# Efficient Compression-Based Low-resource Text Classification

**Bruno Vargas de Souza**[1], **Pedro Garcia Freitas**[1]

[1]Department of Computer Science – University of Brasília (UnB)
CIC/EST Building, Campus Darcy Ribeiro – 70.910-900 – Brasília – DF – Brazil

bruno.vargas@aluno.unb.br, pedro.freitas@unb.br

***Abstract.*** *In recent years, the machine learning community has engaged on training increasingly powerful large language models (LLMs) for text classification tasks. These LLMs demand huge computational capabilities, requiring billions of parameters and enormous amounts of labeled data, which can make them expensive to use, optimize, and deploy in practice. However, recent studies on compressor-based distance metrics for text classification have been proposed, aiming to achieve competitive accuracy when compared with these LLM models. Compression-based models are usually based on k-nearest neighbors (KNN) and demand very few training parameters. In this paper, we propose a compression-based method that uses a Burkhard-Keller tree (BKTree) for similarity search, accelerating the KNN on compressed data. When compared with the state-of-the-art, the proposed method achieved speedups of 20x, 25x, 7x, 6.6x, 8x, 10x, 1.4x, 1.9x, 11x, and 12x for the Brotli, FSST, LZ4, LZAV, LZF, QuickLZ, Shoco, Smaz, Snappy, and ZLib compression algorithms, respectively. The proposed method is able to achieve similar prediction performance results while reducing the average runtime and computational resources, demonstrating its advantage in saving computational resources.*

## 1. Introduction

Nearest neighborss (NNs) have significantly boosted the performance of text classification, a core task in natural language processing (NLP). Most NNs require a lot of data to train effectively, and this need grows as their number of parameters increases. Moreover, for different datasets, hyperparameters require careful tuning with particular tokenization and removing stop words—needs to be customized for each specific model and dataset. While complex deep NNs present great performance on uncovering hidden connections and patterns, they can be excessive for simpler tasks where less complex methods are usually sufficient.

Among the efforts to develop less resource-intensive options than deep NNs, a notable direction explores the use of compressors for text classification. For instance, [Teahan and Harper 2003] and [Frank et al. 2000] are based on the idea that a text document belongs to the class whose language model (created with a compressor) has the lowest cross-entropy with that document. Despite their simplicity, these approaches have not been able to achieve the same quality as NNs. However, more recently, [Jiang et al. 2023] proposed `NPC_Gzip`, a string classification method that combines a text compression with a KNN classifier informed by a compression-based distance metric that achieved comparable performance.

`NPC_Gzip` uses compressors to identify patterns, converting these into similarity scores via a compressor-based distance metric. It employs a naive implementation

of KNN for classification. This method was tested on seven in-distribution and five out-of-distribution datasets showing compelling results. Even with a basic compressor like `gzip`, the method rivalized with several NN on six of the seven in-distribution datasets, surpassing state-of-the-art methods in few-shot learning scenarios. However, despite the impressive classification performance of this method, [Jiang et al. 2023] proposed a naive, brute-force KNN algorithm where, for each string to be classified, a distance matrix is calculated between the the text under prediction against all training samples.

Addressing these shortcomings, we extend the work of [Jiang et al. 2023] to leverage its benefits while improving its computational complexity. Our work redesigns the algorithm and implements data structures to improve NN search. More specifically, we employ a compressed BKTree to store the training data compactly, enabling a more efficient KNN search and replacing the brute-force strategy used by [Jiang et al. 2023]. Moreover, we extend [Jiang et al. 2023] experiments to evaluate additional compression algorithms beyond `gzip`. These experiments were motivated by the fact that there is a myriad of compression algorithms available in the literature. Some of these algorithms are faster than `gzip`, although they may not compress as effectively. With these experiments, we investigate how the compression rate affects the trade-off between prediction and computational efficiency.

The rest of this paper is organized as follows. Section 2 discusses relevant prior work that formed the basis for this study. Section 3 describes the proposed methodology. Sections 4 and 5 present the experimental setup and results, respectively. Lastly, findings from the experiment and our views on future works are reported in Section 6.

## 2. Brief Review of `NPC_Gzip`

The simplest form to implement a KNN algorithm consists of calculating the distance to every single training point for each prediction. Consider a training dataset $\mathcal{D}$, consisting of pairs of strings and their corresponding class labels, an unlabeled query string $\mathbf{q}$ for which a class prediction is sought, and an integer $k$ representing the number of nearest neighbors to consider. For each instance $(\mathbf{s}_i, \mathbf{c}_i) \in \mathcal{D}$, a string metric $d$ (e.g., Leveinshtein, Simon-White, Jaro-Winkler, etc) is used to calculate the distance between $\mathbf{q}$ and the training string $\mathbf{s}_i$, i.e., $d_i = d(\mathbf{q}, \mathbf{s}_i)$. The collection of pairs $\{(d_i, \mathbf{c}_i)\}$ is sorted in ascending order based on their distance values $d_i$. The top $k$ in this sorted list correspond to the $k$ training strings that are most similar to the $\mathbf{q}$, i.e., $\mathcal{N} = \{(d_j, c_j)\}_{j=1}^{k}$. The counting of occurrences of each class label $\mathbf{c}_i$ within the set $\mathcal{N}$ determines the predicted class associated with $\mathbf{q}$. The class label that occurs most frequently among its $k$ nearest neighbors is assigned to the predicted class $\overline{\mathbf{c}}$. Algorithm 1 depicts these steps.

