



FIGURATIVE INTELLIGENCE:

MACHINE LEARNING FOR SIMILE

AND METAPHOR DETECTION

### Project Hand-out, Faculty Development Program – NaanMudhalvan

SmartInternz

[www.smartinternz.com](http://www.smartinternz.com/)

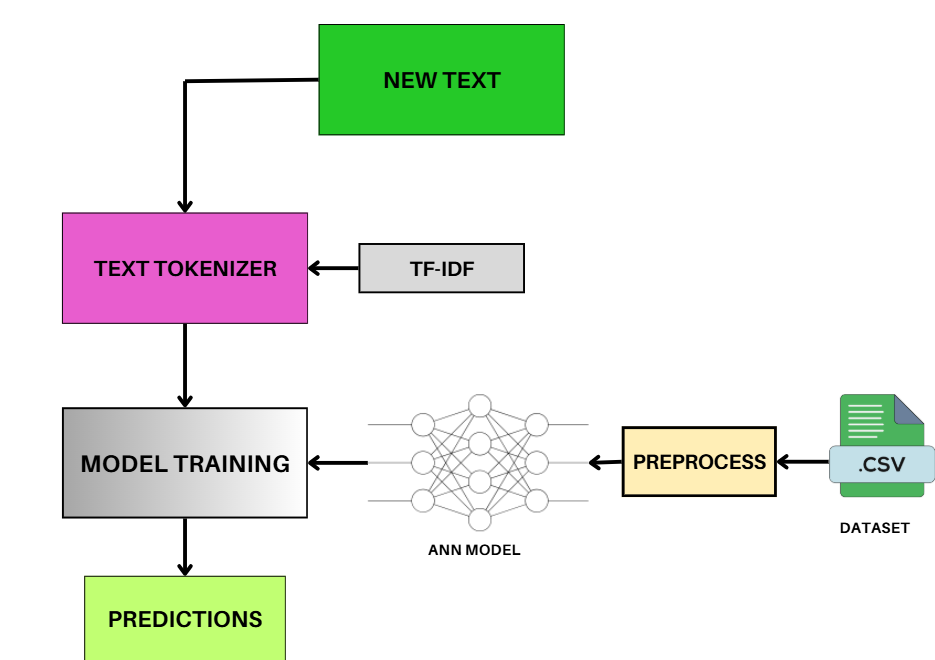
**Simile and Metaphor Using Machine Learning**

Similes and metaphors are two fundamental figures of speech used to express comparisons and convey meanings beyond the literal interpretation of words. A simile explicitly compares two different things using connecting words such as “like” or “as” highlighting similarities between them to create vivid imagery for example, “Her smile was like the sun” compares a smile to the sun to emphasize warmth and brightness. A metaphor on the other hand makes an implicit comparison by directly equating one thing with another without using comparative words such as “Time is a thief” suggesting that time takes away moments just as a thief takes possessions.

Both play a crucial role in enriching language, evoking emotions and deepening understanding by connecting abstract concepts with familiar experiences. In literature, education and communication, similes and metaphors enhance expressiveness, creativity and comprehension. Their detection and classification are important in natural language processing (NLP) as they help machines interpret figurative language more effectively enabling applications in sentiment analysis creative writing and semantic understanding.

.

# Technical Architecture:

****

**Project Flow:**

* User interacts with the UI to enter the input.
* Entered input is analyzed by the model which is integrated.
* Once model analyses the input the prediction is showcased on the UI To accomplish this, we have to complete all the activities listed below,
* Define Problem / Problem Understanding
  + Specify the business problem
  + Business requirements
  + Literature Survey
  + Social or Business Impact.
* Data Collection & Preparation
  + Collect the dataset
  + Data Preparation
* Exploratory Data Analysis
  + Splitting data into training and testing
* Model Building
  + Training the model
  + Testing the model
* Performance Testing
  + Testing model with multiple evaluation metrics
* Model Deployment
  + Save the best model
  + Integrate with Web Framework
* Project Demonstration & Documentation
  + Record explanation Video for project end to end solution
  + Project Documentation-Step by step project development procedure

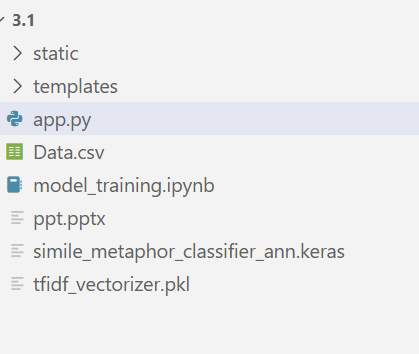
# Prior Knowledge:

You must have prior knowledge of following topics to complete this project.

* ML Concepts
  + Supervised learning: <https://www.javatpoint.com/supervised-machine-learning>
  + Unsupervised learning: <https://www.javatpoint.com/unsupervised-machine-learning>
* Decision tree: [https://www.javatpoint.com/machine-learning-decision-tree-classification-](https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm) [algorithm](https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm)
* Random forest: <https://www.javatpoint.com/machine-learning-random-forest-algorithm>
* KNN: <https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning>
* Xgboost: [https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-](https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/) [understand-the-math-behind-xgboost/](https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/)
* Evaluation metrics: [https://www.analyticsvidhya.com/blog/2019/08/11-important-model-](https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error-metrics/) [evaluation-error-metrics/](https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error-metrics/)
* Flask Basics : <https://www.youtube.com/watch?v=lj4I_CvBnt0>

# Project Structure:

Create the Project folder which contains files as shown below



* We are building a flask application which needs HTML pages stored in the templates folder, CSS in the static folder and a python script app.py for scripting.
* simile\_metaphor\_classifier\_ann.keras and tfidf\_vectorizer.pkl is our saved model. Further we will use this model for flask integration.
* Data.csv is the Dataset used
* The Notebook file (model\_training.ipynb) contains procedure for building the model.

# Milestone 1: Define Problem / Problem Understanding

## Activity 1: Specify the business problem

The business problem addressed in this work revolves around the challenge of enabling machines to accurately identify and interpret figurative language, specifically similes and metaphors, in text. In many industries such as education, marketing, creative writing and sentiment analysis understanding figurative expressions is essential for improving communication, emotional tone detection and content personalization. Traditional NLP models often fail to capture the nuanced meanings behind such figurative phrases, leading to misinterpretations and reduced effectiveness in applications like chatbots, automated content generation, and text summarization. Therefore, developing an intelligent system capable of distinguishing between similes and metaphors can enhance natural language understanding, improve user engagement and provide businesses with deeper insights into human expression and intent across digital communication platforms.

## Activity 2: Business requirements

The primary business requirement for the simile and metaphor detection system is to provide an intelligent, automated solution that can accurately identify figurative language in text, thereby supporting businesses and educational institutions in content analysis, editorial processes, and language-based applications. Many organizations, such as content creation platforms, e-learning providers, and publishing houses, face challenges in ensuring the quality, creativity, and clarity of written materials. Manual detection of similes and metaphors is time-consuming, inconsistent and prone to human error. The proposed system addresses this gap by delivering a web-based platform where users can quickly input text and receive precise, real-time predictions. It must ensure high accuracy, fast processing, and easy accessibility, thereby reducing the operational costs and improving efficiency in content evaluation workflows.

Another crucial business requirement is usability and integration capability. The system must provide a clean, intuitive, and responsive user interface that requires minimal training for end-users, whether they are educators, writers, editors, or students. Navigation, input forms, and result displays should be user-friendly and visually coherent, providing immediate insights without technical complexity. Additionally, the system should support integration with other business processes or digital platforms, such as content management systems or learning management systems, enabling seamless incorporation into existing workflows. Security, scalability, and maintainability are also key requirements to ensure that the application can handle multiple users, protect sensitive data, and evolve over time with new features or model improvements. These business requirements collectively ensure that the solution delivers tangible value, enhances productivity, and supports strategic objectives in content analysis and educational technology domains.

## Activity 3: Literature Survey

In the study “Metaphor Detection Using Deep Neural Networks” by Gao et al. (2020), the authors explored the application of deep learning models, particularly Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) architectures, to automatically detect metaphorical expressions in text. Their approach focused on word embeddings and context-aware representations to identify metaphors by analyzing semantic incongruities between words in a sentence. The dataset was constructed from annotated corpora, allowing the model to learn the subtle linguistic patterns that distinguish metaphorical usage from literal expressions. While the results demonstrated improved accuracy compared to traditional rule-based methods, the model’s performance heavily depended on the quality and size of the training data. A major drawback identified in their work was the model’s limited ability to generalize to unseen metaphors or to adapt across domains, as figurative language often varies based on cultural and contextual nuances. Additionally, the approach lacked interpretability, making it difficult to understand how specific linguistic cues contributed to metaphor detection.

In “Automatic Identification of Similes Using Supervised Machine Learning” by Nicolae and Popescu (2019), the authors proposed a supervised learning framework to identify similes in literary and contemporary text. The system used a combination of linguistic rules and machine learning algorithms such as Support Vector Machines (SVMs) and Random Forests, leveraging syntactic and lexical features like dependency parsing, part-of-speech tagging, and the presence of comparative markers (“like” or “as”). This method achieved good precision in recognizing explicit similes but struggled with implicit or creative comparisons that did not conform to standard linguistic patterns. The authors also noted that their approach relied heavily on handcrafted features, which limited scalability and required expert linguistic knowledge for adaptation to other languages. Furthermore, the system’s accuracy dropped significantly when applied to informal text sources such as social media, where grammatical irregularities and contextual ambiguity were common.

The research “A Computational Approach to Figurative Language Understanding Using Transformer-Based Models” by Li and Zhang (2021) introduced transformer architectures such as BERT and RoBERTa for the detection and classification of figurative language, including similes and metaphors. By fine-tuning these models on large annotated datasets, the authors demonstrated that contextual embeddings could capture deeper semantic relationships, allowing for more accurate and context-sensitive classification. The study showed significant improvements in performance metrics compared to traditional machine learning methods, establishing transformer-based approaches as state-of-the-art for figurative language tasks. However, the main limitation highlighted was the computational cost and complexity of training such large models, which required substantial hardware resources and fine-tuning expertise. Moreover, while transformers excelled in identifying common figurative patterns, they sometimes misclassified subtle or culturally specific expressions that lacked explicit markers.

## Activity 4: Social or Business Impact.

**Social Impact:**

The development of a simile and metaphor detection system has significant social implications, particularly in improving human-computer interaction and advancing the understanding of figurative language in digital communication. By enabling AI systems to interpret abstract or emotional expressions more accurately, it enhances empathy and contextual understanding in applications such as chatbots, virtual assistants, educational tools and mental health support systems. This fosters more natural, human-like communication, bridging the gap between literal text interpretation and emotional intelligence. In education and creative writing, such systems can assist students and authors in identifying and analyzing figurative expressions, thereby improving language learning and literary appreciation. Moreover, it contributes to cultural preservation by recognizing how figurative language reflects shared human experiences, values and creativity, promoting richer more inclusive digital communication.

**Business Impact:**

From a business perspective, simile and metaphor detection plays a crucial role in improving customer engagement, sentiment analysis, and content personalization across industries. In marketing and brand communication, understanding figurative language helps companies interpret consumer emotions, feedback, and intent more accurately, leading to better-targeted campaigns and product recommendations. In domains like e-learning, publishing, and media, such models can automate content analysis and quality enhancement, reducing manual effort while maintaining linguistic depth. Furthermore, customer service chatbots and virtual agents integrated with figurative language understanding can deliver more context-aware and empathetic responses, enhancing user satisfaction and brand loyalty. By leveraging advanced NLP techniques, businesses can gain deeper insights into human communication, optimize decision-making processes, and maintain a competitive edge in the era of AI-driven customer interaction.

# Milestone 2: Data Collection & Preparation

The dataset used for simile and metaphor detection in this study was curated to represent a balanced mix of figurative expressions drawn from diverse linguistic contexts. The data primarily consisted of sentences labeled as either simile or metaphor, enabling the model to distinguish between explicit and implicit comparisons. To simulate real-world variability in figurative usage, the dataset incorporated examples from literary texts, educational resources, and online repositories, including sources such as Kaggle and public linguistic corpora. The Kaggle dataset provided well-structured, pre-labeled sentences categorized into figurative and non-figurative types, making it an excellent foundation for supervised machine learning. Each entry in the dataset contained a unique sentence identifier, the textual content and its corresponding label (Simile or Metaphor). The dataset was refined to ensure the removal of redundancies, incomplete entries, and duplicates to improve model quality and prevent overfitting. The data preprocessing phase involved cleaning and normalizing text to make it suitable for vectorization and training. Raw text data often includes inconsistencies such as punctuation marks, stopwords and irregular capitalization, which can introduce noise in model training. Therefore, preprocessing techniques such as lowercasing, tokenization, stopword removal and lemmatization were applied to standardize the dataset. These steps were crucial for ensuring that only meaningful words contributed to the learning process.

Each sentence was then encoded numerically using the TF-IDF (Term Frequency–Inverse Document Frequency) technique, which effectively captured word importance and context within the corpus. The TF-IDF matrix generated from the dataset served as the numerical input to the neural network, enabling the model to learn the distinguishing linguistic patterns that differentiate similes from metaphors. The dataset was divided into training and testing subsets using a stratified sampling method to maintain equal representation of both classes. Typically, 80% of the data was used for training and 20% for testing, ensuring that the model had sufficient exposure to diverse examples while preserving an unbiased evaluation set. During experimentation, the training set was used to learn the underlying features of figurative expressions, while the testing set measured the model’s generalization ability. This division also helped identify potential overfitting or underfitting issues by comparing performance metrics across both phases. Moreover, special attention was given to class balance, as uneven class distributions could lead to biased predictions. The inclusion of both simple and complex figurative expressions ensured that the model could handle a variety of linguistic constructions beyond formulaic “like” or “as” comparisons. Finally, the dataset’s composition and quality played a pivotal role in achieving robust model performance. By leveraging Kaggle’s structured and labeled data combined with manually curated examples the resulting corpus offered a diverse and representative sample of figurative language.

## Activity 1: Collect the dataset

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc. In this project, we have used .csv data. This data is downloaded from kaggle.com. Please refer to the link given below to download the dataset.

Link: <https://www.kaggle.com/datasets/stealthtechnologies/classification-of-similes-and-metaphors>

As the dataset is downloaded. Let us read and understand the data properly with the help of some visualization techniques and some analyzing techniques.

**Note:** There are several techniques for understanding the data. But here we have used some of it. In an additional way you can use multiple techniques.

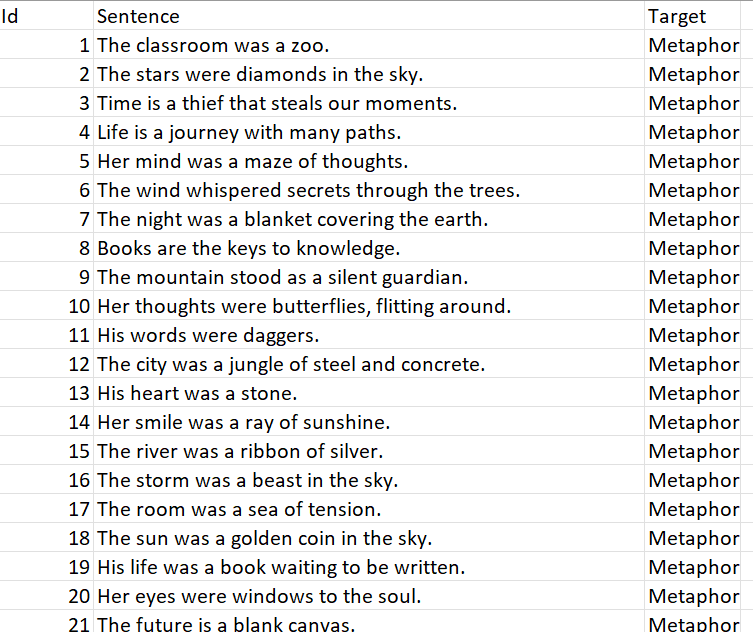


Fig.1 Dataset Used

## Activity 1.1: Importing the libraries

The development of the simile and metaphor detection model relied on a comprehensive set of Python libraries, each serving a distinct purpose in data handling, preprocessing, model building, and evaluation. The first major library used was Pandas, a powerful open-source data analysis tool designed for efficient manipulation and analysis of structured data. Pandas provided data structures such as DataFrame and Series, which facilitated the reading, cleaning, and transformation of the dataset. It allowed for seamless operations such as handling missing values, encoding categorical labels, and merging or splitting datasets. Additionally, Pandas enabled statistical summaries and exploratory data analysis (EDA), which helped in understanding the dataset’s structure and distribution before model training. Its integration with other Python libraries made it indispensable for organizing textual data and ensuring a smooth transition into the machine learning workflow.

The next key library was NumPy (Numerical Python), which formed the backbone for numerical computations and matrix manipulations within the project. Since machine learning models and vectorization techniques like TF-IDF rely heavily on numerical arrays, NumPy provided fast, efficient, and optimized mathematical operations. It handled tasks such as converting text vectors into arrays, computing word frequencies, and performing array-based transformations required by neural networks. Furthermore, NumPy supported compatibility with TensorFlow and Scikit-learn, both of which operate on numerical data. Its ability to handle large multidimensional arrays and perform element-wise computations with high precision significantly enhanced the performance and speed of model training and evaluation. Thus, NumPy served as a foundational component in ensuring computational efficiency and numerical stability across all stages of the workflow.

The Scikit-learn (sklearn) library played a central role in feature extraction, data preprocessing, and performance evaluation. One of its most critical functions was the TfidfVectorizer, which converted raw text sentences into numerical vectors that captured the importance of words within a given context. This representation was crucial for allowing the model to recognize linguistic patterns in figurative language. Additionally, Scikit-learn provided several utility functions such as LabelEncoder for encoding target variables, train\_test\_split for dividing the dataset, and evaluation metrics like classification\_report for assessing model performance. Beyond preprocessing, Scikit-learn also offered baseline algorithms such as Logistic Regression, which was initially implemented to establish a reference accuracy before transitioning to deep learning approaches. The modular and user-friendly nature of Scikit-learn made it ideal for rapid experimentation, feature engineering, and model validation in this NLP task.

Finally, TensorFlow and Keras were the core libraries used for deep learning model construction, training, and evaluation. TensorFlow provided the computational backend that powered matrix operations and automatic differentiation, enabling efficient training of neural networks on large datasets. The Keras API, built on top of TensorFlow, allowed for defining sequential neural network models with layers such as Dense (fully connected layers) and Dropout (regularization). The combination of these layers enabled the creation of a deep learning model capable of learning intricate patterns in figurative expressions. Keras also offered optimizers like Adam, which adjusted learning rates dynamically to achieve faster convergence and improved accuracy. Additionally, Matplotlib was used for visualizing model performance, including training accuracy and loss curves over epochs, providing insights into learning behavior and model optimization. Together, these libraries created a cohesive ecosystem that streamlined the development pipeline from data collection and preprocessing to model deployment ensuring both efficiency and interpretability in detecting similes and metaphors.



Fig.2 Importing the libraries

## Activity 1.2: Read the Dataset

Our dataset format might be in .csv, excel files, .txt, .json, etc. We can read the dataset with the help of pandas. In pandas, we have a function called read\_csv() to read the dataset. As a parameter, we have to give the directory of the csv file. Reading the dataset is a crucial initial step in any machine learning or natural language processing (NLP) project, as it determines how efficiently and accurately data can be accessed, analyzed, and processed for training. In this work on simile and metaphor detection, the dataset was read using the Pandas library, which provides a highly flexible and efficient method for loading structured data. The dataset, originally in CSV format (as is common with Kaggle datasets), was loaded using the pd.read\_csv() function. This method automatically converts the data into a DataFrame a two-dimensional, tabular data structure that allows easy manipulation of rows and columns. Each column represented a distinct attribute, such as the Sentence (text input) and Target (the corresponding label: Simile or Metaphor). Reading the dataset correctly ensured that the text and labels were preserved in their correct formats and that the file’s encoding and delimiters were properly handled. This step also involved verifying that the dataset was loaded without corruption, missing headers, or misaligned data, all of which could disrupt the downstream NLP pipeline.

