ScottyBot: A Modular Embodied Dialogue Agent

Team ScottyBot
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
`scottybot@cs.cmu.edu`

Abstract

Embodied Artificial Intelligence is an emerging field that focuses on creating intelligent agents that can perceive, navigate, and manipulate objects in their environment. While current smart assistants are limited to speech and text-based interactions, developing embodied agents that can engage in natural dialogue and complete physical tasks presents a significant challenge—but one necessary to the advancement of AI. We tackle this challenge within the Amazon Arena Environment: building an agent that grounds dialogue context to both images and actions. `ScottyBot` is capable of holding conversations to complete missions and deliver on users' requests. Our modular, full-stack approach requires the integration of state-of-the-art natural language and computer vision models to track/execute interactions and to maintain natural conversations.

1 Introduction

Building the next generation of smart robotic assistants that seamlessly co-exist and operate in human spaces requires those agents to have situational awareness of their complex multimodal environments [Francis et al., 2022]. As part of the *Alexa Prize Simbot Challenge*, our team seeks to pragmatically model the dialogue conversation between a human (*commander*) and the agent (*follower*), into a sequence of granular instructions that can be executed reliably in the virtual environment.

Prior work on datasets, including ALFRED [Shridhar et al., 2020], TEACh [Padmakumar et al., 2021], and CVDN [Thomason et al., 2019], assume a turn-based dialogue paradigm, where the utterances are well-formed and grammatically-correct natural language instructions that only reference available objects and admissible actions in the environment. This closed-world assumption is also evident in the availability of only static environment maps, which only allow minimal (if any) movement of objects. The utilization of static maps dramatically simplifies the language understanding task, as language does not need to map to state (dialogue or procedural) but only to a location or object.

In contrast, the *Alexa Prize Simbot Challenge* takes several important steps towards realism. First is the reliance on Automatic Speech Recognition (ASR) to provide instructions in real-time: these real utterances are ambiguous—often incorrect, incomplete, or hazardous/out-of-domain. Further, the dynamic environment yields episodic changes in the locations and variations of receptacles and small objects. This necessitates generalization capabilities as well as a pragmatic understanding of the commander's instructions, grounded in visual cues.

In this report, we present a modular approach to the *Alexa Prize Simbot Challenge*, tackling language and visual navigation as co-dependent modules. The *Language Modeling and Parsing* module involves collecting data from the aforementioned simulation environment, to: construct an initial knowledge base of objects and their affordances, build pragmatic models to generate granular instructions given the free-form dialogue, map instructions to low-level action sequences that can be executed on the simulator, track the belief state of the agent, and engage with the human through dialogue for clarification in periods of task uncertainty. The *Vision and Navigation* module focuses

primarily on scene understanding and action-planning, in order to: perform actions that satisfy a user's instruction, leverage image segmentation to identify objects of interest, and to generate a state representation for action-planning. Furthermore, the *Vision and Navigation* module generates a logical sequence of actions to execute in the Amazon Arena Simulation Engine and, additionally, handles environment mapping, game session handling, and dialogue-prompt generation to alert the user about bot failures and to help them understand how to interact with objects where it is not immediately apparent.

2   Related Work

Vision and Language-based navigation and task-completion involve an agent performing a set of tasks based on language input with visual observations grounded in natural language. Earliest iterations of the capabilities of such autonomous agents were detailed in Turnell et al. [2001]. Recent work in this space is based on datasets that do not involve human-like conversation, rather a set of instructions based on which the agent must complete a task [Shridhar et al., 2020]. The TEACh dataset [Padmakumar et al., 2021] includes conversations between two agents in a simulated environment.

Historically, navigation and task completion problems have been solved using Heuristic Action Selection, or more recently through Reinforcement Learning (RL) solutions. While end-to-end Deep RL solutions have performed well on navigation tasks [Mirowski et al., 2018], they are heavily reliant on structured state representation; introducing dialogue, instead, yields a variable state space.

Recent research in the domain [Min et al., 2021] inspires a modular approach to this problem, with proven improvements over other end-to-end solutions [Pashevich et al., 2021]. This approach consists of several modular components: (1) Language Processing (LP), (2) Semantic Mapping, (3) Semantic Search Policy, and (4) a purely-deterministic navigation/interaction policy module. The LP component transforms high-level instructions into a structured sequence of sub-tasks with the help of BERT [Devlin et al., 2018]. This trainable language model adds to the framework's language understanding. The framework was able to achieve SOTA performance on the ALFRED benchmark. Due to the lack of human-like conversation in the dataset, querying, responding, and clarifying through dialogue generation is not necessary.

Another work, LEBP [Liu et al., 2022], has a sequence-generation module similar to ours. The authors employ BART [Yuan et al., 2021] to generate sub-tasks from user instructions and let users confirm the sub-tasks before execution in the environment. However, our system leverages additional fine-tuned language understanding modules, e.g., Name Entity Resolution, to understand specific action, objects, and their attributes in user instruction. Also, instead of letting users confirm the sub-tasks, our agent will ask fluent clarification questions and understand users' responses when necessary.

3   System Design

In this section, we discuss the design of the integrated system that seamlessly switches between the two separate modules (inspired by FILM [Min et al., 2021]), one language-focused and the other vision-focused, to handle all user inputs and actions. The system design includes exposed APIs and function-calls that enable smooth communication between the two modules and, by extension, ensure that the output is consistent and accurate.

3.1   Language Modeling and Parsing Module

A key function of our system is in modeling the interactions between two intelligent agents, the *commander* and the *follower*, who are to collaborate to perform tasks in simulated environments. While this function is not directly applicable to the user, the development of this function is crucial to the user-follower scenario. The follower agent will also be able to identify and deny adversarial commands which conflict with its well-being and with its ability to perform further work with proper justifications. In our user-follower case, the role of the commander is performed by a human, and our agent would be able to carry out instructions given by the human and communicate with them in real-time. The language module ensures this high-level flow:

2

1. Use Natural Language Understanding and Dialogue Management components to understand a user's high-level, free-form language and track the state of various objects and unfulfilled past instructions.

2. Pass down low-level instructions to the Navigation and Vision Module 3.2 responsible for executing path-planning and execution of actions within an environment.

3. Receive success or error codes from the Navigation and Vision Module and appropriately use the Language Generation components to either clarify user instructions or inform the user of inability to perform actions.

