INTEL UNNATI INDUSTRIAL TRAINING PROGRAM-2024

PROBLEM STATEMENT:

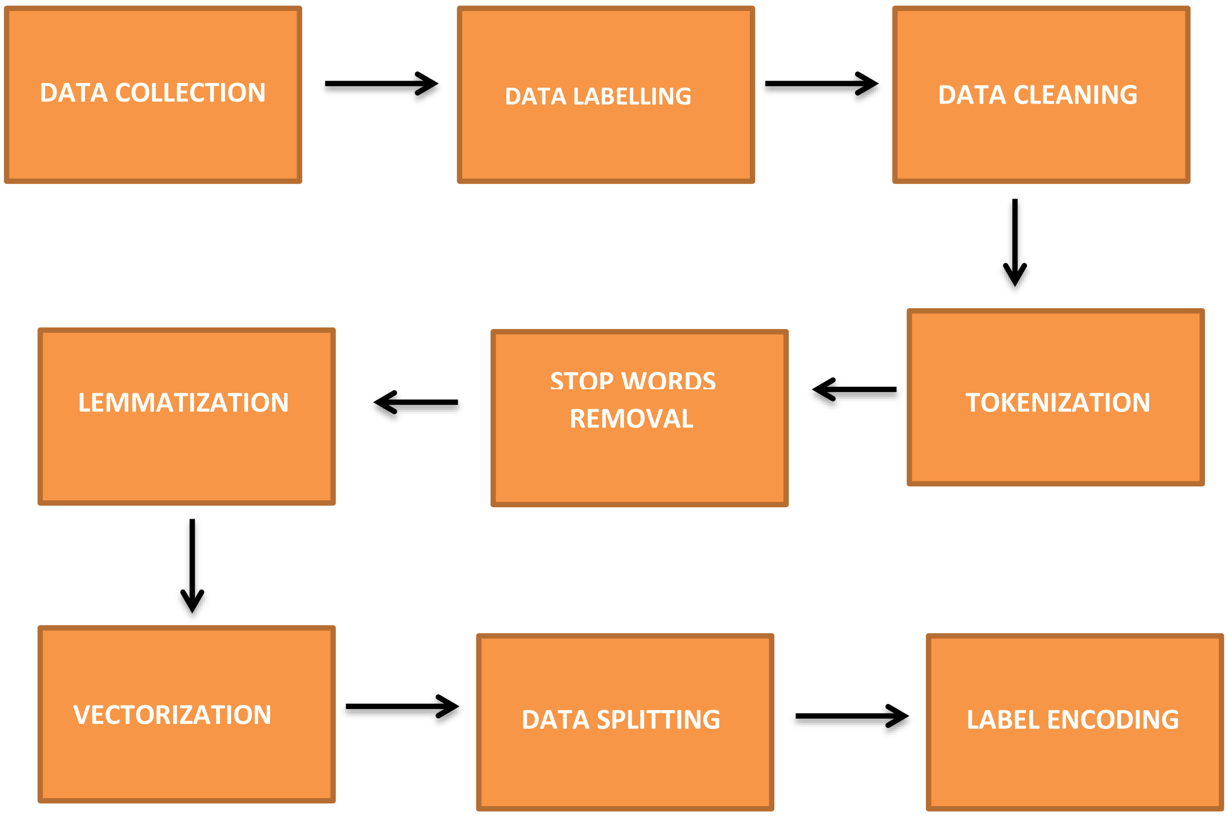
Intel products sentiment analysis from online reviews.

OBJECTIVE:

The objective of this project is to analyze online reviews of Intel products to understand customer sentiments and opinions. By employing natural language processing (techniques and sentiment analysis, this project aims to categorize the reviews into various sentiment categories (positive, negative, future expectations and competition sentiments) and extract insights related to customer satisfaction, common issues, and expectations.

DATA PRE PROCESSING:

STEPS:



DATA COLLECTION:

The reviews for this analysis were collected from two major e-commerce platforms: Amazon and Flipkart, ensuring a comprehensive and up-to-date dataset. These platforms were selected due to their extensive customer base and the high volume of detailed user reviews they provide, making them ideal for conducting a thorough sentiment analysis.

DATA LABELLING:

The dataset is now fully prepared, with each review meticulously analyzed and labeled for sentiment. The sentiment labeling was done manually to ensure accuracy and reliability.

DATA CLEANING:

Handling missing values involves identifying the gaps using functions like isnull().sum(), then filling them with mean, median, or mode for numerical data, or the most common category for categorical data.

TOKENIZATION:

Splitting the collected reviews into smaller units, such as words or phrases, to facilitate analysis. This step is essential for transforming raw text into a structured format suitable for machine learning models in sentiment analysis.

STOP WORDS REMOVAL:

Word clouds were used to visualize the most frequent words in reviews across different sentiments. By removing stop words such as 'and' and 'the', significant themes and key terms in positive, negative, competition sentiments, and future expectations. These word clouds showcase the distinctive traits and standout features within each type of review, highlighting important themes and key terms post stop words removal.

STEMMING VS LEMMETIZATION:

When processing text data, two common techniques for reducing words to their base or root form are lemmatization and stemming. While stemming involves cutting off prefixes or suffixes to achieve this, here lemmatization is utilised to reduce the number of words in the 'ReviewContent' column by reducing them to their most basic form. This aids in ensuring that word variations (such as plurals or different tenses) are handled consistently, which is helpful for tasks like figuring out subjects.

VECTORIZATION:

TfidfVectorizer transforms text data into numerical features using TF-IDF scores. TF-IDF measures the importance of a word in a document based on its frequency in that document and rarity across all documents in the dataset.

DATA SPLITTING:

The dataset was split into training and testing sets to ensure high accuracy in our model. We allocated 70% of the data to the training set (X_train, y_train) for building the model, and 30% to the testing set (X_test, y_test) for evaluating its performance.

LABEL ENCODING:

Label Encoding is used here to convert categorical variables (columns with text values) into numerical labels. Machine learning algorithms typically require numerical input, so categorical data needs to be transformed into a format that can be processed by these algorithms. By encoding 'Category', 'Generation', 'Sentiment', and 'Country' into numerical values, the data becomes suitable for use in training machine learning models.

DATA VISUALIZATION:

Visualize the distribution of sentiment categories in the dataset. By plotting the count of each sentiment, it provides a quick overview of how the sentiments are distributed, whether some sentiments are more prevalent than others, and can help in understanding the balance or imbalance in the dataset.

First converts the 'Date' column to datetime format to simplify handling of dates. It then uses Seaborn's line plot (lineplot()) to visualize how sentiment values change over time, uncovering patterns in how sentiments vary across different dates.

Visualizes the top countries by review count using a bar plot, making it easier to identify and compare which countries contribute the most reviews in the dataset.

Histogram is created using Seaborn (histplot) to visualize how ratings are distributed across the dataset. It helps understand the frequency and pattern of ratings, providing insights into the overall distribution of rating values in the data.

CLASSIFICATION ALGORITHMS:

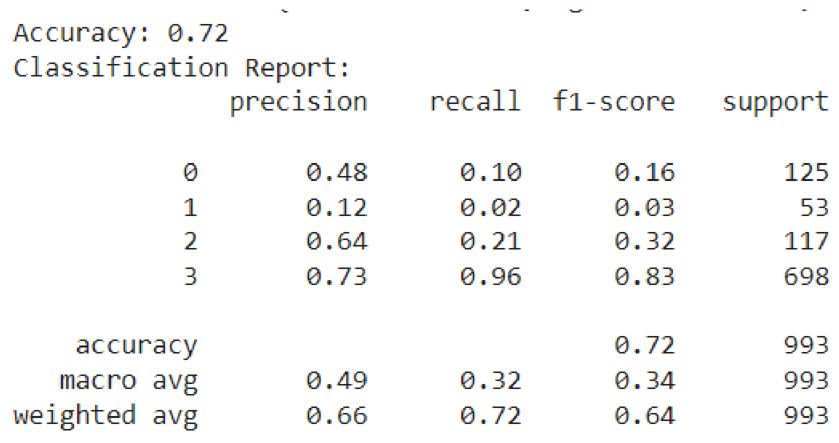
SUPPORT VECTOR CLASSIFIER(SVC):

The SVM classifier with a linear kernel is chosen because it's highly effective in tasks like sentiment analysis, where it excels in understanding the intricate connections between text features and sentiment categories.

Trained on labeled data (X_train, y_train), the SVM learns from the text patterns to accurately predict sentiment labels. The accuracy score (computed with accuracy_score) then evaluates how well the model's predictions match the actual sentiment labels in y_test.

