ENHANCING CONVERSATIONAL AI MODEL PERFORMANCE AND EXPLAINABILITY FOR SINHALA-ENGLISH BILINGUAL SPEAKERS

Dissanayake D.M.I.M., Hameed M.S., Sakalasooriya S.A.H.A.,
Jayasinghe D.T.
(IT19069432, IT19064932, IT19051208, IT19075754)

B.Sc. (Hons) Degree in Information Technology

Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka

September 2022

ENHANCING CONVERSATIONAL AI MODEL PERFORMANCE AND EXPLAINABILITY FOR SINHALA-ENGLISH BILINGUAL SPEAKERS

Dissanayake D.M.I.M., Hameed M.S., Sakalasooriya S.A.H.A.,
Jayasinghe D.T.
(IT19069432, IT19064932, IT19051208, IT19075754)

The dissertation was submitted in partial fulfillment of the requirements for the B.Sc. Special Honors degree in Information Technology

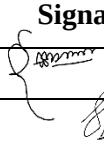
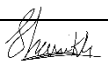
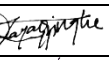
Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka

September 2022

DECLARATION, COPYRIGHT STATEMENT AND THE STATEMENT OF THE SUPERVISORS

We declare that this is our own work, and this dissertation does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of our knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. Also, we hereby grant to Sri Lanka Institute of Information Technology the non-exclusive right to reproduce and redistribute our dissertation in whole or part in future works (such as articles or books).

Name	Student ID	Signature
Dissanayake D.M.I.M.	IT19069432	
Hameed M.S.	IT19064932	
Jayasinghe D.T.	IT19075754	
Sakalasooriya S.A.H.A.	IT19051208	

The above candidates are carrying out research for the undergraduate dissertation under my supervision.

Name of the supervisor: Dr. Lakmini Abeywardhana

Signature of the supervisor: Date: 09/09/2022

Name of the co-supervisor: Ms. Dinuka Wijendra

Signature of the co-supervisor: Date: 09/09/2022

ABSTRACT

Natural language processing has become an essential part of modern conversational AIs. However, applying natural language processing to low-resource languages is challenging due to their lack of digital presence. Sinhala is the native language of approximately nineteen million people in Sri Lanka and is one of many low-resource languages. Moreover, the increase in using “code-switching”: alternating two or more languages within the same conversation, and “code-mixing”: the practice of representing words of a language using characters of another language, has become another major issue when processing natural languages. Apart from natural language processing, the explainability of opaque machine learning models utilized in conversational AIs has become another prominent concern. None of the existing modern conversational AI platforms supports explainability out of the box and relies on just a performance score such as accuracy or f1-score. This paper proposes a no-code conversational AI platform with a series of built-in novel natural language processing, model evaluation, and explainability tools to tackle the problems of processing Sinhala-English code-switching and code-mixing natural language data and model evaluation in modern conversational AI platforms.

Keywords: Conversational AI, Natural Language Processing, Code-switching, Code-mixing, Rasa, Explainable AI

ACKNOWLEDGEMENT

We express our deepest gratitude to our team supervisor Dr. Lakmini Abeywardhana, and co-supervisor, Ms. Dinuka Wijendra, for their support, guidance, and feedback. This endeavour would not have been possible without their knowledge and expertise. We also appreciate the Sri Lanka Institute of Information Technology for hosting an extensive collection of publicly accessible documents and student and course guides on their official websites, which we used to design the conversational AI dataset utilized during this research.

We are also grateful to the SLIIT research project team (CDAP) and lecturers who conducted invaluable research project (RP) module sessions for assisting in devising a well-prepared research timeline, document templates and guidelines. In addition, we should also mention the invaluable feedback given by the research progress evaluation panel throughout the research duration.

Finally, we are grateful to everyone whom we might not have mentioned by their name who assisted our research team in conducting the research at any stage.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	Description
AI	Artificial Intelligence
API	Application Programming Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DIET	Dual Intent Entity Transformer
DIME	Dual Interpretable Model-Agnostic Explanations
GDPR	General Data Protection Regulation
GFI	Global Feature Importance
GUI	Graphical User Interface
GPU	Graphics Processing Unit
IO	Input/Output
IT	Information Technology
LIME	Local Interpretable Model-Agnostic Explanations
LFC	Local Feature Contribution
NLP	Natural Language Processing
NLU	Natural Language Understanding
NLG	Natural Language Generation

PDP	Partial Dependency Plot
RAM	Random Access Memory
SLIIT	Sri Lanka Institute of Information Technology
SHAP	Shapley Additive Explanations
TPU	Tensor Processing Unit
UI	User Interface
XAI	Explainable Artificial Intelligence
CDD	Conversation Driven Development
SaaS	Software-as-a-service
CaaS	Conversational AI-as-a-Service

1. INTRODUCTION

1.1 Background & Literature survey

In this era of Artificial Intelligence (AI), many top-tier applications and platforms utilize Natural Language Processing (NLP). Almost all domains have adopted NLP in ample ways to perform Natural Language Understanding (NLU), Natural Language Generation (NLG), or both. One such popular application of NLP is chatbots. Its name has gradually become Conversational AI after chatbots have adopted AI to perform NLU and NLG instead of following rule-based or pattern-matching techniques. Most conversational AIs have advanced machine learning models that are constantly trained based on conversation-driven development (CDD): training on natural human conversation data, allowing them to handle complex queries and flawless conversations [1].

There are several kinds of capabilities in chatbots. Most important this of chatbot is it attracts the user and improve the impressions regarding the product or the service which chat bot was designed to. In addition, while the human likeness of chatbots for customer service may be somewhat relevant for user experience, it pales in significance when compared to the ability of such chatbots to address inquiries effectively and competently. Since chatbots are available all the time to provide customer services companies can reduce the expenditures up to some level by replacing call centers with chatbots. This increases the customer satisfaction as well because of the quick responses. Therefore, businesses are tent use chatbot as one of their main customer interface.

However, processing natural language queries can be challenging depending on the language of the training data fed to a conversational AI. For example, although the Sinhala language is one of the two official languages of Sri Lanka, with around 19 million speakers out of the total population of 21 million, it still falls under the category of low-resource languages due to its less digital presence [2], [3]. Conversational AIs will encounter difficulties processing such languages due to the limited NLP tools and limitations in acquiring training data. Most Sri Lankans prefer using the Sinhala-

English code-switching typing style, according to a recent survey conducted on 110 people from the public [4]. Processing Sinhala-English code-switched textual data becomes further challenging when compared to processing a low-resource language since there are even fewer resources and NLP tools available, which is one of the main concerns of this paper.

Having multiple equivalent words is one of the many issues in a code-switching corpus [5]. For example, a Sinhala-English code-switching corpus may contain the two similar queries යුනිවර්සිටි එකේ තියෙන උපාධි මොනවද? and University එකේ තියෙන Degrees මොනවද? (What are the degrees available at the University?). Here, the words උපාධි and Degrees essentially have the same underlying meaning, and the same applies to university and යුනිවර්සිටි. A conversational AI trained on an equivalent word-rich code-switching dataset will recognize these as distinct words and assign different features, which degrades the overall model performance. The same conversational AI will fail to understand a user query such as යුනිවර්සිටි එකේ තියෙන ඩිග්‍රීස් මොනවද? (What are the degrees available at the University?), as it does not know that ඩිග්‍රීස් is another equivalent word for Degrees and will discard that as an out-of-vocabulary (OOV) word in most cases.

Entity annotating is another significant task as it enables conversational AIs to recognize entities in user queries that helps the decision-making process and drives a conversation. Name entity annotation helps chatbot to identify name entities in user queries. As an example, when user ask, “how much engineering cost per semester” chat bot have to understand the degree type as the “engineering”. This is dome by name entity identification capability of chatbot. Without this capability chatbot will work as a form filling style instead of understanding the user query. As show in Figure 1.1. 1 most of the people like chat with chatbots with name automatic entity identification

instead of form filling style. To enable this capability chatbot developers need to annotate training data. This is very time-consuming task.

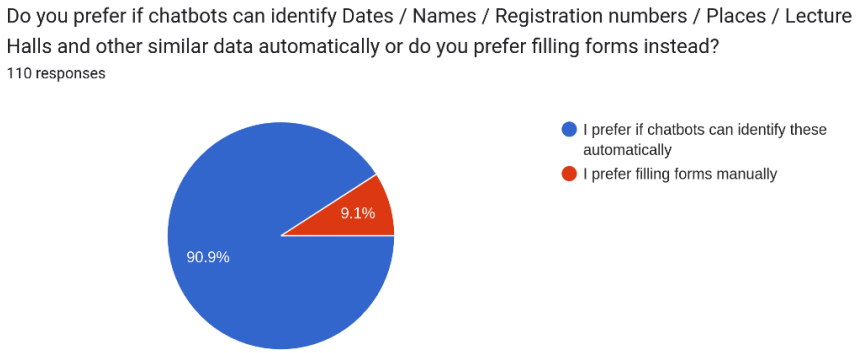


Figure 1.1. 1. Responses for do you prefer if chatbots can identify Dates / Names / Registration numbers / Places / Lecture Halls and other similar data automatically or do you prefer filling forms instead?

Currently available entity annotating tools included with most conversational AI development frameworks have evolved primarily around English language. For example, consider the Sinhala-English code-switched training data samples මේ පුටුවේ price එක කියද? (What is the price of this chair?), පුටුවක මිල කියද? (What is the price of a chair?), මේ පුටුව ගන්න කියක් යනවාද? (How much would it cost to purchase this chair?). There is no existing entity annotating tool that can identify පුටුවේ (of the chair), පුටුවක (of a chair), and පුටුව (the chair) is the same entity.

In the Sinhala language, it is possible to construct words by joining prefixes, root words, and suffixes in a meaningful manner, according to the නාම වරනැගිල්ල: Sinhala noun declension rules [6]. A combination of prefixes and base words can have multiple suffixes. For example, the Sinhala word පොත (book) can have multiple representations in terms of noun declension, such as පොත් (the books), පොතක් (a book), පොතට (to the book), පොතකට (to a book), පොතෙන් (from the book), and පොතකින් (from a book) which have the same base form පොත් combined with various suffixes. These combinations have almost the same meaning from the perspective of

entity annotation. Therefore, this paper proposes a method of annotating entities by assigning entity types to the base form of a specified word.

Although there are numerous conversational AI development frameworks and platforms, only a few of them support low-resource languages such as Sinhala. Many of these frameworks are cloud-based such as Google Dialogflow, Amazon Lex, and Microsoft Language Understanding Intelligent Service (LUIS), and have a graphical user interface (GUI) as the developer console. In addition to that, there are also local conversational AI development frameworks such as Rasa open-source with a command-line interface (CLI) [7]. None of the existing frameworks offers Sinhala typing support within the user interface (UI) or dedicated Sinhala data pre-processing tools essential for development tasks. Although the cloud-based conversational AI frameworks provide an easy-to-use GUI, developers cannot fully configure the process of training machine learning models. In local conversational-AI frameworks, the developers have the trade-off of being limited to a CLI that requires framework-specific expertise.

One of the major issues in supervised ML is that models do not generalize from seen data to unseen data, which is called overfitting [12-shamikh]. Due to overfitting, the model performs well on training data but performs poorly on testing data. This is due to the over-fitted model having difficulty in coping with the pieces of information in the testing data set, which may be different from the training data set. Also, over-fitted models are inclined to memorize all the data, including the unavoidable noise in the training data set, rather than learning the discipline behind the data set [13-shamikh].

Generally, the cause for overfitting is categorized into three kinds [13-shamikh]. The first is, noise learning on the training data set, this occurs either when the training data set is too small, has fewer representative data or else has too many noises. This creates a higher chance of the noises being learnt, which then acts as a basis for predictions [14-shamikh]. The second category is hypothesis complexity, which refers to the trade-off in complexity, which is a trade-off between variance and bias. It refers to the equilibrium between accuracy and consistency. When the algorithm has too

many inputs, the model becomes accurate on average with lower consistency [14-shamikh]. This means the model can be drastically different on different data sets. The third category is multiple comparison procedures that are ubiquitous in induction algorithms, and in other AI algorithms [15-shamikh]. During these processes, multiple items are compared based on the scores from an evaluation function and the item with the maximum score is selected. Nevertheless, this process is likely to choose items that will not improve, or even reduce the classification accuracy.

Underfitting is another issue that is faced when training a model, it occurs when the model is not able to accurately capture the relationship between the input and output variables, which produces a high error rate on both the training data set and the testing data set [16-shamikh]. This occurs when a model is too simple, which can be due to the model requiring more training time, less regularization, or more input features [16-shamikh]. Similar to overfitting, when underfitting occurs, a model is unable to establish the dominant trend within the data, which results in training errors and the model performs poorly. Low variance and high bias are good indicators of underfitting. Due to this behaviour being visible while using the training data set, it is easier to identify underfitted models than overfitted models.

Most users who are non-ML experts are unable to identify when a model either overfits or underfits, by looking at the learning curve. As shown in Figure 1.1.2, which depicts the data retrieved by the survey conducted, it is evident that the majority are unable to identify when a model overfits or underfits by looking at the learning curve. Most of these users are also unable to use methods like callbacks, early stopping and

checkpointing to help stop the model training at the right epoch which is not too high to cause overfitting or else too low to cause underfitting.

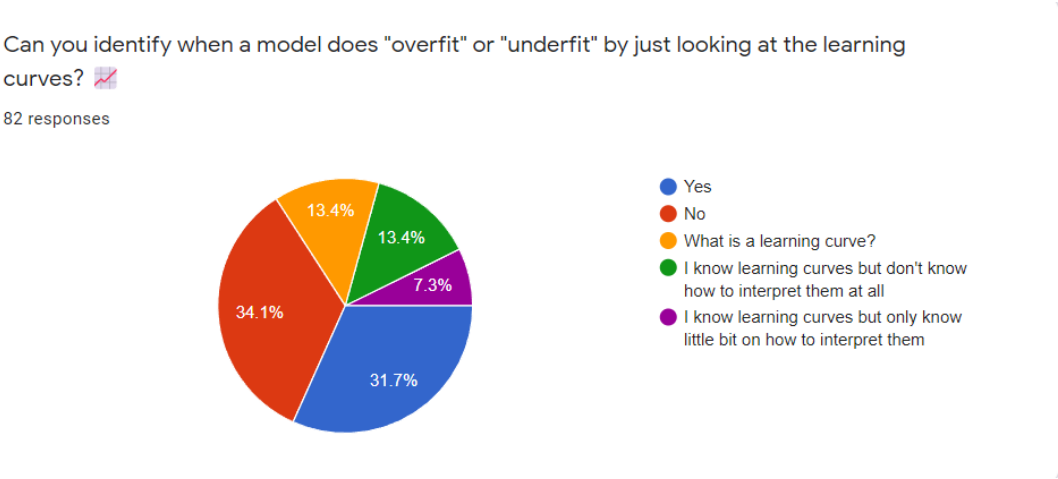


Figure 1.1.2. Summary of responses for whether people can identify whether a model is overfitting or underfitting by looking at the learning curve

Most conversational AI frameworks produce advanced machine learning models apt to the trade-off between model explainability and accuracy [8]. These models are not human-interpretable due to their complex and opaque nature and are known as black-box models. Explainable AI (XAI) allows debugging black-box models by explaining if the model looks at the accurate features [9]. In conversational AIs, developers can utilize XAI to debug machine learning models such as intent classification and entity recognition. The existing frameworks do not have built-in support for explainability.

This paper presents Kolloqe: a no-code conversational AI development platform. Kolloqe has built-in support for typing Sinhala-English code-switching text anywhere in the GUI, expanding the native language support beyond English. It offers Sinhala-English code-switching-specific NLP tools to map equivalent tokens and annotate entities efficiently through reverse-stemming and text similarity-based entity suggestions. The platform also provides the developers full support to configure the NLU pipeline and eliminates the need to interact with complex backends or CLI interfaces. Training and evaluating machine learning models of conversational AIs is supported within the GUI along with easy-to-configure NLU pipelines and model-

specific accuracy and loss curves. The platform also has out-of-the-box support for machine learning model explainability that explains any given query based on feature importance.

1.2 Research Gap

Although some research has explored Sinhala-English code-switched text processing up to an extent, the number of available resources is still considerably low. Reference [2] reveals adequate research papers written around Sinhala and Sinhala-English code-switched text processing and was a valuable resource. A few research papers have focused on developing Sinhala chatbots [3], [10]. Reference [3] discusses how word embedding models such as fastText can be used to increase accuracy and is closer to the objectives of this research. Reference [10] has developed a Sinhala chatbot, claiming it is the first Sinhala chatbot, and it has incorporated NLP components such as morphological analyzers, Sinhala parsers, and knowledge bases. However, it does not focus on code-switched or code-mixed text data processing. Reference [11] embodies training a word2vec model on the University of Colombo School of Computing (UCSC) Sinhala News dataset using the continuous bag of words (CBOW) method. However, the research paper only mentions elementary text preprocessing techniques such as stop word removal and lemmatization. Overall, none of the research papers mentioned above highlights text preprocessing methods for Sinhala-English code-switched or code-mixed textual data.

Reference [12], [13], and [14] have considered Sinhala-English code-switched and code-mixed text data processing. Although they have discussed word-level language detection, code-switching point detection, and pre-training machine learning models, they do not significantly accentuate proper text preprocessing techniques for code-switched textual data. Reference [12] has identified code-switching as a challenge in modern Sinhala text preprocessing, and it has introduced a dictionary mapping method to standardize the representation of Sinhala characters written using the English alphabet. This technique is somewhat similar to the method proposed by this paper in implementing the code-switching keyboard interface. Although [3] and [11] focus on

training word embedding models for Sinhala, they do not consider Sinhala-English code-switching text processing. Although [3] mentions training fastText models for chatbots, it does not emphasize significant changes to the data preprocessing steps.

Only a few research papers are available on automated and semi-automated text annotation tools. The Automated Named Entity Annotation (ANEA) tool [15] inherits knowledge from Wiktionary (an online dictionary). If Wiktionary does not contain the required word, this tool cannot guess the word entity. Brat Rapid Annotation Tool (BRAT) [16] is another tool for auto-annotating entities that introduces the semantic class disambiguation algorithm. GATE Teamware [17] mainly focuses on collaborative annotation, which is not in the scope of the entity annotation method proposed in this paper. YEDDA [18] is another tool that provides an entity recommendation method during the text annotation process by the maximum matching algorithm, which is a text segmentation algorithm. None of these research papers suggests an optimized entity annotating approach or similar work for Sinhala-English bilingual text data.

Tool name	**Collaboration features**	**Name Entity recommendations**	**Sinhala word variation identification**
ANEA	no	yes *(via external knowledge source)*	no
BRAT	no	yes *(via semantic class disambiguation)*	no
GATE	yes	no	no
YEDDA	no	yes *(via maximum matching algorithm)*	no
SIENA *(This research component)*	no	yes	yes

As stated earlier, both the local server conversational AI development framework Rasa open-source and the cloud-based conversational AI development frameworks such as Google Dialogflow, Amazon Lex, and Microsoft LUIS have their advantages, as well as limitations. The advantages are that Rasa open-source has the ability to fully configure ML pipeline and policy components to train ML models for

conversational AIs, and the cloud-based frameworks provide a GUI to build the conversational AI without having to work with the backend. However, Rasa open-source requires working with the backend to configure the YAML files to configure the ML pipeline and policy components, as well as executing CLI commands to train the model, both of which require significant expertise. When it comes to cloud-based frameworks', the ability to configure the ML pipeline and policy components is limited, so much so that users do not get to decide on what pipeline and policy components to use, instead they are required to train the model with whatever the configurations the cloud-platform provides.

Analyzing a model for overfitting or underfitting requires expertise due to the steps that generally need to be followed **Error! Reference source not found.**, which is, splitting the data set into two, as training data and testing data, or else using a cross-validation scheme, where the data is repeatedly split into training and testing subsets **Error! Reference source not found.** The model is then trained and a function such as the learning_curve() function from scikit-learn can be used to plot the learning curve, which will depict two curves, one for the training data and the other for the testing data, which will be plotted showing the validation loss for each epoch **Error! Reference source not found.**For a non-ML expert, these curves would make no sense, since it requires a level of expertise to look at the curve and deduce whether the model is overfitting or underfitting, and what the correct number of epochs would be to stop training at.

Training a model with too many epochs may lead to overfitting, while too few can lead to underfitting. Methods like early stopping can be used to overcome this, where a large number of epochs can be given, and the model will stop training when the model’s performance stops improving against a holdout testing dataset. Overfitting can also be overcome using methods like model checkpoint callbacks. These can be done using an Application Programming Interface (API) like Keras **Error! Reference source not found.**, yet they require having to work with the backend and having considerable knowledge about the Keras API. Also generating learning curves requires using a software like TensorFlow in backend to generate the learning curves which is

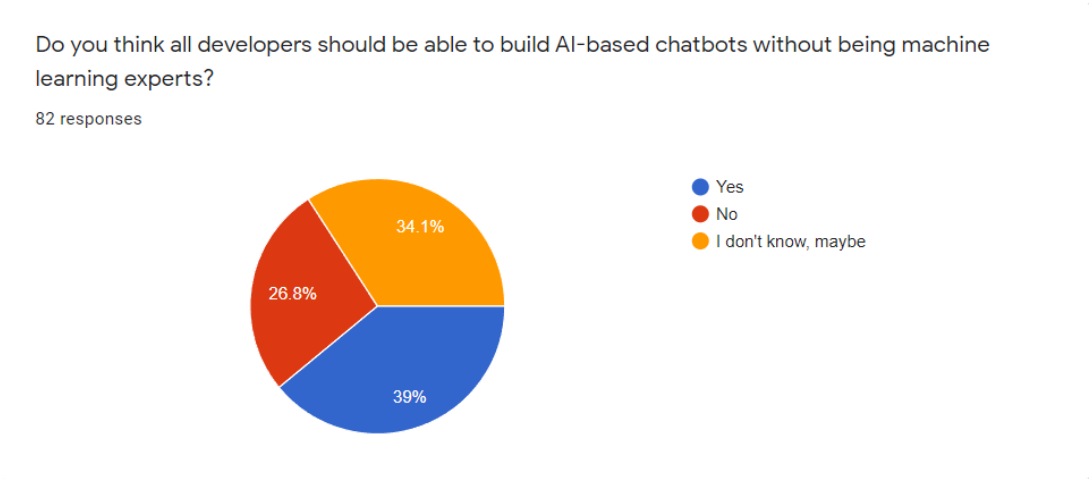


Figure 1.2.1. Summary of responses for whether people think that developers should be able to build conversational AIs without being ML experts

not an easy task. As shown in Figure 1.2.1, which depicts the data retrieved by the survey conducted, it is evident that the majority agree that developers should be able to build AI-based chatbots without having to be ML experts, which requires training models and evaluating them.

Recent XAI research that has introduced Local Interpretable Model-agnostic Explanations (LIME) [19], Shapley additive explanations (SHAP) [20] and explainable AI (XAI) libraries such as Explain Like I'm Five (ELI5) [21] have discussed various methods of implementing explaining for both NLP and non-NLP machine learning models. The LIME paper presents how to generate local explanations in a model-agnostic manner, taking a trained model and relevant set of classes as arguments to the LIME and training an explainable local surrogate model. SHAP paper discusses a slightly different approach while improving upon LIME, addressing its limitations. SHAP can calculate local and global feature importance inspired by shapely values found in game theory [22]. However, SHAP calculates the local feature contributions first and then finds the feature importance globally by summing the absolute SHAP values of each prediction [20]. The python XAI library ELI5 calculates local explanations using LIME. ELI5 also provides global model explanations based on the permutation feature importance technique by replacing words (also known as features in machine learning terminology) with random words (noise), which is closely

related to one of the XAI approaches introduced in this paper. Although some researchers have attempted to calculate the global feature importance based on individual local feature importance scores, none of the referred research has mentioned deriving local feature importance using global-level feature importance, which is the XAI approach introduced the in this paper.

Reference [7] presents a pair of open-source python libraries, Rasa Core and Rasa NLU, specializing in building conversational AI software. The core objective of Rasa is to make machine-learning-based dialogue management and NLU accessible to the average developers. However, Rasa does not have a GUI and is limited to a CLI and yet another markup language (YAML) file-based conversational AI development. Although the objectives of Rasa align with this research, it does not include a built-in GUI-based developer console, Sinhala typing support, Sinhala NLP tools, or machine learning model explainability.

Table 1.1: Comparison of XAI research done that applies to the domain of NLP and methodology used in contrast to this research.