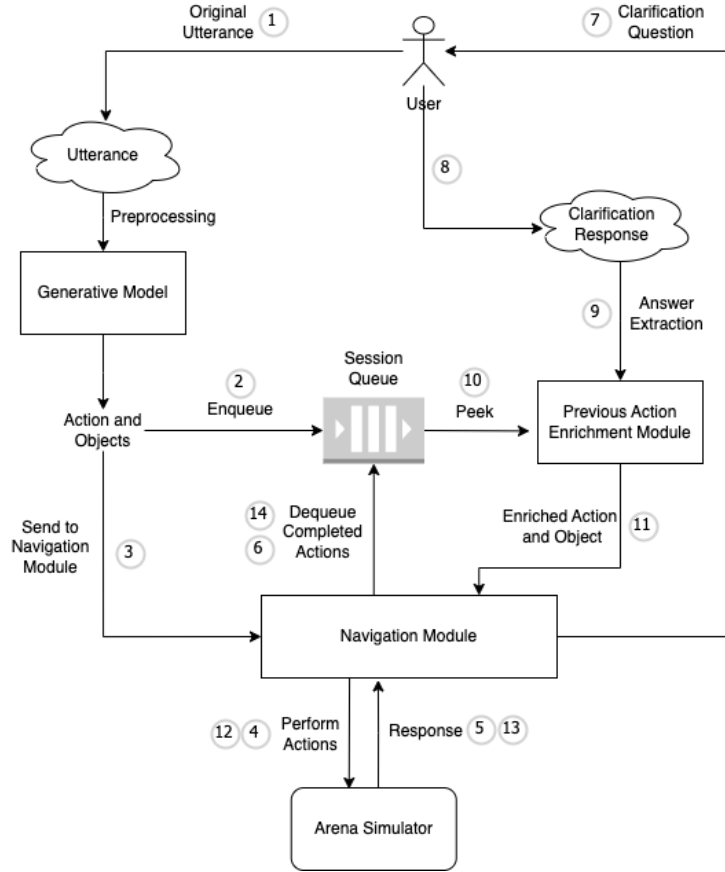



Figure 1: An overview of the state tracking and dialogue management system

3.1.1 Toxicity Check

As soon as an utterance is obtained from the *Alexa Runtime Service*, we evaluate it for profanity and toxicity. If the toxicity content is high, we immediately discard the utterance, and respond to the user in a failure mode, using a variation of *"Sorry, but I cannot help you with that"*. Whereas we first tested the off-the-shelf toxicity detection models on our in-domain utterances, we found that they could not be directly applied to our task: the occurrence of objects such as knife will trigger the models and lead to false-positive predictions, even though the utterance itself may not be toxic contextually (e.g., `Pick up the knife.`) Therefore, we instead fine-tune the toxicity detection model, with a combination of out-of-domain toxicity detection data and a subset of the in-domain utterances, which are labeled as negative. The off the shelf [1] toxicity detection model is trained on the 2nd Jigsaw challenge data and uses ALBERT [Lan et al., 2019] as model backbone. We set the trigger threshold as 0.2, which we empirically find to be able to effectively distinguish positive from negative cases.

---

[1] `https://github.com/unitaryai/detoxify`

3

3.1.2 Co-reference Resolution

If an utterance passes the toxicity check, we consult DynamoDB to fetch past dialogue utterances from the current user to get the dialogue context. We use a SpanBERT [Joshi et al., 2020] model to replace occurrences of object references like 'it', 'there', 'that' and so on with a corresponding object from previous utterances.

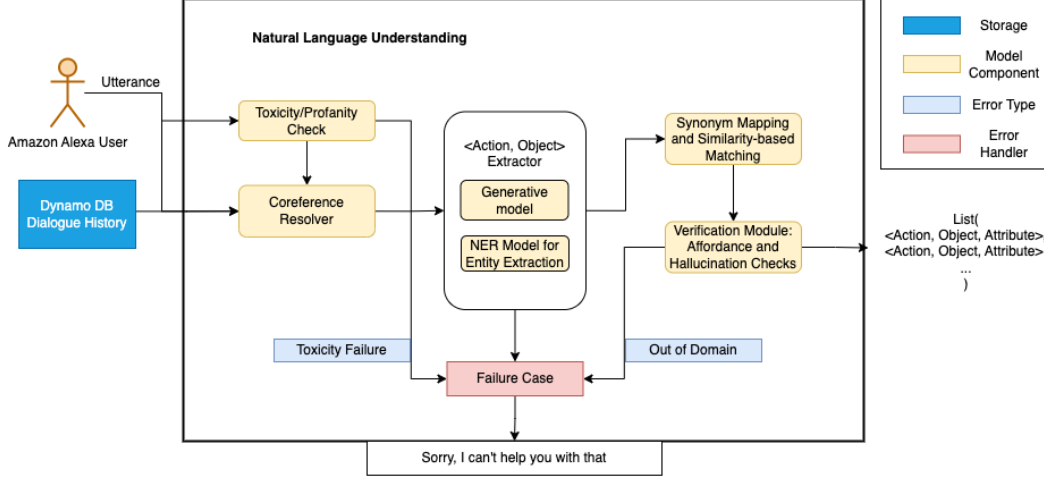



Figure 2: An architectural illustration of the *Language Modeling and Parsing* module.

3.1.3 Semantic Role Labeling with Synthetic Template Match

Semantic Role Labeling [Màrquez et al., 2008] is the task of determining the latent predicate argument structure of a sentence. For the first version of the NLU model, we utilized a pre-trained deep semantic role labeling (SRL) model proposed by Shi and Lin [2019] (checkpoint was provided by AllenNLP). Given an utterance, the SRL model predicts the verb (action) along with the context, which is used to obtain attributes and objects. For example, the sentence "`Plug the control panel onto the laser cannon`" would be parsed as "*[Action: Plug] ARG1:[the control panel] ARG2:[onto the laser cannon]*" by the SRL model. Further, a shallow template-matching strategy was used to map an action with a fixed number of arguments to a sequence of low level actions. For instance, "*Plug*" with 2 arguments would be mapped to the action sequence "*goto X, pickup X, goto Y, place Y*", where "*X*" and "*Y*" are placeholders that would be replaced by control panel and laser cannon in this example, respectively. This strategy allowed us to build a good baseline model, as we found that the deep SRL model fails to correctly identify actions and context in all scenarios. For instance, for the instruction "`Press the red button`" the deep SRL model was not able to identify "`Press`" as the action. Secondly, utilizing a shallow template-match meant that we had to hardcode templates for each possible configuration of the verb argument pair which is tedious and non-generalizable. Informed by these limitations, we chose to discard the SRL and template-match approach, in favor of a more generalized Generative Action Sequence and Named Entity Recognition approach.

3.1.4 Generative Model

We formulate the task of sub-goal prediction as a Sequence-to-Sequence (seq2seq) task: from one utterance of the user, we rewrite the natural language utterance into an ordered list of short commands of canonical form. Each generated command is composed of a predicate, an object, and optionally a list of attributes of the object. For example, "`Please grab the red apple from the fridge.`" will be mapped to "`Go to fridge. Open fridge. Pick up red apple`". The responsibilities of this component are two-fold: 1. It learns to understand and flatten compositional predicates and structures to a sequence of simple actions; this mapping is engineered to be deterministic. 2. It *potentially* learns the semantics of the task environment of interest.

To this end, we leverage a BART model [Lewis et al., 2019] checkpoint that is pre-trained on the CNN-DailyMail summarization dataset [Hermann et al., 2015] and fine-tune it with synthetic data

4

that is generated from pre-defined templates. In some sense, the purpose of this component is to also summarize and re-structure the utterance.

It is noteworthy that, apart from the general instruction-rewriting, we also endow the model with the ability to detect utterances that are greetings, the utterances that are meta-commands, and utterances that are out-of-domain. For the third case, we add utterances from other task-oriented dialogue datasets into training data, and, for the meta-commands, we include utterances such as `"Forget about it."` and `"Don't do that."` labeled as the meta-command category `"cancel"`. We also include utterances such as `"Hi there."` as greetings.

