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

Emojis add emotional nuance and emphasis to online writing, yet manually selecting the right emoji can interrupt typing flow. This report studies **single-label emoji prediction from short social-media text** and presents an end-to-end pipeline: (i) building a cleaned tweet-style dataset of 835,428 English posts across 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 multinomial Naive Bayes), FastText-style subword models [5], TextCNN [6], bidirectional RNNs with attention, and a fine-tuned DistilBERT Transformer [9]. On the held-out test split (83,543 samples), DistilBERT achieves the strongest ranking quality with **top-1 accuracy 26.3%** and **top-5 accuracy 56.6%**. For practical deployment, we export the fine-tuned Transformer to **quantized ONNX** and run it entirely in the browser using Transformers.js and an ONNX Runtime WebAssembly backend, enabling low-latency and privacy-preserving emoji suggestions without server-side inference [8, 12]. We additionally release an interactive web application at https://luongkhang04.github.io/emoji/ and a Microsoft Edge extension at https://microsoftedge.microsoft.com/addons/detail/emoji-suggest/jdjoadakfkdcdllijajaepmplaodkkhj.

Table of Contents

1 INTRODUCTION

1.1 Motivation

Emojis have become a standard layer of meaning in online writing. Beyond decoration, they often convey affect (e.g., joy, frustration), modulate tone (e.g., irony, teasing), or compress an intention into a single symbol that readers recognize instantly. On social media, where messages are short and context is limited, this extra signal can materially change how a post is interpreted and how much engagement it receives.
Despite their usefulness, selecting an emoji is a small but frequent interruption: users must pause, open an emoji panel, search or scroll, and decide among visually similar candidates. This friction is especially noticeable on mobile devices and in fast-paced chat. An emoji suggestion system that predicts likely emojis from the text being typed can reduce this overhead and support smoother, more expressive communication. In this report, we study emoji prediction as a text classification problem and build an end-to-end pipeline that is accurate enough for interactive use.

1.2 Scope of This Work

We focus on *single-label* emoji prediction: given a short English, tweet-like text, the system predicts exactly one emoji from a fixed inventory of 43 classes. This setting matches common UI patterns such as showing a top suggestion or a short ranked list of candidates while a user types.
The accompanying code and assets include:

- a cleaned dataset split into training and test CSV files;
- training and evaluation scripts that benchmark classical baselines and neural architectures under a consistent protocol;
- a lightweight Python inference interface for the best-performing Transformer checkpoint;
- a client-side deployment workflow that exports the trained model to ONNX (with quantization) and runs inference locally in the browser.

By keeping inference on-device (in the browser), the system can provide low-latency suggestions without sending user text to a remote server, which is desirable for privacy and ease of deployment.

1.3 Contributions

The main contributions of this report are:

- a practical preprocessing pipeline tailored to noisy, tweet-style text, including normalization and emoji removal for label purity;
- a unified benchmarking setup comparing sparse TF–IDF baselines and multiple neural model families using top-k accuracy metrics;
- an end-to-end deployment path that converts the strongest model to a quantized ONNX format and integrates it into a static web demo for fast, local inference.

2 RELATED WORK

Emoji prediction sits at the intersection of short-text classification, sentiment/emotion analysis, and representation learning from noisy social signals. This section reviews (i) prior work directly targeting emoji prediction, (ii) sentiment/emotion classification lines of research that are methodologically close, and (iii) recent trends in lightweight deployment of neural NLP models—then positions our work within these threads.

Context of this report. Our project frames emoji prediction as single-label multi-class classification over 43 emojis on tweet-style English text, benchmarks classical and neural model families, and deploys the best model for fully client-side inference in the browser. *(See overall report for full details.)*

2.1 Emoji prediction from text

A major catalyst for emoji prediction research is the SemEval-2018 Task 2 shared task on *Multilingual Emoji Prediction*, which formalized the problem as predicting the most likely emoji given a tweet and evaluated systems using macro F1 across emoji classes (English and Spanish subtasks). The shared-task setting highlighted practical issues central to emoji prediction: label ambiguity, topical vs. affective signals, and class imbalance in naturally occurring emoji frequencies. [2]

Beyond shared tasks, emoji supervision has been used at web scale to learn transferable affective representations. DeepMoji trains a model on ∼1.2B tweets labeled by emojis, showing that emoji prediction can act as a strong form of distant supervision and transfer effectively to sentiment, emotion, and sarcasm detection benchmarks. [4] This line of work motivates the use of pretrained language models (or large-scale pretraining) for emoji-related tasks: emojis are not merely "decorations" but weak labels for underlying affect and pragmatics.