The time complexity of this naive and brute-force approach is dominated by the distance calculations and sorting. Given that string metric distance calculation for two strings of length $r$ and $n$ has complexity of $O(rn)$, the overall prediction complexity for a single query becomes $O(mrn)$, where $m$ is the amount of train instances. This complexity can be computationally intensive for large training datasets or long strings. While naive KNN algorithm has a considerable computational complexity, it has some distinct advantages particularly in certain scenarios. For instance, this brute-force KNN approach guarantees finding the true $kvalue$ nearest neighbors. Tree-based and approximated KNN algorithms, while faster, sometimes sacrifice perfect accuracy for speed. In situations

**Algorithm 1** Brute-Force $k$-NN for String Classification

**Require:** Training dataset $\mathcal{D} = \{(\mathbf{s}_1, \mathbf{c}_1), \ldots, (\mathbf{s}_m, \mathbf{c}_m)\}$.
**Require:** Query string $\mathbf{q}$.
**Require:** Number of neighbors $k \in \mathbb{N}^+$.
**Require:** String distance metric $d(\cdot, \cdot)$
**Ensure:** Predicted class for $\mathbf{q}$.
1: Initialize an empty list of (distance, class) pairs: $\mathcal{L} \leftarrow [\,]$
2: **for** each $(\mathbf{s}_i, \mathbf{c}_i)$ in $\mathcal{D}$ **do**
3: Calculate the distance between $\mathbf{q}$ and $\mathbf{s}_i$: $dist_i \leftarrow d(\mathbf{q}, \mathbf{s}_i)$
4: Add $(dist_i, \mathbf{c}_i)$ to $\mathcal{L}$
5: Sort $\mathcal{L}$ by distance in ascending order
6: Select the first $k$ elements from the sorted list: $\mathcal{N} \leftarrow \{(dist_j, \mathbf{c}_j) \mid \text{first } k \text{ elements}\}$
7: Extract the classes of the KNNs: $\mathcal{C}_{N_K} \leftarrow \{\mathbf{c}_j \mid (dist_j, \mathbf{c}_j) \in \mathcal{N}\}$
8: Predict the class of $\mathbf{q}$ as the mode of the neighbor classes: $\overline{\mathbf{c}} \leftarrow \text{Mode}(\mathcal{C}_{N_K})$
9: **return** $\overline{\mathbf{c}}$

where the decision boundaries are intricate, brute-force ensures the most precise result.

[Jiang et al. 2023] adopted the Algorithm 1 but adopting normalized compression distance (NCD) in place of string metrics such as Levenshtein, Sørensen-Dice, or Jaro-Winkler. While these string metrics are based on edit operations (i.e., syntactic or lexical similarity), NCD is a informational similarity metric. NCD is defined as:

$$\text{NCD}(x, y) = \frac{\text{C}(xy) - \min(\text{C}(x), \text{C}(y))}{\max(\text{C}(x), \text{C}(y))}, \tag{1}$$

where $\text{C}(z)$ is the compressed length of string $z$. It means that, while Levenshtein, Sørensen-Dice, or Jaro-Winkler distances focus on the exact characters, their order, and small structural variations, NCD considers whether the compared strings convey similar information. In other words, NCD can implicitly capture some semantic aspects based on string information (entropy) achieved by string compressibility.

## 3. Proposed Method

The naive brute-force characteristic of Algorithm 1 implies that for every new query input, the distance calculation is performed against all $m$ instances in the training dataset. This is computationally intensive especially for large training datasets since the time complexity for a single prediction is directly proportional to the size of the training dataset ($O(m)$ distance calculations plus sorting overhead). On the other hand, tree-based KNN algorithms organize the data to make search faster since they can reduce the time per query to $O(log\ m)$ or sublinear, depending on data structure and dimension. Based on this, we considered tree-based KNN because it makes nearest-neighbor search feasible and efficient for large or complex datasets.

There are a myriad of tree-based KNN algorithms in the literature such as k-dimensional tree (KDTree), ball tree (BTree), cover tree (CTree), metric tree (MTree), vantage-point tree (VPTree), etc [Yianilos 1993]. Most of these algorithms are based on the assumption that data lives in Euclidean space and each point is a vector of real numbers. In KDTree, for instance, splits are axis-aligned. Moreover, BTree assumes that data