Following the SVM algorithm execution, the detailed classification report is generated, providing essential metrics like precision, recall, F1-score.

The classification report SVC Algorithm:


```
Accuracy: 0.72
Classification Report:
              precision    recall  f1-score   support

           0       0.48      0.10      0.16       125
           1       0.12      0.02      0.03        53
           2       0.64      0.21      0.32       117
           3       0.73      0.96      0.83       698

    accuracy                           0.72       993
   macro avg       0.49      0.32      0.34       993
weighted avg       0.66      0.72      0.64       993
```

RANDOM FOREST:

- Sets up the Random Forest model, ensuring it can handle imbalanced datasets by adjusting the class weights.
- PARAMETER GRID:

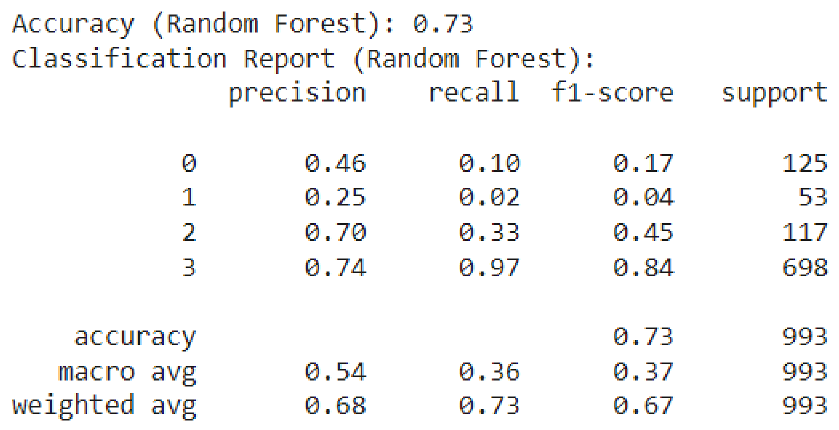
A parameter grid is a set of predefined parameter values used to optimize a machine learning model. It helps systematically explore various combinations of parameters to find the best settings, enhancing the model's performance. The parameter grid is used in the Random Forest model to systematically find the optimal hyperparameters.

- GRID SEARCH CV:

This process involves performing cross-validation to test different combinations of the specified parameters. The goal is to identify the combination that gives the best performance.

- Using the best parameters found from the grid search, the Random Forest model is trained on the provided training data. During this phase, the model learns patterns and relationships within the data to make accurate predictions.
- After training, the model is used to predict sentiment labels for the test data. This step shows how well the model has learned from the training phase
- The model's performance is measured using accuracy and a detailed classification report.

The classification report for Random Forest Algorithm:

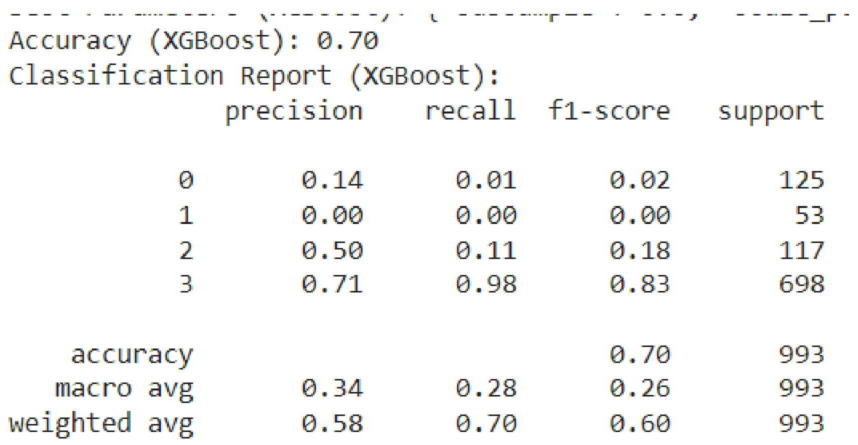


```
Accuracy (Random Forest): 0.73
Classification Report (Random Forest):
              precision    recall  f1-score   support

           0       0.46      0.10      0.17       125
           1       0.25      0.02      0.04        53
           2       0.70      0.33      0.45       117
           3       0.74      0.97      0.84       698

    accuracy                           0.73       993
   macro avg       0.54      0.36      0.37       993
weighted avg       0.68      0.73      0.67       993
```

XGBOOST:

- A grid of hyperparameters is defined for the XGBoost model, specifying settings like the number of trees, learning rate, and tree depth.
- The XGBoost model is initialized with specific parameters to ensure stable and efficient training.
- Through cross-validated hyperparameter tuning, we optimize the model's settings to achieve the highest possible accuracy.
- The best-performing XGBoost model configuration is applied to train the transformed training data.
- The model predicts sentiment labels on the test set and evaluates its performance using accuracy metrics.

The classification report for XGBoost Algorithm is:



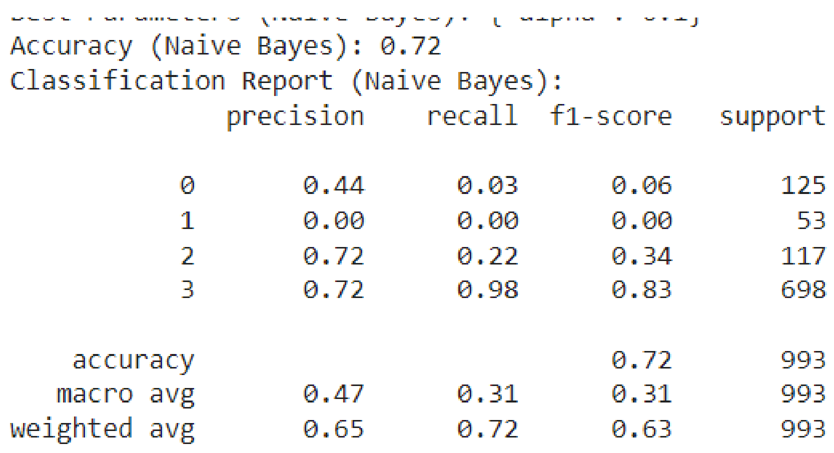
```
Accuracy (XGBoost): 0.70
Classification Report (XGBoost):
              precision    recall  f1-score   support

           0       0.14      0.01      0.02       125
           1       0.00      0.00      0.00        53
           2       0.50      0.11      0.18       117
           3       0.71      0.98      0.83       698

    accuracy                           0.70       993
   macro avg       0.34      0.28      0.26       993
weighted avg       0.58      0.70      0.60       993
```

NAIVE BAYES:

- The data is splitted into train and test sets.
- TfidfVectorizer converts text data into numerical TF-IDF features.
- It limits the maximum number of features to 5000 to manage memory usage and potentially improve model performance.
- A parameter grid is defined specifically for the alpha parameter of the Multinomial Naive Bayes classifier.
- MultinomialNB is initialized as the Naive Bayes classifier, which is suitable for classification tasks with discrete features like TF-IDF.

- RandomizedSearchCV is used to search for the best alpha value through randomized search, optimizing the Multinomial Naive Bayes model.
- RandomizedSearchCV is fitted on the TF-IDF transformed training data (X_train_tfidf, y_train) to find the best parameters (alpha) for the Naive Bayes model.
- After fitting, the best parameters and the best Naive Bayes model are identified from the search results.
- The model predicts sentiment labels on the test set and evaluates its performance using accuracy metrics.

The classification report for Naïve Bayes Algorithm is:



```
Accuracy (Naive Bayes): 0.72
Classification Report (Naive Bayes):
              precision    recall  f1-score   support

           0       0.44      0.03      0.06       125
           1       0.00      0.00      0.00        53
           2       0.72      0.22      0.34       117
           3       0.72      0.98      0.83       698

    accuracy                           0.72       993
   macro avg       0.47      0.31      0.31       993
weighted avg       0.65      0.72      0.63       993
```

Out of the four algorithms evaluated, **Random Forest** performed the best with an accuracy of **0.73**.

