Text Classification — Sentiment Analysis (MI201 Project 3)

1st Carlos Adrian Meneses Gamboa
Ingénieur Degree Programme in STIC
ENSTA Paris
Paris, France
carlos.meneses@ensta-paris.fr

2nd Jose Daniel Chacon Gomez
Ingénieur Degree Programme in STIC
ENSTA Paris
Paris, France
jose-daniel.chacon@ensta-paris.fr

3rd Santiago Florido Gomez
Ingénieur Degree Programme in STIC
ENSTA Paris
Paris, France
santiago.florido@ensta-paris.fr

***Abstract*—This work presents a sentiment analysis system for short English texts and compares classical machine learning methods with a transformer-based approach. Using standard vector representations (bag-of-words and TF-IDF), we train and evaluate several classifiers and benchmark them against a model leveraging BERT embeddings. Results are reported with accuracy and macro-F1, highlighting differences in performance, robustness, and computational cost. The study provides practical guidance on selecting an appropriate sentiment classification pipeline under typical resource constraints.**

***Index Terms*—sentiment analysis, NLP, text classification, TF-IDF, BERT**

I. Introduction

Sentiment analysis of short texts becomes fundamentally important when perceptions are considered a critical information asset for product and service owners [1]. This is especially relevant in the development of emotion-driven systems, which can yield meaningful insights to improve the user or customer experience. For example, these insights can lead to adjustments in customer-support strategies or to more targeted marketing campaigns [2]. As a conceptual input for such improvements, search systems or sentiment-analysis approaches can be adapted to focus on the emotions expressed by the target population. In this context, social networks—and more specifically short messages such as tweets and comments on multimedia platforms—are among the most commonly used sources for conducting this type of analysis.

This project focuses on the automatic sentiment analysis of short English texts. First, an exploratory phase is conducted in which the dataset content is preprocessed, and a preliminary analysis of the information is performed using traditional machine-learning methods. Subsequently, the classification stage is carried out with standard classifiers such as Naive Bayes, Logistic Regression, and Linear SVM, using multiple text representation schemes, including bag-of-words, word-level TF–IDF, and character-level TF–IDF. Model performance is reported using accuracy, macro-F1, and complementary metrics to ensure a fair comparison.

Next, a multilayer perceptron (MLP) trained on vectorized text is evaluated, and an alternative based on BERT embeddings is studied to capture contextual semantics. To this end, the performance of MLP models built for each vectorization approach is compared across four network architectures, each adapted to the amount of information provided by the corresponding vectorizer or by BERT, and oriented toward a final three-class classification. In addition, an appropriate depth is defined according to the level of detail in the input representation in order to reduce overfitting on the training data. Dropout layers are also incorporated between hidden layers to further control overfitting and overtraining.

Finally, in order to improve message classification, strategies based on large language models (LLMs) were evaluated by using the API version of the Gemma 3-4b-it (Gemini) model to compare its performance as a short-text classifier against the previously trained models. In addition, LoRA was used to perform an efficient fine-tuning of BERT-based transformers [3].

II. Q0-Dataset analysis with different classical machine learning model

The Sentiment Data Analysis dataset to which access is available is composed of sets of short tweets in English that are classified into three categories according to their polarity as positive, neutral, or negative in the "sentiment" column. Additionally, there is information related to the tweet metadata in terms of the time of day when it was published, the user's age, and their country of origin in the columns "Age of User" and "country".

It is initially proposed to perform preprocessing of the text field, starting with the removal of null values and then, using NLTK, removing stopwords, which allows obtaining the processed text column without very frequent words in English that do not have a significant semantic contribution [4].

A. Exploratory data analysis (EDA)

An analysis of the class distribution in the training data is performed. As shown in Fig. 1, although the presence of neutral-polarity data exceeds the other classes, there is no substantial imbalance in the training dataset that could be associated with bias in the classifiers to be developed.

It is useful, prior to implementing some machine-learning strategies, to examine the behavior of groups of terms (bag-of-words) with respect to polarity classes, mainly from a frequency-based perspective.

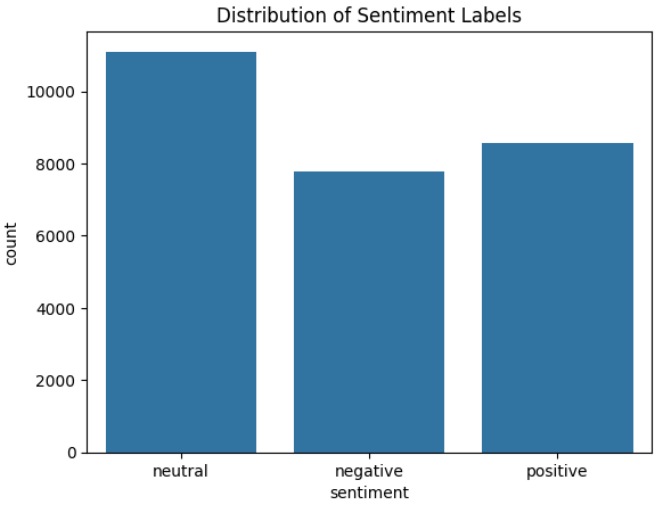


Fig. 1: Distribution of sentiments in training dataset.

This highlights the importance of the concept of a vectorizer, which is responsible for converting collections of texts—such as the content of the "processed text" column—into numerical vectors. This implies obtaining a sparse representation of each document after applying the vectorizer. This process is precisely known as vectorization, and depending on the way the numerical representation is constructed in the sparse matrix, it can be classified into several types. In this work, the following approaches are considered: BoW (Bag of Words), TF–IDF, and Char TF–IDF [5].

- **BoW.** Represents a document by the occurrence of words in n-grams, ignoring the positions they occupy within the document. It builds a dictionary/vocabulary from the words in the corpus, assigns an index to each term, and then counts how many times each word appears in the document, storing those counts in the document vector. In scikit-learn, this is implemented by `CountVectorizer`, which "converts a collection of text documents to a matrix of token counts" [5].
- **TF–IDF.** Starts from a principle similar to BoW in the sense that it also produces a sparse matrix representation; however, instead of relying only on raw frequency counts, it incorporates the inverse document frequency (IDF) term, which penalizes terms that appear in many documents [5]. Conceptually, the TF–IDF weight of a term t in a document d is computed as in Eq. (1), where $\mathrm{tf}(t,d)$ corresponds to the (raw) term frequency in d, and $\mathrm{idf}(t)$ assigns lower weights to terms that are widely distributed across the corpus.

$$\mathrm{tfidf}(t,d) = \mathrm{tf}(t,d) \times \mathrm{idf}(t). \tag{1}$$

In practice (and as implemented in common libraries such as scikit-learn), a smoothed version of IDF is typically used, defined in Eq. (2), where n is the total number of documents in the corpus and $\mathrm{df}(t)$ is the number of documents that contain the term t.

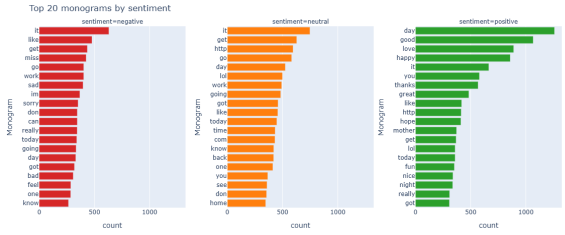
$$\mathrm{idf}(t) = \log\left(\frac{1+n}{1+\mathrm{df}(t)}\right) + 1. \tag{2}$$

Finally, after computing TF–IDF, it is common to normalize each document vector to control for differences in document length and to stabilize the scale of the features. In this work, ℓ_2 normalization is considered, as shown in Eq. (3), where $\mathbf{v}$ denotes the TF–IDF vector of a document.

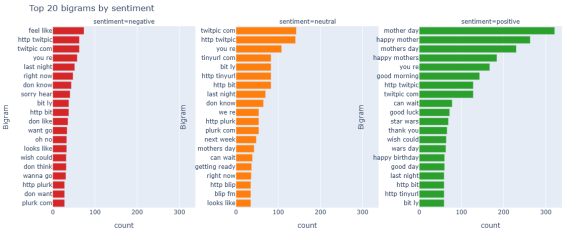
$$\mathbf{v}_{\mathrm{norm}} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}}. \tag{3}$$

- **Char–IDF.** Consists of applying TF–IDF over character n-grams rather than word n-grams, which in scikit-learn can be controlled through the `analyzer` parameter of the vectorizer [6].

After defining the vector representation schemes for the documents, it is proposed to use BoW to extract the 20 most frequent unigrams and bigrams within the training set. In the case of TF–IDF, the objective is to display the 20 unigrams and bigrams with the highest weights in the training dataset. The use of character n-grams is not proposed for this step because, by not forming complete words, the resulting features are less intelligible for the intended analysis.



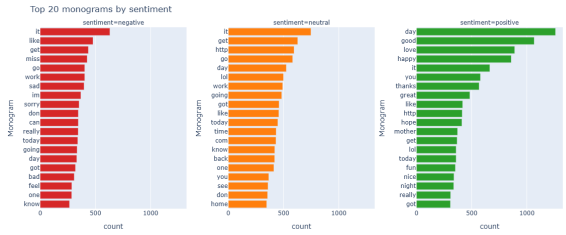
(a) Monograms



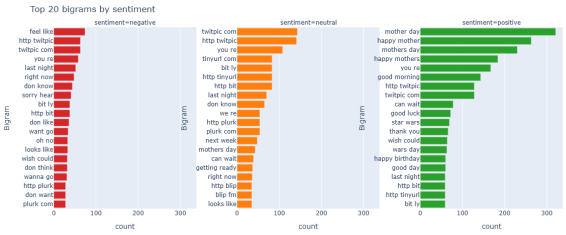
(b) Bigrams

Fig. 2: Top 20 n-gramas with BoW.