3.1.5 Named Entity Recognition Model

The Generative model outputs a sequence of low-level actions. These actions comprise of a set of tokens which need to be classified into their corresponding type such as "action", "object", "attribute", etc.; Named Entity Recognition (NER) [Sundheim, 1995] is a task of tagging entities with their corresponding type. In order to perform NER, we first generate synthetic data annotated with 11 different entities such as action, object, attribute, location, magnitude, direction, etc. in the CONLL format [Sang and De Meulder, 2003]. We then fine-tune a pre-trained DistilBERT-uncased model [Sanh et al., 2019] on token classification task on the synthetic dataset. After brief training, the model learns to classify tokens into their corresponding categories. We tested the model on several out-of-domain examples, including synonyms of objects and objects with different colors; we observed the model to be robust to these instances of data domain distribution shift. This indicated that the model has developed a semantic understanding of tokens rather than merely overfitting to the synthetic data.

3.1.6 Extractive Question Answering

Extractive Question Answering (EQA) is the task of answering a question based on a given context. We use a pre-trained BERT-based EQA model [Liu et al., 2019] from HuggingFace, trained on SQuAD 2.0 [Rajpurkar et al., 2018], `roberta-base-squad2`. This model is used for two purposes. First, to extract the object attributes for a given object; the model takes, as input, the utterance and an object, then outputs the attributes corresponding to the object from the utterance. To further improve the model performance, the extracted attributes are post-processed where we remove the articles, punctuation, and apply part-of-speech tagging to reject the irrelevant attributes (e.g., nouns, verbs, etc.) The other purpose of the model is to extract the answer to the clarification questions from the user's utterance (i.e., the clarification utterance).

3.1.7 Multi-Genre Natural Language Inference

The actions and objects tagged by the NER model may not always be the same as that which is stored in our static knowledge base. In order to map the actions/objects, we find the semantically most similar object and actions from the knowledge base: given an utterance with actions and objects tagged, we generate sentences by sequentially replacing actions and objects with ones in the knowledge base and we find the ones that have highest match scores. We use a synonym library for each object instance, and potential attributes associated with it. We use high precision matching using regular expressions, using full word/phonetic match, leveraging edit/Levenshtein distance as the metric. We update the word based on tuned threshold values.

We observe that this task closely resembles Natural Language Inference (NLI) tasks in NLP. NLI is the task of deciding, given two text fragments, whether the meaning of one text is entailed (can be inferred) from the other. For this, we use BERT-based model DeBERTa [He et al., 2020], from HuggingFace, trained on Multi-Genre Natural Language Inference (MNLI) [Williams et al., 2017] for the task of textual entailment. The MNLI corpus is a crowd-sourced collection of 433k sentence pairs annotated with textual entailment information; we further fine-tune this model with synthetically generated data from the knowledge base.

3.1.8 Verification Module

The verification module accounts for two things: affordances and hallucinations from the generative model. Object affordances signify what actions are executable on a particular object, e.g., a heavy laser machine cannot be picked up. We can therefore discard actions such as *"pick up the laser machine"*, based on our prior knowledge of object affordances. Generative models can often hallucinate.

5

For example, "`Pick it up and place it on the table`" may be converted into *"Pick up the banana. Place on the table. Done."* While our in-domain training on synthetic data minimizes hallucinations, we anyway verify whether the object identified occurred in the dialogue history for the current session, for the tail-end of cases where our model might hallucinate.

3.1.9 Context Tracking and Dialogue Management

State and context tracking is an important component of a dialogue management system. The dialogue management system is responsible for maintaining context over multiple user utterances, to understand the user intent fully before performing a task without any ambiguity. Figure 1 shows an overview of our system. The numbers next to the arrows in the figure represent the order of execution.

As soon as a user utterance comes in, we process it using the models as mentioned below to obtain canonical actions and associated objects (if any). We enqueue these in a queue that we maintain for every game session and then send them to the *Vision and Navigation* module. This module performs the actions which can be unambiguously carried out in the simulator and, based on the simulator response and its own vision information, informs us of the actions completed successfully. We dequeue these actions from the session queue first and then generate a clarification question for the user (if needed), based on the data we receive from the *Vision and Navigation* module.

The generated clarification question is asked to the user. Using the extractive question answering (EQA) model discussed in Section 3.1.6, the user response is processed to extract the exact answer from the utterance. We then peek at our session queue to get the last unexecuted action, enrich it with the extracted clarification response we got from the user, then finally send this enriched canonical action to the *Vision and Navigation* module. This module performs the action in Amazon Arena, dequeues the completed actions from the session queue, and generates more clarification questions for the user to answer (if needed). If the queue becomes empty, it means that we have processed the instructions provided by the user till now; we then await further instructions to complete the task.

3.2 Vision and Navigation Module

This section outlines the system design and the architecture of individual components that make up the *Vision and Navigation* pipeline of `ScottyBot`, as illustrated in Figure 3.

3.2.1 Action Processor

Raw action tuples obtained from the dialogue module serve as input to the *Vision and Navigation* module's action processor. The Action Processor is the brain of the *Vision and Navigation* module: it coordinates with the rest of the modules to decide the next sequence of actions that the bot should perform, in response to user utterances.

- **Action Queue.** The Action Queue module manages the sequence of primitive actions that the bot needs to perform in order to complete a task. The module implements a standard queue, with some exceptions that are described later in this section. The module has two main functions: `enqueue` and `pop`. The `enqueue` function first translates an action into a format that the navigation and interaction modules are designed to expect. These translated actions are then pushed into the queue, which is internally stored as a list of dictionaries representing each action. The `pop` function involves additional logic over a simple queue. Actions can either be executed individually or as a batch of instructions, depending on whether they require intermediate responses from the simulator. Actions that require simulator inputs must be executed individually, while those that do not can be grouped into a batch. The `pop` function yields this batch of actions.

- **Path-planning.** The Path-planning module is a critical component of the navigation module, responsible for generating a sequence of actions necessary to explore a room and locate an object. It operates differently for receptacle and non-receptacle objects. For receptacle objects, the room to be searched is obtained from the environment mapping (see Section 3.2.3), as their location is static across missions. For smaller objects, the bot́s current room is searched. The module populates a path planning queue with a series of actions that navigate the bot to each of the 8 viewpoints in the room. Additionally, an action is added at each viewpoint for the bot to *look around*, providing a complete 360° view of the current location.

