CLASSIFICATION OF TELUGU IDIOMATIC SENTENCES USING ENSEMBLE MODELS

A MAJOR PROJECT REPORT

Submitted by

CH CHAITANYA [RA1911003010992]
CH V M SAI PRANEETH [RA1911003010983]

Under the guidance of

Ms. J. BRISKILAL

Assistant professor, Department of computing technologies

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

DEPARTMENT OF COMPUTING TECHNOLOGIES

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR– 603 203

MAY 2023

SRM

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR–603 203

BONAFIDE CERTIFICATE

Certified that 18CSP109L major project report titled **"CLASSIFICATION OF TELUGU IDIOMATIC SENTENCES USING ENSEMBLE MODELS"** is the bonafide work of **"CH CHAITANYA [RA1911003010992], CH V M SAI PRANEETH [RA1911003010983]"** who carried out the major project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Ms.J. BRISKILAL
SUPERVISOR
Assistant professor
Department of Computing Technologies

Dr.B. ARTHI
PANEL HEAD
Associate Professor
Department of Computing Technologies

Department of Computing Technology Kattankulathur 603 203 SRM IST

Dr.M. PUSHPALATHA
HEAD OF THE DEPARTMENT
Professor
Department of Computing Technologies

INTERNAL EXAMINER

EXTERNAL EXAMINER

Department of Computing Technologies

SRM Institute of Science and Technology
Own Work Declaration Form

Degree/Course :B.Tech in Computer Science and Engineering

Student Names : CH CHAITANYA, CH SAI PRANEETH

Registration Number : RA1911003010992, RA1911003010983

Title of Work : CLASSIFICATION OF TELUGU IDIOMATIC SENTENCES USING ENSEMBLE MODELS

I/We hereby certify that this assessment compiles with the University's Rules and Regulations relating to Academic misconduct and plagiarism, as listed in the University Website, Regulations, and the Education Committee guidelines.

I / We confirm that all the work contained in this assessment is our own except where indicated, and that we have met the following conditions:

- Clearly references / listed all sources as appropriate.
- Referenced and put in inverted commas all quoted text (from books, web,etc.)
- Given the sources of all pictures, data etc that are not my own.
- Not made any use of the report(s) or essay(s) of any other student(s)either past or present
- Acknowledged in appropriate places any help that I have received from others (e.g fellow students, technicians, statisticians, external sources)
- Compiled with any other plagiarism criteria specified in the Course handbook / University website.

I understand that any false claim for this work will be penalized in accordance with the University policies and regulations.

DECLARATION:
I am aware of and understand the University's policy on Academic misconduct and plagiarism and I certify that this assessment is my / our own work, except were indicated by referring, and that I have followed the good academic practices noted above. **Student 1 Signature:** **Student 2 Signature:** **Date:**
If you are working in a group, please write your registration numbers and sign with the date for every student in your group.

ACKNOWLEDGEMENT

We express our humble gratitude to **Dr. C. Muthamizhchelvan**, Vice-Chancellor, SRM Institute of Science and Technology, for the facilities extended for the project work and his continued support.

We extend our sincere thanks to Dean-CET, SRM Institute of Science and Technology, **Dr.T.V.Gopal**, for his invaluable support.

We wish to thank **Dr. Revathi Venkataraman**, Professor & Chairperson, School of Computing, SRM Institute of Science and Technology, for her support throughout the project work.

We are incredibly grateful to our Head of the Department, **Dr. M. Pushpalatha** Professor, Department of Computing Technologies, SRM Institute of Science and Technology, for her suggestions and encouragement at all the stages of the project work.

We want to convey our thanks to **Dr. B.ARTHI**, Associate Professor and Panel members, **Dr.M.karthikeyan** Assistant Professor, **Dr.T.Ragunthar** , Assistant Professor and **Ms.J.Briskilal**, Assistant Professor, Department of Computing Technologies, SRM Institute of Science and Technology, for their inputs during the project reviews and support.

We register our immeasurable thanks to our Faculty Advisor, **Dr.M.Kowsigan**, Associate Professor, Department of Computing Technologies, SRM Institute of Science and Technology, for leading and helping us to complete our course.

Our inexpressible respect and thanks to our guide, **Ms.J.Briskilal**, Assistant Professor, Department of Computing Technologies, SRM Institute of Science and Technology, for providing us with an opportunity to pursue our project under her mentorship. She provided us

with the freedom and support to explore the research topics of our interest. Her passion for solving problems and making a difference in the world has always been inspiring.

We sincerely thank the Computing Technologies Department staff and students, SRM Institute of Science and Technology, for their help during our project. Finally, we would like to thank parents, family members, and friends for their unconditional love, constant support, and encouragement.

CH CHAITANYA [RA1911003010992]

CH V M SAI PRANEETH [RA1911003010983]

ABSTRACT

With the vast amount of text data being populated on the World Wide Web (WWW), Text categorization becomes one of the most inevitable pre-processing approaches in any text processing application. There are numerous different applications for text categorization, such as sentiment analysis, the identification of spam, the extraction of information, and intent detection. The detection of intent is one of the most difficult aspects of text classification since it involves a wide variety of elements, including the employment of polysemic words, metaphors, idioms, and phrases. In this paper, an attempt is made to categorize literals and idioms. A frequent phrase that has come to mean something other than its exact meaning is referred to as an idiom. When it comes to the automatic detection of idioms, several natural language processing (NLP) applications, such as information retrieval (IR), machine translation, and chatbots, present a significant challenge. Every one of these applications comes with its very own special set of difficulties to deal with. In each of these applications, the automatic recognition of idioms plays a crucial role. Text classification is one of the most fundamental tasks in natural language processing (NLP). This activity organizes text into organized groups, often known as text labelling or categorization. The task of identifying idioms as a form of text classification is the focus of this article. pre-trained deep learning models, machine learning models and their ensemble models are utilized for a variety of text classification tasks. The model is validated using a freshly compiled in-house dataset composed of idioms and literal expressions, total 1040 entries and having been annotated by specialists in the relevant field.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ML	Machine Learning
DL	Deep Learning
IR	Information retrieval
SVM	Support vector machine
m-BERT	Multilingual Bidirectional Encoder Representations from Transformers
DL	Deep Learning
BERT	Bidirectional Encoder Representations from Transformers
RoBERTa	Robustly optimized BERT approach
MLM	Masked Language Modelling
NLP	Natural language processing

CHAPTER 1

INTRODUCTION

Text classification can refer to either the process of labelling or organizing textual data into groups, as well as the term classification itself. The process of text classification involves organizing or labelling textual data. One additional way to think about this is as the process of grouping together the various kinds of textual data. The processing of natural languages must invariably incorporate this aspect as an essential component (NLP). Because we are currently living in the digital era, this means that we are constantly exposed to text in a variety of forms. This can be both beneficial and detrimental to our learning. The text that can be found on our social media pages, advertisements, blogs, electronic books, and other forms of digital media are a few examples of this type of content. Classification is a tool that can be of great assistance in determining the content of this text data because the vast majority of it is unstructured. The classification of texts can be put to use in a wide variety of situations.

Some of the uses of this technology include the detection of spam, the performance of sentiment analysis, the labelling of themes, the identification of languages, the categorization of online information, and the identification of the author's purpose. Labeling topics, identifying languages, labelling topics, and classifying online content are some examples of additional applications. In this particular piece of research, the process of text classification is accomplished with the assistance of literals and idioms. To organize and categorize words and phrases, idiomatic and literal classifications of words and phrases are frequently used in natural language processing applications, such as translation algorithms and information retrieval (IR) systems.

The inherent complexities that are present in language interpretations continue to be the focus of a significant amount of research, and natural language is still the subject of this research. This situation becomes even more precarious when computers are used to perform language interpretations automatically. This calls for performing an automatic disambiguation of the text in order to derive the meaning that was intended. In order to make one such attempt, the primary focus of this paper will be on classifying idioms in accordance with the literal expressions that they are supposed to be corresponding to. It has been determined that the task at hand is one of text classification, and XLM-RoBERTa has been utilized in order to carry out the task in order to carry out the implementation.

A phrase is considered to be literal if it contains words that are the same as those found in an idiomatic phrase; however, the literal phrase communicates a meaning that is derived from the words themselves rather than an idiomatic meaning. Words that are used in an idiomatic phrase may also be included in a literal phrase; however, the literal meaning of the words is conveyed rather than the figurative meaning of the words. A phrase is considered to be idiomatic if it contains words that do not have their literal meaning preserved.

Example idiomatic sentences are given below:

1. నేను ఏ పని చేసినా పురిటిలోనే సంధి కొడుతుంది.
   Nenu e pani cesina puriṭilone sandhi koḍutundi
   Whenever I start to work on something it leads to failure

2. రవి గాడు ప్రస్తుతం వదిలేసి పగటి కలలు కంటున్నాడు.
   Ravi gaḍu prastutam vadilesi pagaṭi kalalu kaṇṭunnaḍu

Leaving the present, Ravi is dreaming about future

3. రాము ఆలస్యంగా రావడంతో పంతులు చేతిలో అక్షింతలు పడ్డాయి.

   Ramu alasyanga ravadanto pantulu cetilo aksintalu paddayi

   Ramu was scolded by his teacher because of his late presence

4. వాడు పక్కదారి పట్టించడంలో దిట్ట.

   Vadu pakkadari pattinncadanlo ditta

   He is very talented in making fool of others

5. రమేష్ డ్యామ్ కట్టించడానికి గజ్జ కట్టాడు.

   Rames dyam kattincadaniki gajja kattadu
   Ramu started constructing the dam

Challenges we have encountered due to the lack of automatic idiom recognition.

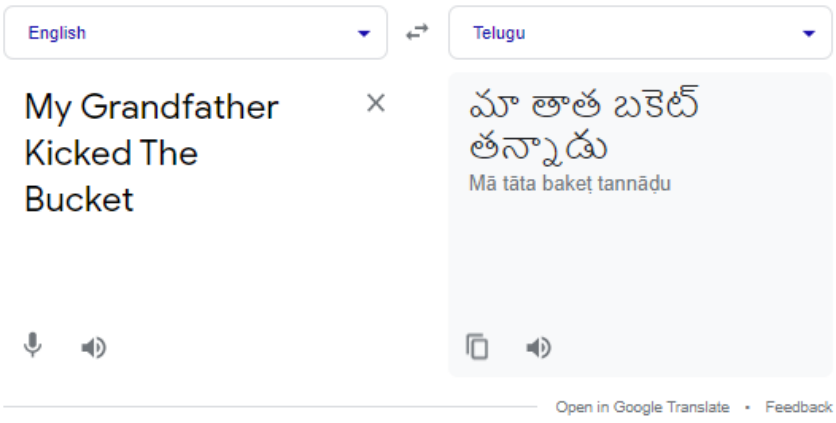


Figure 1.1 depicts translation of English idiomatic sentence.

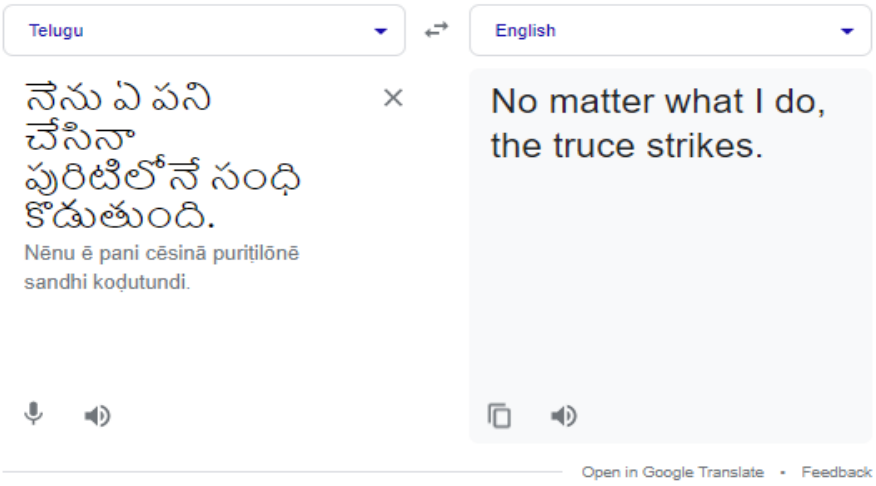


Figure 1.2 depicts translation of telugu idiomatic sentence.

We've illustrated how a machine translation system can mistranslate in figure 1.1 "My Grandpa Died" must appear as the proper translation provided by the machine translator. Because the translator is unable to discern whether the sentence is idiomatic or not, the translation is done only on the basis of the words in the sentence. Like example 1- The translator has given the incorrect translation in figure 1.2 - Which should be "Whenever I start working on something, it leads to failure." Due to a lack of automatic idiom identification, this is one of the areas where these applications do not perform as intended. These are the challenges for the creation of automatic idiom recognition.

1.1 Software Requirement Specifications

Operating System: Windows 7 and above or Linux
Platform: Google Colaboratory
Special Tools: Torch, NumPy, Matplotlib, Sklearn

CHAPTER 2

LITERATURE SURVEY

The literature review was carried out concurrently along two separate dimensions at the same time. The first study approaches the issue from the standpoint of idiom recognition, whereas the second study approaches the issue from the standpoint of utilizing machine learning techniques to classify texts according to whether or not they contain idioms. Because of these two factors, a more in-depth comprehension of the characteristics and categories of machine learning algorithms that are utilized in text classification and idiom recognition has been attained. These algorithms are used to recognize phrases and text. Idiom recognition is made possible by these algorithms, which can be found in most modern computers.

2.1 Text classification works using Machine learning models

"A Comparative Study of Machine Learning Techniques for Text Classification" by Yang et al. (2009) - On several benchmark datasets, this study compares a number of machine learning models for text categorization, including Naive Bayes, SVM, k-NN, and Decision Trees. The accuracy, precision, recall, and F1-score of these models are compared by the authors along with their overall performance. "Text Classification using Machine Learning Techniques: A Review" by Priya and Menon (2019) - This paper offers a thorough analysis of machine learning-based text classification methods. The authors discuss the various models used for text classification, such as Naive Bayes, SVM, Decision Trees, Random Forests, and Gradient Boosting, along with their advantages and limitations.

"Comparison of Machine Learning Techniques for Text Classification" by Alshahrani et al. (2020) - This study examines the effectiveness of various text

categorization machine learning models, such as Naive Bayes, SVM, Decision Trees, Random Forests, and Gradient Boosting, on a number of benchmark datasets. Based on the models' accuracy, precision, recall, F1-score, and AUC, the authors grade them. "A Comparative Study of Supervised Machine Learning Techniques for Text Classification" by Kaur and Singh (2020) - This paper provides a comparative study of several supervised machine learning techniques for text classification, including Naive Bayes, SVM, Decision Trees, Random Forests, and Gradient Boosting. The accuracy, precision, recall, F1-score, and ROC-AUC are used by the authors to compare how well these models perform on various text categorization datasets.

