HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Emoji Prediction on Social Media

Supervisor:

Dr. Nguyễn Bá Ngọc

Students:

Lương Thái Khang - 20224866

Hanoi, January 2026

ABSTRACT

Emoji usage is an important part of modern online communication, but selecting an appropriate emoji can interrupt writing flow. This report presents an end-to-end pipeline for **emoji prediction from text**: (i) constructing a cleaned tweet-style dataset with 43 emoji classes, (ii) benchmarking classical and neural text classifiers, and (iii) deploying the best model for client-side inference in the browser. We compare TF–IDF baselines (linear SVM, logistic regression, and Naive Bayes), FastText-style subword models, TextCNN, bidirectional RNNs with attention, and a fine-tuned DistilBERT Transformer [3, 4, 7]. On the provided test split (83,543 samples), the Transformer achieves the best performance with **top-1 accuracy 26.3%** and **top-5 accuracy 56.6%**. Finally, we export the Transformer to quantized ONNX and integrate it into a static web demo using Transformers.js, enabling fast, privacy-preserving emoji suggestions directly in the browser [6, 10].

Table of Contents

1 INTRODUCTION

1.1 Motivation

Emojis enrich text with emotional nuance, emphasis, and cultural context. On social media and messaging platforms, users often add one or more emojis to convey sentiment (e.g., excitement, sarcasm), summarize a message, or increase engagement. However, manually choosing an emoji requires extra attention and can interrupt writing flow. A lightweight emoji suggestion system can improve user experience by predicting emojis from the text being written.

1.2 Scope of This Repository

This repository focuses on **single-label emoji prediction**: given a short English text (tweet-like sentence), predict exactly one emoji from a fixed set of 43 emojis. The repo includes:

- A cleaned dataset split into train/test CSV files (`data/`).
- A model benchmark suite covering classical baselines and neural architectures (`tune.py`, `train_eval.py`, `ml.py`).
- A Python inference helper for the deployed Transformer model (`infer.py`).
- A browser deployment path based on ONNX + Transformers.js (`scripts/export_onnx.py`, `docs/`).

1.3 Contributions

The main contributions of this work are:

- A practical preprocessing pipeline for tweet-style text with emoji removal and normalization.
- A consistent evaluation framework for top-k metrics over multiple model families.
- A deployment workflow that exports the best Transformer model to quantized ONNX and runs it client-side in a static web application.

1.4 Report Organization

Section 2 reviews related work in text classification and emoji prediction. Section 3 formalizes the task and evaluation metrics. Section 4 describes the dataset and preprocessing. Section 5 presents the models. Section 6 reports experiments and results. Section 7 describes deployment. Finally, Section 8 concludes and discusses limitations and future work.

2 RELATED WORK

2.1 Emoji Prediction and Text Classification

Emoji prediction can be viewed as standard multi-class text classification, closely related to sentiment analysis and emotion recognition in social media text. Classical pipelines typically rely on sparse lexical features (e.g., TF–IDF over word/character n-grams) paired with linear classifiers. These methods are strong baselines for short-text tasks and remain competitive when data is limited [5].

2.2 Neural Models for Text

Neural approaches learn task-specific representations and can better capture compositional patterns beyond surface n-grams:

- **FastText-style models** use word embeddings with subword n-grams to handle misspellings and morphological variations efficiently [3].
- **Convolutional architectures** (TextCNN) extract local n-gram features using 1D convolutions and max pooling, often performing strongly on sentence classification [4].
- **Recurrent encoders** (LSTM/GRU) capture sequential context; attention mechanisms improve pooling by weighting informative tokens more heavily [1].

2.3 Transformers

Transformers replace recurrence with self-attention and dominate many NLP benchmarks [9]. Large pretrained models such as BERT [2] and its compressed variants such as DistilBERT [7] can be fine-tuned for downstream classification tasks with relatively small changes to the model head.

2.4 Client-side Deployment

Running NLP models in the browser is increasingly feasible via ONNX export and WebAssembly backends. ONNX provides an interoperable model format [6], and Transformers.js enables local model loading and inference in JavaScript applications [10]. This repo leverages both to deliver privacy-preserving emoji suggestions without server-side inference.

3 PROBLEM FORMULATION

3.1 Task Definition

Let $\mathcal{E} = \{1, 2, \ldots, C\}$ be the emoji inventory with $C = 43$ candidates. We are given a dataset

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}, \tag{1}$$

where x_i is an input text string and $y_i \in \mathcal{E}$ is the emoji observed with the text. We frame emoji prediction as a **ranking** task: for each query x, the model should produce an ordering over emojis such that relevant emojis appear near the top of the list.