6

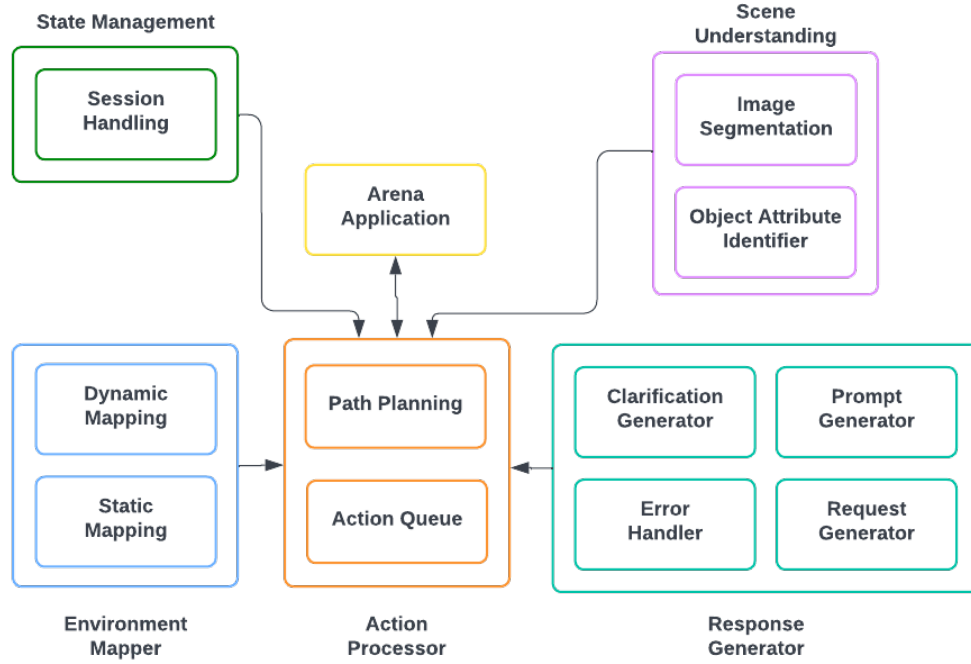



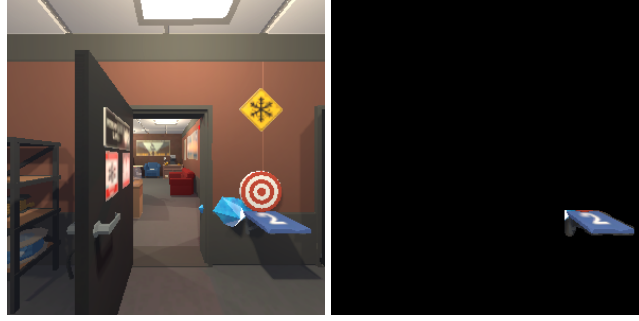
Figure 3: An architectural illustration of the *Vision and Navigation* module.

3.2.2 Scene Understanding

A crucial step in executing instructions is to localize objects in the scene and disambiguate them based on attributes. The dialogue module processes the object in consideration, and also provides attributes associated to the object, that are relevant to the task, for example, "*Go to the Gravity Flipper computer*" would translate to `{"action":  {"GoTo", "Computer"}, "attributes": ["green"]}`. The scene understanding module is broken down into the following sub-modules:

- **Image Segmentation.** The Image Segmentation module is responsible for processing sets of color images with attributes and classes, resulting in a collection of masks for identified instances. This is achieved through the use of the MaskRCNN-based instance segmentation model, provided by Amazon to *Alexa Prize Simbot Challenge* participants, which employs a configurable probability threshold parameter to determine the selection or rejection of masks. The module outputs scores, labels, and predicted masks for each selected instance. The primary color information is processed using the Object Attribute Identifier module, as explained below. If further disambiguation is required for task completion, the Clarification Generator module (Section 3.2.4) is triggered.

- **Object Attribute Identifier.** We observed that most missions just require disambiguation using color. Wall Shelf (Red, Blue), buttons (Red, Green, Blue), computers (Red, Green, Blue, White, Black, Pink) are some examples of objects distinguished through color. We superimpose the binary mask obtained from MaskRCNN over the original image, to obtain only the selected portion of the image. These pixels are passed through a K-Means clustering algorithm to obtain the dominant color as the cluster center. A similarity metric was used to obtain the closest color based on a set of fixed RGB values. This method outperformed other naive attempts, including using statistical methods like mean and median over pixel values, with a near 100% accuracy in color prediction.

7

(a) Example image obtained from Arena Simulator

(b) The mask generated for a *Blue Shelf*

Figure 4: Image segmentation: input (left) and output (right).

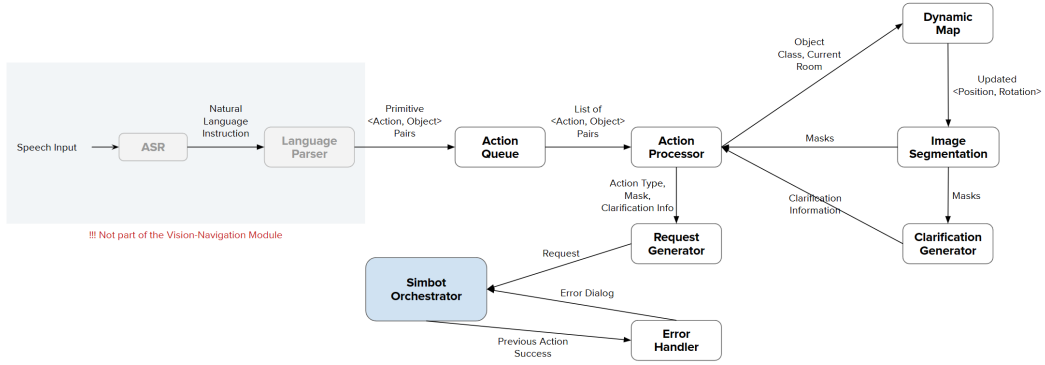



Figure 5: Flow of data through the *Vision and Navigation* Module.

3.2.3 Environment Mapper

The Amazon Arena environment's configuration varies between missions, and the locations of objects within the environment are not constant. The Navigation and Interaction pipeline includes a mapping module that tracks the locations of encountered objects as the bot moves through the environment. To solve this, we include two separate mappings:

Static Mapping   Static maps construct mappings between large receptacle objects whose locations are fixed across missions and layout configurations. We parsed all the game description files to obtain the location mappings of receptacle objects.

Dynamic Mapping   As Arena missions ensure different locations for smaller objects, we create a dynamic mapping at the start of each game on the fly as the bot navigates through rooms. We use the dynamic mapping module to add, update and retrieve object instances from the session-specific map.

The Dynamic Map is a dictionary that stores the mapping of rooms, object classes, attributes, positions, rotations, depths, and hashes of object instances. We build the module to provide the following functionalities:

8

The dynamic map is stored in the following format.

```
dynamic_map = {
    "current_room": {
        "object_class": {
            "attribute_1": {
                "hash": [hash_value_1, hash_value_2, hash_value_3],
                "position": [ {}, {}, {} ],
                "rotation": [ {}, {}, {} ],
                "depth": [ d1, d2, d3 ]
            }
        }
    }
}
```

- `addToDynamicMap` adds an object instance to the dynamic map. It checks if the current room, object class, and attribute exist in the map, and if not, creates the necessary entries.

- `removeFromDynamicMap` removes an object instance when it is picked up by the bot.

- We calculate the best object position based on either the depths or Euclidean distances between positions and the current bot position.

- While looking for an object in the current room, we first perform a lookup in the dynamic map for the object along with the correct attributes. We navigate the robot to the position and rotation stored in the map if found, before performing the actual action requested.

- We experimented using the instance segmentation and depth maps to get the accurate position of an object instance. We convert the position and rotation of the bot, position and depth on an image of the object in a virtual camera space into a 3D point in world coordinates. However, we were not able to accurately obtain the focus of the camera used on the bot, and thus disregarded this method.