Research/ Platform Name	**Can be utilized for NLP**	**Intrinsic/ Post hoc**	**XAI Scope (Local/ Global)**	**Model-specificity**	**Visualization Technique**	**Feature contribution calculation**
Rasa Open Source	Yes	Post hoc	Local	Agnostic	Highlighted text + Raw contribution value plots	Cosine Distance + Local linear surrogate model + Ridge Regression
Google Dialogflow	Yes	Post hoc	Local or Global	Agnostic	Highlighted text + Raw SHAP value plots	LIME, DeepLIFT, and other approaches with SHAP values.
Amazon Lex	Yes (Deep SHAP)	Post hoc	Local	Specific	Refer DeepSHAP [10].	Back-propagating contributions
Microsoft LUIS	Yes	Self-explaining	Local	Specific	Highlighted features	Regularization with explanation specific losses
Kolloqe (*Platform introduced in this research*)	Yes	Post hoc	**Local** with aid of **Global** feature importance	Agnostic	Highlighted features + Raw contribution score plots	**Global feature importance** + Shapley Values based on **model confidence**

2. RESEARCH PROBLEM

At present most businesses have integrated conversational AIs to their websites due to its convenience to customers, which allows the customers to receive the help they need faster, than having to browse the website or contact the business to retrieve the necessary information. According to a survey, 64% of people say they would rather message a business than call it **Error! Reference source not found.**The main reason people prefer messaging over calling is due to the number of times the calls generally do not go through, which can be frustrating to a customer 0

NLP (Natural Language Processing) tools and models for processing Sinhala and Sinhala-English code-switched textual data and feature engineering are considerably low. Developing deep learning-based models for NLP such as word embeddings, language models, and named entity recognizers from scratch requires a large amount of language-specific training data which can be rare for low resource languages. Although widely used and high resource languages such as English have many NLP tools or are in the realm of the NLP research interests, languages such as Sinhala have a considerably low research interest. In [4-dush], it is demonstrated that there is a lot of focus on Sinhala-based research and datasets, however, it also explains that keeping the research finding locked away from public access is one of the key reasons why Sinhala is still considered as a low resource language.

Since there are more bilingual and multilingual speakers today, many researchers focus on building bilingual and multilingual machine learning models that can handle more than a single language. Building multilingual models that can handle hundreds of languages to address modern issues such as code-mixing and code-switching may not be suitable for attaching to a conversational AI since those models can be heavy in size and require a lot of training data not in one language but many.

Generally, users use a physical keyboard that has an English-specific layout when interacting with the computing devices. If they need to type in Sinhala or else code-switch between English and Sinhala, they will have to use a service like Helakuru to type in the Sinhala words they want and then copy and paste it into the conversational AI, which is a cumbersome task, and most users would not even bother to go through the trouble for it. Due to that reason, having a keyboard interface that can handle code-switching and code-mixing is a must. More precisely, the keyboard interface must be attachable and support most web applications.

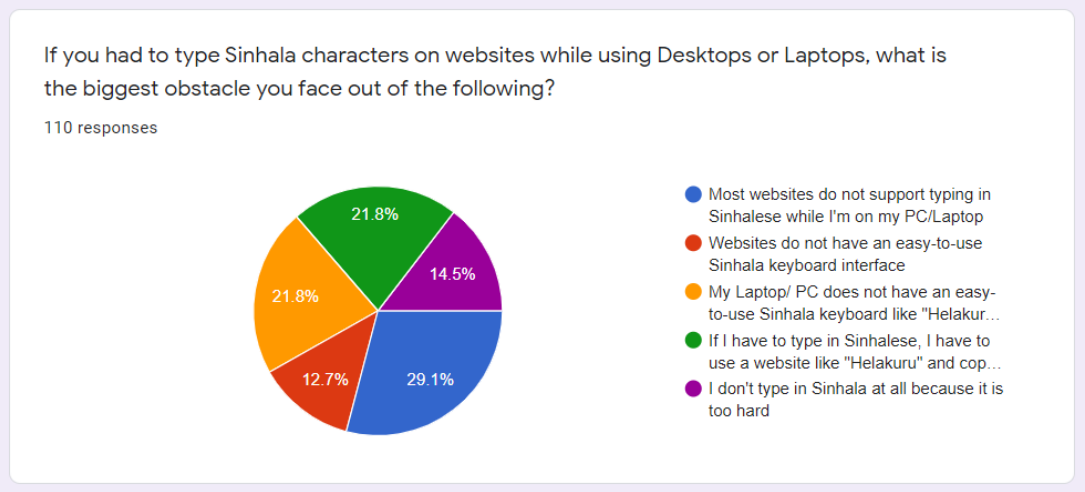


Figure 2.1. Summary of survey responses received for the question related to typing issues that exist for Sinhala in various devices

With the advancement of technology, machine learning models are becoming more complex and denser, because of that the amount of data they needed to train is increasing, therefore a large amount of text data is required to prepare for the model training. As an example, spacy name entity recognition AI model is trained by 741 MBs of text data. [12-akalanka] NCBI-Disease, BC5CDR and DFKI are popular datasets that used to train name entity recognition models which has large number of word count. Training chatbot components also required a large corpus because of that annotating a such large data set is a very tedious task. As well as when the dataset or the corpus contains enormous number of words, it takes a considerable amount of time to do the tagging process because every word in the corpus should be read and identified by the person who is doing the annotation task. Therefore, annotating data

sets for named entity recognition models is a very time-consuming task. Name entity tagging requires domain knowledge as well because the person who does the annotation needs to read the whole text and understand the meaning, therefore it requires expert knowledge regarding the domain. Finding people with domain knowledge is also a challenging task. Also, there is a chance they are not able to do the text annotation due to their lack of technical knowledge.

In this research, only Sri Lankan people are focused, therefore most Sri Lankans are using the Sinhala-English code-switching language style when they interact with a chatbot. This statement is proven by the survey. [13-akalanka] Sinhalese language, also known as Sinhala (සිංහල) is one of the two official languages of Sri Lanka, with about 16 million speakers out of the total population of 21 million also Sinhala is not a worldwide spread language like English. Due to those reasons, there is a lack of NLP tools and Sinhala specified name entity tagging tools designed for the Sinhala language. Therefore, Sinhala English mixed data annotation is even time consuming due to the absence of Sinhala English code-mixed annotation tools and technologies.

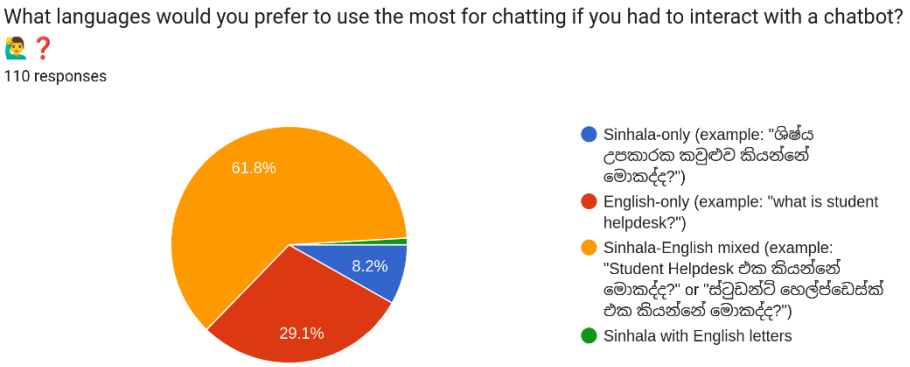


Figure 2.2. Responses for What languages would you prefer to use the most for chatting if you had to interact with a chatbot?

The methods used to develop conversational AIs have changed over the years. At present, most developers use frameworks to build Conversational AIs. There are a few

frameworks that are widely used at present. One of them is Rasa open-source, which is a local server-based framework that provides developers with the ability to build conversational AIs by configuring YAML files and then executing CLI commands to train the model required for the conversational AI. The other frameworks used are Google Dialogflow, Amazon Lex, and Microsoft LUIS, which are all cloud-based frameworks and come with a GUI, which simplifies the conversational AI model training process by eliminating the need to work with the backend. Both these types of frameworks have their limitations. Rasa open source requires the developer to know how to configure the YAML file and use the relevant CLI commands to train the model, which requires expertise. On the other hand, even though the cloud-based frameworks eliminate the need to work with the backend, thus reducing the required expertise, they do not provide the ability to fully configure the ML pipeline and policy components required to train a model the way Rasa open-source does. Instead, the developers are expected to train the model with whichever configurations are suggested by the cloud platform. Putting the strengths of both these frameworks together was the suggested solution, which combines Rasa open-sources' ability to fully configure ML pipeline and policy components along with the cloud-based platforms' GUI, which eliminates the need to work with the backend and simplifies the configuring and training tasks.

Once a conversational AI model is trained, it is important to evaluate the model to ensure it will perform well. Model evaluation is generally done in the backend by the developers who build the conversational AI. It is done by generating accuracy and loss curves and then evaluating them to check whether the model is performing well or whether it is overfitting or underfitting. The process of generating validation curves and evaluating them requires expertise. Hence, the developers of an organization cannot be expected to know how it is to be done. Since there is a need to provide the developers of organizations with a GUI to train models, it is also important to include a feature to generate and evaluate the models using validation curves without having to know the technical know-how and suggest what the best epoch is to train a model.

Integrating a conversational AI into a business's website is not a one-time thing. To cater to customers without issues, the conversational AI must be constantly maintained and improved 0It can become a tedious task having to constantly contact the conversational AI service provider every time there is a need to improve the conversational AI. Hence, the proposed solution was to build a GUI that makes it possible to maintain, monitor, and improve the conversational AI without requiring any ML expertise.

As previously explained, algorithms and machine learning models are getting more complex frequently. Widely utilized applications such as chatbots have employed machine learning and NLP due to high performance and accuracy compared to traditional approaches. These complex algorithms and advanced machine learning techniques, such as deep learning, are opaque by design, and humans fail to comprehend how these models work internally to make decisions [11-ishara]. Most currently available local interpretable XAI approaches are not globally faithful. Reference [1-ishara] states that the authors have only found four research papers in the global explanation category. Many text explainers in the XAI domain focus on generating local model explanations, and only a few explainers support generating global model explanations by aggregating local model explanations. However, the accuracy of the aggregated model explanations is often distrustful since the global faithfulness of the individual local model explanations is not guaranteed. None of the referred studies has considered generating model explanations that are locally and globally faithful, mainly due to global explanation calculation being computationally intensive [1-ishara]. This research component concentrates on finding the local model explanations based on globally important features and enhancing the efficiency of the GFI calculation, making the explanations generated by the introduced XAI approach globally and locally faithful.

Another conspicuous issue is that input perturbation-based XAI techniques are prone to adversarial attacks. Widely utilized XAI text explainers, including LIME and SHAP, employ input perturbation for sampling, which makes the explanations generated by these methods unreliable [31-ishara]. The inconsistent and manipulatable

nature of these XAI techniques poses a direct threat to the reliability of the model explanations. This research component concentrates on eliminating the input perturbation layer to make the delivered model explanations more reliable and immune to adversarial attacks.

It is arguably easy to integrate an existing XAI technique with ML models built from scratch using well-known machine learning frameworks such as TensorFlow([10-ishara]), PyTorch[11-ishara]), or Scikit-learn[12-ishara]) because the developers are fully aware of the ML model architecture, and the ML frameworks provide a comprehensive API to interact with the models. i.e., The developers have the required tools and documentation to enable XAI in models built with well-known ML frameworks from scratch. Most XAI libraries provide built-in support for models built with well-known ML frameworks and are well-documented. For example, LIME has built-in support for Scikit-learn models, and DeepLIFT implementation is fully compatible with TensorFlow and Keras. However, although many NLP-based XAI explainers exist, they cannot generate explanations for all ML model types. It is often impossible to find model explanations for application-specific ML models without comprehensive technical and ML knowledge, even with model-agnostic explainers such as LIME [15-ishara] and SHAP [16-ishara]. The ML model inside the chatbot framework Rasa is an example of such a model. The models trained and stored within Rasa contain a cluster of machine learning models, NLP pipeline components, and metadata. Rasa NLU is responsible for extracting the model, loading the pipeline components, and processing textual data. Generating model explanations for intent classifiers such as the Dual Intent Entity Transformer (DIET) classifier is tedious since Rasa does not provide a proper toolset or an API to interact with specific ML models utilized in the NLU pipeline. Extracting prediction probabilities from DIET on demand is also not directly supported, which is required by many model-agnostic text explainers [2-ishara], [3-ishara], [15-ishara]. However, it is more acceptable if there is a way to interpret these application-specific ML models by chatbot developers to trust the predictions given by the models employed within these widely utilized frameworks instead of having to manually attach an XAI library or only rely on performance scores that do not evaluate the reliability of the model decisions at all.

Moreover, a recent survey that studied the current trends in ML model interpretability and explainability confirms that the majority prefers explanations for the machine learning model predictions. The population of interest in the survey was 110 Undergraduates of the Faculty of Computing of Sri Lanka Institute of Information Technology, selected based on the technical nature of the survey questions. Refer to Appendix A: Survey Form to observe the complete survey questions and responses. Responses in Figure 2.3 clearly show that more than three-fourths of the population has little to no understanding of the NLU techniques used within chatbot frameworks and the intent classification process. Although 32.7% of the population has claimed that they are aware of the terms model interpretability and explainability (Figure 2.4), Figure 2.5 clearly illustrates that only 26.4% have used text explainer tools such as LIME and SHAP. 66.4% of the total population is unaware of model explainability and interpretability.

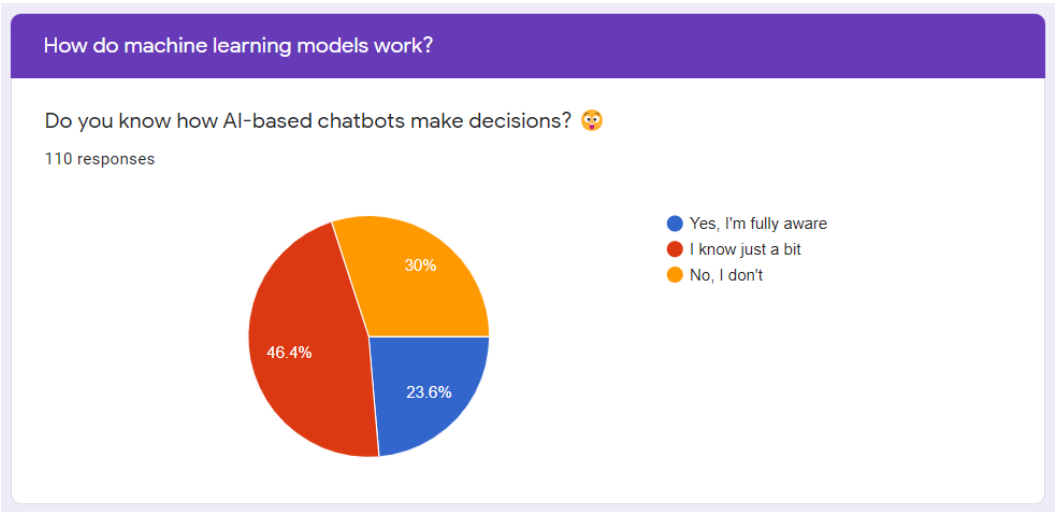


Figure 2.3. Summary of survey responses received for the question "Do you know how AI-based chatbots make decisions?"

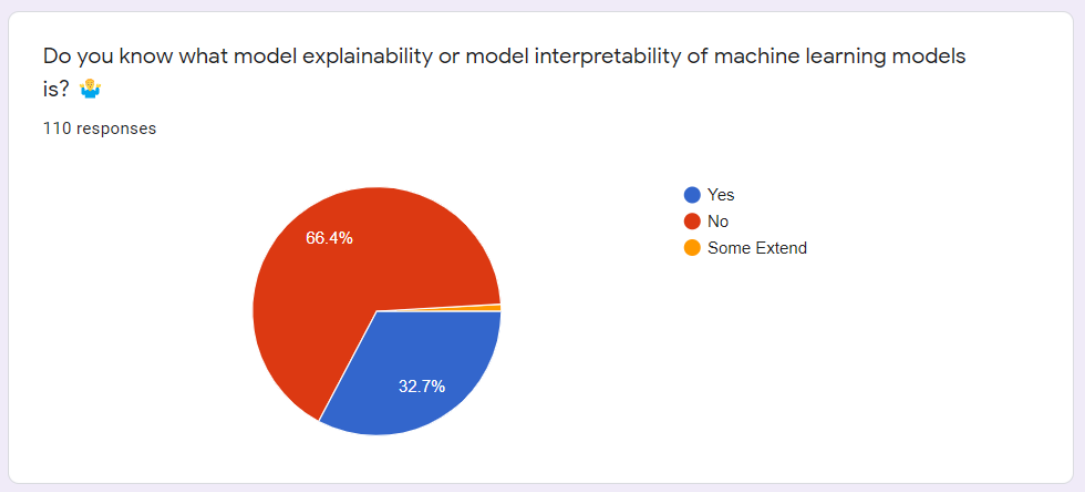


Figure 2.4. Summary of the responses received for the survey question "Do you know what model explainability or model interpretability of machine learning models is?"

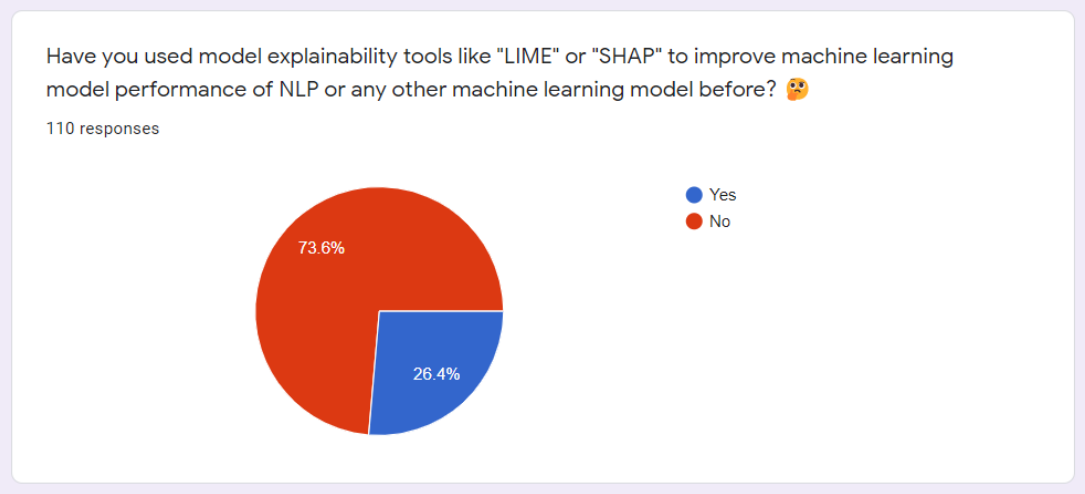


Figure 2.5. Summary of the responses received for the survey question "Have you used model explainability tools”

3. RESEARCH OBJECTIVES

3.1 Main Objectives

The main objective is to develop NLP tools for text preprocessing, feature engineering, training data and machine learning model improvements, model performance evaluation, and model explanation generation on DIET intent classifiers; that can be attached to a domain-specific Rasa conversational AI assistant designed for Sinhala-English code-switched text corpus.

3.2 Specific Objectives

Specific objectives of this research can be listed down as follows.

1. Develop a strategy to suggest entities as a list, based on the descending order of similarity score when a user selects a phrase.

2. Develop a strategy to annotate all variations of phrases at once by selecting a base word of a phrase.

3. Develop a strategy to export the annotated data in Rasa NLU data format

4. Develop a visualization technique to provide user friendly entity suggestions.

5. Develop a novel GUI with the ability to configure ML pipeline and policy configurations using just the GUI and train the machine learning model with just the click of a button while hiding the backend complexity from the user.

6. Develop an algorithm to suggest the best epoch of a trained model by evaluating the loss curve of the model.

7. Develop a GUI to view all the trained machine learning models along with their train and test accuracies and with the ability to delete and download those

models and view the learning curves of these models along with the algorithm to suggest the best epoch integrated into it.

8. Develop a strategy to calculate the global feature importance for individual features and normalize the scores in a logically explainable manner.

9. Develop a strategy to incorporate global feature importance to derive local model-agnostic explanations for the DIET classifier in Rasa framework.

10. Develop a visualization technique to illustrate the local text explanations with the contribution scores towards the prediction in a human-interpretable way.

11. Build a Server with API endpoints that can be consumed by the conversational AI maintenance frontend.

12. Integrate the XAI approach developed with the Rasa framework seamlessly and allow Rasa users to get model explanations for the DIET classifier.

4. METHODOLOGY

4.1 Data Collection

The overall research required two datasets in total. The datasets are domain-specific and comprise of Sinhala-English code-switched text data. There is a notable difference between the two datasets. The differences between the two datasets are clearly explained in subsections 4.1.1 and 4.1.2. To overcome the low resource nature of the gathered Sinhala text, data augmentation techniques were used such as capturing as many code-switched phrases and distinct writing patters as possible to reduce the bias in the dataset. All four research group members generated four versions of the samples for both the datasets using different code-switching styles. Any duplicate data in the datasets were removed to ensure the quality of the data collected.

4.1.1 General Dataset for Machine Learning Model Training

The general dataset contains Sinhala-English code-switched textual data scraped from websites related to SLIIT[13-Ishara] and news articles[14-Ishara]. In addition to that, the dataset also comprises of publicly available documents, such as Microsoft Word and Portable Document Format documents, taken from the same sources since they are both public and official. Data augmentation techniques were utilized overcome the low-resource nature of the collected text corpus. The overall dataset comprises of 720 paragraphs after going through the data augmentation techniques and cleaning, and it is available as a public GitHub repository where it can be cloned and utilized by interested parties for future research and other academic related work[15-Ishara].

4.1.2 Domain-specific dataset for Conversational AI Training

The domain specific dataset used to train the Rasa-based conversational AI contains handcrafted Sinhala-English code-switched textual data based on the first dataset explained in section 4.1.1. The dataset has been designed carefully as a training dataset for the intent classification task of conversational AIs according to the guidelines provided by Rasa[16-Ishara]. As in the first dataset, data augmentation techniques were utilized for this dataset as well to overcome the low-resource issue and capture various sentence patterns. Originally there were around seventy-eight intents (78 classes) and one thousand and seven hundred examples (1700 data points), and each class contained a minimum of ten question examples as a standard. The original chatbot training dataset was refined and a new training dataset comprising of seventy-one intents (71 classes) and six hundred and thirty-five examples (635 data points) were created. Note that chatbot developers can adjust the number of questions per intent to increase the overall performance of the conversational AI at any moment. This chatbot training dataset resides in the same public GitHub repository mentioned in section 4.1.1, where interested parties can clone and utilize it for future research, domain-specific chatbot training, or other related academic work.

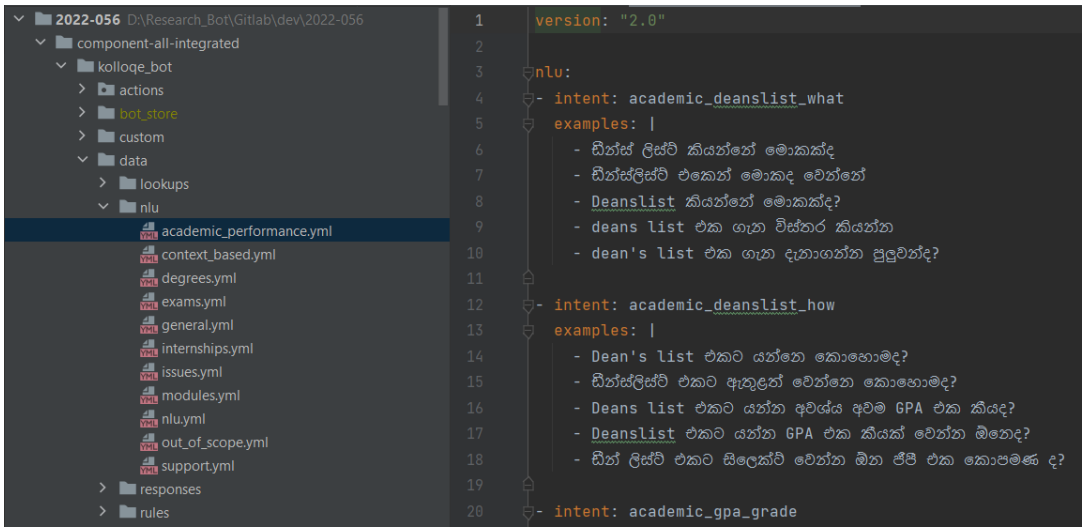


Figure 4.1.2.1. A part of the prepared dataset

4.2 Functional and Non-Functional Requirements

The requirements gathering phase mainly focused on studying existing research and tools on chatbot development and evaluation. After identifying the current requirement for a chatbot development and evaluation tool following functional and not functional requirements are defined.

4.2.1 Functional requirements

The functional requirements of Kolloqe are as follows.

1. Kolloqe should provide named entity suggestions and auto annotation for NLU text data annotation.

2. Kolloqe should be able to let users configure ML pipeline and policy components and train a model using only the UI without the user requiring any knowledge on dependencies required by each component.

3. Kolloqe should be able to let users view all the trained models along with their training and testing accuracies.

4. Kolloqe should be able to suggest the best epoch of a given model when the user provides the patience interval value.

5. Kolloqe should generate explanations for non-technical users.

6. Kolloqe should be able to visualize the generated model explanations in a human-interpretable manner.

7. Kolloqe should be able to export the generated model explanations which should be easily sharable and visualizable.

4.2.2 Non-functional requirements

The non-functional requirements of Kolloqe are as follows.

1. Kolloqe should perform calculations efficiently.
2. Kolloqe should provide reliable and consistent results and outputs.
3. Kolloqe should be easy to maintain.

4.3 Overall Architecture

Figure 4.3.1 depicts the high-level architectural diagram of Kolloqe. As it can be seen from the diagram, two types of users will be interacting with Kolloqe, the first are the end users that will be interacting with the chatbot which comes with the Sinhala-English code-switchable keyboard integrated, the second are the developers that will be interacting with Kolloqe's development console. As it can be seen from the diagram, Kolloqe's features have been denoted by four main components all of which can be done using just the GUI without having to deal with the backend or any CLI commands. The *Entity Annotation* component denotes the feature where users can annotate a corpus which will be used as training data by the custom named entity recognition model in the Rasa NLU pipeline. The *Machine Learning Pipeline* component denotes the feature where users can configure the pipeline and policy components required for a model and then train the model. The *Intent Classification Model Explanations* component denotes the feature where users can input a textual data instance and request explanations which visualizes the importance scores as easily comprehensible bar charts and raw scores. The *Curve Insights* component denotes the

feature where users can view all the trained models with their training and testing accuracies as well the view the learning curves of the model along with the best epoch to use, to train that model, which is suggested by Kolloqe.