3.2 Model Definition

A model is a function $f : \mathcal{X} \times \mathcal{E} \to \mathbb{R}$ that scores the compatibility between an input text x and an emoji y. Given a query x, the model produces a ranking over emojis by sorting them in descending order of their scores $\{f(x, y) : y \in \mathcal{E}\}$.

3.3 Evaluation Metrics

Because multiple emojis can be plausible for a text, we evaluate the **top-k of the ranking**. Let $\text{Top}^k(x)$ denote the set of k highest-scoring predicted emojis for input x. We use **accuracy at top-k** (Acc@k) as the evaluation metric, defined as the proportion of examples where the ground-truth emoji y_i is among the top-k predictions for input x_i:

$$\text{Acc@}k = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\left[y_i \in \text{Top}^k(x_i)\right]. \tag{2}$$

4 DATASET

4.1 Source and Collection

The dataset is derived from tweet-style text collected using `snscrape`. Data collection was performed by issuing queries per emoji and retaining text samples that contain the queried emoji. English filtering is performed using `pycld3`, inspired by the WiLI language identification benchmark [8]. Because scraped social media text is noisy, some non-English samples and duplicates may remain.

4.2 Cleaning and Normalization

The preprocessing notebook (`data-preprocess.ipynb`) applies a simple cleaning function that:

- Removes URLs, mentions, hashtags, and emojis.
- Lowercases and collapses repeated whitespace.

4.3 Data Samples

Figure 1 shows a few data samples after cleaning.

Text (after cleaning)	Emoji label
happy easter! happy day 3 of our easter egg hunt! can you find our hidden egg today..?	🥚
tried that blonde filter thing on tiktok and yep that's defo the next colour for me	😍
i'm crying laughing at this	😂
gm see you soon ny	☀️
you don't need a man to rescue you. you have your own cape	😎

Figure 1: Example data samples after cleaning.

4.4 Train/Test Split Statistics

The final dataset is stored as two CSV files (`data/train_data.csv`, `data/test_data.csv`). Table 1 summarizes the split.

Table 1: Dataset statistics.

Split	Samples	#Classes	Class count range
Train	751,885	43	16,718–17,846
Test	83,543	43	1,858–1,983

5 MODELS

5.1 TF–IDF Baseline Models

As strong non-neural baselines, we employ linear classifiers trained on sparse TF–IDF representations. Given a document corpus, each text is mapped to a high-dimensional feature vector using a union of:

- word n-grams (e.g., $n \in \{1, 2\}$), which capture lexical content and short phrase-level semantics;
- character n-grams (e.g., $n \in \{3, 4, 5\}$), which are effective for modeling morphological patterns, misspellings, and informal language.

For each n-gram feature t in document d, TF–IDF weighting is computed as

$$\text{tfidf}(t, d) = \text{tf}(t, d) \cdot \log \frac{N}{\text{df}(t)},$$

where $\text{tf}(t, d)$ denotes term frequency, $\text{df}(t)$ is document frequency, and N is the total number of documents. This weighting downscales frequent but uninformative tokens while emphasizing discriminative ones. The resulting sparse vectors are used to train the following classifiers:

Linear Support Vector Machine. The linear SVM learns a maximum-margin hyperplane separating classes in the feature space. By maximizing the margin between decision boundaries and minimizing hinge loss, it tends to generalize well in high-dimensional sparse settings typical of text data.

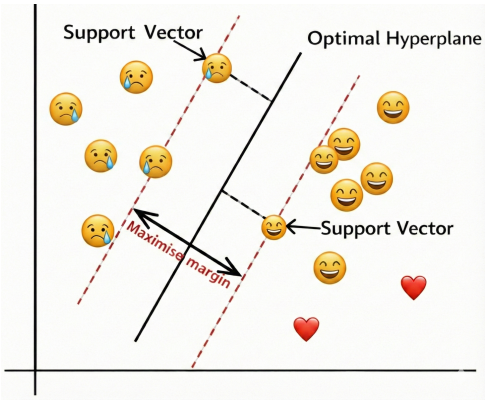

Figure 2: Linear Support Vector Machine.

