INVOCA

Service Buddy Project Report

Students: Amy Cho, Antón de la Fuente, Austin Quinn, Sai Kathika, and Shuyun Tang
Mentors: Mike Lawrence, Prof. Ambuj Singh, and Sikun Lin

UCSB

Abstract

For our project, we wanted to create something that roughly imitates Invoca's Active Conversation product, which turns conversation data into automated action--though on a simpler scale. Instead of live speech data, we mostly worked with Taskmaster-1, a google research dataset that has 13,215 dialogs. The data set contains written dialogs created by workers who wrote the full conversations on their own, acting as both the user and the assistant. As a result, many of the conversation segments seem a bit awkward and forced. Our main question of interest that we sought to solve was whether the models trained on the dialogs could also make accurate classifications based on users speaking colloquially or with mistakes.

Our project can be divided into three main components: the chat client, which are two message boxes connected to a server that the customer and agent can use to communicate with each other. Their messages are sent to the machine learning models, which make a classification predicting which of the six subjects the customer is talking about (Pizza, Movie, Rideshare, Auto Repair, Restaurant, or Coffee). These classifications are made based on important words they use and context they give. The models then give their best guess on the subject to the assistant, which then pulls up forms, links, or other helpful information and sends them to the chat client, where the Agent can use them to make their lives easier.

Our group completed our Minimum Viable Project, in which the Client feeds conversation information to our Models which then consolidate to feed a classification to the assistant, which completes the full circle and gives helpful links back to the Client. The logistic regression model that we decided to use outputs relatively accurate predictions, though sometimes when there is too much ambiguity between the topics the models may choose a topic other than the one the user had in mind. Alternatively, when we use a BERT model, the prediction seems to be much more accurate in detecting small implications of what the user actually wants when they say keywords that might correspond to multiple topics. However, the BERT model takes much more computational power, so we decided to still go with the logistic regression models for our MVP.

Introduction

The primary goal of our project was to build a chat client assistant that uses machine learning models to predict what topic the user is asking a customer service agent about. After the prediction is made, links to relevant websites are automatically given to the user by the assistant, thus saving the customer service agent precious time. There are millions of customer service agents in the United States and they're only growing in demand. By implementing our assistant, we allow agents to gain more information from each call and be able to reach out to more customers in general, making both the customer's and the assistant's work easier.

We had to do some research to help us develop some expertise in fields relevant to our project. To understand the data better, we referred to the documentation for the Google Taskmaster-1 dataset (https://github.com/google-research-datasets/Taskmaster/blob/master/TM-1-2019/README.md). Furthermore, we had to understand the relationships behind different words as well as which common words were irrelevant enough to the meaning of a sentence to just omit altogether (https://en.wikipedia.org/wiki/Most_common_words_in_English). We used some traditional machine learning tools like Scikit-Learn's TfIdf vectorizer, 5-fold cross-validation using a Random Forest Classifier, a kernelized Linear Support Vector Classifier, a Multinomial Naïve Bayes classifier, and a Logistic Regression Model (https://scikit-learn.org/stable/). Lastly, to improve the accuracy of our models, we used the Simple Transformers API (https://simpletransformers.ai), which is built on top of the Hugging Face API.

The Google Taskmaster-1 dataset's 13,215 dialogs are meant to represent conversations between a user and a customer service agent, so we figured that it would serve our purpose well. There were even some dialogs with typos to simulate more realistic conversations, which was more aligned with the type of colloquial conversations that we had hoped to test our product on. With all these tools at our disposal, we were ready to dive into the details and the implementation of our Service Buddy and answer our question of whether we could create an assistant that could actually save a customer service agent time by sending the user helpful links.

