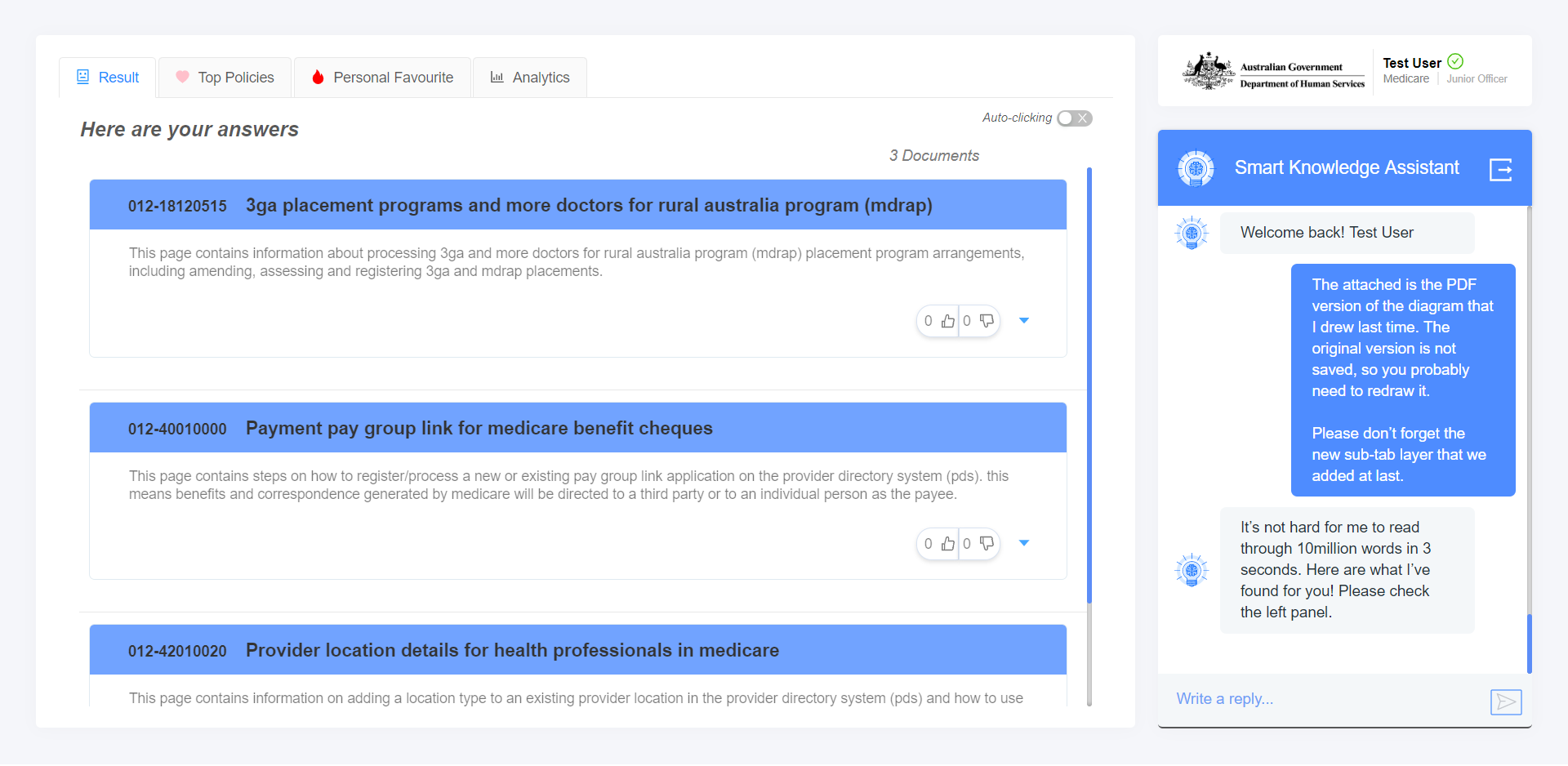
# DHS POC Chatbot Documentation

## Preview:



## Technologies:

Front-end: React JS, SASS, CSS, HTML, ANT Design

Back-end: Python, Flask, elasticsearch

Machine Learning: AllenNLP, Infersent, Bert

(Model trained on TelsaV100 GPU with a 4 CPU, 24G Memory instance on Google Cloud Platform)