By analyzing the results shown in Figs. 2 - 3, it can be observed that some n-grams appearing under both methods correspond to words that, in everyday language use, are commonly associated with the polarity class in which they become most representative. This is the case for the unigram *bad* or *sorry* within the negative polarity class, which appears

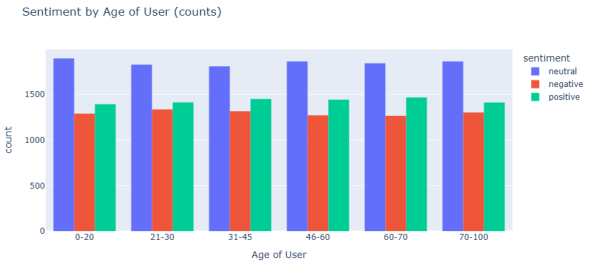


(a) Monograms

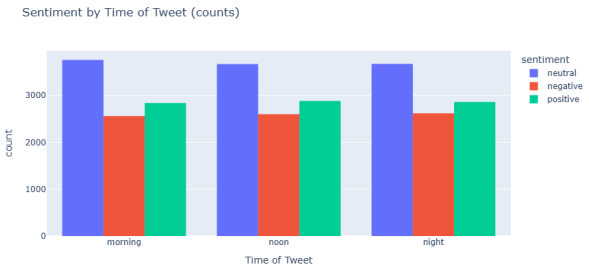


(b) Bigrams

Fig. 3: Top 20 n-gramas with TF–IDF.



(a) Sentiments by User



(b) Sentiments by type of tweet

Fig. 4: Comparative sentiment distribution by category, stratified by user and tweet type

among the top terms for both vectorization methods. Nevertheless, it is evident that, since these are purely statistical approaches, words that were not removed during preprocessing but are very frequent in English usage—such as *it*—may also appear. Similarly, certain bigrams that a speaker might classify as neutral, but that become frequent due to the data-collection context (e.g., *feels like*), appear among the most representative n-grams across multiple classes. This anticipates one of the effects addressed in later stages: training models using frequency-based vectorizations, rather than embeddings that incorporate semantic information into the sparse vector representation of the documents.

In addition to the processed text column, which is the main focus of this work, it is of interest to determine whether some of the other dataset columns exhibit any relationship with sentiment classification. To this end, the class distributions are analyzed as a function of the user's age and the time of day at which the tweet is published, as shown in Fig. 4 However, the class distribution for these categorical variables remains very similar across their respective domains in both cases; therefore, they are not considered relevant for training the sentiment classification model.

In the case of the country in which the tweet is published, a distribution of the countries with the highest number of posts is analyzed in Fig. 5. It is observed that the tweet counts do not vary in a markedly representative way among them, and although the class distributions are slightly different across countries, this feature is not included in later stages. One reason is that the model may "memorize" country-specific patterns that do not hold outside the dataset or that change over time. Additionally, given the shape of the country distribution, many countries have only a small number of examples, which generates rare features and increases the probability of overfitting. For these reasons, this column is ultimately not included in the analysis.

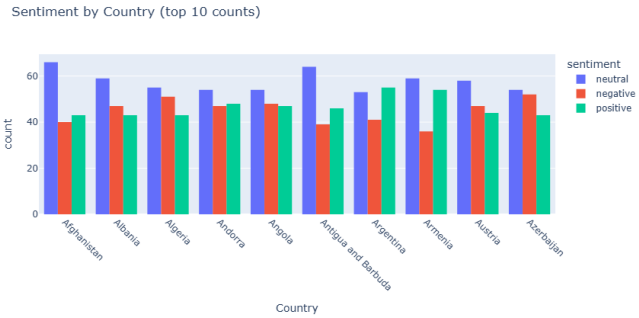


Fig. 5: Distribution of sentiments in the top 10 countries.

Finally, as a last analysis of the dataset, it is proposed to examine whether the length of the preprocessed text has any relationship with the class, and thus whether it could be worth including it as a variable during the training of the classification models. However, as shown in Fig. 6, the distribution of the number of words per message for each class is comparable (or effectively equivalent). For this reason, this variable is also not considered relevant for the objective of the models constructed in the next stage.

B. Classic model training

After the analysis and description of the dataset presented in the previous section, it is proposed to implement traditional machine-learning models that allow the classification

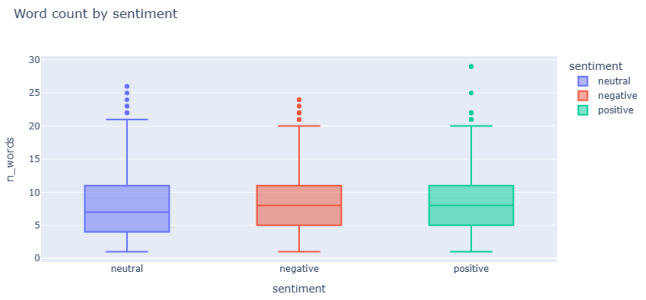


Fig. 6: Word count distribution by sentiment

of received and processed text messages into each of the three polarity (sentiment) categories. Within this work, the concept of classic machine learning models refers to supervised approaches that are not deep learning and do not rely on transformers or embeddings; instead, they operate on an explicit text representation obtained from the effect of one of the previously mentioned vectorizers. This enables following the workflow: document representation, classifier training, evaluation [7].

Within this framework, and considering the variability associated with each of the vectorizers described previously, it is proposed that, for each text representation, a set of classifiers is trained using classic machine-learning algorithms: Multinomial Naive Bayes, Logistic Regression, Support Vector Machine, and Random Forest. However, given the variability of the hyperparameters associated with each classifier and with the vectorizer itself, a hyperparameter search is performed over a pipeline composed of the vectorizer and the selected model, and evaluated via stratified cross-validation. The approach compares the performance obtained for each candidate hyperparameter configuration across the cross-validation folds, and selects the configuration that maximizes a chosen metric. Finally, the best-performing pipeline is refit on the full training set using the selected hyperparameter setting.

In order to adequately describe the operation of the training algorithm that is employed, it is first proposed to detail the influence of each of the hyperparameters considered for every element of the pipeline, both in the case of the vectorizers and in the case of the trained models.

It is convenient to begin with the hyperparameters associated with the vectorizer within the pipeline. The first of these is `ngram_range`, which is a parameter defined as an array of tuples indicating the size of the n-grams that are considered in the construction of the vector representation. These n-grams correspond to words in the case of BoW, to word n-grams in the case of word-level TF–IDF, and to character n-grams in the case of character-level TF–IDF. The `min_df` parameter, in turn, contains a set of minimum thresholds to be evaluated for the document-frequency criterion, that is, the minimum number of documents in which an n-gram must appear in order to be considered. This is useful for removing very rare terms from the representation. The `max_df` parameter specifies the maximum document frequency an n-gram can have to be included in the vectorization, which is useful for removing n-grams that are too frequent and therefore provide limited discriminative information. Finally, `max_features` contains a set of candidate values, where each value corresponds to a limit on the number of features that is tested during the hyperparameter-search phase in order to constrain the vocabulary size, mainly as a function of frequency and computational cost. This limitation is particularly important to adapt the training of the neural networks that are preceded by character-level vectorization in Question Q2, as will be detailed later.

On the other hand, the hyperparameters associated with each model are specific to each algorithm, depending on its nature and mathematical formulation, and they are described below:

- **Multinomial Naive Bayes** `alpha` is an additive smoothing parameter whose main objective is to avoid zero probabilities in the case where a feature does not appear in the training data. Recall that Naive Bayes is formulated as a product of conditional probabilities under an independence assumption; in this context, smoothing works by artificially increasing the observed counts by the value of α [8].
- **Logistic Regression** `C` is a parameter that represents the inverse of the regularization strength: smaller values imply stronger regularization and, therefore, a lower risk of overfitting, whereas larger values imply weaker regularization and a model with more freedom in its parameters. The `penalty` option defines the type of regularization applied by the model: $L1$ penalizes the sum of absolute values of the coefficients, whereas $L2$ penalizes the sum of squared coefficients, which tends to shrink weights more smoothly and does not explicitly encourage sparse solutions where some coefficients become exactly zero. The `solver` parameter refers to the optimization algorithm used during training, and `max_iter` indicates the maximum number of iterations allowed for the solver to converge [9].
- **Support vector machine** The `C` parameter acts as the inverse of the regularization strength. The `estimator__loss` hyperparameter is used to define the loss function of the linear SVM. A `max_iter` parameter is also included to set the maximum number of iterations allowed for convergencec. In addition, the SVM is wrapped within `CalibratedClassifierCV`, which converts the output of the Support Vector Machine into class probabilities, making the predictions more interpretable and, therefore, more suitable for the pipeline hyperparameter-tuning procedure [11].
- **Random forest** `n_estimators` specifies the number of trees (estimators) that are trained in the forest. The `max_depth` parameter defines the maximum depth of each tree; when it is set to `None`, the tree is allowed to grow until its leaves satisfy the stopping requirement defined by `min_samples_leaf`. In contrast, when a

finite value is used, the number of levels in the tree is restricted, reducing the number of nodes and acting as a regularization mechanism that can decrease the risk of overfitting. The `min_samples_split` parameter sets the minimum number of samples required to split an internal node; it can also be used to control overfitting, and therefore it may have a regularization effect as well. Finally, `class_weight` is used to define class weights during training: `None` uses equal weights, while `balanced` adjusts weights as a function of the class frequencies in the training data in order to counteract class imbalance [12].

