International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

AI-Driven Intelligent Business Opportunity Discovery and Tender Matching System with Semantic Analysis and Modeling

Ananya S G[1], Ms. Sujitha.S[2], C P Anushka Teju[3]

[1,3]dept of AIML, [2]Asst.Professor, Dept of AIML
[1,2,3]SJB Institute of technology
Bengaluru, karnataka

Abstract

A business may have the requirement to search for public procurement opportunities within inconsistently worded information. This project includes a Flask web application that can be used to search for tender related information, manage a company profile, save opportunities and view tender information. It also contains the code to implement semantic vectors, process text from PDFs and a generative AI question answering pipeline. The application uses an all-MiniLM-L6-v2 Sentence Transformer model to encode text from tenders and queries. The FAISS IndexFlatL2 was used to create a nearest neighbour search. Text was extracted from pages within the PDF using PyMuPDF. This text was then split into overlapping chunks to allow for character based chunking. In the document question section the chunks are scored based on cosine similarity. A selection of context, the user's company profile and the questioned are passed to the Gemini generative model. However, the source code does not fully link the different components together to create a full semantic tender matching application. The search route will either use a Selenium scrapping function or generative model/fallback function to find potential records. It will then do some literal querying. The match and eligible values displayed in the UI are calculated using a deterministic rule. This does not compare any attributes from the company profile to the tender. PDF files are saved and not processed using the PDF extraction and chunk indexing functions. There is no benchmark, labelled data or performance results. Overall this application can be seen as a working prototype with several AI focused components. This paper explains how it has been implemented and what would be required to bring the different components together to provide evidence based matching and document based answers.

Keywords--Tender discovery, Semantic Search, Sentence Transformer, FAISS, PDF Processing, Retrieval Augmented Generation, Flask, Procurement.

1. INTRODUCTION

It can be challenging to find the right tenders. The procurement notice may have technical, formal and administrational language, which may be different to how the company would describe their products/services. This is likely to cause issues as the supplier may search for tenders with a search term of 'network support' and not find any relevant tenders as it has been described as 'managed information technology services'. Multiple descriptions are given in the tenders and long documents, so it can be difficult to ascertain how big the opportunity is, who can apply, where, how much money and the deadline.

To overcome this issue we have created a web application that can store company information and tenders. This allows searching, storing and viewing of tender information. This also contains some semantic searching and document question answering. We have used Sentence Transformers to create dense vectors from text, this means words with similar meaning can be identified [1] [2]. We have also used FAISS to allow efficient searching of vectors [3]. The document question answering has a retrieval and generation pipeline. This ranks the text chunks in the database based on the question and passes them to the Gemini model.

When looking at the source code there are some issues with the application. We can create the FAISS index but the /search route isn't searching the index. We have selected tenders from either the scraper or fallback generator. We then filter the tenders based on if they contain the search term. The search is scored based on a deterministic function using the company name and tender title. There is no use of the output from the embedding model or any checks if the tender is eligible. We have functions to extract, chunk and index text from PDFs but they aren't being called in the upload route.

This paper discusses how the application has been implemented. It will discuss the architecture, data and limitations of the code. It will also outline how the prototype would need to be improved to ensure the searching and answers are verified.

The main benefit of this system is that tenders can be stored in an easy to review way. The company information can be kept up to date, tender information can be viewed and tenders can be saved for future reference. This is all done in one application meaning it would be an easy starting point for managing the tender discovery. The company information can also be used as a reference for future eligibility features.

As long as the components are working well together the system can be useful. The search results should be relevant, the extracted text from documents should make sense and the answers should be supported by the information retrieved. It is important to ensure that each component is tested individually and understand their behaviour.

This study will look at how the current prototype works. It will discuss how the application is structured, how the data is managed, how the search works and how the documents are processed. It will also highlight what changes need to be made to connect the components and allow them to be tested. This can then be improved further to create a more reliable tender discovery application to help businesses find procurement opportunities.

This system gives a good base for having all tender information and document analysis in one place. How useful the system is will depend on how accurate the information is and how well the components are connected. Improvements are needed to ensure the search is relevant, the eligibility results are verified and the answers are supported by reliable information from the documents. This can make the application more useful for businesses looking for the correct procurement opportunities.