## Architecture:



## How to run:

1. Get access to https://kpmgcomau.visualstudio.com/KPMGAU-IDD-DHS%20Policy%20Bot/\_git/KPMGAU-IDD-DHS%20Policy%20Bot
2. git clone https://kpmgcomau.visualstudio.com/KPMGAU-IDD-DHS%20Policy%20Bot/\_git/KPMGAU-IDD-DHS%20Policy%20Bot
3. create new python environment
4. pip install –r requiremnts.txt (if there is an error, please try to install it independently)
5. cd @frontend
6. install nodejs
7. npm install
8. npm start
9. open a new CMD window and go to the root directory
10. cd @backend
11. start an elasticsearch engine, which you can download from <https://www.elastic.co/downloads/elasticsearch>
12. use es\_load\_index\_json to ingest data into elasticsearch engine
13. install infersent model from <https://github.com/facebookresearch/InferSent>, and save the model into sentence \_similarity folder
14. Download Glovec from <https://nlp.stanford.edu/projects/glove/> and save it into sentence \_similarity/GloVec folder
15. Unzip pytorch.model.zip in bert\_asset folder
16. In @backend folder, run python main.py
17. go to <http://127.0.0.1:8222> (account: [test@test.com](mailto:test@test.com) password:12345678)

## Overview

The smart knowledge agent consists of a series of natural language processing tasks to obtain the document (from a context-specific corpus) that is most likely to contain the answer to the user’s question, as well as the span of text within that document that contains the answer to the question. For each question asked, the app returns the top three documents, along with the tab most likely to contain the answer and the best answer for the question within that tab. When double-clicked, these documents open in the interface. Additionally, there is a template for search analytics and user history available.

## Back end

The back end is made up of five key steps:

1. FAQ matching
2. Document selection
3. Document to tab refinement
4. Tab to sub-tab mapping
5. Machine comprehension

## FAQ matching

There is a small collection of frequently asked questions hard-coded into the back end, with the correct document, tab and answer. If the question asked has a sentence similarity greater than 0.9 with one of these FAQs (based on the InferSent similarity algorithm used in this component), the correct answer for the question with the highest similarity will be returned. If this occurs, the remaining four steps are not required.

InferSent is a sentence embeddings method that provides semantic representations for English sentences, produced by Facebook research. It is trained on natural language inference data and generalizes well to many different tasks. The repo and instructions for installation can be found here: <https://github.com/facebookresearch/InferSent>. GloVe vectors (840B tokens, 300d vectors) are the chosen word vectors. The similarity between two vector representations of sentences is computed using cosine similarity (cosine\_similarity from sklearn).

## Document selection

Elasticsearch is used to retrieve the top three documents that are most likely to answer the question, leveraging named entity recognition and keywords to construct matches. The list of dictionary keywords and Allen keywords sent from the Allen predictor will be split into the phrase and word types. Then the DSL generator composes query pieces for phrases and words separately by using different methods. According to Elastic Search syntax, it uses span multi and span near query with slop and in order parameters for phrases. It uses the bool query to compose query pieces of all keywords and phrases. To control the fuzzy match, it uses a fuzzy query with fuzziness and boosts parameters. In the end, the generator composes a long and compound query that includes all keywords and phrases to feed elastic search.

The DSL generator class also includes a parameter of minimum score that allows the user to put a threshold on the minimum score of return results. In our design, we put a minimum score as 0, but we limit the count of return results to be three so that only the top 3 matched results will be returned.