**Algorithm 2** BK-Tree Construction for NCD-based String Data

**Require:** Training dataset $\mathcal{D} = \{(\mathbf{s}_1, \mathbf{c}_1), \ldots, (\mathbf{s}_m, \mathbf{c}_m)\}$.
**Require:** Normalized Compression Distance (NCD) function $d(\cdot, \cdot)$.
**Ensure:** Root of the constructed BK-Tree ($\mathcal{T}_{\text{root}}$)
Initialize BK-Tree root: $\mathcal{T}_{\text{root}} \leftarrow$ null
**for** each $(\mathbf{s}_i, \mathbf{c}_i)$ in $\mathcal{D}$ **do**
    **if** $\mathcal{T}_{\text{root}} =$ null **then**
        $\mathcal{T}_{\text{root}} \leftarrow$ create_node$(\mathbf{s}_i, \mathbf{c}_i)$
    **else**
        Call InsertNode$(\mathcal{T}_{\text{root}}, \mathbf{s}_i, \mathbf{c}_i, d)$
**return** $\mathcal{T}_{\text{root}}$
**procedure** INSERTNODE(node, $\mathbf{s}$, $\mathbf{c}$, $d$)
    $d \leftarrow d$(node.string, $\mathbf{s}$) ▷ Calculate NCD from current node to instance
    **if** $d$ is not in node.children_distances **then**
        node.children_distances$[d] \leftarrow$ create_node$(\mathbf{s}, \mathbf{c})$ ▷ Create a new child at this distance
    **else**
        Call InsertNode(node.children_distances$[d], \mathbf{s}, \mathbf{c}, d)$ ▷ Recurse into existing child

can be embedded in a Euclidean metric space to compute centroid and radius. Therefore, these methods are unsuitable for enhancing [Jiang et al. 2023] because string metric distances and NCD are non-Euclidean. Moreover, triangle inequality may still hold (Levenshtein does, NCD may not), but no axes and no projection are guaranteed, disabling split along numeric axes.

In order to reduce the computational complexity of the framework established in [Jiang et al. 2023], we developed a compression-based BKTree based on [Burkhard and Keller 1973], which represents a significant optimization over the brute-force approach. The core idea remains the same: find the $k$ nearest neighbors and take the predicted class based on a majority vote. The change is in how those $k$ nearest neighbors are found. Instead of iterating through every single training string, BKTree provided a highly efficient way to prune the search space. Differently from the naive brute-force approach, BKTree requires a "training" step, i.e., before any query string arrives, the training dataset $\mathcal{D}$ must be organized into a data structure called 'BK-Tree'.

The primary advantage of BKTrees lies in its search efficiency, especially for large training datasets. We designed our method based on this data structure because, during a query, a BKTree applies the distance metric to intelligently prune vast portions of the search tree. If a branch of the tree cannot possibly contain a neighbor closer than the current $k$-th best neighbor found so far, that entire branch is skipped, avoiding unnecessary distance calculations. It implies on sublinear complexity, i.e., for typical real-world data, the search time becomes much closer to $O(m)$. Nevertheless, despite of this advantage in terms of query speed, BKTrees demands a preprocessing step that can be computationally intensive, as it involves many distance calculations to establish the tree structure. For a training dataset of $m$ points, the construction can take $O(m^2)$ distance calculations in the worst case, though often much less in practice ($O(mlog\ m)$ for efficient insertions.

**Algorithm 3** Parallelized NCD-based KNN search using BKTree

**Require:** BKTree $\mathcal{T}$ (with $\mathcal{T}_{\text{root}}$ as its root, constructed using Algorithm 2).
**Require:** Query string $\mathbf{q}$.
**Require:** Number of neighbors $k \in \mathbb{N}^+$.
**Require:** NCD function $d(\cdot, \cdot)$.
**Require:** Maximum number of parallel threads/tasks $\tau$.
**Ensure:** Predicted class for $\mathbf{q}$.

Initialize a shared, thread-safe min-priority queue $\mathcal{PQ}_{shared}$ of size $k$.
Initialize a shared, atomically updateable variable $\delta_{shared} \leftarrow \infty$.
Start a pool of $\tau$ worker threads.
Submit initial task to a task queue: TaskQueue.add($(\mathcal{T}_{\text{root}}, \mathbf{q}, k, \mathcal{PQ}_{shared}, \delta_{shared}, d)$)
▷ Worker threads continuously pull from TaskQueue and execute SEARCHBKTREE
**while** TaskQueue is not empty OR any worker thread is active **do**
    Wait for all active tasks to complete.
Extract class labels from $\mathcal{PQ}_{shared}$: $\mathcal{C}_{N_K} \leftarrow \{\mathbf{c} \mid (\text{dist}, \mathbf{c}) \in \mathcal{PQ}_{shared}\}$
Predict the class of $\mathbf{q}$ as the mode of the neighbor classes: $\overline{\mathbf{c}} \leftarrow \text{Mode}(\mathcal{C}_{N_K})$
**return** $\overline{\mathbf{c}}$
**procedure** SEARCHBKTREE(node, $\mathbf{q}$, $k$, $\mathcal{PQ}_{shared}$, $\delta_{shared}$, $d$)
    **if** node $=$ null **then return**
    $d_{node\_query} \leftarrow d(\text{node.string}, \mathbf{q})$
    ACQUIRELOCK($\mathcal{PQ}_{shared}$)
    **if** size($\mathcal{PQ}_{shared}$) $< k$ **then**
        $\mathcal{PQ}_{shared}$.add($d_{node\_query}$, node.$\mathbf{c}$)
    **else**
        **if** $d_{node\_query} < \mathcal{PQ}_{shared}$.peek_max_dist() **then**
            $\mathcal{PQ}_{shared}$.remove_max()
            $\mathcal{PQ}_{shared}$.add($d_{node\_query}$, node.$\mathbf{c}$)
    $\delta_{shared}$.atomic_update($\min(\delta_{shared}, \mathcal{PQ}_{shared}$.peek_max_dist())$)$
    RELEASELOCK($\mathcal{PQ}_{shared}$)
    **for** each ($d_{edge}$, child_node) in node.children_distances **do**
        **if** $|d_{node\_query} - d_{edge}| \leq \delta_{shared}$ **then**
            SUBMITTASKTOQUEUE(TaskQueue, (child_node, $\mathbf{q}$, $k$, $\mathcal{PQ}_{shared}$, $\delta_{shared}$, $d$))