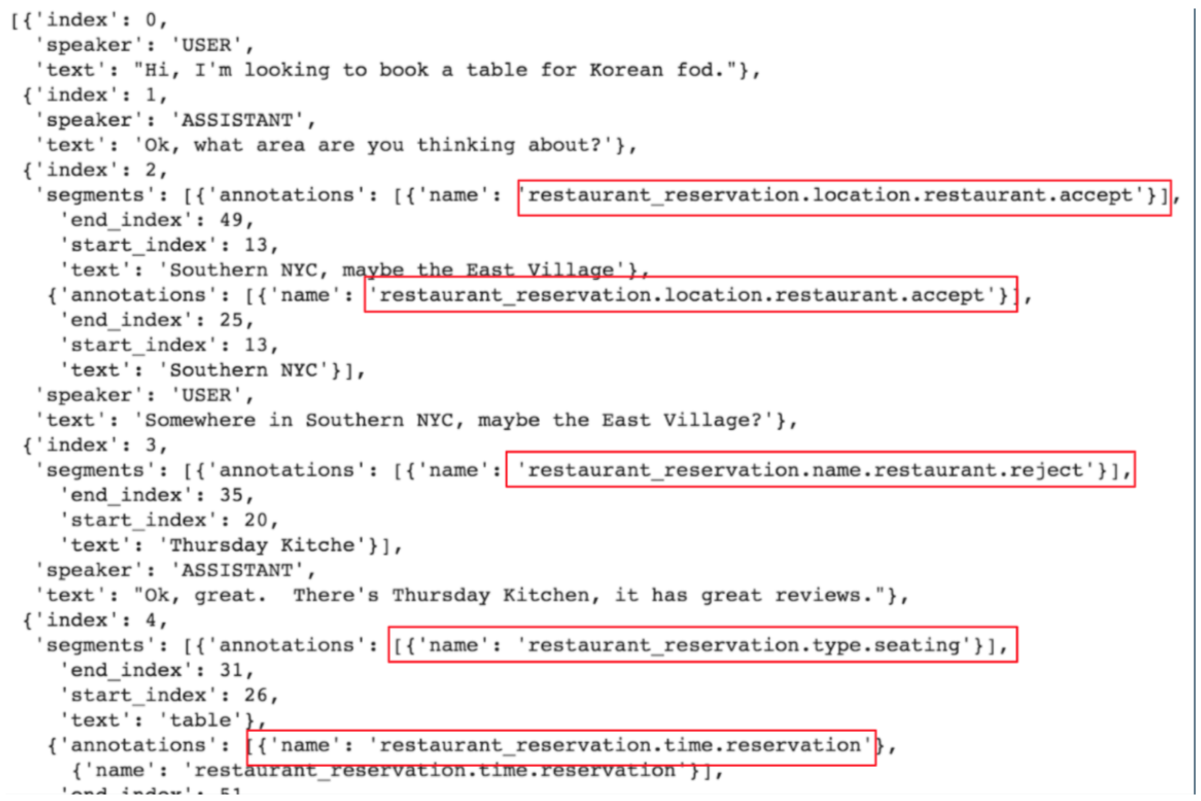
Data

As discussed above, we used a dataset called Taskmaster-1 for our project, which we sourced from a google research dataset. It has 13,215 dialogs, 5,507 of which are spoken dialogs and 7,708 that are written. These dialogs are meant to represent conversations between a user and an assistant, and were created through two different methods. The woz dataset contains spoken conversations between two distinct people, with the assistant role being fulfilled by call center professionals. The solo data set contains written dialogs created by workers who wrote the full conversation on their own, acting as both user and assistant. We did not integrate the two datasets due to the difference in method for data collection, but instead ran models on the datas separately to see which had better generalizability. There were no missing values as the data just consists of nested jsons representing the conversations.

All of these dialogs can be categorized within the 6 domains of ordering pizza, ordering movie tickets, ordering coffee, making a restaurant reservation, making auto repair appointments, or setting up a rideshare service.

The jsons for each conversation can be broken down into the following components: a conversation ID, the “utterance”, which is the array of each dialog or line in the conversation. The utterances are quite nested, also coming with several labels such as an ordering index, speaker indicator, the actual text of the dialogs, as well as segments. Within segments, there is another layer of annotations. These annotations associate labels with specific phrases and classify associated lines of dialog. The annotations indicate the category or topic of the overall conversation, and have more specific sub labels such as location, or number of guests, and whether the suggestion by the chat assistant was accepted or rejected.

If we take a look at the image below, in the red box, we see this annotation specifies that the overall conversation is about a restaurant reservation, and the specific line indicates the user has accepted the chat assistant’s suggestion for restaurant location. Other important annotations we see throughout this conversation also in the red boxes are like the type of seating, number of guests, or time for the reservation -- all accompanied with an accept or reject label.



```
[{'index': 0,
  'speaker': 'USER',
  'text': "Hi, I'm looking to book a table for Korean fod."},
 {'index': 1,
  'speaker': 'ASSISTANT',
  'text': 'Ok, what area are you thinking about?'},
 {'index': 2,
  'segments': [{'annotations': [{'name': 'restaurant_reservation.location.restaurant.accept'}],
    'end_index': 49,
    'start_index': 13,
    'text': 'Southern NYC, maybe the East Village'},
   {'annotations': [{'name': 'restaurant_reservation.location.restaurant.accept'}],
    'end_index': 25,
    'start_index': 13,
    'text': 'Southern NYC'}],
  'speaker': 'USER',
  'text': 'Somewhere in Southern NYC, maybe the East Village?'},
 {'index': 3,
  'segments': [{'annotations': [{'name': 'restaurant_reservation.name.restaurant.reject'}],
    'end_index': 35,
    'start_index': 20,
    'text': 'Thursday Kitche'}],
  'speaker': 'ASSISTANT',
  'text': "Ok, great.  There's Thursday Kitchen, it has great reviews."},
 {'index': 4,
  'segments': [{'annotations': [{'name': 'restaurant_reservation.type.seating'}],
    'end_index': 31,
    'start_index': 26,
    'text': 'table'},
   {'annotations': [{'name': 'restaurant_reservation.time.reservation'},
     {'name': 'restaurant_reservation.time.reservation'}],
```

For the purpose of our project, we wanted to separate each line in the conversation, and then isolate the user lines in order to help simulate or create predictions on the topic of the user language and use these predictions to create our assistant responses. Not only that, we focused on the extraction of these labels or annotations for creating our classification models. To fulfill these requirements we created a dataframe of just the user lines, along with any associated annotated labels, and then cleaned the data by running it through various preprocessing functions. For data preprocessing, we extracted the most important elements in the jsons that we thought were relevant for our analysis and formed them into a tabular dataset. Our final dataset consisted of a conversational ID, the conversation text, the speaker, and annotations.

However, because our project goal was to create a chat assistant that will be able to assist a client with any query that does not necessarily exist in the 6 categories detailed in our available data, there were some concerns that we had to address on how representative our data was for the goal that we want to achieve. These issues stem from the collection methods for this conversational data. For our solo data, the majority of the conversations were collected in a staged, artificial way. Having one person write both sides of a conversation on a set list of 6 topics does not necessarily accurately reflect how a client may interact with our chat assistant for any number of issues they would like help with. As for the woz data, it can be considered a little closer to how actual conversations are held, but at the same time having two distinct speakers does not address the issue of limited topics in our data. It is unrealistic to expect that all of the clients utilizing our chat assistant will have an issue confined to the 6 aforementioned categories. Because of these artificial collection methods, there is some bias as well in the data, as the conversations do not flow as a client and assistant might converse. A more relevant set of conversational data would cover a wider range of topics and contain more organic conversations between two people.