With the set of defined parameters and a specific pipeline composed of a vectorizer and a supervised learning model, a parameter grid is constructed. From this grid, the candidate combinations are enumerated and the performance of each combination is estimated via 5-fold cross-validation, aggregating the metrics across folds in order to obtain a more robust representation of the estimated model. Once the grid-search algorithm identifies an optimal configuration, the selected pipeline is retrained on the full training set. It is worth emphasizing that this design also includes the possibility of enabling or disabling parallelism in each training run, depending on the specific requirements of each execution of the training algorithm.

After concluding the analysis of the method, the training of all pipelines composed of the referenced vectorizers and models is proposed by using the hyperparameter search algorithm, and the results of this process are discussed in the following section.

III. Q1-Classification with Classical Models and Performance Analysis

A total of 12 experimental configurations were trained (3 vectorization schemes $\times$ 4 models). The goal of this stage is to compare the performance of these combinations on the test set and to define a strong classical baseline, which will later serve as a reference point against embedding-based methods and BERT-like models.

The reported metrics are Accuracy, Macro-Recall, and Macro-F1. Macro metrics are particularly relevant because they evaluate performance more evenly across classes, reducing the risk that a dominant class drives the global evaluation.

A. Training time

Each experiment trained with `GridSearchCV` produces a CSV file containing the full detail of all evaluated combinations, including hyperparameters for both the vectorizer and the classifier. These CSV files enable a systematic analysis of configuration changes and an informed selection of the final best-performing pipeline.

In these files, the `mean_fit_time` field represents the average training time per fold for each evaluated configuration. Using this value, it is possible to estimate the overall compute time associated with hyperparameter search.

1) Total time estimate (sequential compute): The sequential total-time estimate is computed as the sum of the average training time per fold for each candidate configuration, multiplied by the number of cross-validation folds:

$$T_{\text{total}} \approx \sum_{i=1}^{n_{\text{candidates}}} \left(\texttt{mean\_fit\_time}_i \times n_{\text{splits}}\right), \qquad n_{\text{splits}} = 5. \tag{4}$$

Here, $n_{\text{candidates}}$ corresponds to the number of rows in the CSV, i.e., the number of hyperparameter combinations evaluated, being the total accumulated:

- $n_{\text{candidates}} = 968$
- $T_{\text{total}} \approx 287{,}978.76$ s

2) Wall-clock time with parallelization: The value above corresponds to a theoretical sequential compute time (i.e., a single-thread estimate). However, since parallelization was enabled with `n_jobs=-1`, the workload is distributed across multiple CPU cores. Therefore, an approximate wall-clock time can be estimated as:

$$T_{\text{wall}} \approx \frac{T_{\text{total}}}{n_{\text{jobs}}}. \tag{5}$$

Assuming 8 effective cores, the wall-clock estimate becomes:

$$T_{\text{wall}} \approx \frac{287{,}978.76}{8} \approx 35{,}997.35 \text{ s} \approx 10.00 \text{ h}.$$

B. Test-set performance

Test performance is summarized with three comparative plots: Macro-F1, Macro-Recall, and Accuracy, referenced as Fig. 7, Fig. 8, and Fig. 9, respectively. These plots show, for each classifier, the performance obtained under each vectorization method, allowing robust trends to be identified immediately.

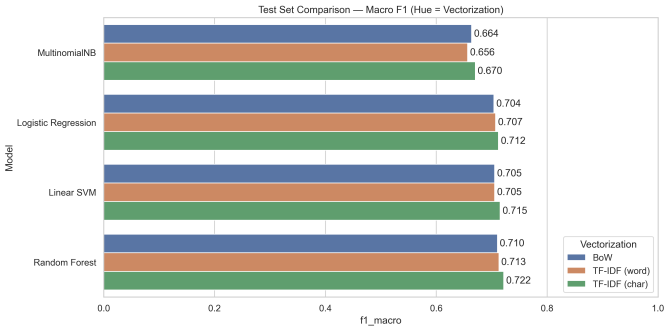


Fig. 7: Test comparison — Macro-F1 (hue = vectorization).

1) Impact of the vectorization method: Overall, the figures indicate that character-level TF–IDF (TF–IDF char) tends to yield the best values across all three metrics. This behavior is consistent with the nature of the dataset (short, informal tweets), where character-based modeling captures important sub-lexical patterns such as spelling errors, morphological variations, elongations (e.g., "soooo good"), hashtags, abbreviations, and prefix/suffix cues.

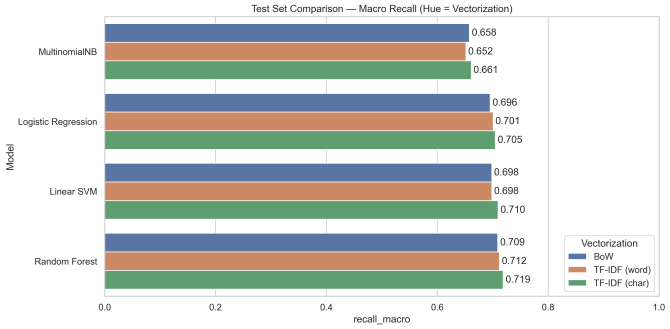


Fig. 8: Test comparison — Macro-Recall (hue = vectorization).

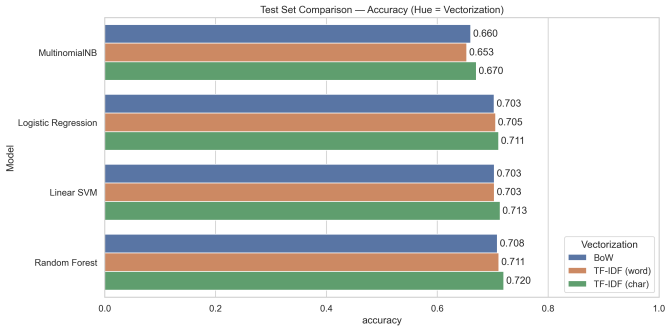


Fig. 9: Test comparison — Accuracy (hue = vectorization).

In contrast, BoW often exhibits the lowest performance because it relies on raw counts without weighting terms by their global relevance. Word-level TF–IDF improves upon BoW, but remains less robust to orthographic noise and non-standard language usage compared to character n-grams.

2) Model comparison and trade-offs: Among classifiers, Linear SVM and Logistic Regression stand out for their consistency and stability, especially when combined with TF–IDF (particularly TF–IDF char). This is expected because BoW/TF–IDF representations produce very high-dimensional and sparse matrices, a regime where linear models are typically both effective and computationally efficient.

Although Random Forest achieves very competitive results (and can even reach the best score), several drawbacks are important in high-dimensional text settings:

- **Scalability and computational cost:** training many trees in large TF–IDF spaces increases time and memory usage substantially, especially under `GridSearchCV`.
- **Overfitting risk with highly specific features:** TF–IDF char introduces many highly particular n-grams; a non-linear model such as Random Forest may capture accidental training patterns and reduce out-of-domain robustness.
- **Lower practical interpretability:** while a single tree is interpretable, a forest with many estimators is harder to justify transparently. In contrast, linear models allow a more direct analysis of discriminative signals.

Therefore, even if Random Forest yields a small absolute gain, linear models remain a strong choice due to stability, cost, and methodological clarity.

3) Confusion matrix analysis (TF–IDF char): To analyze the behavior of the best linear models on the test set, we examine the confusion matrices for character-level TF–IDF with Linear SVM and Logistic Regression. This analysis reveals not only how many predictions are correct, but also which types of errors are most frequent and which classes are most often confused in this three-class setting (negative, neutral, positive).

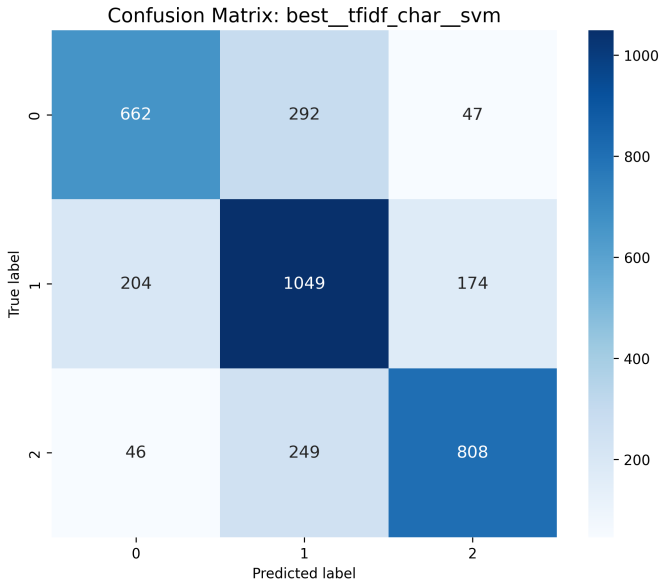


Fig. 10: Confusion matrix of `best__tfidf_char__svm`.