"A Comparative Study of Deep Learning Techniques for Text Classification" by Agarwal et al. (2021) - This article compares the performance of CNNs, RNNs, and Transformers among other deep learning models for text classification. The authors compare these models' performance on many benchmark datasets in terms of accuracy, precision, recall, F1-score, and ROC-AUC. These papers highlight the value of evaluating models using a variety of performance indicators and show how different machine learning models may effectively classify text. They also emphasize how important it is to select the right models based on the data's properties and the task at hand.

2.2 Text classification works using RoBERTa model

The RoBERTa and SVM classifiers were applied in the context of the aggression detection-based classification, according to Arup et al. (2020). Different translations of the dataset into English, Hindi, and Bangla are available for the three languages. This dataset can be used to identify aggressive behaviors. These interpretations can be found below. The datasets in question are amenable to mutual comparison. When compared to the SVM classifier, the RoBERTa model, which was employed for the English subtasks, performed

better with an F-score of 80%. This result was achieved through the application of the RoBERTa model. (Baruah, Das, Barbhuiya & Dey, 2020). During the course of their investigation into the aspect category sentiment analysis, Wexiong et al. (2020) made extensive use of the RoBERTa model as the primary instrument that served as their guide. The classification was completed with the assistance of a dataset from AI Challenger that was accessible to the general public and included fine-grained sentiment analysis (Liao, Zeng, Yin & Wei, 2021). For the purpose of classification, the publicly available dataset (fine-grained sentiment analysis) that was supplied by AI challenger was utilised. In order to acquire the outcomes that were aimed for, this action was taken (Liao et al., 2021). Ankit et al. (2020) was successful in locating and classifying posts on social media that were associated with mental illness because they made use of RoBERTa. The dataset that was used to determine the five distinct class labels was built with a total of three thousand unique posts that were culled from a variety of subreddits. These posts were used to build the dataset. The dataset that was compiled utilized these posts in various ways. When compared to the BERT and LSTM [15] models, RoBERTa demonstrated performance that was clearly more effective than either of those models.

2.3 Text classification works using BERT model

Gonzalez et al. (2020) used the BERT model to identify text in four different datasets: IMDB, RealorNot, Portuguese news, and Chinese hotel reviews. 5000 movie reviews from the IMDB dataset were categorized using analysis. In order to categorize messages according to whether or not they were tweets, the RealorNot tweet dataset was used for binary classification. In order to categorize articles and reviews, respectively, data from Chinese hotel reviews and Portuguese news sources were both examined. Gonzalez-Carvajal and Garrido-Merchan (2020) claim that the BERT approach and conventional machine learning approaches were contrasted in this work.

To enhance the performance of standard LSTM versions, Ashutosh et al. (2019) suggested combining the BERT model and a knowledge distillation method for document classification. Four different datasets—Reuters-21578, the AAPD academic publishing dataset, IMDB, and Yelp '14—were used to test their approach. According to Adhikari, Ram, Tang, and Lin (2019), two instances of multi-label datasets are Reuters-21578 and AAPD.Jihang et al. (2019) studied the categorization of events in Spanish text based on their factuality using a Uruguayan newspaper. For the classification of multilingual texts, the BERT multilingual-cased model and the FACT dataset were also used (Mao & Liu, 2019).

In order to provide a universal solution for fine-tuning a BERT model to perform multiple forms of text categorization, such as those of sentiments, questions, and themes, Sun et al. (2019) presented a range of fine-tuning methodologies. IMDB movie review and Yelp review datasets were used for sentiment classification, the large-scale AG News dataset for topic classification, the open-domain, fact-based TREC dataset for question classification, and the TREC dataset for question classification (Sun, Qiu, Xu, & Huang, 2019). When Issa (2020) used BERT phrase embedding to identify humour in brief texts, the suggested method performed better than more conventional models like the RNN. The ColBERT dataset contains 200k texts, some of which are humorous and some which are not (Annamoradnejad & Zoghi, 2020). According to Munikar, Shakya, and Shrestha (2019), sentiment categorization was carried out using the BERT model on the Stanford Sentiment Treebank (SST) dataset by Manish et al. (2019).

2.4 Works on classification of idioms

Peng [3] presented a method with the intention of recognizing and classifying idiomatic expressions and literal expressions into the categories that are most appropriate for them based on the meaning of the expressions. Sentences were categorized as either literal expressions or idioms using a bag-of-words topic representation based on the subjects that were taken into account, and this was done in order to find the category that they most effectively fit into. This was done in order to classify sentences as either literal expressions or idioms, and this was done in order to determine which category they most effectively fit into. It was necessary to do this in order to categorize sentences as either literal expressions or idioms, and it was also necessary to do this in order to determine which category sentences most naturally fit into. This was done so that sentences could be classified as literal expressions, idioms, or some other type of expression.

This method of classification made use of four separate datasets, including BlowWhistle (which contained a total of seventy-eight sentences, including 51 literals and 27 idioms), MakeScene (which contained a total of fifty sentences, including 20 literals and 30 idioms), LoseHead (which contained a total of forty sentences, including 19 literals and 21 idioms), and TakeHeart (which contained a total of eighty-one sentences, including 61 idioms. Using idiom-based features as the data source, Irena and her colleagues (2017) published an automated method to enhance sentiment analysis. This method was developed by Irena. A dataset that is widely considered as the standard for the industry was chosen to be used for the classification of the data. Additionally, the analysis included a set of guidelines for identifying idioms and sentiment polarity, which were both considered (Spasic, Williams, and Buerki, 2017). When classifying idioms and literals that appear within paragraphs, Naziya (2020) utilized both a rule-based generalization as well as a context-based classification. This was done in order

to categorize literals and idioms that are found within paragraphs. (Shaikh, 2020) In the research that Liu and his colleagues carried out and published in 2018, they suggested that an approach that is based on neural networks be used to make recommendations regarding the utilization of idioms in the writing of essays. Specifically, they suggested that idiomatic expressions be used more frequently in academic writing. Idioms were specifically mentioned as something that should be utilized more frequently in academic writing. Prior to the results being revealed, the degree of similarity between the context that had been provided and the potential idiom recommendations had been calculated and analysed. On the TOEFL corpus and the VUA (VU Amsterdam Metaphor corpus), Xianyang et al. (2020) performed a token-level metaphor classification approach using a BERT model. Xianyang et al. were in charge of this process. The approach described above was completed for each of these two corpora (Chen, Leong, Flor, & Klebanov, 2020).

By integrating late and early fusion strategies, Liu et al. (2017) introduced a supervised ensemble model that identifies idiomatic expressions in addition to literal expressions. By applying this model, the expressions were sorted into the appropriate categories (Liu & Hwa, 2017). The research that Charles and his colleagues did suggested that crowdsourcing could be used as a method for collecting the emotional annotations of idiomatic expressions. This research was conducted by Charles (2018). The Sentiment Lexicon of Idiomatic Expressions (SLIDE), which was created as a result of this method, is far more comprehensive than lexicons that came before it (Jochim et al., 2018). An A Mona et al. came up with the idea of using supervised learning as an approach to the classification of multiword expressions as opposed to literals (2009). This investigation made use of the VNC-Tokens dataset, which is a construction involving a verb and a noun. The authors were successful in accomplishing this goal thanks to the utilization of the dataset.

2.5 Use of ensemble models for text classification

A bagging-based ensemble model (Risch & Krestel, 2020) was provided by Julian et al. (2020) to categorise 6000 hostile and inflammatory social media messages using many updated BERT models. For the classification of idioms and literal utterances, Tadej et al. (2020) recommend the ensemble BERT and ELMo model. This tactic should be aided by contextual embedding. Skvorc, Gantar, and Robnik-Sikonja (2020) completed the monolingual classification using a Slovenian dataset. The text categorization system developed by Harita Reddy, Namratha Raj, Manali Gala, and Annappa Basava (2020) used ensemble models to detect fake news.

Arabic News Article Automated Text Tagging Ashraf Elnagar, Omar Einea, and Ridhwan Al-Debsi in their 2019 work recommend using ensemble deep learning models. The categorization of idioms and literal text in 2022 was accomplished using an ensemble model developed by S. Abarna, J. I. Sheeba, and SP. Devaneyan using knowledge-enabled BERT in deep learning. For the purpose of classifying Tamil idioms and literal texts, J. Briskilal and CN. Subalalitha developed an ensemble model using BERT and RoBERTa.

Idioms and literal texts in Telugu were not categorised utilising ensemble techniques, according to the available studies. To improve the accuracy of results in models, ensemble techniques mix multiple models rather than relying solely on one. The accuracy of the results is significantly increased by the combined models. Applications for ensemble methods in regression and classification are ideal since they improve model accuracy by reducing bias and variance. In this study, we try to categorise literals and idioms using an ensemble model using both deep learning and machine learning models that have been previously trained.

Automatic Categorization of News Articles, Sentiment Text Classification, Predicting Movie Reviews, Hate Speech Detection on Facebook, and Many Other Tasks Have Been Completed in the Telugu Language; However, Idiomatic Sentence Classifier Has Not Been Developed Yet. Up until this point, the only languages supported by any of the existing systems are English and Tamil. We are putting forth the text classifier as a means of determining whether a Telugu text contains literal translations or idiomatic translations. Because Telugu is a language with few available resources, we have decided to compile our own in-house dataset. Through the utilization of the kappa score and the IRR, we were able to enhance the quality of our dataset.

CHAPTER 3

SYSTEM ARCHITECTURE

3.1 DEEP LEARNING MODELS

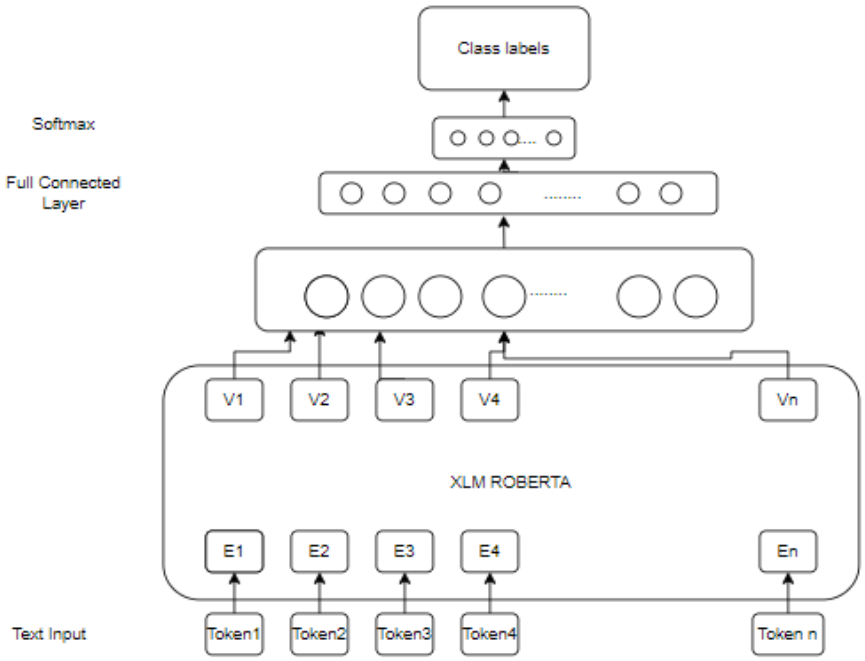


Figure 3.1.1 depicts the architecture diagram of XLM-RoBERTa

The XLM-RoBERTa block diagram can be seen in the figure that can be found above. A RoBERTa variation that has been pre-trained on a sizable corpus of monolingual and multilingual data is called the XLM-RoBERTa (Cross-lingual Language Model - Robustly Optimised BERT Approach) architecture. A series of tokens are entered into the XLM-RoBERTa input layer, which is then processed by an embedding layer that converts the tokens to dense vector representations. The meaning of each token is then captured in the context of the entire sequence using contextualised embeddings created by a multi-layer Transformer encoder, which processes the tokens in a bidirectional manner. A self-attention mechanism is built into each layer of the Transformer encoder,

allowing the model to assess the relative importance of different input sequence elements as it generates the output embeddings. A final output layer processes the Transformer encoder's output to produce predictions for upcoming activities.

XLM-RoBERTa architecture includes several modifications to the original RoBERTa architecture to support cross-lingual pre-training. For example, the model uses a dynamic masking scheme that masks tokens from different languages during pre-training to encourage the model to learn language-independent representations. Additionally, the model includes language-specific embeddings and positional embeddings to help disambiguate between languages and capture the order of the tokens in each sentence. Overall, the XLM-RoBERTa architecture is designed to learn robust, cross-lingual representations that can be fine-tuned for a variety of natural language processing tasks.

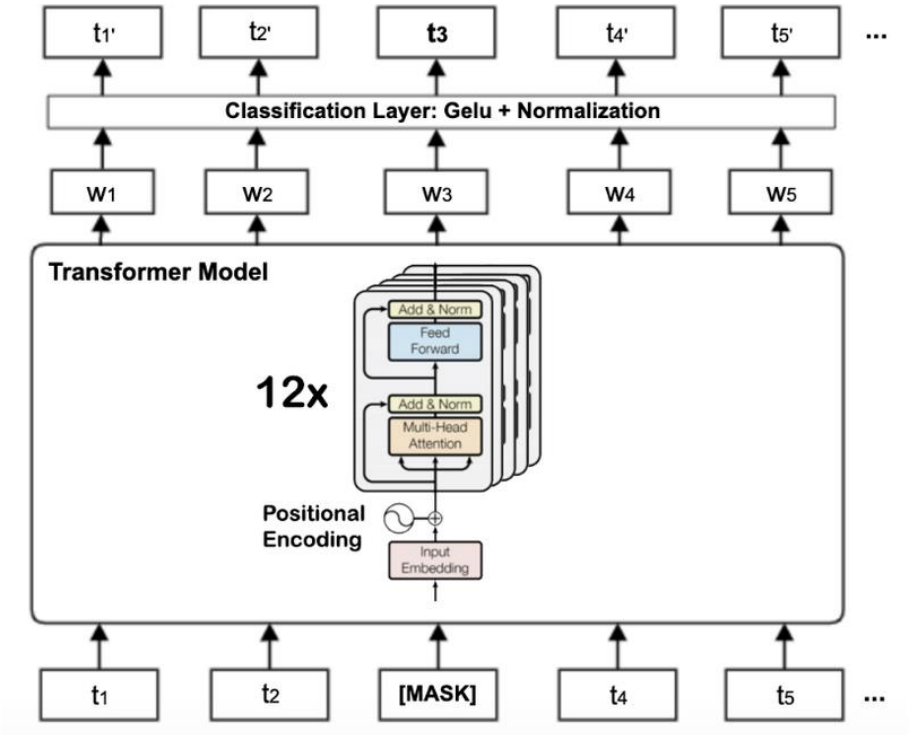


Figure 3.1.2 depicts architecture diagram of m-BERT.

