**ML PROJECT 2019-20**

**BY**

**Vedant Sahai (8364)**

**Elvis Dsouza (8333)**

* **Problem Statement: -**

Medical Analytica - a chatbot for behavioral therapy

* **Description: -**

The web-based application consists of two main users: firstly the patient who will be using this chatbot whenever they feel depressed and seek for music therapy as a treatment to overcome their illness and secondly the admin ( i.e. it can be a doctor or a consultant ) who will be able to see the emotion analysis of the patient on weekly basis.

* **Functionalities: -**
* Admin End----
* Three types of charts namely the line, pie and bar chart displayed based upon the patient chat history weekly to judge the emotion and psychological state of the patient.

* Patient End----
* Chatbot:
  + The main aspect of the whole application wherein the user will chat with the chatbot whenever he feels like it.
  + Chatbot Functionalities:
    - Tell Jokes
    - Tell Quotes overlaid on a picture
    - Tell the weather and temperature of any Indian cities
    - Have general conversations
    - Depending on your emotion (i.e. sad, angry or happy) play appropriate music from YouTube
    - In case of emergencies show nearby clinics, hospitals along with their address and other details (requires you to allow location permission)
* Additional Features:
  + A separate page to show all the hospitals in India and along with it the address and other details of the hospitals, clinics, nursing homes etc.
  + A button to save stories as if you require it to be saved.
* **Technology Stack:**
* Server Side: Python Flask
* Front End: HTML5, CSS3, JavaScript
* Chatbot: RASA Framework
* Operating System: Platform Independent
* **Algorithms & Framework:**
* RASA Framework---
* **Rasa** is an open-source machine learning framework for building [AI assistants and chatbots](http://blog.rasa.com/level-3-contextual-assistants-beyond-answering-simple-questions/). Mostly you don’t need any programming language experience to work in Rasa. Although there is something called “**Rasa** **Action Server**” where you need to write code in Python, that mainly used to trigger External actions like Calling Google API or REST API etc.
* Rasa has two main modules:
* **Rasa NLU** {for understanding user messages}

This is the place, where rasa tries to understand User messages to detect Intent and Entity in your message. Rasa NLU has different components for recognizing intents and entities, most of which have some additional dependencies.

* **Rasa Core** {for holding conversations and deciding what to do next}

This is the place, where Rasa try to help you with contextual message flow. Based on User message, it can predict dialogue as a reply and can trigger Rasa Action Server.

* Algorithms for Pipelining ---
* WhitespaceTokenizer:

Tokenizer using whitespaces as a separator

* Outputs

Tokens for user messages, responses (if present), and intents (if specified)

* Requires

Nothing

* Description

Creates a token for every whitespace-separated character sequence.

* RegexFeaturizer:

Creates a vector representation of user message using regular expressions.

* Outputs

sparse\_features for user messages and tokens. Pattern

* Requires

tokens

* Type

Sparse feature

* Description

Creates features for entity extraction and intent classification. During training, the RegexFeaturizer creates a list of regular expressions defined in the training data format. For each regex, a feature will be set marking whether this expression was found in the user message or not. All features will later be fed into an intent classifier/entity extractor to simplify classification (assuming the classifier has learned during the training phase, that this set feature indicates a certain intent/entity). Regex features for entity extraction are currently only supported by the [CRFEntityExtractor](https://rasa.com/docs/rasa/nlu/components/" \l "crfentityextractor) and the [DIETClassifier](https://rasa.com/docs/rasa/nlu/components/" \l "diet-classifier) components!

* LexicalSyntacticFeaturizer:

Creates lexical and syntactic features for a user message to support entity extraction.

* Outputs

sparse\_features for user messages

* Requires

tokens

* Type

Sparse featurizer

* Description

Creates features for entity extraction. Moves with a sliding window over every token in the user message and creates features according to the configuration (see below). As a default configuration is present, you don’t need to specify a configuration.

* Configuration

You can configure what kind of lexical and syntactic features the featurizer should extract. The following features are available:

Feature Name Description

BOS Checks if the token is at the beginning of the sentence.