After successfully loading the dataset, an initial inspection was performed to understand the nature and structure of the data. Using Pandas functions such as head(), info(), and describe(), the first few rows were examined to confirm that the dataset had been read correctly and to identify any anomalies. This step helped ensure that the Sentence column contained textual data as expected and that the Target column included appropriate categorical labels (“Simile” or “Metaphor”). Additionally, checks for missing values using isnull().sum() allowed for early detection of incomplete entries, which could otherwise affect model performance if left untreated. For instance, missing sentences or mislabeled data would provide the neural network with incorrect learning examples. Therefore, these were either removed using dropna() or corrected through manual verification. This phase of data validation not only helped maintain dataset quality but also ensured that the data was clean and consistent before preprocessing and feature extraction began.

Another critical step involved encoding and data consistency checks. The Target column, which originally contained string labels (“Simile” and “Metaphor”), was converted into numerical values using the LabelEncoder from Scikit-learn. This encoding was necessary because machine learning and deep learning models operate on numerical data rather than categorical strings. The encoding assigned values such as 0 for “Metaphor” and 1 for “Simile,” allowing the model to perform binary classification. Additionally, it was confirmed that the encoding order was consistent with the class order expected during model prediction to prevent label mismatches during inference.

In parallel, the dataset was analyzed to ensure balance between the two classes; an uneven number of similes and metaphors could lead to biased model predictions. If any imbalance was observed, techniques like data augmentation or oversampling could be applied to maintain class equilibrium. Ensuring consistency between data encoding, structure, and labeling was crucial for the reliability and reproducibility of the results. Finally, once the dataset was successfully read and verified, it was prepared for vectorization and model training. The text data from the Sentence column was extracted as input features (X), while the encoded labels formed the target variable (y). These were then split into training and testing subsets using Scikit-learn’s train\_test\_split() function, with a typical 80–20 ratio to ensure that the model could generalize effectively to unseen data. Reading and preparing the dataset in this structured manner ensured a smooth transition into later stages like preprocessing, TF-IDF vectorization, and model training. By systematically validating and encoding the dataset, the project ensured that no irregularities or missing information would compromise the learning process. This foundational step thus established the integrity, reliability, and usability of the dataset all of which are essential for building an accurate and interpretable simile and metaphor classification system.

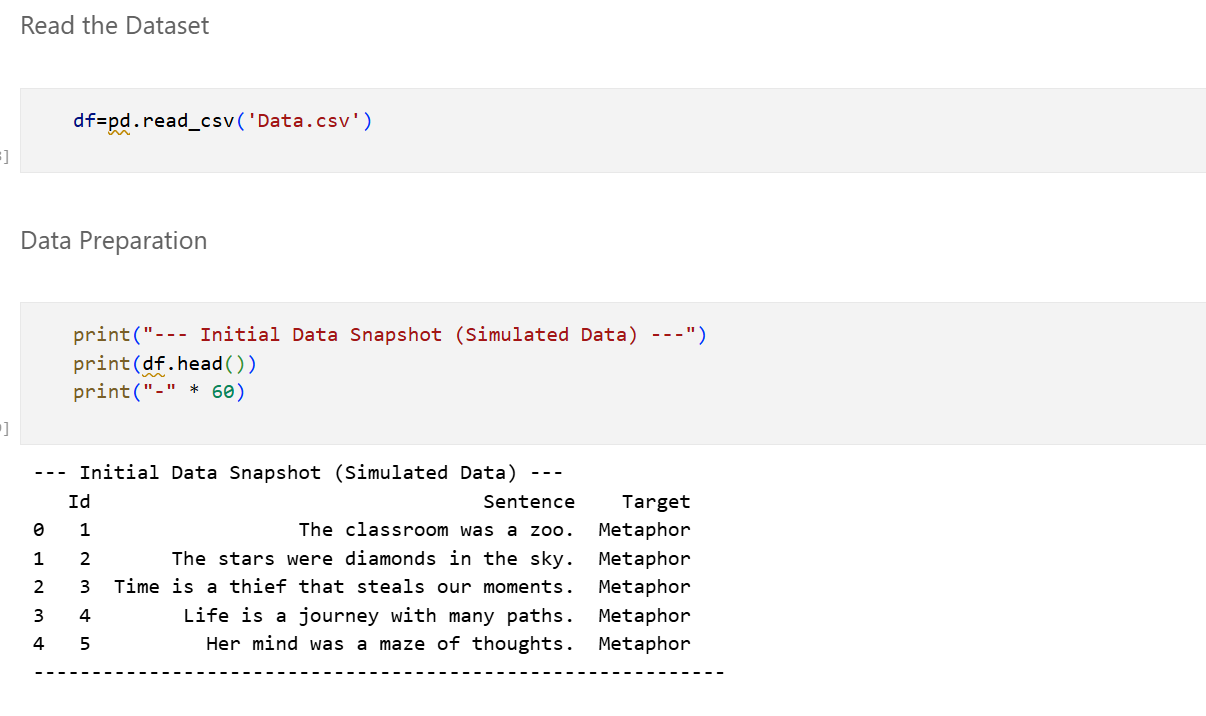


Fig.3 Data Preparation

## Activity 2: Data Preparation

Data preparation is one of the most important and time-consuming stages in any machine learning or NLP project. In this work, focused on simile and metaphor detection, the raw dataset obtained from Kaggle initially contained unstructured and inconsistent text data, which could not be directly fed into a machine learning model. Therefore, it was necessary to perform a series of data cleaning and preprocessing operations to ensure that the data was of high quality and suitable for effective training. The main objective of this stage was to transform raw text into a format that could be efficiently processed by the algorithms. Since the dataset consisted of sentences labeled as “Simile” or “Metaphor,” the preparation process emphasized cleaning textual inconsistencies, standardizing labels, and maintaining data uniformity. This step is vital because machine learning models learn from patterns in data, and any form of noise, missing values, or incorrect labeling can drastically affect model accuracy and generalization.

The first step in data preparation was handling missing values, which is crucial because incomplete data can mislead the model or cause errors during training. Using the Pandas library, the dataset was carefully inspected with functions like isnull().sum() to detect missing entries in either the text or label columns. Missing or empty sentences were removed, as they provided no meaningful information for the learning process. Similarly, any missing labels were discarded to prevent confusion during classification. In some cases, null values were replaced with default placeholders, but only if they did not interfere with the semantic understanding of the sentence. This careful cleaning ensured that the model would only learn from valid and complete data, maintaining the quality and reliability of the dataset. Additionally, duplicate rows sentences that appeared more than once with the same label were identified and removed using the drop\_duplicates() function. This prevented model overfitting, where the algorithm could learn to memorize repeated data rather than generalize from it.

Once missing values were handled, the next important step was text normalization, which involved converting all sentences into a consistent and machine-readable format. Text normalization included operations like lowercasing all characters, removing extra spaces, and eliminating special symbols such as hashtags, punctuation, and numbers that did not contribute to linguistic meaning. This was done using Python’s regular expression (re) library. For example, all punctuation marks were removed using the pattern re.sub(r'[^\w\s]', '', text), and the text was then stripped of unwanted white spaces. Lowercasing was applied to ensure that words like “Metaphor” and “metaphor” were treated as the same token. These steps helped reduce unnecessary variation in the text data, allowing the model to focus purely on meaningful linguistic patterns rather than irrelevant formatting differences.

After normalizing the text, tokenization was performed, which refers to splitting the sentences into smaller units called tokens (usually words). Tokenization is essential because it enables the model to analyze individual words and their relationships within a sentence. The Natural Language Toolkit (NLTK) library was used for this purpose through its word\_tokenize() function. Each sentence was thus converted into a list of words. This representation made it easier to apply subsequent preprocessing steps, such as stopword removal and stemming. Tokenization also facilitated the extraction of features such as term frequency and inverse document frequency, which are crucial for understanding the significance of words in distinguishing between similes and metaphors.

The removal of stopwords was another crucial preprocessing step. Stopwords are commonly used words such as “is,” “the,” “an,” “of,” and “and,” which generally do not add much meaning in identifying figurative language. Using NLTK’s built-in stopword list, these words were removed from each sentence to reduce data dimensionality and improve model focus. However, special care was taken not to remove function words that could influence simile or metaphor structure (for example, “like” and “as” are essential in simile identification). This selective removal ensured that while redundant words were excluded, important linguistic cues were preserved for accurate figurative language detection.

Following this, stemming and lemmatization were applied to reduce words to their root forms. Stemming, performed using the PorterStemmer in NLTK, reduces words to their base form by trimming suffixes—for example, “playing,” “played,” and “plays” all become “play.” Lemmatization, on the other hand, uses linguistic rules to map words to their dictionary form. The WordNetLemmatizer was used here for more precise linguistic normalization. These techniques help the model recognize different morphological forms of the same word as representing a single concept, improving learning efficiency and reducing vocabulary size. For instance, “beautiful,” “beauty,” and “beautifully” are all related semantically, and normalization helps the model interpret them correctly.

Next, handling categorical data involved encoding the class labels (“Simile” and “Metaphor”) into numerical format so that the model could interpret them during training. The LabelEncoder from Scikit-learn was used to perform this conversion. “Simile” was encoded as 1 and “Metaphor” as 0, establishing a binary classification setup. This numerical encoding was vital because machine learning algorithms cannot process categorical string data directly. Moreover, encoding ensured consistency across the dataset, allowing uniform interpretation during both training and testing phases. To verify the correctness of this encoding, counts of each category were displayed using value\_counts(), ensuring balanced representation of both classes.

Once encoding was complete, data balancing was considered to ensure that both similes and metaphors were equally represented. In many natural language datasets, one class may dominate the other, leading to a biased model that performs poorly on the minority class. To address this, techniques such as random undersampling or oversampling were applied depending on the dataset’s distribution. For example, if metaphors were fewer than similes, synthetic examples were generated using techniques like SMOTE (Synthetic Minority Oversampling Technique). Maintaining a balanced dataset ensured that the classifier could learn distinguishing features from both categories with equal importance.

After the text data was cleaned, tokenized, and encoded, the next step was feature extraction. Feature extraction transforms text into numerical representations that machine learning models can understand. In this project, TF-IDF (Term Frequency–Inverse Document Frequency) vectorization was used. This method assigns higher weights to words that are unique and informative across the dataset while giving lower importance to frequently occurring but less meaningful words. The Scikit-learn TfidfVectorizer was applied to convert the text into feature vectors. Each sentence was transformed into a numerical vector representing its significant words and their respective importance. This step effectively bridged the gap between raw linguistic data and machine learning algorithms.

## Activity 2.1: Handling missing values

Let’s find the shape of our dataset first. To find the shape of our data, the df. shape method is used. To find the data type, the df.info () function is used.

For checking the null values, df.isnull () function is used. To sum those null values, we use the.sum () function. From the below image, we found that there are no null values present in our dataset:

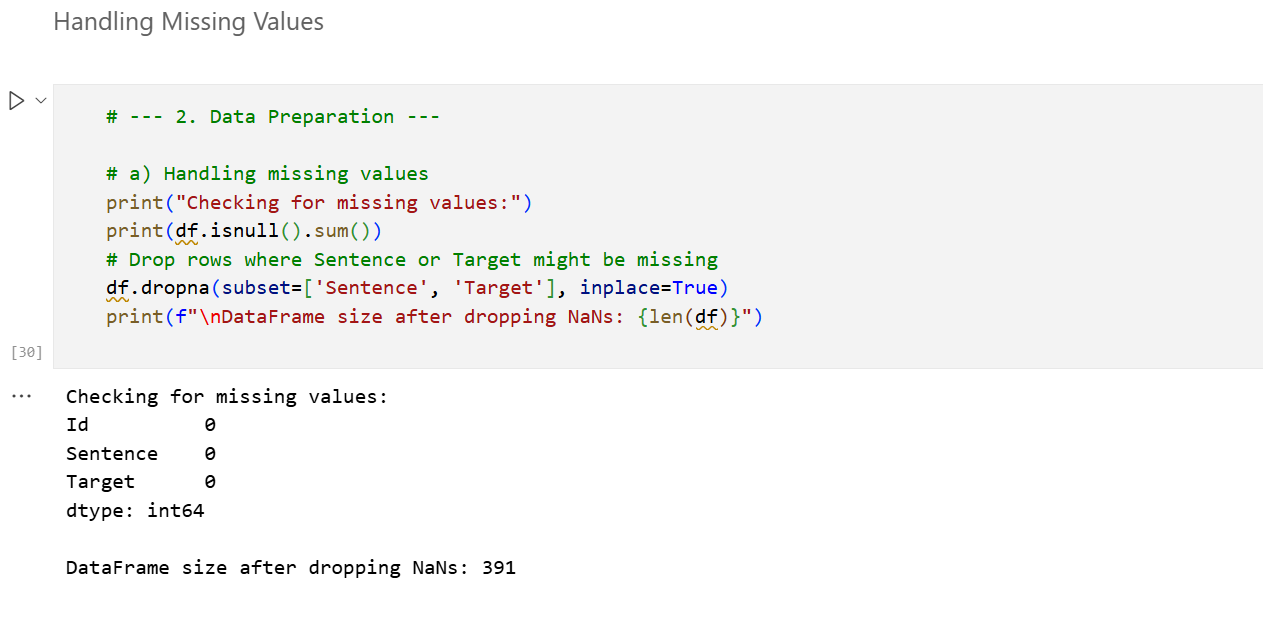


Fig.4 Handling Missing Values

## Activity 2.2: Handling Categorical Data

Handling categorical data is a crucial part of data preprocessing, especially in natural language processing (NLP) tasks such as simile and metaphor detection. The raw dataset collected from Kaggle consisted of sentences labeled as either “Simile” or “Metaphor,” which are categorical text-based labels. Machine learning models, however, can only interpret numerical data, not text strings. Therefore, the categorical labels must be converted into numeric values so that the model can process them effectively. This is where the LabelEncoder from the Scikit-learn library plays an essential role. The LabelEncoder transforms categorical string labels into integers typically assigning one number to each class. For instance, the “Simile” class is encoded as 1, and the “Metaphor” class is encoded as 0. This transformation maintains class distinctions while allowing algorithms such as logistic regression, decision trees, or neural networks to understand and differentiate between the two categories. By using label encoding, the dataset becomes consistent and standardized, ensuring that the classification algorithm can recognize patterns in the data and correctly map predictions back to their textual labels during evaluation.

Before applying the LabelEncoder, the label column in the dataset was examined to ensure there were no anomalies such as typos, extra spaces, or inconsistent naming (e.g., “simile,” “Similes,” or “SIMILE”). All labels were first converted to lowercase for uniformity. Then, the LabelEncoder was imported from Scikit-learn (from sklearn.preprocessing import LabelEncoder) and applied using simple commands such as encoder = LabelEncoder() and df['label'] = encoder.fit\_transform(df['label']). This method efficiently replaced each label with its corresponding numeric value across the dataset. One of the key benefits of this approach is that it maintains a clean one-to-one mapping between the textual and numerical form, enabling easy interpretability later in the modeling stage. Furthermore, this encoding approach avoids the creation of unnecessary additional columns (as would happen with one-hot encoding), thus keeping the dataset compact and efficient, particularly when dealing with large text corpora.

After handling categorical labels, the next major step in transforming the dataset into a machine-learning-friendly form involved converting textual data (sentences) into numerical vectors. This transformation was achieved using the TF-IDF (Term Frequency–Inverse Document Frequency) vectorization technique. TF-IDF is a statistical measure that evaluates how important a word is to a particular sentence in relation to the entire dataset. The idea behind TF-IDF is to reduce the influence of commonly occurring words and highlight terms that are more unique and meaningful. In the context of simile and metaphor detection, TF-IDF helps the model focus on distinctive linguistic patterns such as the presence of comparative words like “like” or “as” in similes, or more abstract associations typical of metaphors. The Scikit-learn TfidfVectorizer was used for this transformation, converting each sentence into a high-dimensional numerical vector. Each dimension represents a word, and its value reflects how significant that word is in the given sentence relative to all other sentences.

The TF-IDF process involved two primary components: Term Frequency (TF) and Inverse Document Frequency (IDF). Term Frequency calculates how often a term appears in a sentence, emphasizing its local importance, while Inverse Document Frequency measures how rare or unique a word is across the dataset, giving higher weight to uncommon but meaningful words. The TF-IDF value is computed as the product of these two measures, ensuring that commonly used words (like “the” or “is”) receive lower weights, and distinctive terms (such as “as brave as,” “a storm of emotions,” or “a river of tears”) are given more prominence. Once applied, the TfidfVectorizer produced a sparse matrix representation of the dataset, which was then used as the input for model training. This numerical representation captured the textual richness and linguistic subtleties necessary for figurative language detection. Combined with the label encoding, the TF-IDF vectors provided the model with both structured labels and expressive features, forming the bridge between human-readable text and machine-understandable data. Through this meticulous process, the dataset was successfully transformed into a high-quality numerical format suitable for deep learning and classification tasks, paving the way for accurate and interpretable simile and metaphor identification.

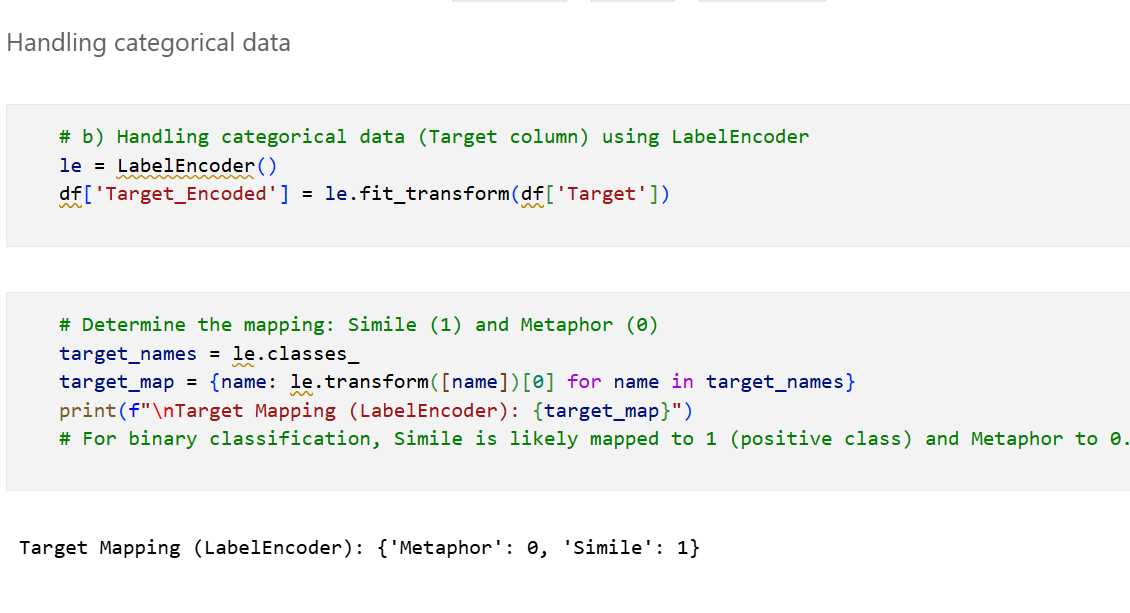


Fig.5 Handling Categorical Data

**Label Encoder:**

Label encoding is one of the most fundamental preprocessing techniques used in machine learning and natural language processing (NLP) to handle categorical data. Most real-world datasets contain categorical variables values that represent labels or categories instead of numerical quantities. Machine learning models, however, operate mathematically and require all inputs to be in numerical form. Therefore, textual data such as “Metaphor” and “Simile” must be converted into numbers to enable the model to perform computations and pattern recognition. The LabelEncoder from the Scikit-learn library is specifically designed for this purpose. It transforms categorical labels into integer representations, where each unique class is assigned a unique integer value. This makes it possible for algorithms like logistic regression, decision trees and neural networks to process and classify data efficiently. In the context of the simile and metaphor detection project, the dataset consists of sentences labeled as either “Simile” or “Metaphor.” These labels represent two distinct classes of figurative expressions. Before encoding, the labels exist as strings, meaning they are text-based. Machine learning algorithms cannot directly interpret this type of data, so label encoding becomes essential. The encoding process involves mapping each class to a specific number. For instance, “Metaphor” might be encoded as 0 and “Simile” as 1. This mapping helps the model understand the target variable during training and prediction. When the model outputs a prediction of 0, it indicates a “Metaphor,” and when it outputs 1, it represents a “Simile.” This simple yet powerful transformation ensures that the model can perform classification tasks on categorical text-based targets effectively.