In order to address these issues, we generalized the two types of requests our chat assistant will receive, which are advance appointments, such as a reservation style request, or real time requests. For the taskmaster data, movie ticket reservations, restaurant reservations, and auto repair appointments would fall under the advance appointment category. Whereas pizza orders, coffee orders, and ride share services all would indicate a real time service. By having a broader catch all category, if for example a customer wants to order a burger and not a pizza, the structure of the language to order a burger and a pizza are not so different. These similarities are what we expect our model to classify in the same broader category as a pizza delivery and will respond in a similar manner for assistance.

The benefits of data collection in this manner are the lowered costs for relatively accurate imitations of conversational data. Not only is it more accessible and less expensive to pay workers to create conversations within a limited topic list, there are no ethical concerns regarding privacy to attain real life conversations.

If anything, any possible ethical considerations for this dataset would revolve around representation. Google created and collected this conversational data to fuel potential nlp projects and relevant projects, so the majority of the conversations are in standard, “proper” english, and there is a distinct lack of slang or conversational cues that could be associated with different demographics, such as different age or cultural groups. Besides that, there aren’t many ethical considerations as these conversations were not collected in a way that violated privacy. Although there is somewhat of a trade off between collecting accurate, more organic conversations and potentially violating privacy or opening up an avenue for information mishandling with collecting more life-like conversational data.

Methods

Latent Dirichlet Analysis

Our main exploratory tool for our data was Latent Dirichlet Analysis. LDA is a generative model that treats a topic as a probability distribution over a set of words found in the dataset, and a document as a probability distribution over a set of topics. The model takes in a number of topics as a hyperparameter and calculates the distribution of words for each topic. The amount of topics was an interesting number to play with, since it gives some idea of how the data clusters. Since our dataset contains 6 classification categories, we initially tried to get 6 topics, but it was difficult to get coherent and well separated topics. Eventually it became clear that 5 topics are best to describe our dataset.

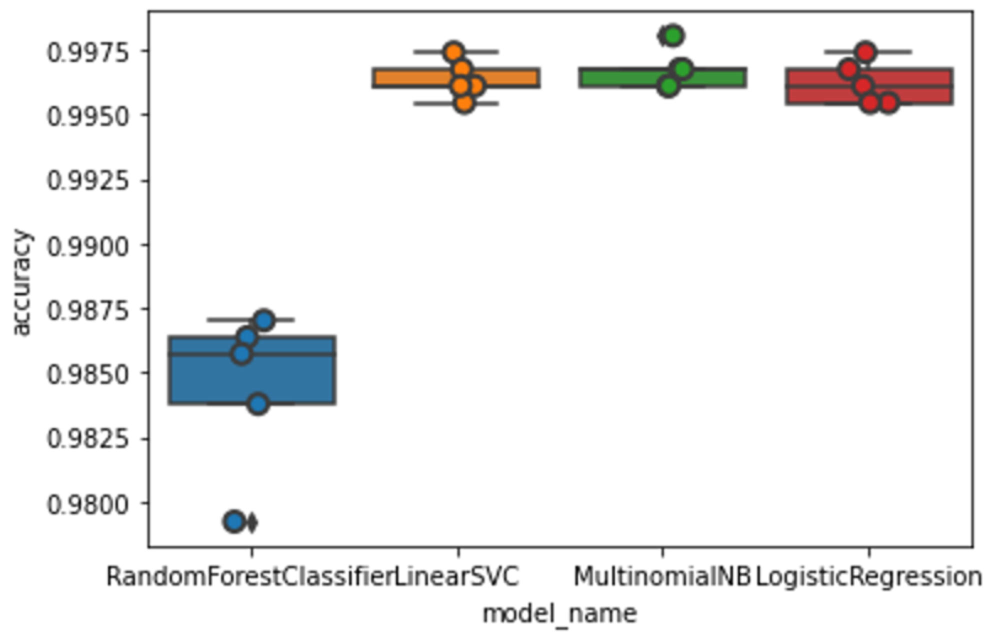
Because topics seemed to be cleanly separated, I started to wonder whether it would be useful to turn those five topic probability distributions into features. For this I did a train test split and retrained an LDA on the training set. After doing some cross validation we arrived at a Random Forest model and a logistic regression model with roughly the same accuracy.

The models generalized well to held out data from the solo dataset. We can see that the topics seem to correspond to the categories. It looks like the two categories with the worst performance are the ones joined into one topic for the model used. After seeing this I tried to train some models on a 6 topic model and I found similar results, where at least two categories had a drop in performance. This led us to question how easy it is for a bag-of-words model to deal with our dataset, since the features are individual words and these are conversations it's possible that a sequence model could deal with classification.