Logistic Regression. Let each document be represented by a TF–IDF feature vector $\mathbf{x} \in \mathbb{R}^d$ and let there be C classes. Logistic regression defines linear class scores (logits)

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b},$$

where $\mathbf{W} \in \mathbb{R}^{C \times d}$ and $\mathbf{b} \in \mathbb{R}^C$. The conditional probability of class c is computed via the softmax function:

$$p(y = c \mid \mathbf{x}) = \frac{\exp(z_c)}{\sum_{j=1}^{C} \exp(z_j)}.$$

Training typically minimizes the negative log-likelihood (cross-entropy) with ℓ_2 regularization:

$$\mathcal{L}(\mathbf{W}, \mathbf{b}) = -\sum_{i=1}^{N} \log p(y_i \mid \mathbf{x}_i) \; + \; \lambda \|\mathbf{W}\|_2^2.$$

Multinomial Naive Bayes. Multinomial Naive Bayes is a generative probabilistic model over discrete features (typically word or n-gram counts). Let $\mathbf{x} = (x_1, \ldots, x_V)$ be the count vector over a vocabulary of size V. Under the conditional independence assumption,

$$p(\mathbf{x} \mid y = c) = \prod_{j=1}^{V} p(w_j \mid c)^{x_j}.$$

With class prior $p(y = c)$, Bayes' rule yields the posterior (up to normalization):

$$p(y = c \mid \mathbf{x}) \propto p(y = c) \prod_{j=1}^{V} p(w_j \mid c)^{x_j}.$$

In practice, prediction is performed in log-space:

$$\hat{y} = \arg\max_c \left[\log p(y = c) + \sum_{j=1}^{V} x_j \log p(w_j \mid c) \right].$$

The parameters are estimated from training counts using Laplace smoothing ($\alpha > 0$):

$$\hat{p}(w_j \mid c) = \frac{N_{jc} + \alpha}{\sum_{t=1}^{V} N_{tc} + \alpha V}, \qquad \hat{p}(y = c) = \frac{N_c}{N},$$

where N_{jc} denotes the total count of token w_j in class c, N_c is the number of documents in class c, and N is the total number of documents.

5.2 FastText-Style Subword Model

We adopt a FastText-style linear text classifier [3] that combines the efficiency of bag-of-features models with the robustness of subword representations. The key idea is to represent a document by an aggregated embedding of its observed tokens, where each token embedding is augmented with character-level n-gram features. This design greatly reduces the impact of data sparsity and allows the model to generalize to rare spellings and out-of-vocabulary (OOV) words that are common in informal social media text.

Subword decomposition and hashing. Given a token w, we first generate its set of character n-grams $G(w)$, typically for $n \in [n_{\min}, n_{\max}]$ (e.g., 3–6). Each n-gram $g \in G(w)$ is mapped to an integer index using a hashing function

$$h(g) \in \{1, 2, \ldots, B\},$$

where B is the number of hashing buckets. Hashing provides a fixed-size parameterization independent of the vocabulary size, keeping memory usage bounded while still allowing the model to share statistical strength across morphologically related strings. Collisions may occur, but in practice a sufficiently large B yields good performance with modest memory.

Token representation. The embedding of a token is computed by combining a learned word-level vector with the vectors of its subword n-grams. Let $\mathbf{u}_w \in \mathbb{R}^d$ denote the word embedding for token w (when available) and let $\mathbf{v}_{h(g)} \in \mathbb{R}^d$ denote the embedding of the bucket to which n-gram g is hashed. The token representation is

$$\mathbf{x}(w) = \mathbf{u}_w + \sum_{g \in G(w)} \mathbf{v}_{h(g)}.$$

This construction allows the model to form a meaningful vector even when w is rare or unseen: as long as its character n-grams have been observed during training, the subword sum provides a robust estimate of its semantics.

Document representation. For a document D consisting of tokens $(w_1, \ldots, w_T)$, we compute a single document vector by averaging token representations:

$$\mathbf{z}(D) \;=\; \frac{1}{T}\sum_{t=1}^{T}\mathbf{x}(w_t).$$

Averaging yields an order-invariant (bag-of-embeddings) representation that is computationally lightweight: inference and training scale linearly with the number of tokens and the number of extracted subword n-grams.