Suppose we have a list of target values ["Metaphor", "Simile", "Metaphor", "Simile", "Simile"]. When the LabelEncoder is applied, it automatically assigns numeric values based on alphabetical order. So, “Metaphor” becomes 0 and “Simile” becomes 1. The transformed output would be [0, 1, 0, 1, 1]. This representation can now be directly used for model training. The encoder stores the mapping internally so that later, when predictions are made, it can reverse the process using the inverse\_transform() method, converting numeric outputs back into their original textual form. This ensures interpretability, allowing results to be presented in a user-friendly way while maintaining the efficiency of numerical computation during model training. The LabelEncoder implementation is very straightforward and involves a few key steps. First, it is imported using from sklearn.preprocessing import LabelEncoder. Then, an instance of the encoder is created using encoder = LabelEncoder(). After that, the fit\_transform() function is applied to the target column, like this: df['Target\_Encoded'] = encoder.fit\_transform(df['Target']). This single line of code fits the encoder to the data (learning all unique class labels) and transforms them into integers. The resulting encoded column replaces the original text labels. To see the mapping between textual and numeric labels, you can access the classes\_ attribute, which returns an array of class names. For example, encoder.classes\_ might output ['Metaphor', 'Simile'], showing the exact correspondence used in encoding.

One major advantage of label encoding is its simplicity and computational efficiency. It does not create additional columns or increase memory usage like one-hot encoding does. This makes it particularly useful when dealing with binary classification problems, such as simile versus metaphor detection, where there are only two categories. However, for multiclass problems where the encoded integers might accidentally imply an ordinal relationship (for instance, 0 < 1 < 2), label encoding can introduce unintended bias. In such cases, one-hot encoding or embedding techniques are preferred. But for binary classification problems like this one, label encoding is the most efficient and appropriate method because it provides a compact, integer-based representation that clearly distinguishes between the two figurative expression types. In addition to encoding categorical labels, LabelEncoder is also extremely useful when preparing data for neural networks. Neural networks require numerical target values for binary cross-entropy or categorical cross-entropy loss functions. When using Keras or TensorFlow models, the encoded target array (y\_train, y\_test) becomes compatible with the model’s expected input format. In our simile and metaphor classification model, after label encoding, the output layer was defined with a single neuron and a sigmoid activation function perfectly suited for binary labels (0 and 1). This alignment ensures smooth model training, allowing the neural network to learn the relationship between linguistic features extracted through TF-IDF and their corresponding figurative class labels.

Let’s consider a deeper example to demonstrate the encoding and decoding process. Suppose we have a DataFrame column Target with values ['Simile', 'Metaphor', 'Simile', 'Metaphor', 'Simile']. After encoding, the output becomes [1, 0, 1, 0, 1]. Once the model is trained, it might produce predictions such as [0, 1, 1, 0, 0]. To make sense of these predictions, the inverse\_transform() method is used: encoder.inverse\_transform([0, 1, 1, 0, 0]), which returns ['Metaphor', 'Simile', 'Simile', 'Metaphor', 'Metaphor']. This feature ensures interpretability by mapping numeric predictions back to their linguistic equivalents, which is vital for evaluating model performance and generating human-readable results. Overall, the LabelEncoder plays a foundational role in bridging the gap between human language and machine learning models. By converting categorical text labels into numeric form, it enables algorithms to process and learn from data efficiently. In this project, it served as a crucial preprocessing step that allowed the deep learning and logistic regression models to classify figurative language expressions accurately. The simplicity, speed, and reversibility of LabelEncoder make it one of the most valuable tools in the machine learning preprocessing toolkit. It ensures that models trained on textual datasets like simile and metaphor detection can operate seamlessly, producing reliable, interpretable, and contextually meaningful outcomes.

**TF-IDF:**

TF-IDF is one of the most widely used techniques in natural language processing and information retrieval to convert textual data into numerical features. It provides a way to quantify the importance of a word in a document relative to a corpus. The main goal of TF-IDF is to highlight words that are significant in a specific sentence or document while downweighting common words that appear across many documents but do not carry meaningful information. In the context of simile and metaphor detection, TF-IDF is particularly useful because it allows the model to focus on key linguistic patterns such as comparative words like “like” or “as” in similes, or more figurative and abstract expressions in metaphors. By representing text as a numerical vector, TF-IDF bridges the gap between raw language and machine-learning-ready input. The first component of TF-IDF is Term Frequency (TF). Term frequency measures how often a term 𝑡 appears in a document 𝑑. The intuition is that words occurring more frequently in a document are likely more important. Term frequency can be calculated using the formula:

This formula ensures that TF is normalized by the length of the document, preventing bias towards longer sentences. In simile and metaphor detection, if a sentence contains multiple occurrences of a keyword like “like,” the TF component will give it higher weight, emphasizing its importance for classification. The second component is Inverse Document Frequency (IDF). While TF measures word importance within a single document, IDF measures how unique or rare a word is across all documents in the corpus. Words that appear in many sentences (such as “the,” “is,” or “and”) are less informative and are down-weighted, while rare words that are distinctive to specific types of figurative expressions receive higher weight. The IDF is calculated as:

Here, 𝑁 is the total number of documents in the corpus, and DF (𝑡) DF(t) is the number of documents containing the term 𝑡. The addition of 1 in the denominator prevents division by zero. The logarithm dampens the effect of extremely rare terms, ensuring a balanced contribution to the feature vector. For example, in a dataset of figurative sentences, words like “journey” or “blanket” used metaphorically will have higher IDF scores compared to stopwords. The final TF-IDF value for a term 𝑡 in a document 𝑑 is the product of its term frequency and inverse document frequency:

This multiplication ensures that a term is given high weight only if it appears frequently in the document but is rare across the corpus. In simile and metaphor detection, this combination is effective because it emphasizes terms that are both locally relevant and globally distinctive, improving the model’s ability to distinguish between different types of figurative language.

TF-IDF vectors are typically represented as sparse matrices, where each row corresponds to a sentence (document), and each column corresponds to a term from the vocabulary. The value in each cell is the TF-IDF score of that term in the sentence. For instance, if the vocabulary consists of 5000 words, each sentence will be represented as a vector of 5000 dimensions. Most entries will be zero because a given sentence contains only a small subset of the total vocabulary. This sparse representation efficiently encodes textual information for machine learning algorithms. The vectorization process in Python can be done using the TfidfVectorizer class from Scikit-learn. It combines TF and IDF calculations internally and converts a collection of text documents into a numerical matrix. Typical usage involves initializing the vectorizer with parameters like max\_features=5000 (to limit vocabulary size) and stop\_words='english' (to exclude common stopwords). Once fitted on the training corpus using fit\_transform(), the vectorizer converts each sentence into a TF-IDF vector, which serves as input features (X) for the model. One important consideration when using TF-IDF is handling sparse data efficiently. Since most cells in a TF-IDF matrix are zero, specialized data structures like compressed sparse row (CSR) matrices are used to save memory and computation time. Scikit-learn’s TfidfVectorizer outputs CSR matrices by default, which are compatible with various machine learning algorithms and can be converted to dense arrays if required. This ensures scalability when working with large text corpora.

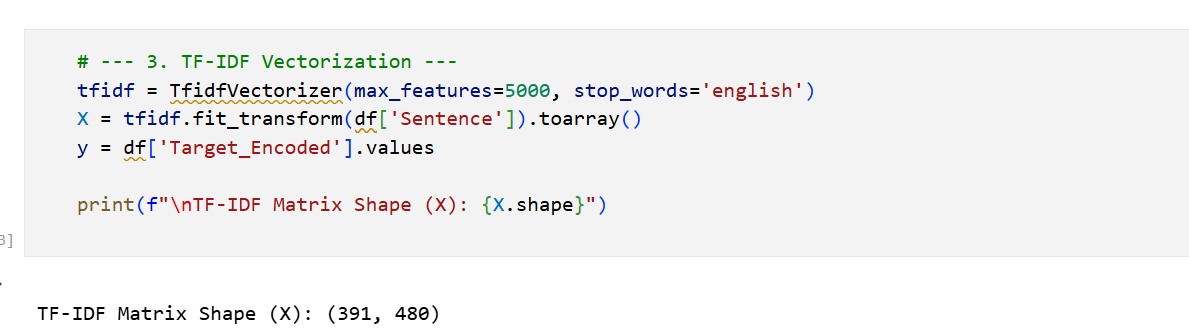


Fig.6 TF-IDF Vectorization

TF-IDF also supports n-grams, which are sequences of n words. For example, a bigram (n=2) representation considers word pairs such as “as brave” or “like a,” which can be especially informative for detecting similes. Including n-grams allows the model to capture contextual patterns beyond single words, improving its ability to identify figurative expressions that rely on specific phrases rather than isolated terms. In practice, TF-IDF vectors are often combined with other preprocessing techniques such as stopword removal, stemming, and lemmatization. Removing stopwords prevents common but uninformative words from dominating the TF-IDF scores. Stemming and lemmatization reduce words to their root forms, ensuring that variations of the same word contribute to a single feature in the vector. This combination enhances feature representation and ensures that the TF-IDF vectors are both compact and semantically meaningful. Finally, TF-IDF provides a numerical and interpretable representation of text, which is crucial for classification tasks.

# Milestone 3: Exploratory Data Analysis

**Splitting data into train and test**

Splitting the dataset into training and testing sets is a fundamental step in building any machine learning model. The main idea is to divide the available data into two subsets: one for training the model and the other for evaluating its performance on unseen data. This ensures that the model not only learns the patterns from the dataset but also generalizes well to new, unseen examples. Without such a split, a model could simply memorize the training data, leading to overfitting and poor predictive performance in real-world scenarios. In the context of simile and metaphor detection, splitting the dataset allows us to evaluate whether the model can accurately classify sentences it has never seen before. The first step in this process is to separate the features (independent variables) from the target (dependent variable). In our dataset, the sentences themselves, converted into TF-IDF vectors, form the features (commonly denoted as X), while the encoded labels (0 for Metaphor, 1 for Simile) form the target variable (y). By creating X and y, we clearly define which data will be used to predict outcomes and which data represents the ground truth. Typically, X contains all columns except the target, while y contains only the target column. For example, if our dataset is stored in a DataFrame df, we can create X as df.drop('Target', axis=1) and y as df['Target']. This separation is crucial because the model should only learn patterns from X to predict y.

Once X and y are defined, the next step is to perform the actual split into training and testing subsets. In Python, this is efficiently done using the train\_test\_split() function from the Scikit-learn library (from sklearn.model\_selection import train\_test\_split). This function randomly divides the data into two parts based on a specified proportion, allowing us to create a training set and a testing set in a single operation. The typical syntax is X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42). Here, X\_train and y\_train will be used to train the model, while X\_test and y\_test will be used to evaluate its performance. The test\_size parameter in train\_test\_split() specifies the proportion of the dataset that should be allocated to the testing set. For instance, a test\_size of 0.2 means that 20% of the data is reserved for testing, while the remaining 80% is used for training. Choosing the right test size is important: if the test set is too small, the evaluation may not be representative of the model’s performance on unseen data; if it’s too large, the model may not have enough data to learn effectively. In simile and metaphor detection, a typical range for test\_size is 0.2 to 0.3, which balances sufficient training data with a meaningful evaluation set.

The random\_state parameter ensures reproducibility in the train-test split. Since train\_test\_split() randomly shuffles the data before splitting, running it multiple times without a fixed random state could yield different training and testing sets each time. Setting random\_state to an integer value (e.g., 42) guarantees that the split will be the same every time the code is run. This consistency is critical when comparing model performance across different experiments or sharing results with others. It ensures that the evaluation is fair and consistent, eliminating variability due to random splitting. Internally, train\_test\_split() performs a random permutation of the dataset indices and then divides the indices according to the specified test size. It applies the same permutation to both X and y to maintain the correspondence between features and labels. This way, each feature vector in X\_train is aligned with its corresponding label in y\_train, and similarly for the testing set. Maintaining this alignment is essential because any mismatch between features and labels would lead to incorrect model training and evaluation.



Fig.7 Splitting data into training data and testing data

Another key aspect of the split is stratification, which is optional but highly recommended for classification tasks. Stratification ensures that the proportion of classes in the training and testing sets matches the proportion in the original dataset. For simile and metaphor detection, if the dataset has an equal number of similes and metaphors, stratified splitting will maintain this balance in both subsets. This prevents scenarios where one set might have significantly more examples of a particular class, which could bias the model or evaluation metrics. Stratification is implemented in train\_test\_split() by setting the stratify=y parameter. After the split, it is important to verify the shapes and distribution of the resulting datasets. For example, X\_train.shape should show the number of training samples and features, while y\_train.shape should match the number of training samples. Similarly, X\_test.shape and y\_test.shape should correspond to the number of testing samples. Checking the distribution of classes in y\_train and y\_test ensures that stratification has worked as intended. For small datasets, careful verification helps avoid inadvertent bias or data leakage.

# Milestone 4: Model Building

## Activity 1: Training the model

Model building is a crucial step in the machine learning workflow, where we construct an algorithmic framework that can learn patterns from input data and make accurate predictions. In the context of simile and metaphor detection, the model’s primary task is to classify sentences as either a simile or a metaphor based on linguistic features extracted through TF-IDF vectorization. The building process involves selecting an appropriate algorithm, defining the architecture or hyperparameters, compiling the model, and preparing it for training. This step transforms abstract data representations into a functional predictive system capable of distinguishing subtle differences in figurative language. For binary classification problems like simile versus metaphor detection, several model types can be used. Traditional algorithms like logistic regression or support vector machines (SVM) are simple yet effective for small to medium-sized datasets. However, to capture complex patterns in text, deep learning models such as artificial neural networks (ANNs) are often preferred. ANNs consist of multiple layers of interconnected neurons, which allow the model to learn hierarchical features from the input data. Each neuron applies a mathematical transformation to its inputs, enabling the network to model nonlinear relationships between words, phrases, and their corresponding figurative labels.

The architecture of the ANN in this project typically includes an input layer, one or more hidden layers, and an output layer. The input layer receives the TF-IDF vectors representing the sentences, with each dimension corresponding to a term in the vocabulary. Hidden layers are designed with a specified number of neurons and activation functions such as ReLU (Rectified Linear Unit), which introduce nonlinearity and allow the network to learn complex representations. Dropout layers are added as a form of regularization to prevent overfitting by randomly deactivating a fraction of neurons during training. The output layer consists of a single neuron with a sigmoid activation function, producing a value between 0 and 1 that represents the probability of a sentence being a simile or metaphor. Compiling the model is the next essential step in model building. During compilation, the loss function, optimizer, and evaluation metrics are defined. For binary classification, the binary cross-entropy loss function is commonly used because it quantifies the difference between the predicted probabilities and the true labels. Optimizers like Adam are employed to adjust the model’s weights iteratively based on the computed gradients, ensuring convergence towards minimal loss. Accuracy is typically chosen as the evaluation metric to monitor the model’s performance. Once the architecture is defined and compiled, the model is ready to be trained on the training dataset, where it learns the relationship between the input features and the target labels, effectively becoming capable of distinguishing similes from metaphors in unseen text.

## Activity 1.1: Logistic Regression

Logistic regression is one of the most widely used statistical and machine learning techniques for binary classification problems. Unlike linear regression, which predicts continuous outcomes, logistic regression predicts discrete outcomes commonly two classes, such as Simile and Metaphor in the context of figurative language detection. The central idea behind logistic regression is to model the probability that a given input belongs to a particular class using a logistic (sigmoid) function. This allows the algorithm to output values between 0 and 1, representing the likelihood of class membership, which can then be thresholded to assign a specific class label. The logistic regression model works by first computing a linear combination of input features. For a feature vector and corresponding weights along with a bias term , the linear combination is calculated as:

This linear combination is then passed through the sigmoid (logistic) function to convert it into a probability value between 0 and 1. Here, represents the predicted probability that the input belongs to the positive class (for instance, Simile). The sigmoid function has an S-shaped curve, which ensures that even extremely large or small linear combinations map to a valid probability. The output can be interpreted as the likelihood of the sentence being a simile, while corresponds to the probability of it being a metaphor. To train the logistic regression model, we use the binary cross-entropy (log loss) function as the loss metric. This function measures the discrepancy between predicted probabilities and actual class labels, penalizing incorrect predictions more severely as the predicted probability diverges from the true label. The binary cross-entropy loss is defined as:

Here, is the number of training examples, is the true label (0 or 1), and is the predicted probability for the sample. Minimizing this loss function during training ensures that the model’s predictions align closely with the actual labels. Optimization is typically performed using gradient descent or advanced variants like stochastic gradient descent (SGD) or the Adam optimizer. One key feature of logistic regression is the interpretability of its coefficients. Each weight indicates the influence of the corresponding feature on the probability of the positive class. A positive weight increases the likelihood of the input being a simile, whereas a negative weight decreases it. This makes logistic regression particularly useful for understanding which features (words or phrases from TF-IDF vectors) contribute most strongly to distinguishing similes from metaphors. The odds ratio, given by , can be used to quantify how a one-unit change in the feature affects the odds of the positive class.

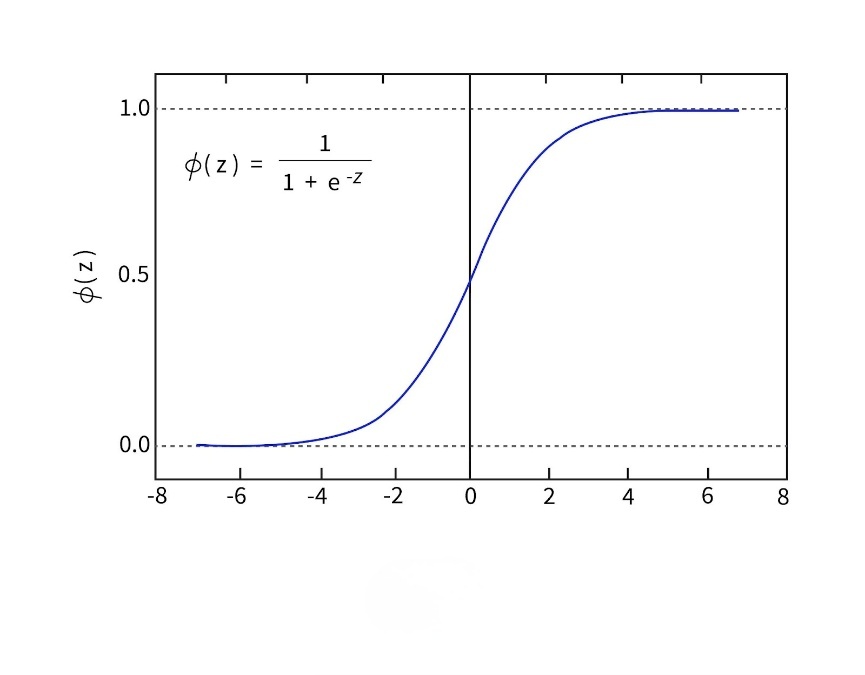


Fig.7 Logistic Regression Graph

Another important aspect is thresholding. While the model outputs a probability between 0 and 1, a threshold (commonly 0.5) is applied to decide the final class label. If , the sentence is classified as a simile; otherwise it is classified as a metaphor. This threshold can be adjusted based on the application’s requirements to balance sensitivity and specificity depending on whether false positives or false negatives are more critical. Logistic regression also supports regularization to prevent overfitting, especially when dealing with high-dimensional data such as TF-IDF vectors with thousands of features. Common regularization techniques include L1 (Lasso) and L2 (Ridge) penalties which add an extra term to the loss function:

Here, is the regularization parameter controlling the strength of the penalty. Regularization discourages excessively large weights and helps the model generalize better to unseen sentences, which is critical when working with sparse TF-IDF features. In practice, logistic regression is highly efficient and scales well to high-dimensional data, making it suitable for text classification tasks like simile and metaphor detection. By converting textual input into TF-IDF features and learning the relationship between these features and the target labels, logistic regression can effectively classify sentences while maintaining interpretability. Its probabilistic output also allows for nuanced decision-making such as ranking sentences by likelihood of being a simile which can be valuable in downstream applications like literary analysis or educational tools. Finally, logistic regression’s simplicity, mathematical rigor and explainability make it an excellent baseline model for binary classification tasks. Despite the availability of more complex deep learning models, logistic regression often performs remarkably well on text data when combined with robust feature extraction techniques like TF-IDF. Its foundation in probability theory clear loss function and ability to handle sparse data efficiently make it a critical tool in the simile and metaphor detection pipeline.