ENSEMBLING TECHNIQUES:

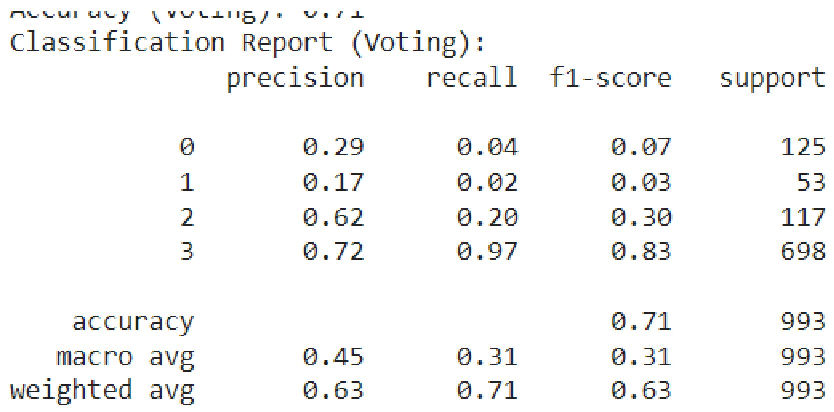
STACKING:

- In this project, we're using a combination of machine learning models to analyze sentiment from online reviews of Intel products. We've set up a Stacking Classifier, which is like an ensemble of models working together. First, we define two base models: a Random Forest and a Gradient Boosting Classifier. These models each have their strengths in capturing patterns and making predictions from the review data.

- Next, we add a meta-model on top of these base models. This meta-model, Logistic Regression in this case, combines the predictions from the base models to make a final decision. It learns to weigh the predictions from each base model based on their performance during training.
- To ensure the reliability of our models, we use Stratified K-Fold Cross Validation. This technique splits the data into folds, trains the models on different subsets, and evaluates them multiple times to get a robust measure of accuracy.
- During cross-validation, we compute accuracy scores to see how well our Stacking Classifier performs across different subsets of the data. After training, we apply the model to predict sentiment on our test dataset and evaluate its performance using metrics like accuracy and a detailed classification report.
- Overall, this approach allows us to leverage the strengths of multiple models to improve the accuracy of sentiment analysis, helping Intel gain valuable insights from customer reviews.

VOTING:

- The data is preprocessed and split into training and testing sets.
- TF-IDF Vectorization is applied to convert text data into numerical features.
- Class weights are computed to handle imbalanced classes in the training data.
- Best hyperparameters for a Random Forest Classifier are obtained from GridSearchCV, including class weights.
- Base models (Random Forest and Gradient Boosting) are defined for the Voting Classifier.
- A soft voting ensemble method is used to combine predictions from base models.
- Stratified K-Fold Cross Validation with 10 folds ensures robust evaluation of model performance.
- Cross-validation scores and mean accuracy are computed and printed.
- The voting model is trained on the training data and evaluated on the test data using accuracy and a classification report.



```
Accuracy (Voting): 0.71
Classification Report (Voting):
              precision    recall  f1-score   support

           0       0.29      0.04      0.07       125
           1       0.17      0.02      0.03        53
           2       0.62      0.20      0.30       117
           3       0.72      0.97      0.83       698

    accuracy                           0.71       993
   macro avg       0.45      0.31      0.31       993
weighted avg       0.63      0.71      0.63       993
```

UNSUPERVISED:

CLUSTERING:

- Performs the Elbow Method to determine the optimal number of clusters for KMeans clustering.
- First, it defines a range of cluster numbers to test, from 1 to 9 and initializes an empty list to store the Sum of Squared Errors (SSE).
- Then, it iterates through each value in this range, initializing a KMeans model with the specified number of clusters using k-means++ for initialization, setting a random seed for reproducibility (random_state=38), and specifying the maximum number of iterations (max_iter=100).
- For each model, it fits the model to the TF-IDF vectors and appends the model's inertia (SSE) to the sse list. Finally, the elbow graph is plotted.

ASSIGNING CLUSTER LABELS:

- The number of clusters (true_k) is set to 4.
- A KMeans model is initialized with the specified number of clusters (n_clusters=true_k), using the k-means++ method for initialization, setting a random seed (random_state=48) for reproducibility, and specifying the maximum number of iterations (max_iter=2000).
- The model is fitted to the TF-IDF vectors, and cluster labels are predicted for each data point. These labels are stored in y_predicted.
- The predicted cluster labels are added as a new column ("cluster") in the DataFrame (df).

- The value counts of the "cluster" column are calculated to determine the number of data points in each cluster.

IDENTIFYING CLUSTER HOTWORDS IN KMEANS CLUSTERING RESULTS:

- Analyzing and printing the top "hotwords" (most significant terms) for each cluster in a KMeans clustering model.
- After fitting the model to TF-IDF vectors, it calculates and sorts the cluster centers to identify the most important features for each cluster.
- The model.cluster_centers_ attribute provides the coordinates of the cluster centers in the feature space, which are sorted using argsort() function to determine the most relevant terms.
- Using vectorizer.get_feature_names_out(), it retrieves the feature names (terms) from the TF-IDF vectorizer.
- It then iterates through each cluster to print the cluster number and the top 20 hotwords associated with that cluster.

PREPARING TEXT DATA WITH BIGRAMS AND TRIGRAMS FOR LDA:

- This process prepares text data for topic modeling using Latent Dirichlet Allocation (LDA).
- It starts by identifying common bigrams (two-word phrases) and trigrams (three-word phrases) in the tokenized text data, ensuring phrases appear at least five times and setting a sensitivity threshold to 100.
- Efficient versions of these models, called Phrasers, are then created to transform the text data into bigrams and trigrams.
- Two functions are defined: one to convert the tokenized texts into bigrams and another to convert them into trigrams. These functions are then applied to the data, resulting in a list of tokenized reviews with bigrams and another list with both bigrams and trigrams.
- This enhancement captures multi-word expressions, providing more meaningful context for the subsequent LDA modeling, thus improving the quality of the generated topics.

REMOVING LOW-VALUE WORDS AND GENERATING A TF-IDF MODEL:

- This process aims to clean text data by removing frequent and less informative words from bigrams and trigrams before creating a TF-IDF (Term Frequency-Inverse Document Frequency) model.
- First, a dictionary (id2word) is made from the bigrams and trigrams.
- The text data is then turned into a bag-of-words (BoW) format using this dictionary.
- Next, a TF-IDF model is created from the BoW corpus. The code goes through each document to find and remove words with low TF-IDF values (below 0.03) and words missing from the TF-IDF model. For each document, it lists low-value words and missing words, adding these to a removal list.
- Finally, it updates each document by filtering out the identified words. This process improves the quality of the text data, making it more suitable for topic modeling or other text analysis tasks.

EVALUATING TOPIC COHERENCE IN LDA MODELS:

- This function helps determine the best number of topics for an LDA topic model by testing different models with varying topic counts. It builds LDA models with different numbers of topics, calculates coherence scores for each, and then returns both the models and their scores.
- Coherence scores measure how understandable and meaningful the topics are, guiding you to choose the optimal number of topics for clear and interpretable results.

DETERMINING THE OPTIMAL NUMBER OF TOPICS FOR TOPIC MODELING:

- Plotting a graph to help determine the best number of topics for a topic modeling task.
- It uses the coherence score, which measures how interpretable and distinct topics are from each other. The limit, start, and step variables control the range of numbers of topics to test.
- It iterates over these numbers and plots the coherence values against them. The graph visualizes how coherence scores change with

different numbers of topics, helping to identify the optimal number where topics are most coherent.

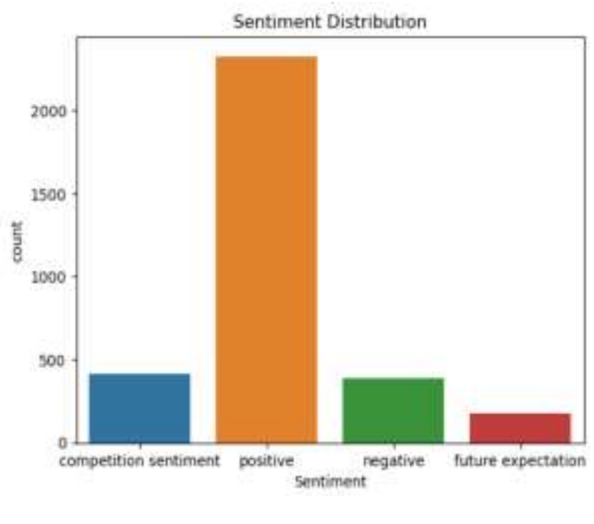
- This process involves printing the coherence scores for each number of topics tested, providing a clear view of how well each set of topics is organized and understandable.