Therefore, our method is intended to be used in scenarios where many query strings must be classified against the same training dataset, and the initial training cost is amortized over many faster queries, making the overall process much more efficient.

The training and querying steps of the proposed method are depicted in Algorithms 2 and 3, respectively. The tree construction itself is a sequential process due to the dependencies in node insertion. Each node insertion requires traversing the tree based on distance calculations, making it challenging to parallelize efficiently without complex concurrent data structures that manage tree modifications. Thus, this phase is typically executed as a preprocessing step (training).

However, once the tree is built, the search operation can significantly benefit from parallel execution. More specifically, the search for nearest neighbors within a BKTree

is performed with parallelization, as different branches of the tree can be explored concurrently. For effective parallel pruning and accurate nearest neighbor identification, two crucial data structures are made globally accessible and thread-safe: shared priority queue ($\mathcal{PQ}_{shared}$) and global best distance bound ($\delta_{shared}$). $\mathcal{PQ}_{shared}$ of size $k$ stores the currently known $k$ nearest neighbors discovered by any active thread. The operations (e.g., additions, removals, etc) must be protected by synchronization mechanisms (e.g., mutexes, locks, etc) to prevent race conditions. The variable $\delta_{shared}$ tracks the distance of the $k$-th best neighbor currently residing in the $\mathcal{PQ}_{shared}$. It is crucial for pruning and it is updated atomically (or under lock protection) whenever $\mathcal{PQ}_{shared}$ modified to ensure that all threads have access to the tightest possible pruning bound.

In Algorithm 3, instead of recursive calls in a single thread, the parallelized `SEARCHBKTREE` procedure dispatches independent exploratory tasks. A task queue is used to manage the pending branches to be explored. An initial task for the root of the BK-Tree is submitted. A pool of worker threads continuously pulls tasks from this queue and executes the `SEARCHBKTREE` procedure for that specific node and its potential subtrees. Each worker thread, upon processing a node, computes NCD for query string. Before updating $\mathcal{PQ}_{shared}$ or $\delta_{shared}$, the thread must acquire a lock to ensure exclusive access, thereby preventing race conditions and maintaining data integrity. Once the update is complete, the lock is released.

Essentially, when deciding whether to explore a child branch, each thread compares the potential range of distances from that child subtree against the globally updated $\delta_{shared}$. Since $\delta_{shared}$ is continuously updated by all active threads, the pruning decision benefits from the collective discoveries, allowing for more aggressive and efficient elimination of irrelevant branches across the entire parallel search. I.e., instead of making direct recursive calls, children that pass the pruning test are submitted as new tasks to the shared task queue, allowing another idle worker thread to continue the search concurrently. Finally, the parallel search concludes when the task queue is empty and all worker threads have completed their assigned tasks, which implies in all reachable nodes have been processed or pruned.

The combination a BKTree with parallelization, we distribute the computational load across multiple processing cores, which reduces the total time required to find the KNNs. Our method makes KNN queries much more scalable, allowing efficient classification of strings against large training sets that would be prohibitive for the original naive brute-force approach proposed by [Jiang et al. 2023].

## 4. Experimental Setup

The proposed method is compared with [Jiang et al. 2023] following the steps illustrated in Figure 1. Although both algorithms presented in Sections 2 and 3 are generic enough to be used on various datasets, we restrict our experiments to the "Low-Resource Fake News Detection Corpora in Filipino"[Cruz et al. 2020]. This dataset contains 3,206 expertly-labeled news samples, half of which are fake news and half of which are legit. The dataset is primarily in Filipino language, with the addition of some English words commonly used in Filipino vernacular. Using the HuggingFace library [Huggingface 2020], we implemented minimal text cleaning ensuring consistent experimental conditions.