EOS Checks if the token is at the end of the sentence.

low Checks if the token is lower case.

upper Checks if the token is upper case.

title Checks if the token starts with an uppercase character and all

remaining characters are lowercased.

digit Checks if the token contains just digits.

prefix5 Take the first five characters of the token.

prefix2 Take the first two characters of the token.

suffix5 Take the last five characters of the token.

suffix3 Take the last three characters of the token.

suffix2 Take the last two characters of the token.

suffix1 Take the last character of the token.

pos Take the Part-of-Speech tag of the token (``SpacyTokenizer`` required).

pos2 Take the first two characters of the Part-of-Speech tag of the token

* CountVectorsFeaturizer:

Creates bag-of-words representation of user messages, intents, and responses.

* Outputs

sparse\_features for user messages, intents, and responses

* Requires

tokens

* Type

Sparse featurizer

* Description

Creates features for intent classification and response selection. Creates bag-of-words representation of user message, intent, and response using [sklearn’s CountVectorizer](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html). All tokens which consist only of digits (e.g. 123 and 99 but not a123d) will be assigned to the same feature.

* DIETClassifier:

Dual Intent Entity Transformer (DIET) used for intent classification and entity extraction

* Description

DIET (Dual Intent and Entity Transformer) is a multi-task architecture for intent classification and entity recognition. The architecture is based on a transformer which is shared for both tasks. A sequence of entity labels is predicted through a Conditional Random Field (CRF) tagging layer on top of the transformer output sequence corresponding to the input sequence of tokens. For the intent labels, the transformer output for the \_\_CLS\_\_ token and intent labels are embedded into a single semantic vector space. We use the dot-product loss to maximize the similarity with the target label and minimize similarities with negative samples.

* Configuration

If you want to use the DIETClassifier just for intent classification, set entity\_recognition to False. If you want to do only entity recognition, set intent\_classification to False. By default, DIETClassifier does both, i.e. entity\_recognition and intent\_classification is set to True.

You can define several hyperparameters to adapt to the model. If you want to adapt your model, start by modifying the following parameters:

epochs: This parameter sets the number of times the algorithm will see the training data (default: 300). One epoch is equalled to one forward pass and one backward pass of all the training examples. Sometimes the model needs more epochs to properly learn. Sometimes more epochs don’t influence the performance. The lower the number of epochs the faster the model is trained.

hidden\_layers\_sizes: This parameter allows you to define the number of feed-forward layers and their output dimensions for user messages and intents (default: text: [], label: []). Every entry in the list corresponds to a feed-forward layer. For example, if you set the text: [256, 128], we will add two feed-forward layers in front of the transformer. The vectors of the input tokens (coming from the user message) will be passed on to those layers. The first layer will have an output dimension of 256 and the second layer will have an output dimension of 128. If an empty list is used (default behaviour), no feed-forward layer will be added. Make sure to use only positive integer values. Usually, numbers of power of two are used. Also, it is usual practice to have decreasing values in the list: the next value is smaller or equal to the value before.

embedding\_dimension: This parameter defines the output dimension of the embedding layers used inside the model (default: 20). We are using multiple embeddings layers inside the model architecture. For example, the vector of the \_\_CLS\_\_ token and the intent is passed on to an embedding layer before they are compared and the loss is calculated.

number\_of\_transformer\_layers: This parameter sets the number of transformer layers to use (default: 2). The number of transformer layers corresponds to the transformer blocks to use for the model.

transformer\_size: This parameter sets the number of units in the transformer (default: 256). The vectors coming out of the transformers will have the given transformer\_size.

weight\_sparsity: This parameter defines the fraction of kernel weights that are set to 0 for all feed-forward layers in the model (default: 0.8). The value should be between 0 and 1. If you set weight\_sparsity to 0, no kernel weights will be set to 0, the layer acts as a standard feed-forward layer. You should not set weight\_sparsity to 1 as this would result in all kernel weights being 0, i.e. the model is not able to learn.

The above configuration parameters are the ones you should configure to fit your model to your data. However, additional parameters exist that can be adapted.

* MemoizationPolicy:

The MemoizationPolicy just memorizes the conversations in your training data. It predicts the next action with confidence 1.0 if this exact conversation exists in the training data, otherwise, it predicts None with confidence 0.0.

