University of Bucharest

Faculty of Mathematics and Informatics

Artificial Intelligence Master

NLP in Video Games

Dissertation Thesis

Scientific Coordinator,
Lect. Dr. Păduraru Ciprian Ionuț

Student,
Holteiu Daniel Ninel

Bucharest, Romania

February 2021

Table of Contents

Contents

Introduction

Machine learning developed a lot in the latest years and artificial intelligence is more and more used in our every day lives. The most visible uses of artificial intelligence are, probably, the virtual assistants such as Siri (created by Apple), Cortana (created by Microsoft) or Alexa (created by Amazon). Apple announced, in 2018, that millions of devices in the world are using Siri voice assistant, although not all the users constantly using it. Also, a study published in 2017 found that 29.9 million people are using voice assistants at least once a month. Another way that AI is being used is in chatbots. A good example is *Lidl UK Messenger Bot* called *'Margot'* which gives users an alternative customer service connection when it comes to wine products. Other good examples of AI usages are Email Filters, Predictive Searches, Photo Editing applications etc.

Video Games are one of the oldest example where Artificial Intelligence is being used. In order to create a virtual world, it, also, has to be populated. This is where AI comes in. Unfortunately, even if machine learning developed a lot in the latest years, video games did not make big steps in this direction. Although AI is present in almost every game, most of the time they are taking predefined actions according to the world's and player's states.

The most used decision making mechanisms in video games are *Decision Trees*. They are used both for letting the player take some decisions and influence the outcome of the game or for AI Bots present in the game, that will take some actions according to the environment.

Objectives

The main objective of this paper is to show that natural language processing solutions can be used in video games to make the experience of the player better and more player friendly. Imagine a wiki page of a video game where you must search in different categories to find wanted information. This paper proposes an idea where the player simply asks of what he wants and the game will present the results immediately, without the need of manual work. It also proposes a way of using natural language to ask for active assistance in the game and not only for information.

History of NLP in Video Games

Being such an AI reliant medium, video games have been very slow in adapting to the use of Natural Language Processing. Because of this, almost no video game uses machine learning or user input to generate dynamic responses.

Mass Effect, created by BioWare is a good example of predefined user speech options that have an influence on the game's story.

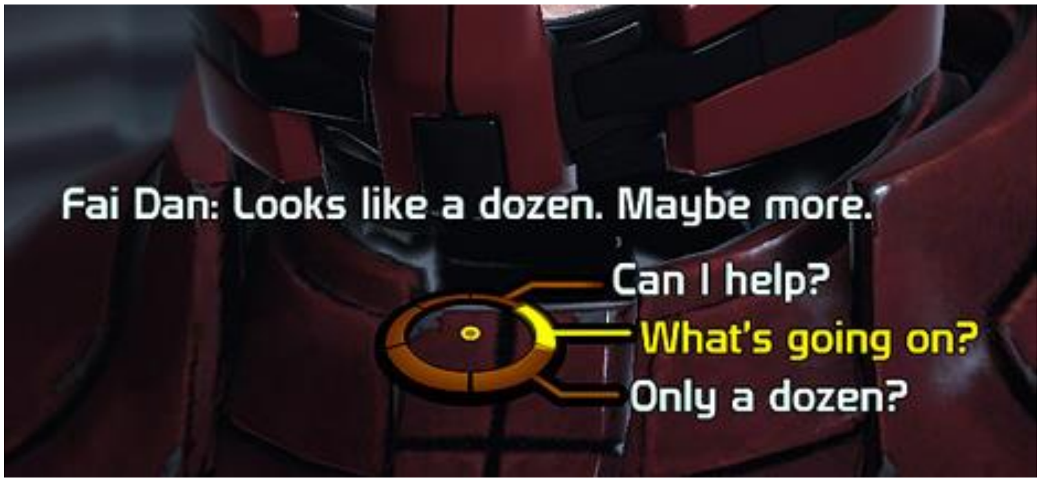


Figure 3.1: "*Mass Effect Conversation Wheel"*

Façade is an AI based game created by Michael Mateas and Andrew Stern. It is one of the first video games where a user inputs natural language in order to control the direction and flow of the game's story. The game won, in 2006, at *"Slamdance Independent Games Festival"* the *“Grand Jury Prize”*. It was also included, in 2010, in the book called *“1001 Video Games You Must Play Before You Die”*.

The story of Façade has the player as a friend of Trip and Grace. The characters invited the player for some drinks, but, the gathering is, however, disturbed by the misunderstanding of the characters. The player can input text to “talk” to the characters driving the conversation to wherever the player wishes to.

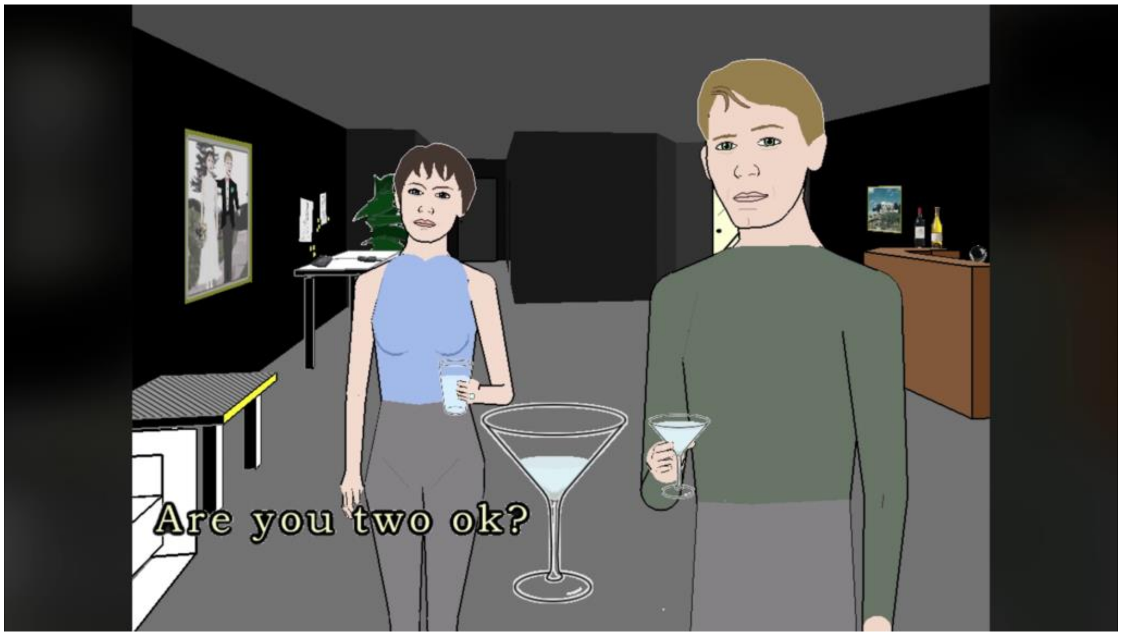


Figure 3.2: “*Façade Game Screenshot”*

Theoretical Concepts

4.1 Natural Language Processing (NLP)

Natural Language Processing (NLP) is a subfield of computer science and artificial intelligence which researches how a computer interacts with human like language. It also studies how teach computers to process large amounts of data containing natural language. Some of the most challenging subjects in NLP are: speech recognition, natural language generation or natural language understanding.

NLP started, at first, in the 1950s when Alan Turing created and published the called "*Computing Machinery and Intelligence"* where he proposed a test which, now, it is called "The Turing Test" which is considered a benchmark of intelligence. The Turing Test (called originally imitation game) is a test for determining how "intelligent" a machine is compared to a human. The Turing Test proposes that the conversations between a human and a machine who generates responses are judged by humans. If the machine cannot be told, reliably, from the human then the machine passes the "Turing Test".

Some of the most commonly researched tasks in NLP are:

1. Processing text to speech
2. Morphological and Syntactic analysis
3. NLP at a higher level:
    a. Text Summarization
    b. Book generation
    c. Dialogue management
    d. etc.

Natural language processing went through different steps in history, starting from Symbolic NLP (1950s – early 1990s) which has its basis on a set of rules on which the computer tries to emutate natural language by applying them to the given data. The next step was

Statistical NLP (1990s – 2010s) which introduces machine learning algorithms for language processing. The last step is Neural NLP (present) which introduces representation learning and deep neural networks in the natural language processing field.

4.2 Natural Language Understanding

Natural Language Understanding (NLU) is a part of Artificial Intelligence and researches the understanding of human-like language and is considered an “AI-hard” problem.

Although NLU is a subtopic of NLP, it has considerable commercial interest in the field as it has multiple real world application: question answering, voice activation, news gathering, automated reasoning, text categorization.

4.3 Fine Tuning

Fine Tunning is a method which helps in creating new models from pre-trained models. It is a method that takes the pretrained weights of a model and use those weights as an initialization for a new model that is trained on new data. The fine tuning method is used to:

- Increase the speed of the training process
- Handle the small datasets

There are several steps that fine tuning consists of:

1. A pre trained neural network model, unsually called the source model, that was trained on a source dataset
2. Creation of a new neural network model called the target model. This model, simply, replicates the designs and the parameters of the pre trained (source) model, an

exception being the output layer. The assumption is that the model parameters are the ones containing the knowledge learned from the source dataset.

3. Addition of a new output layer whose output size is the number of new dataset's target categories and randomly initialize the model parameters for this layer.
4. Train the new model on the given target dataset. The output layer will therefore be trained from scratch and the parameters of all the other layers (pre trained model layers) will be fine tuned based on the parameters of the model.

4.4 BERT

Bidirectional **E**ncoder **R**epresentations from **T**ransformers (BERT) is a new technique that was open-sourced at the end of 2018 by the Google AI Language researchers.

BERT is a type of model trained on a lot of unlabeled text-data which results in a pretrained model that can be fine tuned with only a new additional layer (output layer) which will result in creating a new models which can handle different tasks (i.e. anwering to human questions, intent classification etc.).

There are two types of BERT pretrained models:

1. $BERT_{BASE}$ where L = 12, H = 768 and A = 12 with a total number of parameters equal to 110 million
2. $BERT_{LARGE}$ where L = 24, H = 1024, A = 16 with a total number of parameters equal to 340 million

Where:

- L refers to the number of layers
- H refers to the hidden size
- A refers to the number of self-attention heads

Fine-Tuning a BERT models is straightforward. For each new task just connect the newly created inputs and outputs in the pretrained BERT model and fine tune all the parameters with the new data.