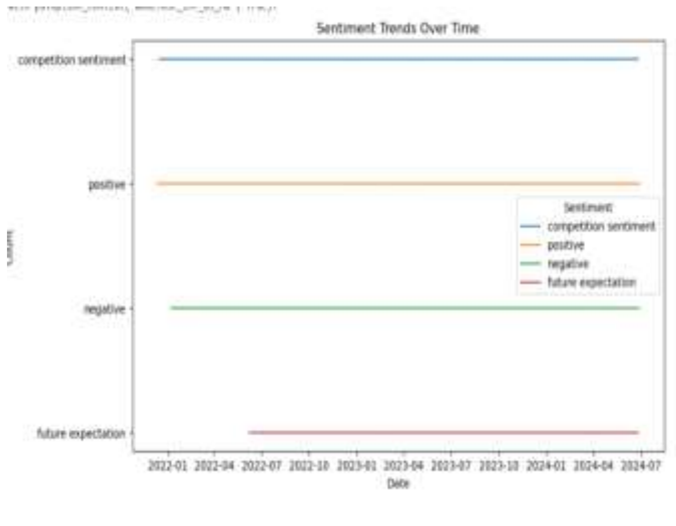
ANALYZING THE OPTIMAL TOPIC MODEL:

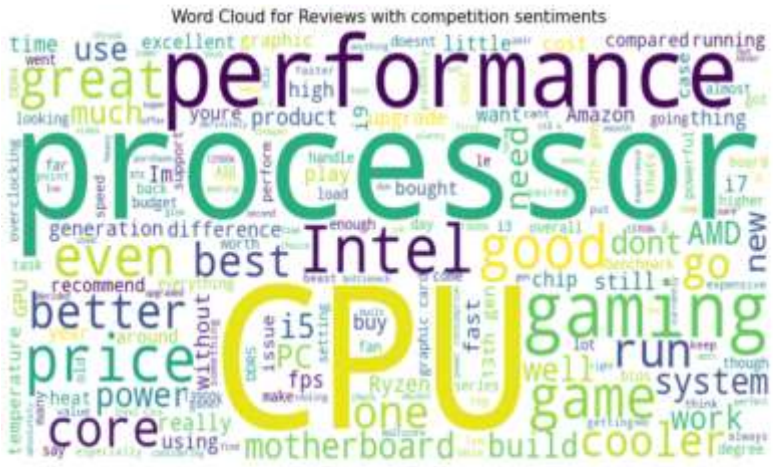
- The optimal topic model (optimal_model) was chosen from a list of trained models (model_list) based on metrics like coherence scores.
- Using optimal_model.show_topics(formatted=False), structured topics were retrieved, and optimal_model.print_topics(num_words=10) was used to display key terms for each topic.

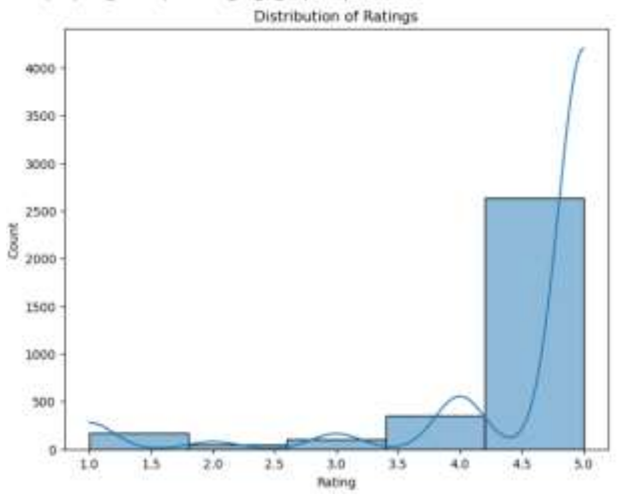
VISUALIZING LDA TOPIC MODEL RESULTS:

- The code snippet uses pyLDAvis to visualize the results of an LDA model directly within Jupyter Notebook or JupyterLab.
- First, pyLDAvis.enable_notebook() enables the interactive visualization feature. Then, pyLDAvis.gensim.prepare(optimal_model, corpus, id2word) prepares the visualization data.
- Here, optimal_model represents the trained LDA model, corpus is the text data used for training, and id2word maps word IDs to actual words in the corpus.
- The resulting visualization (vis) provides interactive displays that help explore the topics identified by the model.









ASSIGNING MAIN TOPICS USING LDA TOPIC MODELING:

- The code snippet applies an LDA model (optimal_model) to categorize reviews in a DataFrame (df) based on their main topics. Each review in the corpus (corpus) is analyzed to determine the probabilities of belonging to different topics.
- By converting these probabilities into a readable format, the code identifies the dominant topic for each review.
- This method helps reveal the main themes within the reviews.

VADER:

- To conduct sentiment analysis, the code initializes a SentimentIntensityAnalyzer object from the vaderSentiment library.
- This tool assesses sentiment using a lexicon and rules-based approach, providing scores for positivity, negativity, neutrality, and an overall compound sentiment score.
- The setup includes downloading the necessary lexicon using nltk.download('vader_lexicon'), ensuring the analyzer has access to a comprehensive set of sentiment indicators.

ENHANCING DATAFRAME WITH SENTIMENT ANALYSIS SCORES:

- In this code snippet, sentiment polarity scores are calculated for each review title in the DataFrame (df) using the Vader sentiment analysis tool initialized earlier.
- The process begins by iterating through each review title, extracting the text and author's identifier.
- For each review title, the sentiment analyzer computes scores for positive,negative, future expectations and competition sentiments and an overall compound sentiment score, storing these results in a dictionary.
- Next, the dictionary is converted into a new DataFrame where each row represents an author and each column contains a specific sentiment score.
- The resulting sentiment scores are then merged back into the original DataFrame (df) based on the author's identifier, enhancing it with valuable insights into the emotional tones expressed in the review titles.

ADDING SENTIMENT LABELS TO DATAFRAME:

- In this section, a function named get_tag is defined to categorize sentiment based on the compound score generated earlier.
- The function evaluates the sentiment score (val) and assigns labels such as 'Negative', 'Somewhat Negative', 'Somewhat Positive', 'Positive', or 'Neutral' accordingly.
- Once the function is defined, it is applied to the 'compound' column of the df_sentiment DataFrame using .apply(), creating a new column named 'Sentiment' that captures the labeled sentiment for each review.

VISUALIZING SENTIMENT SCORES ACROSS COMMENT CLUSTERS:

- Creating a figure with three horizontal subplots using Seaborn (sns.barplot). Each subplot represents sentiment scores—positivity (pos), neutrality (neu), and negativity (neg)—across different comment clusters (cluster) from the df_sentiment DataFrame.

- The plots provide insights into how sentiments are distributed within each cluster. Titles for each subplot ('Positivity', 'Neutrality', 'Negativity') emphasize the sentiment focus, while axis labels clarify the sentiment score ranges and comment cluster distinctions.

UNDERSTANDING SENTIMENT ANALYSIS WITH PYTHON'S NLTK:

- sia.polarity_scores('I am so happy') and sia.polarity_scores('I am so sad') both utilize the SentimentIntensityAnalyzer() tool from the Natural Language Toolkit (NLTK) in Python.
- These lines are used to analyze the sentiment of the provided text strings. In the first case, "I am so happy" is likely to receive a high positive score because the words "happy" and "so" express strong positive emotions.
- Conversely, in the second case, "I am so sad" expresses sadness, a negative emotion, which would result in higher negative scores and possibly lower positive scores.

RUN POLARITY SCORE IN ENTIRE DATASET:

- The code snippet processes a dataset to analyze sentiment using Python's NLTK library. It iterates through each review title in the dataset, calculates sentiment scores (positive, negative, neutral, compound) using SentimentIntensityAnalyzer() from NLTK, and stores the results.

ROBERT a:

- RoBERTa tokenizer from the transformers library in Python is used here. It starts by importing RobertaTokenizer from this library and then proceeds to load the tokenizer for the 'roberta-base' model using RobertaTokenizer.from_pretrained('roberta-base').
- Next, it defines review_lengths as a list comprehension that iterates over each review (review) in the dataset. For each review, it calculates the number of tokens by applying the tokenizer.tokenize(review) method.
- This method breaks down each review into its constituent tokens, and the lengths of these tokenized sequences are stored in the review_lengths list.

- This process is crucial for preparing text data for further analysis or modeling tasks.
- Tokenization transforms raw text into a format that can be processed by machine learning models, enabling tasks such as sentiment analysis or text classification.

PREPARING TEXT DATA WITH ROBERTA TOKENIZER:

- Utilizes the RoBERTa tokenizer from the transformers library along with PyTorch to prepare text data for machine learning tasks.
- It first loads the 'roberta-base' tokenizer and sets a maximum sequence length of 512 tokens. It then tokenizes both training (X_train) and testing data (X_test), ensuring each sequence is padded or truncated to the specified maximum length.
- The tokenized sequences are converted into PyTorch tensors ('pt' format), making them suitable for training neural networks. This preprocessing step is crucial for transforming raw text into a format that can be processed by models like RoBERTa, typically used for tasks such as sentiment analysis or text classification.