More recently, transformer-based approaches (e.g., BERT-style fine-tuning) have been explored for emoji prediction, typically reporting improvements over bag-of-words and earlier neural baselines due to contextual representations and better handling of polysemy and compositionality. [3] [11] These findings align with broader text-classification results where pretrained transformers dominate, especially when fine-tuned on in-domain short text.

2.2 Connections to sentiment and emotion classification

Emoji prediction is tightly related to sentiment analysis and emotion recognition: both aim to map a short utterance into an affective or expressive category. Classical sentiment pipelines often combine sparse lexical features (word/character n-grams with TF–IDF) with linear classifiers, which remain strong baselines for short, noisy text due to their robustness and low variance. [7] While sentiment analysis typically uses coarse labels (positive/negative/neutral) or a small set of emotions, emoji prediction expands the label space and introduces additional complexity: many emojis are topic- or event-linked (e.g., sports, food), and multiple emojis can plausibly fit the same text.

Neural architectures developed for sentence classification (FastText-style subword models, CNNs, RNNs

with attention) are commonly applied to both sentiment and emoji prediction because they capture morphology, local phrases, and limited sequential context efficiently. [5] [6] These models provide a middle ground between sparse linear baselines and large pretrained transformers.

2.3 How our work differs

Compared with prior emoji-prediction work (e.g., SemEval-style settings and large-scale distant supervision), our contribution is *system-oriented and end-to-end*:

- **Controlled label space and benchmarking.** Instead of focusing on leaderboard systems or extremely large emoji inventories, we target a fixed set of 43 emojis and provide a consistent, reproducible benchmark across classical TF–IDF baselines, lightweight neural models, and a fine-tuned transformer. This emphasizes practical trade-offs (accuracy vs. simplicity) under a unified protocol.
- **Top-k evaluation for suggestion interfaces.** Many texts admit multiple reasonable emojis, so we report ranking-oriented metrics (Acc@k) that better match real autocomplete/suggestion UX than strict top-1 accuracy.
- **Deployment as a first-class objective.** While much prior work stops at modeling results, we prioritize *privacy-preserving, client-side* inference by exporting the best transformer model to ONNX and running it in-browser via Transformers.js / ONNX Runtime Web, enabling offline-capable emoji suggestions without server-side text logging.

Overall, the literature suggests a clear progression: sparse linear baselines provide strong performance for short social text; lightweight neural models add robustness and limited compositionality; and pretrained transformers achieve the best accuracy by leveraging contextual representations. Our work follows this progression but extends it with an end-to-end pipeline and a deployment path designed for interactive emoji suggestion in real user environments.

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 [10]. 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.

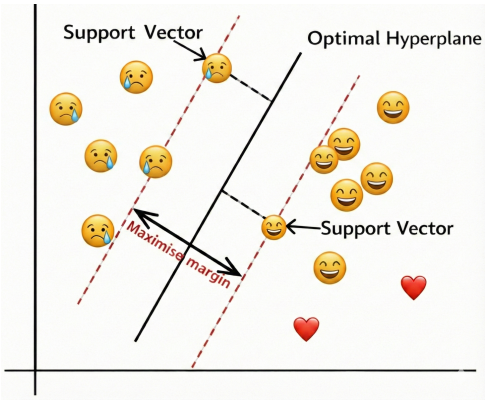

Figure 2: Linear Support Vector Machine.

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 [5] 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 [5].

5.3 TextCNN

TextCNN [6] 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.