The m-BERT block diagram can be seen in the figure that can be found above. The m-BERT (multilingual BERT) architecture is based on the original BERT (Bidirectional Encoder Representations from Transformers) architecture, with modifications made to support multilingual natural language processing. An input sequence of tokens is provided to the m-BERT input layer, which then processes the tokens by mapping them to dense vector representations in an embedding layer. The meaning of each token is then captured in the context of the entire sequence using contextualised embeddings created by a multi-layer Transformer encoder, which processes the tokens in a bidirectional manner. The model can weigh the relative relevance of various input sequence components when creating the output embeddings thanks to the self-attention mechanism contained in each layer of the Transformer encoder. To produce predictions for tasks farther down the pipeline, the Transformer encoder's output is routed through a final output layer.

The key modification made to the original BERT architecture to support multilingual processing is the use of a shared vocabulary across multiple languages, allowing the model to learn to represent words in multiple languages using a single set of embeddings. Additionally, the model includes language-specific input embeddings and positional embeddings to help disambiguate between languages and capture the order of the tokens in each sentence.

3.2 Machine Learning Models

Naïve Bayes

Naive Bayes is a probabilistic algorithm used in machine learning to perform classification problems. It is based on the Bayes theorem, which states that the likelihood of a hypothesis (like a class label) given the evidence is inversely proportional to the likelihood of the evidence given the hypothesis multiplied by the prior probability of the hypothesis (like the likelihood of a class label). The evidence in this case is the features of an input.

The calculation of the likelihood probabilities is made easier in the case of Naive Bayes by the presumption that the characteristics are conditionally independent given the class label. The "naive" assumption is whence the term "Naive Bayes" gets its name.

There are different types of Naive Bayes algorithms, including:

1. **Gaussian Naive Bayes**: In this case, it is assumed that the likelihood of the characteristics given the class label follows a Gaussian (normal) distribution.
2. **Multinomial Naive Bayes**: For discrete data, like as text data, where the features are the frequencies of words in a document, this is employed. It is presumptive that a multinomial distribution describes the likelihood of the features given the class label.
3. **Bernoulli Naive Bayes**: The features are assumed to be binary (either present or missing) and are used for discrete data as well. It is predicated on the idea that a Bernoulli distribution corresponds to the likelihood of the features given the class label.

To use Naive Bayes for classification, the model is trained using a labelled dataset to estimate the prior probability of the classes and the likelihood probabilities of the features given the classes. The Bayes theorem is then used to calculate the likelihood of each class, and when a new input is given, the output is expected to belong to the class with the highest probability. Naive Bayes is a simple and efficient computing strategy for binary and multi-class classification applications. However, if the "naive" assumption of conditional independence is broken or if there is a large amount of feature distribution overlap between classes, its performance may decrease.

SVC

For classification applications, the supervised machine learning method SVC (Support Vector Machine Classifier) is widely utilised. It is a particular kind of discriminative classifier that seeks out the ideal hyperplane to categorise characteristics into different groups. The fundamental principle of SVC is to move the input data into a high-dimensional space where it is simpler to locate a hyperplane that divides the classes using a kernel function. The kernel function might, among other things, be linear, polynomial, or a radial basis function (RBF). In SVC, the hyperplane that maximises the separation of the support vectors, or the data points nearest to the decision boundary, is the one that successfully distinguishes the classes.

The goal is to identify the hyperplane that maximises the margin, or the area between the hyperplane and the support vectors. SVC can still be utilised even if the classes are not linearly separable by permitting occasional misclassifications and adding a slack variable to loosen the restrictions. It is referred to as the soft-margin strategy. Once the hyperplane has been identified, fresh data points can be categorised by being projected into the hyperplane and given the appropriate class. SVC offers a number of benefits, including the capacity to manage high-dimensional data and the efficiency in managing intricate decision boundaries. Additionally, it is resistant to overfitting, particularly when the soft-margin method is used. SVC can be computationally expensive for large datasets and can be sensitive to the kernel function and its parameters. Additionally, it could not function effectively if the data is unbalanced or if there is a large amount of feature space overlap across the classes.

Logistic Regression

A popular supervised machine learning algorithm for classification tasks is logistic regression. It is a type of linear regression that models the probability of a binary or categorical result given one or more input factors. In logistic regression, the independent variables can either be continuous or categorical, and the dependent variable can be binary or categorical. The objective is to estimate the logistic regression model's parameters that best suit the data and can be used to forecast the likelihood of a result for brand-new data points. The logistic regression model maps the input variables to the output probability using a logistic function, often known as a sigmoid function.

The logistic function, which has a range from 0 to 1, is an S-shaped curve that works well for depicting the likelihood of a binary result. The probability of the positive class (for example, 1) is the logistic function's output, and the probability of the negative class (for example, 0) is just 1 minus the probability of the positive class.

Maximum likelihood estimation, which looks for the parameters that maximise the likelihood of observing the provided data given the model, is used to train the logistic regression model. Finding the values of the model coefficients that reduce the discrepancy between the expected probability and the actual results is necessary to achieve this. Once the logistic regression model has been trained, it can be used to forecast the likelihood that fresh data points will fall into the positive category, and a decision threshold may be selected to determine whether the data points should be classified as positive or negative.

Any probability that is more than or equal to 0.5 is often considered positive, while any probability that is less than 0.5 is typically considered negative. The simplicity, interpretability, and suitability of logistic regression for both continuous and categorical variables are only a few of its benefits. It is robust to

noise and outliers and may be applied to binary and multi-class classification tasks. However, logistic regression may not be successful when there is a nonlinear relationship between the independent variables and the result or when the input data is highly dimensional or collinear. Additionally, logistic regression bases its analysis on the independent nature of the observations, which may not always be the case.

3.3 Ensemble Methodology

To increase the predictive effectiveness of models, assembling is a machine learning technique. To provide more accurate predictions, it entails integrating numerous models that were each trained on the same data using various algorithms or parameters. Ensemble is done in the hopes that the models will balance each other's strengths and shortcomings, improving performance.

- By fusing several imperfect models into one solid model, ensemble approaches seek to enhance model prediction.
- Stacking, bagging, and boosting are the three most popular ensemble methods.
- Ensemble techniques are the most effective for regression and classification because they reduce bias and variance while boosting model accuracy.

Two types of ensemble methods:

1. *Independently Construction (*parallel ensemble)
2. *Coordinated Construction (*Sequential ensemble)

Ensemble is a technique that can be used to solve a variety of machine learning issues, such as classification, regression, and clustering. However, it's important

to note that ensemble can be computationally expensive and may not always lead to better performance. It's important to carefully evaluate the results and trade-offs of ensemble before applying it to a problem.

Architecture of ensemble models

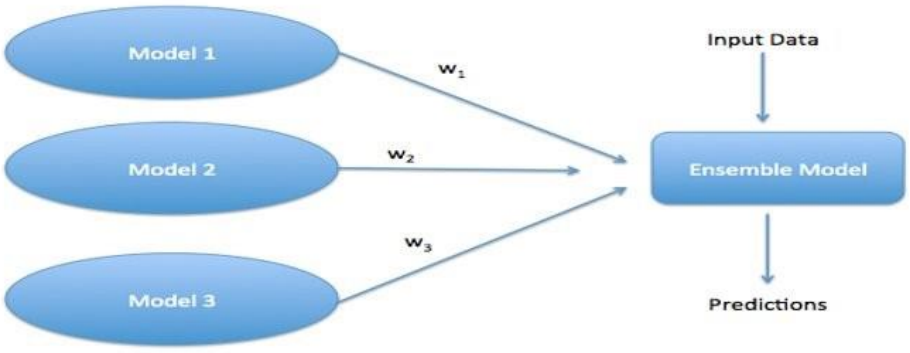


Figure 3.3.1 depicts architecture diagram of the simple average ensemble model

Figure 3.3.1 depicts the architecture diagram of the simple average ensemble model where weights w1, w2 and w3 will be one. With a basic average ensemble model, numerous individual models' predictions are combined by averaging them out. This is a form of ensemble learning technique. By lessening the effect of individual model biases and errors, this can aid in enhancing the accuracy and resilience of the overall prediction. The process of creating a simple average ensemble model involves training multiple models on the same dataset using different algorithms or variations of the same algorithm, and then taking the average of their predictions on new data. The predictions of the individual models can be weighted or unweighted depending on their performance on the validation set.

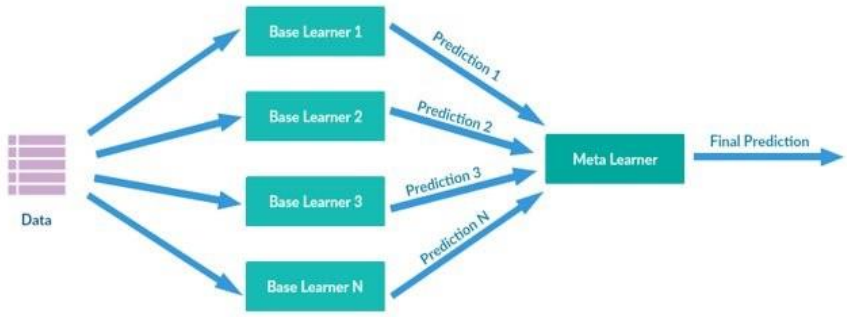


Figure 3.3.2 depicts the architecture diagram of the stacked ensemble model.

Figure 3.3.2 depicts the architecture diagram of the stacked ensemble model. A stacked ensemble model is a type of ensemble learning method where the predictions of multiple individual models are combined by training a meta-model on their outputs. The process of creating a stacked ensemble model involves training multiple individual models on the same dataset, and then using their predictions as features for a higher-level model. The higher-level model is typically a machine learning algorithm, such as a decision tree, random forest, or gradient boosting machine, which is trained on the outputs of the individual models. The idea is that the higher-level model can learn to combine the predictions of the individual models in an optimal way.

CHAPTER 4

METHODOLOGY

The pre-trained deep learning models, machine learning models and their ensemble models were used in this project to classify the idioms and literals that were contained inside the text that was provided as input. A compilation of 1040 sentences that encompasses both the literal and figurative applications of several idiomatic idioms.

4.1 In-House Dataset Creation

Since telugu is a low resource language we created our dataset, The in-house dataset includes 1040 idiom and literal sentence examples. There are a total of 520 idioms and the same number of literal sentences. Both the text sentences and the annotations for our project were authored and annotated by us. Following that, we developed a Google form in order to collect the annotations from other people depending on their perspectives. In order to determine how accurate the dataset is, we used the kappa score in conjunction with the inter-rater reliability. Kappa's score was found using the confusion matrix. The below figure has confusion matrix.

	IDIOM	LITERAL
IDIOM	a	b
LITERAL	c	d

Figure 4.1.1 shows the confusion matrix.

$$\text{Kappa score} = \frac{\text{Observed agreement - chance agreement}}{\text{1-chance agreement}}$$

Figure 4.1.2 shows the kappa score formula.

Here, observed agreement is equal to (a+d)/(a+b+c+d) and chance agreement is equal to ((a+b)*(a+c))+((b+d)*(d+c)*(a+b+d))/(a+b+c+d)).2. The results showed that there was 92% inter-rater reliability, with a kappa score of 0.8394 and an observed agreement of 0.92. It was discovered that the chance of agreement was 0.5016. As a result, the dataset's level of agreement is very high or even flawless.

The complete dataset was split into a train dataset and a test dataset using an 8:2 split ratio.

4.2 Selection of models

Since the machine only understands numbers, we have to use the tokenizer to translate the given Telugu sentence into numbers. While selecting models from machine learning we used the Bag of Words tokenizer to tokenize the Telugu sentences after that we trained the top machine learning algorithms available, and we took the models which are giving accuracy greater than 75% accuracy. After putting the models to the test, we found three algorithms—**SVM, Logistic Regression, and Naive Bayes**— offered good accuracy. Additionally, these models are combined using ensemble methods in order to produce a better model with low bias and variance. Dravidian languages are among the more than one hundred languages that the deep learning models **XLM-RoBERTa** and **m-BERT** have been trained on. Thus, that these models may be trained more effectively than machine learning models and quickly understand Telugu. For this reason, we used these models to create ensemble models using ensemble techniques. We used sentence piece tokenizer, Adamw optimizer,

cross entropy loss function, batch size 32 and 3 epochs when training XLM-RoBERTa and m-BERT. The simple average ensemble model is made by combining the individual results from both XLM-RoBERTa and m-BERT.

4.3 Training The Models

Training a machine learning model involves providing it with a set of input data and expected output, so that it can learn the underlying patterns and relationships between the two. The dataset is initially tokenized with the help of the Bag Of Words tokenizer before being used to train the machine learning models. Second, the dataset is trained using the most efficient machine-learning methods. Due to the high dimensionality, only a few algorithms performed well. SVC, Naive Bayes, and Logistic Regression are the three. For XLM-RoBERTa sentence piece tokenizer is used and for M-BERT BERT tokenizer is used so that these models can perform classification very well. After that we applied ensemble techniques to both machine learning models and deep learning models. We must tokenize because models only understand the numbers as dataset are in string format. The models are trained over tokenized dataset.

4.4 Testing The Models

Examining the models' performance on a set of data unrelated to the data used to train them entails testing the models. This is done to make sure the model is not overfitting to the training data and can generalise well to fresh data. The test data are then used to the trained model to produce predictions. The predictions are compared to the actual outputs to evaluate the model's performance. The models are assessed using the test dataset by determining its accuracy, precision, f1 score, and recall through the process of comparing the predicted outputs and supplied outputs with the assistance of a confusion matrix. This evaluation is carried out with the help of the test dataset.

Once the model has been evaluated, it is important to interpret the results to

understand how the model is making its predictions. This can involve analysing the weights and biases of the model, visualizing the feature importance's.

The next section Show the code implementation.