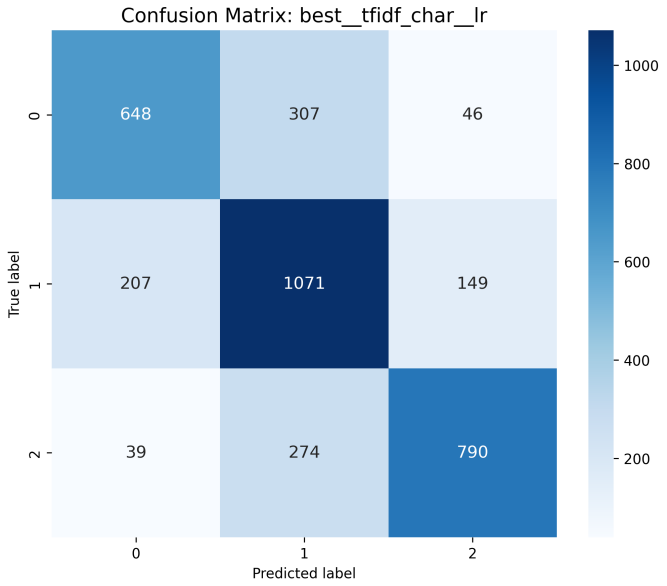


Fig. 11: Confusion matrix of `best__tfidf_char__lr`.

a) General pattern: the "neutral" class is the main source of ambiguity.: In both matrices, most errors concentrate

around the neutral class, which is expected in sentiment analysis of short tweets. Many messages contain weak emotional cues, irony, abbreviations, or context-dependent fragments, which makes it difficult to separate neutral from mildly positive or negative texts.

Concretely, the most frequent errors correspond to polarized examples being "absorbed" into neutral:

- Negative → Neutral: SVM = 292, Logistic Regression = 307
- Positive → Neutral: SVM = 249, Logistic Regression = 274

This pattern suggests that, when polarity is not strongly marked, the model tends to assign neutral as a "safe" option. Practically, the hardest boundary is not Negative vs. Positive, but rather Neutral vs. (Negative/Positive).

b) Good sign: low direct confusion between "negative" and "positive".: A relevant result is that direct confusion between the polarity extremes (Negative ↔ Positive) is relatively low compared to confusions toward neutral:

- Negative → Positive: SVM = 47, Logistic Regression = 46
- Positive → Negative: SVM = 46, Logistic Regression = 39

This indicates that when strong linguistic markers exist, the models rarely invert sentiment polarity. Thus, clear sentiment signals are captured correctly; the main challenge lies in moderate or ambiguous cases where emotion is weak or context-dependent.

c) Subtle differences between SVM and Logistic Regression.: Although both models achieve very similar overall performance, the matrices suggest slightly different behavior regarding the neutral class:

- Logistic Regression predicts "neutral" more often than SVM (stronger attraction toward neutral).
- This is reflected by slightly higher Negative → Neutral and Positive → Neutral confusions for LR.

In trade-off terms:

- SVM tends to keep polarized classes slightly better separated (a small advantage in recovering positives and negatives).
- LR tends to favor the neutral class, which can increase the number of polarized examples classified as neutral.

This is consistent with the nature of the task: neutral is a broad and less semantically-defined class, so small decision-boundary shifts can produce noticeable changes in confusion patterns.

C. Top-20 and Bottom-20 character n-grams per class (LR vs. SVM)

1) Positive class: For the Positive class, the most influential character n-grams are highly consistent across LR and SVM. The Top-20 n-grams are shown in Fig. 12 and Fig. 13. Typical approval fragments such as *love/lov*, *nice*, *good/goo*, *fun*, *best*, *amaz*, *awes*, *yum*, *cool*, and variants of *thank* appear prominently, confirming the benefit of character-level modeling:

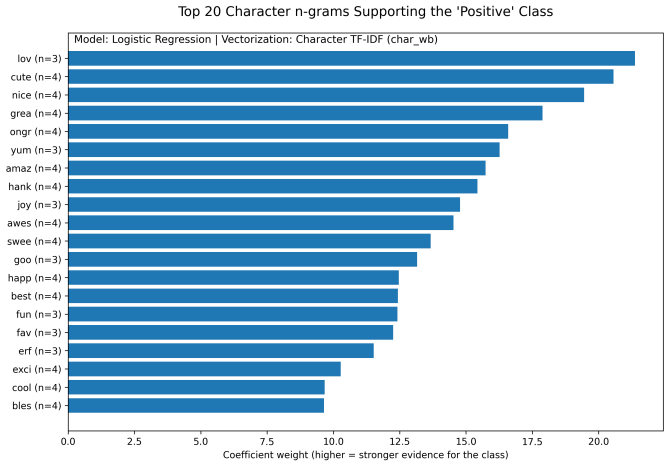


Fig. 12: Positive class: Top-20 character n-grams (Logistic Regression) under TF–IDF char (`char_wb`).

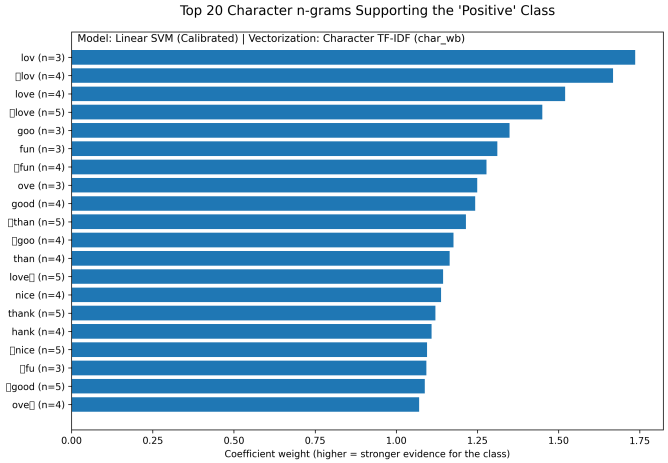


Fig. 13: Positive class: Top-20 character n-grams (Linear SVM) under TF–IDF char (`char_wb`).

even incomplete or abbreviated words still preserve strong sentiment cues.

Looking at the Bottom-20 (Fig. 14 and Fig. 15), the strongest penalties for the Positive class are clear negativity markers such as *sad*, *hate/hat*, *bad*, *suck*, and negations (e.g., *no*, *n't*). In practice, predicting "Positive" not only requires positive evidence but also the absence of strong negative cues.

2) Negative class: For the Negative class, the Top-20 n-grams (Fig. 16 and Fig. 17) reflect discomfort and complaint (*sad*, *hate/hat*, *bad*, *hurt*, *wors*, *sick*, *poor*, and intense fragments such as *uck*). Their presence in both LR and SVM indicates that the negative class is well-defined even when words are abbreviated or incomplete.

Conversely, the Bottom-20 (Fig. 18 and Fig. 19) shows that strong positive cues such as *lov/love*, *thank/than*, *hope*, or *good* penalize the Negative class, demonstrating that the models capture both negative evidence and strong counter-evidence pushing toward Positive.

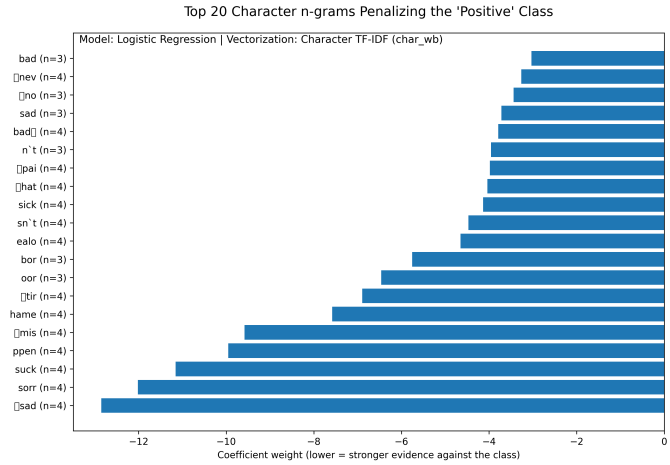


Fig. 14: Positive class: Bottom-20 character n-grams (Logistic Regression) under TF–IDF char (`char_wb`).

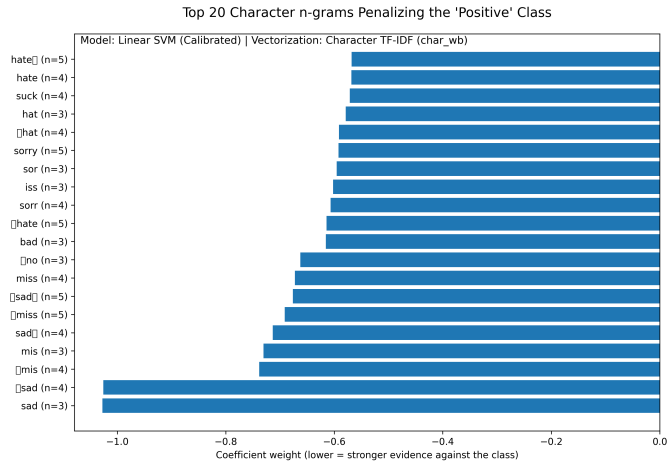


Fig. 15: Positive class: Bottom-20 character n-grams (Linear SVM) under TF–IDF char (`char_wb`).