## Document to tab refinement

Each document is divided into 2-5 tabs – background, process summary, process, references and resources. There is a JSON called ‘question\_tab\_mapping.json’, which contains a list of ~700 sample questions, and the tab (within its relevant document) that contains the answer. Vector representations for each of the questions in this JSON and the question asked by the user are constructed using *InferSent*. Cosine similarity is then used to compute similarities, and subsequently, the tab for the best-matched question is returned.

## Tab to sub-tab refinement

Each tab can be divided into multiple sub-tabs. The sub-tab headings (and content) are contained in the JSON called ‘policies.json’. The sub-tab headings for the document and tab (from the above steps) are extracted for the question. The question and sub-tab heading strings are tokenized (*word\_tokenize*) and them stemmed (*PorterStemmer*) using functions from the *nltk* library. Before computing similarities, a sequence of rules is enforced to reduce the candidate sub-tabs in accordance with the rules described in the section (\*) below. The similarities between each of the (remaining) candidate sub-tab headings and the question asked are then computed using an ensemble model. This ensemble model using a character-based embedding method (*fuzz* from the *fuzzywuzzy* library) and a word-based embedding method (*TF-IDF*, using *TfidfVectorizer* from *sklearn*). The sub-tab heading with the greatest similarity is returned.

The two algorithms in the ensemble model are unsupervised, so the ensemble weights (0.35 and 0.65 for *fuzz* and *TF-IDF* respectively) were chosen by seeing which weighting combination (via a grid search) minimized error on the training set. This training set was manually created by selecting the correct sub-tab for ~180 of the DHS questions provided to us.

(\*) The rules are as follows:

* If the question contains the strings ‘ non’ or ‘ not ‘, then the sub-tab heading must contain at least one of those two strings.
* If the question contains the string ‘enquir’, then so must the sub-tab heading.
* If the question contains the string ‘exampl’, then so must the sub-tab heading.
* If the question contains the string ‘eligib’, then so must the sub-tab heading.

## Machine comprehension

The text belonging to the sub-tab heading from step (4) is extracted from ‘policies.json’. This text and the user’s question are then fed through a machine comprehension component that returns the span of text within the passage that best answers the question. For this step, a *BERT* model is used. *BERT* is a pre-trained language understanding model (from Google research), which can be fine-tuned for downstream tasks – the downstream task, in this case, is question answering. A *PyTorch* implementation of the *BERT* model used in this component is fine-tuned on the SQuAD (v1.1) dataset (this is a benchmarking NLP question answering dataset). This model predicts the start and end tokens from the passage that corresponds with the answer to the user’s question. From this output, the predicted response is returned.

The huggingface (Google research) repo containing the *PyTorch* implementation can be found here: <https://github.com/huggingface/pytorch-transformers>

The original BERT repo can be found here: <https://github.com/google-research/bert>

The SQuAD (v1.1) dataset can be downloaded here: <https://github.com/rajpurkar/SQuAD-explorer>

## Codes/Scripts explanation:

##### Front end:

* App.js – manage the data flow between each component and store some global variables
* components
  1. chatwindow – the chatbot UI for human to type in questions
  2. elasticserach – the UI for elasticsearch
  3. logo – brand logo and user login details
* constants.js – contains the backend entry point where the front end will call API from.

##### Back end:

* main.py – the server script for backend, where loads up all the model and handles most of the backend logic
* subtab.py – a script that defines the sub-tab layer logic
* new\_policies.json – it is used by subtab.py to find pre-defined sub-tabs
* bert.py – the wrapper script for BERT model
* chathistory – this folder stores all the user chat data. If you want to clear the chatbot history, simply delete everything inside this folder
* authentication – this folder holds all the user login details. You can add/delete users in User.json file
* elasticserach\_kpmg – this folder contains DSL query generator, which creates the query sent to elasticsearch
* sentence\_similarity – the core for comparing similarities between different sentences
* bert\_asset – contain all the property files, model, work tokenizer for Bert model
* output2 – the folder saves the fine-tuned Bert weights
* dist – the static HTML files that the server would serve in the production environment
* allennlp\_predict.py – it calls Allennlp APIs. It uses machine comprehension and NER models.