BERT relies on a Transformer. A basic transformer has an encoder which reads the input text and a decoder which produces the predicted results for the specific task. BERT only generates the model which does the language representation which means that it only does the encoding part. The input of the model is a sequence of tokens that are converted into arrays and then are processed, which means that the input has to be pre-processed with extra metadatas:

1. **"Token Embeddings"** – **"[CLS]"** token is added at the beginning of the first sentence. At the end of the same sentence, another token **"[SEP]"** is added.
2. **"Segment Embeddings"** – A simple marker which indicates the segments in a text (denoted by the senteces) so that the model knows the different sentences.
3. **"Positional Embeddings"** – Indicates each token's position in sentence

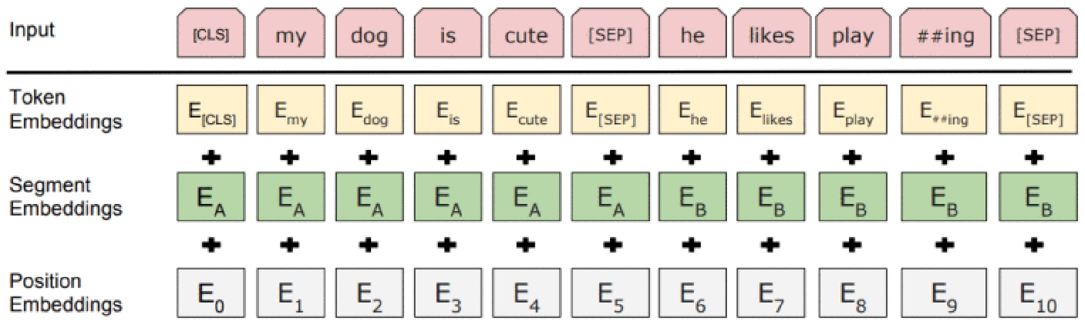


Figure 4.4.1: "*BERT input representation. The inpus embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings*"

BERT managed to obtain new state of the art results on multiple tasks. Some of the most notable being:

1. GLUE – with 80.5% score (7.7% absolute improvement)
2. MultiNLI – with 86.7% accuracy (4.6% absolute improvement)
3. SQuAD v1.1 question answering – with 93.2 F1 score (1.5 absolute improvement)
4. SQuAD v2.0 – with 83.1 F1 score (5.1 absolute improvement)

4.4.1 Intent Classification and Slot Filling

Intent classification is the association of text to a specific purpose. A classifier analyzes the given text and categorizes it into intents of the user.

Slot filling is a common design-pattern in spoken design, usually used so that users are not asked many detailed question and instead are asked a more broad question. Those slots represent, for example, a destination, a departure city or any other information that needs to be extracted from a text.

A joint model is a model doing multiple tasks at the same time. BERT, due to its structure, can easily be extended to a joint intent classification and slot filling model.

The intent can be predicted as:

$\mathbf{y^i = sofrmax(W^i\, h_1 + b^i)}$ where $\mathbf{h_1}$ is the hidden state of the first token **"[CLS]"**

The slot filling part has the other tokens (**h2…hn**) fed into a softmax layer which will lead to classifications for the labels of the given slots.

4.5 Levenshtein Distance

Levenshtein Distance is a metric for string data. Its name comes from the mathematician Vladimir Levenshtein, who proposed this distance function in 1965. It measures the difference between two sequences of string values and, it represents the number of edits (for a single character) which lead to changing one input text to the other input text. The edit operations considered are deletion, insertion and substitution.

Formally, the distance between two strings **x** and **y** is given by **levenshtein(x, y)** where:

- levenshtein(x, y) = |x|, if |y| = 0
- levenshtein (x, y) = |y|, if |x| = 0

- levenshtein (x, y) = levenshtein (t(x), tail(y)), if x[0] = y[0]
- levenshtein (x, y) = 1 + min(levenshtein (t(x), y), levenshtein (x, t(y)), levenshtein (tail(x), t(y))), otherwise.

Where:

- **|x|** is the length of string **x**,
- **|y|** is the length of string **y**
- **t** is a function which deletes the first character of a string and returns the rest
- **z[n]** is the **n**-th character in of string **z**

4.6 Damerau-Levenshtein Distance

Damerau-Levenshtein distance is a string metric coming from information theory and computer science. Its name comes from researcher Frederick J. Damerau and mathematician Vladimir Levenshtein. It measures the difference between two sequences of string values and, it represents the number of edits (for a single character) which lead to changing one input text to the other input text. The edit operations considered are deletion, insertion and substitution, with the addition of transposition compared to the simple “Levenshtein Distance”.

Formally, the Damerau-Levenshtein distance between two texts **x** and **y** is the value function $\mathbf{dist_{x,y}(|x|, |y|)}$ where **i = |x|** is length of string **x** and **j = |y|** is length of string **y**. Where $\mathbf{dist_{x,y}(i, j)}$ represents the value between the prefix of symbol **i** in string **x** and a prefix of symbol **j** in **y**. The definition of the function is:

- $dist_{x,y}(a, b) = 0$, if $a = b = 0$
- $dist_{x,y}(a, b) = dist_{x,y}(a - 1, b) + 1$, if $a > 0$
- $dist_{x,y}(a, b) = dist_{x,y}(a, b - 1) + 1$, if $b > 0$
- $dist_{x,y}(a, b) = dist_{x,y}(a - 1, b - 1) + 1_{(xa <> yb)}$, if $a, b > 0$
- $dist_{x,y}(a, b) = dist_{x,y}(a - 2, b - 2) + 1$, if $a, b > 1$ and $x[a] = y[b - 1]$ and $x[a - 1] = y[b]$

Where:

- $\mathbf{1_{(xa <> yb)}}$ equals 0 when **xa = yb** else it equals **1**

Each of the previous recursive calls match one of the cases covered by Damerau-Levenshtein distance:

- **$dist_{x,y}(a - 1, b) + 1$** - a deletion (from x to y)
- **$dist_{x,y}(a, b - 1) + 1$** - an insertion (from x to y)
- **$dist_{x,y}(a - 1, b - 1) + 1_{(xa <> yb)}$** - a match or mismatch
- **$dist_{x,y}(a - 2, b - 2) + 1$** - a transposition

4.7 Smith-Waterman Distance

Smith-Waterman algorithm is an algorithm which is used to determin regions that are simialr between two strings of nucleic sequences or protein sequences.

This algorithm was created by Temple F. Smith and Michael S. Waterman in 1981. It is based on dynamic programming idea and has, as its base, the "Needleman – Wunsch" algorithm. That being said, there is a guarantee that local alignment, which is also optimal, will be found.

Let **$X = x_1 x_2 \dots x_n$** and **$Y = y_1 y_2 \dots y_m$** be the sequences.

The algorithm has the following steps:

1. Create the "substitution matrix" and the penalty scheme
   a. **sim(x, y)** – Similarity Score
   b. **W_k** – The penalty for gaps of length **k**
2. Create the scoring matrix **S** with the first row and column initialized. This matrix will have the size **(n+1) * (m+1)**.
   $S_{k0} = S_{0l} = 0$ for $0 <= k <= n$ and $0 <= l <= m$
3. Complete the filling of the scoring matrix with the highest value of the function:
   a. **$S_{ij} = S_{i-1, j-1} + sim(x_i, y_j)$**
   b. **$S_{ij} = max_{k >= 1} \{ S_{i-k, j} - W_k \}$**
   c. **$S_{ij} = max_{l >= 1} \{ S_{i, j} - W_l \}$**
   d. **0**
4. Traceback. Start with the biggest score that can be found in S then end with a score of 0.

Implementation

5.1 Game Engine

In order to test the implementation of NLP in video games, a test game was created using Unreal Engine 4 (version 4.25). The implementation of the game is simple and contains:

1. A player character that can move on the map
2. A bot that spawns near the player and takes the commands from the player
3. A simple chat system where the player can write commands for the bot
4. Test locations – situated in different locations on the map

The player can input a command for the bot using the chat. The bot will fetch that input and send it to a server through a http call, which does the whole text processing work. The response is being sent back to the bot and it does an action according to it.

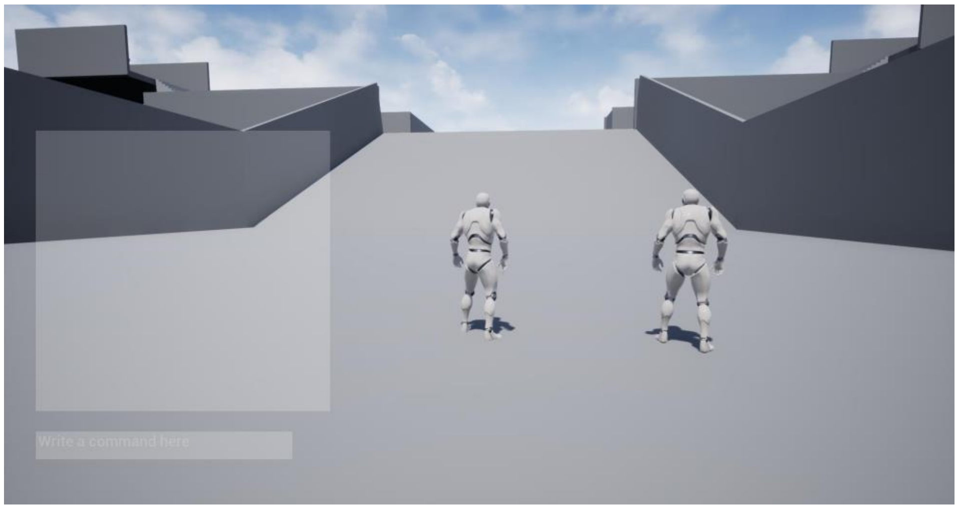


Figure 5.1.1: "*Game Beginning"*

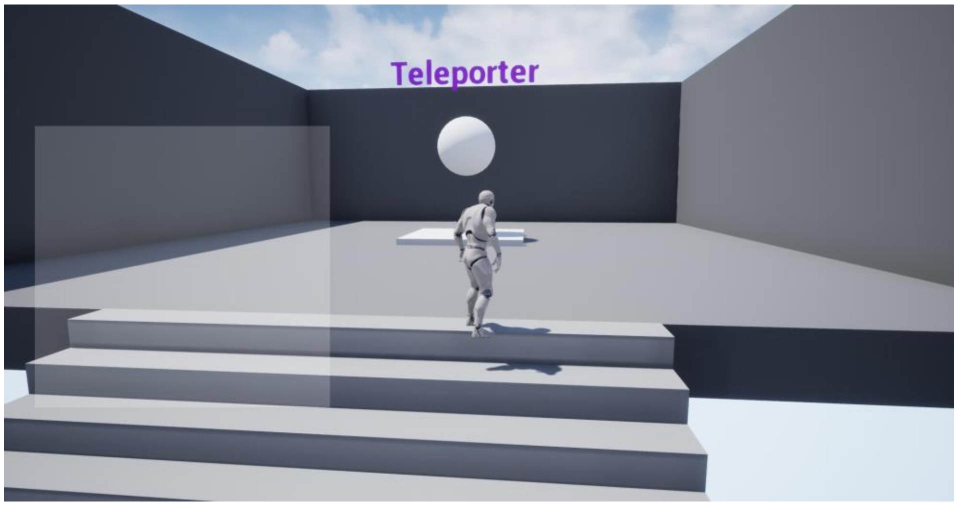


Figure 5.1.2: “*Game Teleporter Location*”