2. Related Work

Contextual language models offer representations that capture the meaning of words in context. BERT was introduced to pretrain language models using bidirectional context for language understanding tasks [4]. Sentence-BERT is an adaptation of transformer models to provide sentence level embeddings that can

be efficiently compared using similarity [1]. The model used here is all-MiniLM-L6-v2 a Sentence Transformer model. This model is used to create embeddings of the tender title and description as well as the query text according to the search method defined. This is useful in overcoming vocabulary mismatch as similar descriptions will be close together in vector space. However, the source code doesn't show that the vectors are currently being used to influence the results returned by the /search route.

Dense retrieval methods have been developed to allow for passage retrieval based on learned representations of the query and passage. An example of this is Dense Passage Retrieval where two encoders are used to encode the question and passage, and the performance of the retrieval is tested on question answering benchmarks [5]. The chunk retrieval used in this project is a simpler approach whereby both the document chunks and query are encoded using the same Sentence Transformer model. The chunks are then ordered based on the cosine similarity. While this is a viable method for a prototype retrieval system, there is no benchmark or labelled question data.

FAISS is a library to perform similarity searches on dense vectors. It contains methods to perform similarity searches on large collections [3]. The index is created using the faiss.IndexFlatL2 method with 384 dimensions. The tender vectors are added to the index as float32 vectors. The index and mapping of IDs is saved to disk. The indexFlatL2 returns exact results for L2 distance between the vectors. The source code includes a method to load and search the index. However, this is never called in the route providing the search results to the user, so it shouldn't be assumed that the search page is influenced by vector retrieval.

Retrieval Augmented Generation combines external text with the generative model. The generative model can then use the retrieved information to create responses [6]. The chatbot implementation used here follows this general pattern. Up to four chunks are retrieved and sent to the prompt along with the company profile and question. This makes sense for high level document question answering. However, the quality and accuracy will depend on whether the correct document was used, if the retrieval was relevant and the generator doesn't go beyond the evidence provided.

There are extra challenges when searching for procurement information. A lot of the information within notices can include both structured and unstructured data. This means the system needs to keep track of where the information has come from to ensure the tender information is accurate. In this project, the source code contains a method to scrape data from the portal and a method to generate data. This means the tender data returned could be generated by the model or using local templates. In a real procurement search system, the URL where the information was sourced from, the time it was retrieved, the document ID and the source of each field would need to be recorded. The retrieval and eligibility decisions would also need to be assessed against verified records.

Problem Statement

A business looking to win procurement work needs to find notices that they are truly capable of providing. Using keyword search may not find relevant opportunities as the language used may be different. Once a notice has been found, details such as turnover requirements, experience, certifications, bid security, scope, location and closing date need to be checked.

The system has been created with user account and profile management, tender storage, search functionality, and tender detail viewing. The semantic indexing and retrieval code has also been created. However, there are multiple areas within the source code that haven't been implemented. The scores shown aren't based on the profile and tender requirements stored, and the tender PDF upload isn't linked to text extraction. Therefore, this should be seen as a prototype interface and component.

3. Proposed System

The project is a Flask web application that allows users to register, manage their company profiles, store tender information, search for tenders, save opportunities, upload documents, and view tender details. The code also includes semantic and RAG capabilities. This is separated into sections:

A. Business profiles and tenders

The business profile includes information about a company including the business name, type, turnover, years of experience, certifications, and location. Tenders include title, department, description, value, EMD, deadline, eligibility, location, source, and time created. Saved tenders link to tenders.

These are good inputs for future eligibility checks. However, in the current code, the calculate_advanced_matrices function does not compare the company's turnover, years of experience, certifications, location and type with the tender information. Instead, it calculates a deterministic seed based on the company and tender names. It then uses this seed to select an eligibility status and score range. This means that the results will be consistent every time, but do not indicate that the tender is eligible.

B. Tenders search

The search route will first check for any keywords in the cache. If it does not find any, it will use the selenium scrape function to scrape the portal. If it does not find any tenders, it will use the Gemini function which has a fallback to local templates. The function will try to scrape BidAssist and extract the structured state data, if this fails it will use BeautifulSoup to parse the cards. The tenders will be stored in SQLite.

It will then filter the tenders to only show those where the title or description contains the search phrase. It will not use the SemanticSearchEngine.search_similar_tenders function. There is functionality to encode the tender title and description, create a FAISS index, and find the nearest neighbours. However, the current search functionality does not use a semantic vector search.