CHAPTER 5

CODING AND TESTING

5.1 XLM-RoBERTa model

Importing Required Libraries

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from collections import defaultdict

import torch
import torch.nn.functional as F
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import confusion_matrix, classification_report

from transformers import XLMRobertaModel, AutoTokenizer, AdamW, get_linear_schedule_with_warmup
```

Figure 5.1.1 depicts the library's required for XLM-RoBERTa model

numPy: a Python library for numerical computing that is used to manipulate data in arrays and matrices.

seaborn: a data visualization library for Python, used for creating visualizations of data.

matplotlib.pyplot: a module within the matplotlib library for creating visualizations and plots in Python.

Default dict : a built-in Python dictionary subclass that allows default values to be assigned to keys that have not yet been seen.

torch: a deep learning library for Python, used for creating and training neural network models.

sklearn.metrics: a machine learning model performance evaluation module found in the Python scikit-learn toolkit.

XLMRoBERTaModel: A RoBERTa-based pre-trained language model that is optimised for cross-lingual tasks is the XLM-RoBERTa model, which is implemented in PyTorch.

AutoTokenizer: A utility class that can be used to automatically select and load a tokenizer for a given language model, based on its name or path.

AdamW: an implementation of the weight decay regularization based AdamW optimizer, a subset of the Adam optimizer.

get_linear_schedule_with_warmup: A utility function that generates a learning rate schedule for use with the AdamW optimizer, with a warm-up period followed by a linear decay.

Reading the dataset

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_excel('/content/PROJECT DATASET.xlsx',names=['text','label'])
```

Figure 5.1.2 depicts reading the dataset

import pandas as pd: provides the pandas library, which is used in Python for data analysis and manipulation, the alias 'pd' to make it easier to use.

from sklearn.model_selection import train_test_split: enables the import of the train_test_split scikit-learn library function, which divides datasets into training and testing sections.
df = pd.read_excel('/content/PROJECT DATASET.xlsx',names=['text','label']): reads an Excel file named ' PROJECT Dataset .xlsx' located in the '/content' directory, and assigns the columns of the dataset the names 'text' and 'label'. The resulting dataset is stored in a pandas DataFrame object named 'df'.

Label Encoding

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df["label_int"] = le.fit_transform(df['label'])
df.head()
```

	text	label	label_int
0	రమేష్ రాజేష్ కి పక్కలో బెల్లంలా ఉన్నాడు	Idiom	0
1	రాజేష్ మాటలే కానీ చేతలు లేదు తెలిసిందే కదా మొర...	Literal	1
2	సెగలేనిదే పొగ రాదన్నట్లు రవి ఎవరు చెప్పకుండా ఏ...	Literal	1
3	మళ్లీ మళ్లీ రిపీట్ చేస్తూ చూస్తున్న వాళ్ళు ఎంత...	Literal	1
4	పొయ్యి కనీసం కొడుకులకు పని పాట లేదు కా బట్టి ద...	Literal	1

Figure 5.1.3 depicts the label encoding of dataset

from sklearn.preprocessing import LabelEncoder: enables the import of the LabelEncoder class from the scikit-learn library, which is used to encode labels as integers.

le = LabelEncoder(): creates a new instance of the LabelEncoder class and assigns it to a variable named 'le'.

df["label_int"] = le.fit_transform(df['label']): creates a new column in the DataFrame named 'label_int' and assigns to it the encoded label values obtained by applying the fit_transform method of the LabelEncoder object 'le' to the 'label' column of the DataFrame.

df.head(): displays the first few rows of the DataFrame to confirm that the label encoding was successful.

Countplot

```
sns.countplot(x=df.label_int)
```

```
<Axes: xlabel='label_int', ylabel='count'>
```

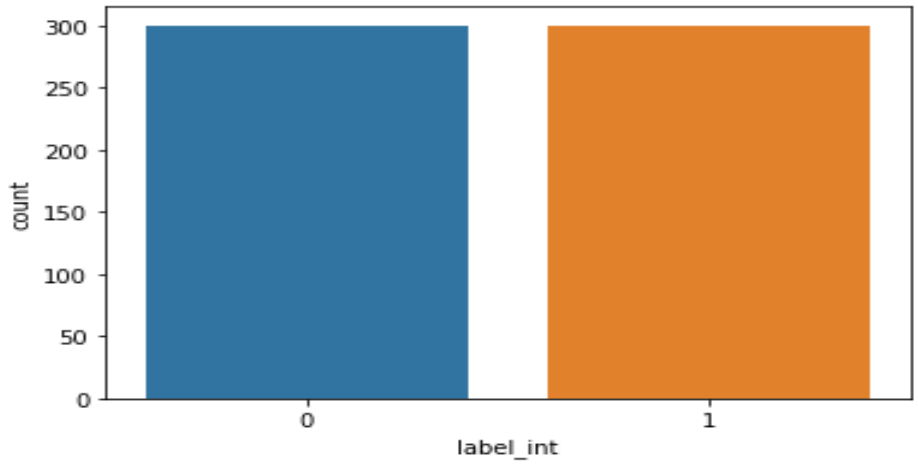


Figure 5.1.4 depicts the countplot of dataset

‘sns.countplot(x=df.label_int)" uses the seaborn library to create a count plot of the values in the "label_int" column of the dataframe "df". A count plot is a kind of bar plot that shows how frequently each categorical value appears in a dataset. In this case, the "label_int" column is assumed to contain categorical data that has been encoded as integers, and the count plot will display the number of occurrences of each integer value in the column.

By using the "x=" parameter to specify the "label_int" column, the count plot will display the counts of each unique integer value in the column along the horizontal axis of the plot. The vertical axis of the plot will display the frequency of occurrence of each integer value. This type of plot is useful for quickly visualizing the distribution of categorical data in a dataset.

Making Datasets and Data Loaders

```
class TeluguDataset(Dataset):

  def __init__(self, text, label, tokenizer, max_len):
    self.text = text
    self.label = label
    self.tokenizer = tokenizer
    self.max_len = max_len

  def __len__(self):
    return len(self.text)

  def __getitem__(self, item):
    text = str(self.text[item])
    target = self.label[item]

    encoding = self.tokenizer.encode_plus(
        text,
        add_special_tokens=True,
        max_length=self.max_len,
        padding='max_length',
        return_token_type_ids=False,
        return_attention_mask=True,
        return_tensors='pt',
        truncation=True,
    )
    return {
        'text': text,
        'input_ids': encoding['input_ids'].flatten(),
        'attention_mask': encoding['attention_mask'].flatten(),
        'targets': torch.tensor(target, dtype=torch.long),
    }
```

Figure 5.1.5 depicts class for encoding the dataset

This code defines a custom dataset class for a Telugu language text classification task. The class overrides the __len__ and __getitem__ methods while deriving from PyTorch's Dataset class.

The **__init__** method initializes the dataset with the Telugu text, labels, tokenizer, and maximum sequence length. The dataset's length, or the number of text instances, is returned via the __len__ method. A specific sample from the dataset can be retrieved using the __getitem__ method and is identified by an index item.

In the **__getitem__** method, the Telugu text example is first tokenized using the provided tokenizer. The resulting input ids and attention mask are then flattened and returned as PyTorch tensors, along with the corresponding target label. The **flatten()** method is used to ensure that the input tensors have the same dimensions, regardless of the length of the input text. Finally, the input ids, attention mask, and target label are returned as a dictionary with keys 'text', 'input_ids', 'attention_mask', and 'targets', respectively.

```
def create_data_loader(df, tokenizer, max_len, batch_size):
  ds = TeluguDataset(
    text=df.text.to_numpy(),
    label=df.label_int.to_numpy(),
    tokenizer=tokenizer,
    max_len=max_len
  )

  return DataLoader(
    ds,
    batch_size=batch_size,
  )
```

Figure 5.1.6 depicts create_dataz_loader function

This function creates a PyTorch DataLoader object from a Pandas DataFrame. It takes as input the DataFrame, a tokenizer object, the maximum length of the tokenized sequences, and the desired batch size for the DataLoader.

The function first creates a **TeluguDataset** object from the DataFrame and the input arguments. It then returns a DataLoader object initialized with the created dataset and the batch size.

Batch Size

```
BATCH_SIZE = 32

train_data_loader1 = create_data_loader(df_train, tokenizer1, MAX_LEN, BATCH_SIZE)
val_data_loader1 = create_data_loader(df_test, tokenizer1, MAX_LEN, BATCH_SIZE)
```

Figure 5.1.7 depicts applying create_data_loader function to dataset

The variable "BATCH_SIZE" is set to 32, which means that the data loader will load 32 examples at a time during training and validation. The batch size is an important hyperparameter that affects the speed and memory usage of training. Smaller batch sizes typically require less memory but may result in slower training, while larger batch sizes may require more memory but can speed up training.

We then use the "create_data_loader" function to create data loaders for the training and validation sets. This function takes as input the training or validation data, the tokenizer used to preprocess the data, the maximum

sequence length, and the batch size.

The "train_data_loader" and "val_data_loader" variables are used to store the data loaders for the training and validation sets, respectively. These data loaders can be used to feed batches of preprocessed data to the model during training and validation.

```
class TeluguClassifier1(nn.Module):

  def __init__(self, n_class):
    super(TeluguClassifier1, self).__init__()
    self.bert = XLMRobertaModel.from_pretrained(PRE_TRAINED_MODEL_NAME1, return_dict=False)
    self.drop = nn.Dropout(p=0.4)
    self.out = nn.Linear(self.bert.config.hidden_size, n_class)

  def forward(self, input_ids, attention_mask):
    _, pooled_output = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask
    )
    output = self.drop(pooled_output)
    return self.out(output)
```

Figure 5.1.8 depicts the class telugu classifier

This is a PyTorch module that implements a classifier for Telugu text using XLM-RoBERTa, a pre-trained transformer model for natural language processing. The **__init__** method initializes the module by loading the pre-trained XLM-RoBERTa model and setting up the neural network architecture. Specifically, it creates a dropout layer to help prevent overfitting, and a linear layer to produce output predictions.This is a basic implementation of a PyTorch neural network model using XLM-RoBERTa as a base model. The **TeluguClassifier1** class extends the **nn.Module** class in PyTorch, which provides a flexible way to define complex neural network architectures. The constructor of this class initializes the XLM-RoBERTa model, followed by a dropout layer and a linear layer for classification. The **forward** method takes the input_ids and attention_mask as inputs and passes them through the XLM-RoBERTa model, followed by the dropout layer and the linear layer for classification. The output of this method is the final classification scores for the input text.

The neural network's forward pass is carried out using the forward technique. It accepts input_ids and attention_mask tensors, which the tokenizer generates and which represent the input text in a way that the trained model can comprehend. The pooled_output tensor, which represents the output of the last layer of the pre-trained model for the full input sequence, is obtained after the input has been passed through the XLM-RoBERTa model. The final output prediction is then generated by passing the pooled_output tensor through the dropout layer and the linear layer. The function returns the output.

```
model1 = TeluguClassifier1(len(class_names))
model1 = model1.to(device)
```

Figure 5.1.9 depicts creation of object model1

The **model1** has been initialized and moved to the specified device (CPU/GPU) using the **to()** method.

Epochs

```
EPOCHS = 3

optimizer1 = AdamW(model1.parameters(), lr=1e-5, correct_bias=False)
total_steps1 = len(train_data_loader1) * EPOCHS

scheduler1 = get_linear_schedule_with_warmup(
  optimizer1,
  num_warmup_steps=0,
  num_training_steps=total_steps1
)

loss_fn1 = nn.CrossEntropyLoss().to(device)
```

Figure 5.1.10 depicts the epochs, optimizer and loss function.

We will train the model for 3 epochs because the variable "EPOCHS" is set to 3. A full traverse across the entire training dataset constitutes one epoch.

The AdamW optimizer from the PyTorch package, a variation of the Adam

optimizer that incorporates weight decay, is used to define the "optimizer" variable. The learning rate is set to 1e-5, which controls how quickly the optimizer adjusts the model weights during training. The "correct_bias" parameter is set to False to exclude the bias correction term in the optimizer update, as recommended in the original AdamW paper.

The number of batches in the training data loader is multiplied by the quantity of epochs to determine the "total_steps" variable. The learning rate during training is modified using this parameter.

```
Epoch 1/3
----------
Training loss 0.687092657883962 accuracy 0.5125
Validation loss 0.5597424283623695 accuracy 0.725

Epoch 2/3
----------
Training loss 0.5597447554270426 accuracy 0.7
Validation loss 0.5769151523709297 accuracy 0.75

Epoch 3/3
----------
Training loss 0.47395495772361756 accuracy 0.7875
Validation loss 0.4083656519651413 accuracy 0.775

CPU times: user 19.6 s, sys: 5.71 s, total: 25.3 s
Wall time: 42.1 s
```

Figure 5.1.11 depicts the accuracy for every epoch.

We will train the model for 3 epochs because the variable "EPOCHS" is set to 3. A full traverse across the entire training dataset constitutes one epoch.

The AdamW optimizer from the PyTorch package, a variation of the Adam optimizer that incorporates weight decay, is used to define the "optimizer" variable. The learning rate is set to 1e-5, which controls how quickly the optimizer adjusts the model weights during training. The "correct_bias" parameter is set to False to exclude the bias correction term in the optimizer update, as recommended in the original AdamW paper.

The number of batches in the training data loader is multiplied by the quantity of epochs to determine the "total_steps" variable. The learning rate during training is modified using this parameter.

Training epochs

```
def train_epoch(
    model,
    data_loader,
    loss_fn,
    optimizer,
    device,
    scheduler,
    n_examples
    ):
  model = model.train()

  losses = []
  correct_predictions = 0

  for d in data_loader:
    input_ids = d['input_ids'].to(device)
    attention_mask = d['attention_mask'].to(device)
    targets = d["targets"].to(device)

    outputs = model(input_ids=input_ids, attention_mask=attention_mask)

    _, preds = torch.max(outputs, dim=1)
    loss = loss_fn(outputs, targets)

    correct_predictions += torch.sum(preds == targets)
    losses.append(loss.item())
```

Figure 5.1.12 depicts train_epoch function.

The **train_epoch** function takes in the model, data loader, loss function, optimizer, device, scheduler, and the total number of training examples. It then trains the model for one epoch, i.e., one pass through the entire training data.

Inside the function, the model is set to train mode using **model.train()**. The correct predictions and losses are initialized to zero.

For each batch of the data loader, the input_ids, attention_mask, and targets are extracted and moved to the device. The **model** is called with **input_ids** and

attention_mask as input, which returns the **outputs**. The **outputs** are passed to the **loss_fn** along with the **targets** to calculate the loss.

The predictions are obtained by taking the argmax of the **outputs** along the dimension 1. The number of items where the preds and targets match is then multiplied by the number of right predictions. The loss is appended to the **losses** list.

The gradients are then calculated using **loss.backward()**. The gradients are clipped using the nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0) function to a maximum value of 1.0. Using optimizer.step(), the optimizer is updated. The scheduler is updated using **scheduler.step()**. Finally, the gradients are set to zero using **optimizer.zero_grad()**.

The function returns the accuracy and average loss of the epoch. The ratio of the number of accurate predictions to all training examples is known as accuracy.

Evaluation the model