3.2.4 Response Generator

A crucial aspect of the navigation pipeline is the interaction with the simulator, and more importantly, the user. The components of the response generator are responsible for constructing structured dialog/action sequences as per requirement. We consciously generate responses through the vision and navigation module, instead of the language model, to leverage visual context and action sequence history, to provide meaningful replies and clarification requests.

- **Request Generator.** The dialogue module elaborated in 3.1 converts natural language utterances from the user into a set of executable action pairs. These actions are converted into the JSON format specified by the Arena Simulator, along with relevant information from the Scene Understanding module 3.2.2. Each instruction is classified into Goto, Interaction and Navigation actions, and corresponding templates are used.

- **Error Handler.** The Error Handler processes status codes from the Arena Simulation Environment for each bot action and generates dialog responses to inform the user of errors and provide instructions to resolve them. Errors from Arena are handled through a careful examination of all errors obtained from the simulator. Errors obtained from the path-planning module 3.2.1 are addressed through messages like *"Sorry, I could not finish executing my tasks, can you give me a simpler task?"* or *"I could not locate the object, are you sure it́s in the current room?"*. The user is then prompted to identify the problem and issue a simpler instruction.

- **Prompt Generator.** The Amazon Arena Environment contains non-real-world objects, such as freeze rays, time machines, and color-changer machines, which may be unfamiliar to users and difficult to operate. The Prompt Generator module addresses this issue by generating prompts for interacting with nearby objects when it is not straightforward to do so. For example, when the bot is near the Laser Machine, the module prompts users to *navigate to the red monitor and turn it on to fire the laser*.

9

- **Clarification Generator.** An important aspect of the *Alexa Prize Simbot Challenge* is dialog interaction and communication. The primary use-case of the Clarification Generator module is to disambiguate between multiple instances of an object, based on the different attributes associated. For example, in the situation where the bot faces two computers in the Robotics Lab, and is asked to go to one, the clarification module is triggered, and the user is asked if they meant the *blue* or *red* computer.

Figure 6: Scenario for Clarification

3.2.5 Maintaining Object History

The `HistoryHelper` class assists in placing objects back in their designated locations. It tracks the history of interactions and finds the closest receptacle for a given object.

1. **Finding the Closest Receptacle**

    - Identifies the closest receptacle for a given object.
    - Calculates the distance between the object and potential receptacles based on position and depth.
    - Selects the receptacle with the shortest distance metric.
    - Updates the latest interaction with the identified receptacle's class and mask.

2. **Placing Objects on the Closest Receptacle**

    - Determines the actions needed to place an object on the closest receptacle.
    - Checks if the latest interaction matches the provided object and room.
    - Generates a sequence of actions to reach the closest receptacle based on the stored interaction information.
    - Returns the generated actions, receptacle class, and associated attributes.

3.2.6 Session Handler

Alexa Prize Simbot Challenge missions allow for multiple users to participate simultaneously, necessitating the implementation of session handling. The system uses DynamoDB to store and retrieve state information associated with independent sessions. Each session is identified by a session identifier, which serves as the key for the corresponding state information in DynamoDB. At the beginning of each turn, the system queries DynamoDB for the state information related to the current session identifier. If the key exists, the state information is initialized for the turn; otherwise, a new empty session state is created. The turn comprises 10 calls to a function that generates actions for the simulator. Upon completion of the turn, the state information is written back to DynamoDB, avoiding the need for repeated read and write operations during the turn.

10

4 Data and Experiments

4.1 Dataset Description

We leverage the Amazon Arena platform and dataset provided by Amazon Alexa [Gao et al., 2023].
The challenge comprises of several missions that need to be performed in the simulated environment.
The Arena simulator contains multi-room layouts and over 200+ interactable objects. The users are
also shown a mini map of the floor plan of a specific layout on the top right corner of the screen. The
allowable actions in Arena can be broadly classified into 2 categories namely Navigation (e.g., go
to the room) and Manipulation (e.g. toggle the light switch). Each mission in Arena comprises of
a series of tasks. These tasks must be first decomposed into a series of sub-tasks and further each
sub-task must be mapped to a series of primitive actions that are compatible with the simulator.

The dataset comprises of three main components:

1. *Vision data*—contains RGB images, ground truth segmentation masks, and metadata about
   object annotations and commands executed prior to capture. It includes about 450k annotated
   segmentation images and a total of 86 object classes, and is used for training instance
   segmentation models like MaskRCNN and MaskFormer.

2. *Trajectory data*—contains ground-truth action trajectories for completing 3.5k+ game
   missions, paired with robot-view images, and annotated with human language instructions
   and synthetic language instructions.

3. *Challenge Definition Format (CDF) data*—stores game mission information including initial
   and goal states, game-related text data, and is used to obtain information about the possible
   locations of receptacles across missions.

4.2 Dataset Challenges

There are nearly 10k game sessions of human-annotated instructions, 55k human-annotated question-
answer pairs within 20k game sessions with QAs. Each game mission is associated with template-
based synthetic language instructions. A key issue with the action trajectory data that Amazon
provided is that it is real data in a particular room environment or layout. However, certain action
sequences can change significantly based on the room configuration, and visual grounding of the
robot. To help reiterate this issue, here is a test scenario designed by the team: `"ScottyBot
Team: Pick up the apple"`. This instruction is mapped to `"Go to the table. Pick up
the apple."` in an environment where the apple was present on a table within the field of view of
the robot, whereas, it is mapped to `"Go to the fridge. Open the fridge. Pick up the
apple. Close the fridge."` in another room layout where the apple was present inside the
fridge. While specifically these breakdowns are correct, they would not hold in any other situation.

To ensure decoupled instructions, we synthetically generate data for training our canonical action
sequence generative model and NER model. Our synthetic dataset contains 116 templates for in-
domain instructions. These templates contain placeholders like `<action>`, `<object:affordance>`,
`<color>`, `<shape>` etc. that were used to create **57,460 high-level utterances** and their corresponding
low-level instructions by randomly replacing objects, actions, and attributes like color or shape with
relevant entities from our static knowledge base of all possible objects and admissible actions in the
Arena environment.

Some examples of such a template is presented in Table 1. As presented in the first template,
`<object:1:BREAKABLE>` is replaced by an object that can be broken in the simulation environment,
and `<action:1:>` is replaced by an action that is synonymous to breaking (based on action 1 being
mapped to object 1 and object 1 affordance being BREAKABLE).

We also use this synthetic data generation scheme for training the NER model to identify objects,
actions, locations, magnitudes, and directions.

11

Input Template	Output Template	Input Sentence	Output Sentence
<action:1> the <object:1:BREAKABLE>	<action:1> the <object:1:BREAKABLE>.	shatter the papercup	instruction: break the paper-cup. done.
<action:2:place> <object:1:PICKUPABLE> in <object:2:RECEPTACLE>	Pick up <object:1:PICKUPABLE>. <action:2:place> on the <object:2:RECEPTACLE>.	drop screw driver in computer	instruction: pick up screw driver. place on the computer. done.
Find <object:1> in <location:1>	Go to the <location:1>. Go to <object:1>.	find bread in warehouse	instruction: go to the warehouse. go to bread. done.
Go to <location:1> and <action:2:go to> <object:2>	Go to the <location:1>. <action:2:go to> the <object:2>.	go to mainoffice and move closer to coffee unmaker	instruction: go to the mainoffice. go to the coffee unmaker. done.