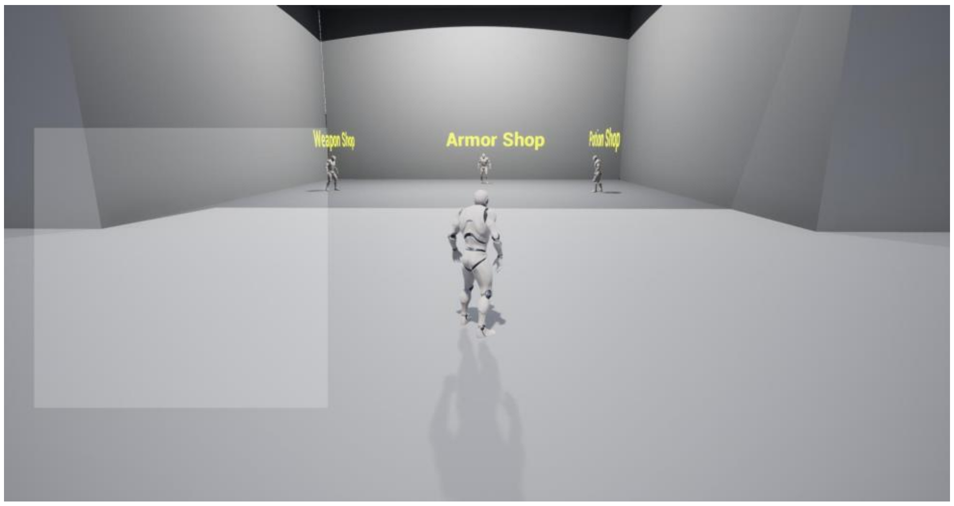


Figure 5.1.3: “*Game Shops Location*”

Figures **5.1.1**, **5.1.2** and **5.1.3** are examples of map locations existent in the game that were used for testing.

5.2 Game Actions and Slots

The implementation of the game and the usage of natural language processing and understanding in the implemented game was limited to two actions.

The first action is called **AnswerQuestion** which has as output an answer a user's question query.

The second action is called **FollowAction** which has as output an action which makes the bot move to a location, that the user requested a path to.

There are, also, slots that are used by the NLP models in order to fetch the location, but can be extended to do more. Those slots are:

1. B-location – Beginning Location slot
2. I-location – Contiuation of a location (a location can contain multiple words)
3. O – Empty slot (nothing of importance to the model)
4. [CLS], [SEP] and [padding] – separators used by BERT

5.3 Models

5.3.1 Three Models

The initial approach for processing the natural language included three different models, each doing a different thing.

The first model was an *intent classification model* which would take the input from the player and decide what the intent is, either *question answering* or *follow action*. According to the result of this first model, one of the other two models would be used.

The second model was a *question answering* model which would take the input from the player and answer to the question, if the intent was decided to be question answering.

The third model was a *location finder* model which would take the input from the player and find, inside that text, the location where the player wants to go, if the intent was decided to be follow action.

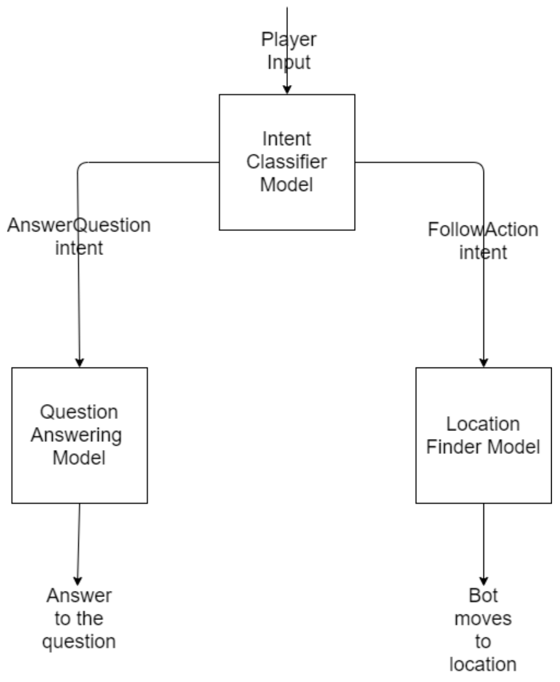


Figure 5.3.1.1: "*Three models data flow*"

This idea was shortly dropped (before full implementation) due to being way too complex for the presented task. Also, the lack of data was a major downside in choosing this implementation idea. The last reason for dropping this idea was a new implementation idea that came up, Joint Model, which seemed to be a better approach.

5.3.2 Joint Model

The second approach that was, shortly, taken into consideration is the idea of a **Joint Intent Classification and Slot Filling Model**. A Joint Intent Classification and Slot Filling model is a model that can predict both the intent of the player and the slots that can be found in player's input. This model merges the *Intent Classifier Model* and a piece of *Location Finder Model* into one single model that would do both.

The model created has the following outputs:

1. **Intent** – A string which represents the intent of the player that can be either *AnswerQuestion* or *FollowAction*
2. **Slots** – A list of strings which represent the slots in the text given by the player. The possible slot values are: *O, B-location* (representing the start marker of a location), *I-location* (representing the continuation of a location). There is one slot per word in the given text.

Example:

- Player input text: "Show me where the potion shop is"
- Intent: FollowAction
- Slots: [O O O O B-location I-location O]

According to the intent that the model outputs there are two possible cases. The first case is the case where the intent is AnswerQuestion which will take the input from the player and answer the question using a question answering model. The second case is the case where the intent is FollowAction in which the location will be extracted from the list of slots.

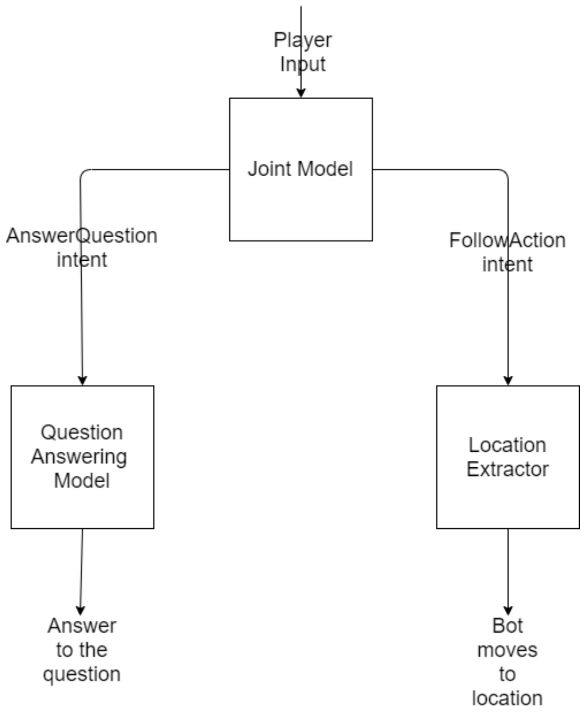


Figure 5.3.2.1: "*Joint model data flow*"

The architecture of the Joint Model is composed of the $\text{BERT}_{\text{BASE}}$ pretrained model that was then fine tuned using new layers that were introduced after the BERT model as the output layers. The input used for this model is composed of an array of arrays: *Input Ids, Segment Ids, Input Mask, Valid Positions*.

The newly created custom layer (base BERT model with the new inputs) will then go into multiple layers that will define the custom output that is required for the Joint Model. This output is used for both the intent classification and slot filling of the Joint Model.

The intent classification layers are:

1. **Dropout** layer with rate **0.1**
2. The result will then go into a **Fully Connected Layer** that has as activation function **softmax**.

The slot filling layers are:

1. **Dropout** layer with rate **0.1**
2. The result will then go into a **Fully Connected Layer** that has as activation function **softmax**
3. The result is then multiplied with the *Valid Positions* input.

The resulted model will have as inputs the previously defined arrays and as output the two layers that were defined: Intent Classification and Slots Filling.

The Optimizer used for this model is **Adam** with a learning rate with the value of **0.00005**.

The loss functions used are as follows:

- For Slots Filling: **Sparse Categorical Crossentropy**
- For Intent Classification: **Sparse Categorical Crossentropy**

The initial loss weights are:

- For Slot Filling: **3.0**
- For Intent Classification: **1.0**

The model was then trained for **10** epochs, using a batch size of **16** and a validation split of **10%**.

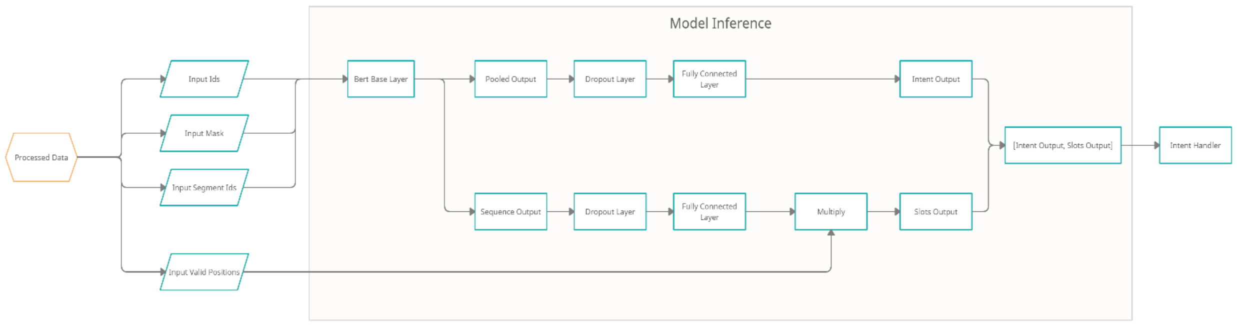


Figure 5.3.2.2: "*Joint Intent Classification and Slot Filling Model Architecture*"

5.3.3 Question Answering Model

The Question Answering Model takes as the input the query of the player and answers to that question if the intent predicted by the Joint Model is question answering.

For this model, the *DeepPavlov* library was used. *DeepPavlov* is an open source conversational AI library built on Tensorflow, Keras and PyTorch. It was built and is maintained by *"Neural Networks and Deep Learning Lab"* at Moscow Institute of Physics and Technology. This library contains a multitude of pretrained models and models that can be fine tuned such as: *Named Entity Recognition, Sentence Similarity, Speech Recognition* etc.

The model used for Question Answering is a *Knowledge Base Question Answering Model*. This model has two inputs:

- The knowledge: representing all the information that the answer will be searched in
- The query: representing the question to be answered

The knowledge text is represented as a text consisting of descriptions of every single location existent on the test map that were concatenated.