* TEDPolicy:

This policy has a pre-defined architecture, which comprises the following steps:

* Concatenate user input (user intent and entities), previous system actions, slots and active forms for each time step into an input vector to pre-transformer embedding layer;
* Feed it to the transformer;
* Apply a dense layer to the output of the transformer to get embeddings of dialogue for each time step;
* Apply a dense layer to create embeddings for system actions for each time step;
* Calculate the similarity between the dialogue embedding and embedded system actions. This step is based on the [StarSpace](https://arxiv.org/abs/1709.03856) idea.
* It is recommended to use state\_featurizer=LabelTokenizerSingleStateFeaturizer(...) (see [Featurization of Conversations](https://rasa.com/docs/rasa/api/core-featurization/#featurization-conversations) for details).
* Configuration

Configuration parameters can be passed as parameters to the TEDPolicy within the configuration file. If you want to adapt your model, start by modifying the following parameters:

epochs: This parameter sets the number of times the algorithm will see the training data (default: 1). One epoch is equalled to one forward pass and one backward pass of all the training examples. Sometimes the model needs more epochs to properly learn. Sometimes more epochs don’t influence the performance. The lower the number of epochs the faster the model is trained.

hidden\_layers\_sizes: This parameter allows you to define the number of feed-forward layers and their output dimensions for dialogues and intents (default: dialogue: [], label: []). Every entry in the list corresponds to a feed-forward layer. For example, if you set dialogue: [256, 128], we will add two feed-forward layers in front of the transformer. The vectors of the input tokens (coming from the dialogue) will be passed on to those layers. The first layer will have an output dimension of 256 and the second layer will have an output dimension of 128. If an empty list is used (default behaviour), no feed-forward layer will be added. Make sure to use only positive integer values. Usually, numbers of power of two are used. Also, it is usual practice to have decreasing values in the list: the next value is smaller or equal to the value before.

number\_of\_transformer\_layers: This parameter sets the number of transformer layers to use (default: 1). The number of transformer layers corresponds to the transformer blocks to use for the model.

transformer\_size: This parameter sets the number of units in the transformer (default: 128). The vectors coming out of the transformers will have the given transformer\_size.

weight\_sparsity: This parameter defines the fraction of kernel weights that are set to 0 for all feed-forward layers in the model (default: 0.8). The value should be between 0 and 1. If you set weight\_sparsity to 0, no kernel weights will be set to 0, the layer acts as a standard feed-forward layer. You should not set weight\_sparsity to 1 as this would result in all kernel weights being 0, i.e. the model is not able to learn.