Table 1: Examples of synthetically generated data. Please note that all example input sentences in this table were created by the ScottyBot team

4.3 Evolution of Action Sequence Generation Models

4.3.1 Iteration 1: Grammar-based model

The first iteration of the action sequence generation model was heuristic rule-based, grammar model. This model was used during the initial skill and model development phase. This model made use of simple POS-tagging of a user utterance, with word-matching of verbs to actions, and objects to nouns. While this served as a good baseline for further development, it fails on multiple scenarios.

Utterance	Predicted Action Sequence	Expected Action Sequence	Comments
Go to the laser	<Go, laser>	<Go, laser>	Works well on granular instructions
Put the control panel into the laser	<place, control panel>	<pickup, Control Panel>, <place, Laser>	Fails on incomplete instructions
Fire the laser using the laser monitor	<Empty>	<toggle, laser monitor>	Sometimes inaccurate in identifying action

Table 2: Examples of synthetically generated data. Please note that all example utterances in this table were created by the ScottyBot team

4.3.2 Iteration 2: Semantic Role Labelling

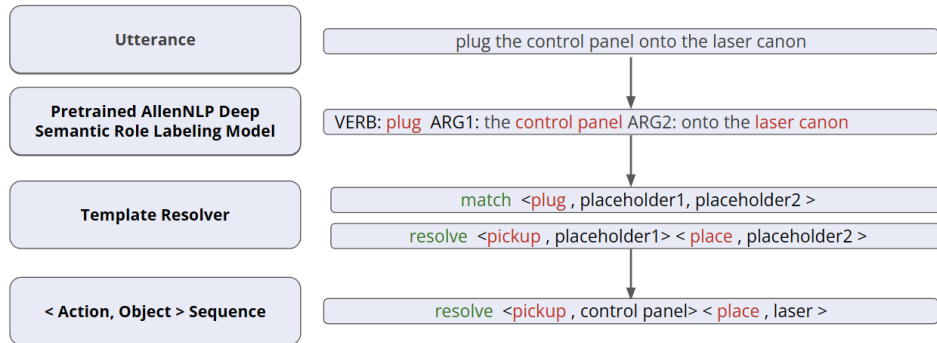



Figure 7: Iteration 2: Deep Semantic Role Labeling pipeline for action sequence generation. Utterances were created by the ScottyBot Team

The next iteration of the action generation module accounted for shortcomings of the grammar based model. Here the utterance is passed through to a pre-trained AllenNLP DeepSRL model 3.1.3. The output is passed through to the template resolver module, which outputs action-object sequences 7. The SRL model however, fails to understand nuances that are implicit to the environment. It is also

12

not able to account for templates that are unseen. Table 3 highlights some examples delineating why we moved away from this model.

Utterance	Predicted Action Sequence	Expected Action Sequence	Comments
Turn on the freeze ray using the monitor	<toggle, freezer>, <pickup, monitor>, <toggle, computer>	<toggle, freezer>, <toggle, computer>	Templates get misused. Has no implicit understanding, hence cannot add information.
Take the soda can to the shelf	<pickup, soda can>, <pickup, soda can>, <goto, shelf>	<pickup, soda can>, <goto, shelf>	Does not know how to handle unseen templates.
Press on the red button	<Empty>	<toggle, button>	Sometimes inaccurate in identifying actions.

Table 3: Examples of failures of the AllenNLP DeepSRL model. Please note that all example utterances in this table were created by the ScottyBot team

4.3.3 Iteration 3: Generative Model

As elaborated in Section 3.1.4, the generative model is trained to output low-level, granular simple instructions, which can be converted into action-sequences in the required format. Generative Model v1.x was trained over the instruction utterances provided in the Arena dataset. Section 4.2 dives deeper into the challenges of the dataset. Due to these challenges, we curated a synthetic dataset, to train Generative Model v2.x. Table 4 gives an overview of the failures and the rectifications due to synthesizing new examples.

Utterance	Prediction from Previous Model	Prediction from Current Model	Comments
Place the red apple on the desk.	Place on the desk.	Pickup red apple. Place desk.	Missing information: apple. Two objects, one action, second action needs to be inferred.
Find the blue bowl.	Go to desk. Go to bowl.	Go to blue bowl.	Spurious correlation between location and objects causes hallucination of desk.
Turn right and press the red button.	Toggle button.	Turn right. Toggle red button.	The previous model was unable to cope with navigation instructions.

Table 4: Comparison of predictions between the previous and current language models. Please note that all example input sentences in this table were created by the ScottyBot team

5   Evaluation and Results

We evaluate the effectiveness of our system through modular testing and metrics applied on the language and navigation modules separately.

5.1   Dialogue Modules

We use several different metrics in order to benchmark the individual components of the Dialogue system. To benchmark the generative model we use *Word Error Rate* and the *Exact Match* criteria. We specifically rely on these metrics as our generative model performs constrained generation and is restricted to outputting a low level sub goal sequence that can be directly run on the simulator. Thus it is important for the generative model to obey word order and accurately generate the expected output that is predefined for the template.

In order to evaluate the NER model we use *precision, recall, F1 score and accuracy* metrics in a one vs rest classification approach. To evaluate the EQA model we use F1 score and Exact Match metrics while we use *Levenshtein distance* and accuracy to evaluate phonetic and synonym matching.

13

5.1.1 Results

Co-reference Resolution. Since we used a pre-trained co-reference resolution model without fine-tuning, we performed a qualitative analysis of its performance with in-domain examples. A few scenarios and corresponding examples tested are listed in Table 5.

Phenomenon Tested	Original Utterance	Coref Resolved Utterance
Single noun phrase before the pronoun	Go to the red computer and turn **it** on.	Go to the red computer and turn **the red computer on**.
Multiple noun phrases before the pronoun	Pick up the bowl. Open the microwave and place **it** inside. Close **it**.	Pick up the bowl. Open the microwave and place **the bowl** inside. Close **the microwave**.
Pronouns other than the most common '*it*'	Find the soda can in the break room and grab **that**.	Find the soda can in the break room and grab **the soda can**.
Cataphora (use of an expression co-refers with a later, more specific, expression)	Can you pick **it** up, the cake over there?	Can you pick **the cake** up, the cake over there?
When resolution is not possible	Go to the quantum lab and please get **that** from **there**?	Go to the quantum lab and please get **that** from **the quantum lab**?

Table 5: Qualitative Evaluation of Co-reference Resolution. Please note that all example utterances in this table were created by the ScottyBot team

Semantic Role Labeling with Template Match We utilized a pre-trained model for semantic role labeling (SRL) without performing any specific fine-tuning on our data due to the difficulty in obtaining annotations for the data in the format required for SRL. Hence we present a qualitative analysis of the SRL model in the Table 6.

As can be observed the SRL approach is sub-optimal as it tends to provide us only with high-level sub-goals which must be further processed in order to obtain action-object sequences that can be executed on the simulator. Furthermore, the SRL approach is also inaccurate in certain cases as can be seen from the fourth example in Table 6. Hence we chose to discontinue this approach.