In order to answer to a question, the *Knowledge Base Question Answering Model* uses the following models:

- BERT – prediction of query template type
- BERT – extraction of entity substrings from the question
- Entity Linking – performs matching the substring with one of the Wikidata entities. It is based on *Levenshtein* distance.

- BIGRU – ranking of candidate relations
- BERT – ranking candidate relation paths
- Query Generator – used to fill query template with candidate entities and relations.

The answers generated by this model will always be a substring of the *Knowledge* text.

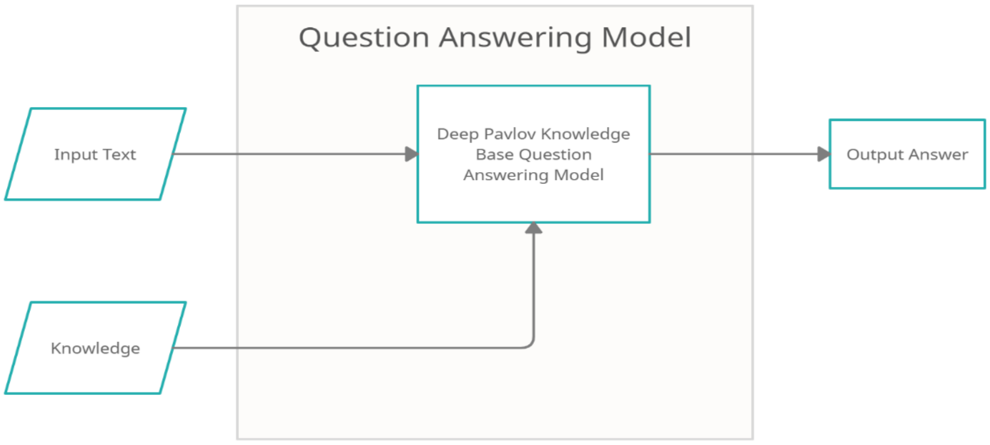


Figure 5.3.3.1: "*Question Answering Model Flow*"

5.4 Location Extractor

Location Extractor is the last step after the Joint Model runs, in case of the *FollowAction* predicted intent of the player. It uses the input text of the player, the slots that were predicted by the Joint Model and the predefined locations (existing in the world) to output a location where the bot has to go or output an error where a location couldn't be found.

The first step is taking the input text of the player and the predicted slots to find the location that the model predicted in the player input. In case of no location found (no location slots were predicted) then there is an error, probably wrongly predicted that it is a follow action.

After finding the location that the player wants to go to, one more step is needed. Taking into account that the extracted location comes from a player (a human), there can be typos in the location string, or the location does not exist. That being said, a similarity function is used to find the most similar location from the already existing locations in the game to the predicted location extracted from the player input. There is, also, a preprocessing step done for all the strings used in the similarity function. The preprocessing envolves getting rid of any white spaces and making every text lowercase.

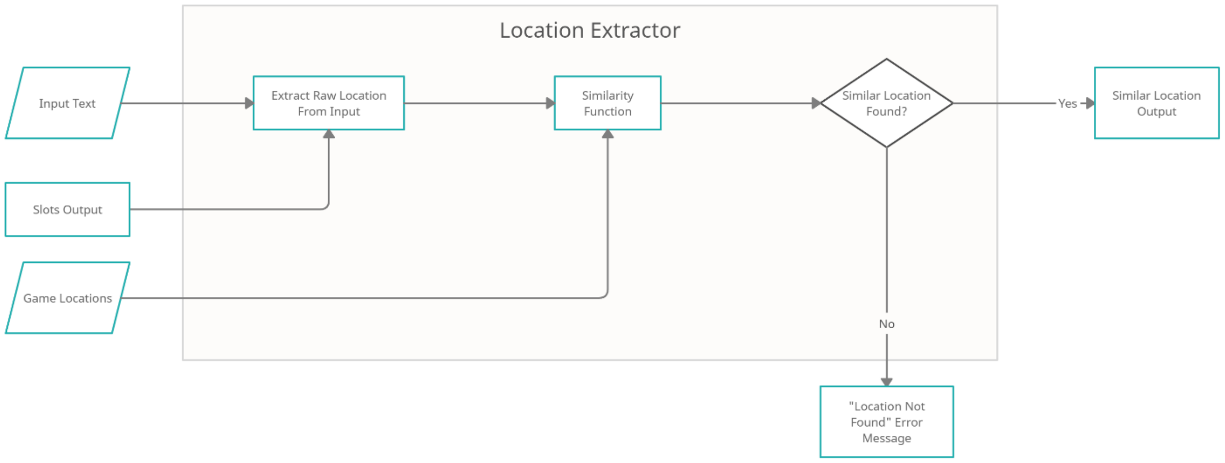


Figure 5.4.1: “*Location Extractor Architecture*”

5.4.1 Levenshtein Distance

Levenshtein distance is the minimum number of edit operations necessary for transforming one sequence into the other. The operations that are taken into consideration are: deletion, insertion and substitution.

Levenshtein Distance is searching for the minium number of changes required to make similar strings, so the minimum levenshtein distance between the input string and the possible locations. There is also a threshold that must be passed in order to consider the two values as being similar. This way the possibility of badly chosen locations due to, for example, length of the string will be reduced. This threshold is set to **8**. If no set passes this threshold then it is consider as a badly input location on the side of the player.

5.4.2 Damerau-Levenshtein Distance

Damerau-Levenshtein distance is the minimum number of edit operations necessary for transforming one sequence into the other. It is, essentially, the same thing as Levenshtein Distance, but, in this case there is one more operation that is added when comparing the two strings, transposition.

Damerau-Levenshtein distance is searching for the minimum number of changes required to make similar strings, so the minimum distance between the input string and the possible locations. There is also a threshold that must be passed in order to consider the two values as being similar. This way the possibility of badly chosen locations due to, for example, length of the string will be reduced. This threshold is set to **8**. If no set passes this threshold then it is considered as a badly input location on the side of the player.

5.4.3 Smith-Waterman Distance

Smith-Waterman algorithm compares segments of the string that have all the possible lengths and tries to find the most optimized similarity value.

Smith-Waterman, compared to the levenshtein distance, will compute a number that is higher the more similar the two strings are, representing the similar segments. That being said, a threshold was also set for it, trying to avoid the case where the input string has nothing in common with all the locations existent. This threshold is set to **1**, so any lower value will not be considered valid.

5.4.4 Sequence Matcher

Sequence Matcher calculates a percentage of how similar two strings are comparing the sequences of the given strings. This algorithm tries to find the longest sequences in both the strings that are exactly the same and will compute what percentage of the strings are similar. That being said, the strings that are not similar at all should be discarded and considered an invalid input, so a threshold was set for it. This threshold is **0.25 (25%)** and every similarity value that is under it will be discarded and will be considered invalid.

5.5 Data

The input data is represented by any text that the player inputs in the chat. The data is then sent to the bot which processes it.

Example of data:

- Where can I heal myself?
- Take me to healing shrine.

The output data is represented by two different types of data:

1. **Intent** – representing the intent of the player predicted by the model which can be either *AnswerQuestion* or *FollowAction*
2. **Slots** – representing an array of tokens that show where in the text that particular information can be found

Data Preprocessing is needed so that the raw data that is sent as input from the player is being recognized by the model. The input can be a text containing any number of characters, so a generic input type must be created to be properly understood by the models.

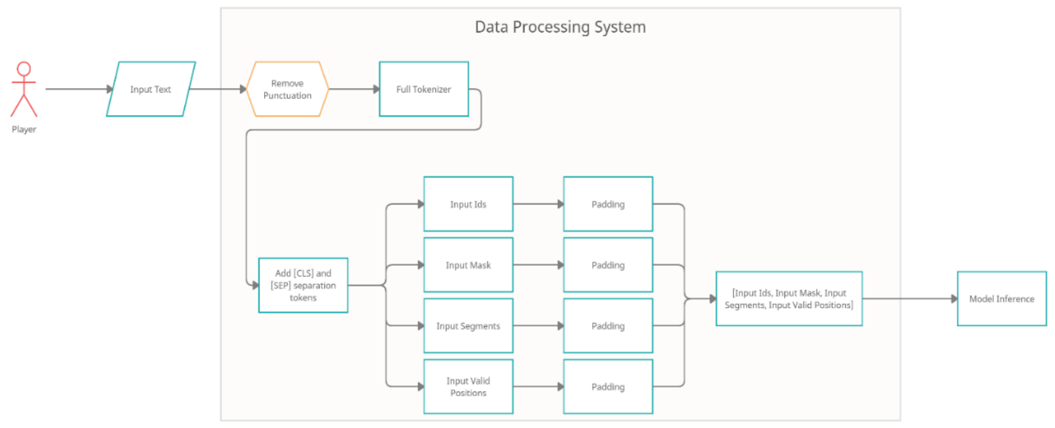


Figure 5.5.1: "*Data Processing System Flow*"

5.5.1 Joint Model Data Preprocessing

For the Joint Model there is an input that is represented by an array of four datas that are preprocessed from the player input text.

1. Input Ids – represents the tokenization of the input
2. Input Mask – is a mask containing only values of 1 padded with 0
3. Input Segments – is a mask containing only values of 0
4. Input Valid Positions – is an array of values of either 0 or 1 where
    a. 1 represents that the given token extracted from the input represents one single word (one-to-one mapping)
    b. 0 represents that the given token comes from a set of multiple tokens of the same word. Only the first token of a word will be considered a *valid position*

The first thing that happens to the player input text is getting rid of all the punctuation and leaving only letters and spaces to be present in the input text. This text, without punctuation, is then tokenized using a *Full Tokenizer*. A full tokenizer will run an end-to-end tokenization of the text that will do a *Basic Tokenization*, which transforms the text into lower case and splits it by white spaces doing a word split of the text and then, for each word, a *Word Piece Tokenizer* runs which does a greedy longest-match-first algorithm and splits the pieces of text (words in this case) into its word pieces (e.g.: "unaffable" will be split into ["un", "##aff", "##able"]).

After the tokenization, two more tokens will be added to the resulted token array: “[CLS]” and “[SEP]” which are separation tokens required by BERT models.

After the tokenization the *Input Ids* can be created by simply converting the resulted tokens to their particular *ids* according to a given vocabulary. The used vocabulary is the one that comes with BERT pretrained model.

The *Input Mask* is then created, which represents an array full of 1s with length of *Input Ids* array.

The *Input Segments* is also created, which represents an array full of 0s.