Linear classifier and training objective. The document embedding is fed into a linear classification layer with parameters $(\mathbf{W}, \mathbf{b})$:

$$\mathbf{s}(D) = \mathbf{W}\mathbf{z}(D) + \mathbf{b},$$

and class probabilities are produced via the softmax function $p(y \mid D) = \mathrm{softmax}(\mathbf{s}(D))$. The model is trained end-to-end by minimizing the cross-entropy loss over the training set:

$$\mathcal{L} \;=\; -\sum_{(D,y)} \log p(y \mid D).$$

Although this model discards word order, the inclusion of character n-grams injects morphological and orthographic information, improving generalization in noisy settings (misspellings, elongations, slang, and code-mixing). Consequently, FastText-style subword models often outperform purely count-based bag-of-words baselines under low-resource conditions while remaining significantly faster and simpler than deep sequence models [3].

5.3 TextCNN

TextCNN [4] applies convolutions over word embeddings to extract local, position-invariant n-gram features for classification. Given an input sentence (or post) tokenized into a sequence $\mathbf{w} = (w_1, w_2, \ldots, w_L)$ of length L, each token w_i is mapped to a dense embedding $\mathbf{x}_i \in \mathbb{R}^d$ via an embedding lookup table $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}|\times d}$. Stacking embeddings yields the sentence matrix:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_L^\top \end{bmatrix} \in \mathbb{R}^{L\times d}. \tag{3}$$

In practice, sequences are padded/truncated to a maximum length (e.g., per-batch maximum or a global $L_{\max}$) to enable batched computation.

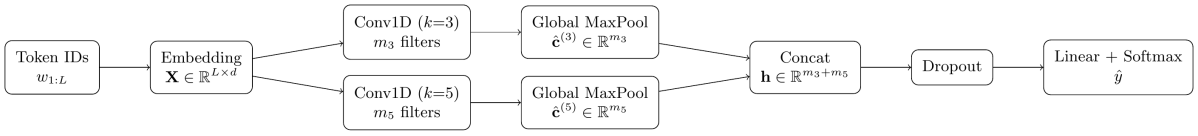

Figure 3: TextCNN architecture with multiple filter widths and global max-over-time pooling.

Convolution over embeddings. To capture patterns of different n-gram lengths, TextCNN uses multiple convolutional filter widths $k \in \mathcal{K}$ (e.g., $\{3, 4, 5\}$). For a given width k, let $\mathbf{W}^{(k,j)} \in \mathbb{R}^{k \times d}$ and $b^{(k,j)} \in \mathbb{R}$ denote the parameters of the j-th filter (out of m_k filters) for that width. The convolution is applied to each contiguous window $\mathbf{X}_{i:i+k-1} \in \mathbb{R}^{k \times d}$:

$$c_i^{(k,j)} = \phi\Big(\langle \mathbf{W}^{(k,j)}, \mathbf{X}_{i:i+k-1} \rangle + b^{(k,j)}\Big), \quad i = 1, \ldots, L - k + 1, \tag{4}$$

where $\langle \cdot, \cdot \rangle$ denotes the Frobenius inner product and $\phi(\cdot)$ is a nonlinearity (typically ReLU). This produces a feature map $\mathbf{c}^{(k,j)} \in \mathbb{R}^{L-k+1}$ for each filter.

Global max-over-time pooling. To convert variable-length feature maps into fixed-dimensional features, TextCNN performs max-over-time pooling:

$$\hat{c}^{(k,j)} = \max_{1 \le i \le L-k+1} c_i^{(k,j)}. \tag{5}$$

Intuitively, each filter learns to detect a particular local pattern, and max pooling retains the strongest activation anywhere in the sequence, making the representation largely insensitive to the pattern's position.

Classification head. All pooled scalars are concatenated into a single vector:

$$\mathbf{h} = \big[\hat{c}^{(k,1)}, \ldots, \hat{c}^{(k,m_k)}\big]_{k \in \mathcal{K}} \in \mathbb{R}^{\sum_{k \in \mathcal{K}} m_k}. \tag{6}$$

We apply dropout to $\mathbf{h}$ for regularization and use a linear layer to predict logits over classes:

$$\mathbf{z} = \mathbf{W}_o \, \mathrm{Dropout}(\mathbf{h}) + \mathbf{b}_o, \qquad p(y = c \mid \mathbf{w}) = \mathrm{softmax}(\mathbf{z})_c. \tag{7}$$

The model is trained end-to-end with cross-entropy loss.
TextCNN is computationally efficient (highly parallelizable) and effective at capturing local syntactic/semantic cues useful for text classification. However, its receptive field is limited to the largest filter width and it does not explicitly model long-range dependencies beyond what can be inferred from local cues and pooling.