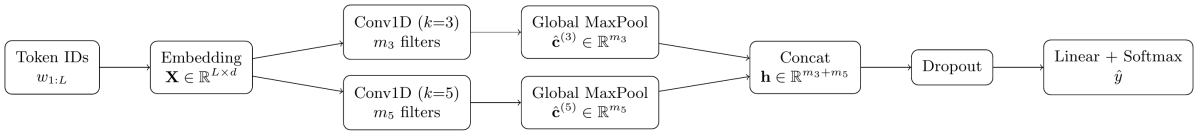

Figure 3: TextCNN architecture with multiple filter widths and global max-over-time pooling.

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.

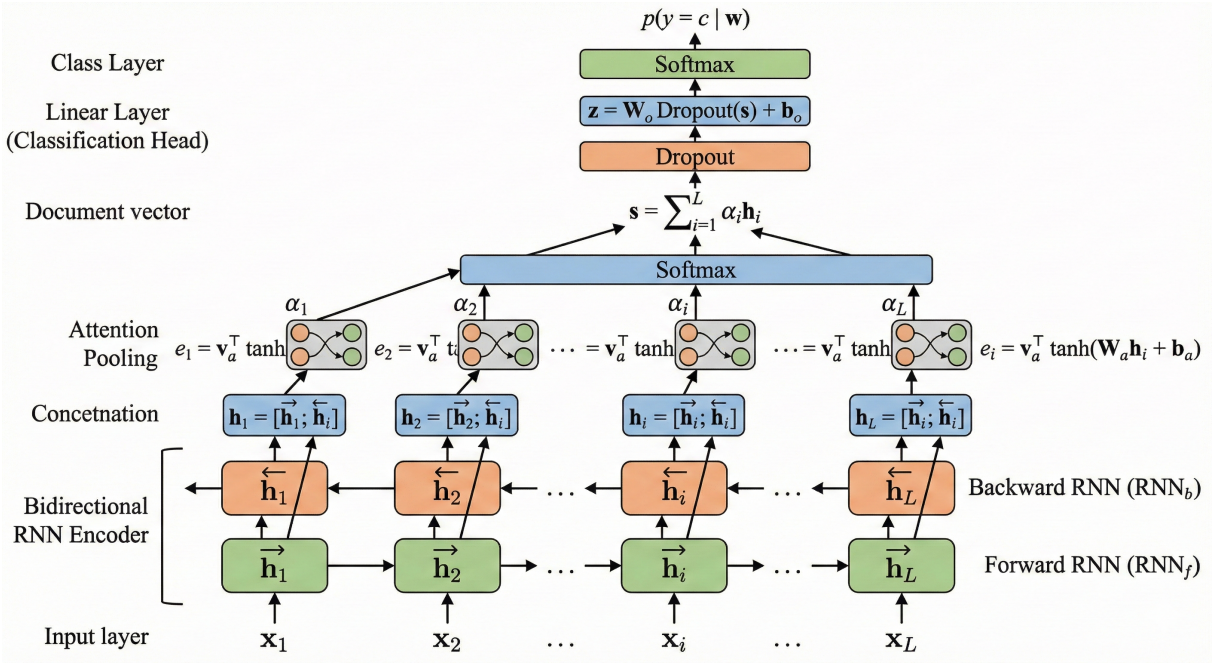

Figure 4: Bidirectional RNN with additive attention pooling for text classification.

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 [9], a compact Transformer encoder distilled from BERT [3], for multi-class emoji prediction. Transformer encoders model contextual meaning through stacked self-attention blocks [11], 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 [3] 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 [9].