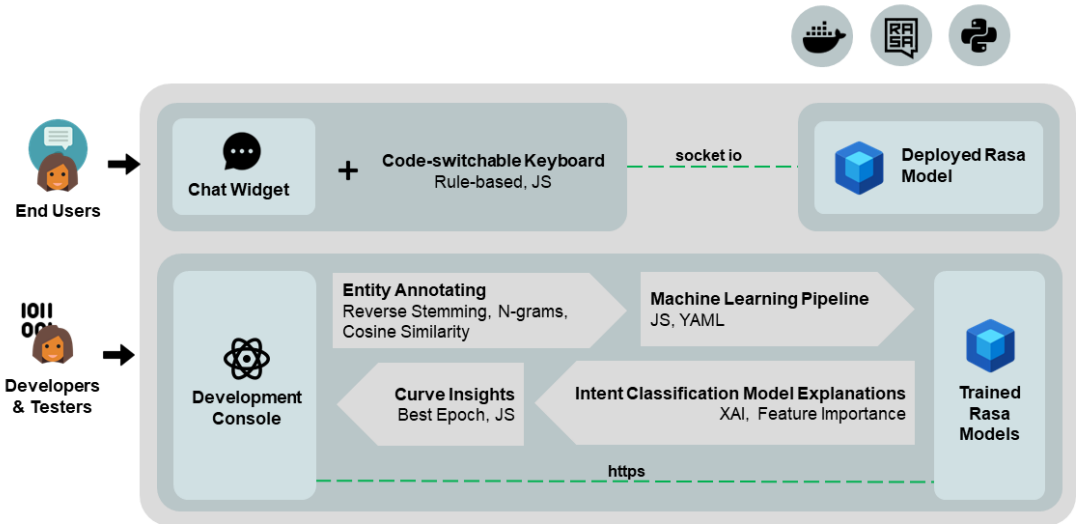


Figure 4.3.1. High-level architectural diagram of Kolloqe, the overall research

4.4 Custom Entity Annotation

Users can annotate their Sinhala English mixed text data efficiently by using Kolloqe UI's Annotations tab, SIENA. The next subsections 4.4.1, 4.4.2 and 4.4.3 provide a detailed explanations how this tool helps user to do the text annotation task easier by providing auto annotations and name entity suggestions.

4.4.1 Entity suggestions

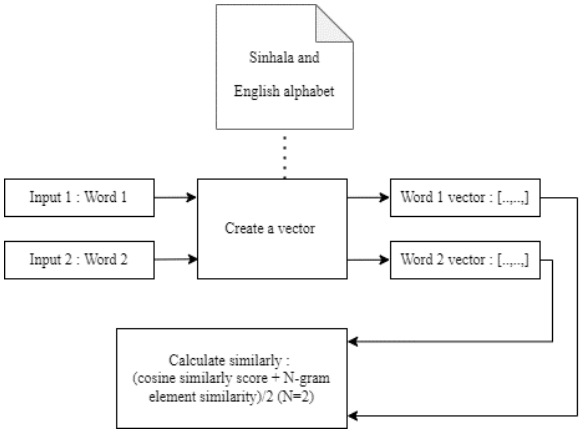


Figure 4.4.1. 1. Text similarity algorithm

SIENA provides name entity suggestions when user select phrase to annotate based on the previously annotated data. When user initially annotate a phrase by selecting a name entity, SIENA converts it into its base form and saves it inside the knowledge base with the selected name entity to provides suggestions. When user select phrase to annotate, SIENA provides list of name entities as suggestions. This is done by calculating similarity scores between base word of selected phrase and the saved base word entries of the knowledge base. Based on the calculated similarity scores, SIENA reorders name entity suggestions list according to the descending order of the calculated similarity scores.

Text similarity calculation is done by getting the average value of cosine similarity and the bi-gram similarity scores. To calculate cosine similarity both the words are converted into vectors by counting the letters of selected phrase against the Sinhala and English alphabet. SIENA English alphabet consists of 26 letters and SIENA Sinhala alphabet consists of 1001 letters therefore each vector has a length of 1027

letters. N-gram similarity calculated by break down phrases into letters and crate bi-gram then calculate the portion of similar elements over all other elements.

To derive the Sinhala base words SIENA uses stemming algorithm. Using this approach, the base form of a given word can be determined without considering

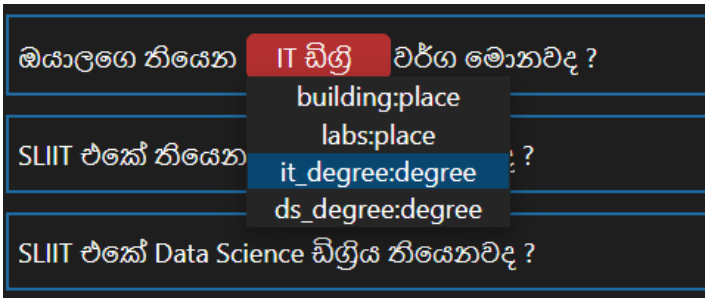


Figure 4.4.1. 3. Initial name entity suggestion list order

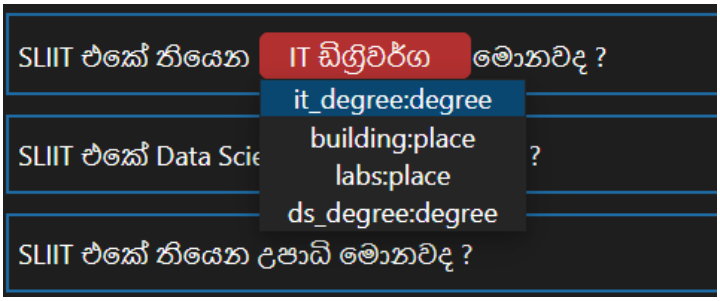


Figure 4.4.1. 2. Sorted name entity suggestion list

its morphological information. By combining prefixes, root words, and suffixes in a meaningful way in accordance with the Sinhala noun declension rules, words can be created in the Sinhala language. A combination of Sinhala prefixes and Sinhala base words can be combined with various suffixes. But unlike suffixes, prefixes drastically alter the meaning of the term. Therefore, by removing suffixes instead of prefixes, the rest of the words have almost same meaning. Then Sinhala words are converted into vowel and consonant formats to remove suffixes. The rules for Sinhala word conjugation make it simple to distinguish suffixes from a given word when it converted into vowel and consonant formats. The converted text was then

double-checked against a list of Sinhala suffixes that is also converted into vowels and consonants. Once the identification is complete, the suffix is eliminated. The final word is then combined with vowels and consonants to return to its original form.

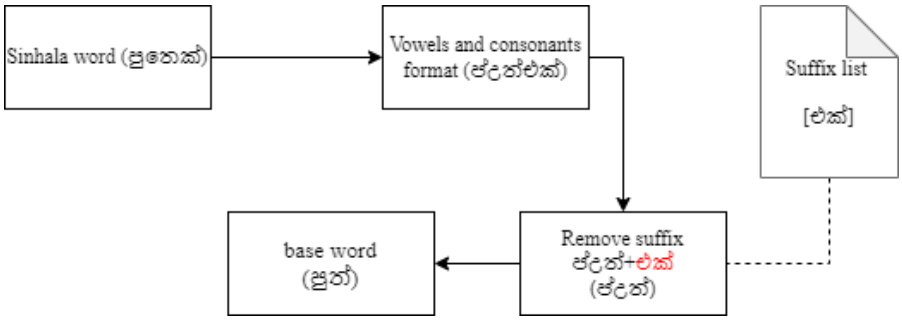


Figure 4.4.1. 4. Stemming algorithm

4.4.2 Auto annotation

Based on previously annotated data, this algorithm is used to carry out auto-annotation activities. This, as opposed to stemming algorithms, establishes a link between the base words and their original phrases. As a result, this method changes a phrase into base form and saves it in the knowledge base along with the name entity that the user provided when they first assigned a name entity to it. Each knowledge base entries consists of usage count as well. SIENA changes a term to its base form when a user selects it for the annotation, then looks it up in the knowledge base. The algorithm will apply name entity of user selected phrases to all the variants of a base word. To do the auto annotation SIENA counts the words inside the user-selected phrase and groups the words by keeping the order of the sentences while ensuring that each group has the same number of words to determine the base word variants. Then, SIENA iterates over each group, transforms it to base form, and determines whether it matches the base form which is chosen by the user. If they match, SIENA will assign

the name entity of the chosen base word to that group and reconstruct the sentence using the word groups.

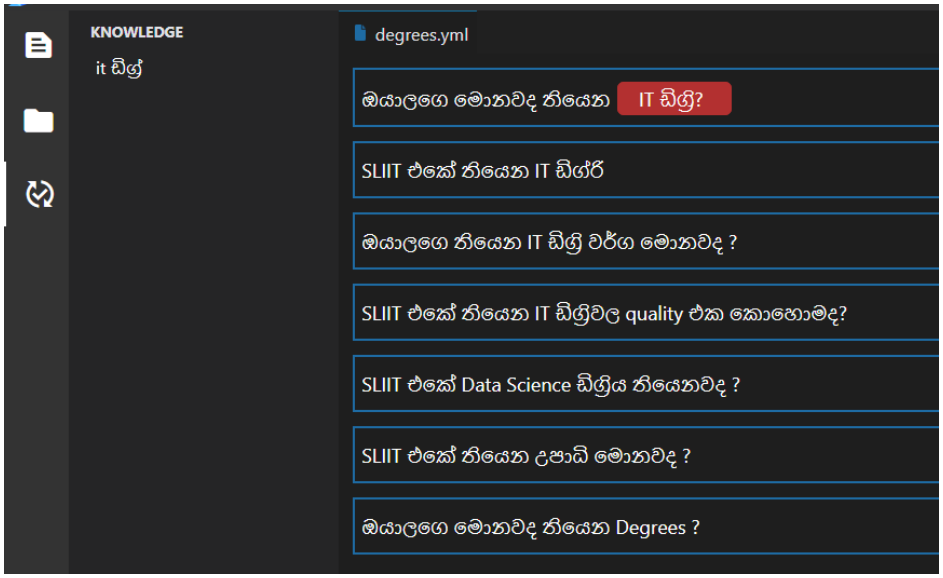


Figure 4.4.2. 2. Before auto annotation

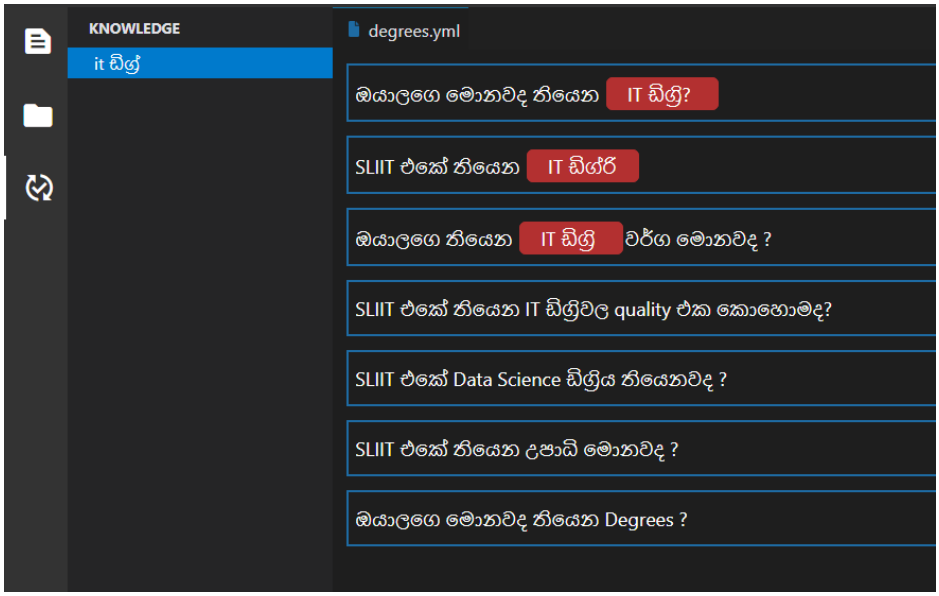


Figure 4.4.2. 1. After auto annotation

4.4.3 Knowledgebase

This component is used to share the gathered knowledge among other instances of SIENA. This includes information related to providing name entity suggestions and name entity recommendations. Therefore, users can reduce the time take to annotate by importing previous knowledge base which is used to tag same kind of text data.

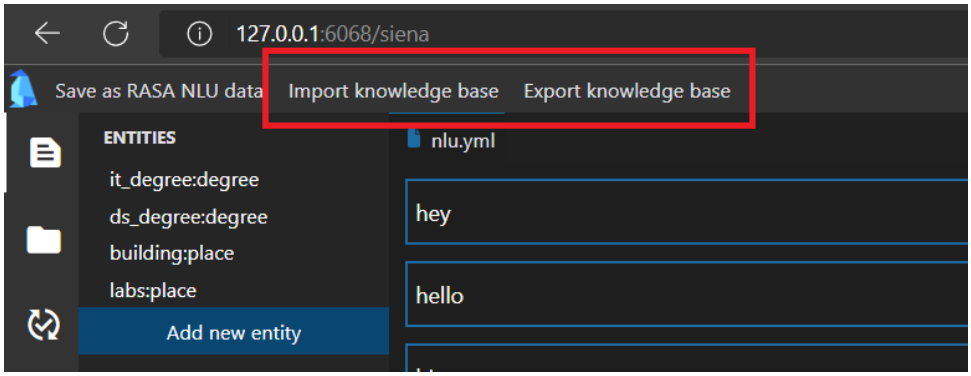


Figure 4.4.3. 1 Import or export knowledge base

4.5 Equivalent Token Mapping

4.5.1 Defining token maps

4.5.2 Real-time token mapping

4.6 NLU Pipeline and Policy Configuration

In the Kolloqe UI, under the *Configurations* tab users can configure and train conversational AI machine learning models. The process behind configuring and training a model will be explained in detail in the subsections 4.6.1 and 4.6.2 below.

4.6.1 Training a model

Initially, after clicking the *Configurations* tab, the users are directed to the UI seen in Figure 4.6.1.1**,** where none of the pipeline or policy configurations has been selected. In case the users have selected a few components and need to unselect them all to go to the default view, they have two options. One of them is choosing *Custom Settings* from the dropdown denoted by *Existing Models* which can be seen in Figure 4.6.1.2, and the other is clicking the red *Reset* button which can be seen at the bottom right corner of the UI.

To train a model, users need to configure the pipeline and policy components required by ticking the checkboxes of the components that need to be included and then changing the values of the parameters of each component as required using the provided dropdown menus and textboxes. Users can refer to the Kolloqe documentation by clicking on *kolloqe docs* which can be seen at the start of both the pipeline and policy components sections, which would redirect the user to the Kolloqe documentation which contain details regarding the pipeline and policy components respectively, Figure 4.6.1.3 shows the pipeline components section of the Kolloqe documentation. Users have been provided with dropdown menus to set the values of most parameters, to reduce the number of possible errors that can be made, which

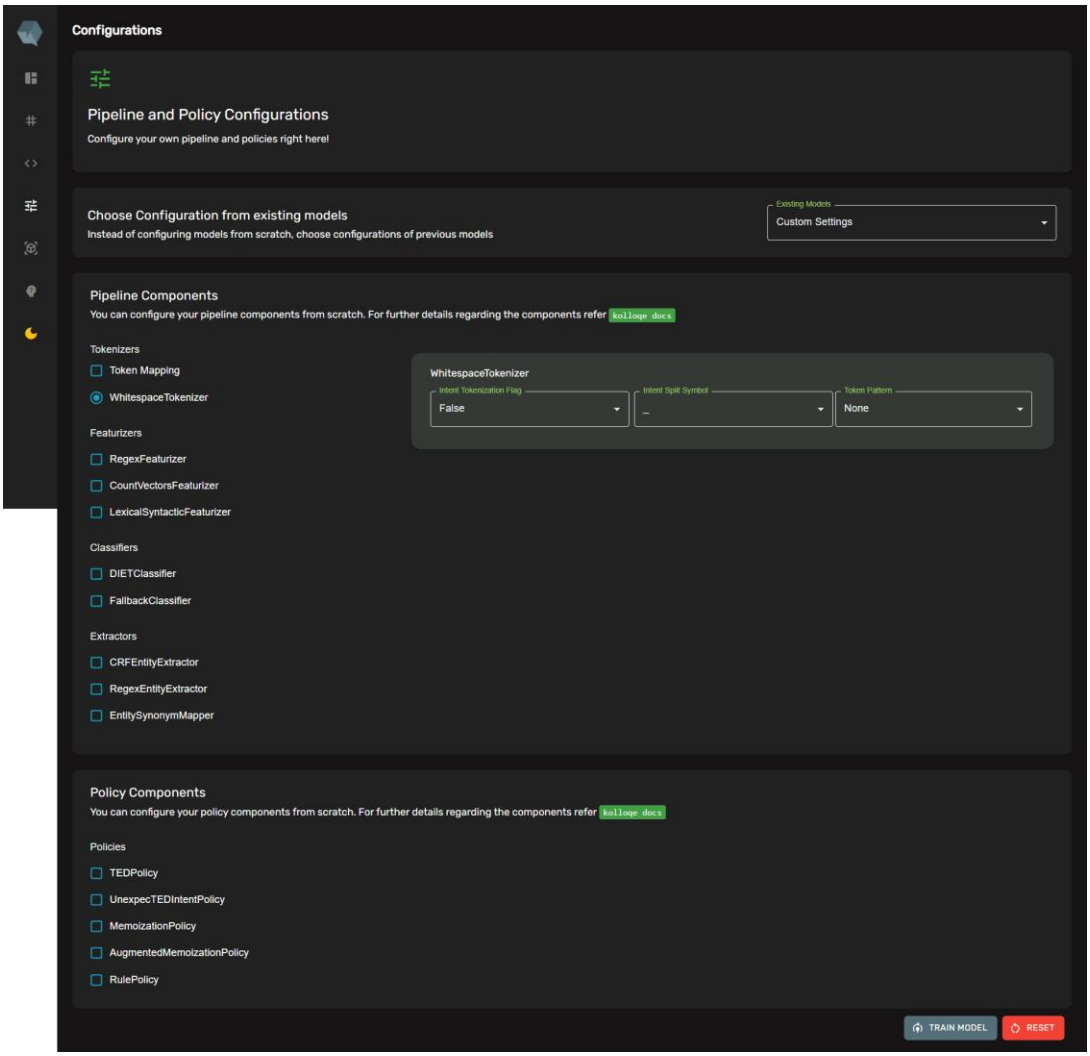


Figure 4.6.1.1. The *Configurations* UI – default view

would be high if users were expected to type in the value of each parameter into a textbox where the user would be able to enter anything. All the textboxes have been configured to validate inputs in real-time to make sure the values entered by the user are valid, if not to denote the user regarding the invalid input. Once all the required components are selected and all the values are set, the user needs to click the *Train Model* button to start the model training. When the *Train Model* button is clicked, the system validates all the input fields and if there are no invalid inputs training proceeds where the configurations are sent to the backend through the API call where the backend configuration file is overwritten as seen in Figure 4.6.1.5 and all the required CLI commands to train the model are invoked using bash commands as seen in Figure 4.6.1.6. Once a model is successfully trained, a toast message appears indicating to the user that the model was trained successfully. Users can also configure the pipeline and policy components using the configurations of an existing model by clicking on the dropdown *Existing Models* and choosing the required model which automatically selects all the components and populates the parameters with the values used in the selected model which can be seen in Figure 4.6.1.4, this is useful in instances where the users' train models with the same components but with slight changes to the values of some of the parameters.

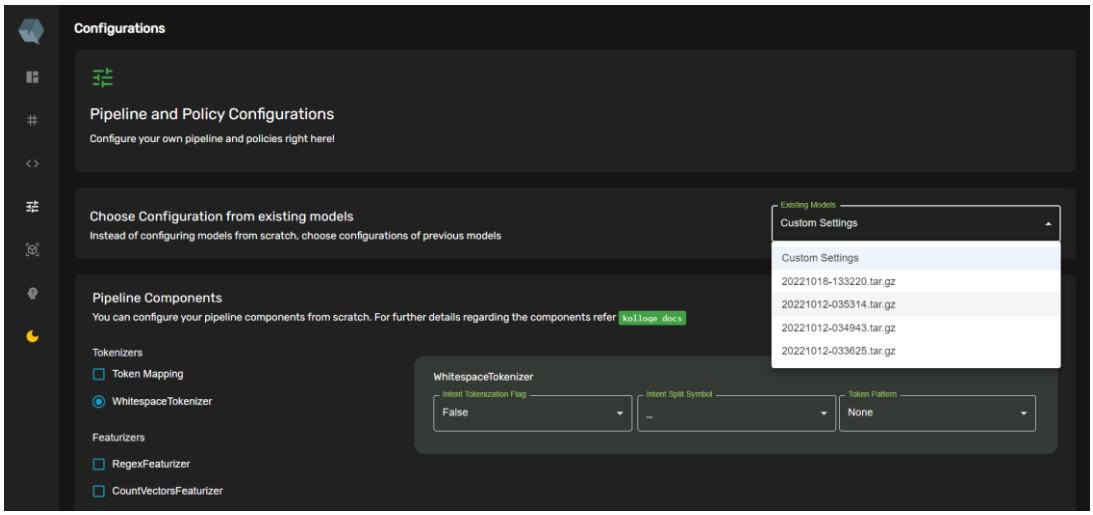


Figure 4.6.1.2. The *Existing Models* dropdown with the option to select *Custom Settings* or else an existing model

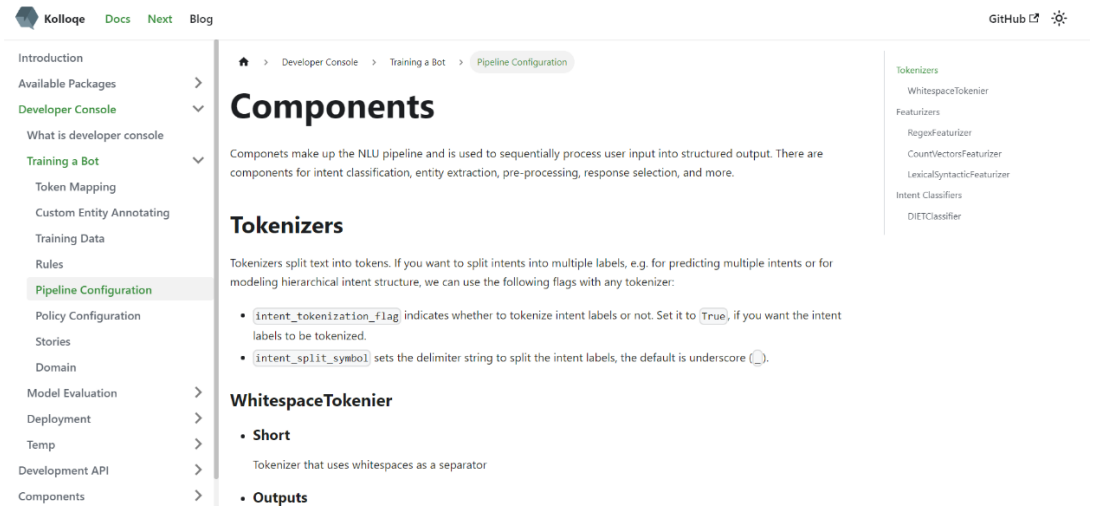


Figure 4.6.1.3. The pipeline configuration section of the Kolloqe documentation

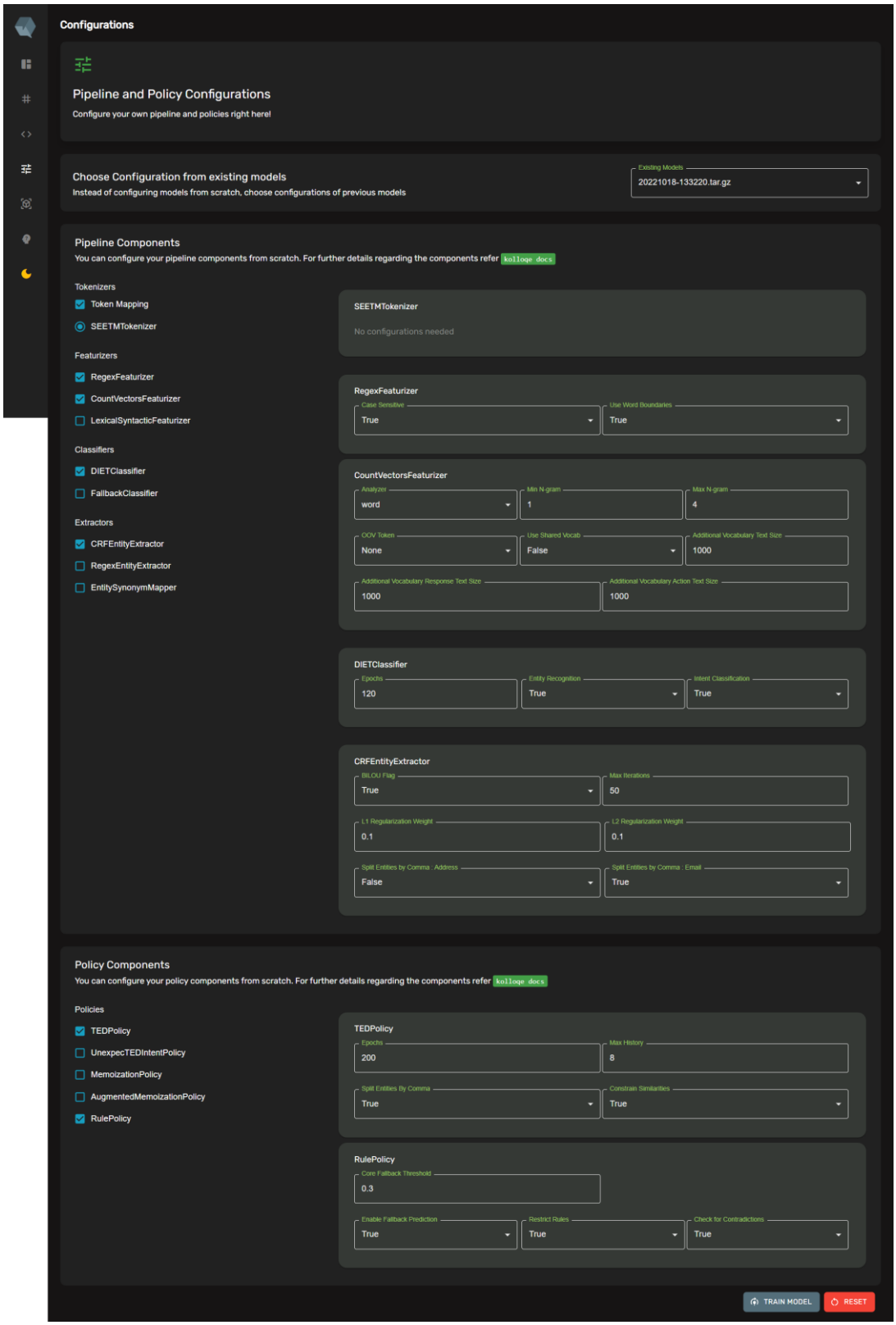


Figure 4.6.1.4. The UI when the configuration of an existing model is chosen from the dropdown menu

```yaml
language: en
pipeline:
  - name: WhitespaceTokenizer
    intent_tokenization_flag: false
    intent_split_symbol: _
    token_pattern: ' '
  - name: RegexFeaturizer
    case_sensitive: true
    use_word_boundaries: true
  - name: CountVectorsFeaturizer
    analyzer: word
    min_ngram: 1
    max_ngram: 1
    OOV_token: None
    use_shared_vocab: false
    additional_vocabulary_size:
      text: 1000
      response: 1000
      action_text: 1000
  - name: DIETClassifier
    epochs: 7
    entity_recognition: true
    intent_classification: true
    evaluate_on_number_of_examples: 150
    evaluate_every_number_of_epochs: 1
    tensorboard_log_directory: ./tensorboard
    tensorboard_log_level: epoch
    checkpoint_model: true
  - name: CRFEntityExtractor
    BILOU_flag: true
    features:
      -   - low
          - upper
          - title
      -   - low
          - upper
          - title
      -   - low
          - upper
          - title
    max_iterations: 50
    L1_c: 0.1
    L2_c: 0.1
    split_entities_by_comma:
      address: false
      email: true
policies:
  - name: TEDPolicy
    epochs: 8
    max_history: 8
    split_entities_by_comma: true
    constrain_similarities: true
  - name: RulePolicy
    core_fallback_threshold: 0.3
    enable_fallback_prediction: true
    restrict_rules: true
    check_for_contradictions: true
```