Whereas [Jiang et al. 2023] considered only GZip as the compression algo-

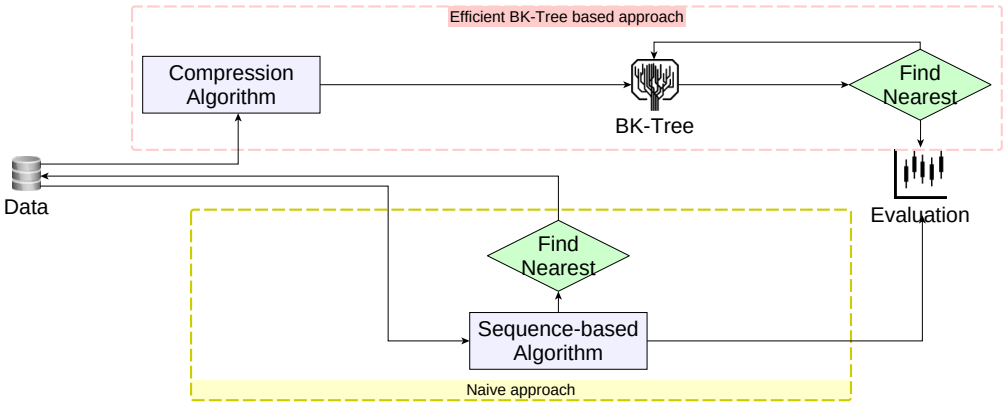


**Figure 1. Flowchart of proposed methodology validation.**

rithm [Gailly and Adler 1992], this paper extends that work by conducting extensive experiments using several other lossless data compression algorithms. More precisely, we conducted our experiments using statistical compressors, such as Brotli [Alakuijala et al. 2018], and Bz2 [Seward 1996], and also using dictionary-based methods like GZip [Gailly and Adler 1992], ZLib [Gailly and Adler 1995], Zstd [Collet 2016], LZAV [Avaneev 2023], and LZF [Lehmann 2008]. Furthermore, we also considered lightweight compressors – such as QuickLZ [Reinhold 2011], Shoco [Schramm 2015], Snappy [Dean et al. 2011], FSST [Boncz et al. 2019], and Smaz [Sanfilippo 2012] – that are designed to prioritize speed over compression rations.

In addition to comparing our proposed BKTree approach with [Jiang et al. 2023] naive method, we also implemented a sequence-based version, as described in Section 2, which does not incorporate string compression. More specifically, we implemented the brute-force KNN using Levenshtein [Levenshtein 1966], Damerau-Levenshtein [Damerau 1964], Jaro-Winkler [Jaro 1989] [Winkler 1990], and Simon-White [White 2000], function as supplementary baselines instead of NCD.

Using a fixed value of $k$=5 for the k-nearest neighbors, we evaluated the prediction performance of the compared algorithms using the following classification metrics: accuracy, precision, recall, and f1-score. The comparison of computational complexity between compared method were performed using the speedup. Specifically, we computed

- $Relative\ Speedup(p) = \frac{\text{Execution time of naive approach}}{\text{Execution time of proposed approach with p threads}}$,
- $Parallel\ Speedup(p) = \frac{\text{Execution time of proposed approach with 1 thread}}{\text{Execution time of proposed approach with p threads}}$,
- The theoretical speed-up, predicted by Amdahl's Law

$$S_{\text{theo}}(N) = \frac{1}{(1-\rho) + \frac{\rho}{N}} \quad \text{with} \quad \rho = \frac{\frac{1}{SU} - 1}{\frac{1}{N} - 1},$$

where $SU$ is the speed-up (for $N$ threads) and $\rho$ is the parallelization fraction.

Core code and compressors were implemented in C/C++ (GCC 13.2) with Python 3.10 orchestration (SciPy). Experiments ran on an AMD Ryzen Threadripper 1950X (16 cores). The source is available.[1]

[1] https://gitlab.com/lisa-unb/comtext/

## 5. Experimental Results

We first conducted our experiments by varying the compression algorithms and applying them to both [Jiang et al. 2023] and the proposed frameworks. For each simulation, we split the dataset into 80% for training and the remaining 20% for testing. We fixed all randomness parameters in the splits, meaning that all simulations, regardless of the framework, observed the same training and testing data. We did this to ensure a fairer comparison. The results from these experiments are reported in Table 1. From this table, the prediction results achieved by the proposed method is referred as "BKTree" (represented by symbol $\mathfrak{B}$). On the other hand, the method presented by [Jiang et al. 2023] is designated as "Naive" (symbolized by $\mathfrak{N}$).

From Table 1, it noticeable that BKTree–based method performs similarly to the naive implementation, yielding practically identical results and proving statistically equivalent in nearly all cases. In fact, for most compressors, the BKTree–based method achieved equal or slightly higher scores than the naive implementation, demonstrating that the optimized search strategy delivers performance gains without sacrificing classification accuracy. Furthermore, since the proposed framework is implemented as a parallel algorithm, the columns denoted by symbols $\mathcal{T}_{opt}$ and $\mathcal{S}_{max}$ indicate which parallelization-related hyperparameters were used to achieve those f1-score, accuracy, etc. More specifically, $\mathcal{T}_{opt}$ indicates the number of threads that achieved the best relative speed-up, and $\mathcal{S}_{max}$ reveals the relative speed-up achieved with that number of threads.