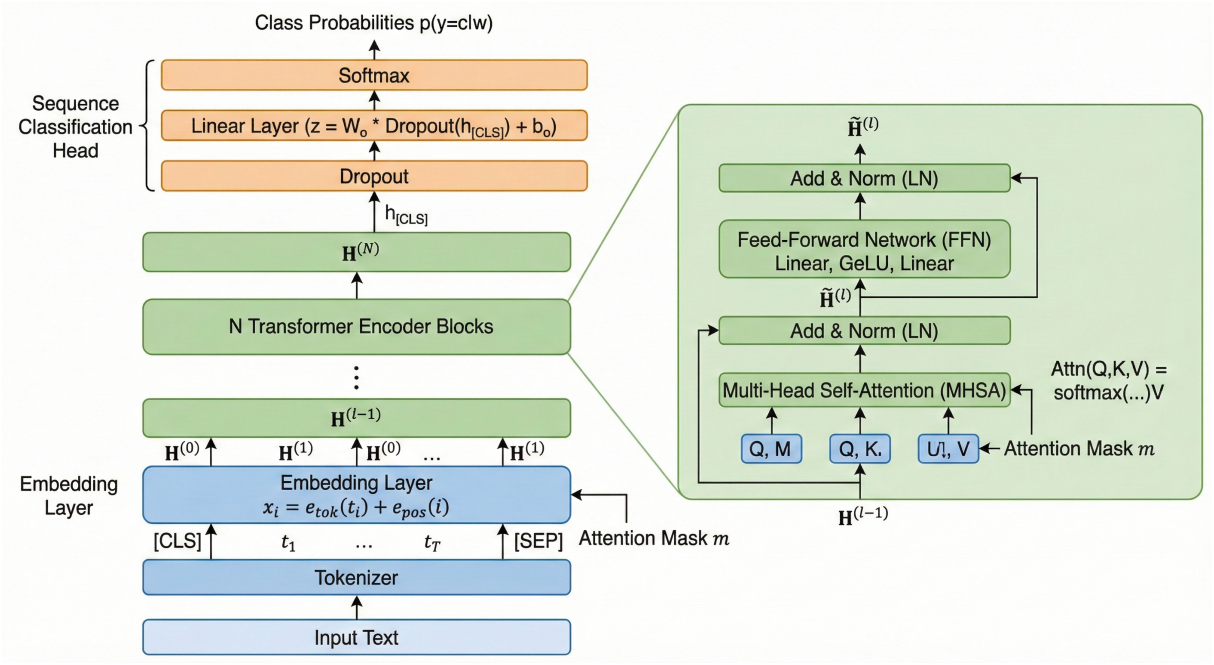

Figure 5: DistilBERT architecture overview.

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) [11].

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 [3].

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 [9].

6 EXPERIMENTS

6.1 Experimental Protocol

All models are trained and evaluated on the dataset split summarized in Table 1. Hyperparameter selection is performed using only the training split (`data/train_data.csv`). Concretely, `tune.py` creates a stratified train/validation split with 80% of the training data used for fitting and 20% used for validation (`-val_size 0.2`), using a fixed random seed of 42 for reproducibility. For each model family, we evaluate a small, fixed grid of candidate configurations; each configuration is trained on the tuning-train split and scored on the tuning-validation split. The best configuration for each model is stored in `outputs/best_params.json`. All tuning trials and their validation metrics are stored in `outputs/tuning_results.csv`.

Classical baselines (TF–IDF + linear classifiers) are implemented as scikit-learn pipelines consisting of a FeatureUnion of word and character TF–IDF vectorizers, followed by a linear classifier (Linear SVM, multinomial Naive Bayes, or multinomial logistic regression). Neural baselines (FastText, TextCNN, and RNNs with attention) are implemented in PyTorch and optimized with Adam; during tuning, they use early stopping with patience 2 on the validation split. For these models, we build a word vocabulary from the tuning-train split only (maximum 50,000 tokens, minimum frequency 2) and truncate sequences to the configured maximum length. The Transformer is fine-tuned end-to-end with AdamW using a linear warmup/decay learning-rate schedule [3]; we keep the epoch that yields the best validation performance. Final testing is performed with `train_eval.py`. For each model, we retrain using the selected hyperparameters on the full training set and evaluate once on the held-out test set (`data/test_data.csv`), producing `outputs/test_results.csv`. Unless specified otherwise, we report Acc@1, Acc@3, and Acc@5 and select the compute device automatically (`cuda` if available, otherwise CPU).

6.2 Evaluation Metrics

For an input text x, the model produces a score vector over the $C = 43$ emoji classes. Accuracy@k (Acc@k) counts a prediction as correct if the ground-truth emoji appears among the k highest-scoring classes. We report Acc@1, Acc@3, and Acc@5.

6.3 Chosen Hyperparameters

Table 2 summarizes the selected configuration for each model and the corresponding validation Acc@k. Full per-trial results are available in `outputs/tuning_results.csv`.