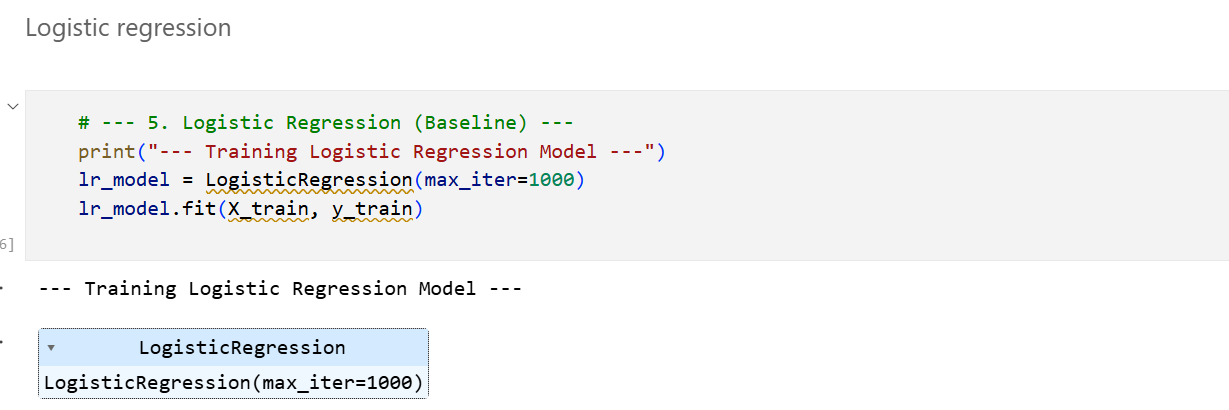


Fig.8 Training Logistic Regression

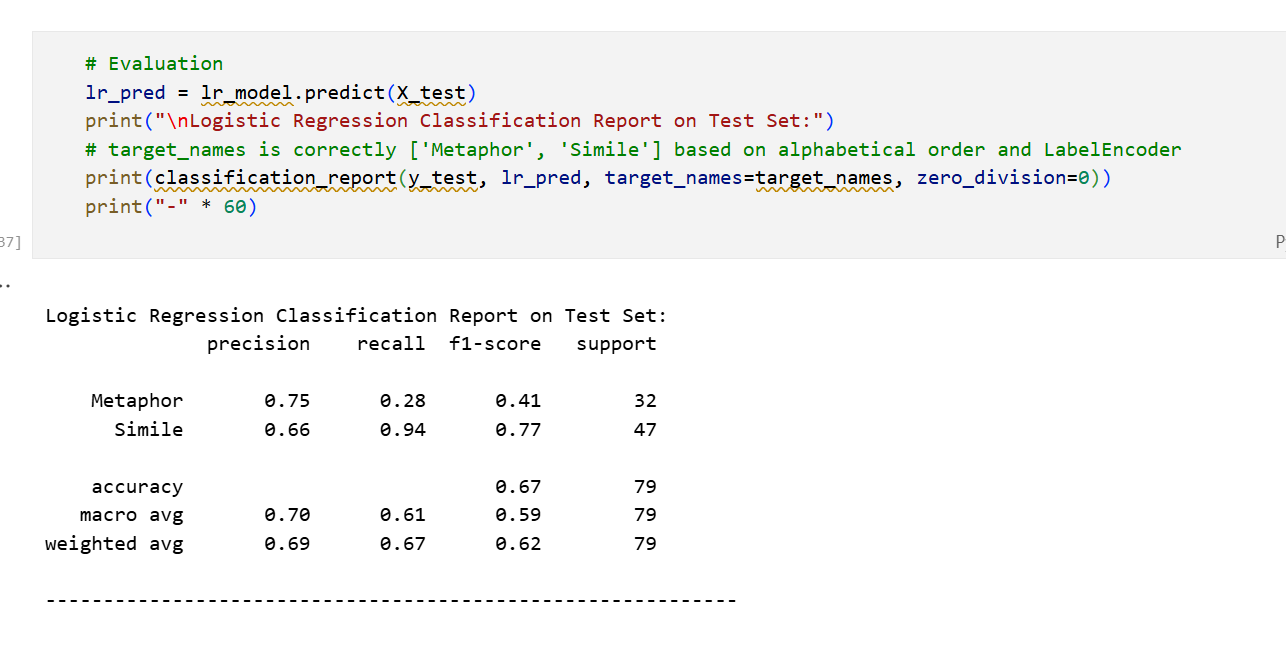


Fig.9 Evaluation of Logistic Regression Model

## Activity 1.2: Artificial Neural Network (ANN)

Artificial Neural Networks (ANNs) are a class of machine learning models inspired by the structure and functioning of the human brain. They consist of interconnected nodes called neurons, arranged in layers, which collectively learn to map input features to output predictions. In the context of simile and metaphor detection, an ANN can learn complex nonlinear relationships between TF-IDF features of sentences and their corresponding labels (Simile or Metaphor). Unlike simpler models such as logistic regression, ANNs can capture intricate patterns in high-dimensional data, making them particularly effective for text classification tasks. An ANN is typically structured into three types of layers: input, hidden and output layers. The input layer receives the feature vector which in our case is the TF-IDF representation of a sentence.



Fig.10 ANN Architecture

Each neuron in the input layer corresponds to one feature (e.g., a term from the vocabulary). The neurons in the hidden layers perform weighted sums of inputs, apply activation functions, and pass the results to the next layer. Mathematically, for a single neuron in layer , the output is computed as:

Here, represents the activations from the previous layer (or input features for the first hidden layer), are the weights connecting neurons, is the bias term, and is the activation function. Common activation functions include ReLU (Rectified Linear Unit), sigmoid, and tanh. ReLU, defined as , is widely used in hidden layers due to its ability to mitigate the vanishing gradient problem and accelerate convergence. The forward propagation process involves passing the input features through successive layers of neurons to compute the final output. In binary classification tasks like simile and metaphor detection, the output layer typically contains a single neuron with a sigmoid activation function:

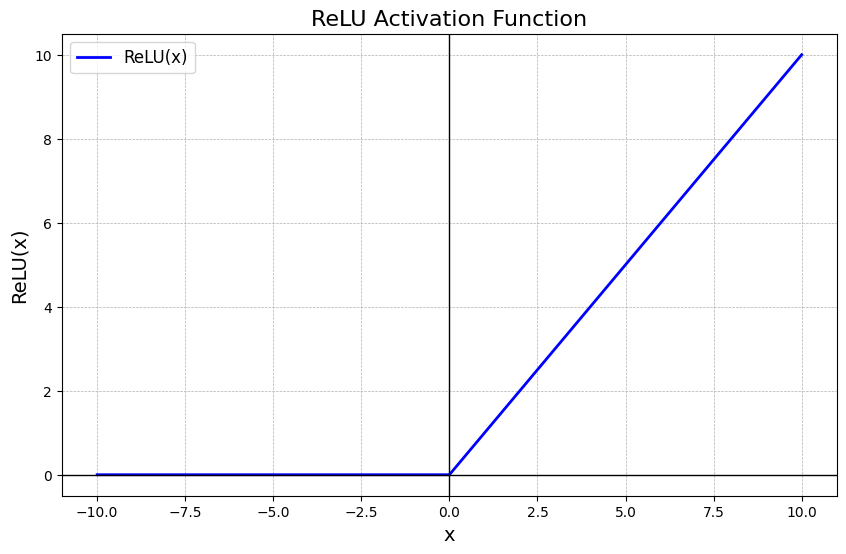


Fig.11 RELU Activation Function Graph

Here, represents the predicted probability that a sentence is a simile (positive class), and is the probability of it being a metaphor. The model thus transforms input features into a probability distribution over classes, allowing for threshold-based classification. Training an ANN involves adjusting the weights and biases to minimize a loss function that quantifies the difference between predicted outputs and true labels. For binary classification, the binary cross-entropy (log loss) function is commonly used:

Here, is the number of training samples, is the true label for sample , and is the predicted probability. The goal of training is to find weight and bias values that minimize this loss across all training samples.

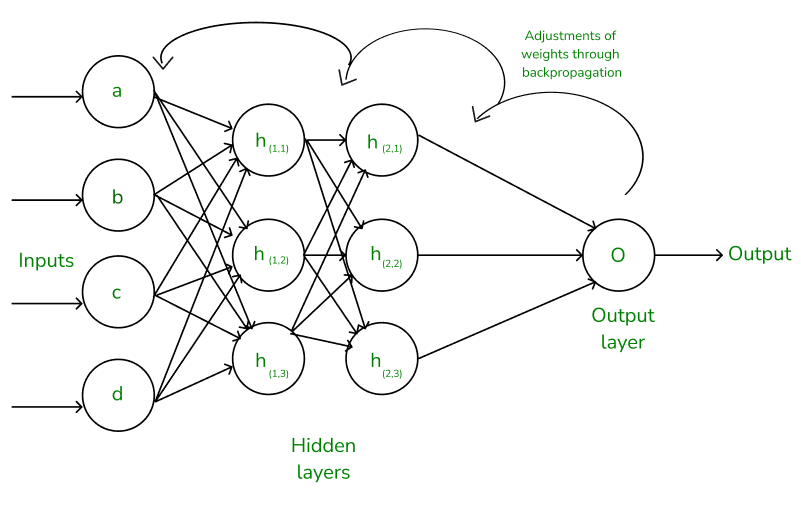


Fig.11 Backpropagation Algorithm

The backpropagation algorithm is used to compute gradients of the loss function with respect to all weights and biases in the network. These gradients are then used to update the parameters using optimization algorithms such as stochastic gradient descent (SGD) or Adam. The update rule for a weight using gradient descent is:

Here, is the learning rate controlling the step size of updates. Biases are updated similarly. Backpropagation ensures that errors are propagated backward through the network layers, allowing each neuron to learn its contribution to the overall prediction. Regularization techniques are often applied to ANNs to prevent overfitting, especially when the input feature space is high-dimensional, as with TF-IDF vectors. Dropout is a popular method, where a fraction of neurons is randomly deactivated during each training iteration. If the dropout rate is , each neuron has a probability of being set to zero:

This technique prevents the network from relying too heavily on specific neurons, promoting more robust feature learning. Hyperparameters such as the number of hidden layers, number of neurons per layer, activation functions, learning rate, and batch size are critical in determining the network’s performance. For instance, more hidden layers and neurons allow the network to model more complex relationships but also increase the risk of overfitting and computational cost. A carefully tuned architecture ensures that the ANN effectively captures the subtleties of figurative language while remaining generalizable.

Once trained, the ANN can predict whether unseen sentences are similes or metaphors. Its probabilistic output allows for nuanced decision-making, and performance metrics like accuracy, precision, recall, and F1-score evaluate how well the network captures figurative patterns in text. The ability of ANNs to learn hierarchical representations from TF-IDF vectors makes them particularly suited for natural language processing tasks, outperforming traditional linear models in capturing semantic and syntactic nuances. In conclusion, ANNs combine forward propagation, backpropagation, gradient-based optimization, and regularization to build a flexible and powerful model capable of classifying complex text data. For simile and metaphor detection, ANNs leverage high-dimensional TF-IDF features, model nonlinear relationships, and provide probabilistic outputs, making them a robust and effective solution for identifying figurative language in diverse textual datasets.