```
def eval_model(model, data_loader, loss_fn, device, n_examples):
  model = model.eval()

  losses = []
  correct_predictions = 0

  with torch.no_grad():
    for d in data_loader:
      input_ids = d['input_ids'].to(device)
      attention_mask = d['attention_mask'].to(device)
      targets = d["targets"].to(device)

      outputs = model(input_ids=input_ids, attention_mask=attention_mask)

      _, preds = torch.max(outputs, dim=1)
      loss = loss_fn(outputs, targets)

      correct_predictions += torch.sum(preds == targets)
      losses.append(loss.item())

  return correct_predictions.double() / n_examples, np.mean(losses)
```

Figure 5.1.13 depicts the eval_model function.

The **eval_model** function evaluates the performance of the trained model on the given dataset using the provided loss function. It takes the following inputs:

- **model**: the trained PyTorch model to evaluate.
- **data_loader**: the PyTorch data loader containing the dataset to evaluate the model on.
- **loss_fn**: the loss function that will be used to assess the model.
- **device**: the device to run the evaluation on.
- **n_examples**: the dataset's overall example counts.

The function first sets the model to evaluation mode using the **model.eval()** method. It then iterates over the batches in the data loader and computes the model's predictions and the corresponding loss for each batch. The predictions are compared against the ground truth targets to calculate the number of correct predictions. Finally, the function returns the accuracy and average loss across all batches.

Getting predictions

```
def get_predictions(model, data_loader):
  model = model.eval()

  data_text = []
  predictions = []
  prediction_probs = []
  real_values = []

  with torch.no_grad():
    for d in data_loader:

      texts = d["text"]
      input_ids = d["input_ids"].to(device)
      attention_mask = d["attention_mask"].to(device)
      targets = d["targets"].to(device)

      outputs = model(
        input_ids=input_ids,
        attention_mask=attention_mask
      )
      _, preds = torch.max(outputs, dim=1)

      probs = F.softmax(outputs, dim=1)

      data_text.extend(texts)
      predictions.extend(preds)
```

Figure 5.1.14 depicts get_predictions function.

The **get_predictions()** function takes a trained model and a data loader, and returns the predicted labels and corresponding probabilities, along with the actual labels and texts. It does so by running the model on the input data, computing the softmax probabilities, and returning the predicted label, probability, actual label, and text for each example in the dataset. Note that the function first puts the model in evaluation mode by calling **model.eval()**, and then disables gradient computation using the **torch.no_grad()** context manager to speed up the computation.

Confusion Matrix

```
def show_confusion_matrix(confusion_matrix):
  hmap = sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30, ha='right')
  plt.ylabel('True Label')
  plt.xlabel('Predicted Label');

cm = confusion_matrix(y_test1, y_pred1)
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
show_confusion_matrix(df_cm)
```

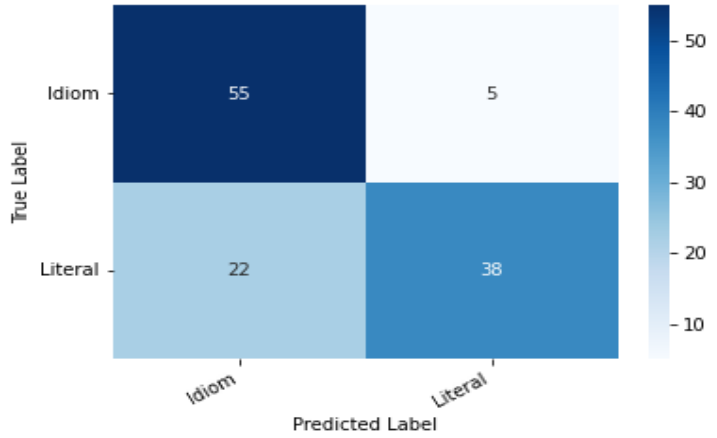


Figure 5.1.15 depicts show_confusion_matrix function.

A confusion matrix, which is commonly produced by contrasting a classifier's predicted labels with the actual labels, is an input for the function.

A confusion matrix is a table that evaluates the performance of a classification model by comparing the expected and actual class labels of a set of test data. Each real class label has a row, and each anticipated class label has a column,

in the matrix. The diagonal of the matrix displays the number of accurate predictions, and the entries off the diagonal display the number of inaccurate estimates.

5.2 BERT MODEL

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_excel('/content/PROJECT DATASET.xlsx',names=['text','label'])
```

```
df.head()
```

	text	label
0	రమేష్ రాజేష్ కి పక్కలో బెల్లంలా ఉన్నాడు	Idiom
1	రాజేష్ మాటలే కానీ చేతలు లేదు తెలిసిందే కదా మొర...	Literal
2	సెగలేనిదే పొగ రాదన్నట్లు రవి ఎవరు చెప్పకుండా ఏ...	Literal
3	మళ్లీ మళ్లీ రిపీట్ చేస్తూ చూస్తున్న వాళ్ళు ఎంత...	Literal
4	పొయ్యి కనీసం కొడుకులకు పని పాట లేదు కా బట్టి ద...	Literal

Figure 5.2.1 depicts the reading of the dataset.

numpy: a Python library for numerical computing that is used to manipulate data in arrays and matrices.

seaborn: a data visualization library for Python, used for creating visualizations of data.

Matplotlib.pyplot: a module within the matplotlib library for creating visualizations and plots in Python.

Default dict : a built-in Python dictionary subclass that allows default values to be assigned to keys that have not yet been seen.

torch: a deep learning library for Python, used for creating and training neural network models.

Countplot

```
sns.countplot(x=df.label_int)
```

```
<Axes: xlabel='label_int', ylabel='count'>
```

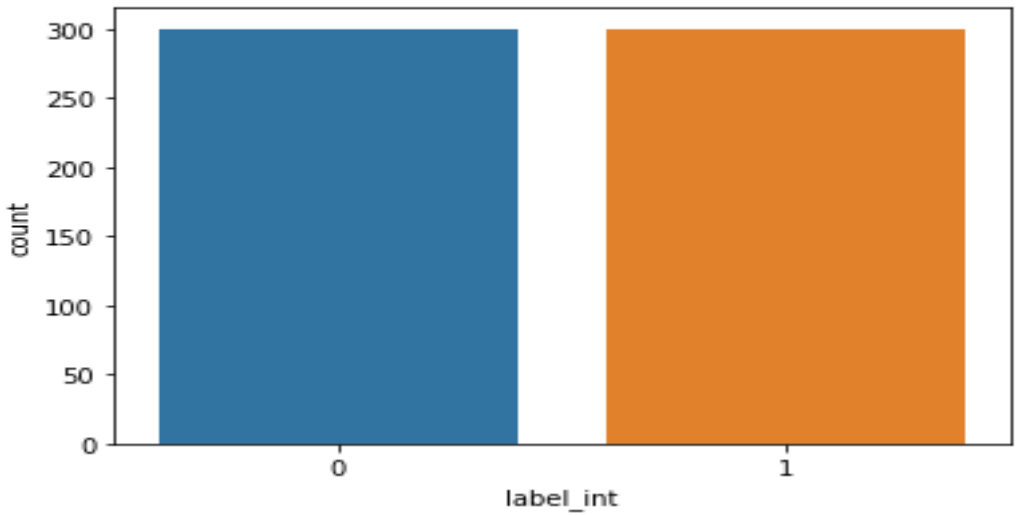


Figure 5.2.2 depicts the countplot of dataset.

In order to construct a bar plot of the frequency distribution of values in the 'label' column of a pandas DataFrame called 'df,' the command sns.countplot(x=df.label) uses the seaborn library.

A high-level interface for designing eye-catching and educational statistics visuals is offered by the seaborn library. A bar plot showing the frequency distribution of a categorical variable is produced using the countplot() function of the seaborn library.

The column name of the DataFrame containing the data to be plotted on the x-axis is specified by the x argument of the countplot() function. It is "label" in this instance. The count of each distinct value in the "label" column will be shown on the y-axis of the final plot.

Pre-Trained Model

```
PRE_TRAINED_MODEL_NAME = 'bert-base-multilingual-cased'
```

```
tokenizer = BertTokenizer.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

```
MAX_LEN = 100
```

```
class TeluguDataset(Dataset):

  def __init__(self, text, label, tokenizer, max_len):
    self.text = text
    self.label = label
    self.tokenizer = tokenizer
    self.max_len = max_len
```

Figure 5.2.3 depicts class Telugu dataset.

Using the BERT pre-trained model supplied by the PRE-TRAINED MODEL NAME option, this line of code constructs a tokenizer object. The BERT model receives input in the form of tokenized sub words or word fragments that are generated by the tokenizer.

The BERT Tokenizer class has a method called from pretrained that loads a pre-trained tokenizer from the given PRE-TRAINED MODEL NAME. A path to a local directory containing the pre-trained tokenizer files or a string giving the name of a pre-trained model housed on Hugging Face's model hub can be used as the PRE-TRAINED MODEL NAME argument. By using the tokenizer object's encode method after it has been built, text can be tokenized.

Data Loader

```
def create_data_loader(df, tokenizer, max_len, batch_size):
  ds = TeluguDataset(
    text=df.text.to_numpy(),
    label=df.label_int.to_numpy(),
    tokenizer=tokenizer,
    max_len=max_len
  )

  return DataLoader(
    ds,
    batch_size=batch_size,
  )
```

Figure 5.2.4 depicts the create_data_loader function.

A PyTorch DataLoader object may be created for a specified DataFrame df of text data using the create data loader method. Three inputs are required by the function: a tokenizer object, a max len parameter, and a batch size parameter.

Epochs

```
optimizer2 = AdamW(model2.parameters(), lr=1e-5, correct_bias=False)
total_steps2 = len(train_data_loader2) * EPOCHS

scheduler2 = get_linear_schedule_with_warmup(
  optimizer2,
  num_warmup_steps=0,
  num_training_steps=total_steps2
)

loss_fn2 = nn.CrossEntropyLoss().to(device)
```

Figure 5.2.5 depicts the optimizer, loss function.

When using PyTorch to train a neural network model, this code block configures the optimizer, learning rate scheduler, and loss function.

The number of training epochs is initially set at 3.

The AdamW optimizer, a well-known version of the famous Adam optimizer that employs weight decay regularisation to enhance generalisation efficiency, is then created as a new instance.

The learning rate, which is set to 1e-5, is specified by the lr parameter (i.e., 0.000001). For the optimizer to stop using bias correction, the correct bias option is set to False.

Then, a learning rate scheduler is made using the get linear schedule with warmup function, which linearly raises the learning rate from 0 during a warmup phase and then linearly lowers it to 0 at the conclusion of training.

The cross-entropy loss is computed using the loss function between the expected and actual class labels. The effectiveness of the model during training is assessed when backpropagation is employed to update the model parameters. To relocate the loss function to the computer (such as a CPU or GPU) that will do the calculations, the to approach is utilized.

Training epochs

```
def train_epoch(
    model,
    data_loader,
    loss_fn,
    optimizer,
    device,
    scheduler,
    n_examples
    ):
  model = model.train()

  losses = []
  correct_predictions = 0

  for d in data_loader:
    input_ids = d['input_ids'].to(device)
    attention_mask = d['attention_mask'].to(device)
    targets = d["targets"].to(device)

    outputs = model(input_ids=input_ids, attention_mask=attention_mask)

    _, preds = torch.max(outputs, dim=1)
    loss = loss_fn(outputs, targets)

    correct_predictions += torch.sum(preds == targets)
    losses.append(loss.item())
```

Figure 5.2.6 depicts the train epoch function.

The **train_epoch** function takes in the model, data loader, loss function, optimizer, device, scheduler, and the total number of training examples. It then trains the model for one epoch, i.e., one pass through the entire training data.

Inside the function, the model is set to train mode using **model.train()**. The correct predictions and losses are initialized to zero.

For each batch of the data loader, the input_ids, attention_mask, and targets are extracted and moved to the device. Input_ids and attention_mask are passed when calling the model, which then delivers the results. The **outputs** are passed to the **loss_fn** along with the **targets** to calculate the loss.

The predictions are obtained by taking the argmax of the **outputs** along dimension 1. The number of items where the preds and targets match is then multiplied by the number of right predictions. The loss is appended to the **losses** list.

The gradients are then calculated using **loss.backward()**. Using nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0), the gradients are clipped to a maximum value of 1.0. Optimizer.step() is used to update the optimizer. The scheduler is updated using **scheduler.step()**. Finally, the gradients are set to zero using **optimizer.zero_grad()**. The function returns the accuracy and average loss of the epoch. The proportion of successful predictions to all training instances is known as accuracy.

Evaluation the model

```
def eval_model(model, data_loader, loss_fn, device, n_examples):
  model = model.eval()

  losses = []
  correct_predictions = 0

  with torch.no_grad():
    for d in data_loader:
      input_ids = d['input_ids'].to(device)
      attention_mask = d['attention_mask'].to(device)
      targets = d["targets"].to(device)

      outputs = model(input_ids=input_ids, attention_mask=attention_mask)

      _, preds = torch.max(outputs, dim=1)
      loss = loss_fn(outputs, targets)

      correct_predictions += torch.sum(preds == targets)
      losses.append(loss.item())

  return correct_predictions.double() / n_examples, np.mean(losses)
```

Figure 5.2.7 depicts the eval_model function.

The **eval_model** function evaluates the performance of the trained model on the given dataset using the provided loss function. It takes the following inputs:

- **model**: the trained PyTorch model to evaluate.
- **data_loader**: the PyTorch data loader containing the dataset to evaluate the model on.
- **loss_fn**: the loss function that will be used to assess the model.
- **device**: the device to run the evaluation on.
- **n_examples**: the dataset's overall example count.

The function first sets the model to evaluation mode using the **model.eval()**

method. It then iterates over the batches in the data loader and computes the model's predictions and the corresponding loss for each batch. The predictions are compared against the ground truth targets to calculate the number of correct predictions. Finally, the function returns the accuracy and average loss across all batches.

Getting predictions

```
def get_predictions(model, data_loader):
  model = model.eval()

  data_text = []
  predictions = []
  prediction_probs = []
  real_values = []

  with torch.no_grad():
    for d in data_loader:

      texts = d["text"]
      input_ids = d["input_ids"].to(device)
      attention_mask = d["attention_mask"].to(device)
      targets = d["targets"].to(device)

      outputs = model(
        input_ids=input_ids,
        attention_mask=attention_mask
      )
      _, preds = torch.max(outputs, dim=1)

      probs = F.softmax(outputs, dim=1)

      data_text.extend(texts)
      predictions.extend(preds)
```

Figure 5.2.8 depicts get_predictions function.

The **get_predictions()** function takes a trained model and a data loader, and returns the predicted labels and corresponding probabilities, along with the actual labels and texts. It does so by running the model on the input data, computing the softmax probabilities, and returning the predicted label, probability, actual label, and text for each example in the dataset. Note that the function first puts the model in evaluation mode by calling **model.eval()**, and then disables gradient computation using the **torch.no_grad()** context manager to speed up the computation.

Confusion Matrix

```
def show_confusion_matrix(confusion_matrix):
  hmap = sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30, ha='right')
  plt.ylabel('True Label')
  plt.xlabel('Predicted Label');

cm = confusion_matrix(y_test2, y_pred2)
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
show_confusion_matrix(df_cm)
```

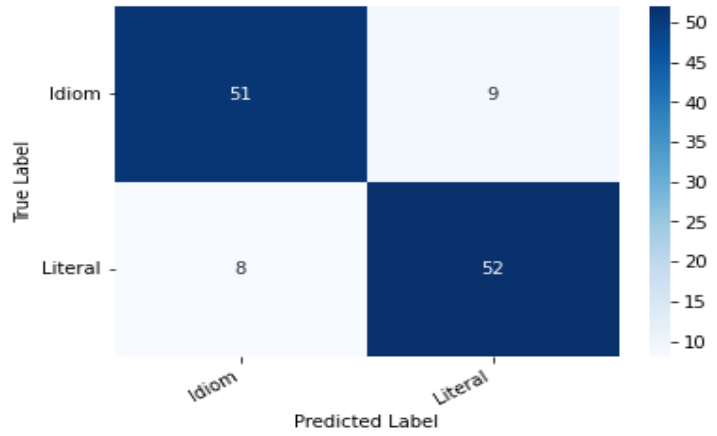


Figure 5.2.9 depicts the show_confusion_matrix function.

A confusion matrix is sent to the show confusion matrix function, which formats it for display.

In order to assess how well a classification model performed, a confusion matrix is a table that contrasts the expected and actual class labels of a set of test data. The matrix has columns for each predicted class label and rows for each real class label. The matrix's diagonal represents the number of accurate predictions, and the entries outside of the diagonal reflect the number of incorrect guesses.