Table 2: Selected hyperparameters and validation accuracies (see `outputs/best_params.json` and `outputs/tuning_results.csv`).

Model	Val Acc@1	Val Acc@3	Val Acc@5	Selected hyperparameters
TF–IDF + SVM	0.2047	0.3519	0.4377	word n-grams: 1–2; char n-grams: 3–5; min df: 2; max features: 200k; $C = 1.0$; class weights: balanced.
TF–IDF + LogReg	0.2308	0.4086	0.5094	word n-grams: 1–2; char n-grams: 3–5; min df: 2; max features: 200k; $C = 2.0$ (solver: saga).
TF–IDF + NB	0.2189	0.4023	0.5068	word n-grams: 1–2; char n-grams: 3–5; min df: 2; max features: 200k; $\alpha = 0.5$.
FastText	0.2112	0.3935	0.4967	embed dim: 100; dropout: 0.2; lr: 10^{-3}; epochs: 5; batch size: 128; min/max subword n: 3/6; buckets: 200k; max words: 200; max subword n-grams: 2000.
TextCNN	0.2015	0.3747	0.4772	embed dim: 200; filters: 128; filter sizes: 3, 4, 5; dropout: 0.5; lr: 5×10^{-4}; epochs: 6; batch size: 128; max length: 200.
BiLSTM + Attention	0.2118	0.3906	0.4924	embed dim: 100; hidden size: 128; dropout: 0.4; lr: 10^{-3}; epochs: 6; batch size: 128; max length: 200.
BiGRU + Attention	0.2129	0.3913	0.4942	embed dim: 100; hidden size: 128; dropout: 0.4; lr: 10^{-3}; epochs: 6; batch size: 128; max length: 200.
Transformer	0.2544	0.4531	0.5588	DistilBERT (`distilbert-base-uncased`); lr: 2×10^{-5}; epochs: 3; batch size: 32; max length: 128.

6.4 Results

Table 3 reports test performance for each model. The Transformer achieves the best overall results, improving Acc@1/Acc@3/Acc@5 over classical baselines.

Table 3: Test-set results (`outputs/test_results.csv`).

Model	Acc@1	Acc@3	Acc@5
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**

6.5 Discussion

Two trends are notable. First, strong TF–IDF baselines (especially logistic regression) remain competitive on short, noisy text, suggesting that sparse n-gram features still capture a large fraction of the signal

for this task. Second, the fine-tuned Transformer benefits from pretrained contextual representations and achieves the best Acc@k. In particular, it improves Acc@1 from 0.2363 (best classical baseline) to 0.2632, Acc@3 from 0.4170 to 0.4596, and Acc@5 from 0.5172 to 0.5659 (Table 3).

7 DEPLOYMENT

7.1 Deployment Overview

While Table 3 shows that several lightweight baselines perform competitively, we deploy the best-performing model (fine-tuned DistilBERT) because it provides the strongest top-k ranking quality for suggestion-style interfaces. The deployment workflow in this repository supports two complementary targets:

- **Python (server/desktop) inference** for quick testing and integration in research code, using Hugging Face Transformers to load the saved checkpoint in `outputs/transformer_deploy`.
- **Client-side browser inference** (static web app and browser extension) by exporting the same checkpoint to ONNX and running it locally with Transformers.js and an ONNX Runtime WebAssembly backend [8, 12].

Both targets share the same label mapping (`id2label.json`) and tokenizer assets. For Python inference, we also ship `training_meta.json` so that `infer.py` can recover the training-time `max_len` when running predictions.

7.2 Python Inference

The simplest way to run the deployed Transformer is the command-line helper `infer.py`. It loads the tokenizer and sequence-classification head from a local model directory (default `outputs/transformer_deploy`), moves weights to the chosen device, and returns the top-k predicted emojis with probabilities. Device selection can be automatic (`-device auto`, which uses CUDA if available) or explicit (e.g., `-device cpu`).