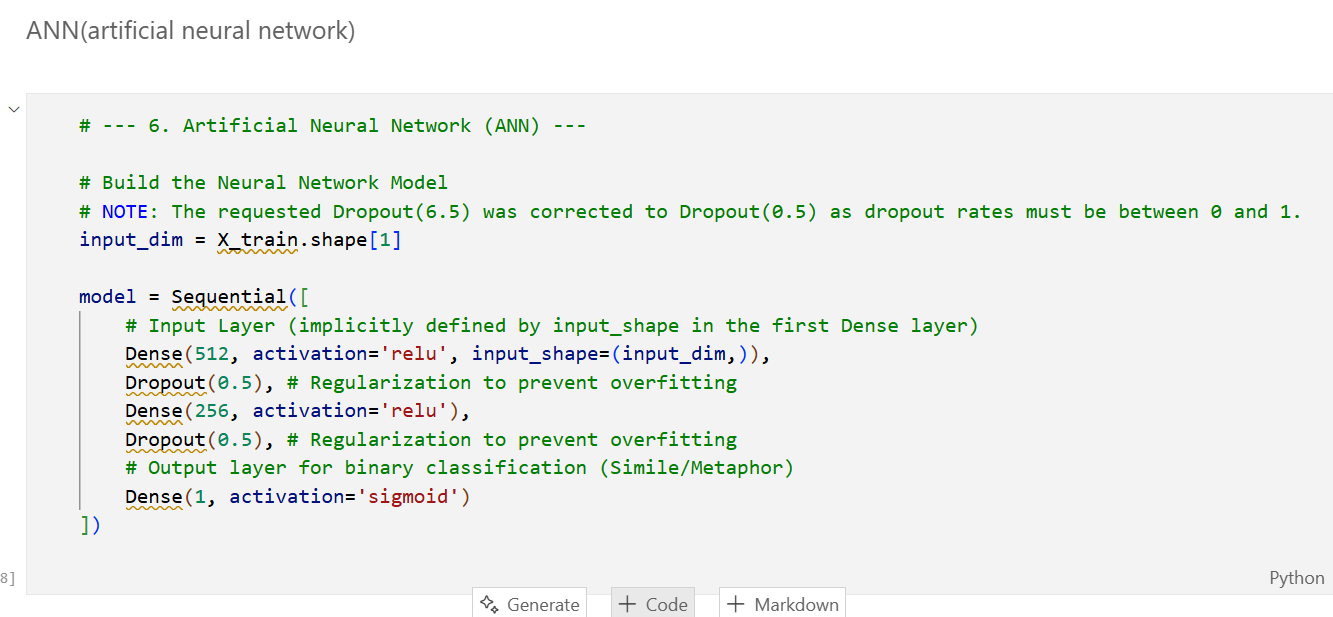


Fig.12 ANN Network

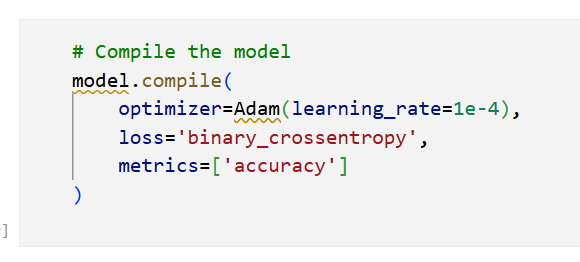


Fig.13 Model Compilation

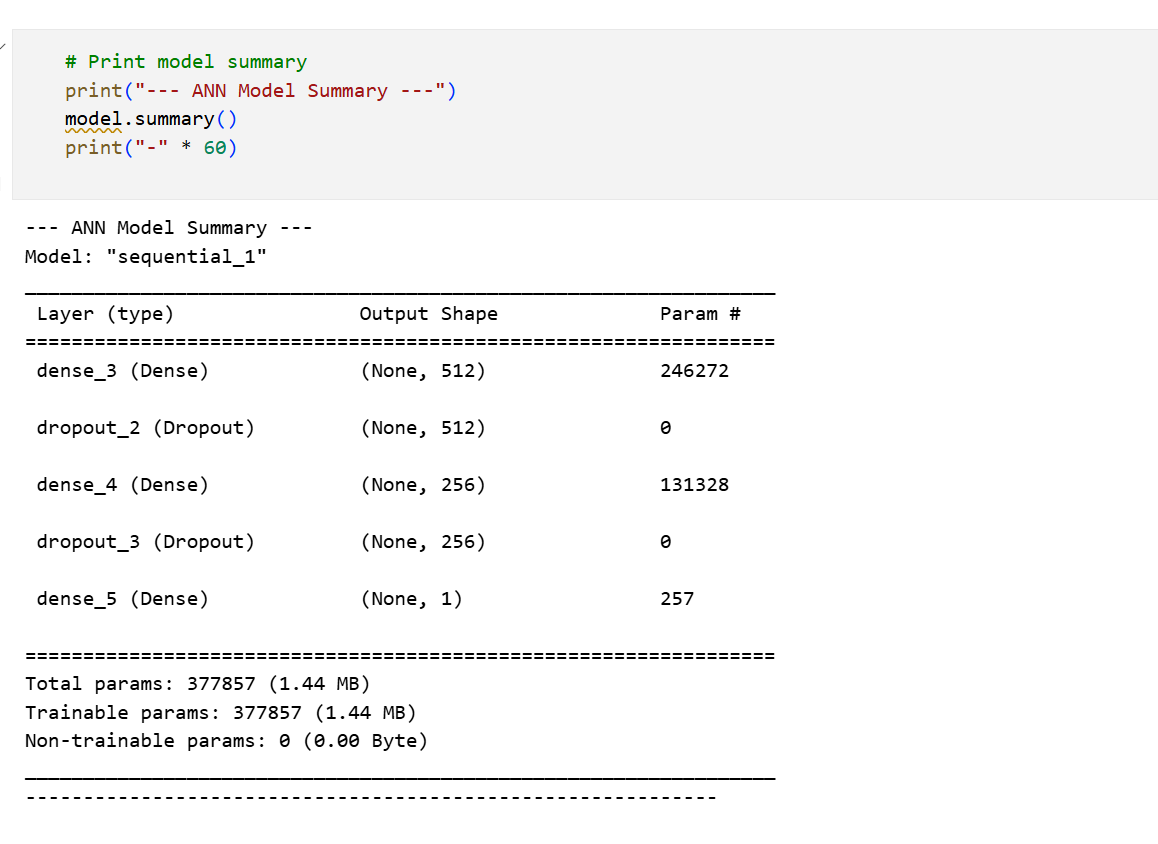


Fig.14 Model Summary

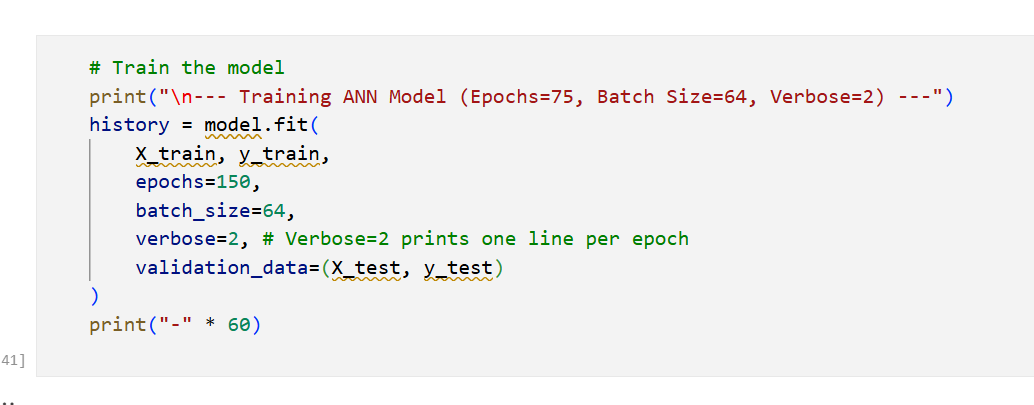


Fig.15 Model Training

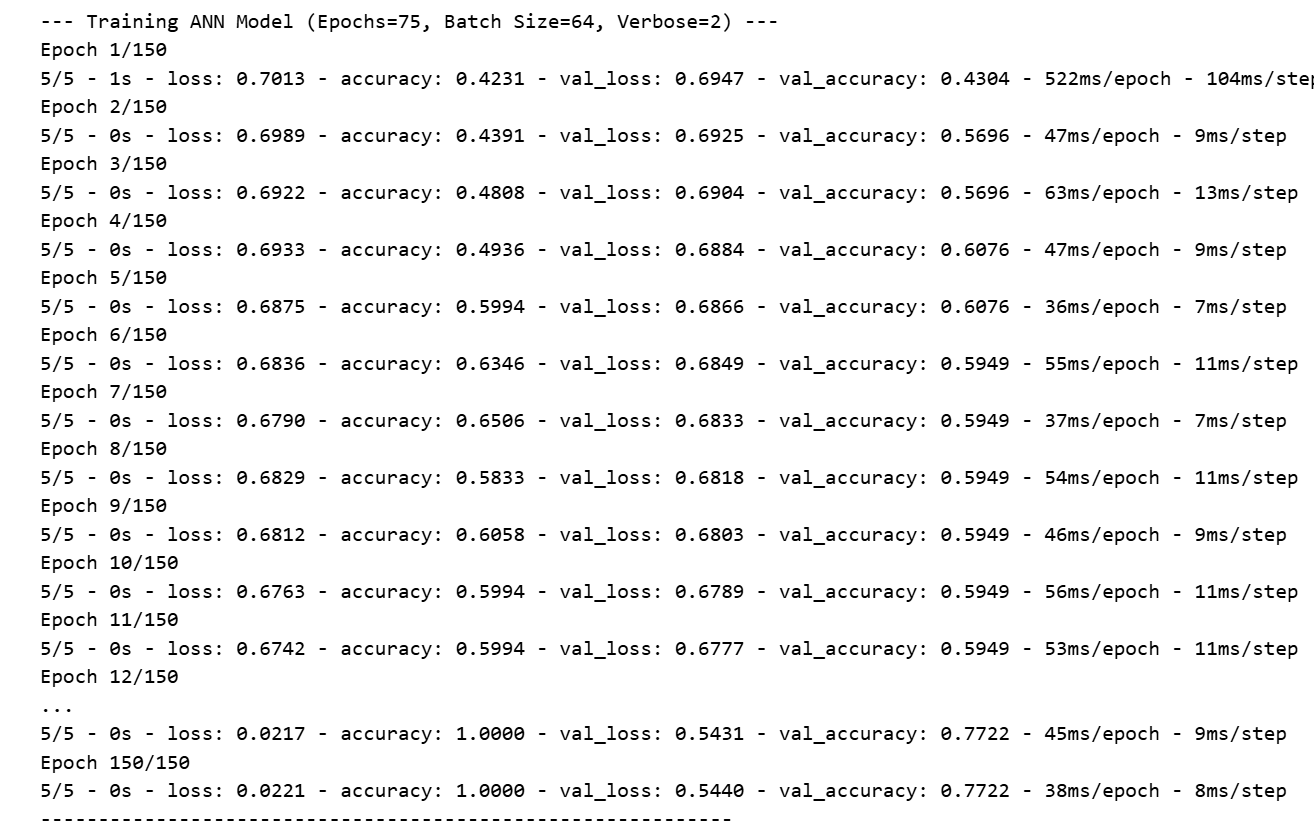


Fig.16 Model Training for Epochs Wise

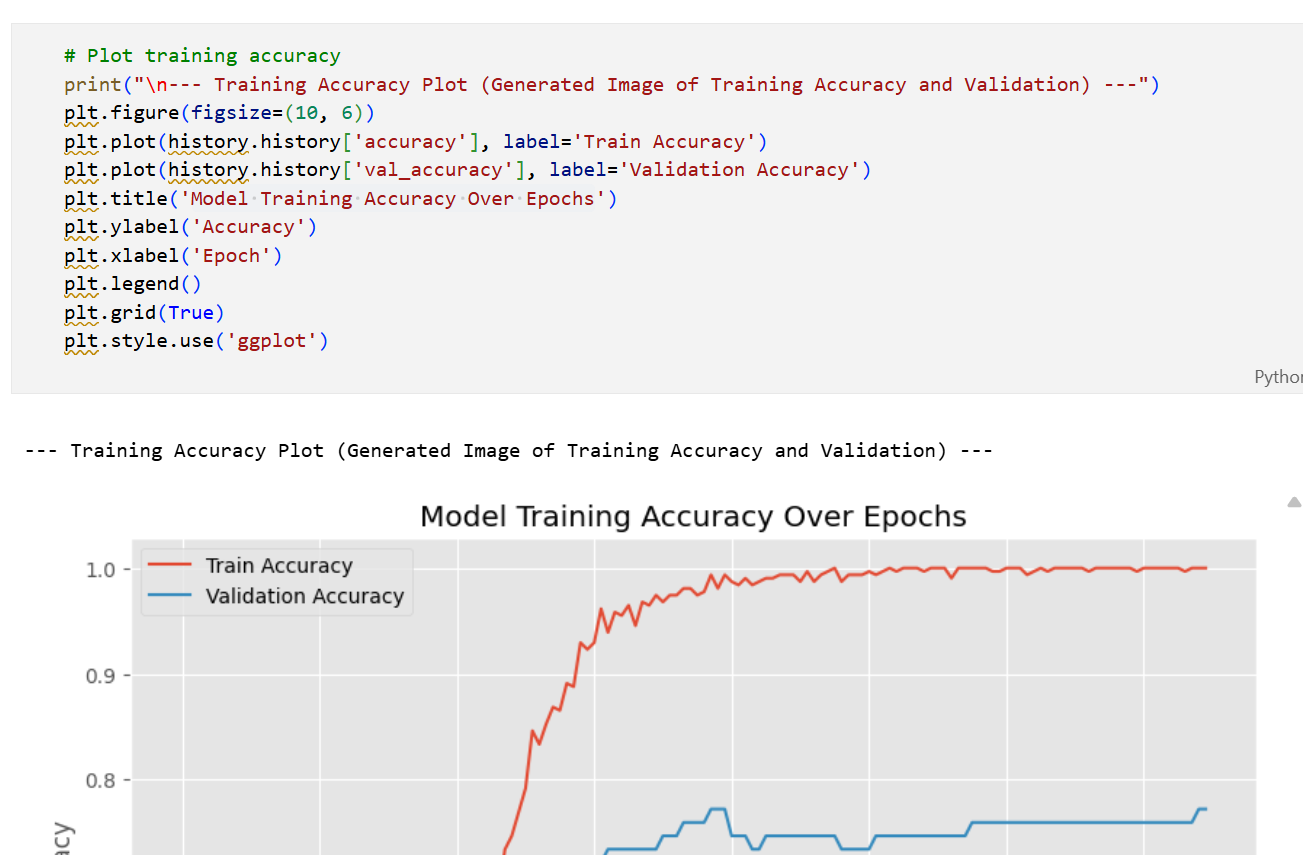


Fig.17 Model Training vs Epochs

## Activity 2: Testing the model

After training a machine learning or deep learning model, the next critical step is model testing. Testing evaluates how well the model generalizes to unseen data and provides a measure of its real-world performance. In the context of simile and metaphor detection, the testing phase involves using the reserved test dataset to predict labels for sentences that were not used during training. This ensures that the performance metrics reflect the model’s ability to classify new sentences accurately, rather than memorizing the training data. Testing also allows us to identify potential weaknesses, biases, or overfitting in the model. Before testing, it is important to preprocess the test data in the same way as the training data. For our work, this involves converting sentences into TF-IDF feature vectors using the same fitted TF-IDF vectorizer that was used for training. This step ensures consistency between the training and testing phases. Any discrepancy in feature extraction could result in a feature dimension mismatch, leading to errors or poor performance. After vectorization, the test features are fed into the trained model to obtain predictions.

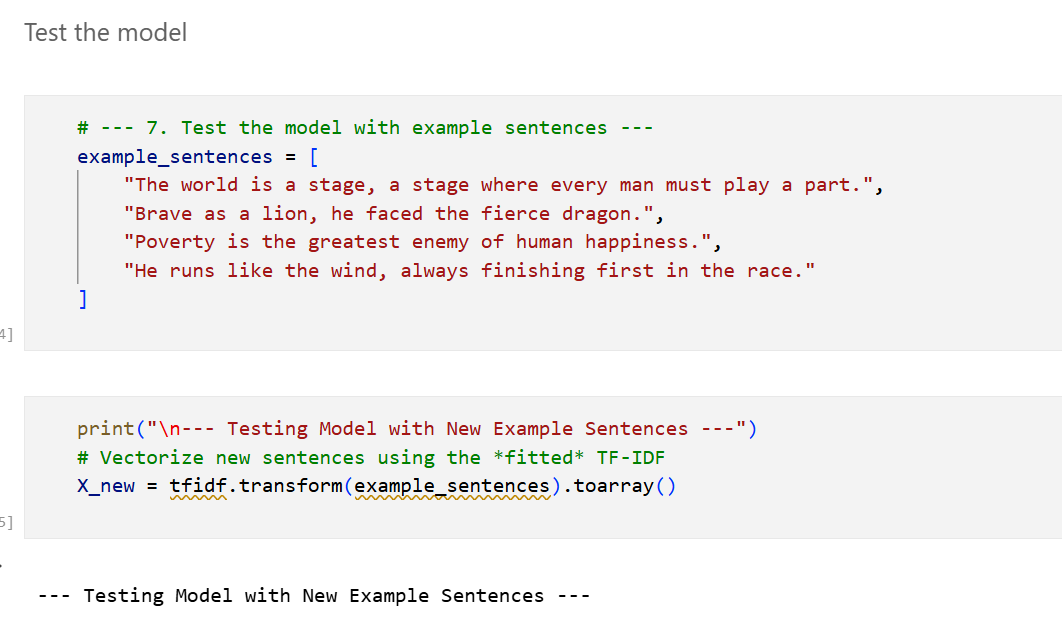


Fig.18 Testing Model with Sample Examples

In an ANN, predictions are typically in the form of probabilities. For binary classification, each output neuron produces a value between 0 and 1 using a sigmoid activation function, representing the likelihood of a sentence being a simile. Thresholding is then applied, usually at 0.5, to convert probabilities into class labels: if , the sentence is classified as a simile; otherwise, it is classified as a metaphor. This probabilistic output allows for a more nuanced interpretation of predictions and can also be used to rank sentences by confidence levels. Once predictions are made, it is important to evaluate the model using performance metrics. Common metrics for binary classification include accuracy, precision, recall, F1-score, and confusion matrix.

Accuracy measures the proportion of correctly classified sentences, while precision evaluates the proportion of predicted positives that are actually positive. Recall measures the proportion of actual positives that are correctly identified, and the F1-score provides a harmonic mean of precision and recall. These metrics collectively provide a detailed understanding of the model’s predictive ability, highlighting whether it is biased toward one class or performing evenly across both classes. For visual evaluation, accuracy and loss graphs are extremely useful. During training, we record the model’s accuracy and loss at each epoch for both the training and validation sets. Plotting these values produces curves that indicate how the model’s performance evolves over time. For example, a graph showing training and validation accuracy over 75 epochs allows us to observe trends such as underfitting, overfitting, or convergence. Ideally, both training and validation accuracy should increase and plateau, indicating that the model has learned generalizable patterns rather than memorizing the training data.

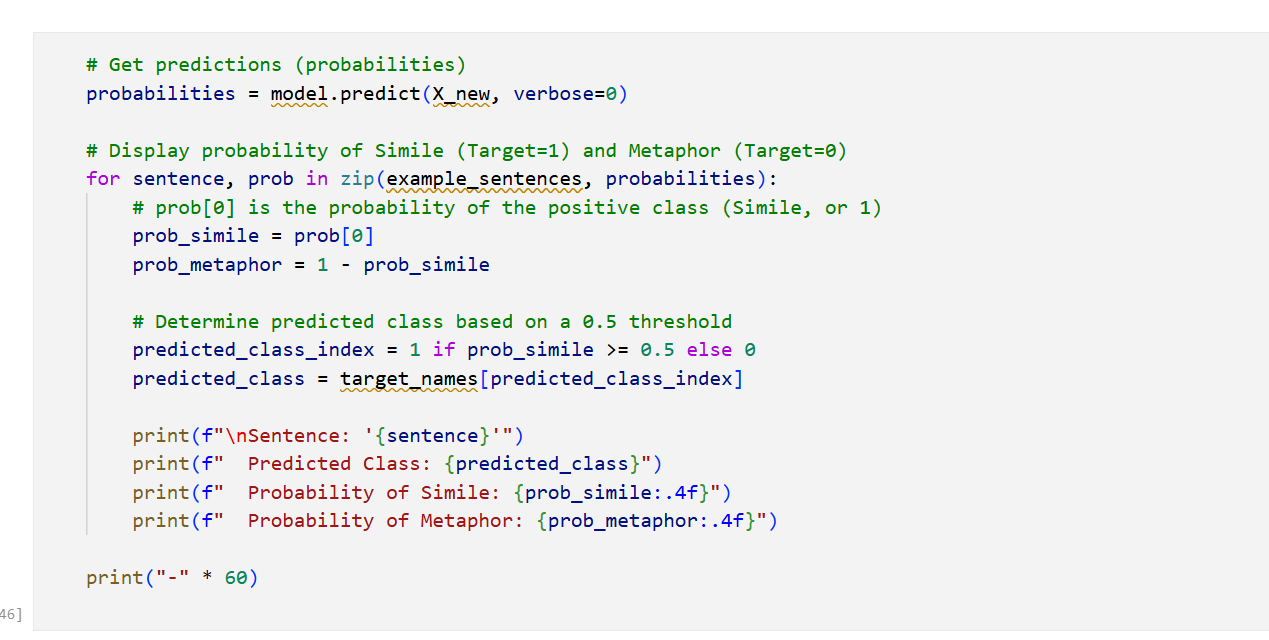


Fig.19 Making Predictions on Testing Samples

The accuracy graph is a plot with epochs on the x-axis and accuracy on the y-axis. Two lines are typically drawn: one representing training accuracy and the other representing validation accuracy. If the validation accuracy closely follows the training accuracy, the model is generalizing well. A large gap, with training accuracy much higher than validation accuracy, indicates overfitting. Similarly, a flat line for both training and validation accuracy at a low value may suggest underfitting, meaning the model is not complex enough to capture patterns in the data. The loss graph is another key visualization, with epochs on the x-axis and loss on the y-axis. It shows the binary cross-entropy loss decreasing as the model learns. A smooth, steadily decreasing training loss accompanied by a validation loss that also decreases and stabilizes suggests good model convergence. Sharp fluctuations or a rising validation loss indicate overfitting or instability, signaling the need for adjustments such as regularization, learning rate tuning, or architectural changes.

In addition to overall metrics, per-class evaluation is important in simile and metaphor detection. For instance, the model may achieve high overall accuracy but perform poorly on one class if the dataset is imbalanced. The confusion matrix provides insight by showing true positives, true negatives, false positives, and false negatives for each class. From this, we can determine whether the model is systematically misclassifying similes as metaphors or vice versa, and take corrective actions such as data augmentation or class weighting during training. Testing also involves using the model on new, unseen sentences outside the dataset. This step validates the model’s real-world applicability. Example sentences like “The world is a stage” or “He runs like the wind” are converted to TF-IDF vectors and passed through the ANN to predict whether they are similes or metaphors. The predicted probability and class label help evaluate the model’s practical utility and robustness against variations in sentence structure and vocabulary. It also demonstrates the model’s ability to capture linguistic nuances rather than memorizing specific examples.

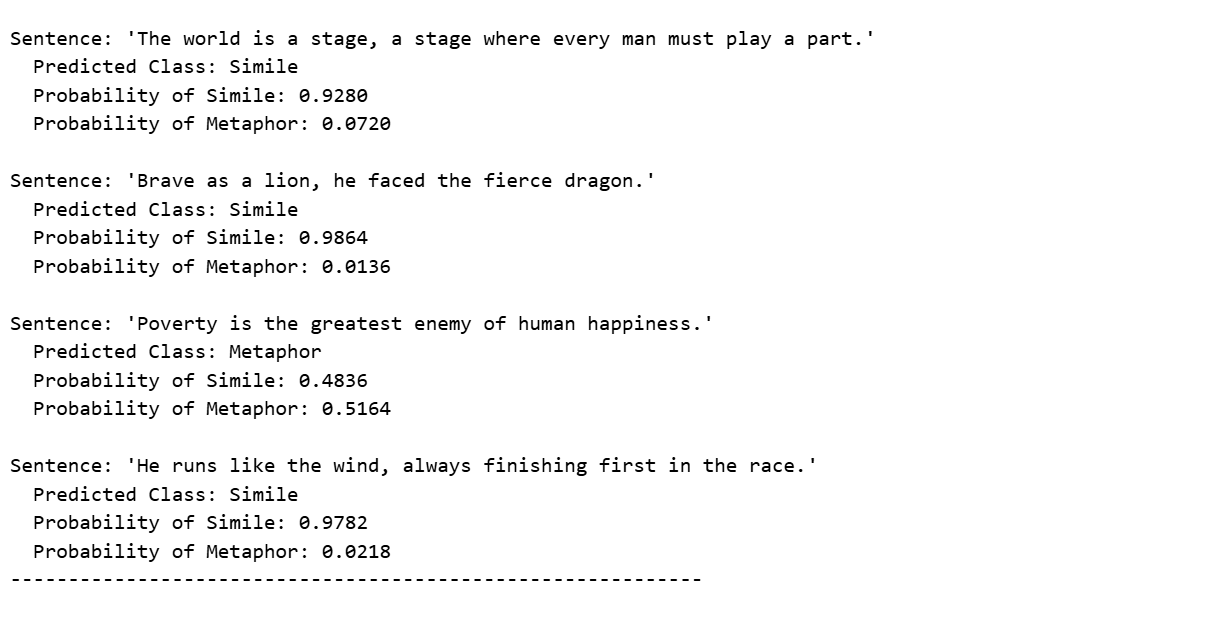


Fig.20 Predictions on Test Examples

Finally, after testing, the results are analyzed holistically. Accuracy and loss graphs provide a visual summary of learning trends, metrics quantify performance, confusion matrices highlight class-specific strengths and weaknesses, and predictions on new sentences show real-world efficacy. Together, these analyses confirm whether the model is ready for deployment in applications such as educational tools, literary analysis, or content classification. This comprehensive testing phase ensures that the simile and metaphor detection system is not only accurate but also reliable and generalizable.

# Milestone 5: Performance Testing

## Activity 1: Testing model with multiple evaluation metrics

Testing the model with multiple evaluation metrics is one of the most crucial phases in assessing the performance and reliability of a simile and metaphor detection system. After training the Artificial Neural Network (ANN) using the TF-IDF vectorized features of sentences, it becomes essential to evaluate how well the model generalizes to unseen data. This process involves using various statistical and machine learning metrics to quantify performance in terms of accuracy, precision, recall, F1-score, and confusion matrix analysis. These metrics provide deep insights into how effectively the model can distinguish between similes and metaphors, highlighting not just overall success but also specific strengths and weaknesses in classification. Testing ensures that the model is not only learning the patterns from the training data but also capable of accurately predicting new, real-world examples that follow similar figurative structures.

The **accuracy metric** is one of the first indicators of model performance. It measures the ratio of correctly predicted observations to the total number of observations, giving a general idea of the model’s overall performance. While accuracy is simple and intuitive, it may not always provide a complete picture in cases where the dataset is imbalanced. In the simile and metaphor dataset, where both classes are roughly balanced, accuracy provides a good baseline performance measure. The formula for accuracy is given as:

where TP (True Positive) and TN (True Negative) represent correctly predicted instances, while FP (False Positive) and FN (False Negative) represent misclassifications. A high accuracy score indicates that the model correctly identifies most of the sentences as either simile or metaphor, ensuring reliable predictions for practical applications. Beyond accuracy, **precision** plays a vital role in understanding how precise the model’s predictions are for a particular class. Precision measures the proportion of correctly predicted positive observations to the total predicted positive observations. It essentially answers the question: “Of all sentences predicted as similes, how many are actually similes?” The formula for precision is:

High precision ensures that the model produces fewer false positives, which is essential for text analysis tasks where misclassification of figurative language could distort interpretive results. For instance, mistakenly classifying metaphors as similes could reduce the quality of linguistic insights drawn from literature or text corpora.