C. Document processing and question answering

The PDF text is extracted using PyMuPDF. It will then split the text into chunks of 1200 characters with an overlap of 250 using a character based chunker. The index_tender_document_chunks function can be used to store the chunks in the tender_chunks table. However, the upload route will only store the PDF and insert the file name into the pdf_documents table. The tender details route will create a short fake text based on the database fields if there are no chunks. Therefore, the chatbot will have chunks of metadata created as context.

The chunks are encoded using the Sentence Transformer along with the user question. They are then ranked based on cosine similarity and up to 4 are selected. These are then passed to Gemini along with the tender and profile information. This is similar to RAG, but the documents depend on the source and content of the chunks.

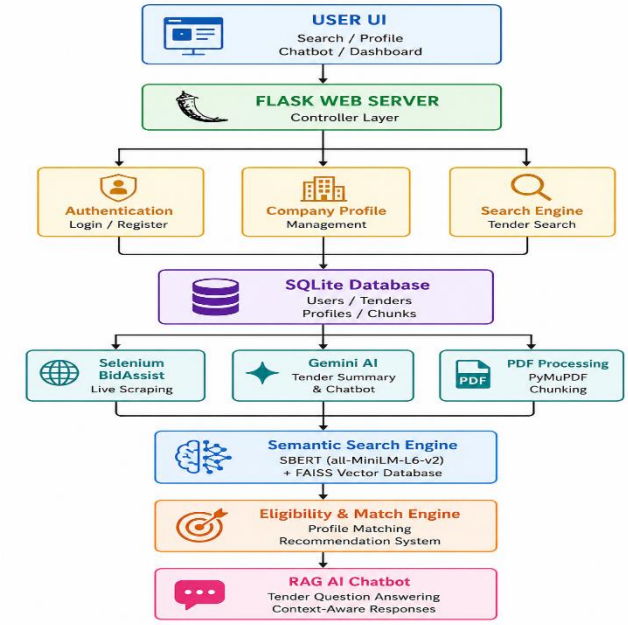
D. Summarisation

The tender details route will pass a prompt to Gemini to summarise the tender based on scope, value/EMD, certifications and turnover, deadlines and risks. The text passed to the function is built from the description and eligibility fields of the tender. It does not use the entire PDF that was uploaded. This means it is a generated summary of the tender fields.

The aim of this project was to create a web application that allows users to register, log in, manage company profiles, and store, search and save tenders. The application uses Sentence Transformers and FAISS to encode text semantically and perform similarity searches. It can extract text from a PDF, chunk the text with overlaps, and use RAG to generate answers using a generative model. Tender details and summaries are displayed on the web. More integration is needed to ensure that the scores and responses generated are reliable.

4. System Architecture

The source code follows this architecture. The diagram shows which functions of the vector search have been implemented.



This diagram shows the control flow from app.py. It should be noted that just because the functions have been defined doesn't mean they are implemented together.

5. Methodology

A. Business Profile

A business profile is created with the following fields: business name, type of business, turnover, number of years in business, certifications and location. This will create or update a row in the company_profiles table with the information provided. These fields can be used to determine if the business is eligible or relevant. However, in the current implementation only the business name will affect the deterministic seed for the matching function. The other fields will not impact the score or status.

B. Tenders

Tenders will be stored into the tenders table. When a search request is made, the cache will be checked using the lower case query as the key. If there is no match in the cache, the browser automation will be attempted first. The scraper will search for the input field, submit the query, wait for the page to load and attempt to extract the window.__INITIAL_STATE__ object. If this fails, it will search the page's HTML for cards.

If there are no tenders returned from the scraper, the Gemini function will be called to retrieve 10 tenders-like JSON objects. If this fails, the tenders will be checked locally first. If there are no local tenders, tenders will be created from the template. Some values, dates and locations will be generated or set by the code. These should not be shown as real tenders as they have not been validated.

The tenders returned will be stored in SQLite. If they were created via the cache miss route, the FAISS index will be created or updated. The tenders will be filtered using a literal match on the title and description. If the request is a cache hit, the FAISS index is not re-created.

C. Semantic Search

A model is loaded using all-MiniLM-L6-v2. The vector size is set to 384. When creating the index, each tender will be converted into a string containing the title and description. The model will encode the string and the vectors will be converted into float32. These will then be added into the FAISS index. When searching, the query will be encoded and the top k vectors will be retrieved from the FAISS index. This defaults to 30.