CUSTOM DATASET PREPARATION FOR TEXT ANALYSIS WITH PYTORCH:

- The provided Python code defines a custom dataset class called ReviewDataset using PyTorch for managing text data in machine learning workflows.
- In the __init__ method, the dataset is initialized with encodings containing tokenized representations of text (X_train_tokens and X_test_tokens) and labels (y_train.values and y_test.values) representing the corresponding target categories.
- The __getitem__ method retrieves an item from the dataset at a specified index (idx). It constructs a dictionary (item) where each tokenized input (encodings) is converted into a PyTorch tensor, and the target label (labels) is also converted into a tensor. This structured format ensures uniformity in dataset handling for model training or evaluation.
- The __len__ method returns the total number of samples in the dataset, determined by the length of the labels array.

- These dataset objects (train_dataset and test_dataset) are essential for preprocessing text data and preparing it for use in training or evaluating machine learning models, such as those used in sentiment analysis or text classification tasks.

CONFIGURING ROBERTA MODEL TRAINING FOR SEQUENCE CLASSIFICATION:

- In this Python code snippet, training is configured for a RoBERTa model using the transformers library, which is widely recognized for its applications in natural language processing tasks. The script begins by loading a pre-trained RoBERTa model (roberta-base) specifically tailored for sequence classification tasks. This model is designed to process textual data and classify it into different categories or classes; in this instance, it's set to classify into 4 classes, a parameter that can be adjusted based on the specific dataset requirements.
- Following this, various training parameters are defined using TrainingArguments. These parameters dictate the course of the training process. They include specifying the output directory (output_dir) where model checkpoints and results will be stored, setting the number of training epochs (num_train_epochs) to determine how many times the model iterates through the entire dataset, and configuring batch sizes (per_device_train_batch_size and per_device_eval_batch_size) to efficiently handle training and evaluation data.
- Additionally, settings such as warmup_steps and weight_decay are incorporated to fine-tune the model effectively, alongside defining logging directories (logging_dir) and intervals (logging_steps) to monitor and record training progress.
- To manage the training procedure, a Trainer object is instantiated. This object takes the RoBERTa model, training arguments, and datasets (train_dataset for training and test_dataset for evaluation). This setup ensures efficient training and evaluation of the model on the specified dataset, preparing it for tasks like sentiment analysis or other forms of sequence classification that require robust comprehension and classification of textual data.

TRAINING AND EVALUATING THE MODEL:

- The provided code snippet initiates and evaluates the training process for a RoBERTa model configured for sequence classification using the transformers library in Python.
- The trainer.train() command executes the training loop, where the model learns from the labeled data provided by train_dataset. During training, the model adjusts its parameters to minimize the prediction error and improve its ability to classify sequences into predefined categories or classes.
- Once training completes, trainer.evaluate() assesses the model's performance using the eval_dataset, which contains unseen data reserved for validation. This evaluation computes metrics such as accuracy, precision, recall, and F1-score to gauge how well the model generalizes to new data. The results of this evaluation are stored in the eval_result variable.
- Finally, the script outputs the evaluation results using print(f"Evaluation results: {eval_result}"), providing insights into the model's effectiveness in classifying sequences based on the configured training settings and dataset.

EVALUATING ROBERTA MODEL PERFORMANCE:

- Evaluating a RoBERTa model's performance using predictions on a test dataset. It begins by importing necessary libraries: \`numpy\` for numerical operations and \`accuracy_score\`, \`classification_report\` from \`sklearn.metrics\` for evaluation.
- The predictions are generated using \`trainer.predict(test_dataset)\`, and \`np.argmax(predictions.predictions, axis=1)\` computes the predicted class labels (\`y_pred\`).
- It then prints a \`classification_report\` detailing precision, recall, F1-score, and support for each class based on comparisons between true labels (\`y_test\`) and predicted labels (\`y_pred\`).

DEEP LEARNING:

LSTM:
TEXT CLASSIFICATION:

- Trains a Long Short-Term Memory (LSTM) neural network model using TensorFlow's Keras API for a text classification task. The model architecture begins with an Embedding layer, which converts input sequences into dense vectors of fixed size suitable for LSTM processing.

- The `input_dim` parameter of 5000 specifies the vocabulary size, and `input_length defines the maximum length of input sequences.

- The LSTM layer with 64 units is added to learn patterns in sequential data, incorporating dropout to prevent overfitting by randomly deactivating 20% of neurons during training and recurrent dropout to regulate the dropout of recurrent units.

- Two fully connected Dense layers follow, each with 128 and 64 units, respectively, using ReLU activation for non-linearity and L2 regularization (`kernel_regularizer=l2(0.001)`) to constrain the complexity of the model and improve generalization.

- The output layer employs a softmax activation function to classify input sequences into one of four classes, assuming a classification task with four possible outcomes.
- The model is compiled with the Adam optimizer, sparse categorical cross-entropy loss function suitable for integer-encoded class labels (`y_train`), and accuracy as the metric for evaluation.

- The `model.summary()` method provides a concise overview of the model architecture, displaying the layers, their output shapes, and the number of parameters.

- An `EarlyStopping` callback is employed to halt training if the validation loss (`val_loss`) fails to improve after 10 epochs, ensuring the model retains its best performance on unseen data (`restore_best_weights=True`).

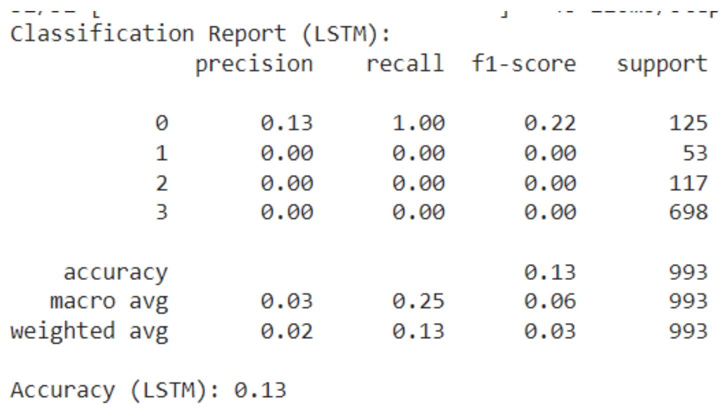
- Training begins with `model.fit()`, where `X_train_pad` and `y_train` are input sequences and corresponding labels. The process runs for up to 100 epochs, adjusting batch size, validation split, and class weights for handling class imbalance if applicable. The `callbacks` parameter integrates early stopping, and `verbose=1` provides detailed progress updates during training.

- This setup optimizes the LSTM model for text classification, leveraging sequential data patterns to achieve accurate predictions while mitigating overfitting through regularization and early stopping strategies.

EVALUATING LSTM MODEL PERFORMANCE:

- This code evaluates a Long Short-Term Memory (LSTM) model built with TensorFlow's Keras API for text classification.
- It predicts labels (`y_pred_classes_lstm`) for the test dataset (`X_test_pad`) using `model_lstm.predict()` and calculates classification metrics like precision, recall, and F1-score using `classification_report`.
- The accuracy of the model is also computed using `accuracy_score`, providing an overall measure of its performance on the test data.

The classification report for LSTM Model is



```
Classification Report (LSTM):
              precision    recall  f1-score   support

           0       0.13      1.00      0.22       125
           1       0.00      0.00      0.00        53
           2       0.00      0.00      0.00       117
           3       0.00      0.00      0.00       698

    accuracy                           0.13       993
   macro avg       0.03      0.25      0.06       993
weighted avg       0.02      0.13      0.03       993

Accuracy (LSTM): 0.13
```

BiDLSTM:

- Begins by loading a dataset from a CSV file containing reviews and their corresponding sentiments.
- It cleans the text by removing non-letter characters and converting everything to lowercase. Then, it tokenizes the cleaned text to numerical sequences using TensorFlow's Tokenizer, limiting the vocabulary size to 5000 words and ensuring all sequences are the same length (200).
- The sentiment labels are encoded numerically for model training. The data is split into training and testing sets for evaluating the model's performance.
- The neural network model, built with Keras, includes layers for embedding words, two bidirectional LSTM layers to capture context from both directions in text sequences, and dense layers for classification.
- The model is trained on the training set, optimizing to minimize categorical cross-entropy loss using the Adam optimizer over 10 training cycles.
- After training, the entire model is saved for future use (`bilstm_model.h5`), along with the Tokenizer and LabelEncoder objects, which are serialized for text preprocessing and model deployment tasks.

SENTIMENT ANALYSIS WITH BIDIRECTIONAL LSTM MODEL:

- This script demonstrates how to use a trained Bidirectional LSTM model for sentiment analysis on text input. First, the clean_text function removes special characters and converts the input text to lowercase. The model is loaded using TensorFlow's load_model function.