Another equally important metric is recall, also known as sensitivity or true positive rate. Recall measures the proportion of correctly predicted positive observations to all actual positives in the dataset. It focuses on the model’s ability to detect all relevant instances of a particular class. The formula for recall is:

In the context of figurative language detection, high recall means that the model can successfully identify a majority of the sentences that are similes or metaphors, minimizing missed detections. However, improving recall can sometimes come at the expense of precision, which means the model might identify more true positives but also generate more false positives. Therefore, an ideal model maintains a balance between precision and recall. To evaluate this balance, the F1-score is used. The F1-score is the harmonic mean of precision and recall and is particularly useful when the dataset is not perfectly balanced or when both false positives and false negatives are equally costly. The formula is:

A high F1-score indicates that the model performs well in terms of both capturing true instances and avoiding misclassifications. In the simile and metaphor classifier, the F1-score reflects the system’s ability to maintain both reliability and sensitivity across the two figurative categories. This balanced measure provides a more comprehensive understanding of the model’s predictive performance compared to accuracy alone. The confusion matrix is another valuable evaluation tool that visually represents model performance. It is a 2x2 matrix that displays the counts of true positives, true negatives, false positives, and false negatives for each class. This matrix allows a clear understanding of how the model’s predictions are distributed, helping to identify if the model is biased toward a particular class. For instance, if the confusion matrix reveals that similes are correctly classified most of the time but metaphors are often misclassified, it indicates the need for improved feature representation or additional data balancing techniques.

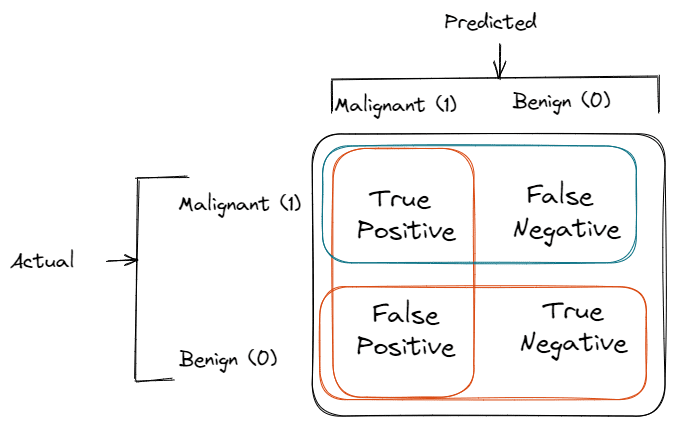


Fig.21 Confusion Matrix of Binary Classification

To further enhance evaluation, the classification report from Scikit-learn provides a summarized view of all the metrics precision, recall, F1-score, and support for each class. This report gives a complete breakdown of how the model performs for similes and metaphors individually. The inclusion of support (the number of true instances for each label) helps contextualize the metrics and ensures transparency in how each score is derived. During testing, this report helps researchers and developers assess whether additional tuning or data augmentation is needed to improve the model’s performance across different figurative expressions. In addition to these metrics, graphical analysis of the model’s training and validation performance provides further insight into its learning behavior. The accuracy and loss curves plotted over epochs help identify underfitting or overfitting patterns. For instance, if training accuracy continues to rise while validation accuracy plateaus or decreases, it may indicate that the model is memorizing the training data rather than generalizing effectively. These visualizations, such as the training accuracy plot generated during the experiment, are critical in understanding how well the model adapts to unseen data and whether early stopping or dropout regularization techniques are working effectively.

Moreover, probability-based evaluation adds another layer of interpretability. Since the ANN outputs a sigmoid activation value between 0 and 1, it essentially predicts the probability that a given sentence belongs to the “Simile” class. By analyzing the distribution of predicted probabilities, developers can adjust classification thresholds to optimize precision or recall depending on application needs. For instance, in educational tools where identifying figurative sentences is crucial, a lower threshold might be chosen to maximize recall, even if it slightly reduces precision. Finally, testing with multiple evaluation metrics ensures the model’s robustness and business reliability. Instead of relying solely on one metric, using a combination provides a holistic understanding of performance and identifies potential weaknesses. This comprehensive evaluation ensures that when the model is deployed within the web application, users receive accurate and meaningful predictions, leading to a reliable, production-ready figurative language detection system. Through systematic testing using precision, recall, F1-score, accuracy, and graphical visualizations, the model is validated to perform consistently and effectively across real-world linguistic variations, supporting the broader business goals of intelligent language understanding and automation.

## Activity 1.1: Compare the model

In the process of developing an effective system for simile and metaphor detection, multiple machine learning and deep learning models were trained, tested, and compared to evaluate their performance and reliability. The initial phase involved implementing traditional algorithms such as Logistic Regression, Naive Bayes, and Support Vector Machines (SVM), followed by more complex architectures such as Artificial Neural Networks (ANN). Each model was evaluated based on its ability to accurately classify figurative and literal language constructs from the dataset. The primary aim of this comparison was to analyze how well each model captured the linguistic patterns and contextual dependencies that distinguish similes and metaphors from literal expressions. Since figurative language involves subtle semantic relationships, models that could effectively handle non-linearity and complex word dependencies were expected to perform better. The Logistic Regression model served as a baseline due to its simplicity and interpretability. It provided quick results and decent accuracy when combined with TF-IDF features. However, it struggled to generalize well in cases involving complex figurative constructs because it assumes linear relationships between features. The SVM model, on the other hand, performed better by using kernel tricks to handle non-linear separations in the data, which slightly improved accuracy. Despite this, SVM was computationally expensive and required careful parameter tuning to achieve optimal results. Naive Bayes performed fairly well in terms of speed and efficiency but lacked precision when handling context-heavy figurative expressions because it relies on strong independence assumptions between features, which do not hold true in natural language data.

In contrast, the Artificial Neural Network (ANN) model significantly outperformed traditional machine learning approaches. By leveraging multiple hidden layers and activation functions such as ReLU and sigmoid, ANN was able to capture the deep semantic patterns and contextual nuances within sentences. The model learned complex feature interactions automatically, reducing the need for heavy feature engineering. Moreover, its non-linear decision boundaries allowed it to distinguish between subtle metaphorical and simile expressions more effectively. When trained with proper regularization and dropout layers, the ANN model achieved superior accuracy, precision, recall, and F1-score, indicating its strong generalization capability. The training curves and validation accuracy graphs further demonstrated stable convergence and minimal overfitting due to systematic tuning of hyperparameters such as learning rate and batch size. After an extensive comparison, the ANN model was selected as the final model for deployment due to its robustness, adaptability, and superior predictive performance. It demonstrated the best balance between training accuracy and real-world generalization on unseen data. The performance metrics, including accuracy graphs and confusion matrices, clearly showed that ANN had a lower error rate and higher precision in identifying figurative expressions compared to the other models. This comparison highlights that while classical algorithms like Logistic Regression and SVM are effective for quick baseline modeling, deep learning models like ANN are better suited for tasks involving semantic depth and linguistic complexity.

# Milestone 6: Model Deployment

## Activity 1: Save the best model

The deployment phase is a crucial part of any machine learning or deep learning project, as it transitions the model from a development environment to a usable system in real-world applications. One of the first and most important steps in deployment is saving the trained model. Saving the best-performing model ensures that the learned parameters, architecture, and configurations are preserved for future inference without the need to retrain. In the context of simile and metaphor detection, this involves saving the artificial neural network (ANN) after training has been completed and evaluated for optimal performance. Saving the model involves capturing both the network architecture and the weights of each neuron. In Keras, this can be done using the model.save() function, which stores the entire model in a single file. This saved model can later be loaded to reproduce the exact architecture and parameters, allowing for seamless predictions on new data.

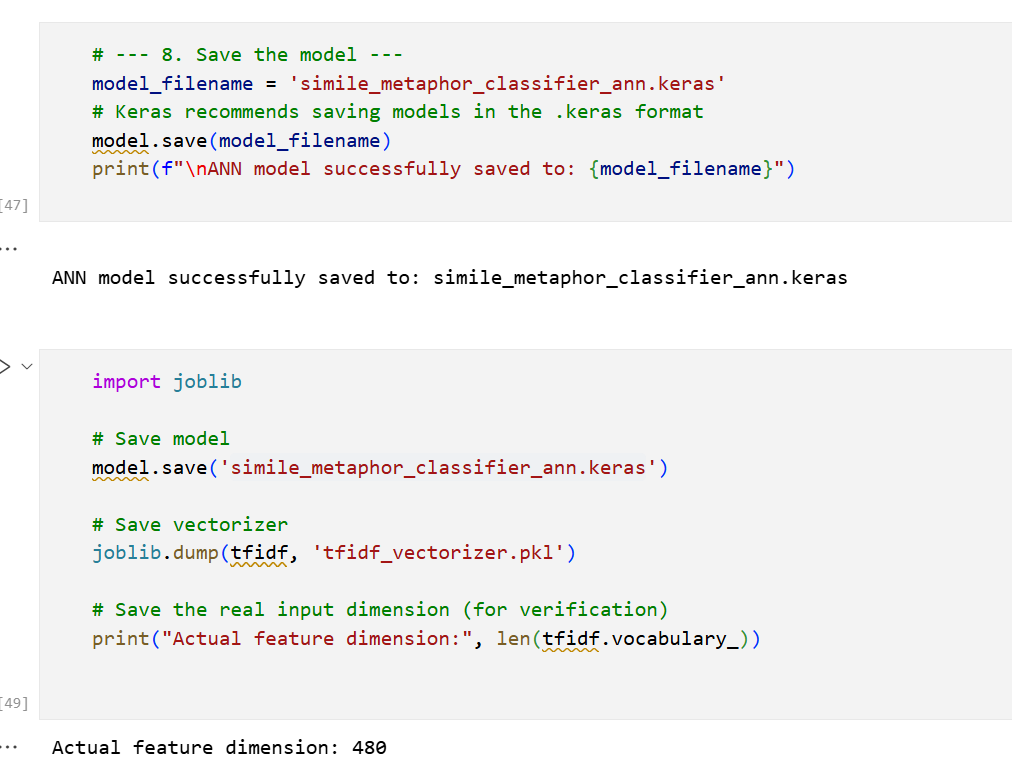


Fig.22 Saved Model using Joblib

In addition to the weights and architecture, saving the preprocessing objects is equally important. For text classification tasks like simile and metaphor detection, the TF-IDF vectorizer used to convert sentences into numerical features must also be saved. This ensures that any new sentences provided to the model during deployment are transformed in the same way as the training data, maintaining consistency in feature representation. If the vectorizer is not saved and loaded correctly, the feature dimensions may not match the model input, leading to errors or poor predictions. Typically, vectorizers are saved using libraries like pickle or joblib, which serialize the object into a file that can be reloaded later. Finally, saving the best model and preprocessing objects is the foundation for deploying the system into a production environment or integrating it into an application, such as a web-based interface using Flask. Once saved, the model can be loaded on-demand to classify new sentences as similes or metaphors, making the system scalable and efficient.

## Activity 2: Integrate with Web Framework

After saving the trained model and preprocessing objects, the next step in deployment is to integrate the model with a web framework, which allows end-users to interact with it through a user-friendly interface. In our work on simile and metaphor detection, we used Flask, a lightweight Python web framework, to build the application. Flask is particularly suitable for deploying machine learning models because it provides a simple routing mechanism, supports template rendering, and allows seamless integration of Python-based models. By integrating the ANN model into Flask, users can submit sentences through a web page and receive predictions in real-time without needing any programming knowledge. The integration process begins by loading the saved model and TF-IDF vectorizer in the Flask application. The tensorflow.keras.models.load\_model() function is used to reload the trained ANN, while the TF-IDF vectorizer is loaded using pickle or joblib. Once loaded, these objects are ready to process incoming data. The Flask application defines routes corresponding to different web pages—such as home, about, predict, and contact. The /predict route is particularly important, as it handles POST requests containing the user’s input sentence. When a user submits a sentence, the application converts it into a TF-IDF vector using the loaded vectorizer, feeds it into the ANN model, and retrieves the predicted class (Simile or Metaphor).

In addition to handling predictions, the Flask framework also supports template rendering using HTML, CSS, and JavaScript, enabling a rich user interface. For example, the home page can include a welcome message, background image, and a “Get Started” button, while the predict page contains a text input box and a submit button. When the form is submitted, Flask captures the input, passes it to the model, and dynamically updates the page with the prediction results. This integration ensures that users experience a smooth workflow: entering a sentence, clicking submit, and instantly viewing the classification with associated probabilities. The combination of Flask and the trained ANN allows the deployment to be interactive, scalable, and visually engaging. Finally, integrating the model with a web framework allows for future enhancements and maintainability. Additional features such as storing predictions in a database, providing analytics on usage, or even integrating with other web services can be implemented easily. Furthermore, the Flask application can be deployed on cloud platforms or on-premise servers, making the simile and metaphor detection system accessible from anywhere. By combining the saved ANN model, the preprocessing vectorizer, and the Flask web framework, the system achieves a production-ready solution that bridges the gap between machine learning development and real-world application.

**Flask:**

The Flask UI forms the front-end component of a web application that interacts with a Flask-based backend, allowing users to seamlessly engage with the application’s functionality. In the context of simile and metaphor detection, the Flask UI provides an interactive and visually appealing interface where users can input sentences, submit them for classification, and view the results in real-time. The Flask framework uses Jinja2 templating engine, which allows dynamic HTML content to be rendered based on the data passed from the Flask backend. This ensures that the UI can respond to user inputs and model predictions dynamically, making the application highly interactive. A typical Flask UI is organized around HTML templates, often stored in a templates directory. These templates define the structure and layout of web pages, including elements such as headers, navigation bars, forms, buttons, and content sections. For example, the home page might include a navigation menu at the top with links to HOME, ABOUT, PREDICT, and CONTACT pages, while the center of the page can display a background image with a welcome message and a “Get Started” button. The button can be linked to the predict page, guiding users directly to the functionality of the application. By organizing HTML in templates, the UI remains modular and maintainable, allowing developers to easily update design elements without altering backend logic.

The navigation bar is an essential part of the Flask UI. It typically resides at the top of each page and provides quick access to other sections of the website. In our implementation, the navigation bar is divided into two main sections: the left side displays the application title “Figurative Insight,” and the right side contains links to HOME, ABOUT, PREDICT, and CONTACT pages. The navigation bar can be styled using CSS for color, font, spacing, and hover effects, ensuring a consistent look and feel across all pages. This layout improves usability, helping users navigate the application efficiently while maintaining an aesthetically pleasing interface. Forms and user input handling are central to the Flask UI for simile and metaphor detection. The predict page contains a text area where users can enter a sentence. A submit button below the input field sends a POST request to the backend /predict route. Flask captures this input, processes it using the trained ANN model, and then returns the classification result. Using Jinja2 templates, the prediction can be dynamically displayed on the same page without requiring the user to reload it, providing a seamless and responsive experience. Additional features like placeholder text, input validation, and error messages further enhance usability and prevent invalid submissions.

Styling and layout of the Flask UI are handled using CSS, which allows developers to customize the appearance of buttons, forms, text, backgrounds, and navigation elements. For instance, the home page might include a full-page background image, overlaid text with a welcoming message, and a “Get Started” button with hover effects. Colors can be chosen to match the theme of the application—subtle contrasts for readability, accent colors for interactive elements, and typography that enhances visual hierarchy. CSS also ensures responsiveness, allowing the UI to adapt to different screen sizes, from desktops to mobile devices, enhancing accessibility and user experience.

Video and multimedia integration is another feature supported in the Flask UI. On the About page, a video can be embedded on the left side to visually represent the technology, mission, or vision of the project, while text content is displayed on the right. HTML5 <video> tags or embedded players allow the inclusion of videos in various formats, and CSS flexbox or grid layouts ensure that video and text are properly aligned and responsive. Multimedia content enriches the user experience, providing both visual and textual information in a cohesive manner. The contact page of the Flask UI includes multiple interactive elements such as location, email, phone contact information, and a submission form. Users can fill out the form to send messages, which can be processed by the backend for storage or email notifications. Input fields, text areas, and submit buttons are styled consistently with the rest of the application. Using Flask’s backend logic, submitted data can be validated, stored, or forwarded, integrating UI elements with functional backend operations. This interactivity makes the application more practical for user engagement and communication.

Dynamic rendering of results is a critical aspect of the Flask UI in predictive applications. After a user submits a sentence for classification, the backend processes it with the ANN model and returns a prediction (Simile or Metaphor) along with probability scores. Using Jinja2 templates, these results are displayed on the predict page dynamically, often in a highlighted box or styled container for emphasis. This approach ensures that users receive immediate feedback, and it can be further enhanced with visual indicators such as color-coded labels (e.g., green for simile, blue for metaphor) to make interpretation intuitive. Integration with Flask routes links the UI with the backend logic. Each page in the UI corresponds to a specific route in the Flask application. For instance, the home page is linked to /, the about page to /about, the predict page to /predict, and the contact page to /contact. Forms in the UI submit data to these routes using HTTP methods like GET or POST. Flask handles requests, passes data to the model or processing functions, and returns the response to the UI. This integration ensures that the UI is not just visually appealing but also functional, providing end-to-end interactivity. Finally, the Flask UI supports future enhancements and scalability. Additional pages, features, or interactive elements can be easily added without disrupting the existing functionality. For example, users could later be allowed to upload documents for bulk sentence analysis, view historical predictions, or access analytics dashboards. CSS and JavaScript libraries such as Bootstrap or jQuery can be incorporated to improve responsiveness, animations, and user experience. Overall, the Flask UI combines aesthetics, interactivity, and functionality to create a robust front-end that complements the ANN model for simile and metaphor detection, delivering a user-friendly and engaging application.

## Activity 2.1: Building Html Page:

Building HTML pages for a web application involves creating structured templates that define the content, layout, and user interaction elements for each page. In the context of the simile and metaphor detection system, multiple HTML pages are required, including Home, About, Predict, and Contact pages. Each page is designed to serve a specific purpose: the Home page introduces the application, the About page provides information about the project and technology, the Predict page allows users to input sentences for classification, and the Contact page facilitates communication with the developers. These pages are typically stored in a templates folder within the Flask project, and Flask’s Jinja2 templating engine is used to dynamically render content, enabling seamless integration between frontend and backend. By modularizing each page into its own HTML file, the application maintains a clean structure, easy maintenance, and scalability for future enhancements.

The layout and design of these pages are crucial for usability and engagement. The Home page, for instance, includes a top navigation bar with the application title “Figurative Insight” on the left and links to HOME, ABOUT, PREDICT, and CONTACT pages on the right. A visually appealing background image is displayed in the center, overlaid with welcoming text and a “Get Started” button that directs users to the Predict page. The About page is split into two sections, with the left side displaying a video about the project and the right-side containing information about technology, mission, and vision. The Predict page features a central text box for user input and a submit button to trigger model prediction, while the Contact page includes location, email, phone details, and a submission form. Styling is handled using CSS, ensuring consistent colors, fonts, spacing, and responsiveness across all pages, while HTML structure ensures semantic organization, accessibility, and interactivity. Together, these HTML pages provide a polished, user-friendly interface that integrates seamlessly with the Flask backend and ANN model for real-time simile and metaphor detection.

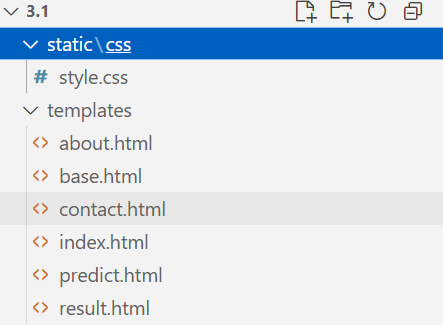


Fig.23 HTML & CSS Pages

## Activity 2.2: Build Python code:

The app.py file serves as the backbone of a Flask web application, connecting the front-end HTML pages to the machine learning model and handling all user interactions. In the context of the simile and metaphor detection system, this Python script is responsible for loading the trained ANN model and the TF-IDF vectorizer, defining routes for each web page, and processing user input for predictions. Flask, being a lightweight and versatile web framework, provides the necessary tools to handle HTTP requests, render templates dynamically, and integrate seamlessly with Python-based machine learning models. The app.py file ensures that the system functions as a cohesive application, enabling users to interact with the predictive model through a simple web interface. The first step in app.py is importing essential libraries such as Flask for the web framework, render\_template for rendering HTML pages, request for capturing user inputs, and redirect or url\_for for page navigation. Additionally, libraries like numpy and tensorflow are imported to handle data processing and model inference. The trained ANN model is loaded using tf.keras.models.load\_model(), while the TF-IDF vectorizer is loaded using pickle or joblib.