Typical usage is:

- `python infer.py -text "i can't believe this happened" -topk 5`
- `python infer.py -model_dir outputs/transformer_deploy -text "so excited for tomorrow" -device auto`

For reproducibility, `infer.py` prints both the weight-load time and the end-to-end inference time, and it uses `training_meta.json` to recover the configured `max_len` unless overridden via `-max_len`.

```
(emoji) D:\Coding\emoji>python infer.py --text "Great to see you" --topk 10
Loading weights: 100%|| 104/104 [00:00<00:00, 4005.14it/s, Materializing
Weight load time: 1262.4 ms
Inference time: 152.9 ms
Text: Great to see you
  1. 😀 (0.2063)
  2. 👍 (0.1747)
  3. 🌞 (0.0625)
  4. 😎 (0.0593)
  5. 👶 (0.0526)
  6. 😘 (0.0522)
  7. ❤️ (0.0462)
  8. 🤍 (0.0363)
  9. 😍 (0.0321)
  10. 🤍 (0.0299)
```

Figure 6: Python inference example.

7.3 ONNX Export and Quantization

For browser deployment, we convert the fine-tuned Transformer checkpoint to ONNX. This produces a framework-agnostic computation graph that can be executed efficiently by ONNX Runtime in JavaScript/WASM environments [8]. In this repository, the script `scripts/export_onnx.py` exports the model in the directory layout expected by Transformers.js [12]:

- model assets (`config.json`, `tokenizer.json`, `id2label.json`, etc.) are copied to the output directory;
- the ONNX file is placed under `onnx/` and renamed to `model.onnx` (or `model_quantized.onnx` after quantization).

The script supports multiple export backends for robustness: it prefers Optimum's ONNX Runtime export if available, otherwise falls back to `transformers.onnx` (and finally to `torch.onnx.export` as a last resort). For deployment, we additionally apply dynamic int8 weight quantization (`-quantize q8`), which reduces model size and typically improves CPU/WASM inference speed with minimal quality degradation.

A standard export command is:

- `python scripts/export_onnx.py`

By default, this exports from `outputs/transformer_deploy` to `docs/models/emoji-model` and applies `q8` quantization. The same script can also export into `extension/models/emoji-model` when rebuilding the browser extension bundle. If `onnxruntime` is not installed, the script still exports ONNX but skips quantization and reports the missing dependency.

7.4 Web Application

The static web demo lives in `docs/` and is designed to run on any static file host (e.g., GitHub Pages). We publish the web application at https://luongkhang04.github.io/emoji/. The main UI (`docs/index.html`) provides a text area and an "emoji bar" that displays the top-k suggestions. The inference logic in `docs/main.js` loads the quantized ONNX model from `docs/models/emoji-model/` using Transformers.js:

- Remote model downloads are disabled (`env.allowRemoteModels = false`) to ensure fully local inference.
- `env.localModelPath` points to `docs/models/`, so the pipeline resolves `emoji-model` from the bundled assets.
- A short debounce window limits inference frequency while typing, improving responsiveness.
- ONNX Runtime WASM threading is enabled (up to 4 threads) when supported by the browser.

The service worker (`docs/sw.js`) caches core assets so the application remains usable offline after the first successful load.