```
print(classification_report(y_test2, y_pred2, target_names=class_names))
```

```
              precision    recall  f1-score   support

       Idiom       0.86      0.85      0.86        60
     Literal       0.85      0.87      0.86        60

    accuracy                           0.86       120
   macro avg       0.86      0.86      0.86       120
weighted avg       0.86      0.86      0.86       120
```

Figure 5.2.10 depicts the classification report of M-BERT

The scikit-learn utility function classification report creates a text report with the key classification metrics for a collection of anticipated and actual class labels.

The following parameters are passed to classification report when it is invoked in this code block:

y test: the test set's actual class labels

y pred: the test set's anticipated class labels

digits: the number of digits to display for floating point values in the report target names: a list of strings containing the names of the classes

The text that is returned by the classification report function includes the metrics shown below for each class:

The ratio of correct positive forecasts to all positive predictions serves as a measure of precision.

Recall is defined as the proportion of precise predictions to all positive samples in the data, also referred to as the "true positive prediction rate."

F1-score: a fair comparison of the two metrics, which is the harmonic mean of recall and accuracy.

Support: The proportion of class-related samples in the test data

5.3 ML ALGORITHMS

Installing vecstack

```
!pip install vecstack
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: vecstack in /usr/local/lib/python3.8/dist-packages (0.4.0)
Requirement already satisfied: scikit-learn>=0.18 in /usr/local/lib/python3.8/dist-packages (from vecstack) (1.0.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.8/dist-packages (from vecstack) (1.7.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (from vecstack) (1.22.4)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.8/dist-packages (from scikit-learn>=0.18->vecstack) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.8/dist-packages (from scikit-learn>=0.18->vecstack) (3.1.0)
```

Figure 5.3.1 depicts installing the required library

Vecstack is a Python library for stacking (ensemble) machine learning models. Stacking is a technique used in machine learning to combine multiple models in order to improve predictive accuracy. In order to make the final prediction, a final meta-model receives input from numerous base models that have been trained on the same dataset. Vecstack provides a simple and flexible API for stacking models in Python. It supports a wide range of base models and meta-models, including linear models, tree-based models, neural networks, and more. You can easily customize the stacking process by choosing the base models, meta-model, and training parameters, and you can also add custom preprocessing steps and feature engineering.

Including required libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from collections import defaultdict
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix, classification_report
```

Figure 5.3.2 depicts the importing the required libraries.

numpy: a Python library for numerical computing that is used to manipulate data in arrays and matrices.

seaborn: a data visualization library for Python, used for creating visualizations of data.

Matplotlib.pyplot: a module within the matplotlib library for creating visualizations and plots in Python.

Default dict : a built-in Python dictionary subclass that allows default values to be assigned to keys that have not yet been seen.

Reading the dataset and printing data frame

```
df = pd.read_excel('/content/PROJECT DATASET.xlsx',names=['text','label'])
```

```
df
```

	text	label
0	రమేష్ రాజేష్ కి పక్కలో బెల్లంలా ఉన్నాడు	Idiom
1	రాజేష్ మాటలే కానీ చేతలు లేదు తెలిసిందే కదా మొర...	Literal
2	సెగలేనిదే పొగ రాదన్నట్లు రవి ఎవరు చెప్పకుండా ఏ...	Literal
3	మళ్లీ మళ్లీ రిపీట్ చేస్తూ చూస్తున్న వాళ్ళు ఎంత...	Literal
4	పొయ్యి కనీసం కొడుకులకు పని పాట లేదు కా బట్టి ద...	Literal

Figure 5.3.3 depicts the reading the dataset.

df = pd.read_excel('/content/PROJECT DATASET.xlsx',names=['text','label']): reads an Excel file named ' PROJECT Dataset .xlsx' located in the '/content' directory, and assigns the columns of the dataset the names 'text' and 'label'. The resulting dataset is stored in a pandas DataFrame object named 'df'.

Label encoding

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df["label_int"] = le.fit_transform(df['label'])
df.head()
```

	text	label	label_int
0	రమేష్ రాజేష్ కి పక్కలో బెల్లంలా ఉన్నాడు	Idiom	0
1	రాజేష్ మాటలే కానీ చేతలు లేదు తెలిసిందే కదా మొర...	Literal	1
2	సెగలేనిదే పొగ రాదన్నట్లు రవి ఎవరు చెప్పకుండా ఏ...	Literal	1
3	మళ్లీ మళ్లీ రిపీట్ చేస్తూ చూస్తున్న వాళ్ళు ఎంత...	Literal	1
4	పొయ్యి కనీసం కొడుకులకు పని పాట లేదు కా బట్టి ద...	Literal	1

Figure 5.3.4 depicts the label encoding of dataset.

le = LabelEncoder(): creates a new instance of the LabelEncoder class and assigns it to a variable named 'le'.

df["label_int"] = le.fit_transform(df['label']): creates a new column in the DataFrame named 'label_int' and assigns to it the encoded label values obtained by applying the fit_transform method of the LabelEncoder object 'le' to the 'label' column of the DataFrame.

df.head(): displays the first few rows of the DataFrame to confirm that the label encoding was successful.

Splitting the dataset

```
df_train, df_test = train_test_split(df, test_size=0.2)
```

Figure 5.3.5 depicts the splitting the dataset.

In order to divide a dataset into training and test sets for machine learning, you can use the scikit-learn library's train_test_split method. The function takes as input the dataset to be split (usually a panda DataFrame or numpy array), and the proportion of the dataset to use for testing. The output of the function is two subsets of the original dataset, one for training and one for testing.

Naïve bayes

```
# Import necessary libraries
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

# Split the dataset into train and test sets
train_data = df_train.iloc[:,[0,2]]
test_data = df_test.iloc[:,[0,2]]

# Create a bag of words representation of the text
vectorizer = CountVectorizer()
train_bow = vectorizer.fit_transform(train_data['text'])
test_bow = vectorizer.transform(test_data['text'])
# Train the model
model = MultinomialNB()
model.fit(train_bow, train_data['label_int'])

# Predict on the test set
predicted3 = model.predict(test_bow)

# Evaluate the model
accuracy = (predicted3 == test_data['label_int']).mean()
print("Accuracy for naive bayes is ", accuracy)
```

```
Accuracy for naive bayes is  0.7666666666666667
```

Figure 5.3.6 depicts the code for Naïve Bayes

MultinomialNB is a classifier in scikit-learn library that is based on the Naive Bayes algorithm. It is applied to challenges involving the classification of texts whose properties are the frequency of the text's words. It assumes that the features are independent of each other and follows a multinomial distribution. It is often used in spam filtering, sentiment analysis, and topic classification tasks. The **MultinomialNB** classifier can be trained on a training set of data using the **fit(X, y)** method, where Y is the target vector and X is the feature matrix. Using the predict(X) method, where X is the feature matrix of the new data, the classifier may be used to predict the target values of a new set of data after being trained. The classifier also provides a **predict_proba(X)** method that

returns the probabilities of each class label for each sample in **X**.

This code snippet demonstrates how to use a Naive Bayes classifier to perform text classification on a dataset. Here's a breakdown of the code:

- First, the necessary libraries are imported: **CountVectorizer** from **sklearn.feature_extraction.text** and **MultinomialNB** from **sklearn.naive_bayes**.
- The code then splits the dataset into training and test sets using the **train_test_split** function, which is not shown in this code snippet.
- The **train_data** and **test_data** variables are created by selecting the columns of interest (the text and label columns) from the respective sets.
- A **CountVectorizer** object is created to convert the text into a bag of words representation. To fit the vocabulary and convert the text into a matrix of token counts, the fit_transform technique is used on the training set. To convert the text in the test set into the same format as the training set, the transform method is invoked.
- A **MultinomialNB** object is created to train the Naive Bayes model. The **fit** method is called on the bag of words matrix and the label column of the training set.
- The **predict** method is called on the bag of words matrix of the test set to make predictions on the test set.
- By contrasting the predicted labels with the actual labels from the test set, the accuracy of the model is determined.

Note that the **label_int** column in the **train_data** and **test_data** variables is assumed to be a binary encoding of the label column, where one class is represented by 0 and the other class is represented by 1. If this is not the case, you may need to perform additional preprocessing steps to encode the labels in the desired format.

Logistic regression

```
model = LogisticRegression()
model.fit(train_bow, train_data['label_int'])

# Predict on the test set
predicted1 = model.predict(test_bow)
# Evaluate the model
accuracy = (predicted1 == test_data['label_int']).mean()
print("Accuracy for logistic regression is", accuracy)
```

```
Accuracy for logistic regression is 0.8
```

Figure 5.3.7 depicts code for logistic regression.

This code snippet demonstrates how to use a logistic regression classifier to perform text classification on a dataset. Here's a breakdown of the code:

- First, the required libraries are imported.
- **CountVectorizer** from **sklearn.feature_extraction.text** and **LogisticRegression** from **sklearn.linear_model**.
- With the help of the train_test_split function, the dataset is then divided into training and test sets.
- The **train_data** and **test_data** variables are created by selecting the columns of interest (the text and label columns) from the respective sets.
- To represent the text as a bag of words, a CountVectorizer object is created. To fit the vocabulary and convert the text into a matrix of token counts, the fit_transform technique is used on the training set. To convert the text in the test set into the same format as the training set, the transform method is invoked.
- A **LogisticRegression** object is created to train the logistic regression model. The **fit** method is called on the bag of words matrix and the label column of the training set.
- The **predict** method is called on the bag of words matrix of the test set to make predictions on the test set.
- By contrasting the predicted labels with the actual labels from the test set, the accuracy of the model is determined.

SVM

```
model = SVC()
model.fit(train_bow, train_data['label_int'])

# Predict on the test set
predicted2 = model.predict(test_bow)

# Evaluate the model
accuracy = (predicted2 == test_data['label_int']).mean()
print("Accuracy for svc is ", accuracy)
```

```
Accuracy for svc is  0.8083333333333333
```

Figure 5.3.8 depicts the code for SVM.

SVC stands for Support Vector Classification. It is a classification algorithm that uses support vector machines to classify data. In layman's words, it looks for the best boundary (hyperplane) to divide the various classes of data.The margin, or the separation between the boundary and the nearest data points of each class, is maximized by selecting the boundary in this manner. SVC can handle both linear and nonlinear classification problems and can work with both binary and multi-class classification problems. The kernel trick, on which the algorithm is built, enables it to transform the input data into a higher dimensional space where it might be simpler to locate a dividing border.

With the help of this snippet of code, you can see how to train a support vector classifier (SVC) on a training set and assess its effectiveness on a test set. The code is broken down as follows:

- An SVC model is instantiated with default hyperparameters.
- The model is trained on the bag of words matrix and the label column of the training set using the **fit** method.
- The **predict** method is called on the bag of words matrix of the test set to make predictions on the test set.
- By contrasting the predicted labels with the actual labels from the test set, the accuracy of the model is determined.

Note that the default hyperparameters of the SVC may not be optimal for the

given dataset and task, and hyperparameter tuning may be necessary to obtain better performance. Additionally, SVCs can be computationally expensive to train on large datasets, while other models, such logistic regression or naive Bayes, might be quicker and adequate for smaller datasets.

5.4 SIMPLE AVERAGE ENSEMBLE

```
final_pred=(predicted1+predicted2+predicted3)/3.0
accuracy = (final_pred == test_data['label_int']).mean()
print("Accuracy for simple average ensembling is ", accuracy)
```

```
Accuracy for simple average ensembling is  0.6583333333333333
```

Figure 5.4.1 depicts the simple average ensemble.

This code snippet demonstrates how to perform a simple average ensemble of three models to make predictions on a test set. Here's a breakdown of the code:

- The predicted labels from the three models (**predicted1**, **predicted2**, and **predicted3**) are added together and divided by 3 to get the average predicted label for each example in the test set. Note that the assumption here is that the three models have equal importance and therefore their predictions should be weighted equally.
- By contrasting the final projected labels with the actual labels of the test set, the ensemble model's accuracy is determined.

Stacked ensemble

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

# Define the individual models
model1 = LogisticRegression()
model2 = SVC()
model3=MultinomialNB()
# Define the meta-model
meta_model = LogisticRegression()

# Define the stacked ensemble model
stacked_ensemble = StackingClassifier(estimators=[('lr', model1), ('svc', model2),('naive_bayes',model3)], final_estimator=meta_model)

# Fit the stacked ensemble model on the training data
stacked_ensemble.fit(train_bow,train_data['label_int'])

# Make predictions on the test data
y_pred = stacked_ensemble.predict(test_bow)
y_pred
```

Figure 5.4.2 depicts the stacked ensemble of ML models.