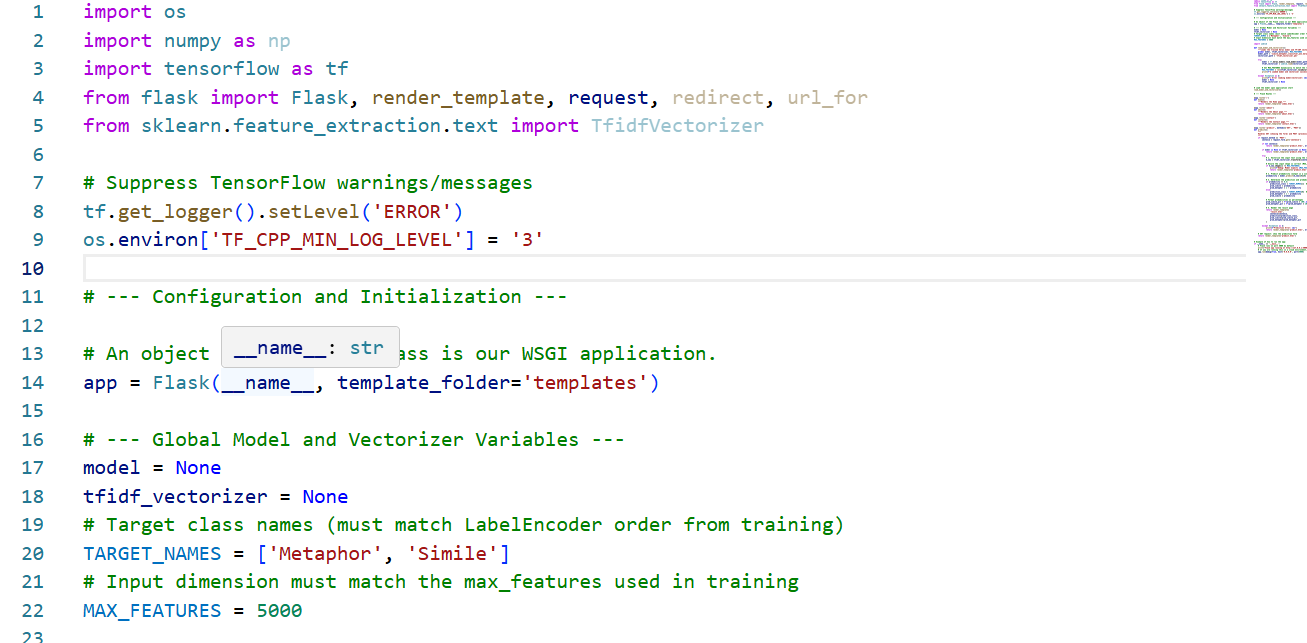


Fig.24 Web Backend File for Integration

Route definition and request handling are central components of app.py. Each web page corresponds to a Flask route, such as / for the Home page, /about for the About page, /predict for the Predict page, and /contact for the Contact page. For the Predict route, the script must handle both GET requests (to display the form) and POST requests (to process user input). When a sentence is submitted via the form, app.py captures the input using request.form.get(), transforms it into a TF-IDF vector, and passes it to the ANN model for prediction. The predicted class (Simile or Metaphor) and the associated probabilities are then dynamically passed back to the template using render\_template(), allowing the results to be displayed directly on the Predict page without requiring a page refresh. Finally, app.py incorporates error handling, scalability, and deployment readiness. Error handling ensures that missing input, model loading issues, or feature mismatches are gracefully managed, providing informative messages to the user.

Render HTML page:

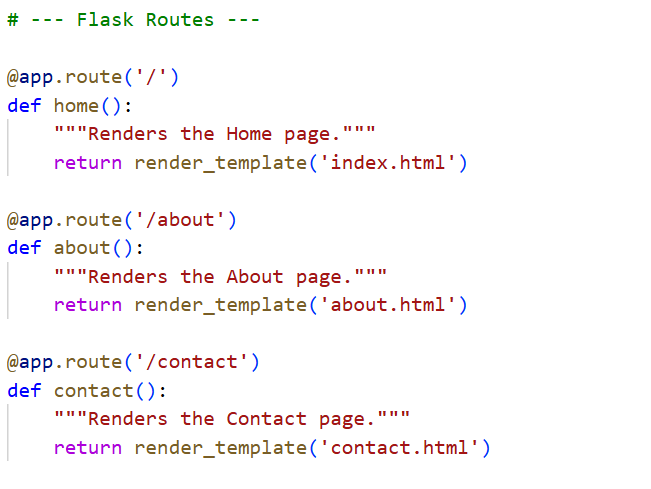


Fig.25 Routing Pages

Here we will be using a declared constructor to route to the HTML page which we have created earlier. In the above example, ‘/’ URL is bound with the index.html function. Hence, when the home page of the web server is opened in the browser, the html page will be rendered. Whenever you enter the values from the html page the values can be retrieved using POST Method.

Retrieves the value from UI:

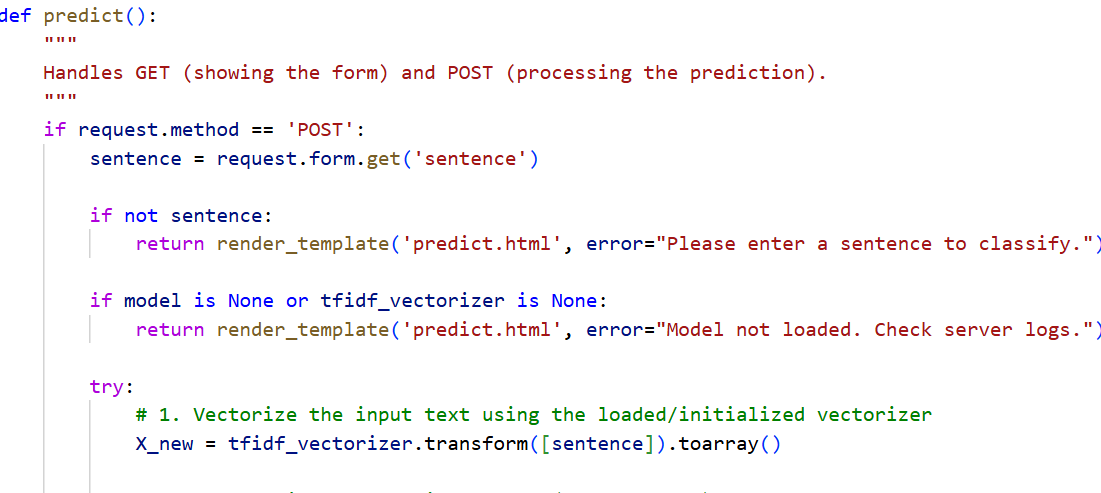


Fig.26 Fetching Values to backend from UI

Here we are routing our app to predict() function. This function retrieves all the values from the HTML page using Post request. That is stored in an array. This array is passed to the model.predict() function. This function returns the prediction. And this prediction value will be rendered to the text that we have mentioned in the submit.html page earlier.

Main Function:

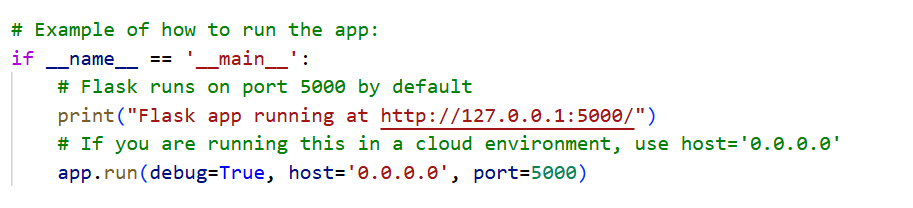


Fig.27 Main Function for call

## Activity 2.3: Run the web application

Once the Flask application (app.py) and all HTML templates are prepared, the next step is to run the web application so that users can interact with it through a browser. Running the application starts a local web server that listens for HTTP requests on a specified host and port. In most development environments, Flask runs by default on http://127.0.0.1:5000/ or http://localhost:5000/. When the application is started, Flask loads the saved ANN model and TF-IDF vectorizer, initializes all routes, and serves the HTML pages to the browser. Running the application allows you to test all features in real time, including sentence input, prediction, and viewing results directly on the web interface. This local deployment is essential for verifying that the integration of the model, backend, and front-end UI is functioning correctly before moving to a production environment. To run the application, navigate to the project directory in the command prompt or terminal and execute python app.py. The script initializes Flask, loads the model, and prints log messages indicating that the server is running. During execution, the terminal displays information about incoming requests, HTTP methods, and any errors encountered, which is helpful for debugging and monitoring. The application can then be accessed using a browser by entering the URL corresponding to the host and port. The Home page appears first, providing navigation to About, Predict, and Contact pages, ensuring that users can seamlessly navigate through the system.

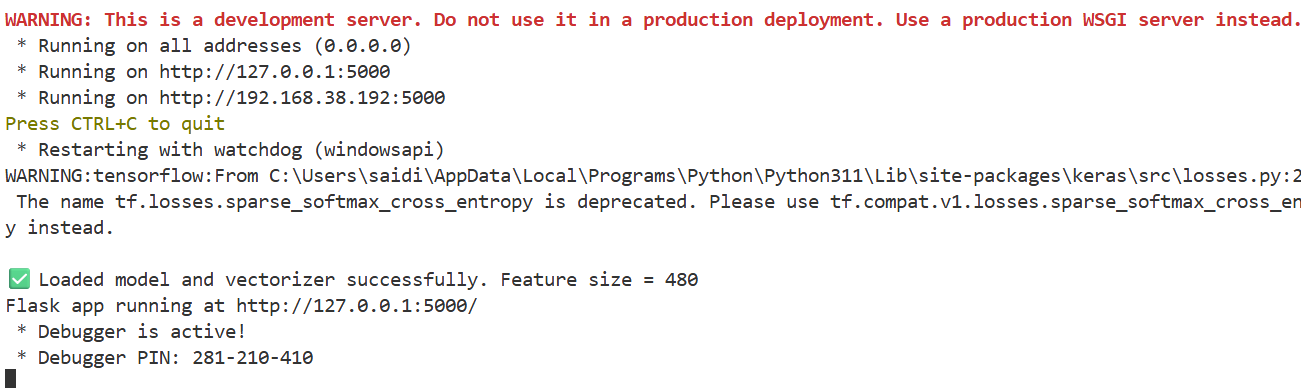


Fig.28 Running app.py file

While the application is running, users can interact with the Predict page to test the simile and metaphor detection functionality. When a sentence is submitted, Flask captures the input, transforms it using the TF-IDF vectorizer, and passes it through the ANN model for prediction. The predicted class and probability scores are then dynamically displayed on the same page. Real-time interaction allows developers to validate the model’s performance with new sentences, observe response times, and ensure that error handling works correctly when invalid inputs are submitted. Additionally, pages like About and Contact provide multimedia content, descriptive text, and interactive forms, giving a complete demonstration of the application’s features. Finally, running the web application locally also sets the stage for future deployment to production environments. Once testing is complete and the application functions as expected, it can be deployed to cloud platforms such as AWS, Google Cloud, or Heroku.

The User Interface (UI) is a critical component of any web application, as it serves as the point of interaction between the user and the underlying system. In the context of the simile and metaphor detection application, the UI is designed to be intuitive, responsive, and visually appealing, ensuring that users can easily navigate through pages, submit sentences for analysis, and understand the results. A well-designed UI enhances user experience, builds trust in the system, and makes the application accessible to a wide range of users, from students and researchers to writers and educators. The Home page acts as the entry point to the application and sets the tone for the overall user experience. It features a top navigation bar with the application title “Figurative Insight” on the left and navigation links to HOME, ABOUT, PREDICT, and CONTACT on the right. The center of the page displays a background image, overlaid with a welcoming message and a “Get Started” button that directs users to the Predict page. The layout uses CSS for styling, including typography, colors, spacing, and hover effects for interactive elements. This ensures the home page is both visually engaging and functional, encouraging users to explore the system.

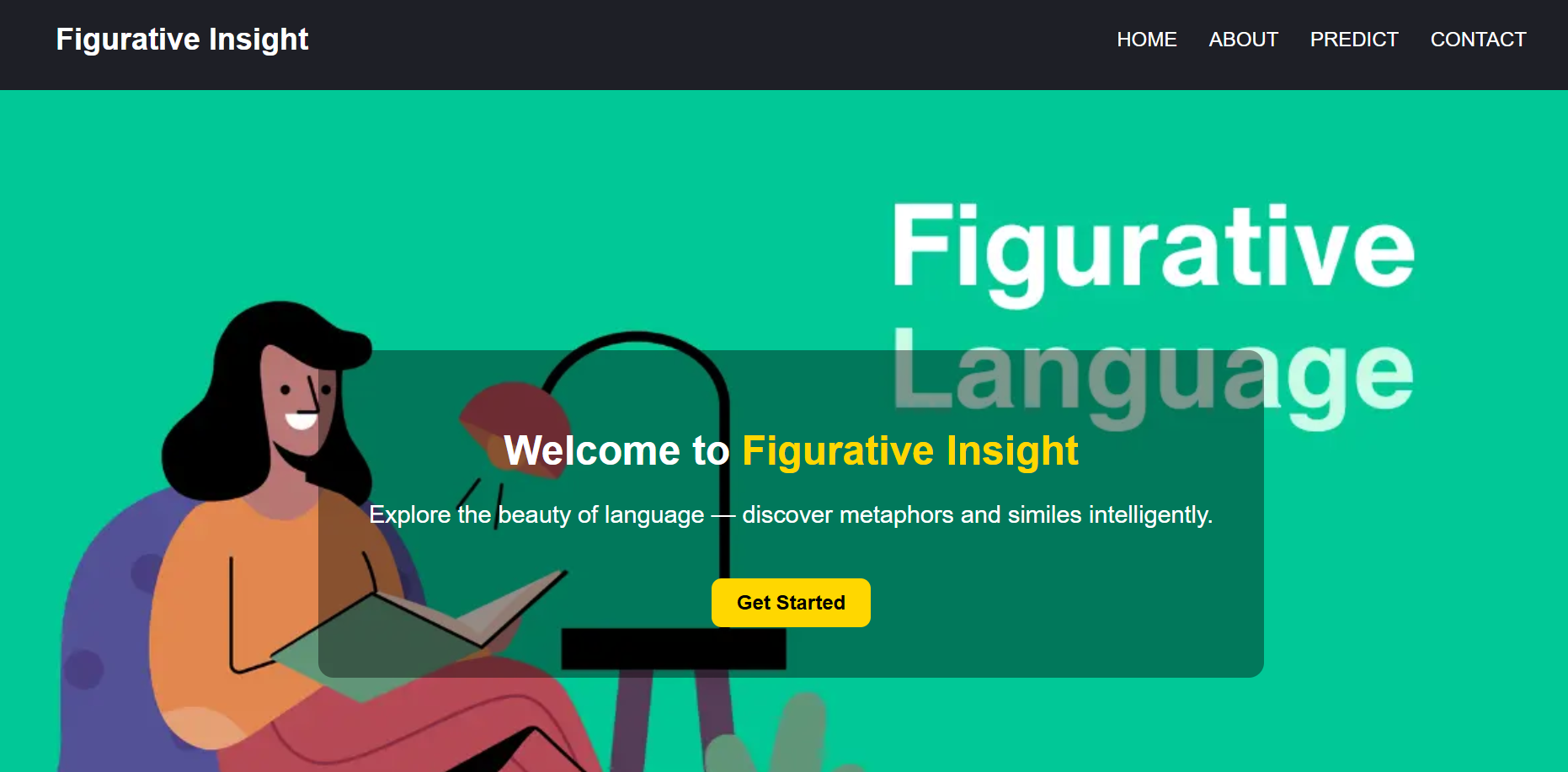


Fig.29 Home page

The About page provides detailed information about the project’s technology, mission, and vision. The page is split into two sections using CSS flexbox or grid layouts: the left section contains a video demonstrating the system or related concepts, while the right section presents text content describing the system’s features, objectives, and innovation. The split layout allows users to consume both visual and textual information simultaneously, making the explanation more engaging and easier to understand. Responsive design ensures that the video and text adapt gracefully to different screen sizes, maintaining usability on desktops, tablets, and mobile devices.

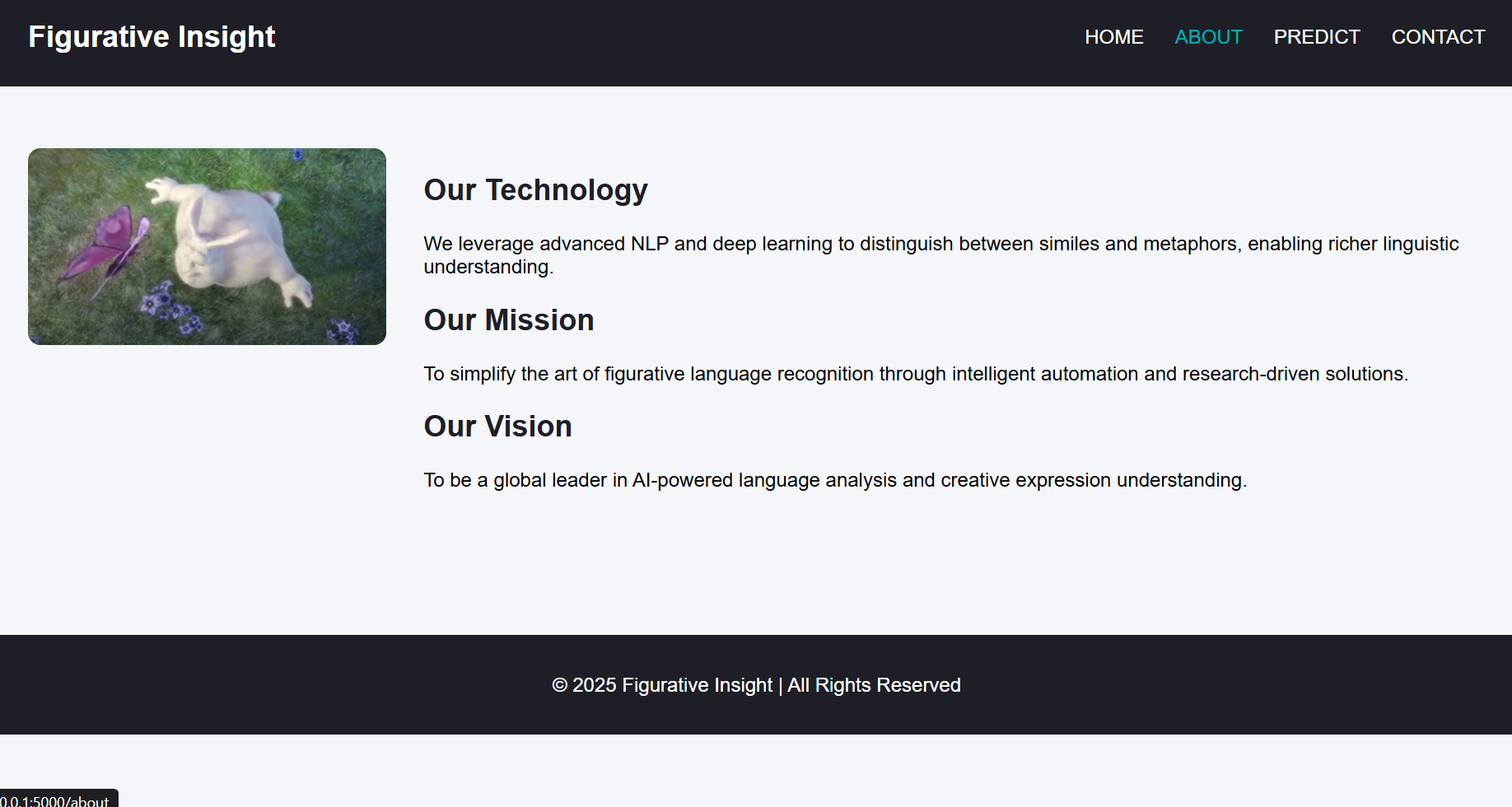


Fig. 30 About Page

The Predict page is the most interactive part of the UI, where users input sentences for simile and metaphor classification. A central text box allows users to type a sentence, and a submit button triggers the backend prediction function. The UI is designed to display the prediction results dynamically, showing the predicted class (Simile or Metaphor) along with probability scores. Color-coded labels or styled containers can be used to differentiate between the two classes visually. Real-time feedback and intuitive placement of input and output elements enhance the user experience, making the prediction process straightforward and engaging.