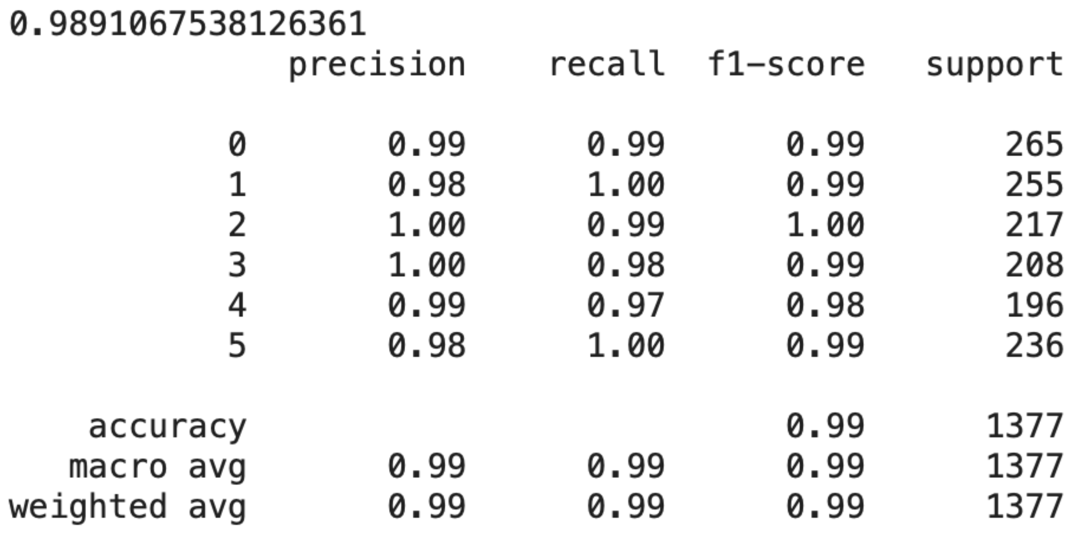
Traditional Machine Learning Methods – Logistic Regression & TfIdf

Once we had some understanding of how our data was distributed and we realized that LDA topics don't work well as features, part of our team focused on using more traditional NLP feature extraction approach and used Scikit-Learn's TfIdf vectorizer (sklearn). In this approach, each word is represented as its frequency in the entire dataset divided by the inverse of the amount of documents it appears in. This is done so that words that are ubiquitous and functional like 'The', or just very common, like 'have' (Most Common Words), get weighted down. When we do this, words that are very common in but only specific documents are scored highest since they are most likely to encode the important information in each document.

Using our TfIdf features we ran a 5-fold cross-validation using a Random Forest Classifier, a kernelized Linear Support Vector Classifier, a Multinomial Naïve Bayes classifier, and a Logistic Regression (Sci-kit-learn). The results of this cv are summarized in the figure below in a box and whisker plot with a strip plot overlaid (each point represents a model outputted by the CV):

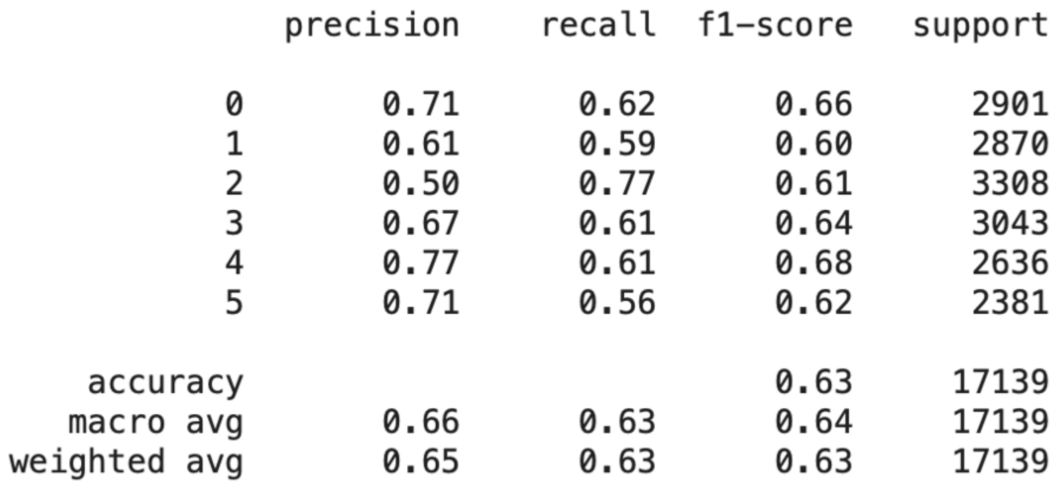


We can see that all of the models were highly accurate, with the Linear SVC, Multinomial Naïve Bayes. In the end, due to our team's familiarity with the algorithm and the negligible differences in accuracy, we settled on a logistic regression for our final model. The best performing hyperparameters were the default for scikit-learn, and we used a random state of 0 for reproducibility. We trained and tested this model on our dataset, the results are summarized below:



```
0.9891067538126361
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       265
           1       0.98      1.00      0.99       255
           2       1.00      0.99      1.00       217
           3       1.00      0.98      0.99       208
           4       0.99      0.97      0.98       196
           5       0.98      1.00      0.99       236

    accuracy                           0.99      1377
   macro avg       0.99      0.99      0.99      1377
weighted avg       0.99      0.99      0.99      1377
```

Because the results here are better than expected, we trained and tested this model on a modified version of our dataset, where the model is fed individual lines for each conversation for prediction. This was meant to simulate the messy nature of a customer service interaction, where it is not guaranteed that a full and coherent conversation will be fed to the model. Model performance on this dataset is summarized below, and represents a drop compared to predictions run on each full conversation:



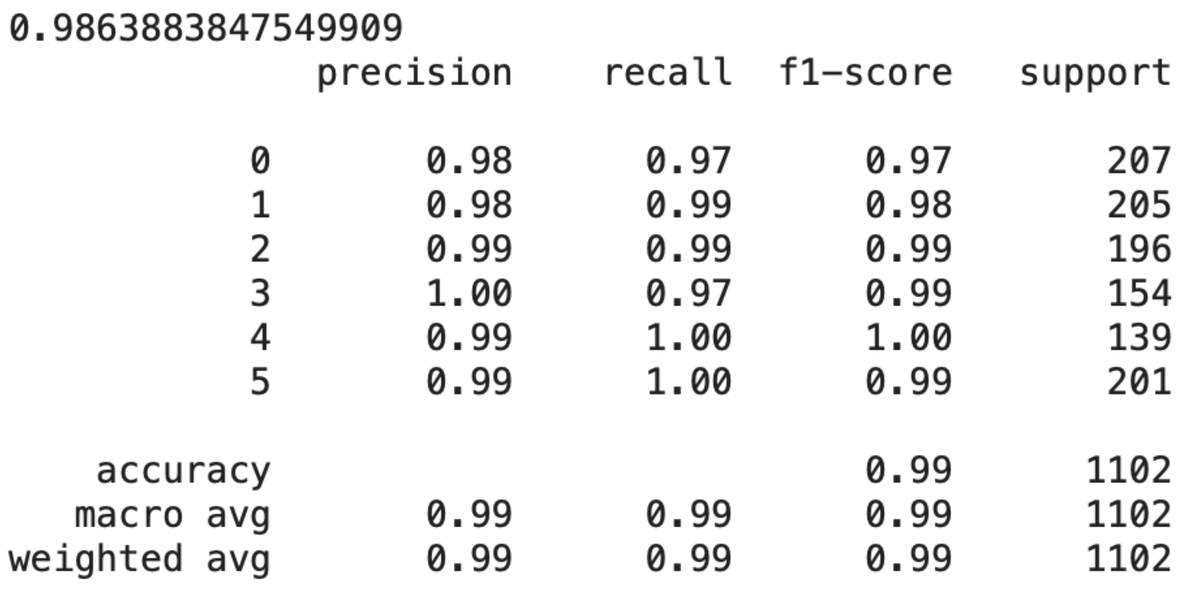
```
              precision    recall  f1-score   support

           0       0.71      0.62      0.66      2901
           1       0.61      0.59      0.60      2870
           2       0.50      0.77      0.61      3308
           3       0.67      0.61      0.64      3043
           4       0.77      0.61      0.68      2636
           5       0.71      0.56      0.62      2381

    accuracy                           0.63     17139
   macro avg       0.66      0.63      0.64     17139
weighted avg       0.65      0.63      0.63     17139
```

The results above seem very poor, however when we consider that a large majority of individual lines are not very informative, it becomes clear that the accuracy is significantly down-weighted. This works as a lower bound of model performance, instead of an accurate representation of actual model usage. As our last step, we saved this model with our fit TfIdf vectorizer in a scikit-learn pipeline for deployment in our system.

Neural Networks – BERT

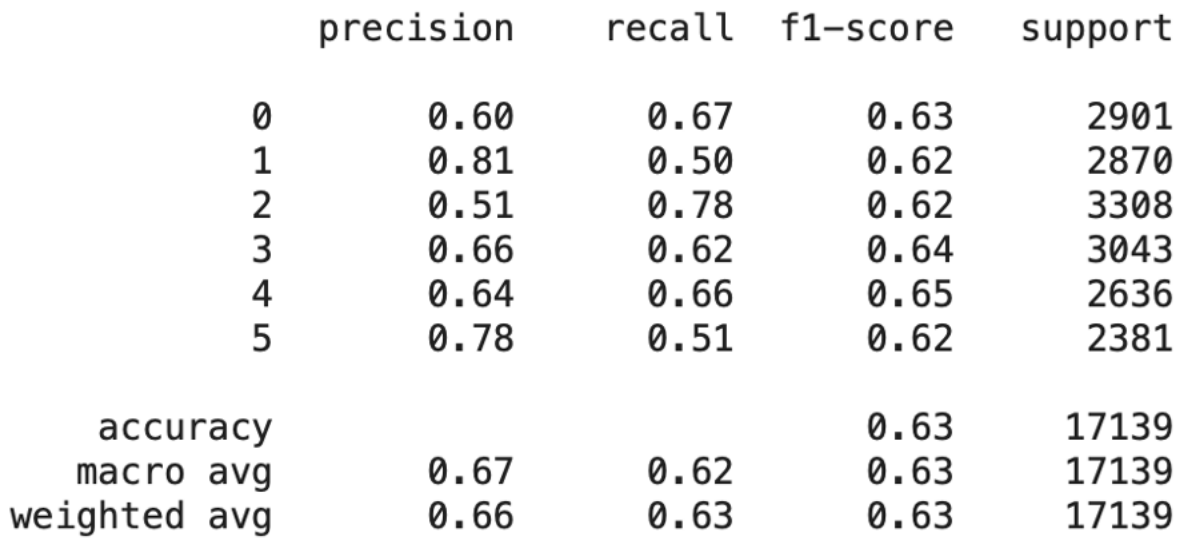
Motivated by our LDA results, part of our team focused on using a sequence model to do classification. We chose BERT, a popular state-of-the-art transformer model. Transformer models are deep learning architectures that can account for sequences by using the attention mechanism (Devlin et al.) instead of using computationally demanding recurrent layers. Transformers instead encode an entire sequence and weight the importance of each element in the sequence (the self-attention mechanism). BERT is a pre-trained Neural Network that can be fine-tuned for downstream tasks such as classification, making it useful for our purposes.