5.4 Bidirectional RNN with Attention

Recurrent neural networks (RNNs) explicitly model sequential structure by processing tokens in order. This inductive bias is useful for short, informal social media text where meaning can depend on word order (e.g., negation, intensifiers, and short multiword expressions). We consider two gated RNN variants—LSTM and GRU—and use a bidirectional encoder so that each token representation incorporates both left and right context.

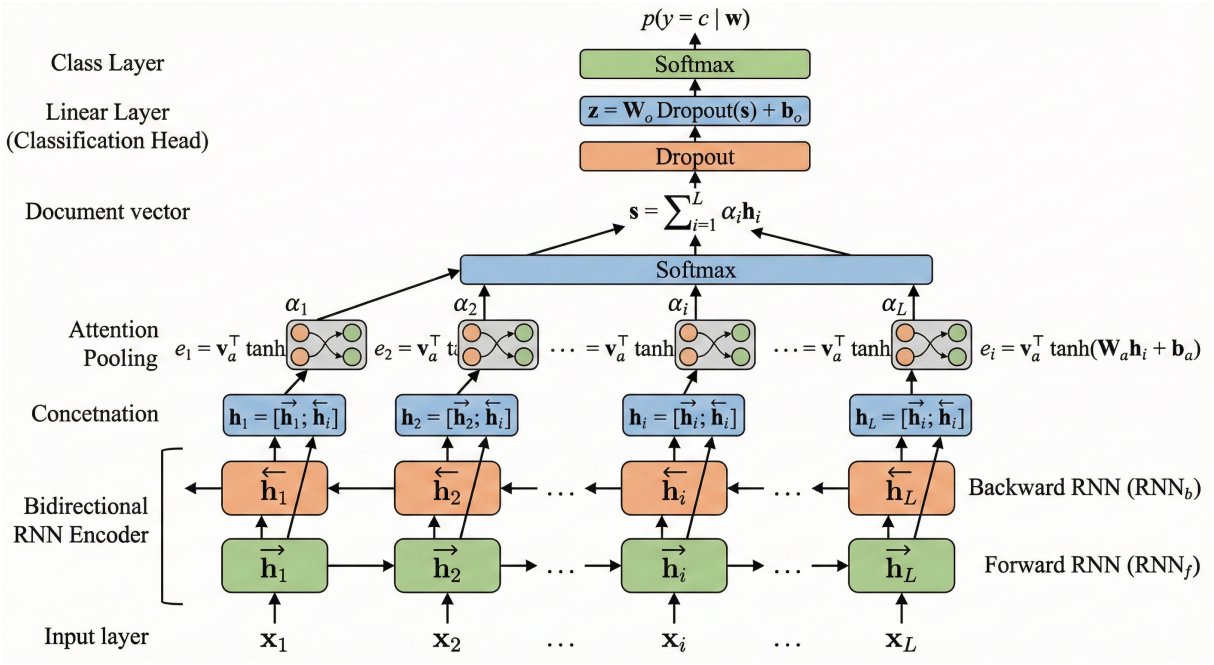

Figure 4: Bidirectional RNN with additive attention pooling for text classification.

Input representation. Given a tokenized text $\mathbf{w} = (w_1, w_2, \ldots, w_L)$ of length L, each token w_i is mapped to an embedding $\mathbf{x}_i \in \mathbb{R}^d$ using a learned lookup table $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$. To enable batched training, sequences are padded/truncated to a maximum length $L_{\max}$, and we maintain a binary padding mask to distinguish real tokens from padded positions.

Bidirectional recurrent encoder. A forward RNN reads the sequence from left to right, while a backward RNN reads it from right to left:

$$\overrightarrow{\mathbf{h}}_i = \text{RNN}_f(\mathbf{x}_i, \overrightarrow{\mathbf{h}}_{i-1}), \tag{8}$$

$$\overleftarrow{\mathbf{h}}_i = \text{RNN}_b(\mathbf{x}_i, \overleftarrow{\mathbf{h}}_{i+1}). \tag{9}$$

The contextual token representation is the concatenation of both directions,

$$\mathbf{h}_i = \left[\overrightarrow{\mathbf{h}}_i; \overleftarrow{\mathbf{h}}_i\right] \in \mathbb{R}^{2h}, \tag{10}$$

where h is the hidden size per direction. Gated cells (LSTM/GRU) mitigate vanishing gradients and allow the model to retain salient information over longer spans compared to a vanilla RNN.