Utterance Text	Predicted Action - Object pairs	Template Matched
Go to the control panel	<go to,[control panel]>	<goto, control panel>
Place the control panel on the laser	<place,[control panel, laser]>	<pickup, control panel> ,<place, laser>
Fire the laser using the laser monitor	<fire, [laser, laser monitor]>	<toggle, laser monitor>
Press on the red button	null	null

Table 6: Qualitative Evaluation of the SRL model. Please note that all example utterances in this table were created by the ScottyBot team

Generative Action-sequence Prediction. As our data is generated using templates, we partition the templates to investigate how the model would perform on templates in the training data and templates absent from the training data. We train a model using templates from version 1 and version 2 (denote it as model A), and another one using all the templates from version 1 to version 4 (denote it as model B).

Table 7: Evaluation results of general commands. We use two metrics for evaluation: EM (exact match) and WER (word error rate). We evaluate two model variants trained on different training sets, where the second training set is a super-set of the previous one.

Model	Test Split	EM	WER
A	A	96.9	0.6
A	B	81.9	7.1
B	B	96.4	0.9

14

Results on General Commands. We first test the models on general commands, which involves requiring the robot to perform some set of operations upon certain objects (or itself) in the environment. For this experiment, we denote the test data with the templates of version 1 and 2 as test split A, and the test data with the templates of version 1 through 4 as test split B, in accordance with the naming of model variants. As shown in Table 7, from the performance of model A evaluating on test split A and model B evaluating on test split B, we can conclude that model capacity is not a bottleneck, as the model maintain good performance on in-domain test data with the addition of new templates. Also, model A performs worse on test split B than model B. This is also expected, since test split B contains unseen templates for model A.

Table 8: Evaluation results of model B on some of the typical templates. We use two metrics for evaluation: EM (exact match) and WER (word error rate).

Template Input	Template Output	EM	WER
Put the \<object1\> in the \<object2	Pick up \<object1\>. Go to the \<object2\>. Place on the \<object2	99.4	0.2
\<Pickup\> \<color\> \<object1\> out of the \<object2:Openable\>	Go to the \<object2\>. Open \<object2\>. Pick up \<color\> \<object1\>	100.0	0.0
\<Pickup\> \<color\> \<object1\> out of the \<object2:Openable\>	Go to the \<object2\>. Open \<object:2\>. Pick up \<color\> \<object1\>.	100.0	0.0

As shown in Table 8, we list some of the templates that we use for generating the data as well as the templated output and the model's performance on the data constructed from these templates. Our template covers not only varieties of predicates and objects, but also attributes. We also examine the error cases and find that some of them are caused by the inconsistent use of articles in the training data. For example, instead of generating *go to the counter*, the model generates *go to counter*. But these would not affect model's downstream behaviors, as we have the NER model in place to extract the information we need for command execution.

Table 9: Evaluation results of special commands. We use two metrics for evaluation: EM (exact match) and WER (word error rate). We evaluate model B on different special commands.

Command Type	EM	WER
Greetings	100.0	0.0
Cancel	100.0	0.0
Out-of-domain	100.0	0.0

Results on special commands Apart from the general commands, we also designate several special classes for multiple purposes. These include: (1) greetings (e.g., `"Hello!"` and `"Hi there."`) (2) the meta-command `"cancel"`, which is used in the clarification phase, in the case where the user do not wish to continue with the current command. The `"cancel"` command clears the clarification queues and state-tracking buffers. (3) out-of-domain utterances, i.e. utterances that are not general commands and also do not fall within the previous cases, they are essentially utterances that are irrelevant to task performing. As shown in Table 9, our model is able to achieve perfect performance upon these commands.

Few-shot example phrases for each of the above 3 special commands were generated using Chat-GPT with the following prompts:

1. Greetings: *Write 50 different greeting phrases a user might say to a generic chatbot*
2. Cancel: *Write 20 different phrases to ask a bot to stop doing something, like "forget it", or "wait, no not that"*
3. Out-of-domain: *Write 100 phrases of normal statements people make in daily conversation, don't put any questions*

15

We train a simple DistilBERT model with a classification head to perform four-way classification between the three special commands and the general instruction commands. In case of the three special intents, we respond from a repository of default dialog responses. The generative model inference code for action subsequence planning is only called in case of instructional language (general command templates).

Named Entity Recognition   We performed a quantitative as well as a qualitative evaluation of the NER model. In the synthetic data that was generated from templates there were about 70K annotated examples after deduplication. However, the number of examples per template were not uniform. For instance, the template "*<action:1:!place> the <color> <object:1>*" had the maximum number of examples (3800) associated with it owing to the high number of combinations possible with the given placeholders. However, a static template such as "*Look up*" only had 1 example associated with it.

To mitigate this uneven data distribution we up-sampled all the minority templates to have a minimum of 1000 examples. In order to do so with a probability p we concatenated an example from the minority template with another example from a different template randomly chosen from amongst the set of available templates using the 'AND' token and with probability $1 - p$ we retained the same example.

After up-sampling, we obtained about 140K examples. With a 90:10 train-test split we utilized around 126K examples for training and around 14k examples were held out for validation. The NER model classifies every token into one of 11 possible classes. Please note that the annotations follow the B-I-O standard for labeling. The results of quantitative evaluation of the NER model are depicted in Table 10 and the results of qualitative evaluation are depicted in Table 11.

We find that the NER model is mostly accurate on all the classes. However there are instances in which the NER model incorrectly classifies a token such as in the case of the fourth example in Table 11. However these minor errors can be easily handled through the verification logic.

Table 10: Quantitative Evaluation of NER model on held out test set

Label	Precision	Recall	F1	Acc
B-ACT	1.0	1.0	1.0	1.0
I-ACT	0.9885	0.9926	0.9905	0.9926
B-OBJ	1.0	0.9998	0.9999	0.9998
I-OBJ	1.0	1.0	1.0	1.0
B-ATTR	1.0	1.0	1.0	1.0
B-DIR	1.0	1.0	1.0	1.0
B-LOC	0.9893	1.0	0.9946	1.0
I-LOC	1.0	1.0	1.0	1.0
B-MAG	1.0	1.0	1.0	1.0
I-MAG	1.0	1.0	1.0	1.0
O	0.9979	0.9967	0.9973	0.9967

Table 11: Qualitative Evaluation of the NER model

Utterance Text	Predicted NER TAGS
go to the laser monitor	B-ACT I-ACT O B-OBJ I-OBJ
rotate left by 60 degrees	B-ACT B-DIR O B-MAG I-MAG
pick up the red box	B-ACT I-ACT B-ATTR B-OBJ
go to the office	B-ACT O O B-LOC

Clarification Question Answering   Our agent will ask clarification questions if it needs more information to continue the task. For example, when user says "go to the computer", and there are two computers in the agent's sight, the agent will ask "what is the color of the computer?" The next response of the user is supposed to be the answer to this question. Here we use a EQA model (introduced in Section 3.1.6) to understand and extract the answer from the user's response. In the previous example, the input to EQA would be {"context": "It is blue." "question": "What is the color of the computer?" }, and the model should output "blue".

16

We use templates to generate over 6800 QA pairs on the objects and properties in the game setting to evaluate the pre-trained EQA model. It has on average, an *F1 score* of **0.72** and *Exact Match Rate* of **0.62**.