Lastly, the *Input Valid Positions* is created. This array can have two types of values according to the tokens present in the array of tokens:

a. 0 – if the token contains “##” which represents a word piece token that is not the first in the sequence
b. 1 – otherwise

All the input arrays (*Input Ids, Input Mask, Input Segments* and *Input Valid Positions*) will then be padded with values of 0 (or cut if longer) to the length of a precalculated value representing the maximum length of a sequence which is calculated at training time and represents the maximum length of the array of ids of tokens that were calculated for each input in the training.

5.5.2 Joint Model Output Data Preprocessing

This step is done only at training time. When training, all the inputs and outputs are already present, as the model is a supervised model.

The output must, also, be preprocessed. The only processing that is being done for both the outputs (*Intent* and *Slots*) is label encoding. A label encoder labels the given values (which are strings) to values between 0 and number of classes - 1.

For *Intents* this will mean that the values of the labels will be either 0 or 1 if it is *AnswerQuestion* or *FollowAction*.

For *Slots* there are multiple labels. The first type of labels are the ones custom created for this particular model: *O*, *B-location*, *I-location*, but there are three more labels that are required by the BERT model: *[padding]*, *[CLS]* and *[SEP]*.

5.5.3 Joint Model Output Data Processing

When running the model, for inference purposes, the input text will be preprocessed and the output will contain two values: *Intent* and *Slots* of the given text. As mentioned in **5.5.2** the outpus are represented by numeric values (labels).

The *Intent* output is a single value on which an invers of the labeling process will be ran.

The *Slots* output, on the other hand, represents an array of labels. The first step in getting the slots is inversing the labeling process. After that, a validation must be done using the *Valid Position Input* values that were processed before predicting the slots. This way it will make sure that the proper text values are chosen for each slot.

5.5.4 Question Answering Data Processing

The Question Answering model does not require any data processing to be done. It simply takes as an input the text that the player used as an input and the knowledge text and outputs a substring of the knowledge text that will be sent as an answer to the player.

5.5.5 Location Extractor Data Processing

The Location Extractor requires a bit of data processing. Given the player input text and the slots that were predicted by the Joint Model, the location will be extracted using the slots *B-*

location and *I-location* from the corresponding positions in the input text. All the locations found will then go through the *Location Extractor* to choose the best location from the list of already existing locations in the world.

5.6 Complete Data Flow

In Figure *5.6.1* the complete data flow is defined. It shows where the data starts from, what systems it goes through, the models and what decisions are being done according to the results.

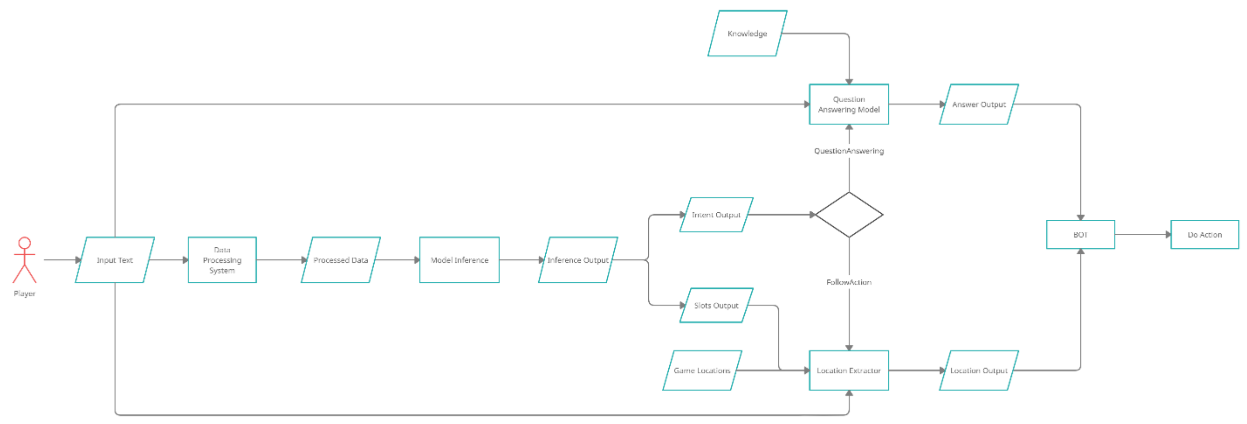


Figure 5.6.1: “*Complete FlowDiagram*”

5.7 Project Initialization

A test project was created in order to be able to test the implementation described above. The project can be found at https://github.com/daneel95/NLP-In-Video-Games .

In order to run a test the following steps must be followed:

1. Run the **Joint Model** – this can be done by going to the directory called JointModel and running **inference.py** main. It will start a flask server on port 5000. It will download the

pretrained models. If another training is required, then run **train_joint_model.py** which will start the training process.

2. Run the **Question Answering Model** – this can be done by going to the directory called QuestionAnswering and running **answer_question.py** main. It will start a flask server on port 5001. No training is required. The context can be changed from within **answer_question.py** file, the global variable called **CONTEXT**. Another way of changing the context is through a HTTP Post call on **/context** endpoint.
3. Open the project game using Unreal Engine 4, version 4.25. The game can be either ran from withing the engine or can be built and ran from outside.
4. When running the game press **ENTER** to insert data into the game chat and ask the bot for information.
5. The game will call the **localhost** on port 5000 to run the Joint Model and wait for a response, so the flask server must be up and running.

Results

The BERT model, as mentioned in 5.3.2, was fine tuned in order to extract information about the intent of the player (either *AnswerQuestion* or *FollowAction*) and the slots which, in this particular case, will only represent a location. According to the intent, the flow will get into either a *Question Answering* model or will simply just extract the location found from the slots.

6.1 Datasets

For this experiment, two different datasets were used which led to two different joint models. Both datasets consist of two sets of datas: training data and test data. The training data was used for training of the model, while test data was used only for testing and extracting metrics of the joint model.

All the data that was used come from csv files with the collumns: text, intent and slots. Example of data:

- Text: "Where can I buy potions?", Intent: "AnswerQuestion", Slots: [O O O O O]
- Text: "Get me to the teleporter", Intent: "FollowAction", Slots: [O O O O B-location]
- Text: "Take me to the healing shrine.", Intent: "FollowAction", Slots: [O O O O B-location I-location]

6.1.1 Dataset 1

The first dataset used for training the joint model was created from the data provided by one person. It consists of **19** entries in the training dataset and the testing was done on only **9** entries.

There are multiple problems with this dataset. The first problem is that it is extremely small so the model may not learn that much from it. The second, and probably the most important one, is the fact that it is biased. Taking into account that the whole dataset was provided by a single person, it means that the whole dataset consists of data that look alike. That being said, the model will learn to understand that person's input, but when provided new input, from another person, it may lead to bad predictions. The last problem that can be seen in this dataset is the fact that it is a "perfect" dataset. In this case, the definition of a "perfect" dataset is a dataset that has no grammatical errors and no typos. Also, all the locations mentioned in this dataset are already existing locations. This is simply unrealistic in the real world. This leads to bad predictions when a typo or a grammatical error occurs.

6.1.2 Dataset 2

The second dataset that is used tries to solve the problems that can be found in the first dataset. This new dataset consists of **156** entries in the training dataset and the testing was done on **36** entries. The increase in dataset entries came from the fact that the second dataset was collected from inputs from multiple people.

From the beginning it can be noticed that the training dataset increased by **~8 times** and the testing dataset increased by **~4 times** so this will lead to more data that can be learned from and also more precise testing of the model. This means that the first problem was, to some extent, solved.

The second problem mentioned in **Dataset 1** was the bias of the data. It came up due to the fact that the whole dataset was created by one single person. This problem was solved, or at least reduced, by collecting data from more people. In this case the data was collected from **6** different people which leads to a way less biased dataset compared the the first one. This way, the model will learn a wider variety of inputs that are different, leading to less bias.

The last problem that was also solved is the "perfect" dataset problem. This time, typos and grammatical errors were purposely introduced in both the training and testing datasets so that it reflects the reality better. Also, new locations that were not present in the training dataset and new “templates” of the input that were not in the training dataset were introduced in the testing dataset so that the extracted metrics are more realistic.

The last thing that was done on Dataset 2 was the balancing of the data by intent, leading to **88** training data with intent AnswerQuestion and **68** training data with intent FollowAction and **17** test data with intent AnswerQuestion and **19** test data with intent FollowAction.

6.2 Joint Model Results

The joint model was trained, at the beginning, on Dataset 1 and then it was retrained on the improved Dataset 2.

As mentioned in 5.3.2 the joint model is, basically, a $\text{BERT}_{\text{BASE}}$ model with new output layers. This BERT model was only fine tuned. An initial result that can be seen immediately is the fast training time due to the fact that only fine tuning was done on the model and not all the millions of parameters had to be retrained. Another reason for the fast training are small datasets used. Even if Dataset 2 is larger than Dataset 1, they are, both, still extremely small.

The testing of the Joint Model was done on the two outputs of the model: Intent and Slots. For the intent Precision, Recall, F1-Score and Accuracy were the metric taken into consideration and for slots the F1-Score was the metric extracted.

6.2.1 Joint Model Results – Dataset 1

On the first dataset, which is a smaller dataset, pretty good results were seen. For Intent, the accuracy of the model was calculated at **89%** which is a lot considering how small and biased the training dataset is. It may, however, have such an accuracy due to the fact that tha test dataset is small.

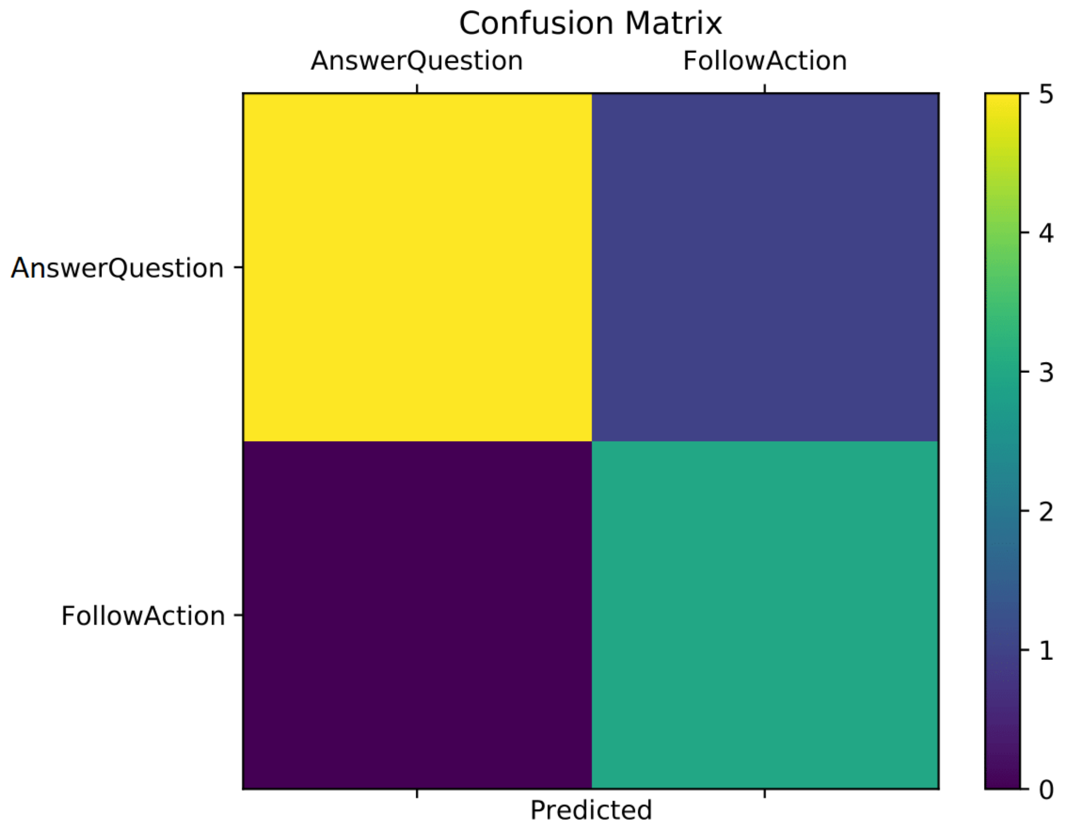


Figure 6.2.1.1: “*Confusion Matrix for Joint Model Intent Classification – Dataset 1*“