**Table 1. Prediction performance metrics obtained using the proposed BKTree-based method compared with the naive KNN approach when the maximum speedup was achieved. $\mathcal{S}_{max}$ represents the maximum relative speedup reached, and $\mathcal{T}_{opt}$ is the number of threads that yielded this optimal speedup. The symbols $\mathfrak{N}$ and $\mathfrak{B}$ represent the naive (baseline) and BKTree (proposed), respectively.**

| C | F1-Score | | Accuracy | | Precision | | Recall | | $\mathcal{T}_{opt}$ | $\mathcal{S}_{max}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\mathfrak{N}$ | $\mathfrak{B}$ | $\mathfrak{N}$ | $\mathfrak{B}$ | $\mathfrak{N}$ | $\mathfrak{B}$ | $\mathfrak{N}$ | $\mathfrak{B}$ | | |
| Brotli | 0.90 | **0.92** | 0.90 | **0.92** | 0.91 | **0.92** | 0.90 | **0.92** | 12 | 19.74 |
| FSST | **0.89** | 0.88 | **0.89** | 0.88 | **0.89** | 0.88 | **0.89** | 0.88 | 24 | 24.73 |
| LZ4 | **0.92** | **0.92** | **0.92** | **0.92** | 0.92 | **0.93** | **0.92** | **0.92** | 8 | 6.8 |
| LZAV | 0.88 | **0.89** | 0.88 | **0.89** | 0.89 | **0.90** | 0.88 | **0.89** | 8 | 6.36 |
| LZF | 0.90 | **0.91** | 0.90 | **0.91** | 0.90 | **0.91** | 0.90 | **0.91** | 16 | 8.03 |
| Quicklz | 0.81 | **0.92** | 0.81 | **0.92** | 0.83 | **0.92** | 0.81 | **0.92** | 12 | 9.93 |
| Shoco | **0.56** | 0.33 | **0.61** | 0.50 | **0.68** | 0.25 | **0.61** | 0.5 | 12 | 1.31 |
| Smaz | 0.43 | **0.54** | 0.52 | **0.55** | **0.57** | 0.56 | 0.52 | **0.55** | 8 | 1.89 |
| Snappy | 0.49 | **0.91** | 0.58 | **0.91** | 0.75 | **0.91** | 0.58 | **0.91** | 16 | 11.38 |
| ZLib | **0.94** | **0.94** | **0.94** | **0.94** | **0.94** | **0.94** | **0.94** | **0.94** | 12 | 11.83 |
| ZStd | **0.95** | 0.94 | **0.95** | 0.94 | **0.95** | 0.94 | **0.95** | 0.94 | 24 | 6.1 |

The relationship between performance and efficiency is illustrated in Figure 2. This figure presents a scatter plot comparing the maximum F1-score against the average execution time for each model, where naive implementations appears in red, BKTree-based approaches in blue, and compressionless sequence-based methods in green. An

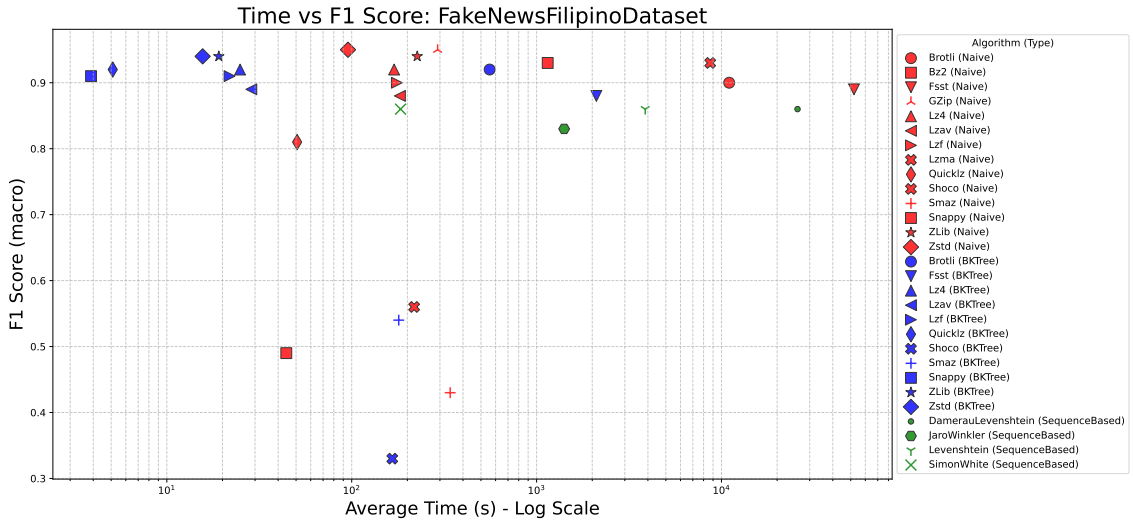


**Figure 2. Scatter plot of maximum F1-score versus average execution time (log scale) for all evaluated compressors. Model types are distinguished by color: Naive (red), BKTree (blue), Sequence-based (green).**

optimal model would be in the upper left quadrant, indicating minimal execution time and maximum classification accuracy. BKTree-based models predominantly occupy this region, showcasing superior efficiency in both speed and accuracy. In contrast, naive implementations, despite competitive accuracy, exhibit longer execution times, positioning them to the right regions of the plot. Compressionless sequence-based models show similar temporal characteristics to their naive counterparts.

