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.

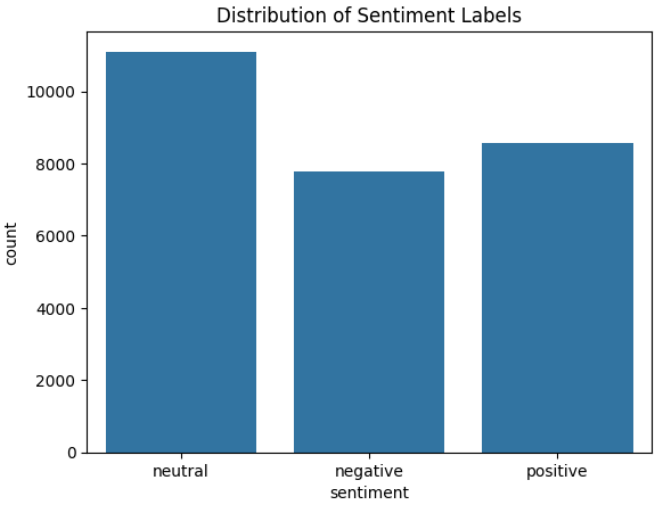


Fig. 1: Distribution of sentiments in training dataset.

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.

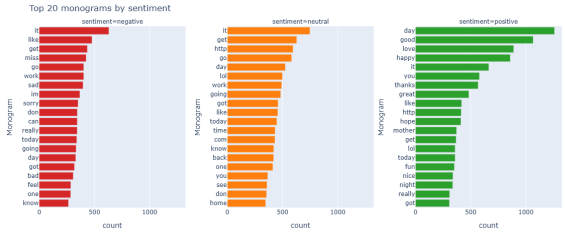
$$\mathrm{idf}(t) = \log\left(\frac{1+n}{1+\mathrm{df}(t)}\right) + 1. \tag{2}$$

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.

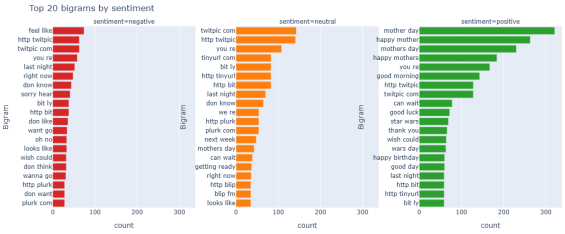
$$\mathbf{v}_{\mathrm{norm}} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}}. \tag{3}$$

- **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.



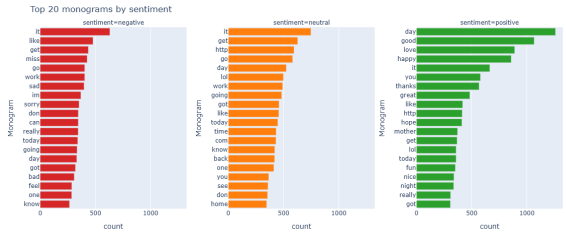
(a) Monograms



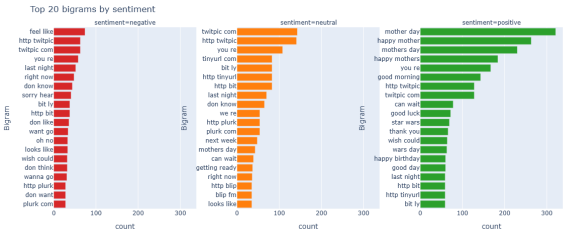
(b) Bigrams

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

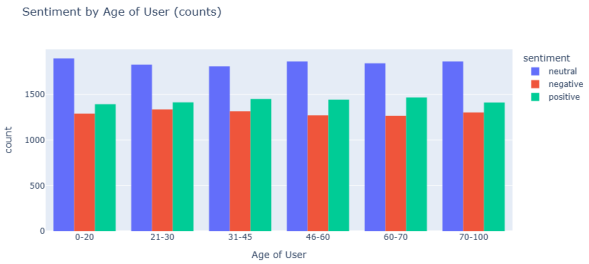


(a) Monograms

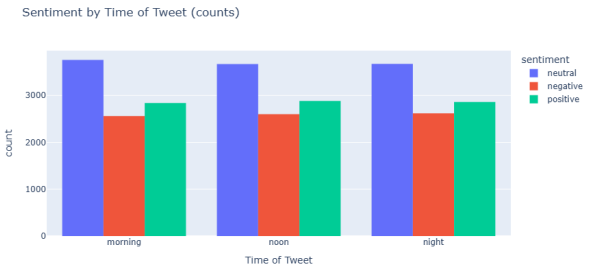


(b) Bigrams

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.

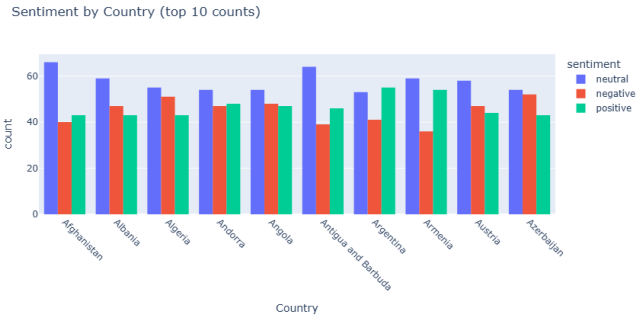


Fig. 5: Distribution of sentiments in the top 10 countries.

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

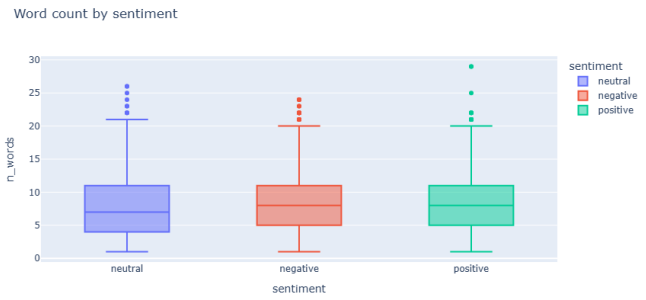


Fig. 6: Word count distribution by sentiment

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. 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.

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.