- The tokenizer and label encoder are deserialized using pickle for text preprocessing and decoding predictions, respectively.
- The preprocess_input function takes a raw text input, cleans it using clean_text, converts it into a sequence of numerical tokens using the tokenizer, and pads the sequence to ensure consistent length (200 tokens in this case).
- An example input_text is provided, representing a customer's review. It is preprocessed using preprocess_input, then fed into the loaded model to obtain a prediction. The prediction is decoded using the label encoder to retrieve the sentiment label, which is then printed as the predicted sentiment.
- This setup allows for efficient sentiment analysis on new text inputs using the previously trained model and saved preprocessing tools.

CONVOLUTIONAL NEURAL NETWORK:

- Trains the Convolutional Neural Network (CNN) model using TensorFlow's Keras API for text classification.
- The model architecture starts with an Embedding layer that converts input sequences into dense vectors suitable for CNN processing. It uses a vocabulary size (`input_dim`) of 5000 and a maximum sequence length defined by `max_len`.

- The CNN architecture includes two Conv1D layers: the first with 128 filters and the second with 64 filters, both employing a kernel size of 5 and ReLU activation functions to detect local patterns in the input sequences.

- MaxPooling1D layers follow each Conv1D layer to down-sample the output, reducing computational complexity while preserving essential features. GlobalMaxPooling1D then aggregates the maximum feature value across all features, capturing the most significant features for classification.

- Subsequently, two Dense layers with 128 and 64 units, respectively, utilize ReLU activation and incorporate L2 regularization to prevent overfitting. Dropout layers with a dropout rate of 50% are included to further enhance generalization by randomly deactivating neurons during training.

- For classification, the output layer employs a softmax activation function, enabling the model to classify input sequences into one of four classes, assuming a multi-class classification task. The model is compiled using the Adam optimizer, with sparse categorical cross-entropy as the loss function, suitable for integer-encoded class labels. The metric used for evaluation is accuracy.

- The `model.summary()` method provides a succinct overview of the model's architecture, detailing the configuration of each layer, their output shapes, and the total number of trainable parameters.

- To monitor and optimize training, an `EarlyStopping` callback is implemented. This callback halts training if the validation loss does not improve after 10 epochs, ensuring the model retains its best performance on unseen data.

- Training commences with `model.fit()`, where `X_train_pad` and `y_train` represent the input sequences and their corresponding labels. The training process spans 18 epochs, with adjustments made to the batch size, validation split, and class weights as necessary for managing class imbalance. `Verbose=1` provides detailed updates on the training progress.

ARTIFICIAL NEURAL NETWORK:

- Prepareing text data for training a machine learning model, focusing on handling class imbalance, tokenizing text, and managing varying sequence lengths.
- First, the features (`X`) and target variable (`y`) are defined from the DataFrame `df`, with `X` being the 'ReviewContent' and `y` being the encoded sentiment labels (`Sentiment_encoded`).
- The data is split into training and testing sets using `train_test_split`, ensuring that the class distribution is maintained across both sets with a 70-30 split (`test_size=0.3`) and a fixed random state for reproducibility.

- A `Tokenizer` is then initialized to handle a vocabulary size of 5000, including an out-of-vocabulary token (`<OOV>`), and fitted on the training data (`X_train`). This tokenizer converts the text data into sequences of integers, where each integer represents a specific word in the vocabulary.

- The length of these tokenized sequences is analyzed, and a histogram is plotted to visualize their distribution. This step helps determine the appropriate maximum sequence length (`max_len`), which is set to the length of the longest review.

- To ensure uniform sequence lengths, the sequences are padded using `pad_sequences`. This function pads shorter sequences with zeros at the end (`padding='post'`) and truncates longer sequences from the end (`truncating='post'`) to match the maximum length (`max_len`).

- Finally, class weights are computed to address class imbalance during training. The `compute_class_weight` function calculates weights for each class based on their frequency, resulting in a balanced class weight dictionary.

TRAINING AN ARTIFICIAL NEURAL NETWORK (ANN) FOR TEXT CLASSIFICATION:

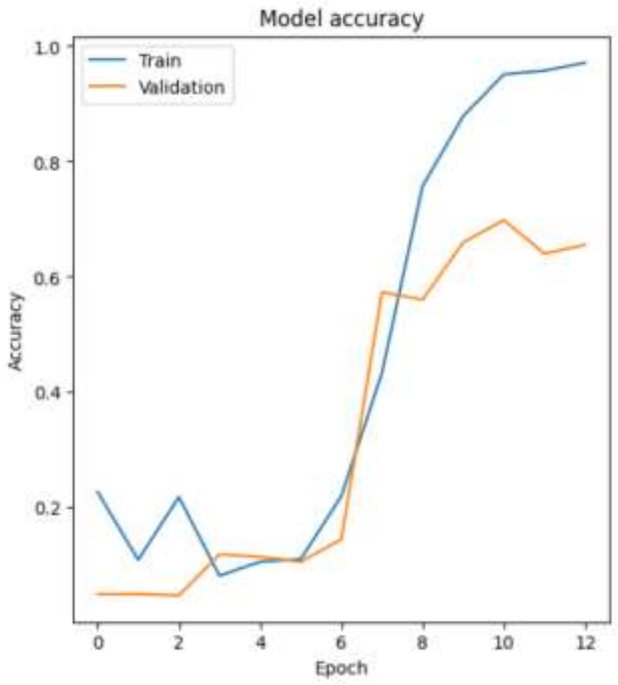
- Training an Artificial Neural Network (ANN) for text classification using TensorFlow's Keras API. The model architecture begins with an Embedding layer that converts input sequences into dense vectors with a specified dimensionality, designed to capture the semantic meaning of words.
- The embedding layer accommodates a vocabulary of 5000 words and sequences up to `max_len`.
- The subsequent Flatten layer transforms the 2D matrix output of the Embedding layer into a 1D vector, making it suitable for the Dense layers.
- The model includes two fully connected Dense layers: the first with 128 units and the second with 64 units, both using ReLU activation functions and L2 regularization to prevent overfitting.
- Dropout layers with a 50% dropout rate follow each Dense layer to further mitigate overfitting by randomly deactivating neurons during training.

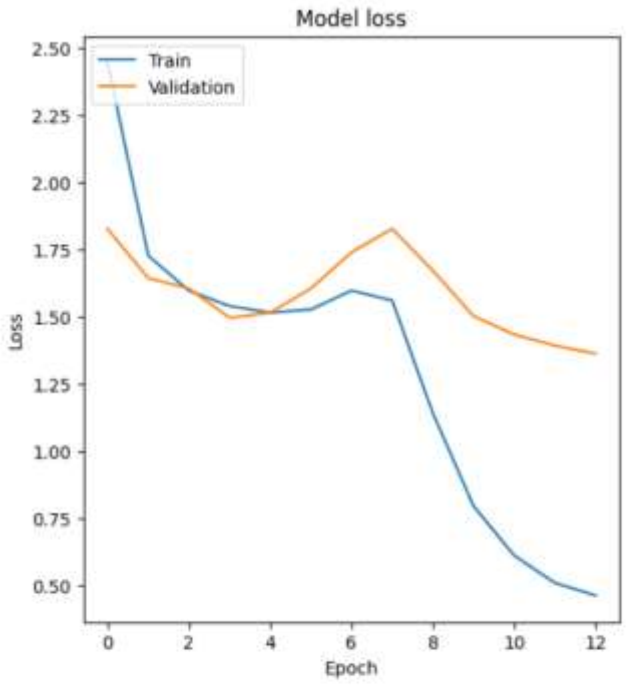
- The output layer uses a softmax activation function to classify input sequences into one of four classes, assuming a multi-class classification task.
- The model is compiled with the Adam optimizer and sparse categorical cross-entropy as the loss function, suitable for integer-encoded class labels (`y_train`). The accuracy metric is used to evaluate the model's performance.
- The model's architecture is summarized using the `model_ann.summary()` method, providing an overview of each layer's configuration and the total number of parameters.
- An `EarlyStopping` callback is set up to monitor the validation loss and halt training if no improvement is observed after 10 epochs, restoring the best weights obtained during training.

- Training begins with the `model_ann.fit()` method, using the padded training sequences (`X_train_pad`) and their corresponding labels (`y_train`). The training process spans 13 epochs, with a batch size of 32, a validation split of 20%, and class weights (`class_weights_dict`) to handle class imbalance. Detailed training progress is displayed with `verbose=1`.