From the confusion matrix in Figure 6.2.1.1 it can be noticed that the AnswerQuestion intent was properly predicted. On the other hand, the FollowAction seem to have been missed a couple of times. It can be seen that not all FollowAction data was considered by the model as actual FollowAction and were predicted as AnswerQuestion.

Those results seem to be in touch with the other metrics that were extracted. For AnswerQuestion intent, the precision on the test dataset is **100%**, the recall is **83%** and the f1-score was calculated at **91%**. This shows that the AnswerQuestion intent was, most of the time, properly predicted.

Compared to the AnswerQuestion intent, the FollowAction intent was a bit missed by the model. With a precision of **75%**, a recall of **100%** and a f1-score of **86%** it is clear that the model missed the FollowAction multiple times. This can also be seen, as mentioned above, in the confusion matrix in figure 6.2.1.1 where it is clear that FollowAction was missed more than the AnswerQuestion.

The slots predictions, even if the training dataset was pretty small, seem to still have been predicted pretty well. The f1-score of this output is **90.47%** which is pretty high considering the dataset. This result may also come due to the fact that the FollowAction intent was poorly predicted and it seems to be on the same line, as locations exist, mostly, for FollowAction intents.

6.2.2 Joint Model Results – Dataset 2

The second dataset which has a higher number of training datas and a higher number of test datas seem to have benefitted from this increase. Its accuracy on the test dataset is **100%** which is the best score that can achieved. There are multiple possible explanations for this “perfect” score. The first one is that the model was overfitted and the test data, even if it was processed, was on the same line as training data which led to perfect predictions. The second possible reason for this score may be the fact that the intent classification is actually extremely easy to learn: there are only two classes that need to be predicted. Lastly, the model may actually be “perfect” when it comes to intent classification. The answer is, probably, somewhere in between. The BERT model is extremely powerful which leads to good results even when using a small dataset. While the dataset is not extremely large, it seems to be enough to learn how to predict only two intents. Theoretically, the more intents that must be predicted, the more data must be provided for a better score.

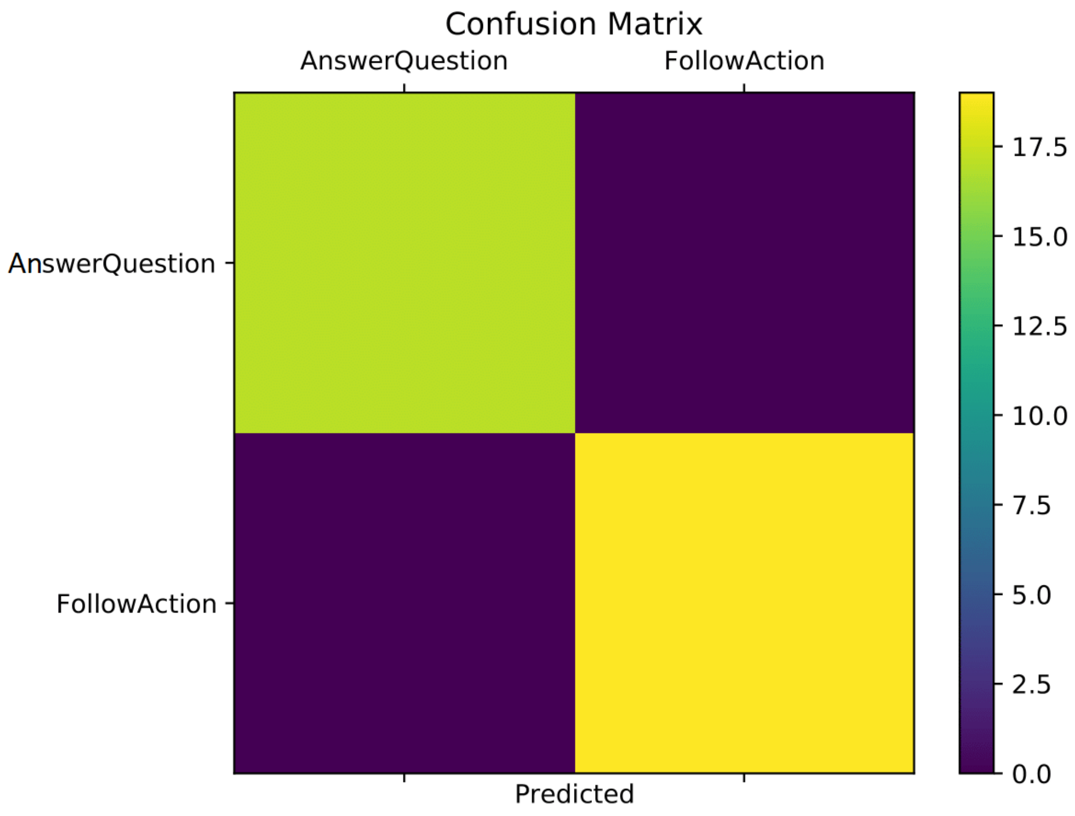


Figure 6.2.2.1: "*Confusion Matrix for Joint Model Intent Classification – Dataset 2*"

From the confusion matrix in Figure 6.2.2.1 it can be noticed that the model is not exactly prefect and it seems to have missed some AnswerQuestion predictions (but there seem to be a small number of missed predictions). Basically, it tends to be perfect but it's not.

The other metrics that were extracted, as in the case of dataset 1, were on the same line as the accuracy and confusion matrix datas. For AnswerQuestion the precision on the test dataset is **100%**, the recall is **100%** and the f1-score is, also, **100%**. The same thing repeats for FollowAction intent with precision of **100%**, recall of **100%** and a f1-score of **100%**. It looks perfect at first sight but, as we noticed in the confusion matrix, it is not exactly perfect. Another reason, as mention when talked about accuracy, may be because it has only two classes that must be predicted and it is easier to learn this than multiple classes.

The slots prediction, as expected, increased. Its f1-score was calculated at **96.62%**. This is a realistic increase from the first dataset results due to more data being used for fine tuning. This shows that the model is able to properly predict the slots from an input of the model. This, however, should be taken with a grain of salt, because there is only one type of slot (referring to location). If more slots are to be added, then more that must be provided to properly predict the slots.

6.3 Question Answering Model Results

The tests that were done on the question answering model, which is a Knowledge Based Question Answering model, showed that the responses of the model are extremely dependent on how the knowledge looks like and also it seems to be dependent on how much information the query gives. As mentioned in 5.3.3 the knowledge was composed of a description from all the locations on the map concatenated into a single text.

For the question answering, there are case when the response to the question is relevant and, sometimes, the response will not be the desired one.

Example of well answered questions:

- Question: “Where can I heal?” – Answer: “Shrine heals your HP to maximum”
- Question: “Where can I teleport?” – Answer: “Teleporter teleports you to the desired location”

Example of badly answered questions:

- Question: “Where can I slay a Dragon?” – Answer: “Weapon Shop”
- Question: “Where can I do tutorial” – Answer: “You can buy weapons at the Weapon Shop”

It can be seen that, when the model properly answers to the question, the answers are good enough for the player to then use that information for a FollowAction. On the other hand, the bad answers are way out of context compared to the question. This may be the case because the knowledge concatenates all the information of the locations and the *Weapon Shop* location description comes before the *Tutorial Dungeon* location description. After swapping the locations descriptions locations it could be seen that the question “Where can I do tutorial” gave the answer “Tutorial Dungeon is a dungeon” which, even if it still makes no sense, this time came from the right description. The reason for the answer having not much sese is because the descriptions weren't, probably, properly created and it simply picked a substring that felt right for the question to be answered.

The conclusion of the results is that the better the knowledge looks like then the better the answers.

6.4 Location Extractor Results

The Location Extractor is, basically, only an interpreter of the slots predicted by the Joint Model over the input text. The extractor simply takes the string from the input text that is represented by the slots then this string is compared with all the possible locations so that the most similar location is selected.

In order to test the extractor it is enough to test the similarity functions, as the slots are tested in the joint model. In order to test the four similarity functions (Levenshtein Distance, Damerau Levenshtein Distance, Smith Waterman and Sequence Matcher) a new test dataset was created. This dataset contains the extracted string from the input text and the expected location value. This new dataset contains **30** datas and were extracted from the training and testing datasets. Then, for this whole new dataset, the accuracy is calculated (how many correct choices were done from the maximum number of choices).

Examples:

- Input: healing shrine, Expected: Healing Shrine
- Input: tutorial, Expected: Tutorial Dungeon
- Input: cstl of dom, Expected: Castle Of Doom Dungeon
- Input: teleport thingy, Expected: Teleporter

6.4.1 Levenshtein Distance Results

The Levenshtein Distance is the first distance that came into mind when a similarity function was taken into account. It has an accuracy of **74.19%**. The Levenshtein Distance similarity was able to properly choose the good location when it came to small typos or small grammatical errors (for example “armr shp” was properly predicted as “Armor Shop”) but when it came to differences in length of the input locations and the expected locations the similarity function failed. For example the input “castle of doom dungeon entrace north” had the expected value of “Castle Of Doom Dungeon” but the similarity function couldn’t find any similar values that would respect the threshold. Another example is the input “castle doom” which had the expected value of “Castle Of Doom Dungeon” but predicted the value “Armor Shop”. This means that, there were less changes necessary to get to the predicted value than to the expected value.

