**Semantic Search in articles using Use Case**

**Conversion to Classification Problem**: The task will be converted as a classification problem where each article is classified into categories based on its semantic meaning. The query will also be transformed into a vector, and articles will be ranked based on the closest semantic match.

Dataset : data collected from chatgpt more than 400 sample split into train and test , 20% for test and 80% for training two categories one for AI and other sport dataset features text and

Baseline experiments: the goal of the experiments compare between models that can capture semantic meaning of the text

**Overall conclusion**

sentence transformers capture semantic meaning best than word2vec but run time would be long compared with word2vec i used vector db to imporve run  time  , word2vec has limited vocabulary to depend on training data

**We used two techniques**

1. google word2vec model skip gram based on neural network

2. Sentence Transformers Model **Transformer Architecture**: Sentence Transformers use the Transformer architecture, which is based on **self-attention mechanisms**. Transformers are capable of capturing long-range dependencies between words in a sentence by weighing the importance of each word relative to the others in the context.

Every part will be implemented as class in the notebook

**-word2vec class**

The word2vec class is designed to handle text data and classify it using Word2Vec embeddings along with a K-Nearest Neighbors (KNN) model. This class facilitates the conversion of text into vector representations using a pre-trained Word2Vec model, followed by training and evaluating a KNN classifier on the resulting embeddings

### Key Methods:

1. **\_\_**init\_\_(self, word2vec\_model, df)
   * **Purpose**: Initializes the class with a pre-trained Word2Vec model and a DataFrame. It computes the average word vector for each text entry in the DataFrame and splits the data into training and testing sets.
   * **Parameters**:
     + word2vec\_model: A pre-trained Word2Vec model to generate word embeddings.
     + df: A DataFrame with the text data in a column named 'text' and the corresponding labels in 'class\_id'.
2. get\_average\_word\_vector(self, concept)
   * **Purpose**: Computes the average word vector for a given text (concept). The word vector for each word in the concept is fetched from the pre-trained Word2Vec model, and the final representation is an average of these vectors.
   * **Parameters**:
     + concept: The input text string for which the vector representation is calculated.
   * **Returns**: A numpy.ndarray representing the averaged word vector.
3. test\_word2vec\_model(self)
   * **Purpose**: Trains a K-Nearest Neighbors (KNN) classifier using the training data and evaluates its performance (accuracy and F1 score) on the test data.
   * **Returns**: A tuple containing the accuracy and F1 score of the KNN classifier on the test data.

**Tools Used**

* Python : The programming language used for implementing the Word2vec class and its methods.
* Gensim: A library used to load the pre-trained Word2Vec model and compute word embeddings.
* Pandas: A library used to handle the dataset as a DataFrame.
* NumPy: Used to store and manipulate the averaged word vectors and labels as arrays.
* Scikit-learn: A machine learning library used to implement the K-Nearest Neighbors (KNN) classifier and evaluate its performance.
* NLTK (Natural Language Toolkit): A library used for tokenizing the text data (word\_tokenize).

## External Resources Used

* **Gensim Word2Vec**: Pre-trained Word2Vec model used for word embedding calculations.
* **Scikit-learn Documentation**: Used for implementing KNN and evaluating classification metrics like accuracy and F1 score.

**Results**

Evaluation Results for K-Nearest Neighbors Classifier using word2vec model :

Accuracy: 82.76%

F1 Score: 0.86%

**-sentences transformers (Part two ) :**

#### **Key Methods in the Class**

* \_\_init\_\_(self, embeddings, df): Initializes the class with an embedding model and DataFrame, splitting it into training and test sets.
* vector\_db(self): Sets up a Chroma vector database client, adds documents with metadata for similarity search.
* follow\_steps(self): Retrieves the top-3 most similar documents for each test document, prints similarities, and identifies the most frequent class. And following the code process
* test(self): Evaluates the model on the test set by predicting the most frequent class among the top-3 similar documents and calculating accuracy and F1 score.
* pipline(self): Executes the full pipeline: sets up the vector database, performs similarity search, and evaluates accuracy and F1 score by using test data

**Tools Used**

Python programing language

Sentence-Transformers (for generating text embeddings)

Chroma (for storing and querying vector data)

Pandas (for data manipulation and handling)

Scikit-learn (for data splitting and evaluation metrics)

Counter (from collections module for majority voting)

* 1. What was the biggest challenge you faced when carrying out this project?

run time in transformers I used vector DB to enhance the response

* 1. What do you think you have learned from the project?

remember the differences between skip gram and CBOW