Figure 4.6.1.6. The backend YAML configuration file used to train the ML model for the conversational AI

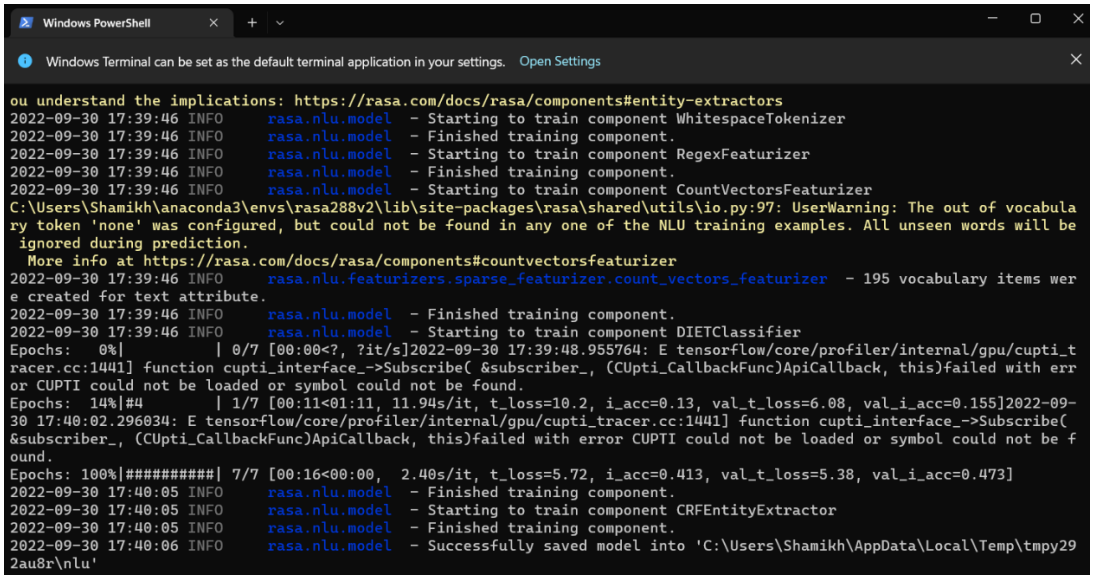


Figure 4.6.1.5. All the necessary CLI commands to train the model invoked using bash command calls to the backend

4.6.2 Training Queues

When a model is configured and put to train a UUID v4 is generated in the frontend to uniquely identify the frontend user that the request is coming from, this is then sent to the backend where all the necessary CLI commands to train the model are invoked using bash commands, this is then put into a training queue along with the UUID v4, and the training process id. And when a user aborts a training session which can be done by clicking the *Abort* button on the bottom right corner of the *Configurations* UI, which only appears when a model training is in progress as seen in Figure 4.6.2.1, that user's UUID v4 is sent to the backend and it is checked against the training queue to check whether an entry exists with the said UUID v4, if so the necessary CLI commands to abort the training is invoked using bash command calls for the relevant training process id, which in turn aborts the training.

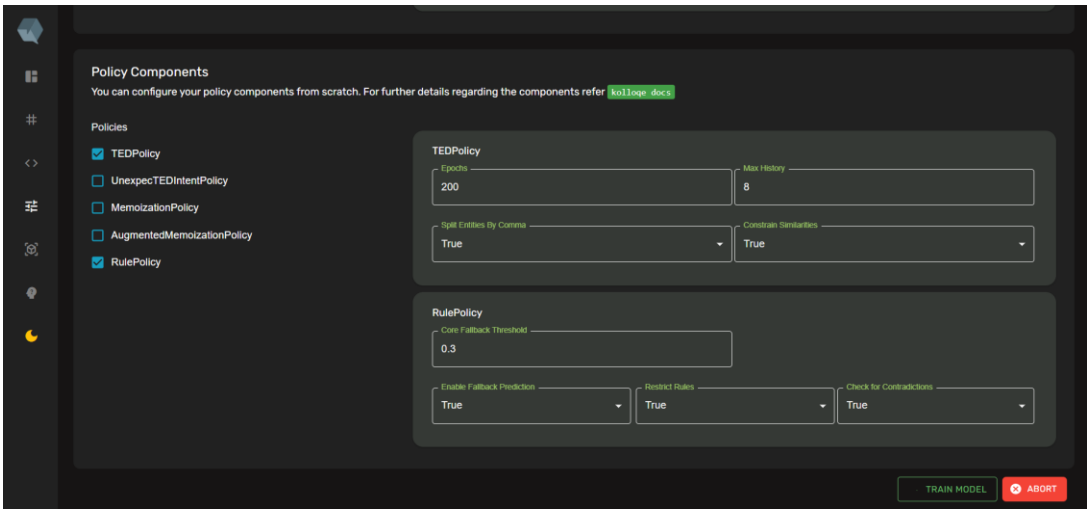


Figure 4.6.2.1. The UI when the model training is in progress and the *Abort* button is visible to abort training

4.7 ML Model Evaluation

Once a model is trained, user can view the trained model by navigating to the *Models* UI where all the trained models are displayed along with the name of the model and each model's train and test accuracy below the model's name as seen in Figure 4.7.1.

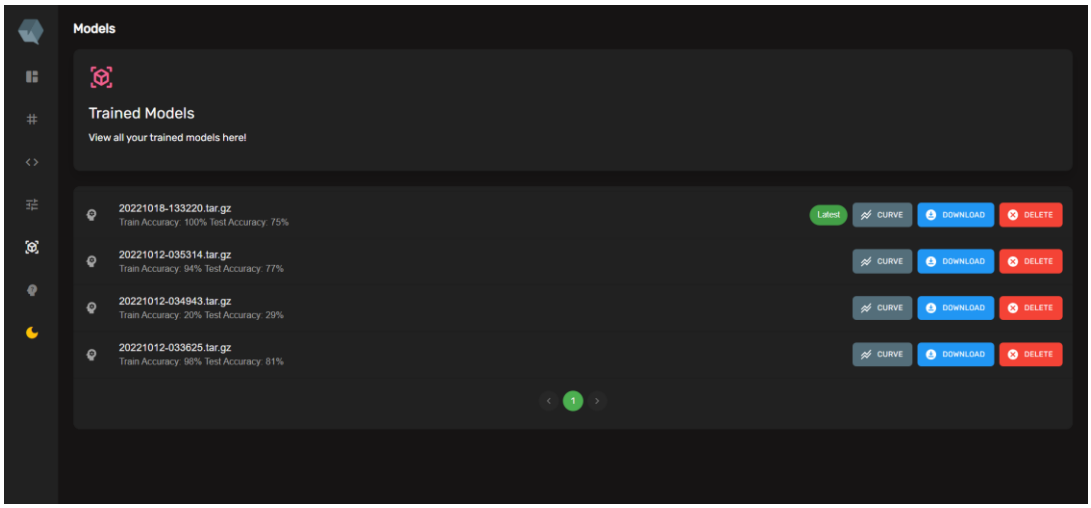


Figure 4.7.1. The *Models* UI listing all the trained models along with their train and test accuracies

4.7.1 Accuracy and Loss Curves

Users can click the *Curve* button to view both the accuracy curve and loss curve of each model as shown in Figure 4.7.1.1 and Figure 4.7.1.2 respectively. Users can use these learning curves to evaluate the trained models.

4.7.2 Curve Insights

Users can view curve insights provided by the application by clicking on the *Curve* button and then scrolling down passing the accuracy and loss curve graph. Figure 4.7.2.1 shows the UI where the loss curve is plotted, along with the best epoch suggested on top based on the patience interval chosen by the user. The moving average and the moving standard deviation are calculated for the train and test curves taking into account the chosen patience interval, which is the number of epochs to calculate the moving average with. Then 2 times the moving standard deviation is used to plot another two curves above and below the train and test curve. The training curve

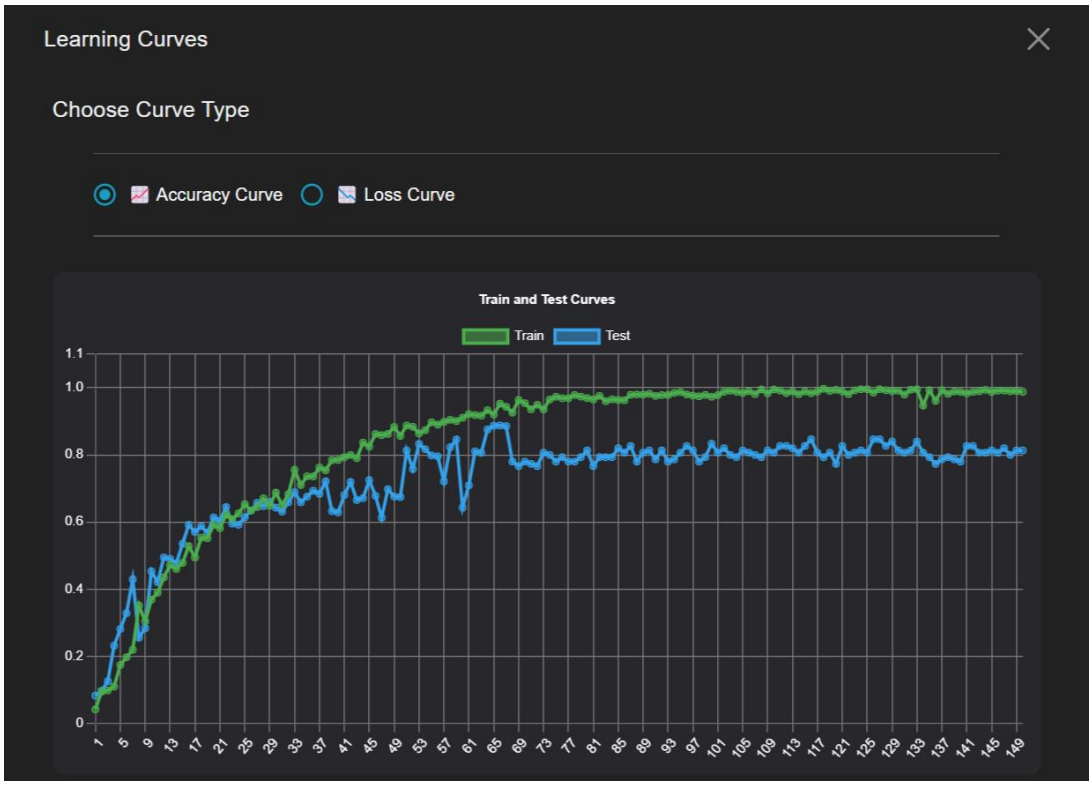


Figure 4.7.1.2. The accuracy curve of a model

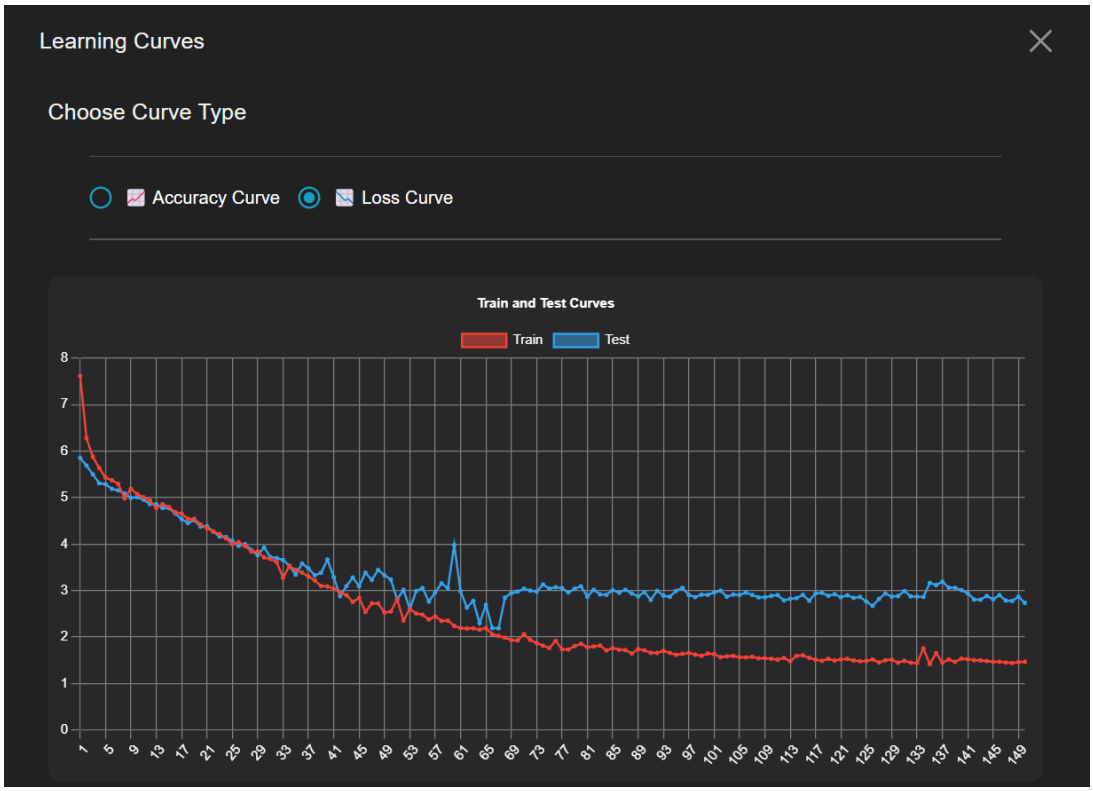


Figure 4.7.1.1. The loss curve of a model

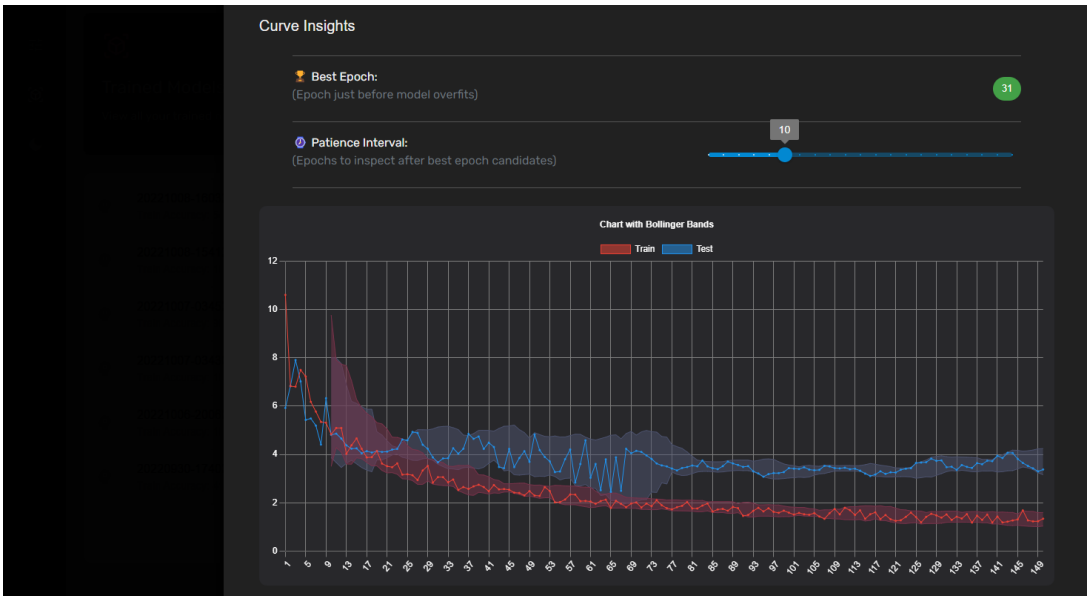


Figure 4.7.2.1. The best epoch suggested based on the chosen patience interval

standard deviation curve, while the red band below it is the negative 2 times standard deviation curve. The same is done for the test curve which is depicted in blue. The best is suggested by taking the lowest test epoch that is within the positive and negative moving standard deviations of the training curve, which are also known as Bollinger Bands. Initially, the first epoch is taken as the lowest epoch and then each corresponding lower epoch is taken as the lowest epoch, the number of epochs checked once an epoch is taken as the lowest depends on the chosen patience interval, which is taken as the number of epochs to check before terminating the algorithm and suggesting the current lowest epoch as the best epoch.

4.8 Code-switchable Chat Widget

4.9 Explainable AI for Trained Models

4.9.1 Generating explanations

4.9.2 On-demand explanation visualizations

4.10 Tools and Technologies

4.10.1 Kolloqe Package Implementation

Kolloqe has utilized Python 3.8 as the backend programming language. The Kolloqe implementation supports both Python 3.7 and 3.8. This version restriction is due to Rasa 2.8.8 which does not support newer Python versions. The Rasa open-source package was used to train a chatbot and test the Kolloqe functionalities.

4.10.2 Kolloqe Server Implementation

The Kolloqe server and API are a single Flask-based server that facilitates both local and production-level deployments. The production deployment of the Kolloqe server relies on a Web Server Gateway Interface (WSGI) server, waitress[23-Ishara]. The waitress server was primarily selected due to its extensive support for Windows and Linux operating systems, which is provided out-of-the-box. The Kolloqe server is run by running the CLI command *kolloqe*. The server can be run in both development and production modes by changing the environment variable file in the backend. Running the server in development mode disables the waitress production deployment mode and executes the flask server directly.

4.10.3 Kolloqe Server Frontend Development

The frontend of the Kolloqe server utilizes React as the frontend development framework primarily due to component reusability. The frontend is a single page application that uses Node 16.15.0 and utilizes material-ui, uuid, prop-types, react-router, framer-motion, and bollinger-bands NPM based packages.

4.10.4 Integrated Development Environment

PyCharm and Visual Studio Code were used as the primary IDE for the frontend and backend respectively development. Also *.env* files were used to hold the required environmental variables and credentials and used a requirements.txt file to state required Python packages for future references and deployment.

4.10.5 Source Code and Versioning

GitLab is used as the source code management platform and release management of Kolloqe which is not publicly visible since Kolloqe is a proprietary application. All releases of Kolloqe adhere to semantic versioning, making it easy to identify major, minor, and patch releases and breaking changes.

Table 4.10.5.1. Summary of Tools and Technologies utilized

Task	Tools to be used
Kolloqe Python package development	Python 3.8, NumPy, Pandas, sklearn, Rasa 2.8.8, PyCharm, Visual Studio Code, Google CoLab, Anaconda Distribution, Dotenv
Kolloqe server frontend implementation and algorithm development	Flask, Flask-Cors, psutil, waitress, Bootstrap, React JS, Axios, UUID4, Material-UI, framer-motion, prop-types, react-router, bollinger-bands
Process queue implementation	SQLite3
Chatbot development	Rasa 2.8.8, Rasa-SDK 2.8.0
Source code and version management	GitHub, GitLab, semantic-versioning, PyPI
Testing	Jest, Docker

Table 4.10.5.1 summarizes the tools and technologies stated in all the sub-sections of section 4.10.

4.11 Commercialization Aspect of the Product

Each research component of the overall research project provides applications for various Natural Language Processing and Machine Learning model evaluation-related tasks. Although the outputs of each research component can help design many products and services, they were all integrated to build a conversational AI that fully supports Sinhala-English code-switching. The main reason for developing a conversational AI are as follows.

1. Conversational AIs are a trending topic, and there is an increasing demand for Sinhala-based conversational AIs. Many businesses are looking to increase

their customer reach by using conversational AIs for a vast range of business tasks, including providing technical assistance, automating manual tasks such as booking tickets, and providing the information requested by the customers. Although that is the case, it is hard to find Sinhala-based conversational AIs, especially in Sri Lanka. Thus, developing Sinhala-based conversational AIs was identified as a potential business opportunity.

2. Although there are many conversational AI development frameworks, most of them lack evaluation tools to debug machine learning models. In frameworks like Rasa, model evaluations are highly technical, and the average developers without machine learning knowledge fail to identify problems and debug the machine learning models. Solving this issue by allowing non-technical users to maintain and evaluate machine learning models and conversational AIs is another business opportunity where evaluation tools can be built and released as add-on features.

3. Businesses are moving towards cloud-based solutions, especially SaaS products from traditional standalone applications, web applications, and on-premises solutions, where the maintenance cost is high. A cloud-based highly configurable conversational AI would be an appropriate solution for many businesses where the effort for maintenance is considerably low.

The end product of the overall research concentrates on designing a solution for the above potential business ideas and opportunities. The conversational AI developed as the research end-product is mainly a SaaS or simply a CaaS (conversational AI-as-a-service) product that a business can purchase with a set of add-on features, as illustrated in Figure 6.1. In addition to the CaaS packages, on-premises and demo cloud-based conversational AI packages are available for affordability and convenience. The end product of the overall research has the following archivable user benefits.

1. Businesses can purchase a CaaS package and eliminate conversational AI maintenance efforts.

2. Developers can use code-less maintenance tools to maintain conversational AIs they purchase without having in-depth knowledge about the backend deployment and the conversational AI development framework.

3. Businesses can purchase evaluation tools as add-ons and generate model explanations and evaluate machine learning models themselves to avoid extra maintenance costs. Here, the generated evaluations are easy to understand by non-technical users, which is not a feature of any existing chatbot framework.

4. Users of conversational AI can easily use the Sinhala-English code-switchable keyboard interface, and businesses can attract more users from having this feature as it eliminates the need to use third-party Sinhala typing services.

5. Businesses have the freedom to purchase either the CaaS packages or on-premises packages as per their need, while anyone can test the demo conversational AI for a period of 1 month before deciding to purchase any of the other packages that have a cost assigned.

6. Businesses can reach a vast customer range through Sinhala-English code-switching-based conversational AIs, especially within the Sri Lankan market, and it is possible to eliminate the need for having a dedicated customer care staff. It will dramatically lower the expenses of the business.

The proposed commercialisation plan of the end product of the overall research component contains a set of convenient packages and the offered features of each package differ based on the cost attached to it. The distribution of the features and the package cost were carefully planned and designed by analyzing the existing purchasable packages of similar SaaS products and conversational AIs. Table 6.1 clearly illustrates the feature variation and the cost difference of the packages offered by the proposed commercialisation plan.