6.4.2 Damerau-Levenshtein Distance Results

Damerau-Levenshtein Distance is an extension of the Levenshtein Distance so it made sense to try and extend it. It has an accuracy of **74.19%**, which si the same accuracy as Levenshtein Distance. In fact, exactly the same results were seen on this similarity algorithm as the Levenshtein algorithm with the same problems occurring. That being said, it did well when it came to small typos or grammatical errors and it did really bad when it came to differences in length of the input location and the expected location.

6.4.3 Smith-Waterman Distance Results

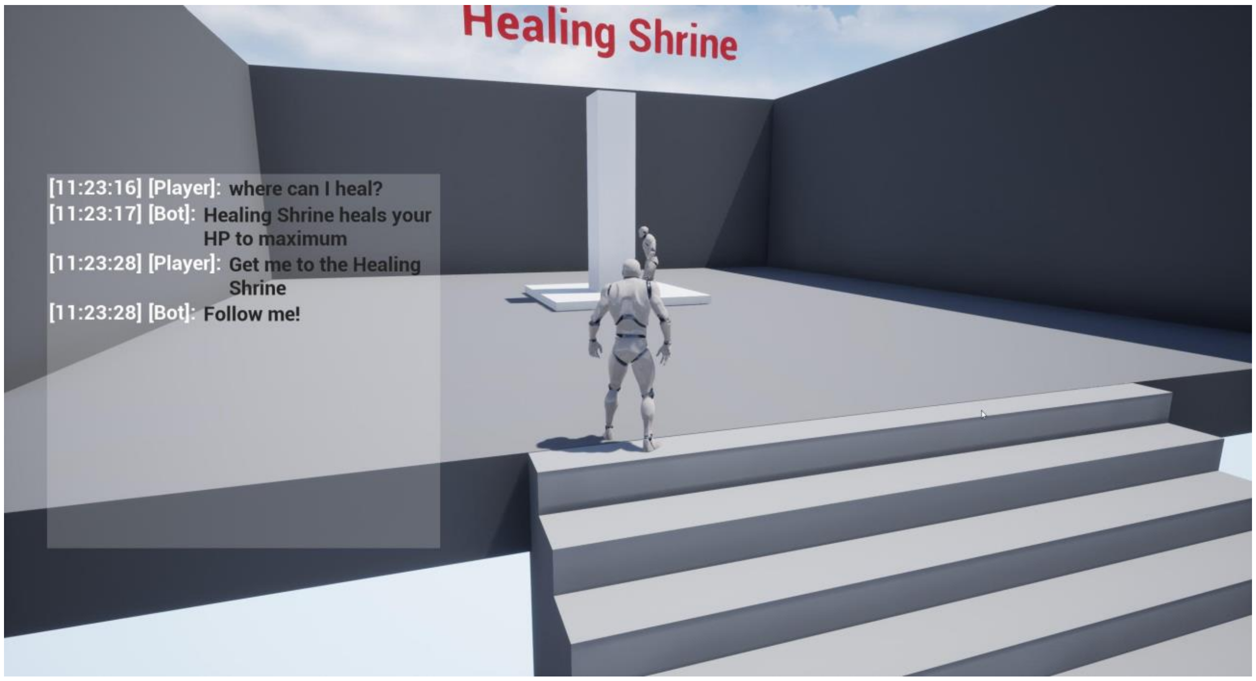
The Smith-Waterman Distance was chosen so that another approach could be taken into consideration when it comes to the similarity function. It has an accuracy of **61.29%**, which is lower than both Levenshtein and Damerau-Levenshtein similarity functions. From the results extracted, it seems that the Smith-Waterman is still good enough when it comes to small typos and grammatical erros but does way worse than the Levenshtein similarity when it comes to different lengths. A good example is “pot shop” which has as expected value “Potion Shop” but was predicted to be “Weapon Shop” which is really bad taking into account that the input text gave a lot of information to the extractor.

6.4.4 Sequence Matcher Distance Results

The last similarity function that was taken into account is a simple Sequnce Matcher Distance similarity function. It has an accuracy of **96.77%**, which is the highest of the four similarity functions taken into account. The only badly predicted input was “doom” which has the expected location as “Castle Of Doom Dungeon” but was predicted to be “Dragon Dungeon”. This input seem to have been badly predicted because of the lack of information from the input location. This resul places the Sequence Matcher on the 1st place when it comes to similarity function in this project.

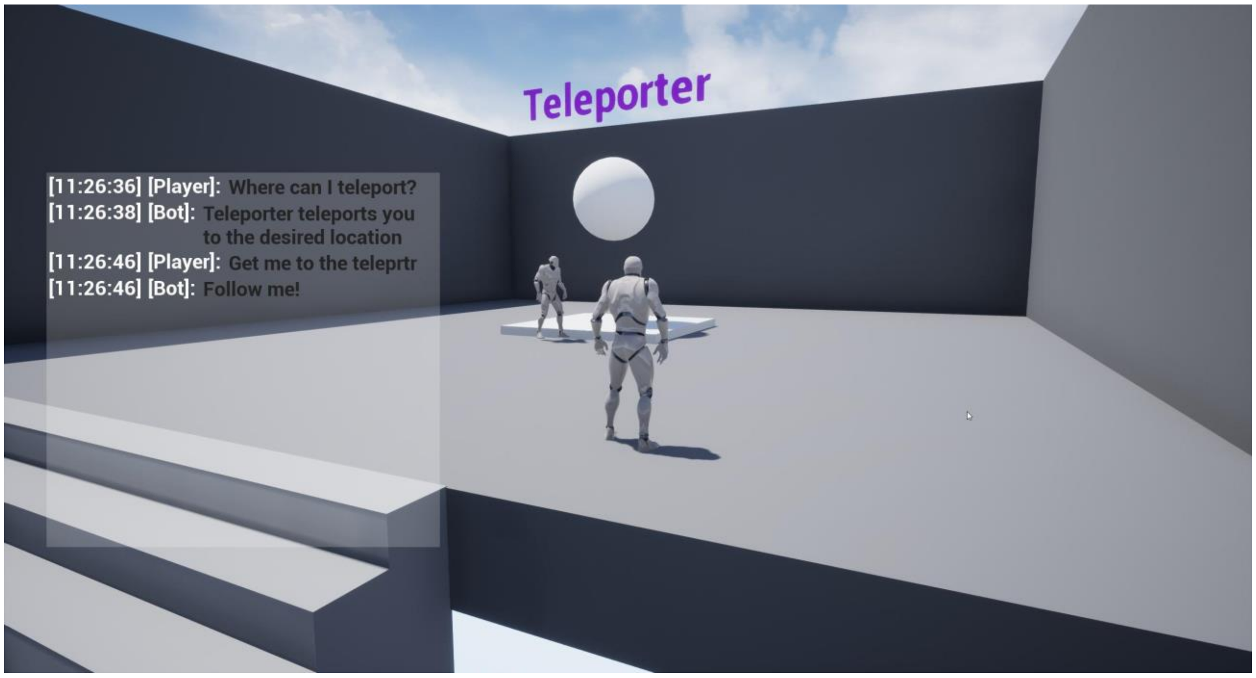
6.5 Game Flow Examples

All the results presented above have an inpact in the game itself. Bellow some screen shots of the game, where the bot responded to the user input and took different actions, can be seen.



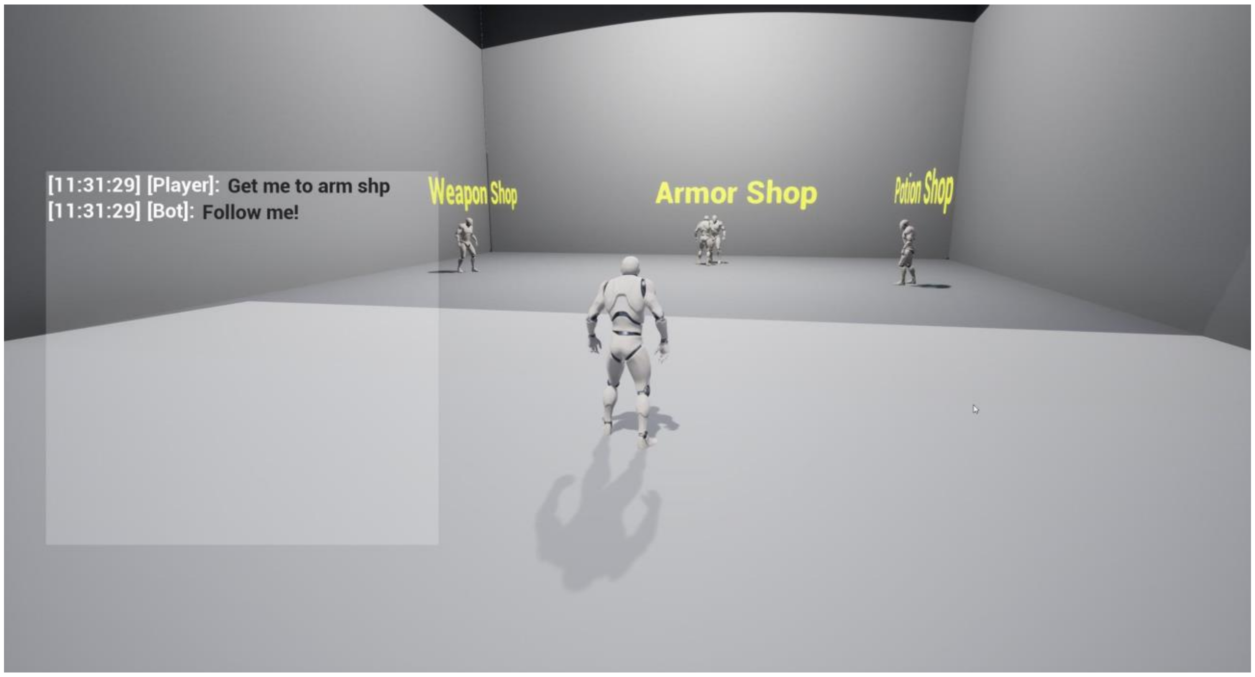
[11:23:16] [Player]: where can I heal?
[11:23:17] [Bot]: Healing Shrine heals your HP to maximum
[11:23:28] [Player]: Get me to the Healing Shrine
[11:23:28] [Bot]: Follow me!