Additive attention pooling. To aggregate token-level information into a single fixed-dimensional document vector, we apply an additive attention mechanism [1]. Each hidden state is scored by a small feed-forward network:

$$e_i = \mathbf{v}_a^\top \tanh(\mathbf{W}_a \mathbf{h}_i + \mathbf{b}_a), \tag{11}$$

$$\alpha_i = \frac{\exp(e_i)}{\sum_{j=1}^{L} \exp(e_j)}, \tag{12}$$

where $\mathbf{W}_a \in \mathbb{R}^{r\times 2h}$, $\mathbf{b}_a \in \mathbb{R}^r$, and $\mathbf{v}_a \in \mathbb{R}^r$ are learned parameters. In batched training, the padding mask is used so that padded positions do not receive probability mass. The resulting document representation is a weighted sum of hidden states:

$$\mathbf{s} = \sum_{i=1}^{L} \alpha_i \mathbf{h}_i \in \mathbb{R}^{2h}. \tag{13}$$

Intuitively, attention learns to assign higher weights to tokens that are more predictive of the target emoji (e.g., sentiment-bearing words, named entities, or topic keywords), and the weights can be inspected to provide a degree of interpretability.

Classification head and training objective. The attended vector $\mathbf{s}$ is regularized with dropout and passed to a linear layer to obtain class logits:

$$\mathbf{z} = \mathbf{W}_o \,\mathrm{Dropout}(\mathbf{s}) + \mathbf{b}_o, \qquad p(y = c \mid \mathbf{w}) = \mathrm{softmax}(\mathbf{z})_c. \tag{14}$$

The model is trained end-to-end by minimizing cross-entropy loss. In practice, gradient clipping is commonly used to stabilize optimization for recurrent networks.
Compared to uniform average/max pooling, attention provides a task-adaptive summary of the sequence. Relative to convolutional models such as TextCNN, bidirectional RNNs can capture dependencies spanning many tokens, but they are less parallelizable and typically slower due to the sequential nature of recurrent computation.

5.5 Transformer-Based Model: DistilBERT Fine-Tuning

We fine-tune DistilBERT [7], a compact Transformer encoder distilled from BERT [2], for multi-class emoji prediction. Transformer encoders model contextual meaning through stacked self-attention blocks [9], allowing each token to attend to all others and capture both local and long-range interactions.

Subword tokenization and inputs. Given an input text, a WordPiece-style tokenizer [2] converts it into a sequence of subword tokens and inserts special tokens to form

$$(\texttt{[CLS]}, t_1, \ldots, t_T, \texttt{[SEP]}),$$

where T varies per example. Each token is mapped to an integer ID, and sequences are padded/truncated to a fixed maximum length $L_{\max}$ for batching. We also create an attention mask $\mathbf{m} \in \{0,1\}^{L_{\max}}$ indicating which positions correspond to real tokens (1) versus padding (0); the mask prevents padded positions from influencing attention scores.

Embedding layer. Let d be the model hidden size. The input IDs are mapped to token embeddings and combined with learned positional embeddings:

$$\mathbf{x}_i = \mathbf{e}_{\mathrm{tok}}(t_i) + \mathbf{e}_{\mathrm{pos}}(i) \in \mathbb{R}^d, \qquad i = 1, \ldots, L_{\max}. \tag{15}$$

DistilBERT uses a single-sequence encoder and therefore omits segment (token-type) embeddings used in some BERT variants [7].