For our implementation, we made use of the Simple Transformers API (Raajapakse), which is built on top of Hugging Face, another API that can make use of either Tensoflow or Pytorch. Simple transformers was chosen because it made fine-tuning BERT for classification very simple and easy, and since our application didn't call for any sort of detailed hyperparameter tuning this was the most convenient API. Training was done on a Colab notebook with GPU (NVIDIA Tesla K80), and the model was then deployed on our app's environment, using a CPU. This proved to be an issue later on, and will be discussed below.
We trained our model using the same datasets as above and had some marginal improvement in our results on both the original dataset and the single line dataset. Results for the full conversation dataset are summarized below (we used a slightly different train-test split than for the Logistic regression):



```
0.9863883847549909
              precision    recall  f1-score   support

           0       0.98      0.97      0.97       207
           1       0.98      0.99      0.98       205
           2       0.99      0.99      0.99       196
           3       1.00      0.97      0.99       154
           4       0.99      1.00      1.00       139
           5       0.99      1.00      0.99       201

    accuracy                           0.99      1102
   macro avg       0.99      0.99      0.99      1102
weighted avg       0.99      0.99      0.99      1102
```

Results for the single line dataset:



```
              precision    recall  f1-score   support

           0       0.60      0.67      0.63      2901
           1       0.81      0.50      0.62      2870
           2       0.51      0.78      0.62      3308
           3       0.66      0.62      0.64      3043
           4       0.64      0.66      0.65      2636
           5       0.78      0.51      0.62      2381

    accuracy                           0.63     17139
   macro avg       0.67      0.62      0.63     17139
weighted avg       0.66      0.63      0.63     17139
```

Surprisingly the models have essentially the same performance, which goes against the hypothesis that we drew about bag-of-words models above.

Practical Considerations – Loading and Prediction Time

We needed some factor to decide between which model to use. BERT is a deep neural network, and when it's deployed on a CPU it can be very slow to load and it can also take some time to make predictions. On the other hand, logistic regression models are very simple and computationally efficient. In order to compare the efficiency of the models we looked at the loading time for each model in 100 iterations and took some summary statistics. Loading times summarized in the table below:

Loading Times

	Logistic Regression	BERT
Average	0.03616147518157959	2.5777918672561646
Standard Deviation	0.007062063959123201	0.39994809116168595
Max	0.10477280616760254	5.0834009647369385
Min	0.03266310691833496	2.2447962760925293

We also timed prediction time for a subset of 100 conversations in the smaller WOZ dataset, results summarized below:

Prediction Times

	Logistic Regression	BERT
Average	0.0009344015057385238	6.727950773239136
Standard Deviation	0.00017769722312244236	0.3256914815720181
Max	0.0030469894409179688	7.861032962799072
Min	0.0007290840148925781	5.911123991012573

Prediction Times

	Logistic Regression	BERT
Average	0.0009344015057385238	6.727950773239136
Standard Deviation	0.00017769722312244236	0.3256914815720181

Max	0.0030469894409179688	7.861032962799072
Min	0.0007290840148925781	5.911123991012573

BERT takes longer to load and it has significantly slower prediction times on a CPU. This proved to be a real obstacle for our implementation, so we decided to build our prototype with the Logistic Regression model instead. This fact may change with access to fast GPUs and by caching the model in our app. Unfortunately we did not have enough time to figure out how to cache the model, and although we have GPU access through Google Colab, that's a less than ideal environment to run a Flask app on.

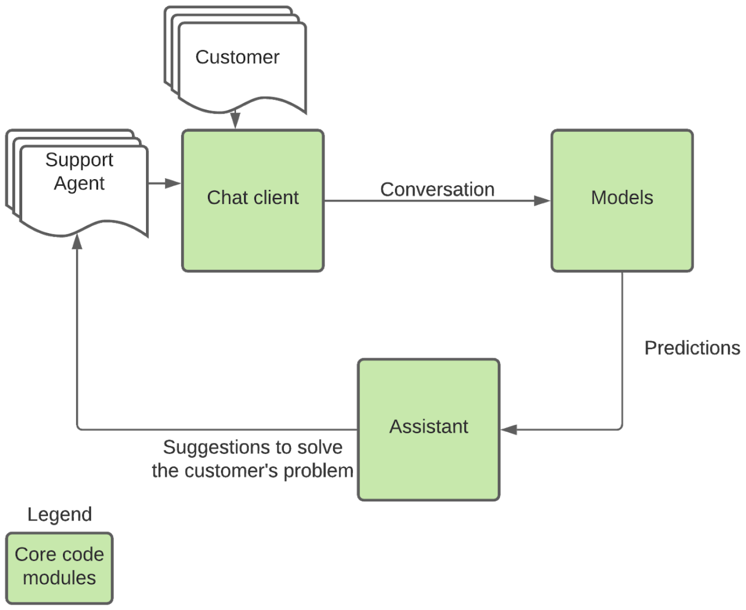
One of the most important aspects of our project is how we will visualize and display the results of the machine learning models we created. After much research we decided to set up a React app with a Flask and MongoDB backend using Docker. This implementation prioritizes reproducibility and versatility.

Before building the entire application and backend we decided to create a client and server test application in Python. This test application was used to demonstrate the classifying power of our models and also feed more conversation data into the models in real time.