* **Outputs: -**

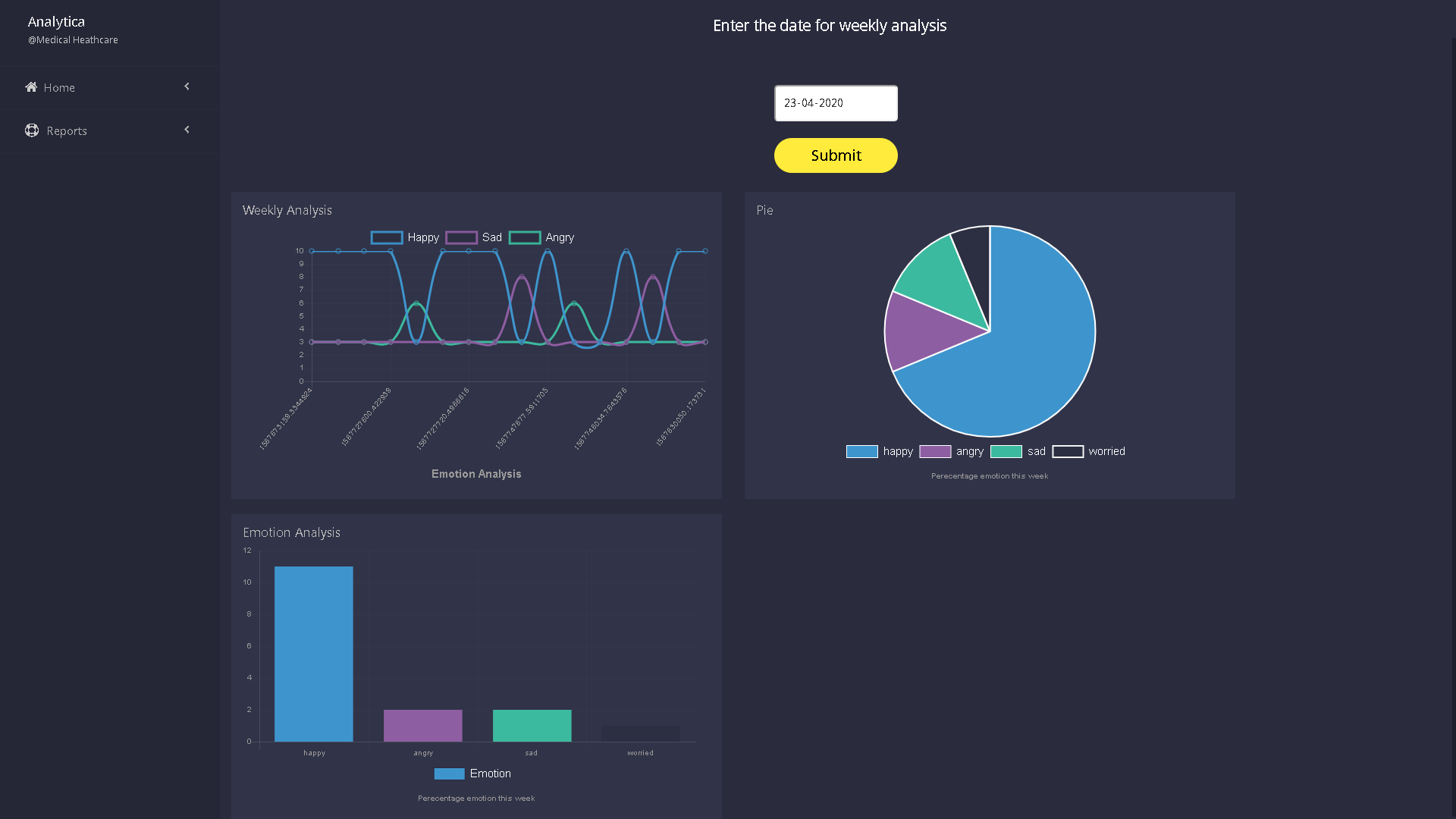


Fig 1- Emotion Analysis of a Patient



Fig 2- Chatbot replies a joke