The hyperparameters $\mathcal{T}_{opt}$ and $\mathcal{S}_{max}$ depicted in Table 1 correspond to the maximum points on the oranges graphs displayed in Figure 3. This figure shows multiple graphs depicting the number of threads versus speedup curve, with each graph corresponding to the results obtained by a different compressor. The blue curves represent the parallel speedup, indicating how many times faster the parallel BKTree is compared to the serial BKTree when utilizing a certain number of threads. The orange curves correspond to the relative speedup. As stated in Section 4, the relative speedup indicate how many times the parallel BKTree-based is faster than the naive approach with same compressor.

The graphs in Figure 3 indicate that while parallelization offers gains in resource utilization, its scalability quickly saturates, i.e., the speedup stops increasing even with more threads. Nevertheless, when comparing the curves in each graph, we can observe that the orange curve is almost always asymptotically above the blue one. This indicates that the proposed method's main advantage lies not in parallelization, but in its BKTree-based approach, which has a significantly lower computational complexity than the naive alternative. For instance, FSST achieves a 25$\times$ relative speed-up against the naive baseline and nearly 20$\times$ parallel speed-up over single-thread execution, showing that additional threads significantly reduce runtime. Other compressors also benefit from multithreading, with even less scalable algorithms exhibiting several-fold throughput improvements. Together, these results highlight the effectiveness of the proposed approach.

## 6. Conclusions

In this paper, we proposed a BKTree-based approach for compression-based string classification. Our method achieves accuracy comparable to [Jiang et al. 2023], which demonstrated performance on par with neural network classifiers on in-distribution datasets and superior to both pretrained and non-pretrained models on out-of-distribution datasets. While the prediction metrics are comparable to state-of-the-art methods, the proposed method's primary advantage lies in its computational efficiency. The BKTree integration dramatically reduces comparison operations in nearest neighbor searches, while evaluating various compression algorithms reveals key trade-offs between compression ratio and computational efficiency. Experimental results show substantial speedups over the [Jiang et al. 2023] brute-force baseline. Snappy and QuickLZ offer exceptional time efficiency, whereas ZLib and Zstd provide high classification accuracy at superior speeds. Future work will expand evaluation to diverse textual domains. Exploring alternative parallelization strategies for the BKTree and integrating lightweight learning techniques are promising directions to enhance the applicability of compression-based methods.

## References


Alakuijala, J., Farruggia, A., Ferragina, P., Kliuchnikov, E., Obryk, R., Szabadka, Z., and Vandevenne, L. (2018). Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems*, 37:1–30.

Avaneev, D. (2023). LZAV: Fast in-memory lz77-based data compressor (header-only c/c++). GitHub repository. Available at `https://github.com/avaneev/lzav`, with performance around 480 MB/s compression and 2800 MB/s decompression :contentReference[oaicite:1]index=1.

Boncz, P., Neumann, T., and Leis, V. (2019). FSST: Fast static symbol table string compression. CWI / research publication and GitHub. Lightweight random-access string compression; `https://github.com/cwida/fsst`.

Burkhard, W. A. and Keller, R. M. (1973). Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236.

Collet, Y. (2016). Zstandard (zstd): A fast lossless compression algorithm. Reference C implementation and specification. First released August 31, 2016; format standardized in IETF RFC 8878 (February 2021); official site: `https://facebook.github.io/zstd/`.

Cruz, J. C. B., Tan, J. A., and Cheng, C. (2020). Localization of fake news detection via multitask transfer learning. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 2596–2604.

Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176.

Dean, J., Ghemawat, S., and Gunderson, S. H. (2011). Snappy: A fast compressor/decompressor. Software library. Open-sourced by Google; C++ implementation with 250MB/s compression, 500MB/s decompression; `https://google.github.io/snappy/`.

Frank, E., Chui, C., and Witten, I. H. (2000). Text categorization using compression models. In *Proceedings of the Conference on Data Compression*, page 555.

Gailly, J. and Adler, M. (1992). gzip: Gnu file compression utility. Software and format specification. Initial release 31 October 1992; official site: https://www.gnu.org/software/gzip/.

Gailly, J. and Adler, M. (1995). zlib: A lossless data compression library. Software library and documentation. First released May 1, 1995; official site: https://zlib.net/.

Huggingface (2020). Fake news filipino (jcblaise/fake_news_filipino). https://huggingface.co/datasets/jcblaise/fake_news_filipino. Accessed: 2025-05-20.

Jaro, M. A. (1989). Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. In *JSM Proceedings, Social Statistics Section*.

Jiang, Z., Yang, M. Y. R., Tsirlin, M., Tang, R., Dai, Y., and Lin, J. (2023). "low-resource" text classification: A parameter-free classification method with compressors. In Rogers, A., Boyd-Graber, J. L., and Okazaki, N., editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 6810–6828. Association for Computational Linguistics.

Lehmann, M. A. (2008). LibLZF: A very small and fast lz77-based compression library. Project homepage. Last updated August 25, 2008; BSD-style license; official site: https://oldhome.schmorp.de/marc/liblzf.html.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710.

Reinhold, L. M. (2011). QuickLZ: A very fast lz compression library. Official website and source code. Described as the world's fastest compression library ( 308 MB/s); original C version v1.5.0 available at http://www.quicklz.com/, and ports in Go and Rust exist :contentReference[oaicite:1]index=1.