Figure 6.5.1: *"Game Image of Healing Shrine location pinpointed by the bot and question about Healing Shrine answered"*



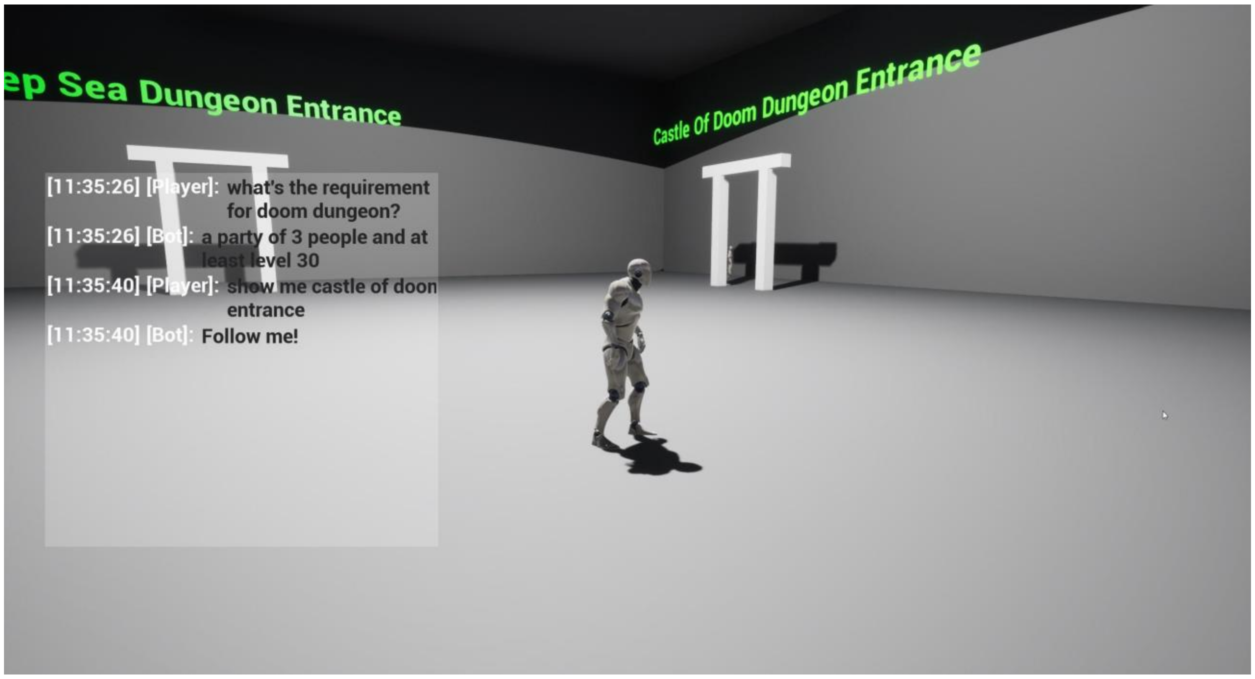
[11:26:36] [Player]: Where can I teleport?
[11:26:38] [Bot]: Teleporter teleports you to the desired location
[11:26:46] [Player]: Get me to the teleprtr
[11:26:46] [Bot]: Follow me!

Figure 6.5.2: *"Game Image of Teleporter location pinpointed by the bot and question about Teleporter answered"*



[11:31:29] [Player]: Get me to arm shp
[11:31:29] [Bot]: Follow me!

Figure 6.5.3: *“Game Image of Shops location pinpointed by the bot”*



[11:35:26] [Player]: what's the requirement for doom dungeon?
[11:35:26] [Bot]: a party of 3 people and at least level 30
[11:35:40] [Player]: show me castle of doon entrance
[11:35:40] [Bot]: Follow me!

Figure 6.5.4: *“Game Image of Castle Of Doom Dungeon Entrance location pinpointed by the bot and question about Castle Of Doom Dungeon requirements answered”*

Improvements and Extensions

The results presented in the previous sections seem to be primising, but further improvements and extensions can be done in order to make it usable in a real world (real game) context.

7.1 Improvement to datasets

The reliability of the models can be improved by fine tuning the them on more data that is, also, less biased. The more sparse the dataset the better the model, in theory. In this paper it could be seen that an extremely small dataset with a lot of biased data gave worse results and when the dataset became a little bit less biased (data gathered from multiple people) it could be seen that the models improved. So this leads to the theory that, the more data given to the models, the better the results. Also, if the models are extended to do way more things than they do in this paper, the data improvement becomes a requirement for good results.

7.2 Increase Intents and Slots numbers

In this paper, the Joint Model created had only two intents (AnswerQuestion and FollowAction) and only one type of slot representing the location. In a real life game the number of intents is higher than that, mostly depending on the game type. Let's say that the game, for example, is a Role Playing Game (RPG). In this case the intents can be AnswerQuestion, FollowAction, Heal, FightAlongside, FightMinions etc. Those intent examples can lead to slots representing location, what to fight (for example: "Help me fight the Dragon boss"), when to heal ("Heal me when my HP is lower than 50%") etc. The number of intents and slots is only bound to the needs of the game and can be extended easily with the only requirement being retraining (fine tuning again) with a new dataset containing new datas for the new intents and slots.

7.3 Change Question Answering Model

The main downside of the Knowledge Based Question Answering Model, as found in the experiment done in this paper, seem to be the knowledge text itself. The main problem seem to have been that, if the model finds information that, for it, seem to be good then it will respond with that. In real life this may not be the case as the answer may be different from what it is expected.

The first idea of improving the Question Answering model is, first of all, changing how we look at the knowledge. Instead of combining all the information into a big chunk of text, it might be better to just try and find in what part of the information to try and find an answer, so, basically, another preprocessing on the query to lower the search text.

Another idea is to train a model to answer question. BERT can be used for this. Unfortunately, this idea requires a well defined dataset so that the model has a good knowledge base, but, if trained correctly and on a good dataset it should have better results than a knowledge based model.

7.4 GPT-3

Generative Pre-trained Transformer 3 (GPT-3) is a new model created by OpenAI research laboratory which tries to produce text that is comparable to that of a human. It was introduced in May 2020 and began testing in July 2020. According to the OpenAi researchers, the model has 175 billion parameters.

Instead of using BERT to fine-tune the model, a good idea of extending the models used in this paper is to use GPT-3 and fine-tune new models which may show better results for this particular case.

7.5 Online Training

If it can be assumed that the models presented can be incorporated into a game it is obvious that different people will have interactions with the models. That being said, a good idea would be to do an online training to further fine-tune the models. The players will play the role of "data creators" while playing the game and using the models. This way the model will continuously improve to the needs of the players of the game.

7.6 Improve Model Understanding

While testing the models it was obviously hard to find out why the models were making some choices in some particular cases which led to some behaviours that were hard to understand.

While researching, an interesting tool called *Language Interpretability Tool (LIT)* was found. It is a visual and interactive model - understanding tool for any natual language processing model. The job of this tool to answer to questions regarding the models: "What kind of examples does the model perform poorly on?", "Why did my model make this prediction?" or "Does my model behave consistently?". If we can answer to these questions, then, the more insight can be extracted from the model and its behaviour can be understood better. This tool was not used in this paper, because it requires a reimplementation of all the models according to the interfaces that it comes with.

Conclusions

The objective of the paper was achieved with the presented implementation. The models created managed to use the natural language introduced by a player to make decisions in the game.

It could be observed that the more biased the data the worst the model is. We presented in the paper the two different datasets, one consisting of the smaller dataset with biased data (coming from one person) which had worse results than the second dataset which was less biased, because those datas came from multiple people. So a general conclusion would be that, the more diverse the dataset, the better.

Other important factors in the dataset that should be introduced are the grammatical errors and typos. Typos and grammatical errors are extremely common and should be properly understood by the models. If there is one bad letter, the model should act as if it was written fine. This problem was solved in the second dataset presented in the paper in which typos and grammatical errors or purposely introduced.

The distance algorithm seem to have given good results. Instead of using an overly complex model for doing this, a simple similarity algorithm was good enough. More than that, the simplest Sequence Matcher gave the best results when calculating similarity. This was needed because, as mentioned previously, typos and grammatical errors happen often, which may have led to “unknown locations”. This way a similar location from the list of the already existing ones was chosen.

To conclude, the results of this paper show that natural language can be introduced in video games as an assistant for the player. It is obvious that, with more and more complex games, it is hard to find the information need if searching, for example, in an in game wiki. It is more straight forward to use natural language to ask a question or to ask for assistance with the game.

Bibliography

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, October 2018 (revised May 2019) (https://arxiv.org/pdf/1810.04805v2.pdf)

[2] Qian Chen, Zhu Zhuo, Wen Wang, “BERT for Joint Intent Classification and Slot Filling”, February 2019 (https://arxiv.org/pdf/1902.10909.pdf)

[3] OpenAI Team, “Language Models are Few-Shot Learners”, May 2020 (https://arxiv.org/pdf/2005.14165.pdf)

[4] Aston Zhang, Zack C. Lipton, Mu Li, Alex J. Smola, “Dive into Deep Learning” book (https://d2l.ai/index.html)

[5] Lian Meng, Minlie Huang, “Dialogue Intent Classification with Long Short-Term Memory Networks”, 2018

[6] Zhiwei Zhao, Youzheng Wu, “Attention-based Convolutional Neural Networks for Sentence Classification”, 2016

[7] Hantei Zhang, Hua Xu, Ting-En Lin, “Deep Open Intent Classification with Adaptive Decision Boundary”, 2020

[8] G. Mesnil et al., "Using Recurrent Neural Networks for Slot Filling in Spoken Language Understanding", 2014

[9] Gakuto Kurata, Bing Xiang, Bowen Zhou, Mo Yu, “Leveraging Sentence-level Information with Encoder LSTM for Semantic Slot Filling”, 2016

[10] Yufan Wang, Li Tang, Tingting He, “Attention-Based CNN-BLSTM Networks for Joint Intent Detection and Slot Filling”, 2018

[11] Mauajama Firdaus, Shobhit Bhatnagar, Asif Ekbal, Pushpak Bhattacharyya, “A Deep Learning Based Multi-task Ensemble Model for Intent Detection and Slot Filling in Spoken Language Understanding”, 2018

[12] Natural language processing, Wikipedia(https://en.wikipedia.org/wiki/Natural_language_processing)

[13] Natural-language understanding, Wikipedia(https://en.wikipedia.org/wiki/Natural-language_understanding)

[14] Levenshtein distance, Wikipedia(https://en.wikipedia.org/wiki/Levenshtein_distance)

[15] Damerau-Levenshtein distance, Wikipedia(https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance)

[16] Smith-Waterman algorithm, Wikipedia(https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm)

[17] "joint-intent-classification-and-slot-filling-based-on-BERT" (https://github.com/90217/joint-intent-classification-and-slot-filling-based-on-BERT)

[18] ChatDesk, "Lidl-Chatbot" (https://www.chatbotguide.org/lidl-chatbot)

[19] Ana Bera, "Is Siri Better Than Google?" (https://safeatlast.co/blog/siri-statistics/)

[20] Samia Khalid, "Towards Machine Learning" (https://towardsml.com/2019/09/17/bert-explained-a-complete-guide-with-theory-and-tutorial/)

[21] Moscow Institute of Physics and Technology (MIPT), "DeepPavlov", (http://docs.deeppavlov.ai/en/master/, https://github.com/deepmipt/DeepPavlov)