This code snippet demonstrates how to use the **StackingClassifier** from scikit-learn to create a stacked ensemble model for text classification. Here's a breakdown of the code:

- The necessary libraries are imported: **StackingClassifier** from **sklearn.ensemble**, **LogisticRegression** and **SVC** from **sklearn.linear_model** and **sklearn.svm** respectively.
- Three individual models are defined: **model1** is a logistic regression classifier, **model2** is a support vector classifier, and **model3** is a multinomial naive bayes classifier.
- A meta-model is defined as a logistic regression classifier. The predictions of the various models are combined using the meta-model.
- A stacked ensemble model is created using the **StackingClassifier** constructor. The individual models and the meta-model are passed as arguments to the **estimators** and **final_estimator** parameters, respectively.
- The stacked ensemble model is fit on the bag of words matrix and the label column of the training set using the **fit** method.
- The **predict** method is called on the bag of words matrix of the test set to make predictions on the test set.

```
accuracy = (y_pred == test_data['label_int']).mean()
print("Accuracy for stacked ensembling models is", accuracy)
```

```
Accuracy for stacked ensembling models is 0.8166666666666667
```

Figure 5.4.3 depicts the accuracy for ensemble models.

Simple average ensemble for DL algorithms

```
final_pred_prob=(y_pred_probs1+y_pred_probs2)/2
```

```
pred=[]
for res in final_pred_prob:
  pred.append(np.argmax(res))
```

