is likely the most complex component on the frontend, and has several responsibilities:

- Flattening nested data structures to display them in tabular form.
- Transforming columns to render them appropriately, such as showing URLs/emails as links, using emojis for booleans, or formatting dollar amounts.
- Applying various filtering and/or sorting options to explore data.
- Paginating records to avoid overwhelming the browser's DOM with elements.

We made several other smaller components as well to reuse their styles and functionalities throughout the project, such as the header on each page, the various form elements (inputs/labels), simple buttons, and a modal dialog. Communication with the backend is handled through asynchronous fetch functions, and the resulting state by Svelte to take advantage of the reactivity and update the UI accordingly.

The new prospecting search page is a standard form interface, offering fields for several search parameters to refine results: a location, employee count and revenue ranges, keywords, and a (sub)sector. Searches are sent as an HTTP POST request to the backend API, and the user

automatically redirected to the search results page to wait for results from CoreSignal. The results page continually polls the backend every two seconds to check if results from the search are available yet, timing out with an alert after five minutes, as this seemed to be a generous cutoff for even large result pools. Utilizing short polling should mitigate potential issues resuming dropped connections, and incurs a negligible performance penalty. Once the search completes, the results page's table will automatically be populated with the appropriate results, and the user allowed to browse through them. All filtering is performed locally to ensure responsiveness and again avoid connection troubles.

The CRM import page is implemented as a simple drag-and-drop file upload interface, with a sidebar to show the attached files and mark them as either the opened/unopened email records. Files are again submitted with a standard HTTP POST request, with icons providing upload status feedback to the user. The CRM data page's table is much simpler, as the uploads are (typically) near instant and thus do not require any polling.

Authentication is handled by a JSON web token (JWT) stored in a cookie with the Secure (only sent over HTTPS), HttpOnly (cannot be accessed via JS), and SameSite=Lax (mitigates cross-site request forgery) attributes. Using cookies allows for transparent authentication in the various fetch functions, as cookies are sent automatically by the browser with each request. JWTs were chosen for their simplicity, with each token containing only a user's ID and some expiry information. Storing the user ID enables stateless identity verification, since the tokens are signed with the backend's secret key to prevent tampering. Attempting to spoof any contained information will invalidate the signature and be denied. Tokens are also re-issued automatically with each request and will expire after seven days of inactivity.

The deployment-ready site gets compiled using the Vite build system into static HTML/CSS/JS files to then be served by Nginx. Interactivity is handled entirely by the compiled client-side JS code to make for a performant, high-uptime web app, as serving static files requires little additional complexity or resources.

4. Retrospective

Over the course of this project we all gained valuable experience in building a full-stack application with a backend, frontend, and machine learning model. We gained experience working on a team and planning a project with the agile methodology. We learned new tools and languages, and honed existing skills, most notably with the Go, Python, and TypeScript programming languages.

We learned how to implement an API, make requests to external APIs, interface with databases, generate similarity scores with word embeddings, and implement login/log-out authentication flows–all of which are very valuable for us going forward as computer scientists. We also learned how to use containerization tools like Docker and Docker Compose, and deployed a containerized version of our application to Digital Ocean.

We did have issues with shifting requirements over the course of the project–particularly in relation to data formats and requirements–but we were able to overcome these issues without too much difficulty. One very notably example was during the last sprint, when one major planned feature changed due to an unforeseen difficulty with our client. Our original plan was to use Microsoft Single Sign-On (SSO) to allow our client to use their existing Microsoft account infrastructure to sign into our web application, but this was not possible due to difficulties with Navagant's external IT company. While this was a fairly major setback, we were able to implement a replacement authentication solution, with the downside that it requires the creation of user accounts (rather than using existing ones).

We were very effective as a team, and we did a very effective job of outlining and delegating tasks, with each of us taking ownership of major parts of the project. We communicated effectively (and had meetings somewhat regularly), using Discord, email, and Zoom for messaging and synchronous meetings (Zoom for meetings with our client).

In terms of future improvements to our product, adding the aforementioned Microsoft SSO implementation would be a major improvement for our client, and we could likely improve the model to generate prospect response predictions in addition to the similarity score. It could also be improved with expansions to the user management settings, with SMTP-powered email password resets being an obvious choice.

While we are very happy with the end result of our project, there definitely were rooms for improvement in the way we managed it. Notably, we defined very broadly encompassing user stories for our product, which led to us not finished any of our major stories until the last sprint (when we finished all of them). We should have broken down our core stories into smaller ones that we could have finished earlier, ideally with major parts being finished in each sprint. We also didn't fully utilize our issue board for task management until mid-way through the second sprint. It would have been easier to manage and delegate tasks (and understand what other group members were working on), if we had been more active in adding and updating issues on the board.

Overall, we are very happy with our final product, as we were able to deliver on all the major functionalities we originally planned, and we will be handing off a high quality, well documented code base to our client.

5. Project Links

Note: Due to client NDA, repository access is limited to a whitelist on both GitLab and GitHub. DLQF Lab members should use GitHub, CS-5934 graders should use the GitLab links.

- Backend Repository (GitLab): https://code.vt.edu/droptable/backend
  - GitHub Mirror: https://github.com/DLQF/navagantai-backend
- Frontend Repository: https://code.vt.edu/droptable/webui
  - GitHub Mirror: https://github.com/DLQF/navagantai-frontend
- Model Repository: https://code.vt.edu/droptable/model
  - GitHub Mirror: https://github.com/DLQF/navagantai-model

References

[1] J. Moiron, *sqlx*. [Online]. Available: https://github.com/jmoiron/sqlx

[2] V. Néel and J. Picques, *Go CSV*. [Online]. Available: https://github.com/gocarina/gocsv

[3] D. Nielsen, "Tree boosting with xgboost-why does xgboost win" every" machine learning competition?," 2016.

[4] M. Piernik and T. Morzy, "A study on using data clustering for feature extraction to improve the quality of classification," *Knowledge and Information Systems*, vol. 63, no. 7, pp. 1771–1805, 2021.

[5] S. Farhang, W. Hayes, N. Murphy, J. Neddenriep, and N. Tyris, "A Deep Learning Approach for Imbalanced Tabular Data in Advertiser Prospecting: A Case of Direct Mail Prospecting," *arXiv preprint arXiv:2410.01157*, 2024.

[6] S. Khodabandehlou and M. Zivari Rahman, "Comparison of supervised machine learning techniques for customer churn prediction based on analysis of customer behavior," *Journal of Systems and Information Technology*, vol. 19, no. 1/2, pp. 65–93, 2017.

[7] Y. Zhang, "Prediction of customer propensity based on machine learning," in *2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, 2021, pp. 5–9.