This is how the semantic search works. This is not how the results are filtered in the current implementation of /search.

D. FAISS Index

An index is created using IndexFlatL2. The tender embeddings are added and saved into tenders_search.index. A map is created to show the index position to the tender ID in the database. This is saved into map_store.pkl. When performing a vector search, both files will be loaded. The index will then be used to get the tender ID based on the vector position.

This is simple to implement for a prototype. It will search for exact matches based on L2 distance. As the vectors are not normalised in the source code, the FAISS index will not always return the same results as cosine similarity. The function to get the document chunks will calculate the cosine similarity explicitly.

E. PDF Extract and Chunk

The PDF is opened using PyMuPDF. The text is extracted from each page. Any pages with no text will be ignored. The remaining text will be combined. The text will then be split into chunks of 1200 characters with an offset of 950 characters. This means there will be an overlap of 250 characters. This is a character based chunk and will not maintain page or sentence boundaries.

The function index_tender_document_chunks will remove any existing chunks for the tender. The text will be extracted from the PDF. The text will then be chunked. Any non empty chunks will be added to the tender_chunks table. This function is present in the source code but is not called in the /upload route.

The file will be stored in the uploads folder and the filename will be recorded. The tender detail route will create the chunks from a string made up from the database fields.

F. RAG Chatbot

The /api/chatbot/ask endpoint will take a tender ID and message. The tender title and current user profile will be retrieved. The query_relevant_chunks function will be called. All chunks for the tender and question will be encoded using the Sentence Transformer. The cosine similarity will be calculated. The top 4 chunks will be combined.

The context, question, tender title and user profile fields will be added into the Gemini prompt. The prompt will ask for 5 bullet points. The prompt will also ask the model to mention the company name. The response will be returned along with the grounding_used boolean. This will confirm that there was some non empty context provided. It will not confirm that the answer is correct or from the context.

G. Summary

The generate_tender_summary function is called from the tender detail route. The prompt asks for the response to be summarised into 5 sections: scope, value/EMD, certifications/turnover restrictions, deadlines and risks. The input is a short string made up from the tender description and eligibility. This means it can summarise the fields stored but the full tender PDF is not passed into this function.

6. Experimental Setup and Evaluation

No formal experimental dataset, labelled relevance judgments, benchmark, test results, or user-study data were provided with the source code. It is therefore not possible to report precision, recall, F1-score, ranking quality, latency, or other performance measurements. The code alone also does not establish that the Selenium scraper successfully retrieves authoritative records in a repeatable environment, or that a configured Gemini API call succeeds.

A code-level functional evaluation can still identify the intended operations and their integration status:

TABLE I. **CODE-LEVEL FUNCTIONAL EVALUATION**

Workflow operation	Evidence in the source	Evaluation status
Register, log in, and manage a business profile	Flask routes and SQLite operations are defined	Implemented in application code
Store tender records and saved opportunities	Tables and routes are defined	Implemented in application code
Generate tender embeddings	Sentence Transformer encoding is used in the index builder	Implemented as a function
Create and query a FAISS index	Index creation and search methods are defined	Implemented as methods; search method is not called by /search
Retrieve search results	/search obtains records and applies title/description substring checks	Implemented, but not through the FAISS search method

Extract uploaded PDF text	PyMuPDF extraction function is defined	Not connected to the upload route
Create and store PDF chunks	Chunking and indexing functions are defined	Not invoked by the upload route
Retrieve chunks for a question	Stored chunks are ranked by cosine similarity	Implemented when chunks exist
Generate a chatbot answer	Retrieved context and question are sent to Gemini	Implemented subject to configuration and API availability
Summarize a tender	Tender description and criteria are sent to Gemini	Implemented for stored fields, not the uploaded full PDF
Calculate evidence-based eligibility	Current score function uses a deterministic seed	Not implemented

This assessment describes code paths, not successful end-to-end execution. Confirming runtime behavior would require the templates, database state, API configuration, sample PDFs, portal access, and test logs.

7. Results and Discussion

This source provides an example of working application design. Users can create a profile, search for tenders, save opportunities and view tender details. There are also implemented functions for embedding tender text, building a FAISS index, extracting text from PDFs, chunking text, ranking chunks and prompting a generative model.