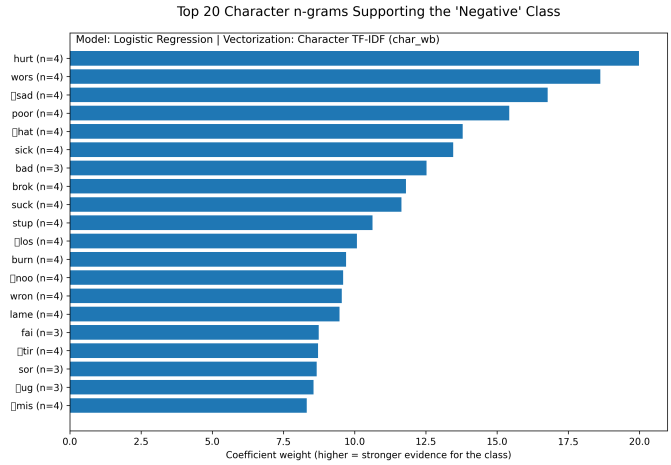


Fig. 16: Negative class: Top-20 character n-grams (Logistic Regression) under TF–IDF char (`char_wb`).

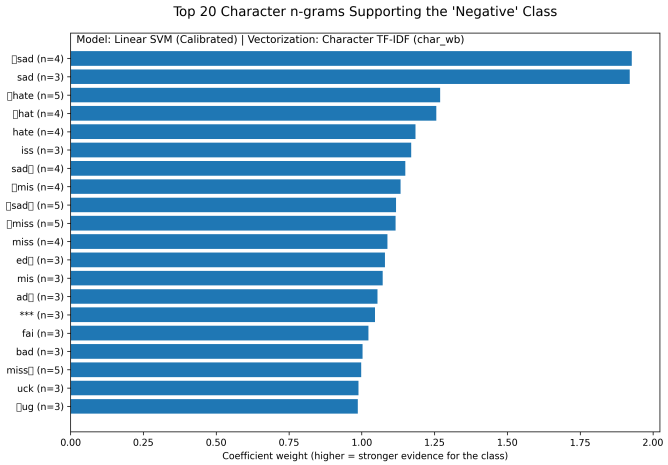


Fig. 17: Negative class: Top-20 character n-grams (Linear SVM) under TF–IDF char (`char_wb`).

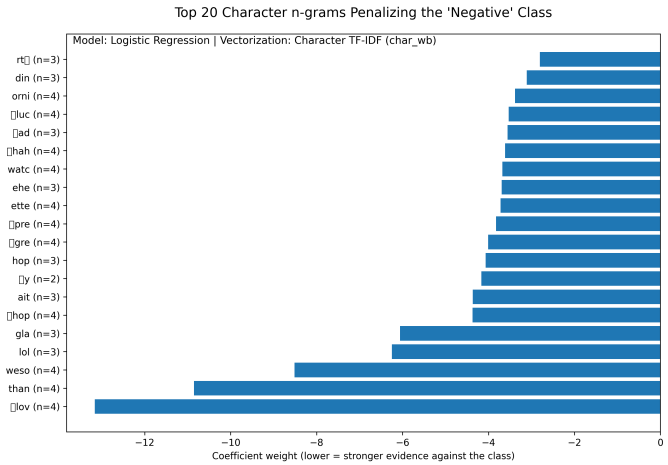


Fig. 18: Negative class: Bottom-20 character n-grams (Logistic Regression) under TF–IDF char (`char_wb`).

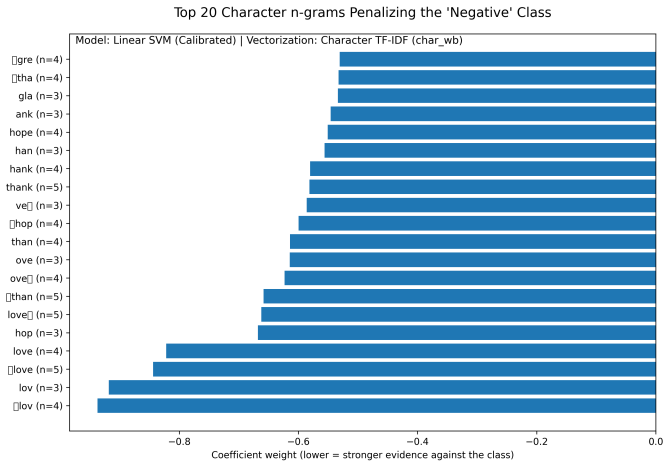


Fig. 19: Negative class: Bottom-20 character n-grams (Linear SVM) under TF–IDF char (`char_wb`).

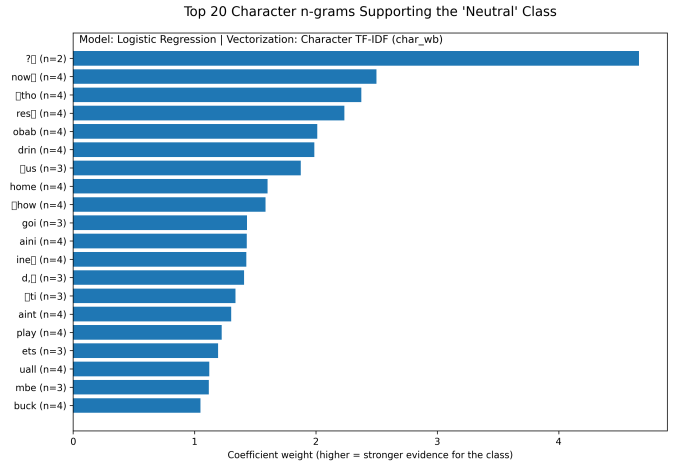


Fig. 20: Neutral class: Top-20 character n-grams (Logistic Regression) under TF–IDF char (`char_wb`).

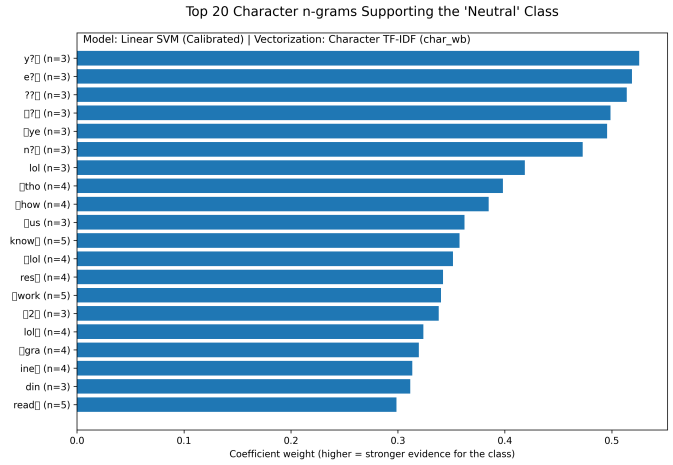


Fig. 21: Neutral class: Top-20 character n-grams (Linear SVM) under TF–IDF char (`char_wb`).

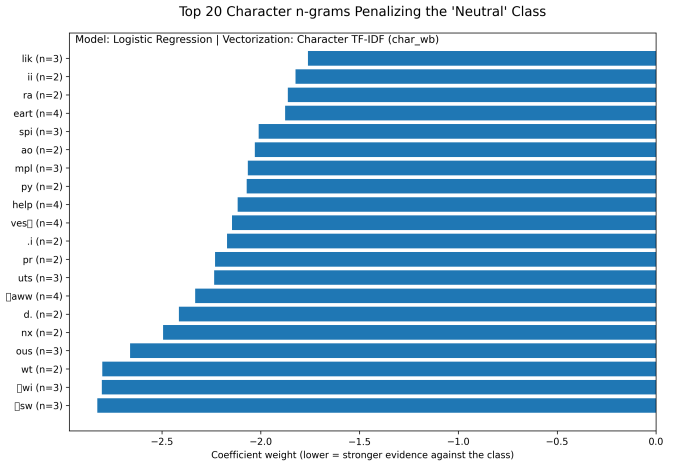


Fig. 22: Neutral class: Bottom-20 character n-grams (Logistic Regression) under TF–IDF char (`char_wb`).

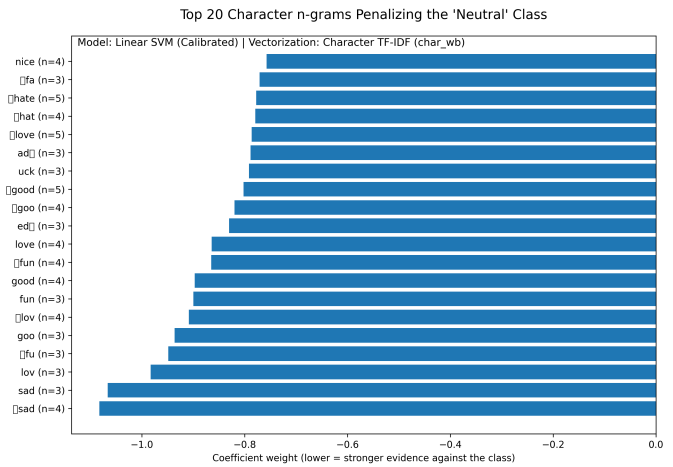


Fig. 23: Neutral class: Bottom-20 character n-grams (Linear SVM) under TF–IDF char (`char_wb`).