Table 6.1: Feature variations and cost difference of packages proposed by the commercialisation plan

Feature	Packages				
	Demo	On-Prem	CaaS		
			Starter	Pro	Genius
Intents	10	Unlimited	20	180	400
API Integrations	2	Unlimited	2	110	200
Bot Analytics	✓	✓	-	✓	✓
CDD	✓	✓	✓	✓	✓
Sinhala Entity Annotating	✓	-	-	✓	✓
ML Evaluation Tools	-	-	-	-	✓
Maintenance Fee	-	2 Free + $9.99 per additional call	-	-	-
Trial Duration	1 Month	-	-	-	-
Package Price	Free	$199.99	$9.99	$34.99	$49.99

Figure 6.1: proposed commercialisation plan

5. TESTING & IMPLEMENTATION RESULTS & DISCUSSION

5.1 Results

5.1.1 Kolloqe Server Testing Results

The test cases in Table 5.1.1.1 and Table 5.1.1.2 covers the tasks of setting up the Kolloqe server in development and production mode respectively, using the CLI.

Table 5.1.1.1. Test Case for Testing Kolloqe Server in Debug Mode

<table>
<tr><td colspan="6">Project ID: 2022-056-IT19064932</td></tr>
<tr><td colspan="6">Project Name: Kolloqe</td></tr>
<tr><td colspan="6">Project Function: Kolloqe Development Server Deployment</td></tr>
<tr><td colspan="3">Test case ID: 001</td><td colspan="3">Test case designed by:
ID No: IT19064932
Name: Hameed M.S.</td></tr>
<tr><td colspan="6">Test Priority (High/Medium/Low): Medium</td></tr>
<tr><td colspan="6">Test Description: Test the Kolloqe server initialization in debug mode</td></tr>
<tr><td colspan="6">Prerequisite: Kolloqe Python package must be installed in the virtual environment</td></tr>
<tr><td colspan="6">Test Steps:
Step 1: Open a terminal and activate the conda environment
Step 2: Go to the project directory in the terminal
Step 3: Run kolloqe command with –debug to turn on debugging
Step 4: Wait until the Kolloqe server starts in the debug mode and click the URL to open a web browser and confirm the server is up and running</td></tr>
<tr><td>Test ID</td><td>Test Inputs</td><td>Expected Outputs</td><td>Actual Output</td><td>Result (Pass/Fail)</td><td>Comments</td></tr>
<tr><td>001</td><td>kolloqe --debug command</td><td>Kolloqe server must initialize in the debug mode and serve the Kolloqe developer console on port 6070</td><td>It can be seen in the console that the server is starting in development mode. The server loads as expected in port 6070. The developer console opens confirming that the server hosts the Kolloqe developer console without any issues.</td><td>Pass</td><td>Kolloqe server is served in debug mode as required</td></tr>
</table>

Table 5.1.1.2. Test Case for Testing Kolloqe Server in Production Mode

Project ID: 2022-056-IT19064932					
Project Name: RASAC					
Project Function: RASAC Production Server Deployment					
Test case ID: 002			**Test case designed by:** **ID No:** IT19064932 **Name:** Hameed M.S.		
Test Priority (High/Medium/Low): Medium					
Test Description: Test the RASAC server initialization in production mode					
Prerequisite: RASAC Python package must be installed in the virtual environment					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Go to the project directory in the terminal Step 3: Run *rasac server* command Step 4: Wait until the RASAC server starts in production mode and click the URL to open a web browser and confirm the server is up and running					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
002	*rasac server* command	RASAC server must initialize in the production mode and serve the RASAC developer console on port 6069	It can be seen in the console that the server is starting in *production* mode. The server loads as expected in port 6069. The developer console opens confirming that the server hosts the RASAC developer console without any issues.	Pass	RASAC server is served in production mode as required

5.1.2 Custom Entity Annotation Results

The main objective of implementing SIENA is reduce the time taken to text annotation. To achieve the main objective SIENA consists of name entity suggestion feature (shown in **Error! Reference source not found.**), auto annotation feature (shown in **Error! Reference source not found.**) and portable knowledge base component (shown in **Error! Reference source not found.**) To test the effectiveness of SIENA tool, provided seventeen lines of text document which contained sentences related to SLIIT degrees. This document provided text dataset to four users and recorded time taken to annotate with SIENA tool and the manual annotation method. SIENA tool achieved 54.7% average effectiveness when compared to manual annotation as shown in the Table 5.1.2. 1.

Table 5.1.2. 1. SIENA user testing results

User	**Time taken to annotate using SIENA (seconds)**	**Time taken to manual annotating (seconds)**	**Time efficiency percentage (%)**
User 1	135	184	36.3
User 2	114	178	56.1
User 3	106	181	70.7
User 4	125	195	56.0
Average	120	184.5	54.7

SIENA tool's main functionalities are test by using the following test cases.

Table 5.1.4. 1. Test case 001

Project ID: 2022-056-IT19051208	
Project Name: SIENA	
Project Function: Name entity suggestions	
Test case ID: 001	**Test case designed by:** **ID No:** IT19051208 **Name:** Sakalasooriya S.A.H.A.
Test Priority (High/Medium/Low): Medium	
Test Description: The name entity suggestion list should be rearranged once user annotate a similar phrase.	
Prerequisite: User should enter list of name entities to SIENA tool using the entity management window.	

Test Steps: Step 1: Load a file to start the annotation Step 2: Highlight first word Step 3: Click on a name entity from the list (near to the highlighted area) Step 4: Highlight similar word (used in step 2) Step 5: Observer the order of name entity suggestion list.					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
001	**Input:** Selected “IT ඩිග්‍රි” as the first word and assigned “degree_type” as the name entity. Selected “IT ඩිග්‍රියේ” as the second word	Entity list rearrange by coming up “degree_type” as the first in the entity list	Entity list rearranged by coming up “degree_type” as the first in the entity list	pass	Entity suggestion function is working properly.

Table 5.1.4. 2. Test case 002

Project ID: 2022-056-IT19051208					
Project Name: SIENA					
Project Function: Auto annotation					
Test case ID: 002			**Test case designed by:** **ID No:** IT19051208 **Name:** Sakalasooriya S.A.H.A.		
Test Priority (High/Medium/Low): Medium					
Test Description: Auto annotate all the variation of base word when user selects a base word from auto annotation menu.					
Prerequisite: User should annotate at least one name entity to appear base word in auto annotation windows.					
Test Steps: Step 1: Load a file to start the annotation Step 2: Click on base word list icon Step 3: Click on base word Step 4: Observer the auto annotation					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments

002	**Input:** Selected “it ඩිශ්” as the base word.	Auto annotate all the identified variations of the selected base word.	Auto annotated all the identified variations of the selected base word.	pass	Auto annotation function is working properly. Auto annotation success notifications works properly

Table 5.1.4. 3. Test case 003

Project ID: 2022-056-IT19051208					
Project Name: SIENA					
Project Function: Exporting knowledge base					
Test case ID: 003			**Test case designed by:** **ID No:** IT19051208 **Name:** Sakalasooriya S.A.H.A.		
Test Priority (High/Medium/Low): Medium					
Test Description: Export knowledge base as a CSV file.					
Prerequisite: User should annotate at least one name entity to contain details inside the exported knowledge base.					
Test Steps: Step 1: Click on export knowledge button Step 2: Save the downloaded file.					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
003	**Input:** No inputs	Web browser ask to save the knowledge base file.	Web browser asked to save the knowledge base file.	pass	Knowledge base export function works properly. Knowledge base export notification works properly.

Table 5.1.4. 4. Test case 004

Project ID: 2022-056-IT19051208					
Project Name: SIENA					
Project Function: Auto annotation					
Test case ID: 004			**Test case designed by:** **ID No:** IT19051208 **Name:** Sakalasooriya S.A.H.A.		
Test Priority (High/Medium/Low): Medium					
Test Description: Auto annotate all the variation of base word when user selects a base word from auto annotation menu.					
Prerequisite: User should have up and running SIENA instance.					
Test Steps: Step 1: Click on import knowledge base button Step 2: Find previously exported knowledge base file Step 3: Upload selected file					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
004	**Input:** Navigate to previously exported knowledge base file and select it in file selection dialog box.	Update current knowledge base	Updated current knowledge base	pass	Knowledge base import function works properly. Knowledge base import notification works properly.

5.1.3 Equivalent Token Mapping Results

pending

5.1.4 Codeless Model Improvement Results

One of the main reasons behind implementing a GUI to configure the ML pipeline and policy components was to reduce the time taken to configure the model compared to manually typing the YAML configuration file in the backend. To test the time difference between the two approaches, a controlled experiment was set up where four (4) users were given a YAML configuration file and were asked to manually type the file again to mimic developers manually typing the YAML configuration file (this experiment makes the task easy since the test subjects are provided with all the configurations already which saves time from having to go through documentations to know what the required syntax is for a component), and they were also asked to look at the provided file and configure the components using the GUI, and the timing taken for the two tasks were recorded and the recorded timings can be seen in Table 5.1.2.2. As we can see, there is a huge improvement in the efficiency when configuring the model using the RASAC GUI compared to manually typing the configuration file, with the average efficiency being 449%.

Table 5.1.2.2. Comparing the time taken to manually type a YAML configuration file for a ML model vs using the RASAC GUI to configure the model

User	**Time taken to manually type the config file (min:sec)**	**Time taken to configure using the GUI (min:sec)**	**Efficiency Percentage (%)**
User 1	10:26	2:30	317%
User 2	14:28	2:28	486%
User 3	15:15	2:25	531%
User 4	09:36	1:39	482%
Average	12:43	2:16	449%

The test cases from Table 5.1.2.2 to Table 5.1.2.5 covers the tasks of configuring the ML pipeline and policy components from scratch, configuring ML pipeline and policy components using existing model configurations, configuring training models, and aborting a model that is in the process of being trained respectively, using the Configurations GUI

Table 5.1.2.3. Test Case for Testing the Pipeline and Policy Components in the Configurations GUI

Project ID: 2022-056-IT19064932					
Project Name: RASAC					
Project Function: Pipeline and Policy Configuration Selection					
Test case ID: 001			**Test case designed by:** **ID No:** IT19064932 **Name:** Hameed M.S.		
Test Priority (High/Medium/Low): High					
Test Description: Configurations selected in the UI should be passed to the config file in the backend					
Prerequisite: The RASAC server must be running					
Test Steps: Step 1: Select the required pipeline and policy components from the configuration UI Step 2: Click the *train* button in the UI Step 3: Check the YAML config file in the backend					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
001	Choosing the required pipeline and policy components in the UI	The backend YAML config file being populated with the selected pipeline and policy components	The backend YAML config file is populated with the selected pipeline and policy components	Pass	Pipeline and policy component selection works as required

Table 5.1.2.4. Test Case for Testing the Option to Populate the Pipeline and Policy Components in the Configurations GUI using existing models

Project ID: 2022-056-IT19064932					
Project Name: RASAC					
Project Function: Pipeline and Policy Existing Model Configurations					
Test case ID: 002			**Test case designed by:** **ID No:** IT19064932 **Name:** Hameed M.S.		
Test Priority (High/Medium/Low): High					
Test Description: Selecting pipeline and policy component configurations from existing models					
Prerequisite: The RASAC server must be running					
Test Steps: Step 1: Click the dropdown denoted by *Existing Models* Step 2: Select a model from the dropdown Step 3: Check the pipeline and policy components automatically selected in the UI against the components previously selected when configuring the selected model					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
002	Select a model from the *Existing Models* dropdown	The pipeline and policy components automatically selected in the UI are the same as the ones selected when training the selected model	The correct pipeline and policy components are automatically selected in the UI	Pass	Choosing existing model configurations work as required

Table 5.1.2.5. Test Case for Testing the Option to Train a Model in the Configurations GUI

Project ID: 2022-056-IT19064932					
Project Name: RASAC					
Project Function: Successfully train a ML model					
Test case ID: 003			**Test case designed by:** **ID No:** IT19064932 **Name:** Hameed M.S.		
Test Priority (High/Medium/Low): High					
Test Description: Training an ML model using the UI after selecting the required components					
Prerequisite: The RASAC server must be running					
Test Steps: Step 1: Select the required pipeline and policy components from the configuration UI Step 2: Click the *train* button in the UI Step 3: Check for the toast message once training ends					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
003	Select the required configurations and train the model	• Receiving a toast message saying the model was trained successfully • The trained model can be viewed in the *Models* tab	• A toast message is displayed saying the model was trained successfully • The trained model is available to be viewed in the *Models* tab	Pass	Model training works as required

5.1.5 ML Model Evaluation Results

Table 5.1.3.1 shows the results of a test conducted, where four users who are data science undergraduates, were shown the loss curve of two ML models and were asked to state what they thought was the best epoch. And then the algorithm was used to find the best epoch on the said two curves while changing the patience interval values between 5, 10, 15, 20, and 25.

The test cases from **Error! Reference source not found.** to **Error! Reference source not found.** covers the tasks of checking a model's training and testing accuracy, viewing the accuracy curve of a model, viewing the loss curve of a model, and testing out the suggesting the best epoch feature respectively, using the *Models* GUI.

Table 5.1.3.1. Comparing the best epoch according to four users against the best epoch suggested by the algorithm

User	**Best epoch according to user**	**Best epoch suggested by algorithm at different patient interval values**				
		5	**10**	**15**	**20**	**25**
Model with 150 Epochs (Figure 5.1.3.1)						
User 1	67	42	53	67	67	67
User 2	67					
User 3	67					
User 4	67					
Model with 90 Epochs (Figure 5.1.3.2)						
User 1	36	36	36	52	60	68
User 2	36					
User 3	36					
User 4	36					

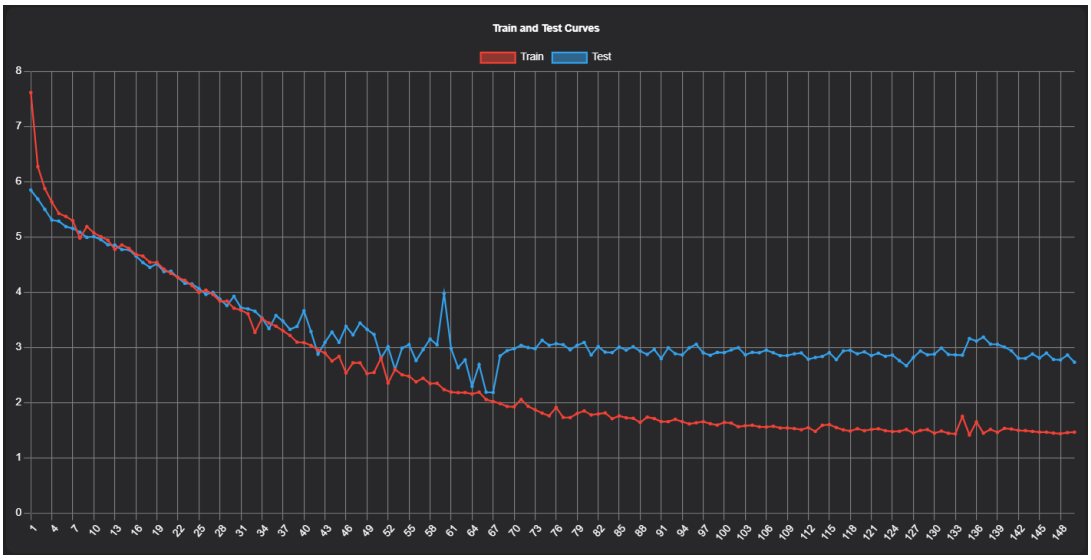


Figure 5.1.3.1. A loss curve with 150 epochs

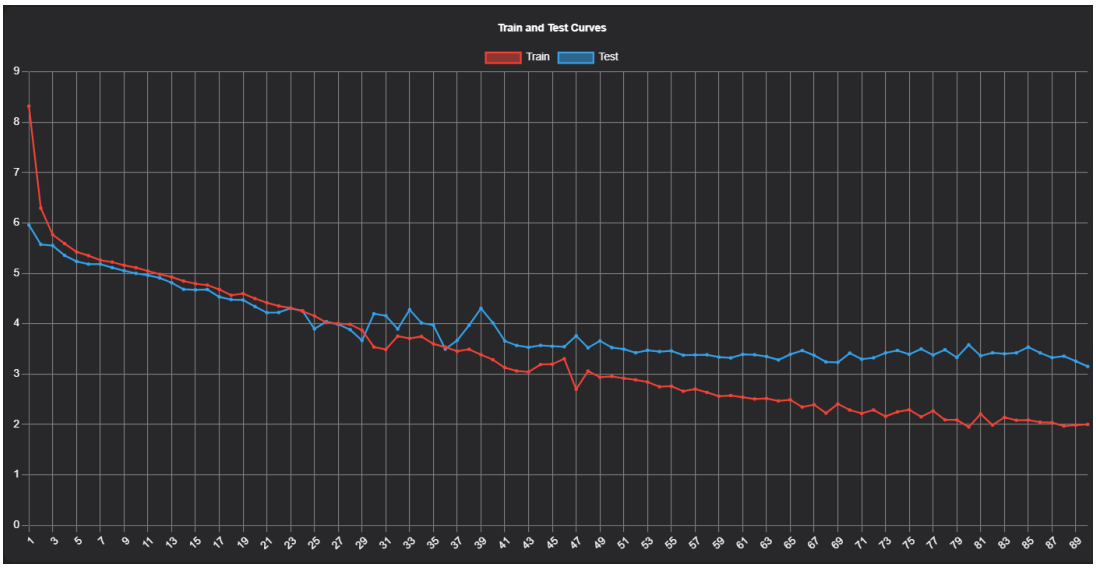


Figure 5.1.3.2. A loss curve with 90 epochs

5.1.6 Code-switchable Chat Widget Results

5.1.7 Explainable AI for Trained Models Results

The total algorithm execution time for DIME is the summation of the time taken to calculate the initial confidence of the unmodified dataset, GFI compute time for each token and the DFI calculation time for all sifted tokens. Table 5.1.2.6 presents the execution time for GFI calculation for the unmodified dataset, token-wise GFI calculation, and DFI calculation observed over five distinct data instances. Based on the observed execution times, the author gained the following insights.

Table 5.1.2.6. DIME algorithm execution time evaluation

Data Instance	Number of Distinct Tokens	Execution time observed (seconds)			
		Initial global confidence	Token-wise GFI	DFI	Total Execution Time
SLIIT එකේ තියෙන උපාධි මොනවද?	5	22.515625	SLIIT: 18 එකේ: 17 තියෙන: 17 උපාධි: 17 මොනවද: 18	0.182187	109.697812
ඩීන්ස් ලිස්ට් කියන්නේ මොකක්ද	4	23.546875	ඩීන්ස්: 17 ලිස්ට්: 18 කියන්නේ: 17 මොකක්ද: 17	0.134154	92.681029
GPA වලට අදාල grades දැනගන්න පුළුවන්ද?	6	24.25	GPA: 18 වලට: 18 අදාල: 18 grades: 18 දැනගන්න: 18 පුළුවන්ද: 18	0.196699	132.446699
ස්ලිට් එකේ scholarships දෙනවද?	4	22.53125	ස්ලිට්: 18 එකේ: 18 Scholarships: 17 දෙනවද: 17	0.088495	92.619745
බොහොම ස්තුතියි, සුබ දවසක්!	4	22.421875	බොහොම: 18 ස්තුතියි: 18 සුබ: 18 දවසක්: 18	0.066278	94.488153
Column Total	23	115.265625	406	0.667813	521.933438
Column Average	4.6 (4 or 5)	23.053125	81.2 (Instance-wise) or 17.652173 (Token-wise)	0.133563	104.3866876

1. Overall, for a training dataset with approximately 635 examples and a data instance comprising 4 or 5 words, the algorithm can conclude the execution in about 104.39 seconds (1.74 minutes).

2. For a training dataset with approximately 635 examples, the algorithm takes about 23.05 seconds to calculate the initial global confidence score.

3. For a training dataset with approximately 635 examples, DIME calculates GFI roughly in 17.65 seconds for a single token. If the data instance contains around 4 or 5 words, the GFI calculation takes about 81.2 seconds (1.35 minutes).

4. For a training dataset with approximately 635 examples and a data instance with 4 or 5 words, the algorithm takes roughly 0.13 seconds to calculate the DFI.

If the number of examples in the training dataset remains unchanged, the initial global confidence calculation takes a constant time and only GFI and DFI calculation time depend on the number of unique words present in the data instance.

It is worth noticing that GFI calculation is more compute intensive than DFI calculation, and the overall algorithm execution time is proportional to the number of examples in the dataset and the number of distinct tokens in the data instance.

The DIME XAI package has undergone a series of integration tests after integrating the sub-packages of DIME into a standalone Python package. Test cases created for each integration testing task are available from Table 5.1.3.1 through Table 5.1.3.8, and the test cases covered all primary tasks performed in the DIME CLI and

GUI. Note that test cases 004 to 010 comprises smoke tests as they assessed the most critical functionalities of the integrated package.

Table 5.1.3.2. Integration test case for DIME project initialization

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME project initialization					
Test case ID: 004			**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.		
Test Priority (High/Medium/Low): High					
Test Description: Create a new DIME project using DIME CLI and verify necessary files and directories are created.					
Prerequisite: DIME Python package must be installed in the virtual environment					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Create a new folder as the new DIME project root Step 3: Run *dime inti* command Step 4: Specify a sub-folder location to create a new DIME project. (Press enter to select the current directory) Step 5: Wait until the project is successfully created and observe the created files.					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
004	*dime init* command	DIME directories and files created in the new folder where the project was initialized	Observed that all folders and dime_config.yml file is correctly created.	pass	This was tested on both Windows and Linux. For Linux-based testing, Docker was used.

Table 5.1.3.3. Integration test case for DIME project initialization at rasa root

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME Project Initialization					
Test case ID: 005			**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.		
Test Priority (High/Medium/Low): High					
Test Description: Create a new DIME project using DIME CLI on directory with a Rasa project and verify necessary files and directories are created and overlapping files are ignored.					
Prerequisite: DIME Python package must be installed in the virtual environment					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Go to an existing Rasa project directory in the terminal Step 3: Run *dime inti* command Step 4: Specify a sub-folder location to create a new DIME project. (Press enter to select the current directory) Step 5: Wait until the project is successfully created and observe the created files.					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
005	*dime init* command	DIME directories and files created in the directory with an existing Rasa project, without overlapping the *data* and *models* directories	Observed that all folders and dime_config.yml file is correctly created without replacing the existing Rasa NLU data or models.	pass	This was tested on both Windows and Linux.

Table 5.1.3.4. Integration test case for DIME CLI explainer

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME CLI Explainer					
Test case ID: 006		**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.			
Test Priority (High/Medium/Low): Medium					
Test Description: Trigger DIME explainer from CLI interface and observe if configurations are loaded as expected from the *dime_config.yml* file.					
Prerequisite: DIME Python package must be installed in the virtual environment and a trained Rasa model, and the dataset must be available in the project directory					
Test Steps: Step 1: Open *dime_config.yml* and verify that a data instance is specified, and the metric is confidence. Step 2: Open a terminal and activate the conda environment Step 3: Open the project directory in the terminal Step 4: Run *dime explain* command Step 5: Wait until explainer runs successfully and CLI visualizations are rendered.					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
006	*dime explain* command	DIME CLI Explainer must load configurations and data instance from *dime_config.yml*, generate the explanation, and visualize in the CLI.	Observed that the configurations and the data instance was loaded correctly. The CLI explainer generated the explanation and rendered it to the CLI without any issue.	pass	This was again tested with the –*instance* flag to see if the data instance gets overridden. The result was a success.

Table 5.1.3.5. Integration test case for DIME CLI visualizer

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME CLI Visualizer					
Test case ID: 007			**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.		
Test Priority (High/Medium/Low): Medium					
Test Description: Trigger DIME visualizer from CLI interface and observe if a valid explanation JSON object gets rendered into the CLI.					
Prerequisite: DIME Python package must be installed in the virtual environment and an already generated explanation JSON file must reside in the project directory					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Go to the project directory in the terminal Step 3: Run *dime visualize* command with *--explanation* flag with the correct explanation filename. (*dime_results_20220917_210016.json*) Step 4: Wait until explainer runs successfully and CLI visualizations are rendered.					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
007	*dime visualize -e filename* command	DIME CLI Visualizer must load the specified explanation object, verify the content, and visualize it in the CLI.	Observed that the specified explanation file content is correctly read and rendered in the CLI as expected.	pass	Tested with multiple explanation objects.

Table 5.1.3.6. Integration test case for DIME Server

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME Server Deployment					
Test case ID: 008			**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.		
Test Priority (High/Medium/Low): Medium					
Test Description: Spin up the DIME server and test the server initialization in production and debug modes.					
Prerequisite: DIME Python package must be installed in the virtual environment and a valid *dime_config.yml* file must reside in the project directory.					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Go to the project directory in the terminal Step 3: Run *dime server* command with *--debug* to turn on debugging Step 4: Wait until the DIME server starts in the debug mode and open up the URL in a web browser and confirm that the server is up and running.					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
008	*dime server --debug* command	DIME Server must initialize in the debug mode and serve the DIME developer console on port 6066	Observed that the server loads as expected in port 6066. The developer console was opened and confirmed that the server hosts the DIME developer console without an issue.	pass	The same steps were repeated without the *--debug* flag to test the production mode with *waitress*.