Sanfilippo, S. (2012). Smaz: Small string compression library. GitHub repository. Compresses very short strings (e.g. the $\rightarrow$ 1 byte); BSD-3; https://github.com/antirez/smaz.

Schramm, C. (2015). shoco: A fast compressor for short strings. GitHub repository. C library optimized for very short strings; MIT license; https://github.com/Ed-von-Schleck/shoco.

Seward, J. (1996). bzip2: A block-sorting file compressor. Source code and documentation. Released July 1996; official site: https://www.bzip.org/.

Teahan, W. J. and Harper, D. J. (2003). Using compression-based language models for text categorization. *Language modeling for information retrieval*, pages 141–165.

White, S. (2000). How to strike a match: A simple algorithm for string similarity. http://www.catalysoft.com/articles/StrikeAMatch.html. Accessed 2025-05-20.

Winkler, W. E. (1990). String comparator metrics and enhanced decision rules in the fellegi–sunter model of record linkage. Technical report, U.S. Bureau of Census.

Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *Soda*, volume 93, pages 311–21.

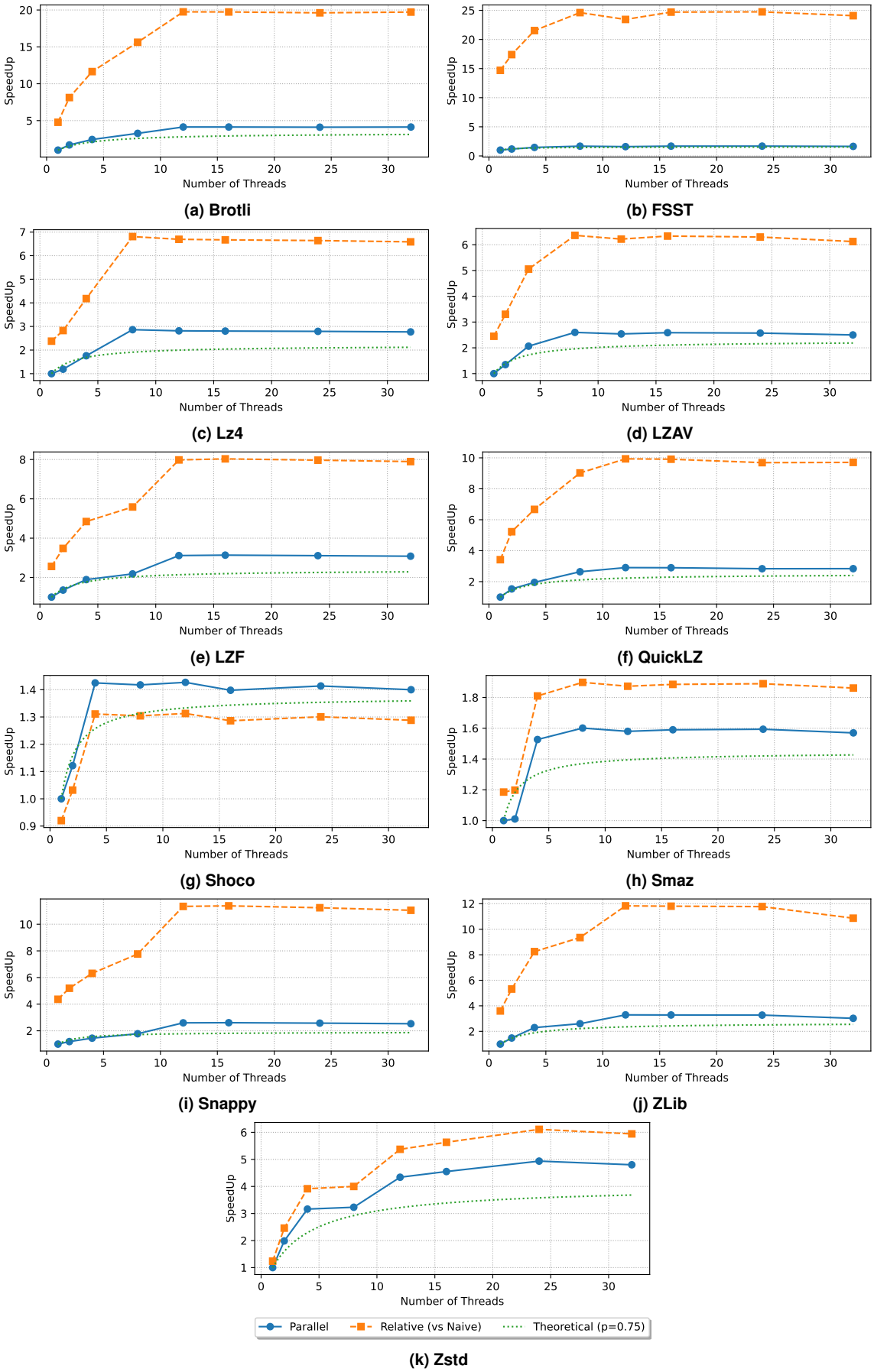


**Figure 3. Per-compressor relative, parallel, and theoretical speedup curves.**