3) Neutral class: Neutral is the most interesting class to interpret. The Top-20 n-grams (Fig. 20 and Fig. 21) tend to be more structural than sentimental (e.g., punctuation such as question marks and discourse connectors), suggesting that the model leverages patterns typical of informative statements or questions.

In the Bottom-20 (Fig. 22 and Fig. 23), Neutral is penalized when clear polarity signals appear, whether positive (*love/lov*, *good*, *fun*, *nice*) or negative (*sad*, *hate*, etc.). In other words, Neutral is largely defined by the absence of strong sentiment cues, which helps explain why it is often the most difficult class.

D. Synthesis and practical considerations

After comparing all trained configurations, the most robust and consistent results are obtained with character-level TF–IDF (TF–IDF char) and linear models. The two best approaches (in terms of the balance across Accuracy, Macro-Recall, and Macro-F1) are:

- Linear SVM + TF–IDF char
- Logistic Regression + TF–IDF char

Although Random Forest can achieve a slightly higher score (see Fig. 7), the metric gap is not large enough to justify its disadvantages in this context (higher computational cost, higher overfitting risk, and lower interpretability). For these reasons, we select TF–IDF char + Linear SVM as the primary classical baseline, with TF–IDF char + Logistic Regression as a very close alternative.

IV. Q2-MLP Architectures for Short-Text Sentiment Classification

In this section, it is proposed to employ multilayer perceptron (MLP) architectures for training classifiers that efficiently identify polarity in short English texts. In order to define the structure and training process of the set of MLPs that are implemented, it is proposed to first discuss the concrete pipeline that enables obtaining a final classification.

The first element of the pipeline is the vectorizer, which can be of any of the three types previously considered. In order to define the hyperparameters of each vectorizer, the hyperparameter setting associated with the vectorizer is selected from the parameter grid of the best-performing pipeline that includes that specific vectorizer.

In the case of the selected vectorizers for word-level TF–IDF and BoW, the number of resulting features is 6689. However, for character-level TF–IDF, as expected given the definition of its n-grams, the number of features exceeds 100,000 when no explicit limit is imposed. This makes the training of an MLP within this pipeline significantly more expensive in terms of GPU and RAM, and it would also require defining network structures different from those used with the other vectorizers. For this reason, the best-performing hyperparameter configuration for the character-level TF–IDF vectorizer is retained, but its number of features is restricted to 10,000 so that its dimensionality is comparable to that of the other vectorizers. This enables implementing equivalent MLP structures across pipelines, differing only in the input dimension of the first layer while keeping the same number of hidden layers, in a way that remains consistent.

The next element in the pipeline is the MLP used for classification; however, several components are required for training and defining its structure. The first of these is an object that converts the sparse matrix produced by the vectorizer into a matrix in Compressed Sparse Row (CSR) format, enabling efficient row-wise access, namely `SparseBoWDataset` [13].

This CSR matrix is then used to construct a `DataLoader` a PyTorch iterator that, at each training epoch, traverses the training set and yields a batch of samples per iteration. Before each epoch, the samples are shuffled in order to reduce the risk of bias due to ordering effects. In this setup, `collate` acts as the function that converts the sparse batch matrices into dense matrices and then into PyTorch tensors, returning a dictionary of the form $\{$`"x"`: `X_batch`, `"label"`: `y_batch`$\}$, which is directly usable for the forward pass and for computing the loss [14].

After defining these components, the discussion can be completed regarding the proposed neural network structure to be implemented. In order to obtain a model that fits both the input requirements and the output classification task, a set of four moderately deep networks is proposed (3 or 4 hidden layers), while maintaining a common output head for the final three-class classification. The decision to limit depth is motivated by the fact that the input representations (BoW/TF–IDF/char or TF–IDF) already condense a large portion of the discriminative information at the feature level; therefore, excessively increasing the number of layers can increase model variance and promote overfitting without a proportional gain in generalization.

In this way, the differences among the proposed network structures focus on two main elements in the search for improved performance. The first is the modification of the layers according to a funnel-shaped architecture, and the second is the adjustment of the regularization strategy by varying the dropout. With this in mind, it is proposed to analyze in more detail each of the structures that are proposed and implemented for each type of vectorization. In this way, the differences among the proposed network structures focus on two main elements in the search for improved network performance. The first is the modification of the layers according to a funnel-shaped architecture, and the second is the adjustment of the regularization strategy by varying the dropout. With this in mind, it is proposed to analyze in more detail each of the structures that are proposed and implemented for each type of vectorization. In each case, the network input is organized as a matrix whose size depends on the batch size, defined consistently across all neural networks as 128, and on the input dimension, which corresponds to the number of features produced by the selected vectorization method. Therefore, for the word-based models this dimension is 6689, while for the character-based model it is 10,000.

- **MLP_1024_512_256_drop0_3**
  - **Input layer (reshape).**
  - **Layer 1** (`Linear INPUT_DIM` $\rightarrow$ 1024): transforms the vector representation into a dense representation of 1024 units.
  - **Activation 1**: ReLU.
  - **Regularization 1**: Dropout ($p = 0.3$).
  - **Layer 2** (`Linear` 1024 $\rightarrow$ 512): compresses the representation to 512 units.
  - **Activation 2**: ReLU.
  - **Regularization 2**: Dropout ($p = 0.3$).
  - **Layer 3** (`Linear` 512 $\rightarrow$ 256): reduces the representation from 512 to 256 units.
  - **Activation 3**: ReLU.
  - **Regularization 3**: Dropout ($p = 0.3$).
  - **Output layer** (`Linear` 256 $\rightarrow$ `NUM_CLASSES`): projects the final representation to 3 classes, corresponding to the sentiment/polarity categories of the short messages and, therefore, to the class decision.
- **MLP_2048_1024_512_drop0_2_gelu**
  - **Input layer (reshape).**
  - **Layer 1** (`Linear INPUT_DIM` $\rightarrow$ 2048): projects the vectorized input into a 2048-dimensional space, increasing capacity to capture feature combinations compared to the previous structure.
  - **Activation 1**: GELU.
  - **Regularization 1**: Dropout ($p = 0.3$).
  - **Layer 2** (`Linear` 2048 $\rightarrow$ 1024): intermediate compression to 1024 units.
  - **Activation 2**: GELU.
  - **Regularization 2**: Dropout ($p = 0.2$).
  - **Layer 3** (`Linear` 1024 $\rightarrow$ 512): additional compression to a 512-dimensional space.
  - **Activation 3**: GELU.
  - **Regularization 3**: Dropout ($p = 0.2$).
  - **Output layer** (`Linear` 512 $\rightarrow$ `NUM_CLASSES`): produces the final logits.
- **MLP_1536_768_384_192_drop0_25_SiLU**

  - **Input layer (reshape).**
  - **Layer 1** (`Linear INPUT_DIM` $\rightarrow$ 1536): initial projection to 1536 units, providing high capacity without reaching the maximum expansion of the previous architecture.
  - **Activation 1**: SiLU.
  - **Regularization 1**: Dropout ($p = 0.25$).
  - **Layer 2** (`Linear` 1536 $\rightarrow$ 768): reduction to 768 units, consolidating learned patterns.
  - **Activation 2**: SiLU.
  - **Regularization 2**: Dropout ($p = 0.25$).
  - **Layer 3** (`Linear` 768 $\rightarrow$ 384): reduction to 384 units.
  - **Activation 3**: SiLU.
  - **Regularization 3**: Dropout ($p = 0.25$).
  - **Layer 4** (`Linear` 384 $\rightarrow$ 192): final reduction to 192 units, enforcing a compact representation before the decision layer.
  - **Activation 4**: SiLU.
  - **Regularization 4**: Dropout ($p = 0.25$).
  - **Output layer** (`Linear` 192 $\rightarrow$ `NUM_CLASSES`): outputs logits.
- **MLP_4096_2048_1024_drop0_1_ReLU**
  - **Input layer (reshape).**
  - **Layer 1** (`Linear INPUT_DIM` $\rightarrow$ 4096): high-capacity projection to 4096 units, seeking to maximize the latent space in order to combine a larger number of features.
  - **Activation 1**: ReLU.
  - **Regularization 1**: Dropout ($p = 0.1$).
  - **Layer 2** (`Linear` 4096 $\rightarrow$ 2048): compression to 2048 units while maintaining high capacity.
  - **Activation 2**: ReLU.
  - **Regularization 2**: Dropout ($p = 0.1$).
  - **Layer 3** (`Linear` 2048 $\rightarrow$ 1024): compression to 1024 units as the representation prior to the output.
  - **Activation 3**: ReLU.
  - **Regularization 3**: Dropout ($p = 0.1$).
  - **Output layer** (`Linear` 1024 $\rightarrow$ `NUM_CLASSES`): produces final logits for classification.

For the training of all the proposed MLPs, the cross-entropy criterion is employed. Training is performed for 50 epochs, with a fast stopping criterion of 1×10^{-4} based on the change in training loss between consecutive epochs. In addition, a learning rate `lr` is defined to control the step size of the parameter updates during optimization, which is carried out using the Adam optimizer, independently of the network architecture under consideration. In this way, each network is trained separately by minimizing the loss function and updating the weights through the adaptive stochastic gradient descent method Adam, allowing their performance to be compared under equivalent training conditions.