Table 5.1.3.7. Integration test case for DIME Server explanations

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME Server Explanation Generation					
Test case ID: 009		**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.			
Test Priority (High/Medium/Low): High					
Test Description: Spin up the DIME server and test the server initialization in production and debug modes to test explanation generation.					
Prerequisite: DIME Python package must be installed in the virtual environment and a valid *dime_config.yml* file must reside in the project directory.					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Go to the project directory in the terminal Step 3: Run *dime server* command with *--debug* to turn on debugging Step 4: Wait until the DIME server starts in the debug mode and open the URL in a web browser and confirm that the server is up and running. Step 5: Go to the dashboard page and change the data instance Step 6: Click on *explain* button to start an explanation job Step 7: Insect terminal and verify the request is processing Step 8: Wait till the job finishes and inspect if the generated visualization appears at the bottom of the dashboard page Step 9: Go to the *explanations* page and verify the latest explanation object is available (based on the timestamp) Step 10: Open the explanation and verify that it is correctly rendering Step 11: Go to the dashboard page and verify that the explanations count on the top status bar has increased or not					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
009	Run *dime server --debug* Input any data instance in the dashboard	Explanation job must run, and the request id must be attached. After the job completes, result must render, and the statistics and the explanations list should be automatically updated.	Observed that the server runs the job with a valid request id. The results were rendered, statics were updated, and the explanation list contained the latest object.	pass	This was repeatedly done in the production environment and for various input data instances.

Table 5.1.3.8. Integration test case for DIME Server visualizations

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME Server Visualization					
Test case ID: 010			**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.		
Test Priority (High/Medium/Low): High					
Test Description: Spin up the DIME server and test the server initialization in production and debug modes to test explanation visualization.					
Prerequisite: DIME Python package must be installed in the virtual environment and a valid *dime_config.yml* file must reside in the project directory.					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Go to the project directory in the terminal Step 3: Run *dime server* command with *--debug* to turn on debugging Step 4: Wait until the DIME server starts in the debug mode and open up the URL in a web browser and confirm that the server is up and running. Step 5: Navigate to the *explanations* tab in the sidebar Step 6: Under explanations list, click on *visualize* button under any explanation Step 7: Inspect the explanation visualization overlay					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
010	Run *dime server --debug* command	Explanation must be rendered when clicking on the *visualize* button and an overlay should slide from the right side of the screen. Both GFI and DFI must be correctly rendered.	Observed that the overlay appears as expected and GFI and LFI visualizations are correctly rendered.	pass	The same steps were repeated for multiple explanations.

Table 5.1.3.9. Integration test case for DIME Server peak feature

Project ID: 2022-056-IT1906932					
Project Name: DIME-XAI					
Project Function: DIME Server Explanation Peak					
Test case ID: 011			**Test case designed by:** **ID No:** IT19069432 **Name:** Dissanayake D.M.I.M.		
Test Priority (High/Medium/Low): Medium					
Test Description: Spin up the DIME server and test the server initialization in production and debug modes and verify the *peak* feature is working as expected.					
Prerequisite: DIME Python package must be installed in the virtual environment and a valid *dime_config.yml* file must reside in the project directory.					
Test Steps: Step 1: Open a terminal and activate the conda environment Step 2: Go to the project directory in the terminal Step 3: Run *dime server* command with *--debug* to turn on debugging Step 4: Wait until the DIME server starts in the debug mode and open up the URL in a web browser and confirm that the server is up and running. Step 5: Navigate to the *explanations* tab in the sidebar Step 6: Click on *peak* button on the top-left corner Step 7: Pick a previously downloaded explanation object from the file open dialog Step 8: Verify if the visualizations are rendering and not permanently persisting					
Test ID	Test Inputs	Expected Outputs	Actual Output	Result (Pass/Fail)	Comments
011	*dime server --debug* command	The picked explanation object must be visualized right away, but should not upload it to the server, thus the explanation count, and the list must remain unchanged.	Observed that the selected object was correctly rendered when peaked. The object was not uploaded, and explanations the count and the list remained unchanged.	pass	The same steps were repeated for multiple explanation objects.

5.2 Research Findings

5.2.1 Custom Entity Annotation

Redjui

5.2.2 Equivalent Token Mapping

5.2.3 NLU Pipeline and Policy Configuration

5.2.4 ML Model Evaluation

5.2.5 Code-switchable Chat Widget

5.2.6 Explainable AI for Trained Models

5.3 Discussion

Pending

5.4 Member Contribution

Pending

6. CONCLUSION

Pending

REFERENCES

[1]. M. T. Ribeiro, S. Singh, C. Guestrin, "'why should I trust you?': Explaining the predictions of any classifier," 16 Feb 2016. [Online]. Available: https://arxiv.org/abs/1602.04938

[2]. T. Bunk, D. Varshneya, V. Vlasov, A. Nichol, "DIET: Lightweight Language Understanding for Dialogue Systems," 2020. [Online]. Available: https://arxiv.org/abs/2004.09936

[3]. G. Caldarini, S. Jaf, K. McGarry, "A Literature Survey of Recent Advances in Chatbots," 2022. [Online]. Available: https://arxiv.org/abs/2201.06657

[4]. F. K. Došilović, M. Brčić and N. Hlupić, "Explainable artificial intelligence: A survey," 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, pp. 0210-0215, doi: 10.23919/MIPRO.2018.8400040.

[5]. M. Danilevsky, K. Qian, R. Aharonov, Y. Katsis, B. Kawas, P. Sen, "A Survey of the State of Explainable AI for Natural Language Processing," 2020. [Online]. Available: https://arxiv.org/abs/2010.00711

[6]. Y. Jin and B. Sendhoff, "Pareto-Based Multiobjective Machine Learning: An Overview and Case Studies," IEEE Trans. Syst. Man Cybern. Part C Appl. Rev., vol. 38, no. 3, pp. 397-415, May 2008.

[7]. G. Montavon, W. Samek, and K.-R. Muller, "Methods for interpreting and understanding deep neural networks," Digit. Signal Process., vol. 73, pp. 1-15, Feb. 2018

[8]. S. R. Islam, W. Eberle, S. K. Ghafoor, en M. Ahmed, “Explainable Artificial Intelligence approaches: A survey,” 2021. [Online]. Available: https://arxiv.org/abs/2101.09429

[9]. P. Gohel, P. Singh, en M. Mohanty, “Explainable AI: current status and future directions”, 2021. [Online]. Available: https://arxiv.org/abs/2107.07045

[10]. S. Lundberg, S. Lee, “A Unified Approach to Interpreting Model Predictions,” 2017. [Online]. Available: https://arxiv.org/abs/1705.07874

[11]. H2O.ai, *Interpretable Machine Learning Using LIME Framework - Kasia Kulma (PhD), Data Scientist, Aviva.* (Dec. 19, 2017). Accessed: Dec. 25, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=CY3t11vuuOM&list=TLPQMjYxMjIwMjHjkkneYrcmyg&index=2

[12]. Xainlp2020.github.io, “*XAI for Natural Language Processing*”, [online] Available at: https://xainlp2020.github.io/xainlp [Accessed 24 January 2022].

[13]. D. Rajagopal, V. Balachandran, E. Hovy, en Y. Tsvetkov, “SelfExplain: A self-explaining architecture for neural text classifiers”, 2021. [Online]. Available: https://arxiv.org/abs/2103.12279

[14]. A. Shrikumar, P. Greenside, en A. Kundaje, “Learning important features through propagating activation differences”, 2017. [Online]. Available: https://arxiv.org/abs/1704.02685

[15]. Lloyd S Shapley. “A value for n-person games”. In: Contributions to the Theory of Games 2.28 (1953), pp. 307–317.

[16]. C. Molnar, 2022. "*Interpretable Machine Learning*", 2022. [online] Christophm.github.io. Available at: https://christophm.github.io/interpretable-ml-book [Accessed 24 January 2022].

[17]. T. Bocklisch, J. Faulkner, N. Pawlowski, A. Nichol, "Rasa: Open Source Language Understanding and Dialogue Management," 2017. [Online]. Available: https://arxiv.org/abs/1712.05181

[12-shamikh] "Conversational AI: What it is, why you should care and how to do it right," Social Media Marketing & Management Dashboard, 15-Dec-2021. [Online]. Available: https://blog.hootsuite.com/conversational-ai/. [Accessed: 28-Jan-2022].
[13-shamikh] X. Ying, "An overview of overfitting and its solutions", in Journal of Physics: Conference Series, 2019, vol 1168, bl 022022.
[14-shamikh] G. Paris, D. Robilliard, en C. Fonlupt, "Exploring overfitting in genetic programming", in Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, bll 267–277.
[15-shamikh] D. D. Jensen en P. R. Cohen, "Multiple comparisons in induction algorithms", Machine Learning, vol 38, no 3, bll 309–338, 2000.
[16-shamikh] By: IBM Cloud Education, "What is underfitting?," IBM. [Online]. Available: https://www.ibm.com/cloud/learn/underfitting. [Accessed: 28-Jan-2022].
[17-shamikh] M. A. Babyak, "What you see may not be what you get: a brief, nontechnical introduction to overfitting in regression-type models", Psychosomatic medicine, vol 66, no 3, bll 411–421, 2004.

[18-shamikh] G. Varoquaux, L. Buitinck, G. Louppe, O. Grisel, F. Pedregosa, en A. Mueller, “Scikit-learn: Machine learning without learning the machinery”, GetMobile: Mobile Computing and Communications, vol 19, no 1, bll 29–33, 2015.
[19-shamikh] F. Pedregosa et al., “Scikit-learn: Machine learning in Python. the Journal of machine Learning research 12 ”, 2011.
[20-shamikh] K. Team, “Simple. flexible. powerful.,” Keras. [Online]. Available: https://keras.io/. [Accessed: 08-Feb-2022].
[22-shamikh] “Facebook,” Facebook.com. [Online]. Available: https://www.facebook.com/business/news/insights/5-reasons-messaging-is-taking-flight-with-travelers. [Accessed: 17-Oct-2022].
[23-shamikh] J. Lee, “Do virtual reference librarians dream of digital reference questions?: A qualitative and quantitative analysis of email and chat reference”, Australian Academic & Research Libraries, vol 35, no 2, bll 95–110, 2004.
[24-shamikh] M. A. Nezami en R. Rukham, “Crowdsourced NLP Retraining Engine in Chatbots”, in International Conference on Emerging Technologies and Intelligent Systems, 2021, bll 311–320.

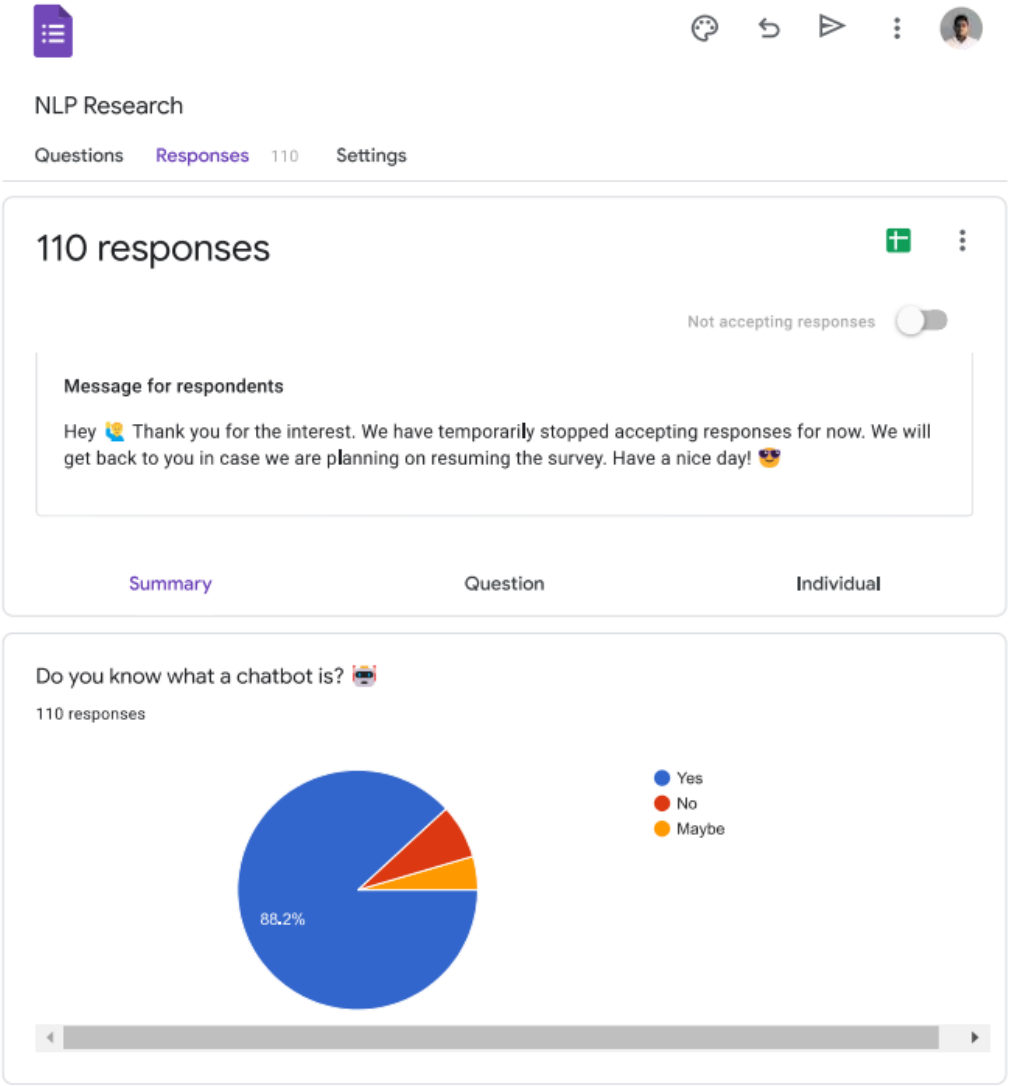
GLOSSARY

Glossary

APPENDICES

Appendix A: Survey Form

Figure A.1: Complete survey form questions and responses – part 1


NLP Research
Questions
Responses 110
Settings
110 responses
Not accepting responses
Message for respondents
Hey 🙋 Thank you for the interest. We have temporarily stopped accepting responses for now. We will get back to you in case we are planning on resuming the survey. Have a nice day! 😎
Summary
Question
Individual
Do you know what a chatbot is? 🤖
110 responses
Yes
No
Maybe
88.2%

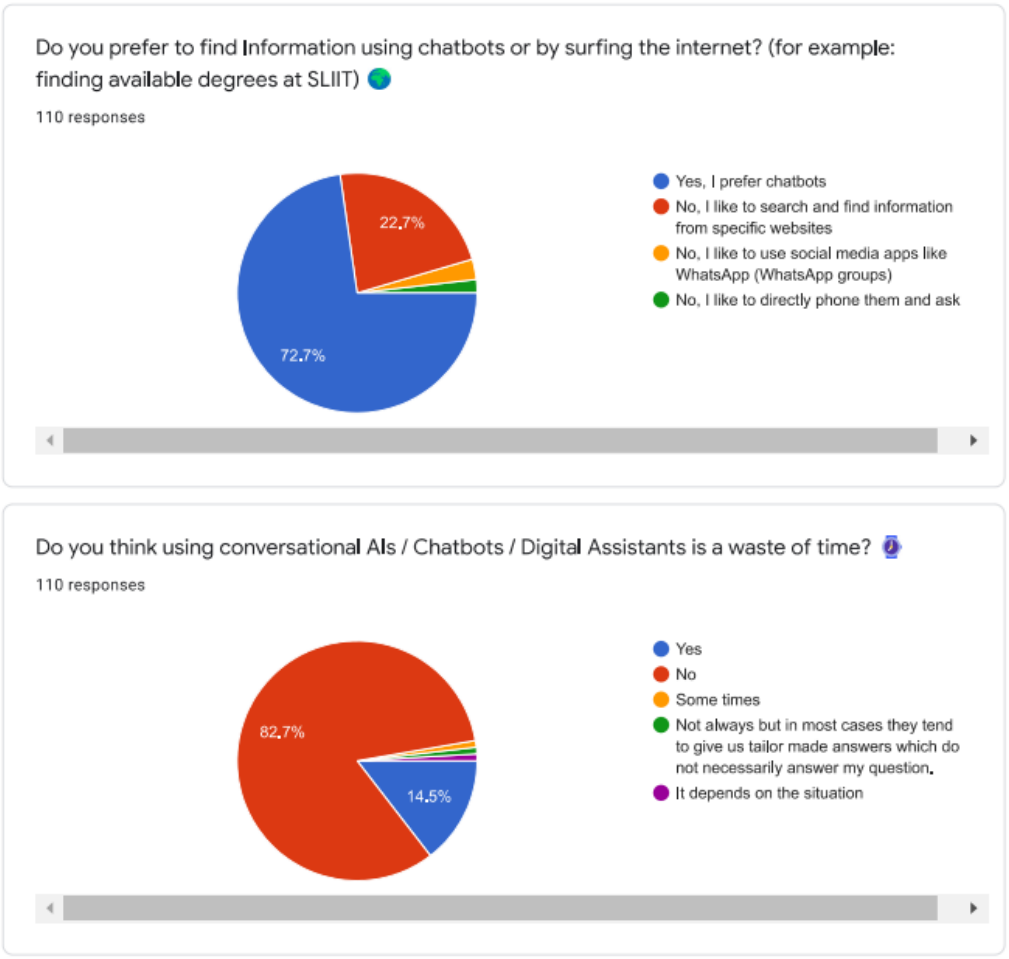


Figure A.2: Complete survey form questions and responses – part 2

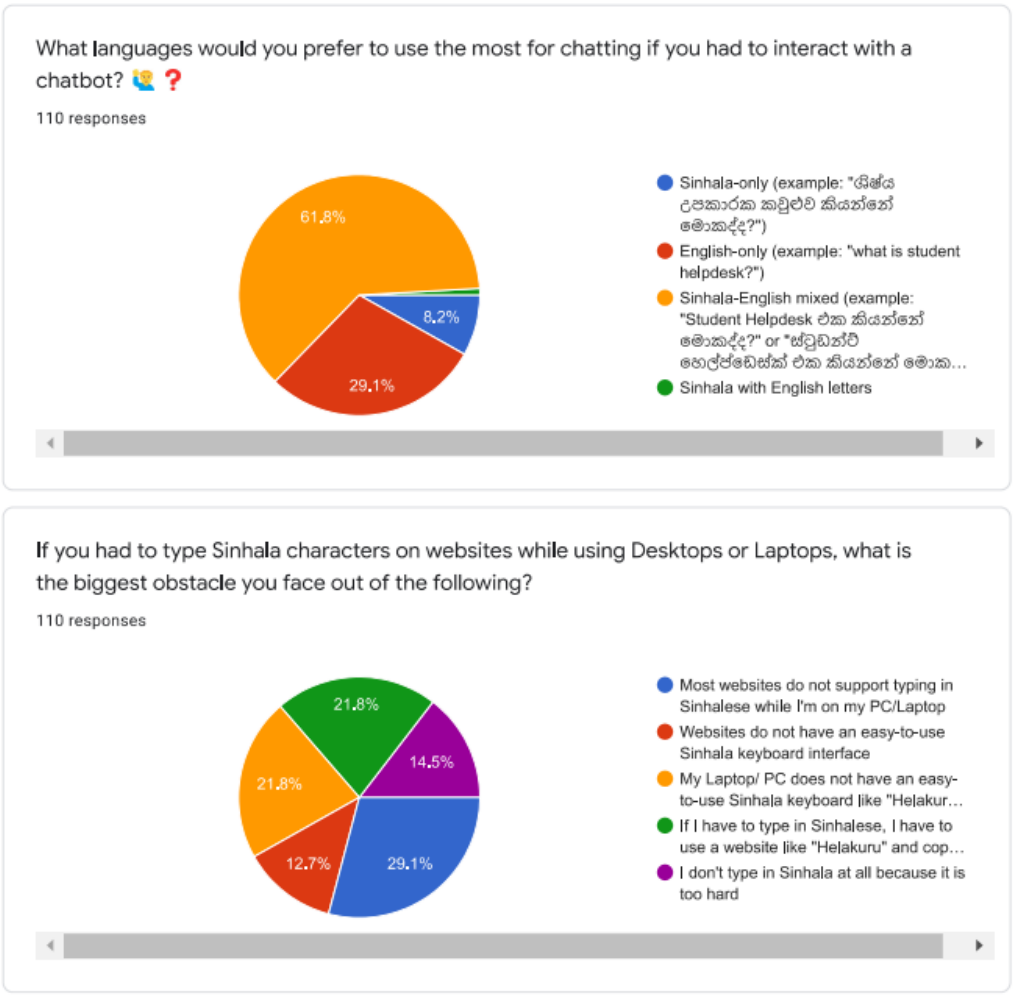


Figure A.3: Complete survey form questions and responses – part 3

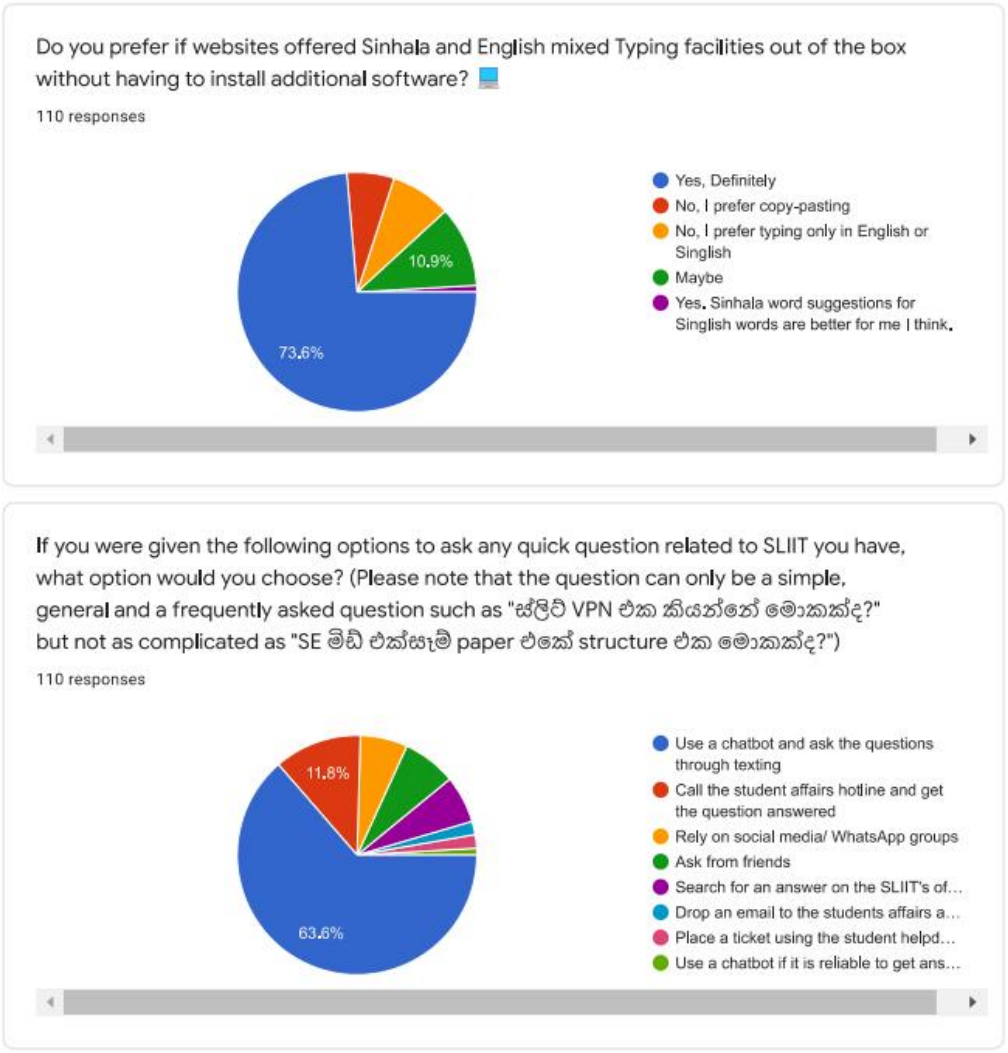


Figure A.4: Complete survey form questions and responses – part 4

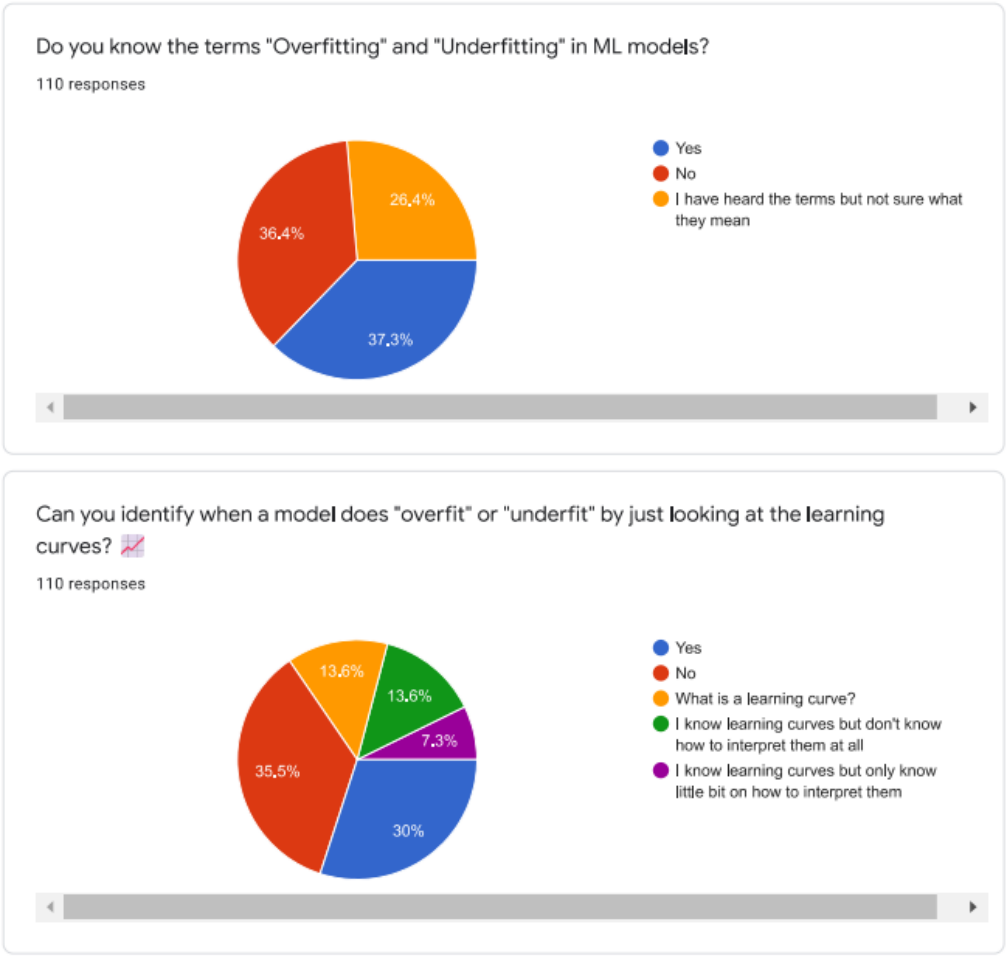
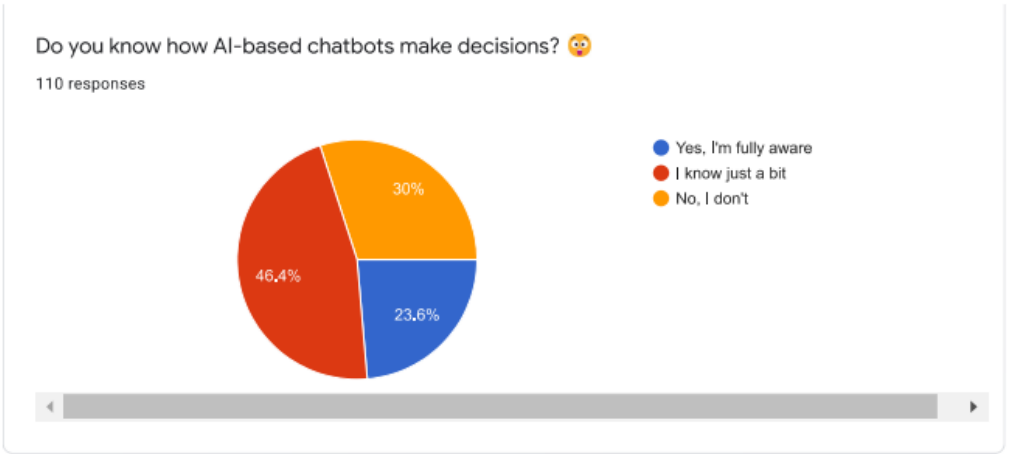