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test1,pred)
```

```
0.8666666666666667
```

Figure 5.4.4 depicts the simple average ensemble.

This code snippet appears to be using probability estimates generated by two different models (**y_pred_probs1** and **y_pred_probs2**) to make predictions on a test set. Specifically, it calculates the simple average of the probability estimates for each sample, and then takes the argmax of each averaged probability vector to obtain the predicted class labels. Finally, it calculates the accuracy of the predicted labels against the true labels of the test set using the

accuracy_score function from **sklearn.metrics**.
Note that using probability estimates for ensemble can be more informative than using hard class labels, as it provides more information about the confidence or uncertainty of the models. However, the appropriate method for ensemble probability estimates can depend on the specific task and the characteristics of the models being used, and other methods such as weighted averaging or stacking may be more effective than simple averaging in some cases. Additionally, other metrics like precision, recall, or the AUC-ROC may be helpful in assessing the performance of the ensemble if the accuracy score is insufficient on its own.

CHAPTER 6

RESULTS DISCUSSION

The model was validated by using an in-house dataset that consisted of 1040 different sentences, with 520 of them being idiomatic sentences and the remaining 520 being literal sentences [14]. The three parameters utilized to gauge performance were precision, recall, and accuracy.

6.1 Evaluation metrics

Evaluation metrics for Stacked ensemble model

Labels	Precision	Recall	F1-score	Accuracy
Idiom	0.79	0.91	0.85	0.82
Literal	0.86	0.69	0.77	0.82

The evaluation metrics for the stacked ensemble model are shown in Table 6.1.1.

Evaluation metrics for XLM-RoBERTa

Labels	Precision	Recall	F1-score	Accuracy
Idiom	0.75	0.98	0.85	0.84
Literal	0.98	0.72	0.83	0.84

The evaluation metrics for XLM-RoBERTa are displayed in Table 6.1.2.

Evaluation metrics for m-BERT

Labels	Precision	Recall	F1-score	Accuracy
Idiom	0.74	0.98	0.85	0.83
Literal	0.98	0.70	0.82	0.83

Evaluation metrics for m-BERT are displayed in Table 6.1.3.

Evaluation metrics for simple average ensemble model

Labels	Precision	Recall	F1-score	Accuracy
Idiom	0.77	0.98	0.87	0.86
Literal	0.98	0.75	0.85	0.86

Evaluation metrics for simple average ensemble are displayed in Table 6.1.4.

Metrics for recall and precision can be found by calculating three separate values, including true positive (TP), false positive (FP), and false negative (FN). In the portion of this article that follows, you'll discover the equations for both precision and recall.

Precision = T P / (T P + F P)

where the variable FP stands for the number of literals that were mislabeled as idioms and the variable TP stands for the number of idioms that were correctly categorized as idioms.

Recall = T P / (T P + F N)

The number of idioms that their database wrongly displays as literals is defined by FN. The confusion matrix for the XLM-RoBERTa and M-BERT models, which consider precision and recall, is shown below.

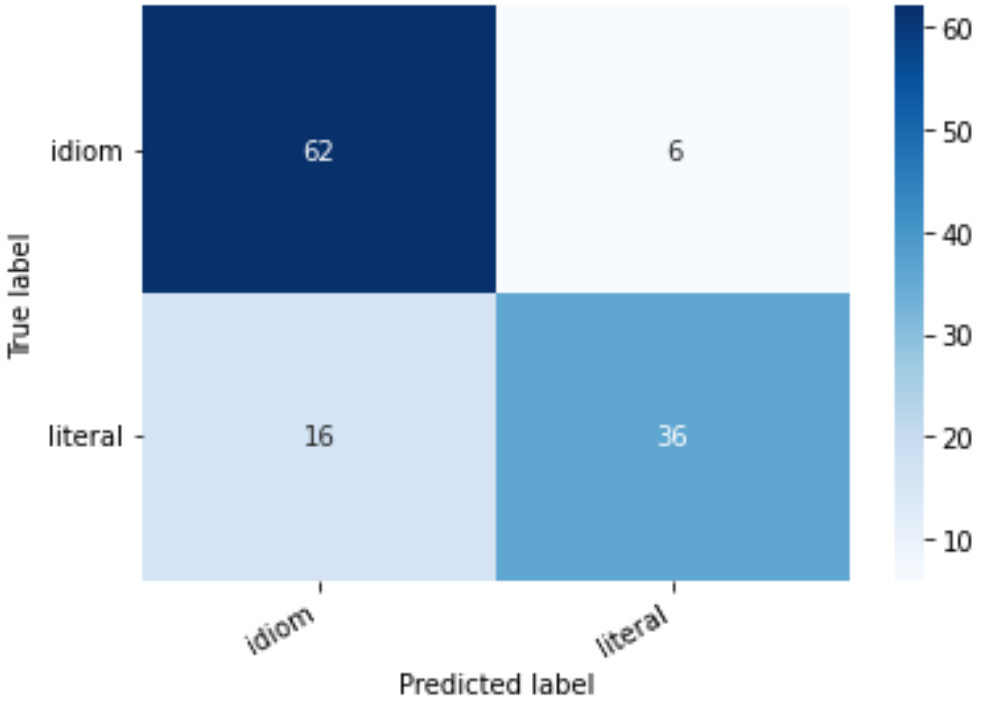


Figure 6.1.5 depicts confusion matrix for Stacked ensemble model.

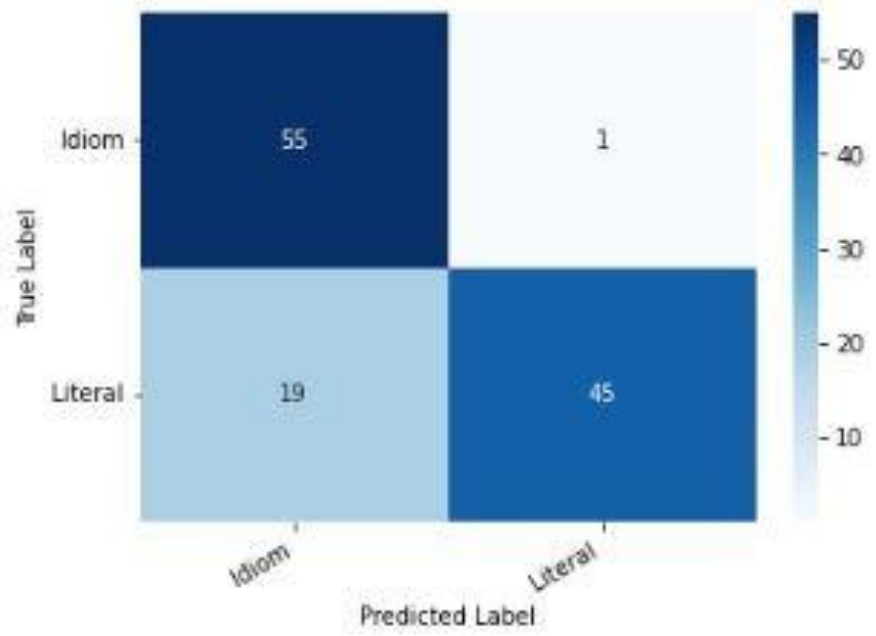


Figure 6.1.6 depicts Confusion matrix for XLM-RoBERTa

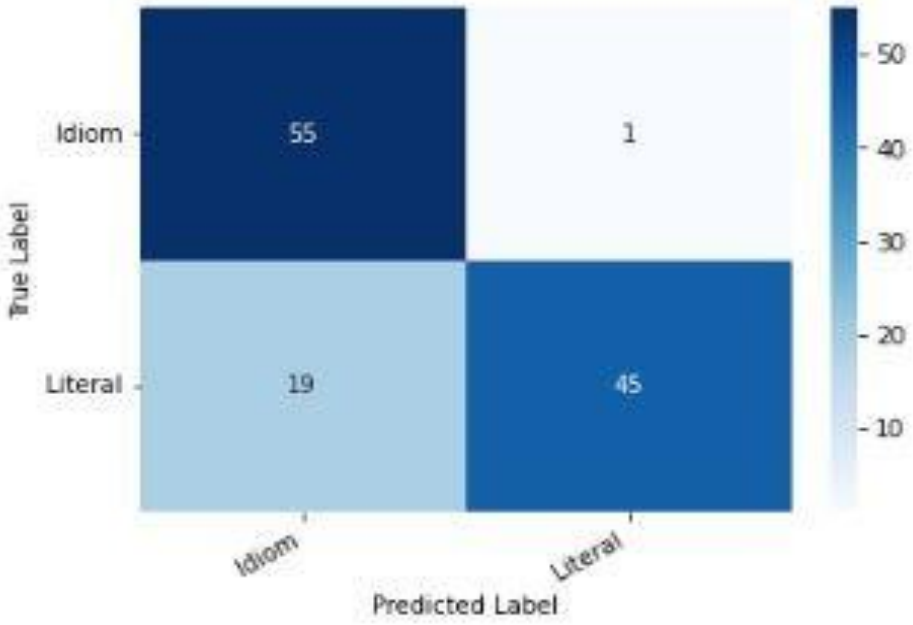


Figure 6.1.7 depicts Confusion matrix for M-BERT

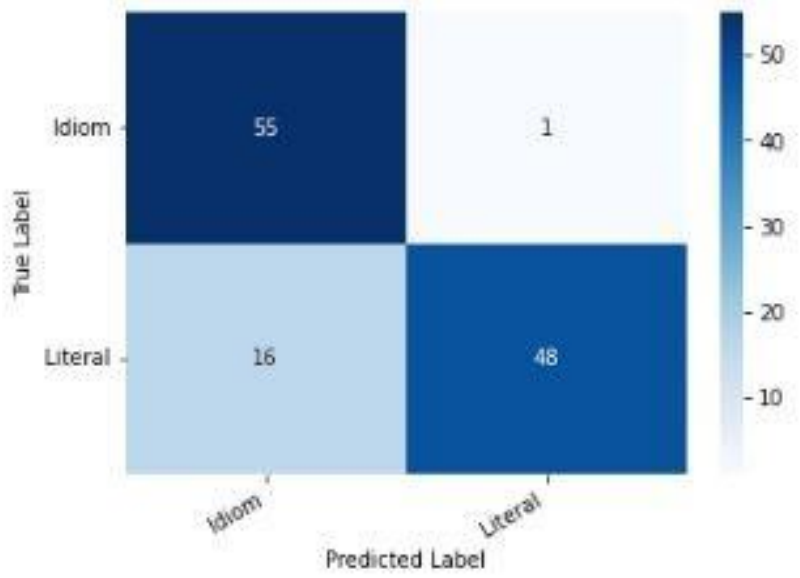


Figure 6.1.8 depicts the confusion matrix for the Simple Average Ensemble Model.

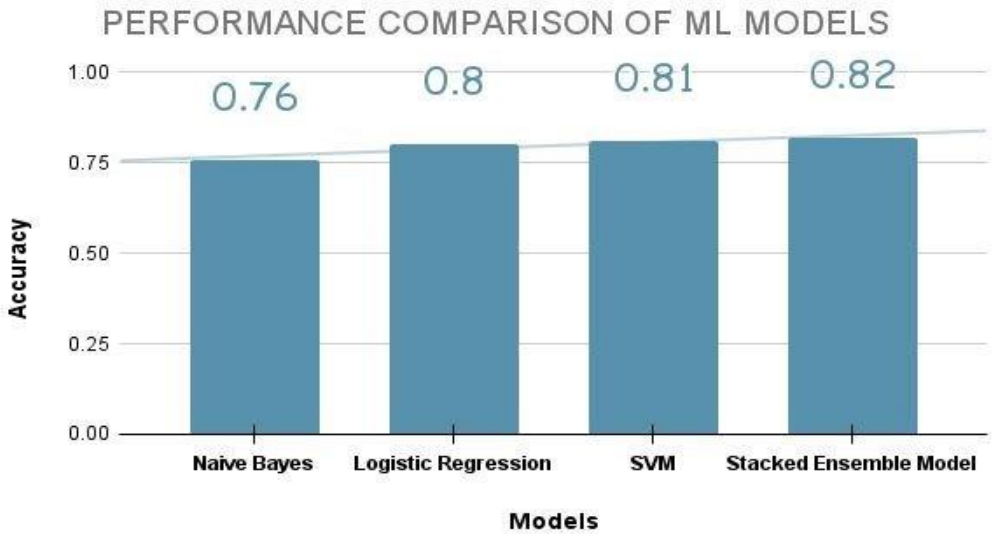


Figure 6.1.9 compares the accuracy of machine learning models with that of a stacked ensemble model.

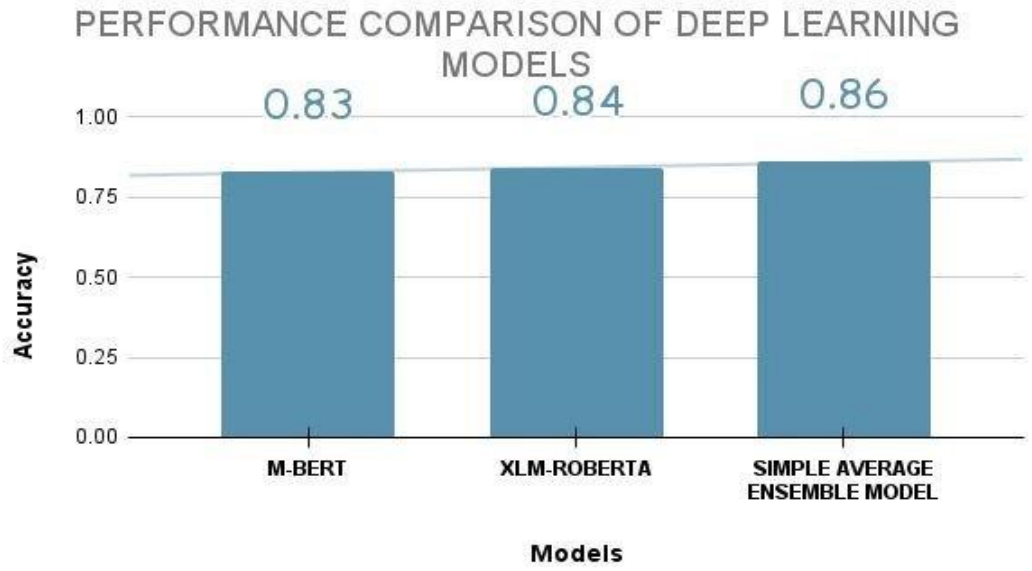


Figure 6.1.10 compares the accuracy of the DL and its stacked ensemble model.

MODEL	PRECISION	RECALL	F1-SCORE	ACCURACY
Stacked ensemble using ML	0.86	0.8	0.8	0.82
XLM-RoBERTa	0.87	0.85	0.84	0.84
m-BERT	0.86	0.84	0.83	0.83
Simple average ensemble using DL	0.88	0.87	0.86	0.86

Table 6.1.11 compares the precision, recall, accuracy, and f1-score of DL and Ensemble models.

Table 6.1.11 reveals that the simple average ensemble model outperformed pre-trained deep learning algorithms and stacked ensemble using ML in terms of precision, recall, f1-score, and accuracy. According to Fig 6.1.9, the stacked ensemble model outperforms well compared to Naïve Bayes, Logistic regression and SVM with accuracies of 0.82,0.76,0.8 and 0.81 respectively. From Figure 6.1.10 the simple average ensemble model generates superior results than XLM-RoBERTa and m-BERT. Based on results the stacked ensemble model is predicting nearer to m-BERT. Due to the 160 GB of text

used to pre-train the basic RoBERTa model, 16 GB of which came from the Books Corpus and the English Wikipedia that is used in BERT, the additional data included the Common Crawl News dataset, which has 63 million items and 76 gigabytes, the Web text corpus, and the Common Crawl Stories dataset (31 GB), which made the simple average ensemble model perform well. In addition to this, the XLM-RoBERTa model makes use of dynamic masking, and the m-BERT model makes use of static masking. Both contribute to the robustness of the simple average ensemble model.

CHAPTER 7

CONCLUSION

The m-BERT and XLM-RoBERTa baseline models are combined to create the ensemble prediction model for classification of literal and idiomatic sentences that is suggested in this study. The accuracy is now 2% higher than with the baseline models. Numerous NLP applications, including chatbots and machine translation, can be implemented using the recommended approach. This model can be used to classify a statement as literal or idiomatic and translate it accordingly in machine translation. This model will increase the value of the NLP applications by being incorporated into them. In order to train this model over a wider range of literal and idiomatic statements, we must expand the dataset in order to make it more predictive.

REFERENCES

[1] Feldman A, Peng J (2013) Automatic detection of idiomatic clauses. In: International conference on intelligent text processing and computational linguistics. Springer, Berlin, Heidelberg.

[2] Fazly A, Cook P, Stevenson S (2009) Unsupervised type and token identification of idiomatic expressions. Comput Linguist 35(1):61–103.

[3] Peng J, Feldman A, Vylomova E (2018) Classifying idiomatic and literal expressions using topic models and intensity of emotions.

[4] Singh G et al (2019) Comparison between multinomial and Bernoulli Naïve Bayes for text classification. In 2019 International conference on automation, computational and technology management (ICACTM). IEEE

[5] Bahassine S et al (2020) Feature selection using an improved Chi-square for Arabic text classification. J King Saud Univ-Comput Inf Sci 32(2):225–231

[6] Birke, J., Sarkar, A.: A clustering approach for nearly unsupervised recognition of nonliteral language. In: 11th Conference of the European Chapter of the Association for Computational Linguistics (2006)

[7] Rother, K., Rettberg, A.: Ulmfit at germeval-2018: A Deep Neural Language Model for the Classification of Hate Speech in German Tweets. pp. 113–119 (2018)

[8] Howard, J., Ruder, S.: Universal language model fine-tuning for text classification (2018). arXiv preprint arXiv: 1801.06146.

[9] Peng, J., Feldman, A., Street, L.: Computing linear discriminants for idiomatic sentence detection. Res. Comput. Sci. Spec. Iss. Nat. Lang. Proces. Appl. 46, 17–28 (2010).

[10] Xiaoyu Luo, June 2021 Efficient English text classification using selected Machine Learning Techniques.

[11] Ashwin Sanjay Neogi, Kirti Anilkumar Garg, Ram Krishn Mishra, Yogesh K Dwivedi Sentiment analysis and classification of Indian farmers' protest using twitter data (2019).

[12] Ningfeng Sun and Chengye Du News Text Classification Method and Simulation Based on the Hybrid Deep Learning Model (2021).

[13] Briskilal, J., & Subalalitha, C. N. (2022). An ensemble model for classifying idioms and literal texts using BERT and RoBERTa. Information Processing & Management, 59(1), 102756.

[14] Briskilal, J., & Subalalitha, C. N. (2021). Classification of Idioms and Literals Using Support Vector Machine and Naïve Bayes Classifier. In Machine Vision and Augmented Intelligence—Theory and Applications (pp. 515-524). Springer, Singapore.

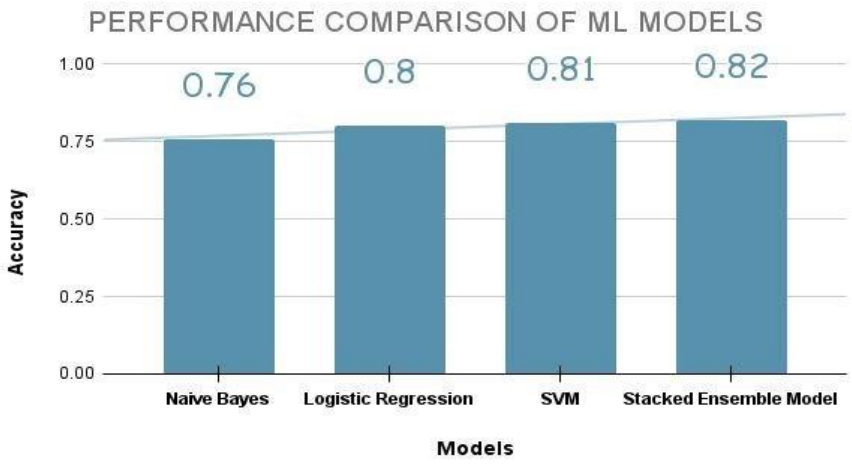
[15] Briskilal, J., & Subalalitha, C. N. (2022). Classification of Idiomatic Sentences Using AWD-LSTM. In Expert Clouds and Applications (pp. 113-124). Springer, Singapore.

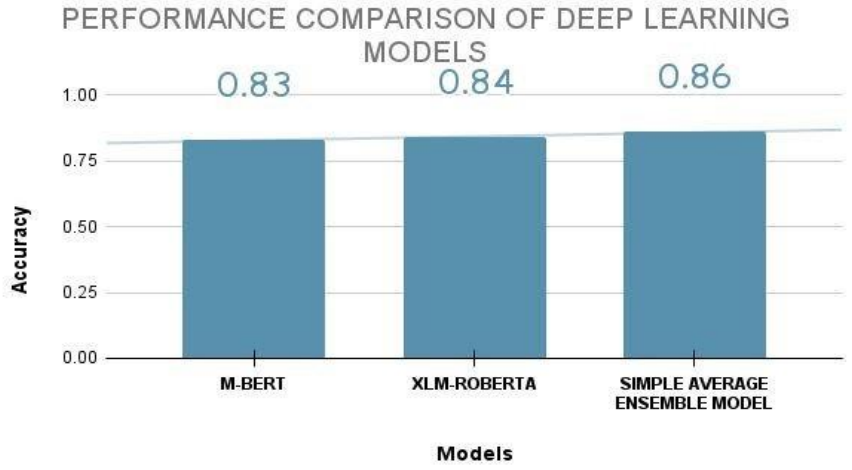
[16] Medhat, W., Hassan, A., & Korashy, H. (2014). Sentiment analysis algorithms and applications: A survey. *Ain Shams engineering journal*, *5*(4), 1093-1113.

[17] Sahayak, V., Shete, V., & Pathan, A. (2015). Sentiment analysis on twitter data. *International Journal of Innovative Research in Advanced Engineering (IJIRAE)*, *2*(1), 178-183.

[18] Kharde, V., & Sonawane, P. (2016). Sentiment analysis of twitter data: a survey of techniques. *arXiv preprint arXiv:1601.06971*.

APPENDIX 1





The stacked ensemble model of ML models outperforms well compared to Naïve Bayes, Logistic regression and SVM with accuracies of 0.82,0.76,0.8 and 0.81 respectively. But the stacked ensemble model is unable to beat the deep learning models. The baseline models of m-BERT and XLM-RoBERTa are combined in the ensemble prediction model for categorization of idiomatic and literal phrases that is suggested in this research. Accuracy is 2% higher now than it was with the baseline models. The suggested methodology can be used to build a range of NLP applications, including chatbots and machine translation. By classifying a sentence as literal or idiomatic, this model can be used to translate it using machine translation in line with that classification. This model will increase the value of the NLP applications by being incorporated into those programmes. In order to train this model across more literal and idiomatic utterances and improve its predictive abilities, the dataset must be expanded.

PLAGIARISM REPORT

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY **(Deemed to be University u/ s 3 of UGC Act, 1956)**		
Office of Controller of Examinations		
REPORT FOR PLAGIARISM CHECK ON THE DISSERTATION/PROJECT REPORTS FOR UG/PG PROGRAMMES **(To be attached in the dissertation/ project report)**		
1	Name of the Candidate **(IN BLOCK LETTERS)**	CH CHAITANYA
2	Address of the Candidate	Andhra Pradesh, Prakasam District, Tangutur Mandal, Tangutur, 523274, Near Union Bank Branch-2 Mobile number: 8688771151
3	Registration Number	RA1911003010992
4	Date of Birth	09 August 2002
5	Department	Computer Science and Engineering
6	Faculty	Engineering and Technology, School of Computing
7	Title of the Dissertation/Project	Classification of Telugu Idiomatic Sentences Using Ensemble Models
8	Whether the above project /dissertation is done by	Group: a) If the project/ dissertation is done in group, then how many students together completed the project: 2 (Two) b) Mention the Name & Register number of other candidates: CH V M SAI PRANEETH, RA1911003010983
9	Name and address of the Supervisor / Guide	Ms.J.Briskilal Assistant Professor Department of Computer Science and Engineering SRM Institute of Science and Technology Kattankulatur - 603203. **Mail ID:** briskilj@srmist.edu.in **Mobile Number:** 9840574851

10	Name and address of Co-Supervisor / Co- Guide (if any)	NIL **Mail ID: Mobile Number:**
11	Software Used	Turnitin
12	Date of Verification	10 - May - 2023
13	**Plagiarism Details: (to attach the final report from the software)**	

Chapter	Title of the Chapter	Percentage of similarity index (including self citation)	Percentage of similarity index (Excluding self-citation)	% of plagiarism after excluding Quotes, Bibliography, etc.,
1	INTRODUCTION	1%	1%	1%
2	LITERATURE SURVEY	3%	3%	3%
3	SYSTEM ARCHITECTURE	2%	2%	2%
4	METHODOLOGY	1%	1%	1%
5	RESULTS DISCUSSION	2%	2%	2%
6	CONCLUSION	0%	0%	0%
Appendices		0%	0%	0%

I / We declare that the above information has been verified and found true to the best of my / our knowledge.

ch. chaitanya CH CHAITANYA **Signature of the Candidate**	J. B **Name & Signature of the Staff (Who uses the plagiarism check software)**
J. B **Name & Signature of the Supervisor/Guide**	**Name & Signature of the Co-Supervisor/Co-Guide**

M. Pushpalatha

Name & Signature of the HOD

chaitanya

ORIGINALITY REPORT

9%	3%	5%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	J Briskilal, C.N. Subalalitha. "An ensemble model for classifying idioms and literal texts using BERT and RoBERTa", Information Processing & Management, 2022 Publication	2%
2	Submitted to Asia Pacific University College of Technology and Innovation (UCTI) Student Paper	<1%
3	Submitted to INTI Universal Holdings SDM BHD Student Paper	<1%
4	"Chinese Computational Linguistics", Springer Science and Business Media LLC, 2019 Publication	<1%
5	Submitted to Napier University Student Paper	<1%
6	"Intelligent Information and Database Systems", Springer Science and Business Media LLC, 2021 Publication	<1%

PAPER PUBLICATION

Our paper titled "Classification of Telugu Idiomatic Sentences Using Ensemble Models" was submitted to IEEE ICICT 2023 Conference. and our paper got accepted and the conference is completed.

IEEE
XPLORE COMPLIANT ISBN
979-8-3503-9849-6

ICICT

Presentation Certificate

This is to certify that

J Briskilal

have successfully presented the paper entitled

An Ensemble Method to Classify Telugu Idiomatic Sentences using Deep Learning Models

at the
6th International Conference on Inventive Computation Technologies (ICICT 2023)
organized by Tribhuvan University, Pulchowk Campus, Lalitpur, Nepal
held on 26-28, April 2023.

Session Chair

Organizing Secretary
Dr. Joy Iong Zong Chen

Conference Chair
Prof. Dr. Subarna Shakya

IEEE
XPLORE COMPLIANT ISBN
979-8-3503-9849-6

ICICT

Presentation Certificate

This is to certify that

Ch. V M Sai Praneeth

have successfully presented the paper entitled

An Ensemble Method to Classify Telugu Idiomatic Sentences using Deep Learning Models

at the
6th International Conference on Inventive Computation Technologies (ICICT 2023)
organized by Tribhuvan University, Pulchowk Campus, Lalitpur, Nepal
held on 26-28, April 2023.

Session Chair

Organizing Secretary
Dr. Joy Iong Zong Chen

Conference Chair
Prof. Dr. Subarna Shakya

IEEE
XPLORE COMPLIANT ISBN
979-8-3503-9849-6

ICICT

Presentation Certificate

This is to certify that

Ch. Chaitanya

have successfully presented the paper entitled

An Ensemble Method to Classify Telugu Idiomatic Sentences using Deep Learning Models

at the
6th International Conference on Inventive Computation Technologies (ICICT 2023)
organized by Tribhuvan University, Pulchowk Campus, Lalitpur, Nepal
held on 26-28, April 2023.

Session Chair

Organizing Secretary
Dr. Joy Iong Zong Chen

Conference Chair
Prof. Dr. Subarna Shakya