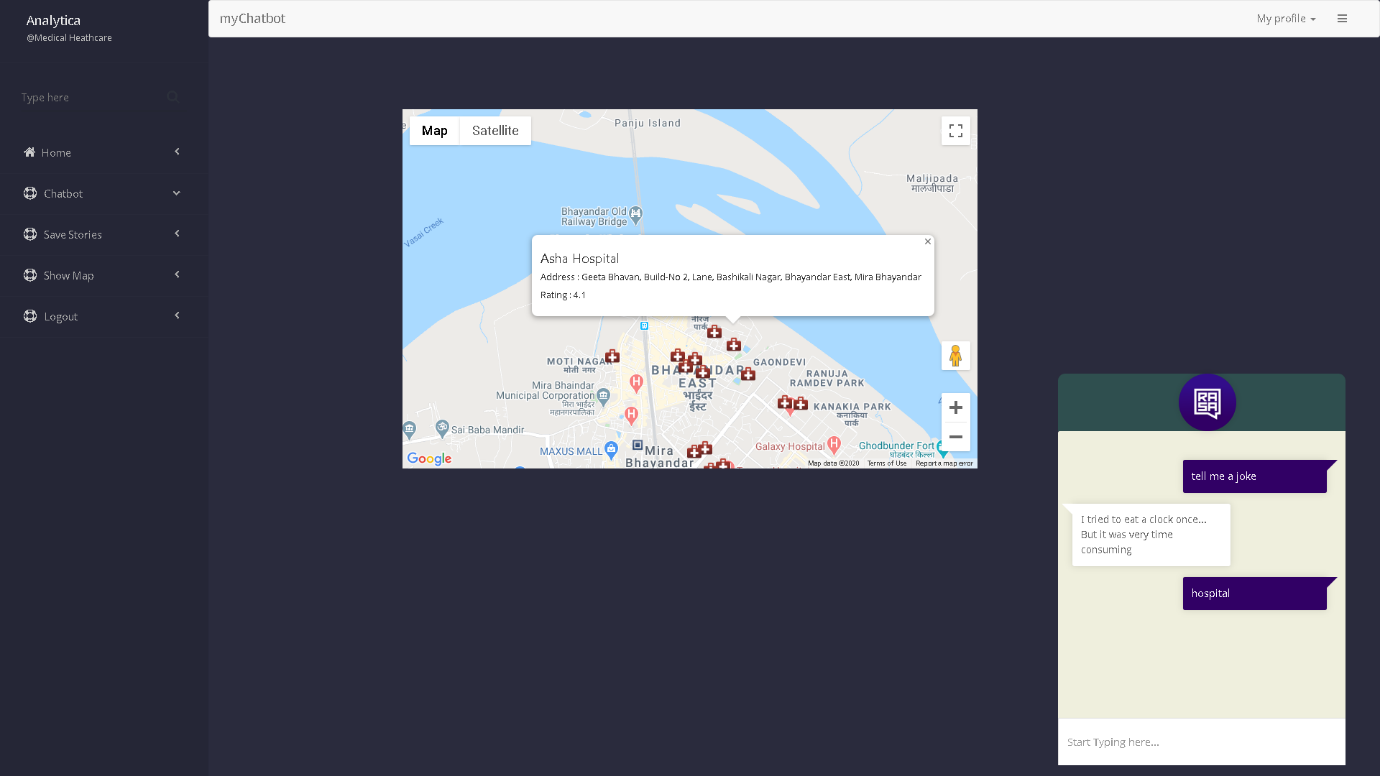


Fig 3- Chatbot replies nearby hospitals

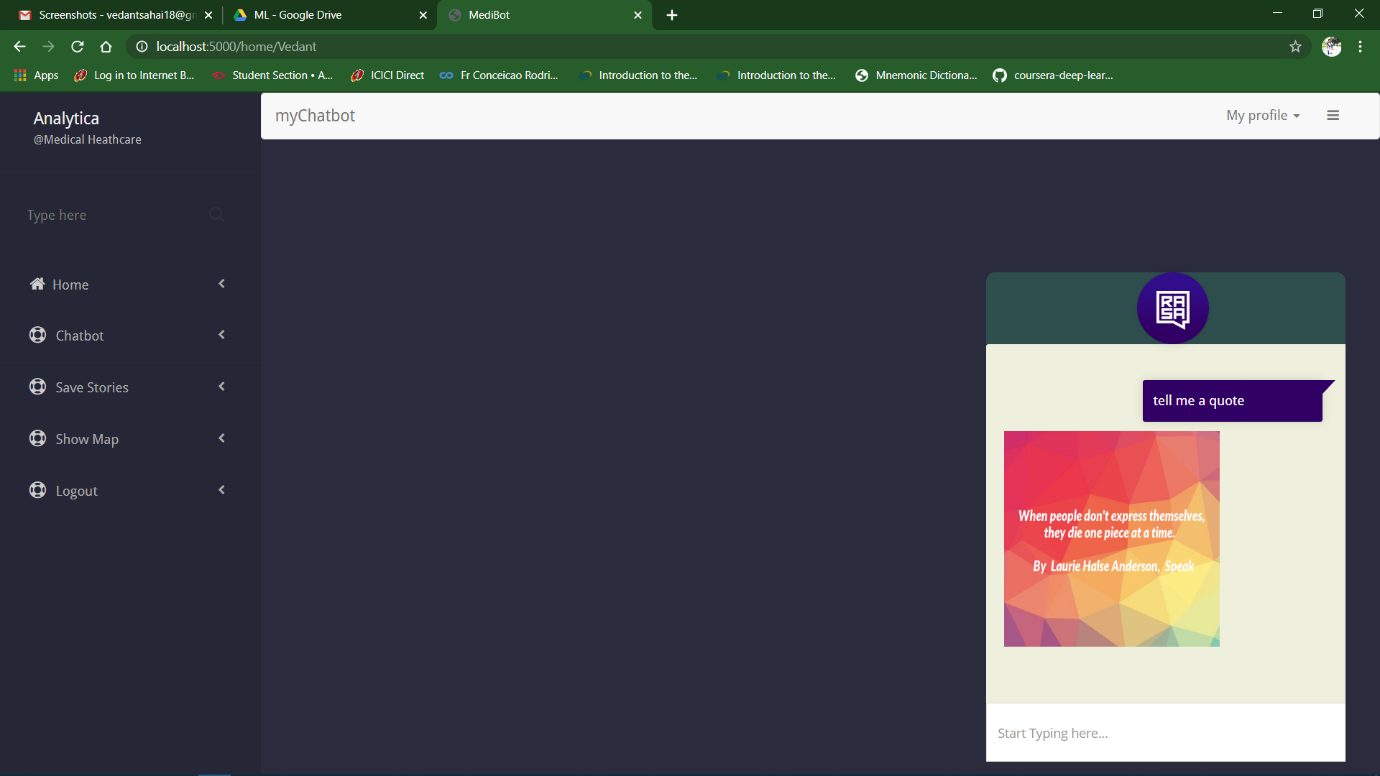