Training is carried out with an epoch-based loop, as proposed in the course development. During each pass, the `train_loader` (the DataLoader for the training set) provides the input dictionary containing the tensors and their labels, which are transferred to the computing device, in this case the GPU. For each batch, the optimizer is reset and the model output is computed, followed by the corresponding loss value using cross-entropy loss, as defined previously. Subsequently, backpropagation is executed and the model parameters are updated through an `Adam` optimizer step. This procedure is applied for up to 50 epochs, subject to the early stopping criterion, and the trained model is saved at the end either as an in-memory object or as a `.pt` file.

Considering the design and training choices described above, these models are obtained for each of the vectorized representations, and their results are analyzed later. However, the point previously discussed remains relevant: frequency-based vectorization methods provide only a representation derived from a frequency analysis of the document, and they do not account for aspects such as token position within the text or semantic considerations. This motivates the reflection that it may be convenient to implement a denser vector representation that incorporates this conceptual information through word tokenization. In this way, the use of BERT embeddings is considered, and therefore the evaluation of MLP structures within a pipeline in which the first stage is not frequency-based vectorization, but rather an embedding-based vectorization using BERT. The causes and consequences of this implementation are discussed in the specific sections devoted to BERT embeddings, and especially in terms of the implications it introduces with respect to an approach based on raw text.

Changing the vectorization approach implies that the elements used for training the neural networks described previously, as well as the network structure itself, must be modified. For this reason, it is proposed to first describe the change in the DataLoader structure, which, unlike the case of frequency-based vectors, no longer provides the final features but instead yields tokenized text, since the features (embeddings) are generated when the tensors are passed through BERT. In this setup, the `collate` function is not applied because the BERT tokenizer already returns fixed-size, compatible tensors. Thus, the DataLoader outputs `"input_ids"` + `"attention_mask"` + `"label"`, and the embedding is produced by passing these tensors through BERT [15].

This DataLoader is converted into embeddings through an embedding-extraction function, by setting BERT to evaluation mode and executing the forward pass. After that, pooling is applied in order to obtain a single vector per text, and the resulting batches are concatenated. The final output is therefore a dense matrix, which is standardized using a `StandardScaler` that transforms each feature so that, on the training set, the mean is zero and the standard deviation is one. This ensures that the embedding dimensions have a comparable scale, which improves training stability and prevents some dimensions from having a dominant effect due to higher variance [16].

The output of the scaling step is converted into a DataLoader for the MLP that is comparable to the one used when train-

ing the vectorizer-based methods. This DataLoader constructs batches of dense embeddings and their corresponding labels in the form {"x": X_batch, "label": y_batch}, which are then used for training the model and for its subsequent performance evaluation.

Given the embedding-based vectorization process and the role of the DataLoaders at this stage with BERT and the MLP, it is appropriate to discuss the general modifications in the structure of the models trained for the pipelines that use BERT embeddings. The first effect is that the input dimension changes considerably: it goes from more than 6000 input features to a much smaller and fixed input size, which in the base case is 768. BERT produces a dense output matrix containing real-valued activations in almost all dimensions, which is stored as a dense vector; by contrast, frequency-based vectorizations yield sparse representations with many zeros. This difference implies that the MLP design must control capacity (width and depth) in relation to the input space: for sparse, high-dimensional inputs, regularization plays a fundamental role to avoid memorization effects caused by an input space that is very high-dimensional and mostly empty. In the case of BERT, the dense input implies that most of the representational complexity is handled by the BERT forward pass that produces the embeddings, so the subsequent mapping from that informative vector space to the target classes becomes simpler.

Additionally, when using BERT embeddings, the transformations applied during the forward pass already encode syntactic and semantic information in the vector according to the characteristics of the BERT model being used, whereas TF–IDF corresponds to a comparatively superficial representation that mainly reflects frequency analysis. Therefore, for pipelines in which BERT provides the input to the MLP, the representation is already highly informative and the required MLP complexity decreases considerably; in some cases, even a linear head is sufficient to achieve the expected performance [17]. Considering these conditions, the four architectures proposed for the BERT-embedding pipelines are less complex, with fewer layers and smaller widths than those used in the previous pipelines, and they are the following.

1) **LinearHead_baseline (single linear layer)**
   - **Input layer (reshape).**
   - **Final layer** (Linear $768 \rightarrow$ NUM_CLASSES): directly projects the 768-dimensional embedding to the 3 classes (sentiment/polarity), with no hidden layers or intermediate activations.
   - **Output**: corresponds to the final class decision.
2) **MLP_256_64_drop0_2**
   - **Input layer (reshape).**
   - **Layer 1** (Linear $768 \rightarrow 256$): transforms the input representation into 256 units.
   - **Activation 1**: ReLU.
   - **Regularization 1**: Dropout ($p = 0.2$).
   - **Layer 2** (Linear $256 \rightarrow 64$): reduces/compresses the representation to 64 units.
   - **Activation 2**: ReLU.
   - **Regularization 2**: Dropout ($p = 0.2$).
   - **Output layer** (Linear $64 \rightarrow$ NUM_CLASSES): projects the final representation to 3 classes and defines the polarity prediction.
3) **MLP_128_32_drop0_2**
   - **Input layer (reshape).**
   - **Layer 1** (Linear $768 \rightarrow 128$): compresses the vector representation to 128 units.
   - **Activation 1**: ReLU.
   - **Regularization 1**: Dropout ($p = 0.2$).
   - **Layer 2** (Linear $128 \rightarrow 32$): reduces the representation to 32 units to extract more compact patterns.
   - **Activation 2**: ReLU.
   - **Regularization 2**: Dropout ($p = 0.2$).
   - **Output layer** (Linear $32 \rightarrow$ NUM_CLASSES): projects the output to the 3 sentiment classes and corresponds to the final decision.
4) **MLP_512_128_drop0_3**
   - **Input layer (reshape).**
   - **Layer 1** (Linear $768 \rightarrow 512$): expands/transforms the representation into a 512-unit dense space to capture more complex relationships.
   - **Activation 1**: ReLU.
   - **Regularization 1**: Dropout ($p = 0.3$).
   - **Layer 2** (Linear $512 \rightarrow 128$): compresses the representation to 128 units while preserving the most relevant information.
   - **Activation 2**: ReLU.
   - **Regularization 2**: Dropout ($p = 0.3$).
   - **Output layer** (Linear $128 \rightarrow$ NUM_CLASSES): projects the final representation to the 3 classes (sentiment/polarity) and defines the class decision.

For training, cross-entropy loss is defined as the optimization criterion. In addition, a maximum of 50 training epochs is proposed, with a fast stopping criterion of 1×10^{-4}, and a learning rate of 1×10^{-4} is used. The Adam optimizer is defined and applied to all trained networks. It is also important to highlight that the training algorithm used for these MLPs is the same as the one employed for training the other models, which allows their performance to be compared under consistent training conditions. The results of all pipelines described in this section are analyzed in the following section.

V. Q5-BERT EMBEDDINGS VS. RAW REPRESENTATIONS

Although the difference implied by representing raw text through any of the classic frequency-based vectorization methods and the use of BERT embeddings has already been addressed in the MLP training section, it is appropriate to discuss in greater detail the difference between these two forms of numerical representation for textual elements: approaches based on frequency occurrence counts and transformer-based representations.

The first element that stands out among these differences is that BERT is a transformer encoder that is pre-trained with the objective of producing bidirectional contextual representations. This means that the resulting token representations are encoded using both left and right context, that is, the words that appear before and after each token within the tokenized sequence [18].

BERT produces contextualized token-level representations which, through a pooling operation, yield a dense, fixed-size embedding per text. This embedding corresponds to a 768-dimensional vector, which becomes the input dimension of the classifier. This significant difference in dimensionality between using a frequency-based vectorization model and using BERT embeddings is the first differentiating factor that motivates the use of the latter over the traditional approach.

To highlight the effect that this change in dimensionality has on the training of the classifier model, it is proposed to isolate the MLP from the pipeline and analyze the number of parameters of the network as a function of its input, and to quantitatively compare the difference between the size of a network that receives as input a frequency-based vectorizer representation, such as the character-level case used in training, and one that uses BERT embeddings. For this exercise, the parameter-count Eq. (6) for a fully connected layer is used, which consists of a weight matrix and a bias term, and is deduced from the infromation provided by PyTorch [19].

$$params = (infeatures \cdot outfeatures) + outfeatures \quad (6)$$

By applying this equation to a neural network with the structure $\texttt{INPUT\_DIM} \rightarrow 1024 \rightarrow 512 \rightarrow 256 \rightarrow 3$, and using Eq. (7), a total of 10,897,923 parameters is obtained for the MLP, whereas with the same structure but with an input layer adapted to a BERT embedding, as shown in Eq. (8), 1,444,355 parameters are obtained, a reduction of $\approx 86.75\%$. This reduction is highly relevant when pursuing generalization and training stability, since it decreases the risk of overfitting: the relationship between training data and parameters makes memorization of noise or spurious dynamics less likely. Additionally, a model can be obtained that is less sensitive to regularization strategies, and, focusing only on the MLP training, the computational cost of training is reduced, an effect that becomes evident in the training performed during the testing phase detailed in Q2 and Q3.