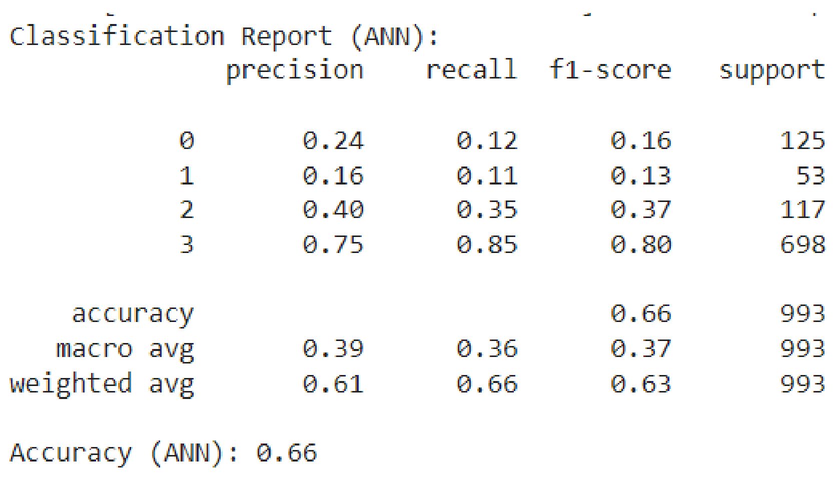
VISUALIZING TRAINING AND VALIDATION PERFORMANCE:

- This code visualizes the training process of the ANN model by plotting the accuracy and loss for both training and validation datasets over each epoch.
- The first plot shows the accuracy, highlighting how well the model performs on the training data compared to the validation data. The second plot displays the loss, indicating how much error the model has during training and validation.
- These plots help assess the model's learning and generalization capabilities by comparing performance metrics throughout the training period. The graphs are displayed side by side for a comprehensive view of the training progress.





performance across different classes, and calculating the overall accuracy score to measure its effectiveness in sentiment classification.



```
Classification Report (ANN):
              precision    recall  f1-score   support

           0       0.24      0.12      0.16       125
           1       0.16      0.11      0.13        53
           2       0.40      0.35      0.37       117
           3       0.75      0.85      0.80       698

    accuracy                           0.66       993
   macro avg       0.39      0.36      0.37       993
weighted avg       0.61      0.66      0.63       993

Accuracy (ANN): 0.66
```

DATA PREPARATION FOR SENTIMENT ANALYSIS MODEL TRAINING WITH RATINGS:

- This script focuses on preparing the data for training a sentiment analysis model using natural language processing techniques. It starts by defining the features and target variable: X_text represents the textual content of reviews, X_rating denotes the normalized ratings associated with each review, and y contains the encoded sentiment labels.
- To ensure an unbiased evaluation of the model, the dataset undergoes a stratified split into training and testing sets. .
- This method preserves the proportional representation of sentiment classes in both subsets, crucial for maintaining model performance metrics like accuracy and precision.
- Next, the textual data (X_text) is processed using a tokenizer from TensorFlow's Keras API. The tokenizer converts each review into sequences of numerical tokens and assigns indices to each unique word in the corpus.
- Additionally, sequences are padded or truncated to a maximum length based on the longest review's length in the training set. This uniformity in sequence length facilitates efficient batch processing during model training.

- To address potential class imbalance in the sentiment labels (y_train), class weights are computed using compute_class_weight from scikit-learn. These weights adjust the contribution of each class during model training, ensuring that less frequent classes have a proportionally greater impact on the model's learning process.
- Furthermore, the processed training (X_text_train_pad, X_rating_train) and testing data (X_text_test_pad, X_rating_test) are saved into pickle files (X_train.pkl, X_test.pkl, y_train.pkl, y_test.pkl). This storage method optimizes data retrieval and enhances overall workflow efficiency, especially when dealing with large datasets.

INTEGRATING TEXT AND RATING FEATURES IN SENTIMENT ANALYSIS MODEL:

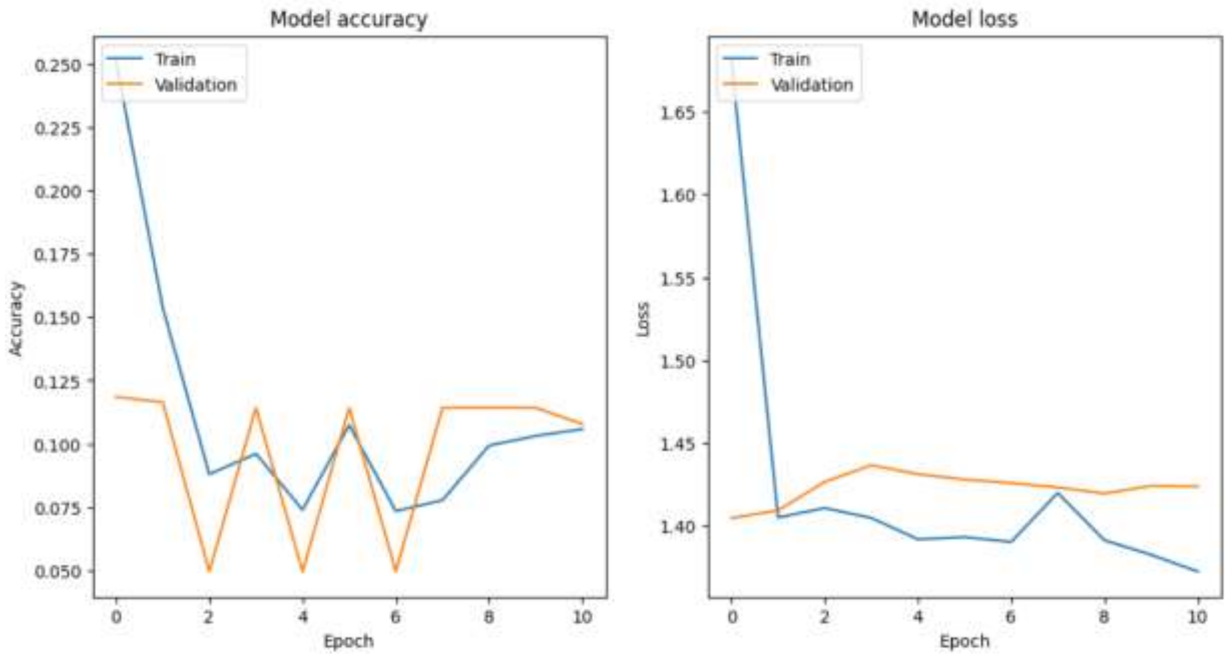
- This script defines an artificial neural network (ANN) model that incorporates both textual content and rating features for sentiment analysis. The model architecture involves two input layers: `text_input` for textual data and `rating_input` for normalized ratings associated with each review.
- For the `text_input`, a tokenizer converts each review into sequences of numerical tokens, which are then embedded using an embedding layer.
- This layer transforms sparse input vectors into dense vectors of fixed size, facilitating meaningful representation of words in the review corpus. The `flatten` layer reshapes the embedded sequences into a flat vector suitable for further processing.
- Simultaneously, the `rating_input` accepts single-dimensional inputs representing normalized ratings, ensuring the model considers this additional numerical feature alongside textual content. These inputs are concatenated after processing to merge the features from both streams.
- The subsequent layers apply nonlinear transformations to the concatenated features, enhancing the model's ability to capture complex patterns in the combined data. Dropout layersare strategically placed to mitigate overfitting by randomly dropping a fraction of input units during training.
- The output layer uses softmax activation to predict the probability distribution across the sentiment classes (assuming 4 classes). The

model is compiled with an Adam optimizer and sparse categorical cross-entropy loss function, optimizing for accuracy during training.

- During training (`model_ann_rating.fit`), the concatenated inputs (`X_text_train_pad` and `X_rating_train`) are fed into the model along with sentiment labels (`y_train`). The process is iterated over multiple epochs, with batch size and validation split parameters specified. Optionally, class weights (`class_weights_dict`) are incorporated to handle class imbalance, ensuring the model's robustness across all sentiment categories.

VISUALIZING MODEL TRAINING PERFORMANCE:

- This snippet visually tracks an artificial neural network (ANN) model's progress during training for sentiment analysis. The first subplot shows the evolution of training (`accuracy`) and validation (`val_accuracy`) accuracies across epochs, revealing how well the model generalizes to new data.
- In the second subplot, `loss` and `val_loss` curves depict the model's error minimization during training and validation phases.
- Lower loss values indicate improved predictive accuracy.
- These visualizations offer insights into the model's learning behavior, aiding adjustments to enhance performance and address issues like overfitting or underfitting in sentiment analysis tasks.