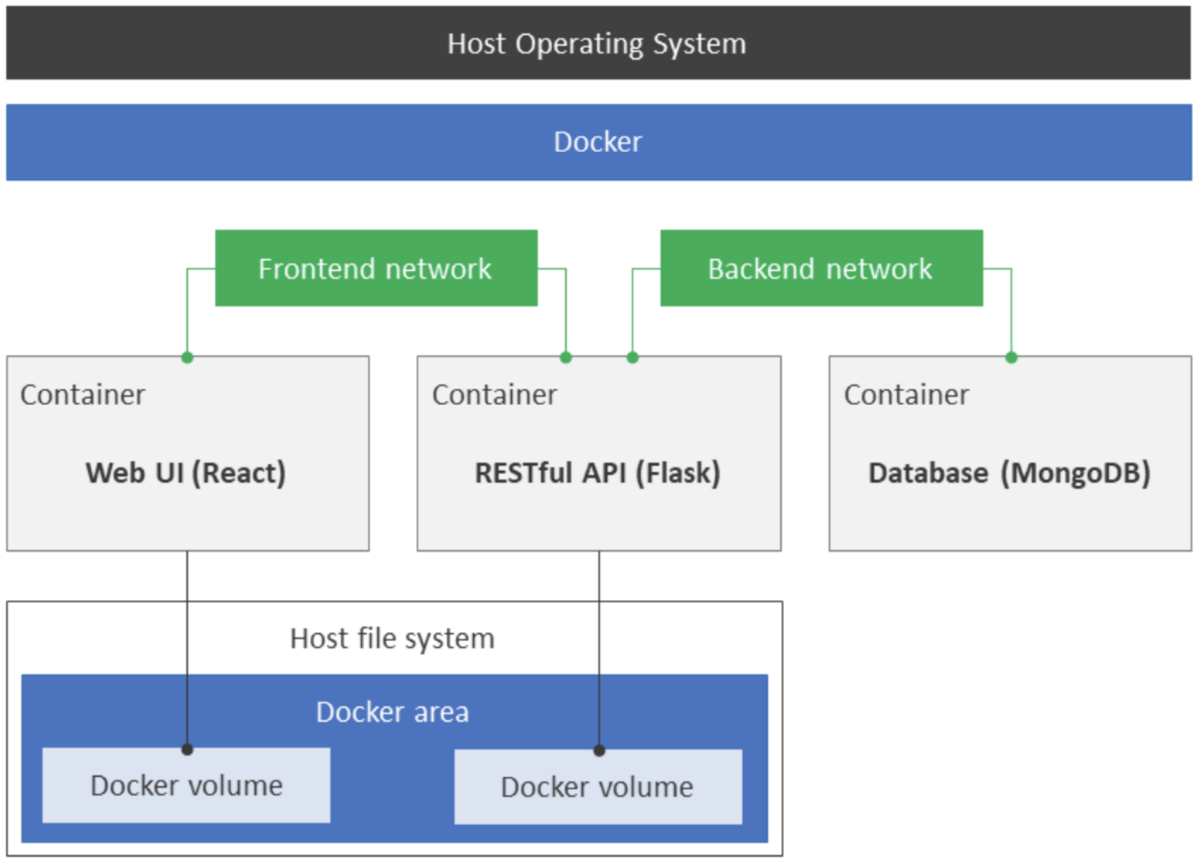
Many of the technologies we were using in our project were novel to most of us, so there was a fair amount of research done to learn more about the tech stack. React is a front-end javascript library for building user interfaces. Flask is a micro web framework written in python that will be used to connect to our database using MongoEngine and host our machine learning models. MongoDB is a document-oriented database program that uses JSON-like documents with optional schemas.

The frontend network will use react for the user interface and get the predictions for machine learning models from the RESTful Flask API. The backend network will use MongoDB to store the chat logs which will feed into the machine learning models in Flask. The reason for using docker is it creates a self-contained development environment that adapts to the users system specifications. The isolation and the portability allows our team members to quickly start working on our app and share the containers using docker hub.

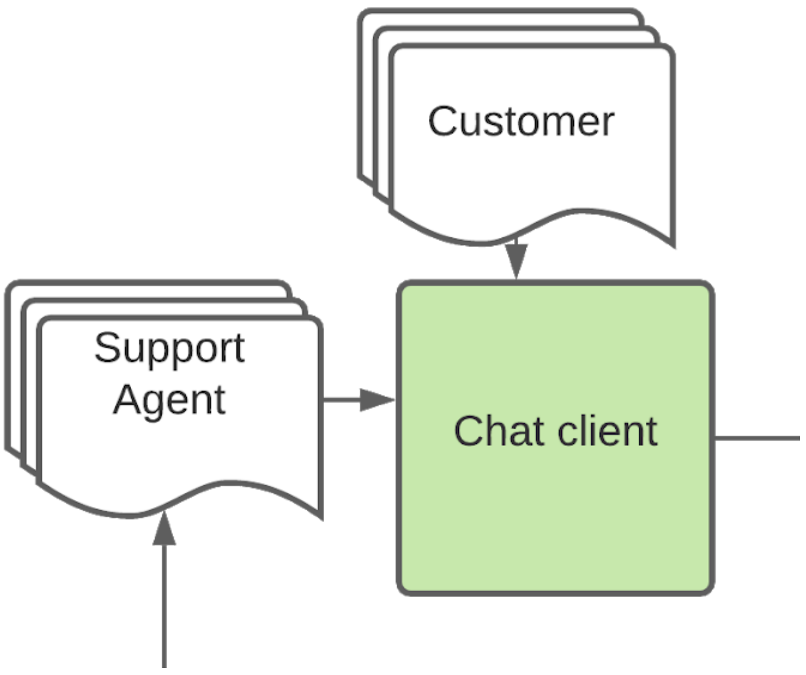
In the visualization below you will be able to see the scope of the entire product and the different components and technologies involved.