Phonetic and Synonym Matching For the actions and objects in the user utterance tagged by the NER model, we find the matching actions and object in the knowledge base. To resolve the phonetic error with the ASR, we find the action/object that has most similar double-metaphone representation in the knowledge base. To calculate the similarity, we use the Levenshtein distance. We generate synthetic data of utterances with objects and actions replaced with phonetically similar words. With a threshold of **55** on the Levenshtein distance, we get an accuracy of **97%** and with a threshold of **70**, we get an accuracy of **87%**.

To test the synonym matching using the MNLI model, we generated synthetic data of the form `"Action Object1. [SEP] Action Object2"` and `"Action1 Object. [SEP] Action2 Object"` with labels 0, 1, 2 corresponding to contradiction, neutral and entailment, respectively. With the model fine-tuned on this data, we got an accuracy of **98%** on the test dataset, consisting of 150 data-points. Table 12 shows qualitative results of phonetic and synonym matching. The highlighted words in the left column are the actions/objects that are matched with highlighted actions/objects from the knowledge base.

Table 12: Qualitative Evaluation of Phonetic and Synonym Mapping

Utterance Text	Corrected Text
pore coffee in the mug	**pour** coffee in the mug
put it in the **ball**	put it in the **bowl**
go to the **seat**	go to the **chair**
brake the **closet**	**break** the **wardrobe**

5.2 Navigation Modules

The MaskRCNN model was evaluated on a test dataset of 60,000 images obtained from the latest build of the Arena simulator. The model was evaluated using two different metrics, `uberMAP` and `cocoMAP`, on all classes, as well as individual subsets of small, medium, and large classes. Classification of small, medium, and large is done as follows:

- `small`: $0 <$ area < 1296 pixel squares
- `medium`: $1296 <$ area < 9216 pixel squares
- `large`: $9216 <$ area < 90000 pixel squares

Table 13: Evaluation Results

MaskRCNN Model Evaluation Results						
Number of	uberMAP			cocoMAP		
Classes	Small	Medium	Large	Small	Medium	Large
86	0.914	0.921	0.849	0.545	0.672	0.690

6 Conclusion and Future Work

Our team's primary contribution to the *Alexa Prize Simbot Challenge* has been in the area of vision-language navigation. Specifically, we focused on modularizing this process and introducing a novel approach to natural language understanding. Our approach involves using generative modeling to break down complex natural language utterances into simple actionable statements, which we ground in visual cues from state-of-the-art object detection and instance segmentation models. We ensure that the system maintains historical information without relying on large language models, which we believe makes it more robust in situations where these models may not be accurate or have access to

17

all relevant information. We have tested our approach thoroughly in simulated environments, and we believe that our work represents a fresh approach over existing approaches to vision-language navigation

With respect to future work, we would like to focus our efforts towards improving our bot's performance through multiple enhancements. We would like to train our own vision model that is able to identify occluded object instances better. We would also like to improve our clarification module, through the integration of the "highlight" functionality. This would additional options to choose multiple highlighted options on the screen, or cycling through multiple options one-by-one.

18

References

Jonathan Francis, Nariaki Kitamura, Felix Labelle, Xiaopeng Lu, Ingrid Navarro, and Jean Oh. Core challenges in embodied vision-language planning. *Journal of Artificial Intelligence Research*, 74: 459–515, 2022.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks, 2020. URL `https://arxiv.org/abs/1912.01734`.

Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gökhan Tür, and Dilek Hakkani-Tür. Teach: Task-driven embodied agents that chat. *CoRR*, abs/2110.00534, 2021. URL `https://dblp.org/rec/journals/corr/abs-2110-00534.bib`.

Jesse Thomason, Michael Murray, Maya Cakmak, and Luke Zettlemoyer. Vision-and-dialog navigation. *CoRR*, abs/1907.04957, 2019. URL `http://arxiv.org/abs/1907.04957`.

D.J. Turnell, Masuma Fatima, and Q.V. Turnell. Simbot—a simulation tool for autonomous robots. volume 5, pages 2986 – 2990 vol.5, 02 2001. ISBN 0-7803-7087-2. doi: 10.1109/ICSMC.z2001.971965.

Piotr Mirowski, Matt Grimes, Mateusz Malinowski, Karl Moritz Hermann, Keith Anderson, Denis Teplyashin, Karen Simonyan, koray kavukcuoglu, Andrew Zisserman, and Raia Hadsell. Learning to navigate in cities without a map. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL `https://proceedings.neurips.cc/paper/2018/file/e034fb6b66aacc1d48f445ddfb08da98-Paper.pdf`.

So Yeon Min, Devendra Singh Chaplot, Pradeep Ravikumar, Yonatan Bisk, and Ruslan Salakhutdinov. Film: Following instructions in language with modular methods, 2021. URL `https://arxiv.org/abs/2110.07342`.

Alexander Pashevich, Cordelia Schmid, and Chen Sun. Episodic transformer for vision-and-language navigation, 2021. URL `https://arxiv.org/abs/2105.06453`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Hao Liu, Yang Liu, Hong He, and Hang Yang. Lebp - language expectation & binding policy: A two-stream framework for embodied vision-and-language interaction task learning agents. *ArXiv*, abs/2203.04637, 2022. URL `https://arxiv.org/abs/2203.04637`.

Weizhe Yuan, Graham Neubig, and Pengfei Liu. Bartscore: Evaluating generated text as text generation. *CoRR*, abs/2106.11520, 2021. URL `https://arxiv.org/abs/2106.11520`.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations, 2019. URL `https://arxiv.org/abs/1909.11942`.

Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. Span-BERT: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77, 2020. doi: 10.1162/tacl_a_00300. URL `https://www.aclweb.org/anthology/2020.tacl-1.5`.

Lluís Màrquez, Xavier Carreras, Kenneth Litkowski, and Suzanne Stevenson. Semantic role labeling: An introduction to the special issue. *Computational Linguistics*, 34:145–159, 06 2008. doi: 10.1162/coli.2008.34.2.145. URL `https://aclanthology.org/J08-2001`.

Peng Shi and Jimmy Lin. Simple bert models for relation extraction and semantic role labeling. *ArXiv*, abs/1904.05255, 2019. URL `https://arxiv.org/abs/1904.05255`.

19

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019. URL `https://arxiv.org/abs/1910.13461`.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28, 2015.

Beth M. Sundheim. Named entity task definition, version 2.1. 1995. URL `https://api.semanticscholar.org/CorpusID:86808326`.

Erik F Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2019. URL `https://arxiv.org/abs/1910.01108`.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL `http://arxiv.org/abs/1907.11692`.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad, 2018. URL `https://arxiv.org/abs/1806.03822`.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention, 2020. URL `https://arxiv.org/abs/2006.03654`.

Adina Williams, Nikita Nangia, and Samuel R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference, 2017. URL `https://arxiv.org/abs/1704.05426`.

Qiaozi Gao, Govind Thattai, Xiaofeng Gao, Suhaila Shakiah, Shreyas Pansare, Vasu Sharma, Gaurav Sukhatme, Hangjie Shi, Bofei Yang, Desheng Zheng, et al. Alexa arena: A user-centric interactive platform for embodied ai. *arXiv preprint arXiv:2303.01586*, 2023.

20