LARGE LANGUAGE MODEL META-LEARNING APPROACH:

- This code snippet facilitates logging into the Hugging Face platform programmatically using Python. It begins by importing necessary libraries: `transformers` for working with transformer models and `torch` .

SETTING UP MACHINE LEARNING ENVIRONMENT AND CUSTOM MODELS:

- The provided Python code snippet sets up an environment for machine learning and natural language processing tasks. It begins by importing essential libraries such as `os` for system operations and `torch` for tensor computations. The `datasets` module is imported to facilitate dataset loading, while the `transformers` library is extensively utilized for its capabilities in handling pre-trained models tokenization, and specialized configurations.
- Additionally, the code imports custom classes `LoraConfig` and `PeftModel` from the `peft` module, and `SFTTrainer` from `trl`, highlighting the integration of custom models and training procedures into the workflow.
- Overall, these imports establish a robust foundation for leveraging pre-trained models, custom configurations, and specialized training methods essential for advanced machine learning applications in natural language understanding and processing.r utilizing PyTorch functionalities.
- The `login` function from `huggingface_hub` is then imported, which is essential for interacting with the Hugging Face Hub. The `login` function is subsequently called with an authentication token (`token`) provided as a parameter. This token serves as a secure credential that grants access to the user's Hugging Face account.

- Once authenticated, users can upload, manage, and version control various resources such as models and datasets on the Hugging Face platform.

CONFIGURING MODEL BEHAVIOR WITH QLORA PARAMETERS:

- The `model.config` settings play a pivotal role in configuring the behavior and training setup of the model.
- The `problem_type` parameter is set to `"single_label_classification"`, indicating that the model is specifically designed to handle classification tasks where each input corresponds to a single class label.
- Setting `use_cache` to `False` ensures that the model does not store intermediate computations, which can be advantageous in scenarios where memory usage needs to be optimized or where inputs are dynamically changing.
- The `pretraining_tp = 1` parameter likely signifies a specific phase or type of pretraining task that the model underwent during its development. These configurations are essential as they tailor the model's operations to efficiently handle classification tasks while optimizing performance and resource usage.

TOKENIZING DATA WITH LLAMA:

- The LLaMA tokenizer is loaded using `AutoTokenizer.from_pretrained` with `trust_remote_code=True`, ensuring safe retrieval of tokenization rules.
- The tokenizer's `pad_token` is set to `tokenizer.eos_token`, indicating padding is applied with end-of-sequence tokens. Both training and testing data are tokenized with `max_length=512`, padding sequences to the right and truncating where necessary.
- This process prepares the data for further processing and input into the model for tasks such as classification or generation.

CUSTOM DATASET AND DATA COLLATION:

- The `ReviewDataset` class defined here inherits from PyTorch's `Dataset` class and is designed to handle tokenized encodings and corresponding labels. The `__getitem__` method retrieves an item at a specific index, converting the encodings and labels into a dictionary formatted for PyTorch's DataLoader.
- Labels are converted to tensors for compatibility with PyTorch's training routines. The `__len__` method returns the total number of samples in the dataset.
- For dynamic padding during batch processing, the `DataCollatorWithPadding` from the `transformers` library is employed. This data collator ensures that sequences within each batch are padded to the maximum length encountered among sequences in that batch, optimizing efficiency during training.

FORMATTING FUNCTION FOR SFT TRAINER:

- The `formatting_func` defined here is tailored for use with the `sfttrainer`. It takes an `example` dictionary containing `input_ids`, `attention_mask`, and `labels` keys, which are typically outputs from a tokenizer when processing text data.
- The function returns a dictionary with these keys intact, ensuring that the data format aligns with the requirements of the `sfttrainer` for training or evaluation tasks.

CUSTOMIZED TRAINING WITH SFT TRAINER:

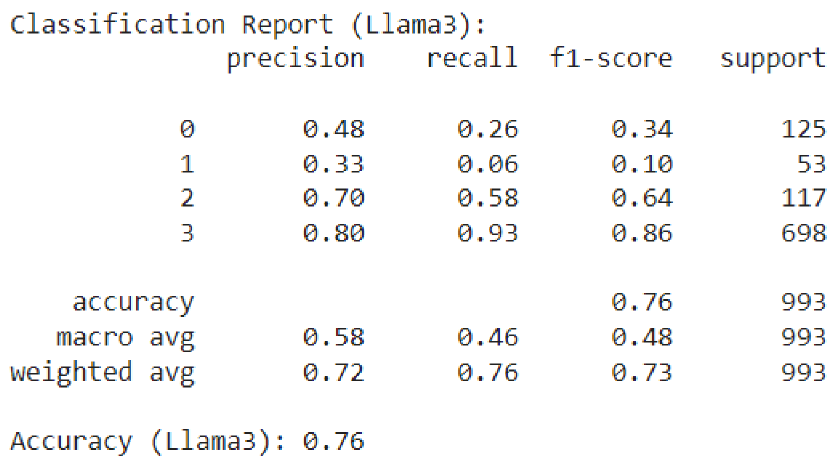
- A custom `SFTTrainer` named `CustomSFTTrainer` is defined by extending the base `SFTTrainer` class from the `trl` library.
- This custom trainer inherits the functionality of `SFTTrainer` while allowing for specific modifications to the training process.
- The `training_step` method within `CustomSFTTrainer` overrides the default training step behavior. It prepares inputs, computes the loss using the model and inputs, and then performs a backward pass to

update gradients. If `gradient_accumulation_steps` is greater than 1, the loss is normalized accordingly before backpropagation.

- Training parameters such as `output_dir`, `num_train_epochs`, `per_device_train_batch_size`, and others are set using `TrainingArguments`.
- These parameters control various aspects of the training process, including batch size, learning rate, optimization algorithm (`optim`), and logging frequency.
- An instance of `CustomSFTTrainer` is initialized with the defined model, training arguments, datasets (`train_dataset` and `test_dataset`), tokenizer, data collator for padding, `peft_config` for LoRA configuration, and a `formatting_func` for data formatting.
- This setup ensures that the model is trained under customized conditions suited for sequential classification tasks (`TaskType.SEQ_CLS`).
- The `trainer.train()` method is called to initiate the training process, where the model iteratively learns from the training dataset.
- And finally evaluating the model using the trainer.evaluate() method.

MODEL EVALUATION AND PERFORMANCE:

- After training the model using custom fine-tuning techniques with `trainer.train()`, the next step involves evaluating its performance. The code snippet begins by making predictions on the `test_dataset` using the trained model stored in `trainer`.
- This is achieved through `predictions = trainer.predict(test_dataset)`, where the results are stored in `predictions`.
- Subsequently, `np.argmax’ extracts the predicted class labels by selecting the indices with the highest probabilities across classes.
- To assess the model's performance, a classification report is generated using `classification_reporT, which details metrics like precision, recall, and F1-score for each class based on comparisons between the true labels and predicted labels.
- Additionally, the accuracy score is calculated to provide a single metric indicating the model's overall predictive accuracy.



```
Classification Report (Llama3):
              precision    recall  f1-score   support

           0       0.48      0.26      0.34       125
           1       0.33      0.06      0.10        53
           2       0.70      0.58      0.64       117
           3       0.80      0.93      0.86       698

    accuracy                           0.76       993
   macro avg       0.58      0.46      0.48       993
weighted avg       0.72      0.76      0.73       993

Accuracy (Llama3): 0.76
```

Among the techniques utilized, the Llama model stands out as the most effective for sentiment analysis in our project analyzing the highest accuracy.

When using Class weights for data imbalance

Model	Accuracy	Recall	F1-Score	Precision
SVC	0.72	0.72	0.64	0.66
RF	0.73	0.73	0.67	0.68
XGB	0.714	0.71	0.63	0.63
Naïve B	0.72	0.72	0.63	0.65

When using SMOTE for data imbalance

Model	Accuracy	Recall	F1-Score	Precision
SVC	0.702	0.7	0.62	0.63
RF	0.72	0.72	0.67	0.66
XGB	0.68	0.68	0.66	0.65
Naïve B	0.53	0.53	0.57	0.67

When using DL models

Model	Accuracy	Recall	F1-Score	Precision	Epochs	Optimizer	Neurons
ANN	0.7	0.69	0.63	0.61	10	adam	512
CNN	0.69	0.69	0.64	0.66	20	adam	300
LSTM	0.13	0.13	0.03	0.02	100	adam	128

CONCLUSION:

This project focuses on Intel sentiment analysis from online reviews, leveraging machine learning, deep learning techniques, LLM models, and ensemble techniques to achieve superior accuracy. By systematically applying these methods, valuable insights into customer sentiments are extracted, aiding Intel in optimizing products and enhancing customer satisfaction effectively.