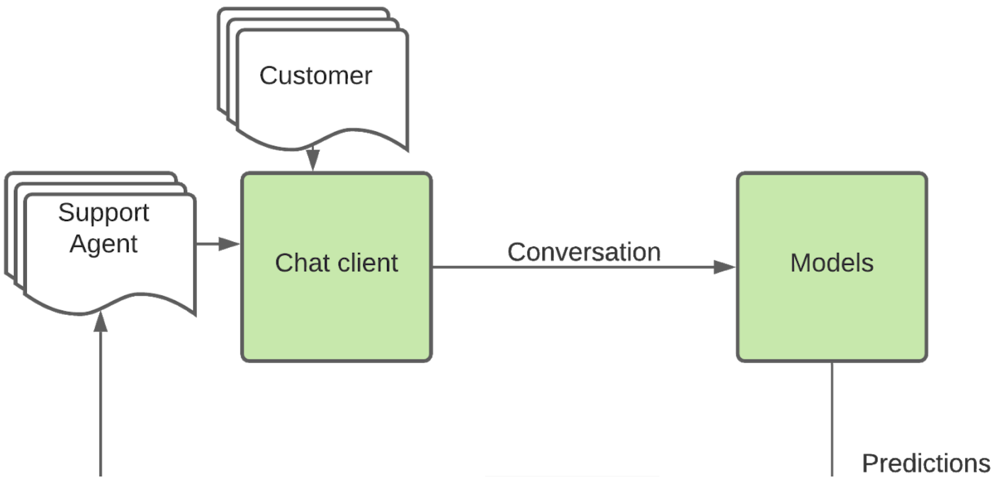
First off, as you already know our team has a very diverse background and to make it easier for all team members to work in the same environment we decided to employ docker. Docker alleviates many of the problems that arise from working on different systems each with their own unique system specifications and it also creates a self-contained development environment that adapts to any user's system specifications. The isolation and the portability allows our team members to quickly start working on our app and share the containers using docker hub. Additionally, this would greatly speed up development and allow for more detailed testing and debugging.



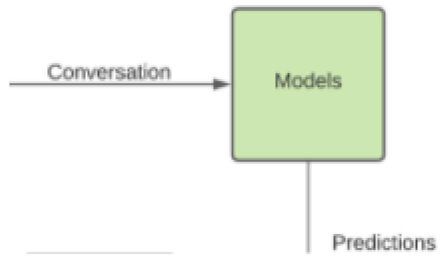
In the visualization below you can see the customer and support agent both interacting with a chat client that was created using react on the front end, with NodeJS + Socket.io web socket library on the back end. Using these frameworks I was able to create a real time chat application that Invoca's employees can directly interact with. The idea is that the customer support agent and customer will both be using this application to communicate doubts and concerns in real time.



After the conversation has terminated this chat log will be forwarded to our predictive models as you can see in the visualization. For these chat logs to be transferred they have to be stored in a safe and secure database.The database we will be utilizing is MongoDB which is an extremely fast NoSQL database meaning it uses JSON-like documents. The chat log from this database will be fed into our prediction loop where we take the raw text as input and output our predictions.



To deploy our machine learning models we will be utilizing Flask which is a micro web framework written in python that will be used to connect to our database using MongoEngine. Flask will play the critical role of interacting with both the frontend network to relay the predictions of our machine learning models to React and the backend to route the conversations from the database into our models. We are using mongoengine to connect flask to mongodb and query the conversations needed for our machine learning models. Instead of outputting a simple text prediction we wanted to create mock html loading pages to reflect the prediction. These html pages will have a visual and helpful links to help the customer service agent assist their customer better.



Conclusion

Some considerations for additional features or possible aspects to focus on for future work include fine-tuning the models or having bar graphs with live updates on how certain the model is of each classification category. We think it'd be a really cool graphic to be able to witness how the model gains confidence as the customer mentions more and more specific information, like the word "coffee," for example. If we were perhaps given 3 quarters to work on this project, we'd like to expand our idea to a fully automated chatbot that can communicate with the customer and return important links so that we don't even need an Agent. Another potential consideration if we were to continue work on our project would be to possibly integrate Yelp API and data to be able to return more accurate and relevant links or locations for assisting clients with requests related to food or general business recommendations. Integrating popular rideshare services such as Uber or Lyft would also be a way to bring our product to the next level.

For quality of life improvements, we considered adding an auto fill in function for our order templates based on information from a prior conversation between our client and assistant. Although there are definitely many interesting and useful ideas that we could've worked into our product, we ultimately focused our efforts into finishing and polishing up our MVP.

Bibliography

Google-Research-Datasets. “Google-Research-Datasets/Taskmaster.” *GitHub*, github.com/google-research-datasets/Taskmaster/blob/master/TM-1-2019/README.md.

“Most Common Words in English.” *Wikipedia*, Wikimedia Foundation, 11 May 2021, en.wikipedia.org/wiki/Most_common_words_in_English.

Rajapakse, Thilina. *Simple Transformers*, simpletransformers.ai/.

“Scikit-Learn.” *Scikit*, scikit-learn.org/stable/.

Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova Kristina,“BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”