$$\begin{aligned} P_1\,(10000 \rightarrow 1024) &= 10000 \cdot 1024 + 1024 = 10{,}241{,}024, \\ P_2\,(1024 \rightarrow 512) &= 1024 \cdot 512 + 512 = 524{,}800, \\ P_3\,(512 \rightarrow 256) &= 512 \cdot 256 + 256 = 131{,}328, \\ P_4\,(256 \rightarrow 3) &= 256 \cdot 3 + 3 = 771, \\ P_{\text{total}} &= P_1 + P_2 + P_3 + P_4 = 10{,}897{,}923. \end{aligned} \quad (7)$$

$$\begin{aligned} P_1\,(768 \rightarrow 1024) &= 768 \cdot 1024 + 1024 = 787{,}456, \\ P_2\,(1024 \rightarrow 512) &= 1024 \cdot 512 + 512 = 524{,}800, \\ P_3\,(512 \rightarrow 256) &= 512 \cdot 256 + 256 = 131{,}328, \\ P_4\,(256 \rightarrow 3) &= 256 \cdot 3 + 3 = 771, \\ P_{\text{total}} &= P_1 + P_2 + P_3 + P_4 = 1{,}444{,}355. \end{aligned} \quad (8)$$

The output of frequency-based vectorization models is generally a sparse matrix, which implies that the information is effectively concentrated in only a few active indices. This is directly related not only to the sparsity already mentioned, but also to the implicit presence of null values in the CSR matrix: even if they are not explicitly stored in memory, they expose the model to a considerable sensitivity to network width and to regularization methods, requiring strict control to avoid overtraining. This clearly complicates not only obtaining optimal performance, but also defining the network structure, since subtle variations in layer width can have a substantial effect on the risk of overfitting. In contrast, when using BERT embeddings the matrix is dense and low-dimensional. This typically reduces the size of the downstream classifier needed, and therefore makes the selection of an MLP structure and the choice of regularization parameters easier and less critical. The simplicity in selecting the neural network structure further increases when considering the third key factor proposed to differentiate the advantages of embeddings over classical methods, namely, the sensitivity of each method to context.

Context sensitivity is perhaps the most noticeable and marked difference between the two methods, and it favors the implementation of pipelines with BERT embeddings rather than with frequency-count-based vectorizers. This is because BERT is robustly pre-trained on much larger language corpora and, in addition, it is specifically designed to incorporate context into the vector definition of the words in a document, thanks to the structure and functioning detailed in Section Q4. In contrast, classical vectorization strategies can only capture superficial contextual information obtained through frequency counts or, at best, through the analysis derived from that frequency space, which limits their representation of fundamental semantic aspects that can be decisive when classifying a short message into a sentiment/polarity category. Furthermore, the fact that a BERT embedding is so robust, condensed, and information-rich enables the exploration of simpler models for message classification. That is, in BERT the complexity is mainly concentrated in the pre-training of the model used to obtain the embeddings, and the MLP is isolated from that complexity and only needs to map and classify highly informative vectors. In contrast, when traditional vectorization methods are used, part of the complexity that is covered by BERT pre-training must be handled by the neural network itself, which makes it possible in the BERT case to use networks with fewer layers, or even a single linear classification head, while still achieving results comparable to more complex MLP models trained with traditional vectorization techniques.

Although the benefits of using BERT as an embedding extractor become evident, it is essential to emphasize that using BERT as a frozen embedding extractor is not the same as using it adapted to the task, that is, applying fine-tuning that allows the internal representations to align with the classification objective and with the distribution of the training data. This is of critical importance, for example, in short-text analysis or classification tasks such as the one considered here, where adopting a fine-tuning strategy such as LoRA, which is described in Q6, enables obtaining better classifier performance than using BERT only for frozen embedding extraction. It is through these fine-tuning strategies that the main disadvantage of BERT as a frozen embedding extractor can be mitigated with respect to classical vectors, namely that its representation is external to the domain of the training data. In other words, it is sensitive to context by BERT's own definition, but not to parameters such as the frequency of token presence in the training set and other properties that arise because the transformer training is already fixed and, in frozen embedding extraction, no learning takes place.

It is ultimately concluded that BERT is more attractive than classical methods mainly due to its ability to represent semantics and context, which classical methods do not possess. This is therefore associated with a denser input that requires less complexity in the classification MLP. In addition, the possibility of implementing fine-tuning strategies that counteract the domain-agnostic effect of the embedding extractor's pre-training, and that allow the domain of the training dataset to permeate the document embedding representation, can substantially improve model performance.

VI. Conclusion

This section is intended to establish a set of conclusions regarding the implementation of sentiment classification methods for short English texts. First, it is important to emphasize the analysis carried out in the preliminary phase of the dataset and its main characteristics. From this analysis, it is observed that, given the available training information, the primary variable of interest for training this type of classifier is undoubtedly the processed text after stopword removal, mainly because the class distribution with respect to the other variables included in the dataset either remains constant, as in the case of age ranges, or could imply an increased risk of memorizing incorrect patterns, since the elements of each class within the non-considered category are not representative of the overall compilation, as in the case of the country variable.

In terms of training classical machine-learning models, the classification results achieve high performance, mainly due to the consideration of multiple model pipelines combining a vectorizer with the classifier and the corresponding variations in their hyperparameter grids. This leads to a total of 956 trained models, from which the best-performing configurations are selected. Additionally, across the three types of representations, the best results are obtained for SVM, Logistic Regression, and Random Forest, with slight advantages in terms of F1 and accuracy for Random Forest in most cases. Nevertheless, models such as Logistic Regression and SVM are more intelligible and interpretable; therefore, and considering that the performance differences between Random Forest and Logistic Regression/SVM are not sufficiently significant to indicate that Random Forest is the best classification option for NLP models, it is important to note that, despite its slightly better metrics, Random Forest involves a substantially higher training cost in computation time and resource usage.

Although, in the case of training classical models, the relevance of the vectorization method and its selection is addressed and referenced, the training algorithm with grid search makes it possible to mitigate this effect when selecting a pipeline, since the search over vectorizer parameters is included within the grid-search procedure. However, in the case of MLPs, the smaller number of candidates, the influence of each type of vectorization on the network structure, and the incorporation of frozen BERT embeddings prior to MLP training lead the discussion to consider not only the best model, but rather the best-performing pipeline, including both vectorization and classification.

References

[1] T. Finn and A. Downie, "How can sentiment analysis be used to improve customer experience?," *IBM Think*, accessed Jan. 18, 2026. [Online]. Available: https://www.ibm.com/think/insights/how-can-sentiment-analysis-be-used-to-improve-customer-experience

[2] B. Pang and L. Lee, "Opinion mining and sentiment analysis," *Foundations and Trends in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008.

[3] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," *arXiv preprint* arXiv:2106.09685, Jun. 2021, doi: 10.48550/arXiv.2106.09685.

[4] The NLTK Project, "Sample usage for corpus," *NLTK Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://www.nltk.org/howto/corpus.html

[5] scikit-learn developers, "Feature extraction," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/feature_extraction.html

[6] scikit-learn developers, "sklearn.feature_extraction.text.TfidfVectorizer," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[7] F. Sebastiani, "Machine Learning in Automated Text Categorization," *ACM Computing Surveys*, vol. 34, no. 1, pp. 1–47, Mar. 2002, doi: 10.1145/505282.505283. [Online]. Available: https://nmis.isti.cnr.it/sebastiani/Publications/ACMCS02.pdf

[8] scikit-learn developers, "sklearn.naive_bayes.MultinomialNB," *scikit-learn Documentation*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html. Accessed: Jan. 18, 2026.

[9] scikit-learn developers, "sklearn.linear_model.LogisticRegression," *scikit-learn Documentation*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html. Accessed: Jan. 18, 2026.

[10] scikit-learn developers, "sklearn.svm.LinearSVC," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[11] scikit-learn developers, "sklearn.calibration.CalibratedClassifierCV," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html

[12] scikit-learn developers, "sklearn.ensemble.RandomForestClassifier," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[13] SciPy developers, "scipy.sparse.csr_matrix," *SciPy Documentation*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html. Accessed: Jan. 18, 2026.

[14] PyTorch contributors, "Data Loading and Processing Tutorial," *PyTorch Tutorials*. [Online]. Available: https://docs.pytorch.org/tutorials/beginner/data_loading_tutorial.html. Accessed: Jan. 18, 2026.

[15] Hugging Face, "BERT," *Transformers Documentation*. [Online]. Available: https://huggingface.co/docs/transformers/en/model_doc/bert. Accessed: Jan. 18, 2026.

[16] scikit-learn developers, "sklearn.preprocessing.StandardScaler," *scikit-learn Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[17] Sutriawan, Supriadi Rustad, Guruh Fajar Shidik, and Pujiono, "Performance Evaluation of Text Embedding Models for Ambiguity Classification in Indonesian News Corpus: A Comparative Study of TF-IDF, Word2Vec, FastText, BERT, and GPT," *Ingénierie des Systèmes d'Information*, vol. 30, no. 6, pp. 1469–1482, June 2025, doi: 10.18280/isi.300606. [Online]. Available: https://www.iieta.org/journals/isi/paper/10.18280/isi.300606

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, Jun. 2019, doi: 10.18653/v1/N19-1423. [Online]. Available: https://aclanthology.org/N19-1423/

[19] PyTorch contributors, "torch.nn.Linear," *PyTorch Documentation*, accessed Jan. 18, 2026. [Online]. Available: https://docs.pytorch.org/docs/stable/generated/torch.nn.Linear.html