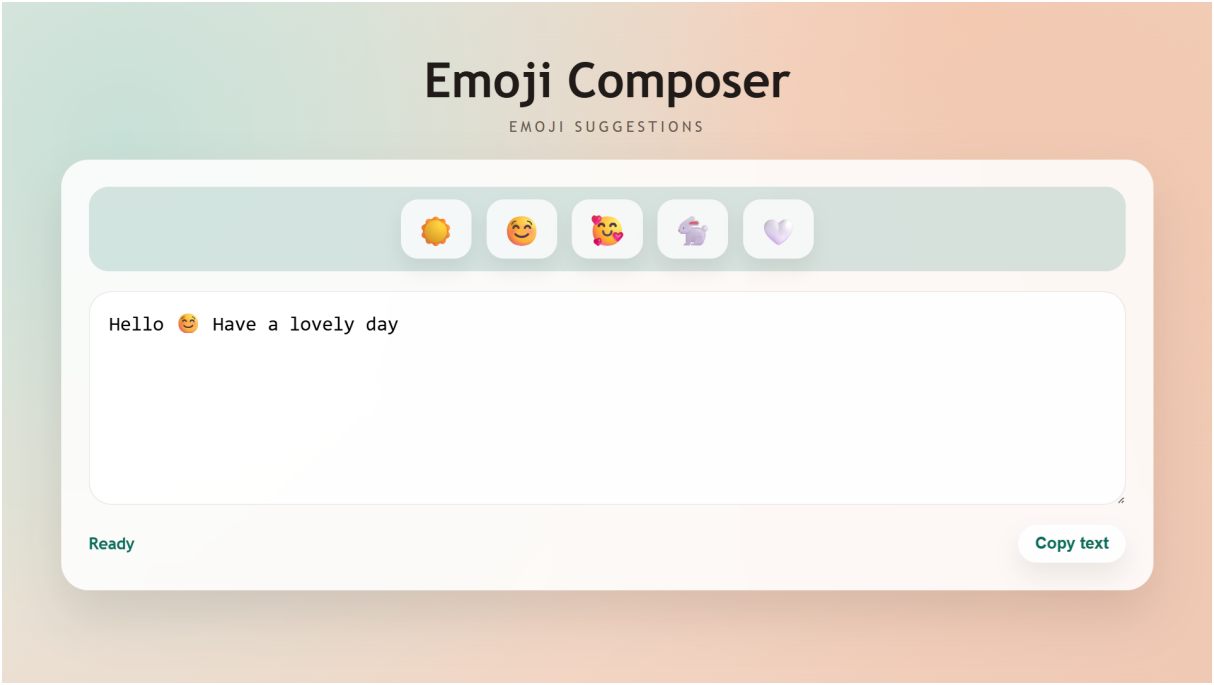

Figure 7: Web application on desktop.

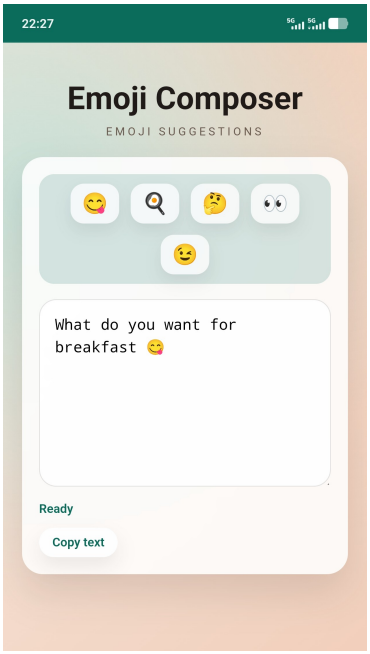

Figure 8: Web application on mobile.

7.5 Browser Extension

In addition to the standalone web demo, we provide a lightweight browser extension (folder `extension/`) that suggests a single emoji inline while the user types. The extension is implemented as a Manifest V3 content script that runs on all pages and attaches listeners to editable targets (`input`, `textarea`, and `contenteditable` elements). Its workflow is:

- Extract the most recent sentence fragment (splitting on punctuation) and remove any existing emojis.
- Run a top-1 prediction with the local ONNX model using Transformers.js and ONNX Runtime WASM.
- If the predicted probability exceeds a small threshold, render a compact suggestion tooltip near the cursor.
- Insert the suggested emoji when the user presses `Tab`, adding surrounding whitespace when appropriate.

All inference runs locally in the browser: the model (`extension/models/emoji-model/`) and runtime assets (`extension/vendor/`) are packaged with the extension and exposed via `web_accessible_resources`. This design avoids sending user text to a server and keeps latency low enough for interactive typing assistance.

We publish the extension on Microsoft Edge Add-ons at https://microsoftedge.microsoft.com/addons/detail/emoji-suggest/jdjoadakfkdcdllijajaepmplaodkkhj.