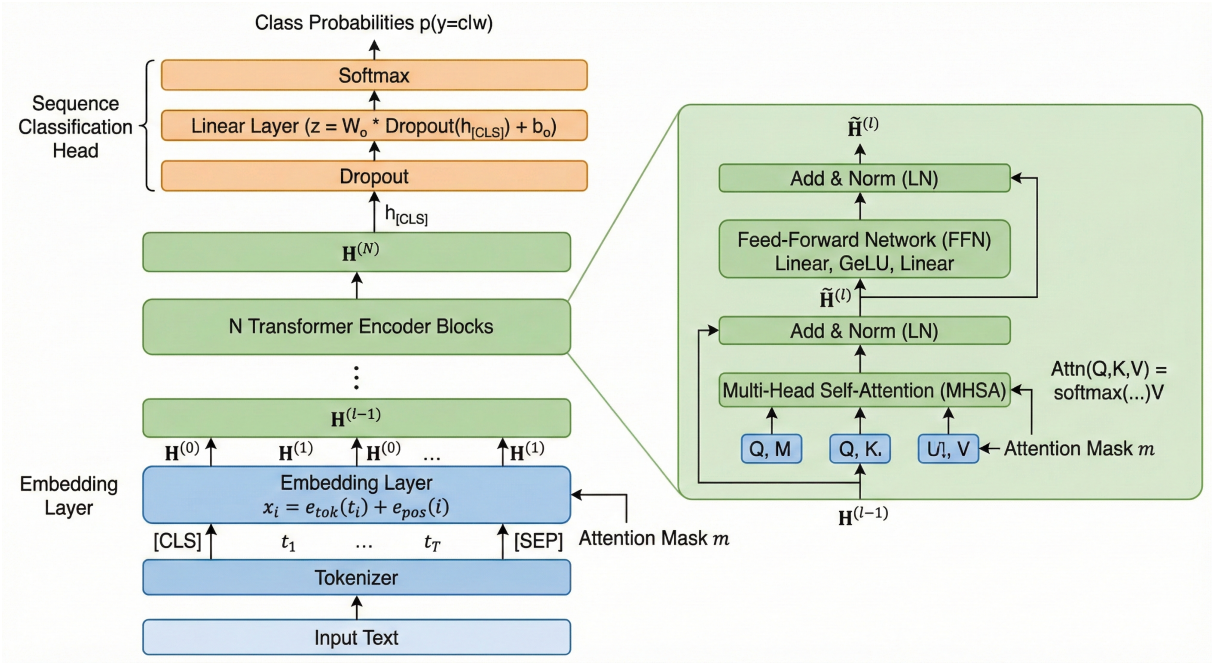

Figure 5: DistilBERT architecture overview.

Multi-head self-attention. Given the hidden states from layer $\ell - 1$, stacked into a matrix $\mathbf{H}^{(\ell-1)} \in \mathbb{R}^{L_{\max} \times d}$, each attention head computes query, key, and value projections:

$$\mathbf{Q} = \mathbf{H}^{(\ell-1)}\mathbf{W}^{Q}, \quad \mathbf{K} = \mathbf{H}^{(\ell-1)}\mathbf{W}^{K}, \quad \mathbf{V} = \mathbf{H}^{(\ell-1)}\mathbf{W}^{V}. \tag{16}$$

Self-attention produces contextualized representations by weighting values based on similarity between queries and keys:

$$\mathrm{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}, \tag{17}$$

where d_k is the per-head key dimension and $\mathbf{M}$ is a masking matrix derived from $\mathbf{m}$ that assigns large negative values to padded positions so they receive near-zero attention probability. Multiple heads are computed in parallel and concatenated, followed by an output projection (omitted for brevity) [9].

Transformer encoder block. Each layer applies self-attention and a position-wise feed-forward network with residual connections and layer normalization:

$$\tilde{\mathbf{H}}^{(\ell)} = \mathrm{LN}\left(\mathbf{H}^{(\ell-1)} + \mathrm{Dropout}(\mathrm{MHSA}(\mathbf{H}^{(\ell-1)}))\right), \tag{18}$$

$$\mathbf{H}^{(\ell)} = \mathrm{LN}\left(\tilde{\mathbf{H}}^{(\ell)} + \mathrm{Dropout}(\mathrm{FFN}(\tilde{\mathbf{H}}^{(\ell)}))\right), \tag{19}$$

where $\mathrm{FFN}(\mathbf{h}) = \mathbf{W}_2\,\sigma(\mathbf{W}_1\mathbf{h} + \mathbf{b}_1) + \mathbf{b}_2$ is applied independently to each position. Stacking N such layers yields the final contextual sequence representation $\mathbf{H}^{(N)}$.

Sequence classification head and objective. For classification, we use the final hidden state corresponding to the `[CLS]` token as a pooled sequence representation, $\mathbf{h}_{\texttt{[CLS]}} \in \mathbb{R}^d$. A linear classifier maps it to logits over C emoji classes:

$$\mathbf{z} = \mathbf{W}_o\,\mathrm{Dropout}(\mathbf{h}_{\texttt{[CLS]}}) + \mathbf{b}_o, \qquad p(y = c \mid \mathbf{w}) = \mathrm{softmax}(\mathbf{z})_c. \tag{20}$$

All parameters are fine-tuned end-to-end by minimizing cross-entropy loss. In our implementation, we use AdamW optimization with a linear warmup/decay learning-rate schedule, which is standard for Transformer fine-tuning [2].