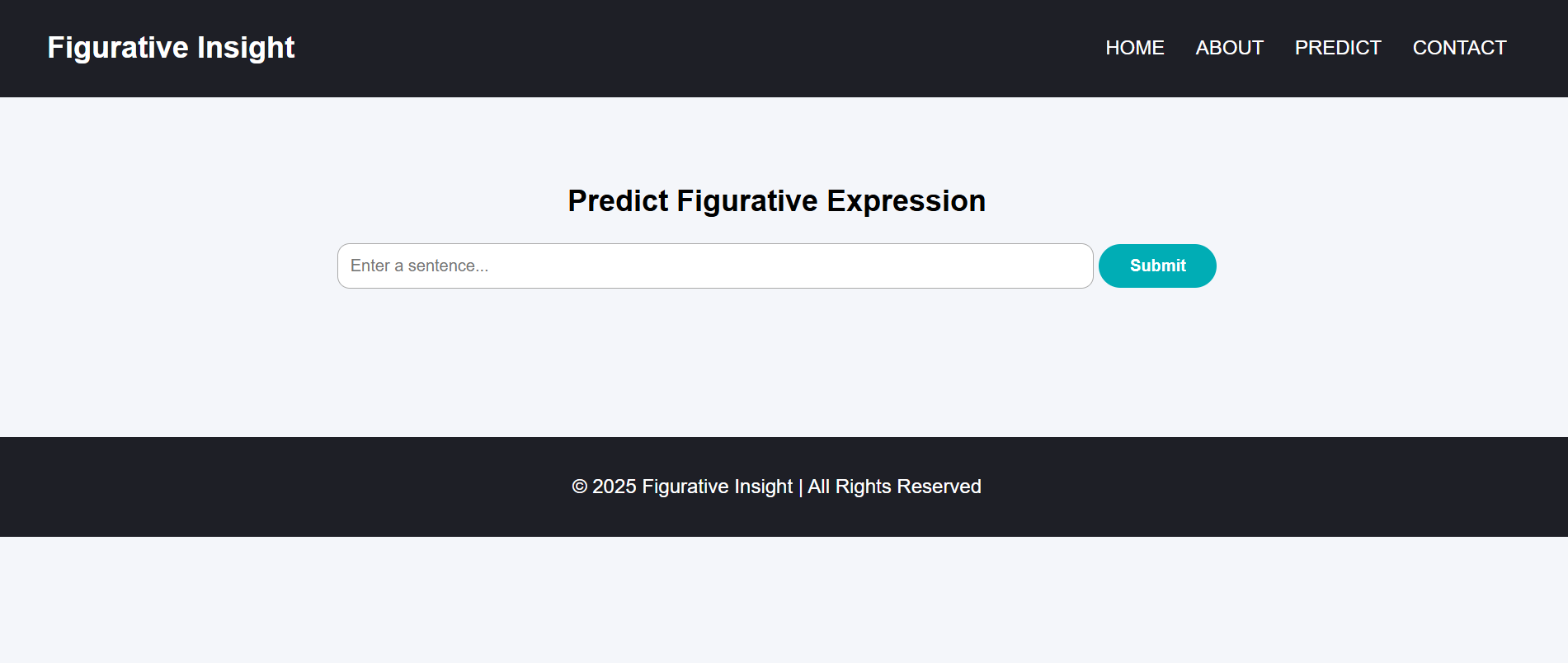


Fig.31 Prediction Page

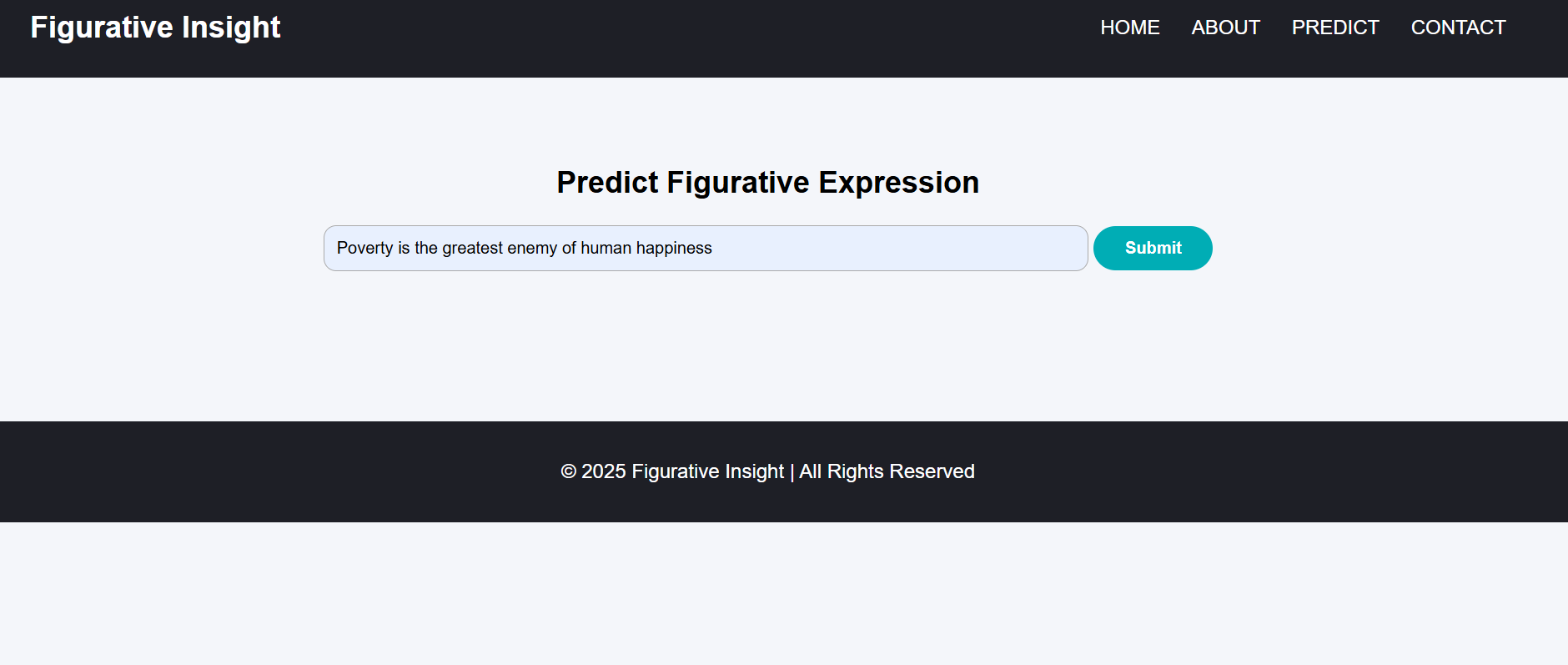


Fig.32 Prediction Page

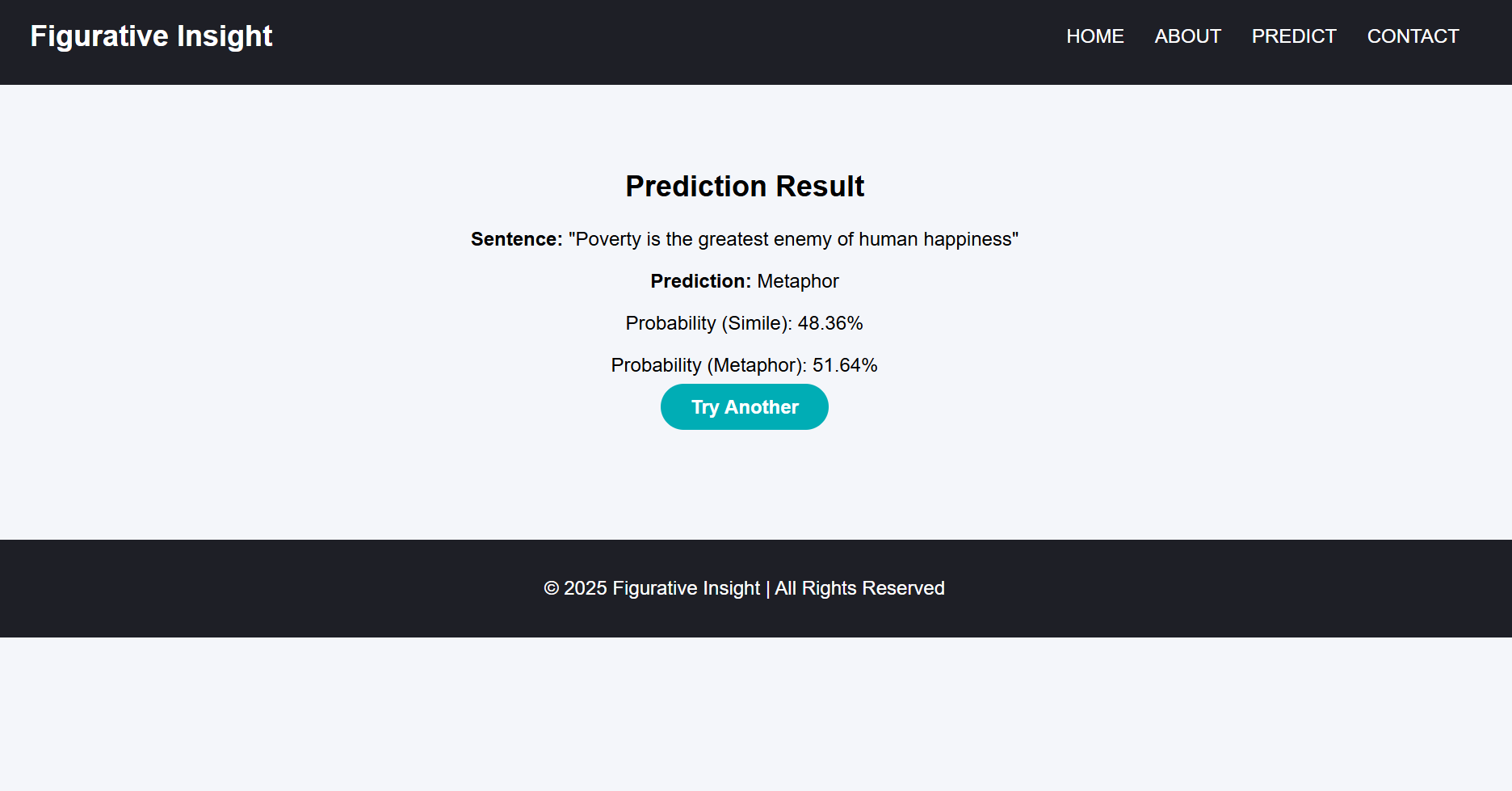


Fig.33 Results Page

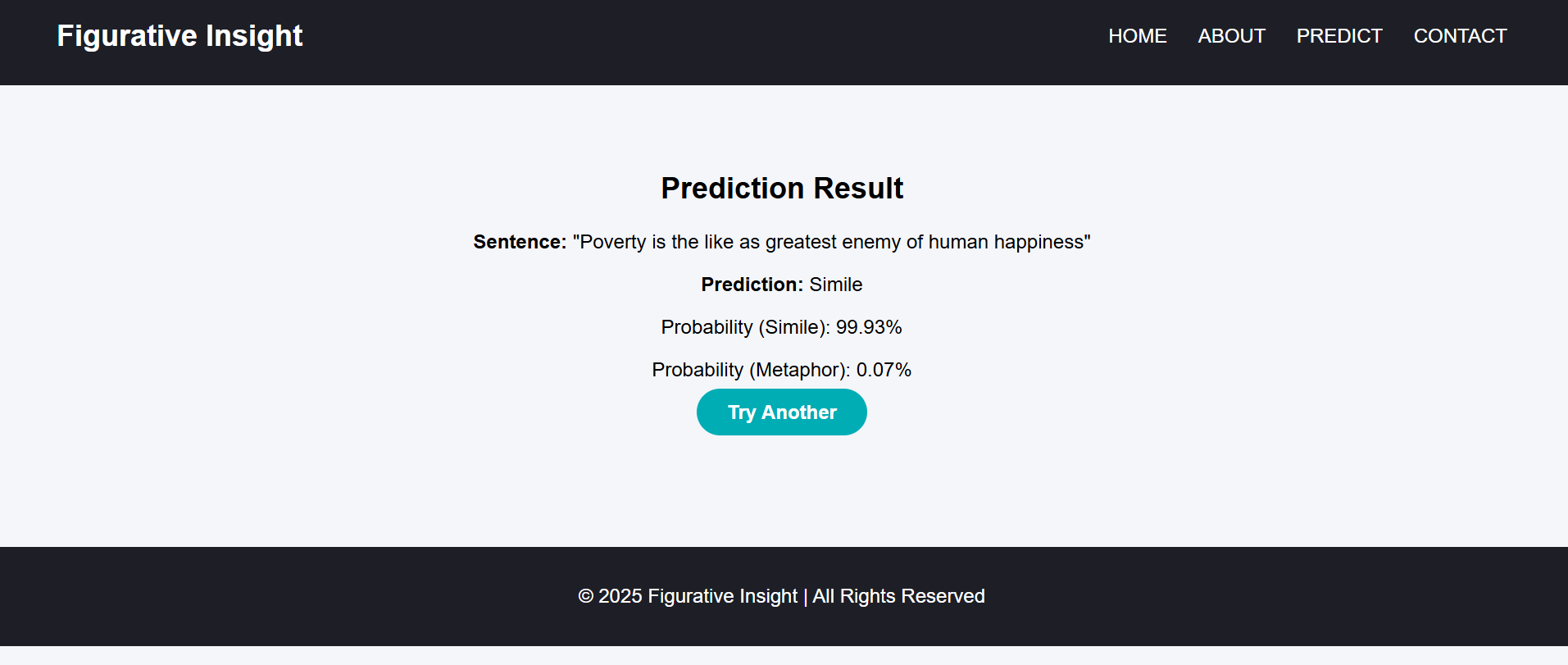


Fig.34 Results Page

The Contact page facilitates communication between users and developers. It includes information such as location, email, and phone numbers, along with a submission form for sending messages or queries. Input fields and the submit button are styled consistently with the rest of the application. Backend integration allows the form data to be captured and processed, either stored in a database or sent via email.

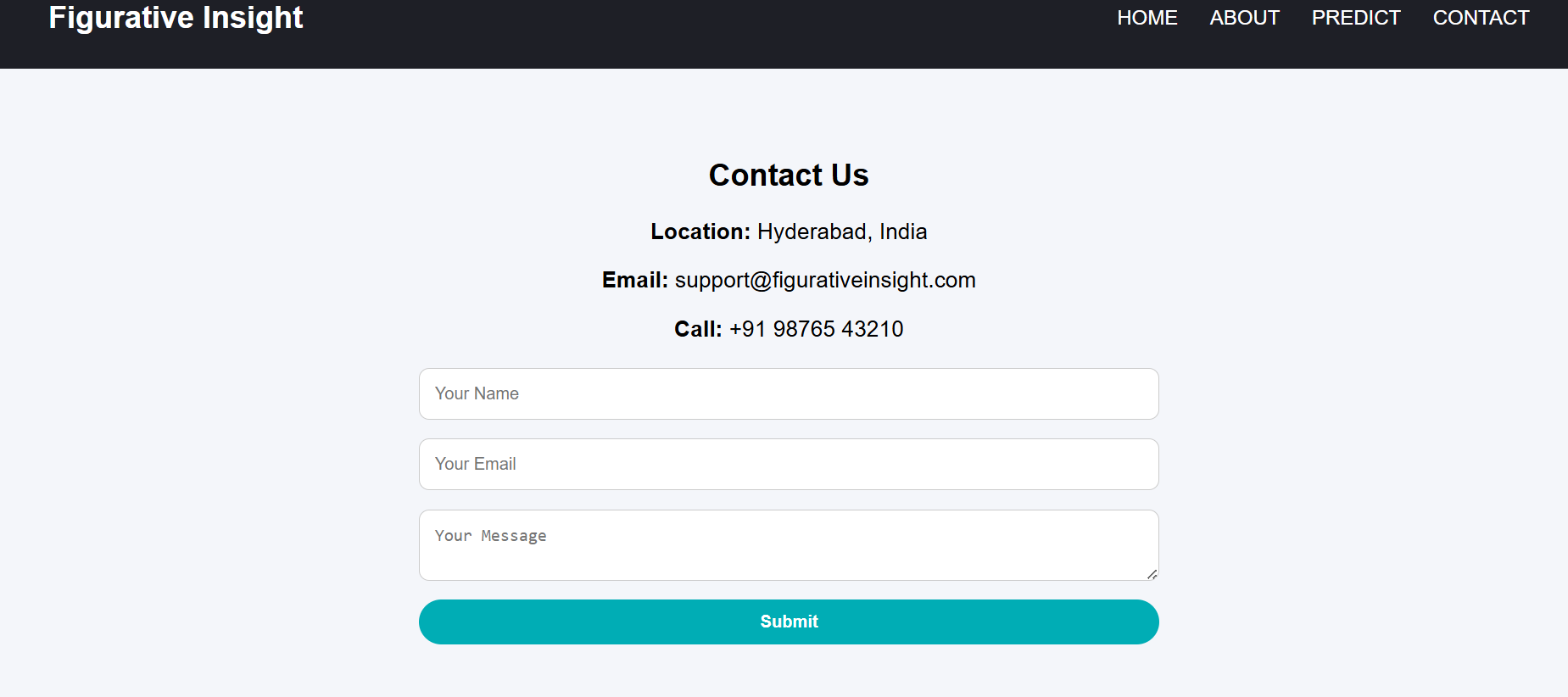


Fig.35 Contact Us Page

Styling and visual design elements play a significant role in the UI. CSS is used to define colors, fonts, spacing, borders, hover effects, and layouts. For instance, the home page background image may use an overlay to make text more readable, while buttons may include hover effects to indicate interactivity. Typography is selected to enhance readability, and spacing between elements ensures a clean and organized look. Responsive design principles allow the application to scale across different devices, ensuring that users have a consistent and pleasant experience regardless of their screen size. Navigation and interactivity are central to the UI design. The top navigation bar provides quick access to all major sections of the application, while buttons and links guide users through a logical flow, from learning about the system to making predictions and contacting the developers. Interactive elements like buttons, forms, and dynamically updated results improve engagement and usability. JavaScript or minimal jQuery can be used to add animations, validate form inputs, or enhance user interactions without overwhelming the simplicity of the UI.

Dynamic content rendering is handled through Flask’s Jinja2 templating engine, which allows data from the backend to be displayed seamlessly on the front-end pages. For example, the Predict page uses Jinja2 to dynamically show the prediction results after the user submits a sentence. This eliminates the need for page reloads, providing a smooth and responsive user experience. Other pages, such as About and Contact, can also use dynamic content to display updated information, videos, or notifications, ensuring that the UI remains interactive and up-to-date. Accessibility and usability are important considerations in the UI. The application uses semantic HTML elements, proper labeling of forms, and sufficient contrast in colors to ensure readability. Input validation, placeholder texts, and informative error messages guide users in submitting correct data, reducing confusion and errors. These design choices make the system more inclusive and user-friendly, allowing users of different technical expertise to interact with the application effectively. Finally, the UI supports future enhancements and scalability. Additional features, such as uploading multiple sentences, viewing prediction history, or accessing analytics dashboards, can be incorporated without major redesign. CSS frameworks like Bootstrap can be integrated for improved responsiveness, and additional JavaScript features can enhance interactivity.

**Milestone 7: Project Demonstration & Documentation**

Below mentioned deliverables to be submitted along with other deliverables

## Activity 1:- Record explanation Video for project end to end solution

**Activity 2:- Project Documentation-Step by step project development procedure**