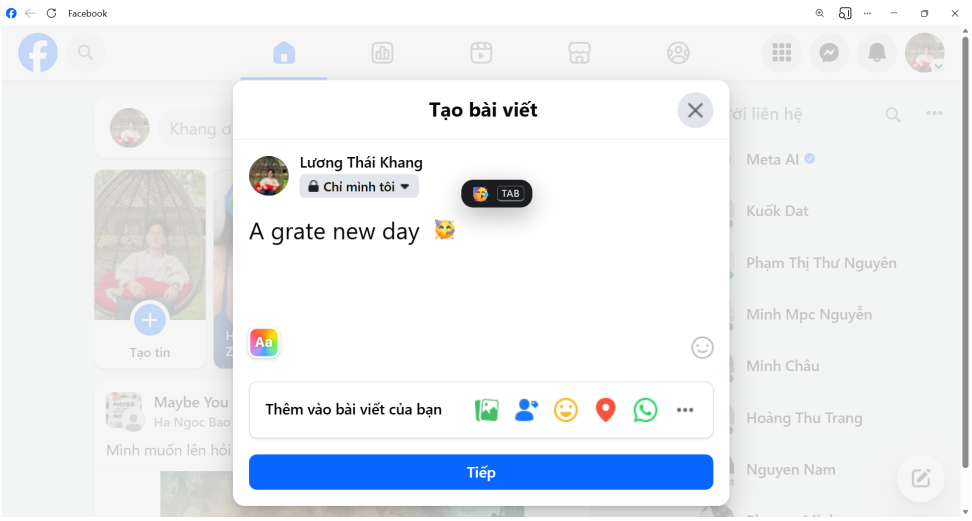

Figure 9: Browser extension example.

8 CONCLUSION

This report presented an end-to-end emoji prediction system that spans data construction, model benchmarking, and practical deployment for interactive use. Starting from tweet-style text, we built a cleaned dataset of $C = 43$ emoji classes and established a consistent evaluation protocol using top-k accuracy to reflect the fact that multiple emojis may be plausible for the same message. :contentReference[oaicite:0]index=0

Across a diverse set of baselines—TF–IDF with linear classifiers, FastText-style subword models, TextCNN, and bidirectional RNNs with attention—we observed that strong sparse-feature methods remain competitive on short, noisy social text. However, the fine-tuned DistilBERT Transformer achieved the best ranking quality on the held-out test set, reaching Acc@1 = 0.2632 and Acc@5 = 0.5659, which makes it particularly suitable for suggestion-style interfaces where users benefit from multiple candidates. :contentReference[oaicite:1]index=1

Beyond model accuracy, we demonstrated a deployment pathway designed for real-world usability. The best checkpoint is exported to ONNX, optionally quantized, and executed client-side via Transformers.js and ONNX Runtime (WASM), enabling fast, privacy-preserving inference directly in the browser. We additionally provided both a static web demo and a lightweight browser extension to illustrate how the model can support emoji suggestions during typing without sending user text to a server. :contentReference[oaicite:2]index=2

8.1 Limitations and Future Work

Despite encouraging results, several limitations remain. First, the current task is formulated as *single-label* classification, while real messages often contain multiple emojis (or none), and emoji usage can be highly user- and context-dependent. Second, the dataset is derived from scraped tweet-like text and may contain residual noise, topical bias, and platform-specific conventions that limit generalization to other domains (e.g., chat, forums, or multilingual settings). Third, our evaluation focuses on top-k accuracy; for deployment, *confidence calibration* and thresholding are also important for deciding when to suggest an emoji and how to rank candidates under uncertainty. :contentReference[oaicite:3]index=3

Future work could therefore explore: (i) multi-label and "no-emoji" modeling, (ii) improved calibration (e.g., temperature scaling) for more reliable probabilities and better UX, (iii) domain adaptation and multilingual training to broaden robustness, and (iv) personalization mechanisms that learn from user feedback while preserving privacy (e.g., on-device lightweight adapters). Finally, larger or more specialized pretrained encoders may further improve accuracy, but should be evaluated jointly with latency and model size constraints to maintain a responsive in-browser experience. :contentReference[oaicite:4]index=4

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] Francesco Barbieri et al. "SemEval 2018 Task 2: Multilingual Emoji Prediction". In: *Proceedings of the 12th International Workshop on Semantic Evaluation*. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 24–33. DOI: 10.18653/v1/S18-1003. URL: https://aclanthology.org/S18-1003/.

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

[4] Bjarke Felbo et al. "Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 1615–1625. DOI: 10.18653/v1/D17-1169. URL: https://aclanthology.org/D17-1169/.

[5] 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.

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

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

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

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

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

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

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