Compared to CNN/RNN architectures trained from scratch, DistilBERT benefits from large-scale pretraining and can leverage richer syntactic/semantic features. The main trade-off is higher computational cost and memory usage, though DistilBERT remains significantly smaller and faster than full-sized BERT [7].

6 EXPERIMENTS

6.1 Experimental Protocol

Hyperparameters are tuned with `tune.py` using a train/validation split and a small, fixed grid per model family. The best configuration is selected by macro F1@1 and stored in `outputs/best_params.json`. Final evaluation is performed with `train_eval.py` on the held-out test set, producing `outputs/test_results.csv`. All experiments use $k \in \{1, 3, 5\}$ for top-k metrics and a fixed random seed (default 42). Neural models are trained with Adam/AdamW optimizers; Transformer fine-tuning uses a linear warmup/decay schedule [2].

6.2 Results

Table 2 reports test performance for each model. The Transformer achieves the best overall results, improving both top-1 and top-5 accuracy over classical baselines.

Table 2: Test-set results (`outputs/test_results.csv`).

Model	Top-1 Acc	Top-3 Acc	Top-5 Acc
TF–IDF + SVM	0.2109	0.3602	0.4460
TF–IDF + LogReg	0.2363	0.4170	0.5172
TF–IDF + NB	0.2234	0.4064	0.5112
FastText	0.2153	0.3996	0.5042
TextCNN	0.2094	0.3821	0.4864
BiLSTM + Attention	0.2254	0.4064	0.5101
BiGRU + Attention	0.2213	0.3983	0.5009
Transformer	**0.2632**	**0.4596**	**0.5659**

6.3 Discussion

Two trends are notable. First, strong TF–IDF baselines (especially logistic regression) remain competitive on short, noisy text. Second, the fine-tuned Transformer benefits from pretrained representations and achieves the best top-k performance, which is important for emoji suggestion interfaces where multiple emojis can be shown to the user.

7 DEPLOYMENT

7.1 Python Inference

The script `infer.py` provides a minimal CLI for running inference with a saved Hugging Face-style model directory (default `outputs/transformer_deploy`). It loads the tokenizer and model, computes

probabilities, and prints the top-k predictions for each input text.

7.2 Browser Demo (Client-side ONNX)

The `docs/` directory contains a static web application that loads a local model using Transformers.js [10]. Remote model downloads are disabled; all assets are served from `docs/models/emoji-model/`. The main UI (`docs/index.html`) provides real-time emoji suggestions while typing; `docs/debug.html` offers a debugging interface for batch predictions and latency measurements.

7.3 ONNX Export and Quantization

To enable browser inference, `scripts/export_onnx.py` exports the Transformer model to ONNX [6] and optionally applies dynamic quantization to int8 for smaller downloads and faster CPU execution. The exported model is stored under `docs/models/emoji-model/onnx/` and is consumed by the web app.

8 CONCLUSION

This report described a complete emoji prediction system from dataset construction to browser deployment. We benchmarked classical TF–IDF baselines, lightweight neural architectures, and a fine-tuned DistilBERT Transformer. On the provided test set, the Transformer achieved the best top-k accuracy, making it well-suited for suggestion-based user interfaces. We also presented a practical deployment workflow based on ONNX export, quantization, and client-side inference via Transformers.js.

8.1 Limitations and Future Work

The current formulation predicts a *single* emoji per text, while real messages may contain multiple emojis. Future work could explore multi-label prediction, calibration of confidence scores for better ranking, and domain adaptation to non-twitter text. Additionally, stronger pretrained models or multilingual training could improve robustness across topics and languages.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *International Conference on Learning Representations (ICLR)*. 2015.

[2] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of NAACL-HLT*. 2019.

[3] Armand Joulin et al. "Bag of Tricks for Efficient Text Classification". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*. 2017.

[4] Yoon Kim. "Convolutional Neural Networks for Sentence Classification". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014.

[5] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[6] *ONNX: Open Neural Network Exchange*. https://onnx.ai/. Accessed 2026-02-01. 2026.

[7] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[8] Martin Thoma. *The WiLI benchmark dataset for written language identification*. arXiv:1801.07779. 2018.

[9] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[10] Xenova. *Transformers.js*. https://github.com/xenova/transformers.js. Accessed 2026-02-01. 2026.