Fig 4- Chatbot replies a quote

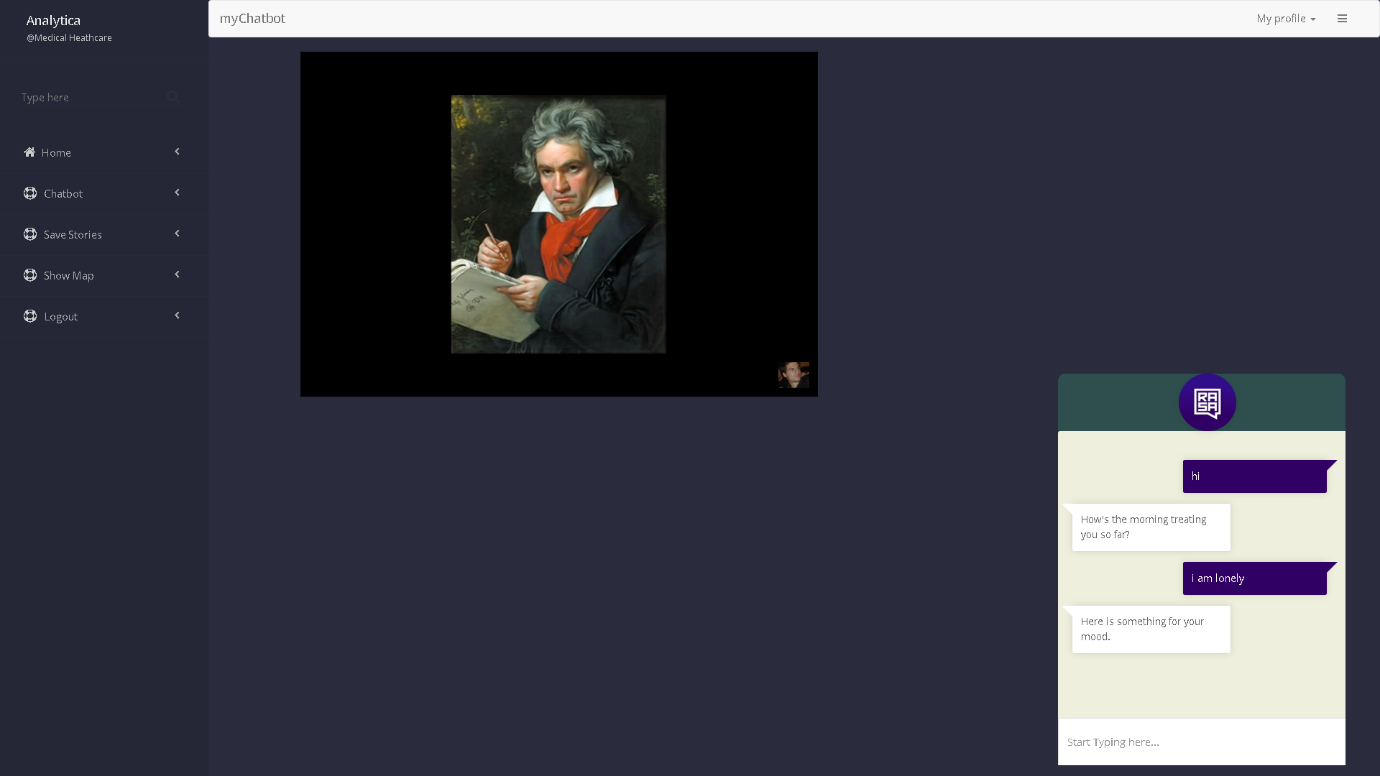


Fig 5- Chatbot replies music videos as per your emotion