Figure A. 5: Complete survey form questions and responses – part 5



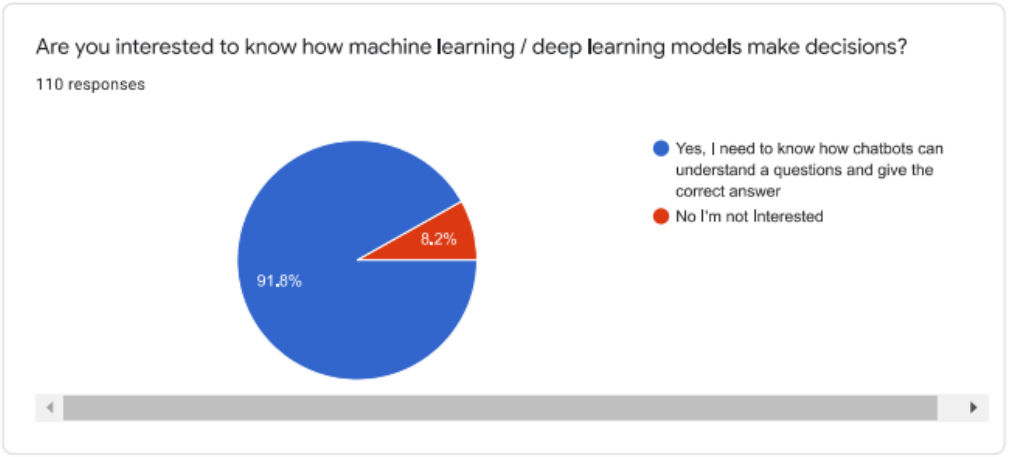


Figure A.6: Complete survey form questions and responses – part 6

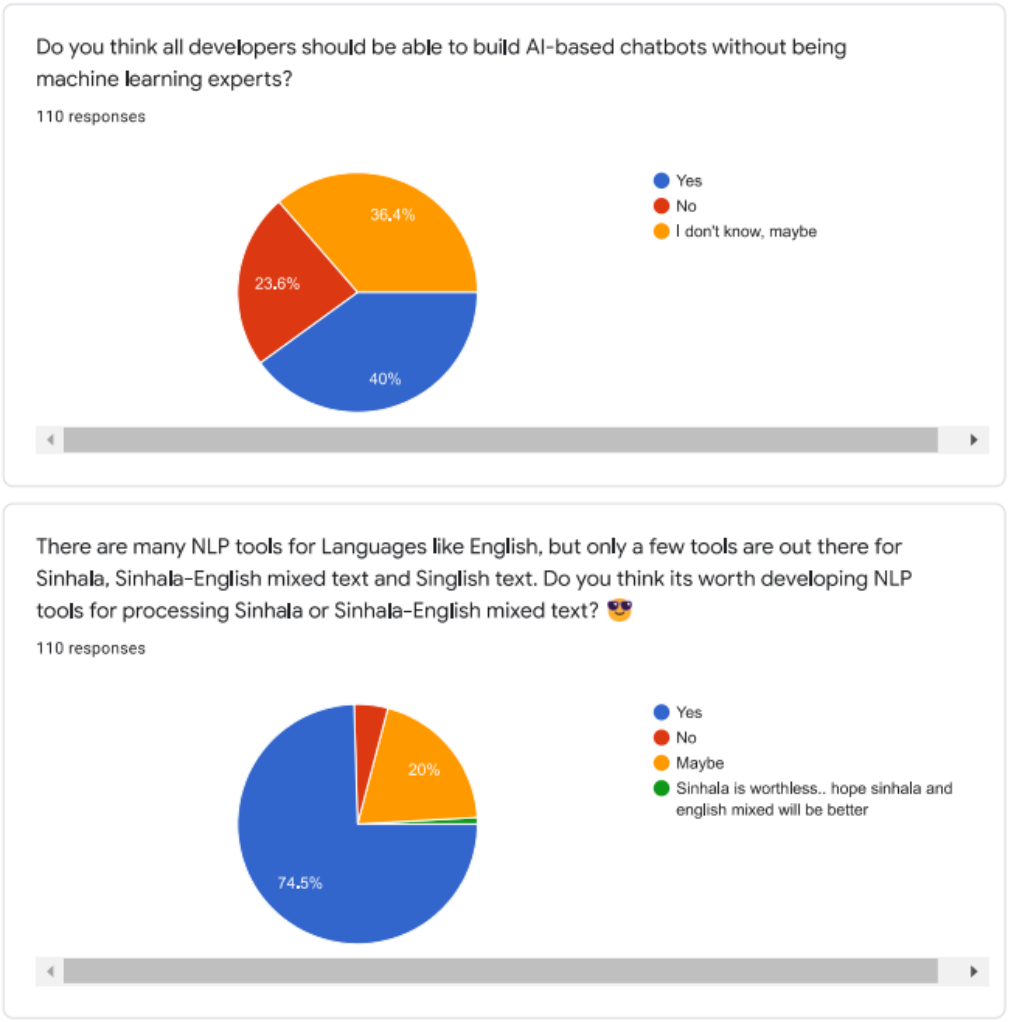


Figure A.7: Complete survey form questions and responses – part 7

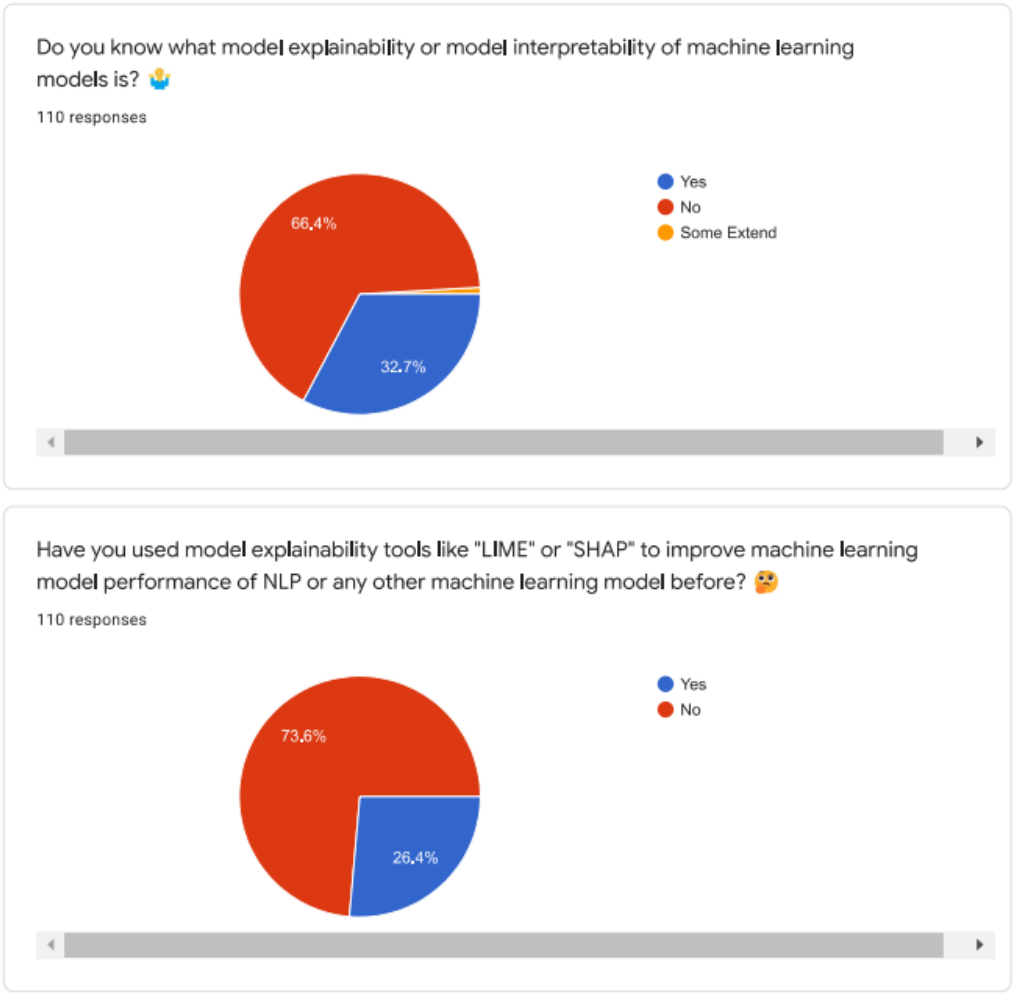


Figure A.8: Complete survey form questions and responses – part 8

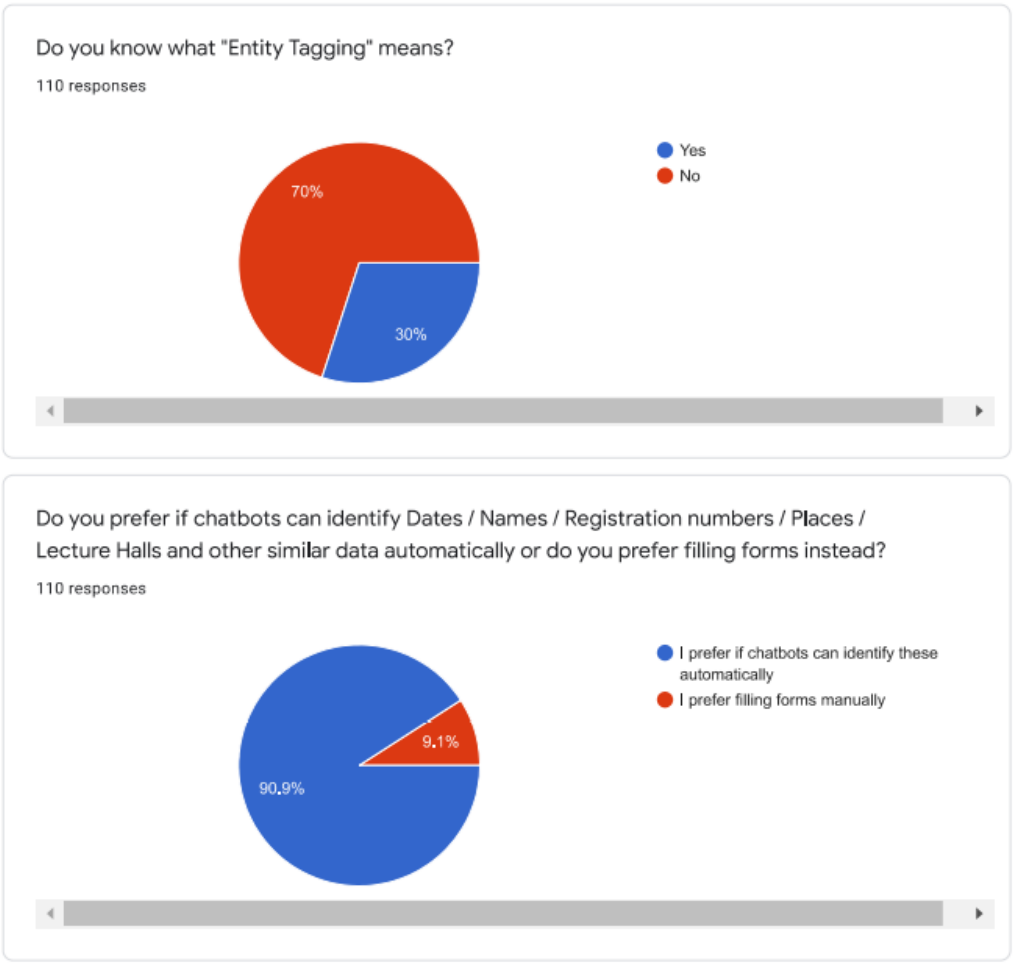


Figure A.9: Complete survey form questions and responses – part 9

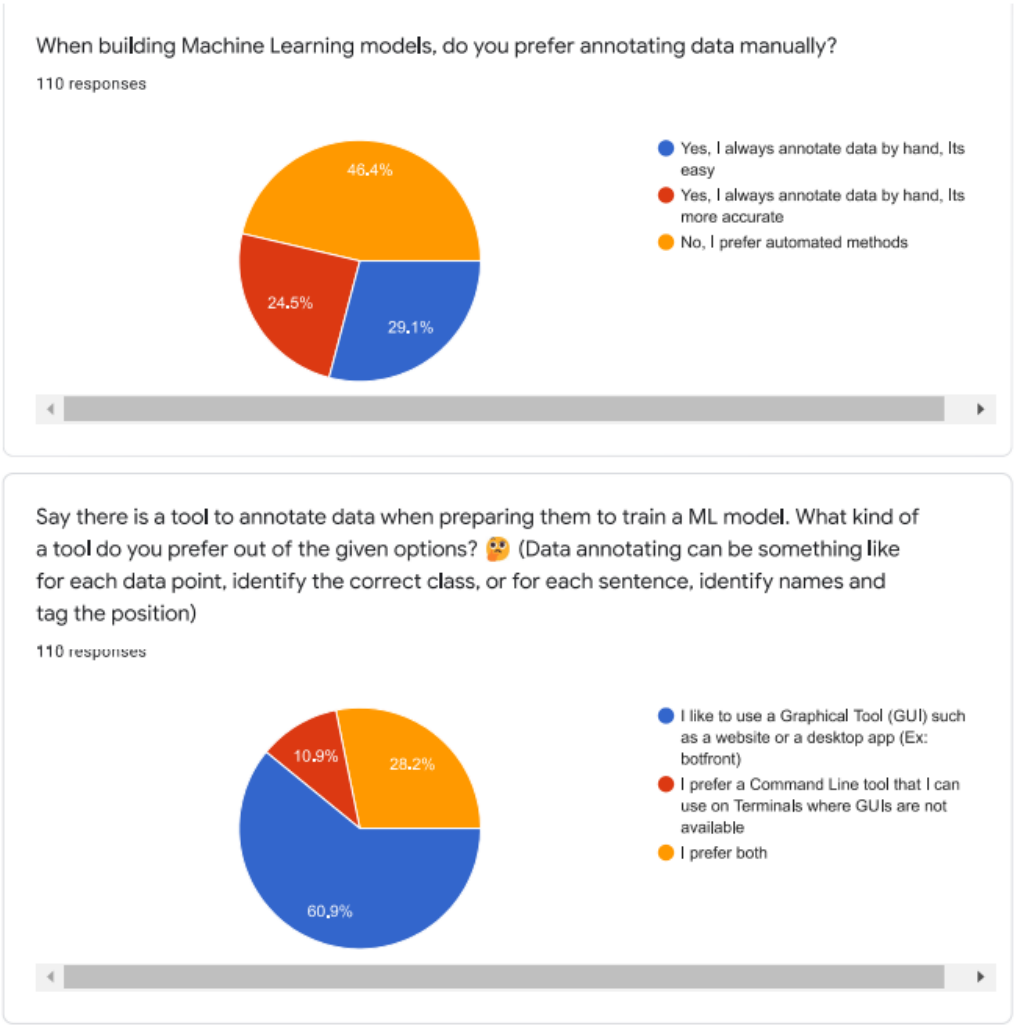


Figure A.10: Complete survey form questions and responses – part 10

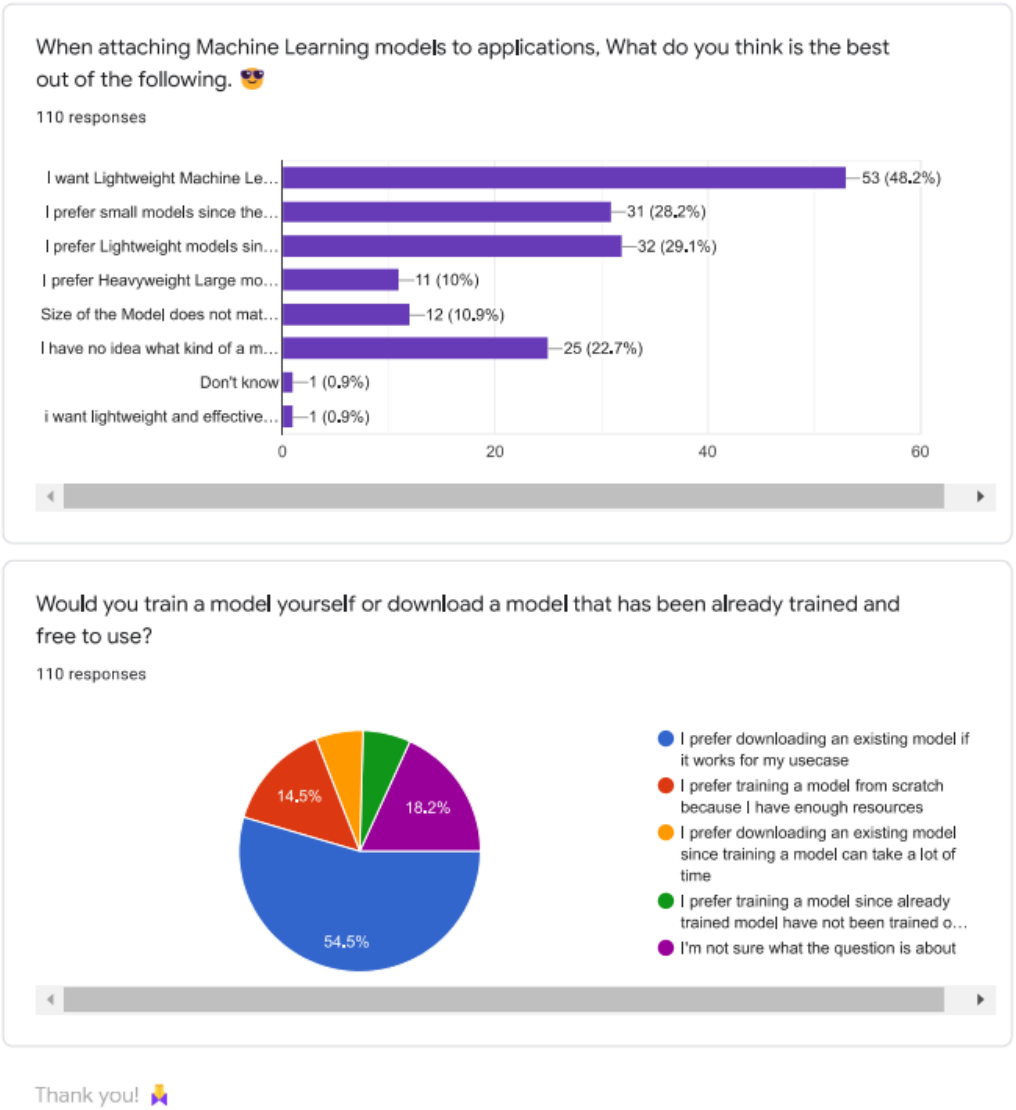


Figure A.11: Complete survey form questions and responses – part 11

Appendix B: Supervision Confirmation Emails

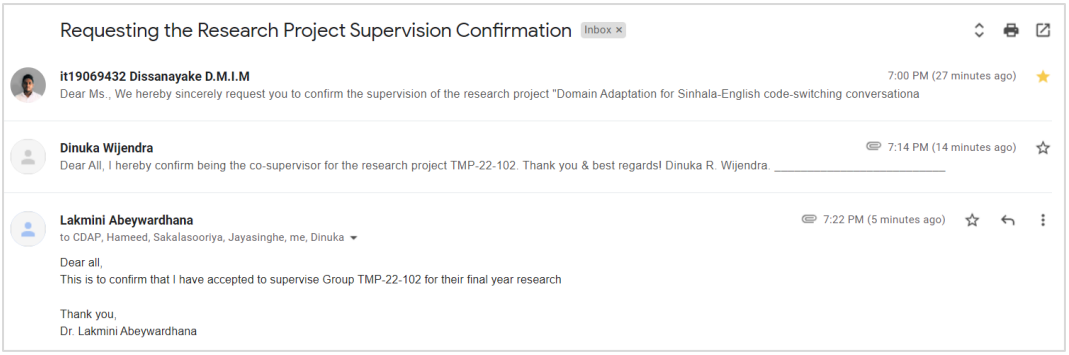


Figure B.1. Research project supervision confirmation email

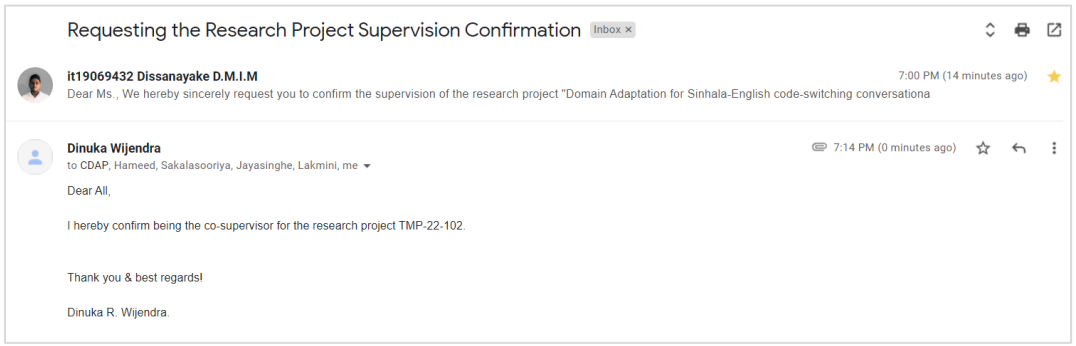


Figure B.2. Research project co-supervision confirmation email

Appendix C: Kolloqe User Interfaces

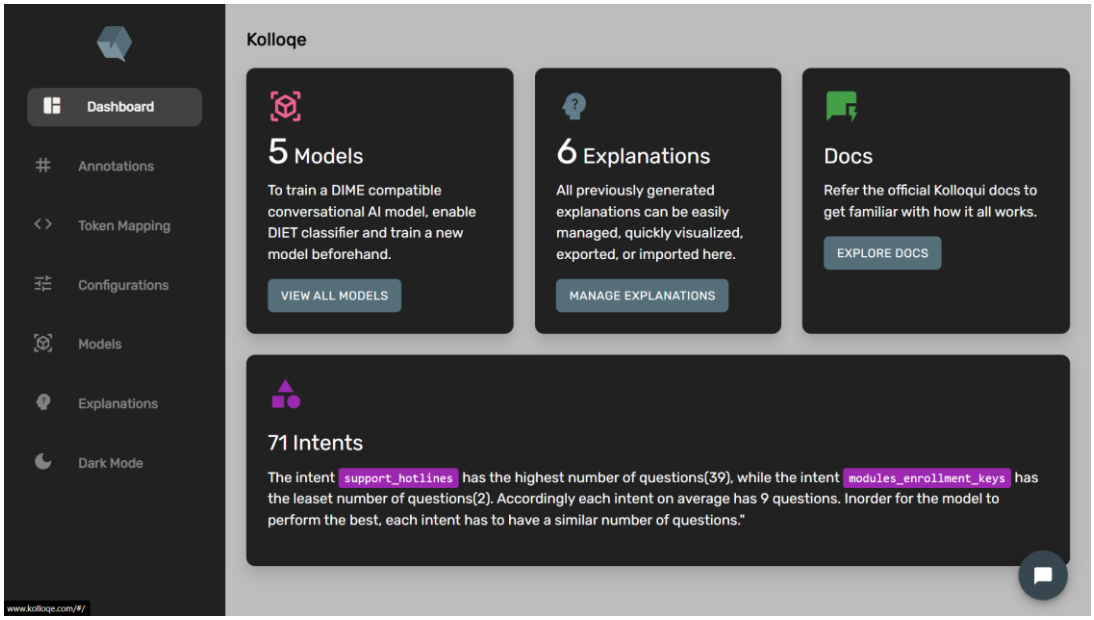


Figure C.1. SEETM dashboard – light mode

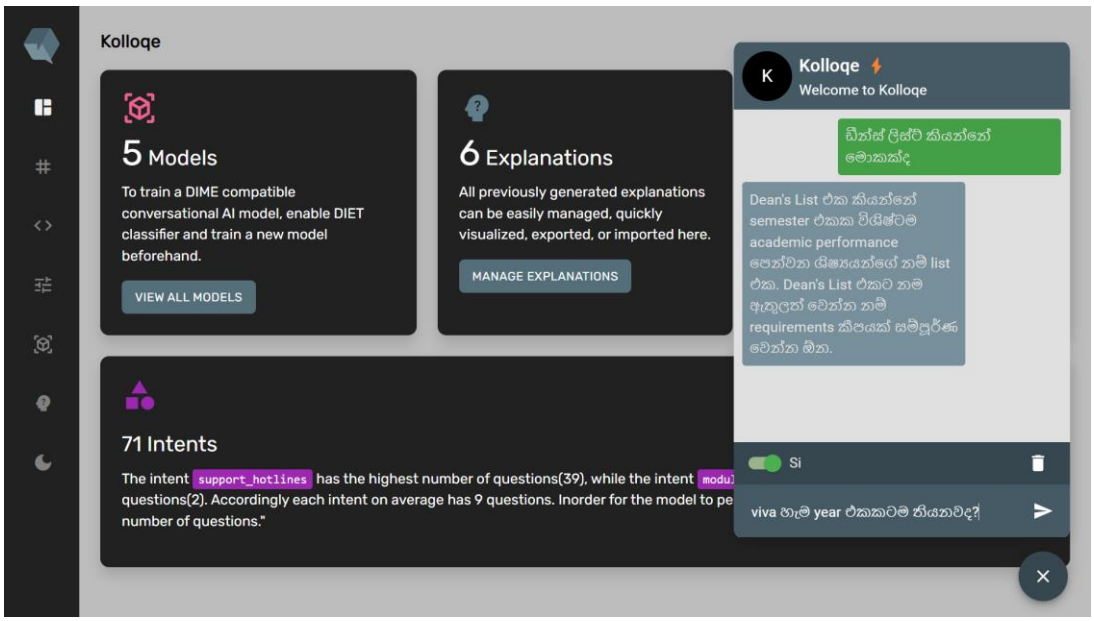


Figure C.2. SEETM dashboard – light mode

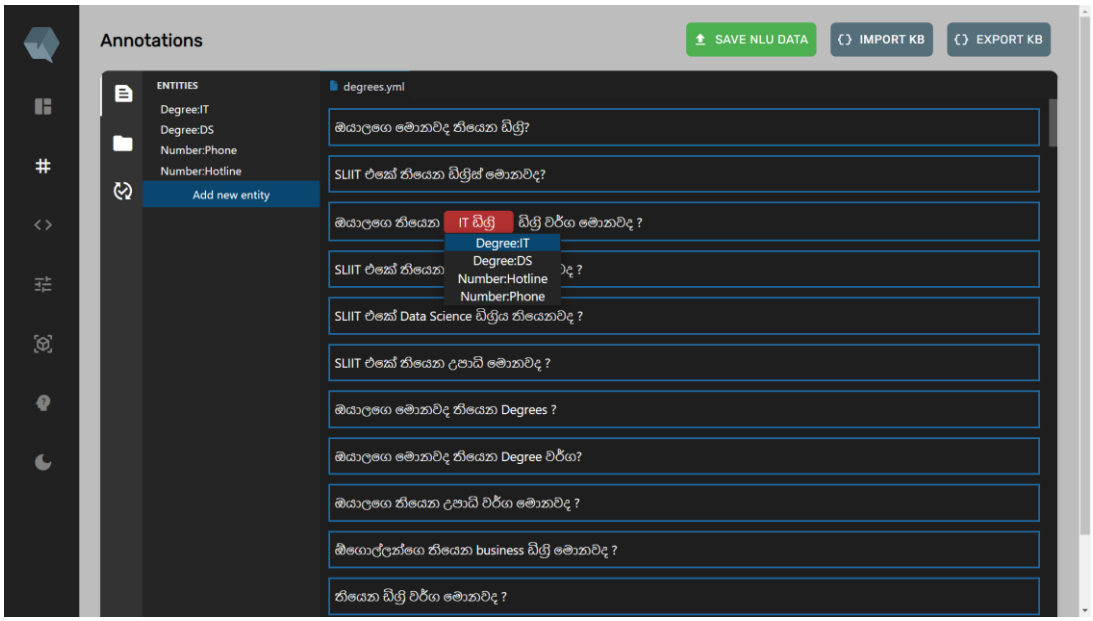
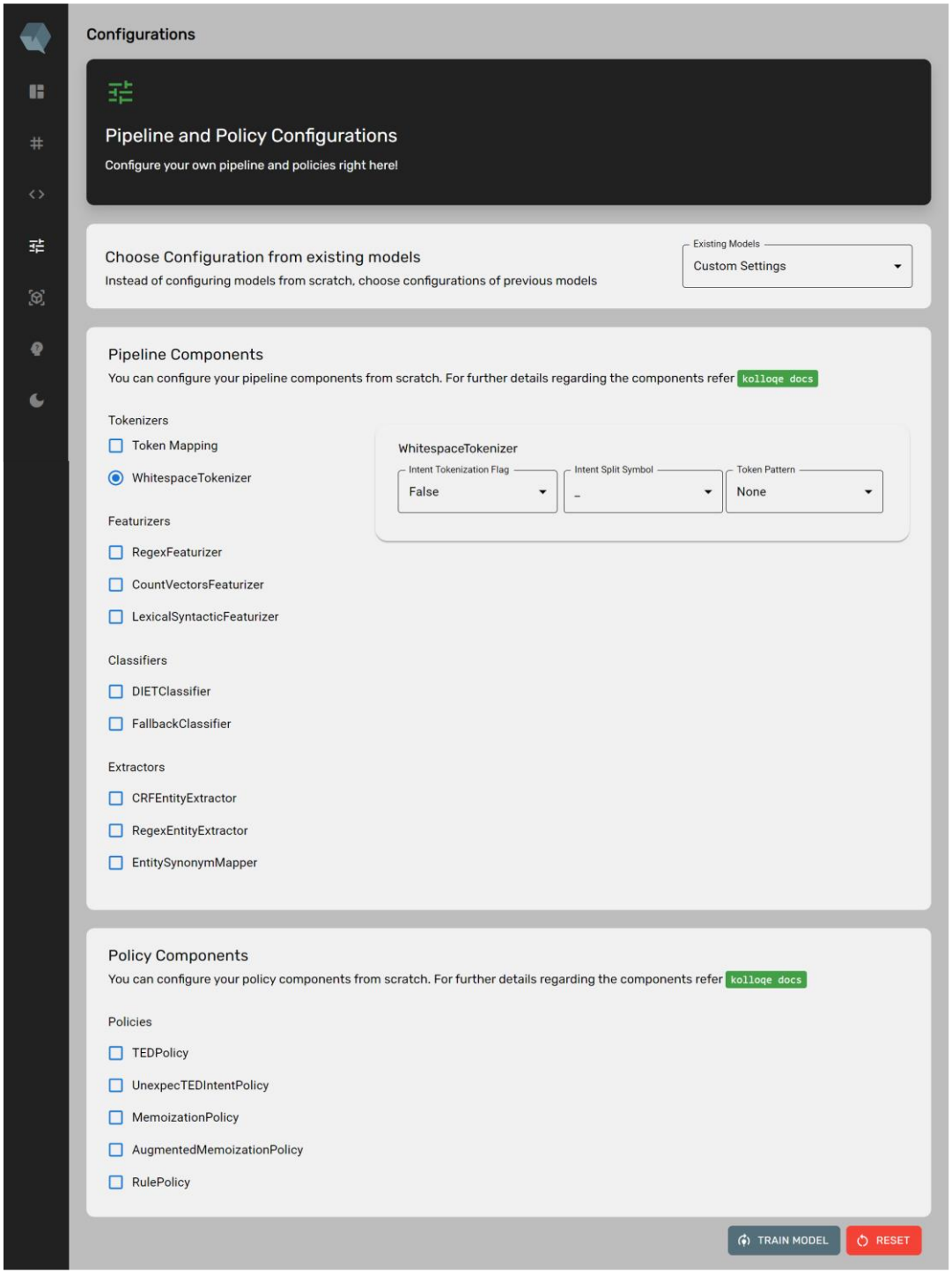


Figure C.3. SEETM dashboard – light mode


Configurations
Pipeline and Policy Configurations
Configure your own pipeline and policies right here!
Choose Configuration from existing models