However, there are some differences between what is available and how things work in practice. The semantic index is created using one of the search methods, however the results returned by /search are using substring matching rather than vector similarity. Whilst the score function is deterministic meaning the results shown will be consistent, this does not check the company's turnover, experience or certifications against the tender's requirements. This should not be taken as an indicator of whether the company will be eligible or likely to win.

The document pipeline also has some differences. Whilst files are saved upon upload, the upload method does not run the extract_text_from_pdf or index_tender_document_chunks functions. The tender detail view can generate chunks from the metadata of the tender if there are none present. This means the chatbot response may come from fields in the database rather than the uploaded PDF. The summary method also creates a string of description and criteria rather than the full document. This means the system is currently not very useful for detailed compliance questions.

The scraper and fallback sources also need to be handled carefully. Whilst the code can generate records using a language model or local templates if the scraper fails, this may only be useful for testing the interface. This source does not confirm that these records will relate to real active tenders. If this were to be deployed for users, it would need to be clearly marked or turned off to ensure the data being used is verified.

8. FUTURE WORDS

Improvements to be made in the future:

Link semantic search to search by querying the FAISS index and mapping results back to tender documents. Think about how to convert the vector distances into something meaningful. Consider normalising the scores and creating a hybrid search using keywords and vectors. Include evidence based filtering. Extract specific requirements from tenders and match against the profile fields. Provide evidence at the level of requirements. Any unknown values should be marked as not determined. Allow users to upload PDFs. Check that the file is the correct type/size, extract text and save the text in chunks with knowledge of which page they came from.

Improvements to citations. Provide the page number and passage that the answer can be found from. Tell the generator to tell the user if the text does not answer their question. Ensure citations come from verified procurement sources. Connect to official portals or verified sources, save the link and time of retrieval. Ensure that any verified data is clear from any demo or generated data. Create evaluation data. Gather real tender queries, relevant documents, eligibility examples and question/answer examples. This will allow you to evaluate the search and grounding. Improvements to chunking and multilingual. Consider whether sentence or structure based chunking would improve the current character based chunking. Consider language specific models if needed. Security improvements. Consider how you will handle secrets, file names during upload, session setup, database access and debug settings in production. User notifications and proposal support. Once verified matching is working reliably you could consider adding saved searches, deadline reminders and help with preparing documents.

9. CONCLUSION

This project contains a prototype Flask application providing functionality to manage business profiles and tenders, using semantic indexing, PDF handling, chunking and Gemini generation. The code can be seen to have a solid basis for a decision support system, however the core functionality is not yet fully integrated. The search results shown are not provided by the FAISS search, the eligibility ratings are not based on evidence but deterministic. Additionally, the uploaded PDFs are not being processed by the document search pipeline. There are no formal results to show.

The prototype provides a single application which allows for storing company profiles, searching for tenders, semantic similarity and document based question answering. The system also provides a basis for assessing eligibility, however more improvements would need to be made to ensure accurate results.

Overall this is a prototype showing the relevant components of software in the system and the user facing functionality. The search functionality would need to be linked to the routes, answers provided would need to be based on extracted information and be traceable back to specific pages. The tenders would need to be verified to ensure they are from valid sources. This would then allow for claims to be made about the applicability of the tenders and the overall performance of the system.

REFERENCES

1. N. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” in Proc. 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China, 2019, pp. 3982–3992, doi: 10.18653/v1/D19-1410.

2. N. Reimers and I. Gurevych, “Making Monolingual Sentence Embeddings Multilingual using Knowledge Distillation,” in Proc. 2020 Conference on Empirical Methods in Natural Language Processing, 2020, pp. 4512–4525, doi: 10.18653/v1/2020.emnlp-main.365.
3. J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” IEEE Transactions on Big Data, vol. 7, no. 3, pp. 535–547, 2021, doi: 10.1109/TBDATA.2019.2921572.
4. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in Proc. 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, vol. 1, 2019, pp. 4171–4186, doi: 10.18653/v1/N19-1423.
5. V. Karpukhin et al., “Dense Passage Retrieval for Open-Domain Question Answering,” in Proc. 2020 Conference on Empirical Methods in Natural Language Processing, 2020, pp. 6769–6781, doi: 10.18653/v1/2020.emnlp-main.550.
6. P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 9459–9474.