Instead of configuring models from scratch, choose configurations of previous models
Existing Models
Custom Settings
Pipeline Components
You can configure your pipeline components from scratch. For further details regarding the components refer
kolloqe docs
Tokenizers
Token Mapping
WhitespaceTokenizer
Featurizers
RegexFeaturizer
CountVectorsFeaturizer
LexicalSyntacticFeaturizer
Classifiers
DIETClassifier
FallbackClassifier
Extractors
CRFEntityExtractor
RegexEntityExtractor
EntitySynonymMapper
WhitespaceTokenizer
Intent Tokenization Flag
False
Intent Split Symbol
_
Token Pattern
None
Policy Components
You can configure your policy components from scratch. For further details regarding the components refer
kolloqe docs
Policies
TEDPolicy
UnexpecTEDIntentPolicy
MemoizationPolicy
AugmentedMemoizationPolicy
RulePolicy
TRAIN MODEL
RESET

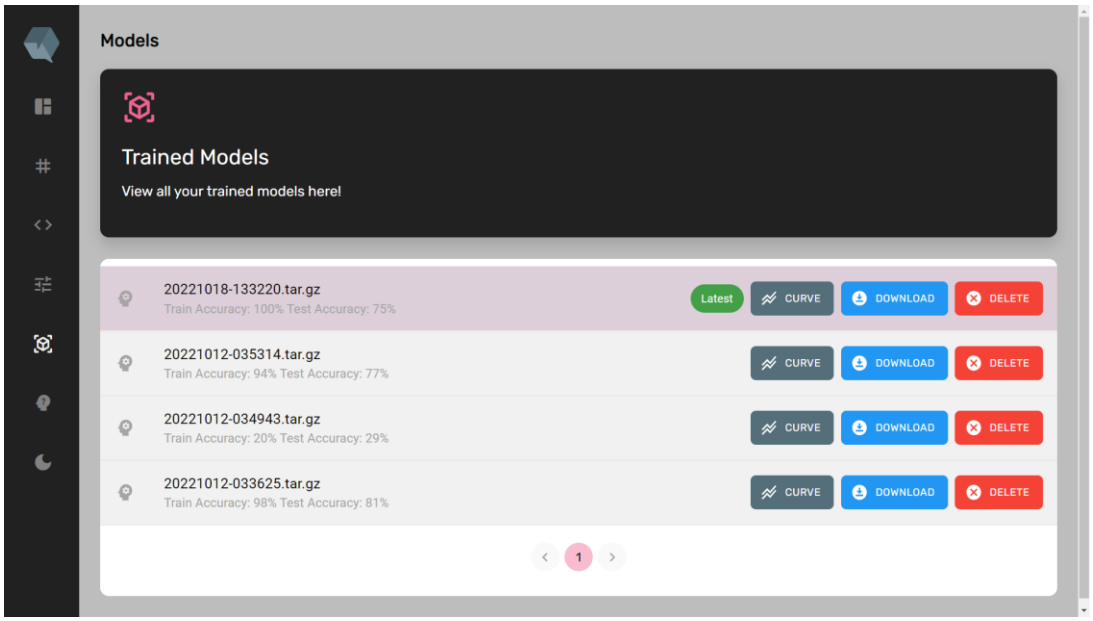


Figure C.6. SEETM dashboard – light mode

Figure C.7. SEETM dashboard – light mode

Figure C.8. SEETM dashboard – light mode

Figure C.9. SEETM dashboard – light mode

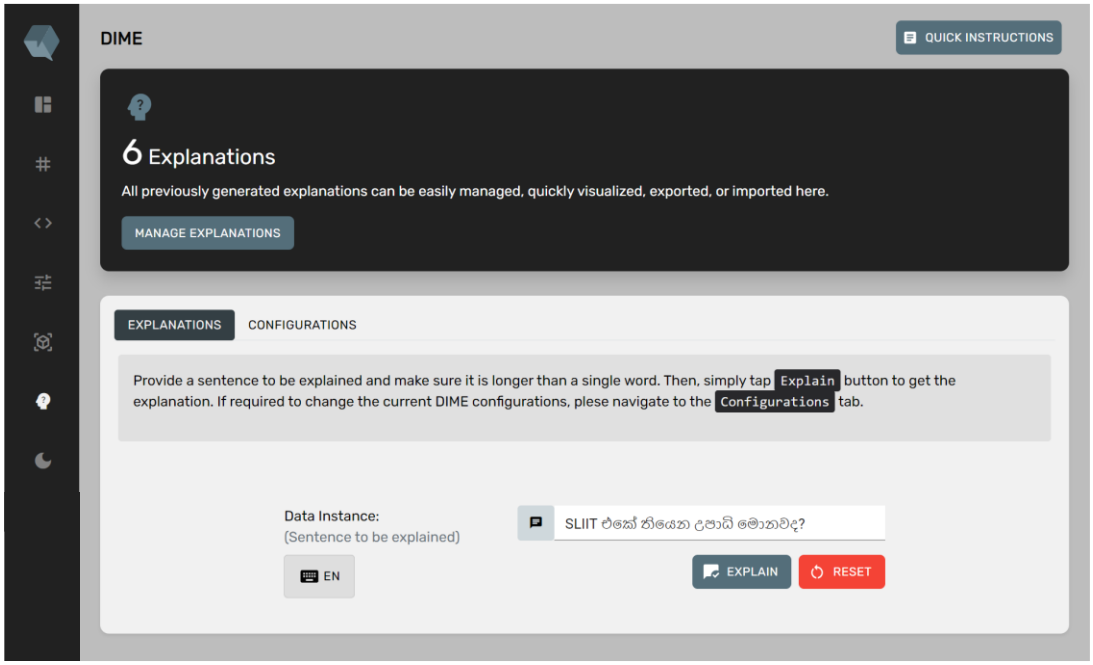


Figure C.10. SEETM dashboard – light mode

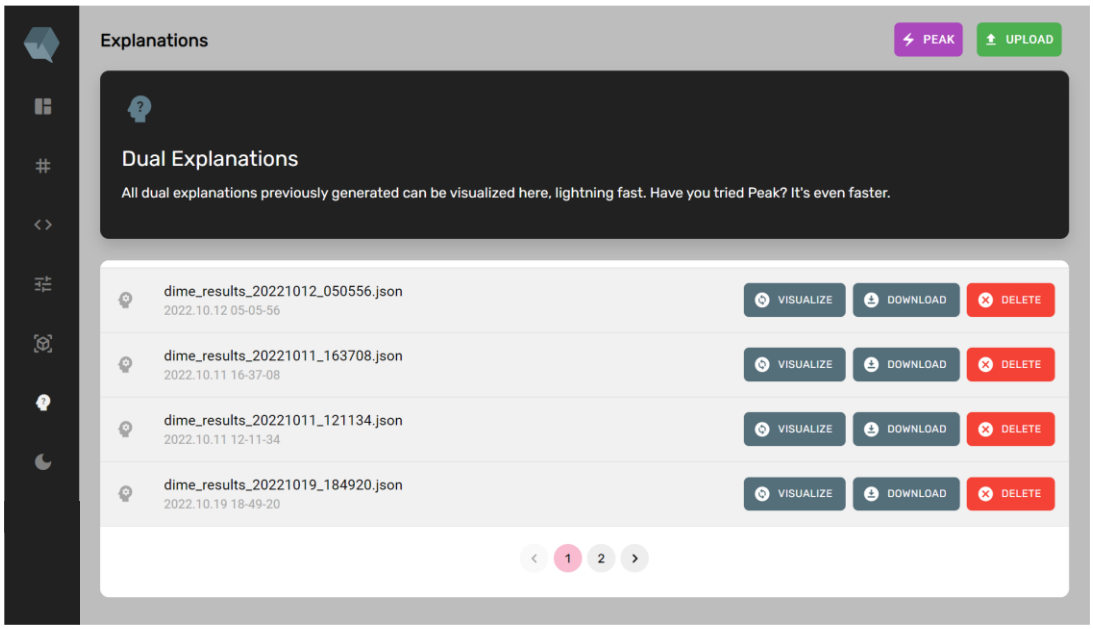


Figure C.11. SEETM dashboard – light mode

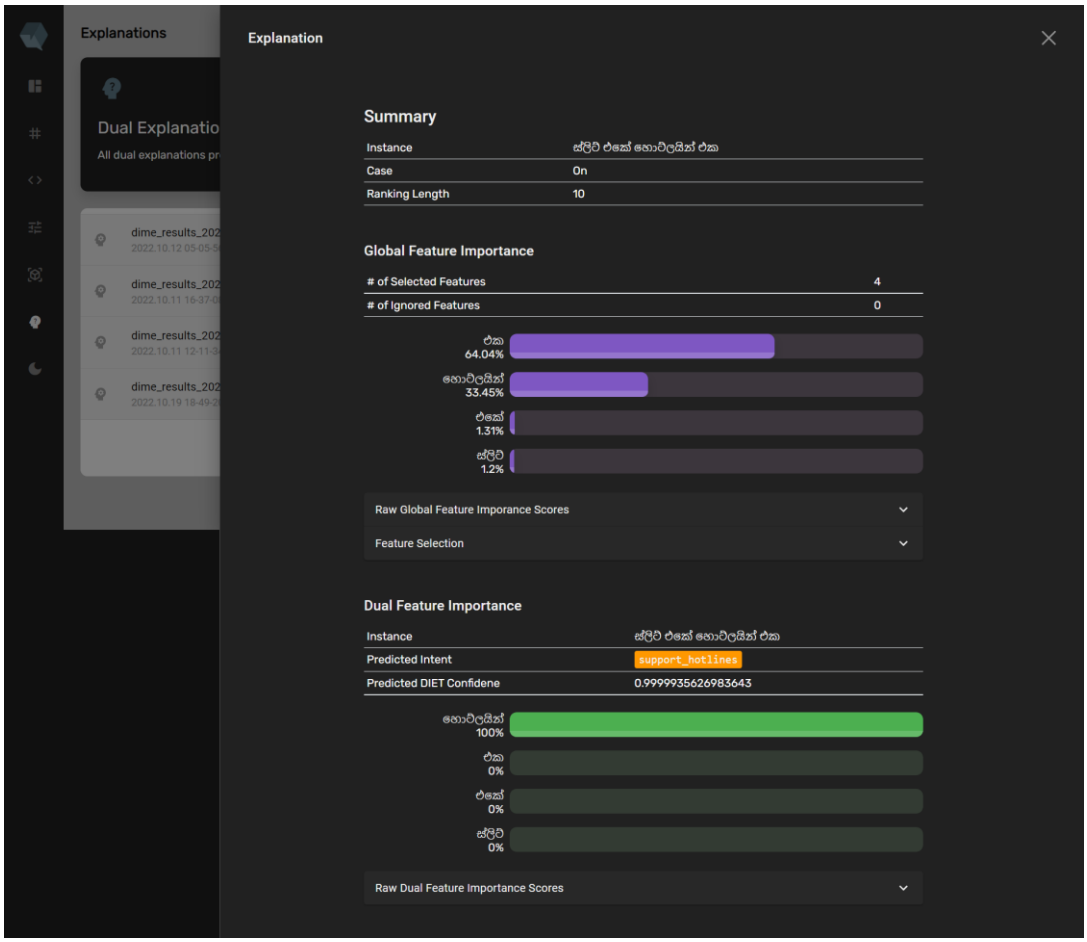


Figure C.12. SEETM dashboard – light mode

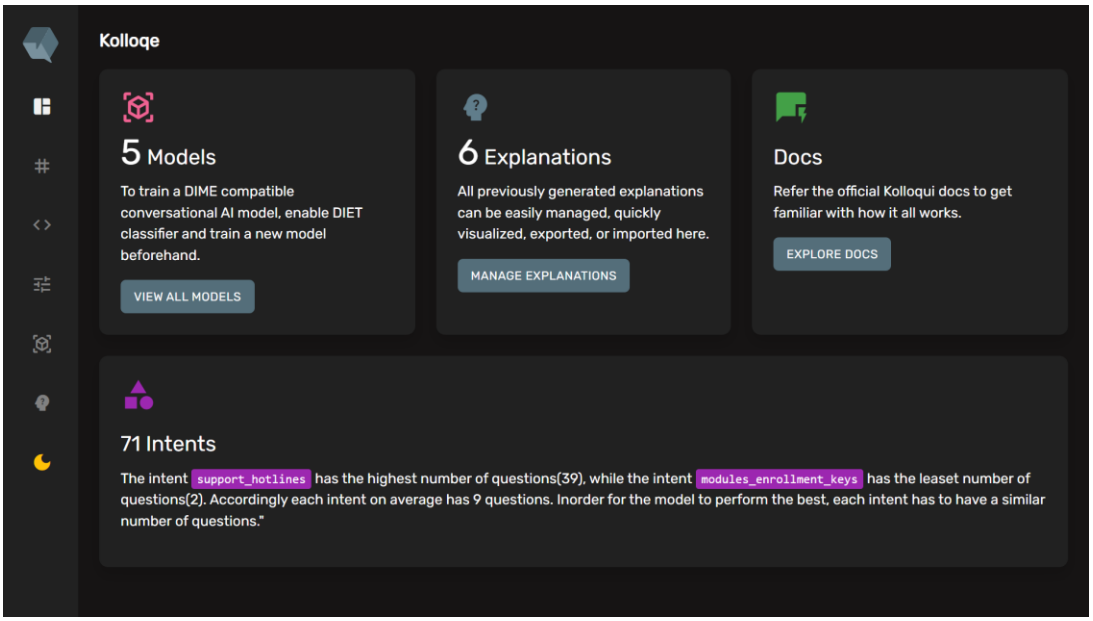


Figure C.13. SEETM dashboard – light mode

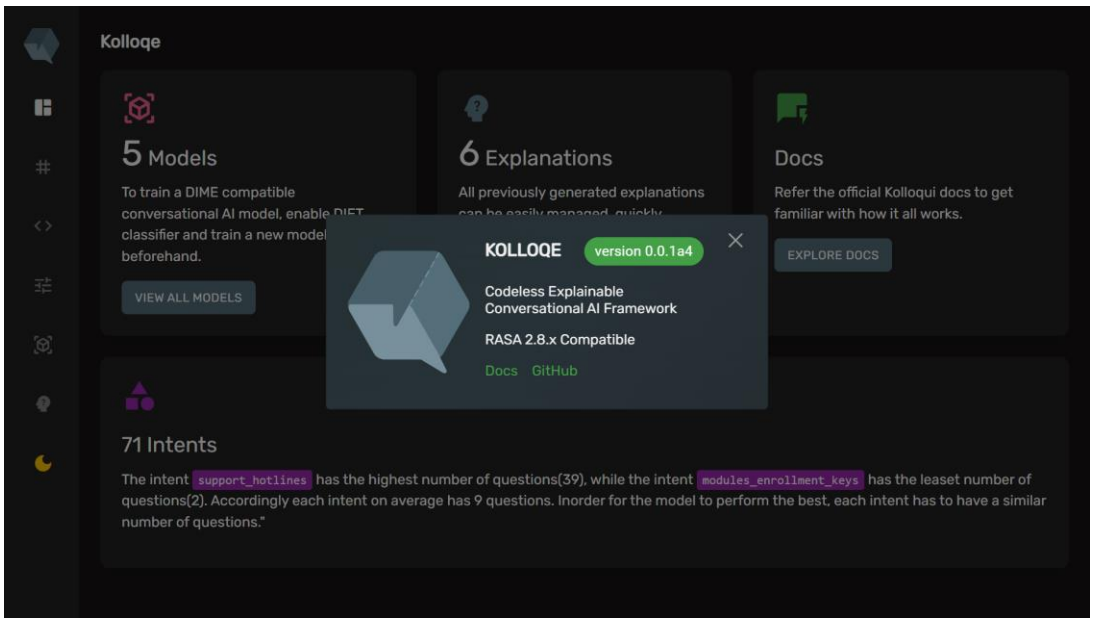


Figure C.14. SEETM dashboard – light mode